ALGORITHMS and THEORY of COMPUTATION HANDBOOK

Edited by

MIKHAIL J. ATALLAH

ALGORITHMS and THEORY of COMPUTATION HANDBOOK

Edited by
MIKHAIL J. ATALLAH
Purdue University

CRC Press

Boca Raton London New York Washington, D.C.

Library of Congress Cataloging-in-Publication Data

Algorithms and theory of computation handbook/edited by Mikhail Atallah.
p. cm.
Includes bibliographical references and index.
ISBN 0-8493-2649-4 (alk. paper)
1. Computer algorithms. 2. Computer science. 3. Computational complexity. I. Atallah, Mikhail.
QA76.9.A43 A43 1998 98-38016
511.3—dc21 CIP

Visit the CRC Press Web site at www.crcpress.com

International Standard Book Number 0-8493-2649-4
Library of Congress Card Number 98-38016
Printed in the United States of America 2 3 4 5 6 7 8 9 0
Printed on acid-free paper

Preface

The purpose of *Algorithms and Theory of Computation Handbook* is to be a comprehensive treatment of the subject for computer scientists, engineers, and other professionals in related scientific and engineering disciplines. Its focus is to provide a compendium of fundamental topics and techniques for professionals, including practicing engineers, students, and researchers. The handbook is organized around the main subject areas of the discipline, and also contains chapters from applications areas that illustrate how the fundamental concepts and techniques come together to provide elegant solutions to important practical problems.

The contents of each chapter were chosen so that the computer professional or engineer has a high probability of finding significant information on a topic of interest. While the reader may not find in a chapter all the specialized topics, nor will the coverage of each topic be exhaustive, the reader should be able to obtain sufficient information for initial inquiries and a number of references to the current in-depth literature. Each chapter contains a section on "Research Issues and Summary" where the reader is given a summary of research issues in the subject matter of the chapter, as well as a brief summary of the chapter. Each chapter also contains a section called "Defining Terms" that provides a list of terms and definitions that might be useful to the reader. The last section of each chapter is called "Further Information" and directs the reader to additional sources of information in the chapter's subject area; these are the sources that contain more detail than the chapter can possibly provide. As appropriate, they include information on societies, seminars, conferences, databases, journals, etc.

It is a pleasure to extend my thanks to the people and organizations who made this handbook possible. My sincere thanks go to the chapter authors; it has been an honor and a privilege to work with such a dedicated and talented group. Purdue University and the universities and research laboratories with which the authors are affiliated deserve credit for providing the computing facilities and intellectual environment for this project. It is also a pleasure to acknowledge the support of CRC Press and its people: Bob Stern, Jerry Papke, Nora Konopka, Jo Gilmore, Suzanne Lassandro, Susan Fox, and Dr. Clovis L. Tondo. Special thanks are due to Bob Stern for suggesting to me this project and continuously supporting it thereafter. Finally, my wife Karen and my children Christina and Nadia deserve credit for their generous patience during the many weekends when I was in my office, immersed in this project.

Contributors

Eric Allender
Rutgers University,
New Brunswick, New Jersey

Alberto Apostolico
Purdue University,
West Lafayette, Indiana,
and Università di Padova,
Padova, Italy

Ricardo Baeza-Yates
Universidad de Chile,
Santiago, Chile

Guy E. Blelloch
Carnegie Mellon University,
Pittsburgh, Pennsylvania

Stefan Brands
Brands Technologies,
Utrecht, The Netherlands

Bryan Cantrill
Brown University,
Providence, Rhode Island

Vijay Chandru
Indian Institute of Science,
Bangalore, India

Chris Charnes
University of Wollongong,
Wollongong, Australia

Maxime Crochemore
Université de Marne-la-Vallée,
Noisy le Grand, France

Yvo Desmedt
University of Wisconsin –
Milwaukee,
Milwaukee, Wisconsin

Angel Díaz
IBM T.J. Watson Research Center,
Yorktown Heights, New York

Peter Eades
The University of Newcastle,
New South Wales, Australia

Ioannis Z. Emiris
INRIA Sophia-Antipolis,
Sophia-Antipolis, France

David Eppstein
University of California,
Irvine, California

Vladimir Estivill-Castro
The University of Newcastle
Callaghan, Australia

Eli Gafni
U.C.L.A.,
Los Angeles, California

Zvi Galil
Columbia University,
New York, New York

Sally A. Goldman
Washington University,
St. Louis, Missouri

Raymond Greenlaw
Armstrong Atlantic
State University ,
Savannah, Georgia

Concettina Guerra
Purdue University
West Lafayette, Indiana,
and Università di Padova,
Padova, Italy

Dan Halperin
Tel Aviv University,
Tel Aviv, Israel

Christophe Hancart
Université de Rouen,
Mont Saint Aignan, France

H. James Hoover
University of Alberta,
Edmonton, Alberta,
Canada

Giuseppe F. Italiano
Universitá "Ca' Foscari" di Venezia,
via Torino, Venezia Mestre, Italy

Tao Jiang
McMaster University,
Hamilton, Ontario, Canada

Erich Kaltofen
North Carolina State University,
Raleigh, North Carolina

David Karger
Massachusetts Institute of
Technology,
Cambridge, Massachusetts

Lydia Kavraki
Stanford University,
Stanford, California

Rick Kazman
Carnegie Mellon University,
Pittsburgh, Pennsylvania

Samir Khuller
University of Maryland,
College Park, Maryland

Andrew Klapper
University of Kentucky,
Lexington, Kentucky

Philip N. Klein
Brown University,
Providence, Rhode Island

Richard E. Korf
University of California,
Los Angeles, California

Andrea S. LaPaugh
Princeton University,
Princeton, New Jersey

Jean-Claude Latombe
Stanford University,
Stanford, California

Thierry Lecroq
Université de Rouen,
Mount Saint Aignan, France

D.T. Lee
Northwestern University,
Evanston, Illinois

Ming Li
University of Waterloo,
Waterloo, Ontario,
Canada

Michael C. Loui
University of Illinois at
Urbana-Champaign,
Urbana, Illinois

Bruce M. Maggs
Carnegie Mellon University,
Pittsburgh, Pennsylvania

Russ Miller
State University of New York at
Buffalo,
Buffalo, New York

Rajeev Motwani
Stanford University,
Stanford, California

Petra Mutzel
Max-Planck-Institute fur
Informatik,
Saarbrucken, Germany

Victor Y. Pan
City University of New York,
Bronx, New York

Steven Phillips
AT&T Bell Laboratories,
Murray Hill, New Jersey

Josef Pieprzyk
University of Wollongong,
Wollongong, Australia

Patricio V. Poblete
Universidad de Chile,
Santiago, Chile

Balaji Raghavachari
University of Texas at Dallas,
Richardson, Texas

Prabhakar Raghavan
IBM Almaden Research Center,
San Jose, California

Rajeev Raman
King's College, London,
Strand, London,
United Kingdom

M.R. Rao
Indian Institute of Management,
Bangalore, India

Bala Ravikumar
University of Rhode Island,
Kingston, Rhode Island

Kenneth W. Regan
State University of New York at
Buffalo,
Buffalo, New York

Edward M. Reingold
University of Illinois at
Urbana-Champaign,
Urbana, Illinois

Rei Safavi-Naini
University of Wollongong,
Wollongong, Australia

Hanan Samet
University of Maryland,
College Park, Maryland

Jennifer Seberry
University of Wollongong,
Wollongong, Australia

Cliff Stein
Dartmouth College,
Hanover, New Hampshire

Quentin F. Stout
University of Michigan,
Ann Arbor, Michigan

Wojciech Szpankowski
Purdue University,
West Lafayette, Indiana

Roberto Tamassia
Brown University,
Providence, Rhode Island

Stephen A. Vavasis
Cornell University,
Ithaca, New York

Samuel S. Wagstaff, Jr.
Purdue University,
West Lafayette, Indiana

Joel Wein
Polytechnic University,
Brooklyn, New York

Jeffery Westbrook
AT&T Bell Laboratories,
Murray Hill, New Jersey

Neal E. Young
Dartmouth College,
Hanover, New Hampshire

Albert Y. Zomaya
The University of Western
Australia,
Nedlands, Perth, Australia

Contents

1

Algorithm Design and Analysis Techniques[1]

Edward M. Reingold
University of Illinois at Urbana-Champaign

We outline the basic methods of algorithm design and analysis that have found application in the manipulation of discrete objects such as lists, arrays, sets, graphs, and geometric objects such as points, lines, and polygons. We begin by discussing **recurrence relations** and their use in the analysis of algorithms. Then we discuss some specific examples in algorithm analysis, **sorting** and **priority queues.** In the next three sections, we explore three important techniques of algorithm design—**divide-and-conquer, dynamic programming,** and **greedy heuristics.** Finally, we examine establishing lower bounds on the cost of any algorithm for a problem.

1.1 Analyzing Algorithms

It is convenient to classify algorithms based on the relative amount of time they require: how fast does the time required grow as the size of the problem increases? For example, in the case of arrays, the "size of the problem" is ordinarily the number of elements in the array. If the size of the problem is measured by a variable n, we can express the time required as a function of n, $T(n)$. When this function $T(n)$ grows rapidly, the algorithm becomes unusable for large n; conversely, when $T(n)$ grows slowly, the algorithm remains useful even when n becomes large.

[1]Supported in part by the National Science Foundation, grant numbers CCR-93-20577 and CCR-95-30297. The comments of Tanya Berger-Wolf, Ken Urban, and an anonymous referee are gratefully acknowledged.

We say an algorithm is $\Theta(n^2)$ if the time it takes quadruples (asymptotically) when n doubles; an algorithm is $\Theta(n)$ if the time it takes doubles when n doubles; an algorithm is $\Theta(\log n)$ if the time it takes increases by a constant, independent of n, when n doubles; an algorithm is $\Theta(1)$ if its time does not increase at all when n increases. In general, an algorithm is $\Theta(T(n))$ if the time it requires on problems of size n grows proportionally to $T(n)$ as n increases. Table 1.1 summarizes the common growth rates encountered in the analysis of algorithms.

TABLE 1.1 Common Growth Rates of Times of Algorithms

Rate of Growth	Comment	Examples
$\Theta(1)$	Time required is constant, independent of problem size	Expected time for hash searching
$\Theta(\log \log n)$	Very slow growth of time required	Expected time of interpolation search of n elements
$\Theta(\log n)$	Logarithmic growth of time required—doubling the problem size increases the time by only a constant amount	Computing x^n; binary search of an array of n elements
$\Theta(n)$	Time grows linearly with problem size—doubling the problem size doubles the time required	Adding/subtracting n-digit numbers; linear search of an n-element array
$\Theta(n \log n)$	Time grows worse than linearly, but not much worse—doubling the problem size somewhat more than doubles the time required	Merge sort or heapsort of n elements; lower bound on comparison-based sorting of n elements
$\Theta(n^2)$	Time grows quadratically—doubling the problem size quadruples the time required	Simple-minded sorting algorithms
$\Theta(n^3)$	Time grows cubically—doubling the problem size results in an 8-fold increase in the time required	Ordinary matrix multiplication
$\Theta(c^n)$	Time grows exponentially—increasing the problem size by 1 results in a c-fold increase in the time required; doubling the problem size *squares* the time required	Some traveling salesman problem algorithms based on exhaustive search

The analysis of an algorithm is often accomplished by finding and solving a recurrence relation that describes the time required by the algorithm. The most commonly occurring families of recurrences in the analysis of algorithms are linear recurrences and divide-and-conquer recurrences. In the following subsection we describe the "method of operators" for solving linear recurrences; in the next subsection we describe how to obtain an asymptotic solution to divide-and-conquer recurrences by transforming such a recurrence into a linear recurrence.

Linear Recurrences

A *linear recurrence with constant coefficients* has the form

$$c_0 a_n + c_1 a_{n-1} + c_2 a_{n-2} + \cdots + c_k a_{n-k} = f(n) \,, \tag{1.1}$$

for some constant k, where each c_i is constant. To solve such a recurrence for a broad class of functions f (that is, to express a_n in closed form as a function of n) by the *method of operators*, we consider two basic operators on sequences: $\mathcal{S}$, which shifts the sequence left,

$$\mathcal{S}\,\langle a_0, a_1, a_2, \ldots\rangle = \langle a_1, a_2, a_3, \ldots\rangle \;,$$

and C, which, for any constant C, multiplies each term of the sequence by C:

$$C\,\langle a_0, a_1, a_2, \ldots\rangle = \langle Ca_0, Ca_1, Ca_2, \ldots\rangle \;.$$

These basic operators on sequences allow us to construct more complicated operators by sums and products of operators. The sum $(A + B)$ of operators A and B is defined by

$$(A + B)\,\langle a_0, a_1, a_2, \ldots\rangle = A\,\langle a_0, a_1, a_2, \ldots\rangle + B\,\langle a_0, a_1, a_2, \ldots\rangle \;.$$

The product AB is the composition of the two operators:

$$(AB)\,\langle a_0, a_1, a_2, \ldots\rangle = A\,(B\,\langle a_0, a_1, a_2, \ldots\rangle)\ .$$

Thus, for example,

$$\left(\mathcal{S}^2 - 4\right)\langle a_0, a_1, a_2, \ldots\rangle = \langle a_2 - 4a_0, a_3 - 4a_1, a_4 - 4a_2, \ldots\rangle\ ,$$

which we write more briefly as

$$\left(\mathcal{S}^2 - 4\right)\langle a_i\rangle = \left\langle a_{i+2} - 4a_i\right\rangle\ .$$

With the operator notation, we can rewrite Eq. (1.1) as

$$P(\mathcal{S})\,\langle a_i\rangle = \langle f(i)\rangle\ ,$$

where

$$P(\mathcal{S}) = c_0\mathcal{S}^k + c_1\mathcal{S}^{k-1} + c_2\mathcal{S}^{k-2} + \cdots + c_k$$

is a polynomial in $\mathcal{S}$.

Given a sequence $\langle a_i\rangle$, we say that the operator A *annihilates* $\langle a_i\rangle$ if $A\langle a_i\rangle = \langle 0\rangle$. For example, $\mathcal{S}^2 - 4$ annihilates any sequence of the form $\langle u2^i + v(-2)^i\rangle$, with constants u and v. Here are two important facts about annihilators:

FACT 1.1 *The sum and product of operators are associative, commutative, and product distributes over sum. In other words, for operators A, B, and C,*

$$\begin{array}{lcllcl} (A+B)+C & = & A+(B+C) & (AB)C & = & A(BC)\ , \\ A+B & = & B+A & AB & = & BA\ , \end{array}$$

and

$$A(B+C) = AB + AC\ .$$

As a consequence, if A annihilates $\langle a_i\rangle$, then A annihilates $B\langle a_i\rangle$ for any operator B. This implies that the product *of two annihilators annihilates the* sum *of the sequences annihilated by the two operators—that is, if A annihilates $\langle a_i\rangle$ and B annihilates $\langle b_i\rangle$, then AB annihilates $\langle a_i + b_i\rangle$.*

FACT 1.2 *The operator $(\mathcal{S} - c)$, when applied to $\langle c^i \times p(i)\rangle$ with $p(i)$ a polynomial in i, results in a sequence $\langle c^i \times q(i)\rangle$ with $q(i)$ a polynomial of degree one less than $p(i)$. This implies that the operator $(\mathcal{S} - c)^{k+1}$ annihilates $\langle c^i \times$ (a polynomial in i of degree $k)\rangle$.*

These two facts mean that determining the annihilator of a sequence is tantamount to determining the sequence; moreover, it is straightforward to determine the annihilator from a recurrence relation. For example, consider the Fibonacci recurrence

$$\begin{array}{rcl} F_0 & = & 0 \\ F_1 & = & 1 \\ F_{i+2} & = & F_{i+1} + F_i\ . \end{array}$$

The last line of this definition can be rewritten as $F_{i+2} - F_{i+1} - F_i = 0$, which tells us that $\langle F_i\rangle$ is annihilated by the operator

$$\mathcal{S}^2 - \mathcal{S} - 1 = (\mathcal{S} - \phi)\,(\mathcal{S} + 1/\phi)\ ,$$

where $\phi = (1+\sqrt{5})/2$. Thus we conclude from Fact 1.1 that $\langle F_i \rangle = \langle a_i + b_i \rangle$ with $(\mathcal{S} - \phi)\langle a_i \rangle = \langle 0 \rangle$ and $(\mathcal{S} - 1/\phi)\langle b_i \rangle = \langle 0 \rangle$. Fact 1.2 now tells us that

$$F_i = u\phi^i + v(-\phi)^{-i} ,$$

for some constants u and v. We can now use the initial conditions $F_0 = 0$ and $F_1 = 1$ to determine u and v: These initial conditions mean that

$$\begin{aligned} u\phi^0 + v(-\phi)^{-0} &= 0 \\ u\phi^1 + v(-\phi)^{-1} &= 1 \end{aligned}$$

and these linear equations have the solution

$$u = v = 1/\sqrt{5} ,$$

and hence

$$F_i = \phi^i/\sqrt{5} + (-\phi)^{-i}/\sqrt{5} .$$

In the case of the similar recurrence,

$$\begin{aligned} G_0 &= 0 \\ G_1 &= 1 \\ G_{i+2} &= G_{i+1} + G_i + i , \end{aligned}$$

the last equation tells us that

$$\left(\mathcal{S}^2 - \mathcal{S} - 1\right)\langle G_i \rangle = \langle i \rangle ,$$

so the annihilator for $\langle G_i \rangle$ is $(\mathcal{S}^2 - \mathcal{S} - 1)(\mathcal{S} - 1)^2$, since $(\mathcal{S} - 1)^2$ annihilates $\langle i \rangle$ (a polynomial of degree 1 in i) and hence the solution is

$$G_i = u\phi^i + v(-\phi)^{-i} + \text{(a polynomial of degree 1 in i)} ,$$

that is,

$$G_i = u\phi^i + v(-\phi)^{-i} + wi + z .$$

Again, we use the initial conditions to determine the constants u, v, w, and z.

In general, then, to solve the recurrence (1.1), we factor the annihilator

$$P(\mathcal{S}) = c_0\mathcal{S}^k + c_1\mathcal{S}^{k-1} + c_2\mathcal{S}^{k-2} + \cdots + c_k ,$$

multiply it by the annihilator for $\langle f(i) \rangle$, write down the form of the solution from this product (which is the annihilator for the sequence $\langle a_i \rangle$), and then use the initial conditions for the recurrence to determine the coefficients in the solution.

Divide-and-Conquer Recurrences

The divide-and-conquer paradigm of algorithm construction that we discuss in Section 1.3 leads naturally to divide-and-conquer recurrences of the type

$$T(n) = g(n) + uT(n/v) ,$$

for constants u and v, $v > 1$, and sufficient initial values to define the sequence $\langle T(0), T(1), T(2), \ldots \rangle$. The growth rates of $T(n)$ for various values of u and v are given in Table 1.2. The growth rates in this table

TABLE 1.2 Rate of Growth of the Solution to the Recurrence $T(n) = g(n) + uT(n/v)$, the Divide-and-Conquer Recurrence Relations

$g(n)$	u, v	Growth Rate of $T(n)$
$\Theta(1)$	$u = 1$	$\Theta(\log n)$
	$u \neq 1$	$\Theta(n^{\log_v u})$
$\Theta(\log n)$	$u = 1$	$\Theta[(\log n)^2]$
	$u \neq 1$	$\Theta(n^{\log_v u})$
$\Theta(n)$	$u < v$	$\Theta(n)$
	$u = v$	$\Theta(n \log n)$
	$u > v$	$\Theta(n^{\log_v u})$
$\Theta(n^2)$	$u < v^2$	$\Theta(n^2)$
	$u = v^2$	$\Theta(n^2 \log n)$
	$u > v^2$	$\Theta(n^{\log_v u})$

Note: The variables u and v are Positive Constants, Independent of n, and $v > 1$.

are derived by transforming the divide-and-conquer recurrence into a linear recurrence for a subsequence of $\langle T(0), T(1), T(2), \ldots \rangle$.

To illustrate this method, we derive the penultimate line in Table 1.2. We want to solve

$$T(n) = n^2 + v^2 T(n/v) \ ,$$

so we want to find a subsequence of $\langle T(0), T(1), T(2), \ldots \rangle$ that will be easy to handle. Let $n_k = v^k$; then,

$$T(n_k) = n_k^2 + v^2 T(n_k/v) \ ,$$

or

$$T\left(v^k\right) = v^{2k} + v^2 T\left(v^{k-1}\right) \ .$$

Defining $t_k = T(v^k)$,

$$t_k = v^{2k} + v^2 t_{k-1} \ .$$

The annihilator for t_k is then $(\mathcal{S} - v^2)^2$, and thus

$$t_k = v^{2k}(ak + b) \ ,$$

for constants a and b. Since $n_k = v^k$, $k = \log_v n_k$, so we can express the solution for t_k in terms of $T(n)$,

$$T(n) \approx t_{\log_v n} = v^{2\log_v n}\left(a\log_v n + b\right) = an^2 \log_v n + bn^2 \ ,$$

or

$$T(n) = \Theta\left(n^2 \log n\right) \ .$$

1.2 Some Examples of the Analysis of Algorithms

In this section we introduce the basic ideas of algorithms analysis by looking at some practical problems of maintaining a collection of n objects and retrieving objects based on their relative size. For example, how can we determine the smallest of the elements? Or, more generally, how can we determine the kth largest of the elements? What is the running time of such algorithms in the worst case? Or, on the average, if all $n!$ permutations of the input are equally likely? What if the set of items is dynamic—that is, the set changes through insertions and deletions—how efficiently can we keep track of, say, the largest element?

Sorting

How do we rearrange an array of n values `x[1]`, `x[2]`, ..., `x[n]` so that they are in perfect order—that is, so that `x[1]` $\leq$ `x[2]` $\leq \ldots \leq$ `x[n]`? The simplest way to put the values in order is to mimic what we might do by hand: take item after item and insert each one into the proper place among those items already inserted:

```
 1    void insert (float x[], int i, float a) {
 2      // Insert a into x[1] ... x[i]
 3      // x[1] ... x[i-1] are sorted;  x[i] is unoccupied
 4      if (i == 1 || x[i-1] <= a)
 5        x[i] = a;
 6      else {
 7        x[i] = x[i-1];
 8        insert(x, i-1, a);
 9      }
10    }
11
12    void insertionSort (int n, float x[]) {
13      // Sort  x[1] ... x[n]
14      if (n > 1) {
15        insertionSort(n-1, x);
16        insert(x, n, x[n]);
17      }
18    }
```

To determine the time required in the worst case to sort n elements with `insertionSort`, we let t_n be the time to sort n elements and derive and solve a recurrence relation for t_n. We have

$$t_n = \begin{cases} \Theta(1) & \text{if } n = 1, \\ t_{n-1} + s_{n-1} + \Theta(1) & \text{otherwise}, \end{cases}$$

where s_m is the time required to insert an element in place among m elements using `insert`. The value of s_m is also given by a recurrence relation:

$$s_m = \begin{cases} \Theta(1) & \text{if } m = 1, \\ s_{m-1} + \Theta(1) & \text{otherwise}. \end{cases}$$

The annihilator for $\langle s_i \rangle$ is $(\mathcal{S}-1)^2$, so $s_m = \Theta(m)$. Thus the annihilator for $\langle t_i \rangle$ is $(\mathcal{S}-1)^3$, so $t_n = \Theta(n^2)$. The analysis of the average behavior is nearly identical; only the constants hidden in the Θ-notation change.

We can design better sorting methods using the divide-and-conquer idea of the next section. These algorithms avoid $\Theta(n^2)$ worst-case behavior, working in time $\Theta(n \log n)$. We can also achieve time $\Theta(n \log n)$ by using a clever way of viewing the array of elements to be sorted as a tree: consider `x[1]` as the root of the tree and, in general, `x[2*i]` is the root of the left subtree of `x[i]` and `x[2*i+1]` is the root of the right subtree of `x[i]`. If we further insist that parents be greater than or equal to children, we have a **heap**; Fig. 1.1 shows a small example.

A heap can be used for sorting by observing that the largest element is at the root, that is, `x[1]`; thus to put the largest element in place, we swap `x[1]` and `x[n]`. To continue, we must restore the heap property which may now be violated at the root. Such restoration is accomplished by swapping `x[1]` with its larger child, if that child is larger than `x[1]`, and the continuing to swap it downward until either it reaches the bottom or a spot where it is greater or equal to its children. Since the tree-cum-array has height $\Theta(\log n)$, this restoration process takes time $\Theta(\log n)$. Now, with the heap in `x[1]` to `x[n-1]` and

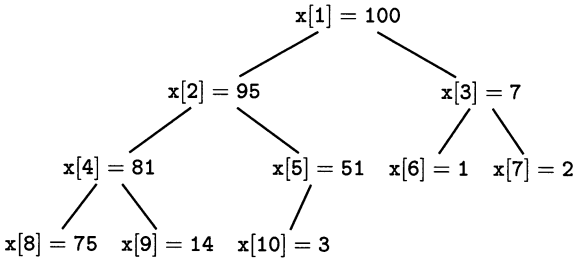


FIGURE 1.1 A heap—that is, an array, interpreted as a binary tree.

`x[n]` the largest value in the array, we can put the second largest element in place by swapping `x[1]` and `x[n-1]`; then we restore the heap property in `x[1]` to `x[n-2]` by propagating `x[1]` downward—this takes time $\Theta(\log(n-1))$. Continuing in this fashion, we find we can sort the entire array in time

$$\Theta\left(\log n + \log(n-1) + \cdots + \log 1\right) .$$

To evaluate this sum, we bound it from above and below, as follows. By ignoring the smaller half of the terms, we bound it from below:

$$\begin{aligned} \log n + \log(n-1) + \cdots + \log 1 &\geq \underbrace{\log \frac{n}{2} + \log \frac{n}{2} + \cdots + \log \frac{n}{2}}_{\frac{n}{2} \text{ times}} \\ &= \frac{n}{2} \log n \\ &= \Theta(n \log n) ; \end{aligned}$$

and by overestimating all of the terms we bound it from above:

$$\begin{aligned} \log n + \log(n-1) + \cdots + \log 1 &\leq \underbrace{\log n + \log n + \cdots + \log n}_{n \text{ times}} \\ &= n \log n \\ &= \Theta(n \log n) . \end{aligned}$$

The initial creation of the heap from an unordered array is done by applying the above restoration process successively to `x[n/2]`, `x[n/2-1]`, ..., `x[1]`, which takes time $\Theta(n)$.

Hence, we have the following $\Theta(n \log n)$ sorting algorithm:

```
1     void heapify (int n, float x[], int i) {
2       // Repair heap property below x[i] in x[1] ... x[n]
3       int largest = i;  // largest of x[i], x[2*i], x[2*i+1]
4       if (2*i <= n && x[2*i] > x[i])
5         largest = 2*i;
6       if (2*i+1 <= n && x[2*i+1] > x[largest])
7         largest = 2*i+1;
8       if (largest != i) {
9         // swap x[i] with larger child and repair heap below
10        float t = x[largest]; x[largest] = x[i]; x[i] = t;
11        heapify(n, x, largest);
12      }
13    }
```

```
14
15      void makeheap (int n, float x[]) {
16        // Make x[1] ... x[n] into a heap
17        for (int i=n/2; i>0; i--)
18          heapify(n, x, i);
19      }
20
21      void heapsort (int n, float x[]) {
22        // Sort  x[1] ... x[n]
23        float t;
24        makeheap(n, x);
25        for (int i=n; i>1; i--) {
26          // put x[1] in place and repair heap
27          t = x[1]; x[1] = x[i]; x[i] = t;
28          heapify(i-1, x, 1);
29        }
30      }
```

We will see in Section 1.6 that no sorting algorithm can be guaranteed always to use time less than $\Theta(n \log n)$. Thus, in a theoretical sense, heapsort is "asymptotically optimal" (but there are algorithms that perform better in practice).

Priority Queues

Aside from its application to sorting, the heap is an interesting data structure in its own right. In particular, heaps provide a simple way to implement a *priority queue*—a priority queue is an abstract data structure that keeps track of a dynamically changing set of values allowing the following operations:

`create`: Create an empty priority queue.

`insert`: Insert a new element into a priority queue.

`decrease`: Decrease the value of an element in a priority queue.

`minimum`: Report the smallest element in a priority queue.

`deleteMinimum`: Delete the smallest element in a priority queue.

`delete`: Delete an element in a priority queue.

`merge`: Merge two priority queues.

A heap can implement a priority queue by altering the heap property to insist that parents are less than or equal to their children, so that the smallest value in the heap is at the root, that is, in the first array position. Creation of an empty heap requires just the allocation of an array, an $\Theta(1)$ operation; we assume that once created, the array containing the heap can be extended arbitrarily at the right end. Inserting a new element means putting that element in the $(n+1)$st location and "bubbling it up" by swapping it with its parent until it reaches either the root or a parent with a smaller value. Since a heap has logarithmic height, insertion to a heap of n elements thus requires worst-case time $O(\log n)$. Decreasing a value in a heap requires only a similar $O(\log n)$ "bubbling up." The smallest element of such a heap is always at the root, so reporting it takes $\Theta(1)$ time. Deleting the minimum is done by swapping the first and last array positions, bubbling the new root value downward until it reaches its proper location, and truncating the array to eliminate the last position. Delete is handled by decreasing the value so that it is the least in the heap and then applying the `deleteMinimum` operation; this takes a total of $O(\log n)$ time.

The merge operation, unfortunately, is not so economically accomplished—there is little choice but to create a new heap out of the two heaps in a manner similar to the `makeheap` function in heap sort. If there are a total of n elements in the two heaps to be merged, this re-creation will require time $O(n)$.

There are better data structures than a heap for implementing priority queues, however. In particular, the *Fibonacci heap* provides an implementation of priority queues in which the delete and `deleteMinimum` operations take $O(\log n)$ time and the remaining operations take $\Theta(1)$ time, *provided we consider the time required for a sequence of priority queue operations, rather than the individual times of each operation.* That is, we must consider the cost of the individual operations *amortized over the sequence of operations*: Given a sequence of n priority queue operations, we will compute the total time $T(n)$ for all n operations. In doing this computation, however, we do not simply add the costs of the individual operations; rather, we subdivide the cost of each operation into two parts, the *immediate cost* of doing the operation and the *long-term savings* that result from doing the operation—the long-term savings represent costs *not* incurred by later operations as a result of the present operation. The immediate cost minus the long-term savings give the **amortized cost** of the operation.

It is easy to calculate the immediate cost (time required) of an operation, but how can we measure the long-term savings that result? We imagine that the data structure has associated with it a bank account; at any given moment the bank account must have a nonnegative balance. When we do an operation that will save future effort, we are making a deposit to the savings account and when, later on, we derive the benefits of that earlier operation we are making a withdrawal from the savings account. Let $\mathcal{B}(i)$ denote the balance in the account after the ith operation, $\mathcal{B}(0) = 0$. We define the amortized cost of the ith operation to be

$$\begin{aligned}\text{amortized cost of } i\text{th operation} &= (\text{immediate cost of } i\text{th operation}) \\ &\quad + (\text{change in bank account}) \\ &= (\text{immediate cost of } i\text{th operation}) + (\mathcal{B}(i) - \mathcal{B}(i-1)) .\end{aligned}$$

Since the bank account $\mathcal{B}$ can go up or down as a result of the ith operation, the amortized cost may be less than or more than the immediate cost. By summing the previous equation, we get

$$\begin{aligned}\sum_{i=1}^{n}(\text{amortized cost of } i\text{th operation}) &= \sum_{i=1}^{n}(\text{immediate cost of } i\text{th operation}) \\ &\quad + (\mathcal{B}(n) - \mathcal{B}(0)) \\ &= (\text{total cost of all } n \text{ operations}) + \mathcal{B}(n) \\ &\geq \text{total cost of all } n \text{ operations} \\ &= T(n) ,\end{aligned}$$

because $\mathcal{B}(i)$ is nonnegative. Thus defined, the sum of the amortized costs of the operations gives us an upper bound on the total time $T(n)$ for all n operations.

It is important to note that the function $\mathcal{B}(i)$ is not part of the data structure, but is just our way to measure how much time is used by the sequence of operations. As such, we can choose *any rules* for $\mathcal{B}$, provided $\mathcal{B}(0) = 0$ and $\mathcal{B}(i) \geq 0$ for $i \geq 1$. Then, the sum of the amortized costs defined by

$$\text{amortized cost of } i\text{th operation} = (\text{immediate cost of } i\text{th operation}) + (\mathcal{B}(i) - \mathcal{B}(i-1))$$

bounds the overall cost of the operation of the data structure.

Now, to apply this method to priority queues. A *Fibonacci heap* is a list of heap-ordered trees (not necessarily binary); since the trees are heap ordered, the minimum element must be one of the roots and we keep track of which root is the overall minimum. Some of the tree nodes are *marked*. We define

$$\begin{aligned}\mathcal{B}(i) &= (\text{number of trees after the } i\text{th operation}) \\ &\quad + K \times (\text{number of marked nodes after the } i\text{th operation}) ,\end{aligned}$$

where K is a constant that we will define precisely during the discussion below.

The clever rules by which nodes are marked and unmarked, and the intricate algorithms that manipulate the set of trees, are too complex to present here in their complete form, so we just briefly describe the simpler operations and show the calculation of their amortized costs:

`create`: To create an empty Fibonacci heap we create an empty list of heap-ordered trees. The immediate cost is $\Theta(1)$; since the numbers of trees and marked nodes are zero before and after this operation, $\mathcal{B}(i) - \mathcal{B}(i-1)$ is zero and the amortized time is $\Theta(1)$.

`insert`: To insert a new element into a Fibonacci heap we add a new one-element tree to the list of trees constituting the heap and update the record of what root is the overall minimum. The immediate cost is $\Theta(1)$. $\mathcal{B}(i) - \mathcal{B}(i-1)$ is also 1 since the number of trees has increased by 1, while the number of marked nodes is unchanged. The amortized time is thus $\Theta(1)$.

`decrease`: Decreasing an element in a Fibonacci heap is done by cutting the link to its parent, if any, adding the item as a root in the list of trees, and decreasing its value. Furthermore, the marked parent of a cut element is itself cut, and this process of cutting marked parents propagates upward in the tree. Cut nodes become unmarked, and the unmarked parent of a cut element becomes marked. The immediate cost of this operation is no more than kc, where c is the number of cut nodes and $k > 0$ is some constant. Now, letting $K = k + 1$, we see that if there were t trees and m marked elements before this operation, the value of $\mathcal{B}$ before the operation was $t + Km$. After the operation, the value of $\mathcal{B}$ is $(t+c) + K(m-c+2)$, so $\mathcal{B}(i) - \mathcal{B}(i-1) = (1-K)c + 2K$. The amortized time is thus no more than $kc + (1-K)c + 2K = \Theta(1)$ since K is constant.

`minimum`: Reporting the minimum element in a Fibonacci heap takes time $\Theta(1)$ and does not change the numbers of trees and marked nodes; the amortized time is thus $\Theta(1)$.

`deleteMinimum`: Deleting the minimum element in a Fibonacci heap is done by deleting that tree root, making its children roots in the list of trees. Then, the list of tree roots is "consolidated" in a complicated $O(\log n)$ operation that we do not describe. The result takes amortized time $O(\log n)$.

`delete`: Deleting an element in a Fibonacci heap is done by decreasing its value to $-\infty$ and then doing a `deleteMinimum`. The amortized cost is the sum of the amortized cost of the two operations, $O(\log n)$.

`merge`: Merging two Fibonacci heaps is done by concatenating their lists of trees and updating the record of which root is the minimum. The amortized time is thus $\Theta(1)$.

Notice that the amortized cost of each operation is $\Theta(1)$ except `deleteMinimum` and `delete`, both of which are $O(\log n)$.

1.3 Divide-and-Conquer Algorithms

One approach to the design of algorithms is to decompose a problem into subproblems that resemble the original problem, but on a reduced scale. Suppose, for example, that we want to compute x^n. We reason

that the value we want can be computed from $x^{\lfloor n/2 \rfloor}$ because

$$x^n = \begin{cases} 1 & \text{if } n = 0, \\ (x^{\lfloor n/2 \rfloor})^2 & \text{if } n \text{ is even}, \\ x \times (x^{\lfloor n/2 \rfloor})^2 & \text{if } n \text{ is odd}. \end{cases}$$

This recursive definition can be translated directly into

```
1     int power (int x, int n) {
2      // Compute the n-th power of x
3       if (n == 0)
4         return 1;
5       else {
6         int t = power(x, floor(n/2));
7         if ((n % 2) == 0)
8           return t*t;
9         else
10          return x*t*t;
11      }
12    }
```

To analyze the time required by this algorithm, we notice that the time will be proportional to the number of multiplication operations performed in lines 8 and 10, so the divide and conquer recurrence

$$T(n) = 2 + T(\lfloor n/2 \rfloor) ,$$

with $T(0) = 0$, describes the rate of growth of the time required by this algorithm. By considering the subsequence $n_k = 2^k$, we find, using the methods of the previous section, that $T(n) = \Theta(\log n)$. Thus above algorithm is considerably more efficient than the more obvious

```
1     int power (int k, int n) {
2      // Compute the n-th power of k
3       int product = 1;
4       for (int i = 1; i <= n; i++)
5         // at this point power is k*k*k*...*k (i times)
6         product = product * k;
7       return product;
8     }
```

which requires time $\Theta(n)$.

An extremely well-known instance of divide-and-conquer algorithm is **binary search** of an ordered array of n elements for a given element—we "probe" the middle element of the array, continuing in either the lower or upper segment of the array, depending on the outcome of the probe:

```
1     int binarySearch (int x, int w[], int low, int high) {
2       // Search for x among sorted array w[low..high]. The integer
3       // returned is either the location of x in w, or the location
4       // where x belongs.
5       if (low > high) // Not found
6         return low;
7       else {
8         int middle = (low+high)/2;
9         if (w[middle] < x)
```

```
10            return binarySearch(x, w, middle+1, high);
11          else if (w[middle] == x)
12            return middle;

13          else
14            return binarySearch(x, w, low, middle-1);
15        }
16      }
```

The analysis of binary search in an array of n elements is based on counting the number of probes used in the search, since all remaining work is proportional to the number of probes. But, the number of probes needed is described by the divide-and-conquer recurrence

$$T(n) = 1 + T(n/2) \ ,$$

with $T(0) = 0$, $T(1) = 1$. We find from Table 1.2 (the top line) that $T(n) = \Theta(\log n)$. Hence, binary search is much more efficient than a simple linear scan of the array.

To multiply two very large integers x and y, assume that x has exactly $n \geq 2$ decimal digits and y has at most n decimal digits. Let $x_{n-1}, x_{n-2}, \ldots, x_0$ be the digits of x and $y_{n-1}, y_{n-2}, \ldots, y_0$ be the digits of y (some of the most significant digits at the end of y may be zeros, if y is shorter than x), so that

$$x = 10^{n-1}x_{n-1} + 10^{n-2}x_{n-2} + \cdots + x_0 \ ,$$

and

$$y = 10^{n-1}y_{n-1} + 10^{n-2}y_{n-2} + \cdots + y_0 \ .$$

We apply the divide-and-conquer idea to multiplication by chopping x into two pieces, the most significant (leftmost) l digits and the remaining digits:

$$x = 10^l x_{\text{left}} + x_{\text{right}} \ ,$$

where $l = \lfloor n/2 \rfloor$. Similarly, chop y into two corresponding pieces:

$$y = 10^l y_{\text{left}} + y_{\text{right}} \ ,$$

because y has at most the number of digits that x does, y_{left} might be 0. The product $x \times y$ can be now written

$$\begin{aligned} x \times y &= \left(10^l x_{\text{left}} + x_{\text{right}}\right) \times \left(10^l y_{\text{left}} + y_{\text{right}}\right) , \\ &= 10^{2l} x_{\text{left}} \times y_{\text{left}} \\ &\quad + 10^l \left(x_{\text{left}} \times y_{\text{right}} + x_{\text{right}} \times y_{\text{left}}\right) \\ &\quad + x_{\text{right}} \times y_{\text{right}} \ . \end{aligned}$$

If $T(n)$ is the time to multiply two n-digit numbers with this method, then

$$T(n) = kn + 4T(n/2) \ ;$$

the kn part is the time to chop up x and y and to do the needed additions and shifts; each of these tasks involves n-digit numbers and hence $\Theta(n)$ time. The $4T(n/2)$ part is the time to form the four needed subproducts, each of which is a product of about $n/2$ digits.

The line for $g(n) = \Theta(n)$, $u = 4 > v = 2$ in Table 1.2 tells us that $T(n) = \Theta(n^{\log_2 4}) = \Theta(n^2)$, so the divide-and-conquer algorithm is no more efficient than the elementary-school method of multiplication. However, we can be more economical in our formation of subproducts:

$$\begin{aligned} x \times y &= \left(10^n x_{\text{left}} + x_{\text{right}}\right) \times \left(10^n y_{\text{left}} + y_{\text{right}}\right), \\ &= 10^{2n} A + 10^n C + B \ , \end{aligned}$$

where

$$\begin{aligned} A &= x_{\text{left}} \times y_{\text{left}} \\ B &= x_{\text{right}} \times y_{\text{right}} \\ C &= \left(x_{\text{left}} + x_{\text{right}}\right) \times \left(y_{\text{left}} + y_{\text{right}}\right) - A - B \ . \end{aligned}$$

The recurrence for the time required changes to

$$T(n) = kn + 3T\,(n/2) \ .$$

The kn part is the time to do the two additions that form $x \times y$ from A, B, and C and the two additions and the two subtractions in the formula for C; each of these six additions/subtractions involves n-digit numbers. The $3T(n/2)$ part is the time to (recursively) form the three needed products, each of which is a product of about $n/2$ digits. The line for $g(n) = \Theta(n)$, $u = 3 > v = 2$ in Table 1.2 now tells us that

$$T(n) = \Theta\left(n^{\log_2 3}\right) \ .$$

Now

$$\log_2 3 = \frac{\log_{10} 3}{\log_{10} 2} \approx 1.5849625\cdots \ ,$$

which means that this divide-and-conquer multiplication technique will be faster than the straightforward $\Theta(n^2)$ method for large numbers of digits.

Sorting a sequence of n values efficiently can be done using the divide-and-conquer idea. Split the n values arbitrarily into two piles of $n/2$ values each, sort each of the piles separately, and then merge the two piles into a single sorted pile. This sorting technique, pictured in Fig. 1.2, is called *merge sort*. Let $T(n)$ be the time required by merge sort for sorting n values. The time needed to do the merging is proportional to the number of elements being merged, so that

$$T(n) = cn + 2T\,(n/2) \ ,$$

because we must sort the two halves (time $T(n/2)$ for each half) and then merge (time proportional to n). We see by Table 1.2 that the growth rate of $T(n)$ is $\Theta(n \log n)$, since $u = v = 2$ and $g(n) = \Theta(n)$.

1.4 Dynamic Programming

In the design of algorithms to solve optimization problems, we need to make the optimal (lowest cost, highest value, shortest distance, and so on) choice among a large number of alternative solutions; dynamic programming is an organized way to find an optimal solution by systematically exploring all possibilities without unnecessary repetition. Often, dynamic programming leads to efficient, polynomial-time algorithms for problems that appear to require searching through exponentially many possibilities.

Like the divide-and-conquer method, dynamic programming is based on the observation that many optimization problems can be solved by solving similar subproblems and then composing the solutions

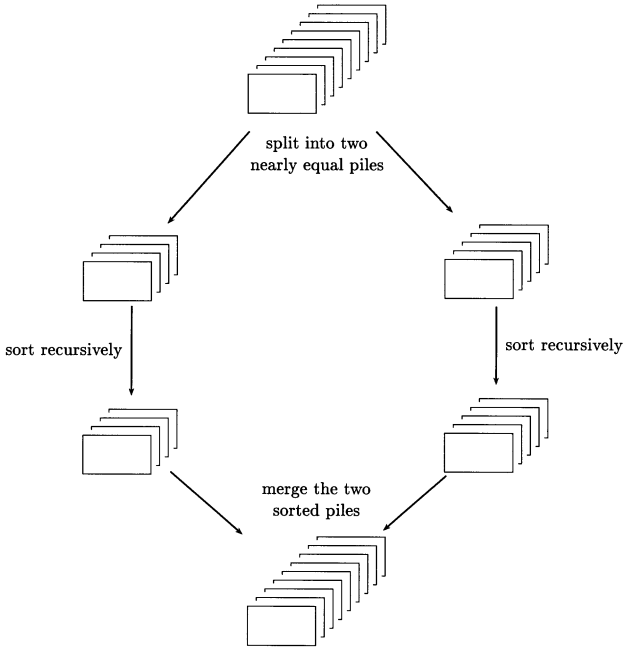


FIGURE 1.2 Schematic description of merge sort.

of those subproblems into a solution for the original problem. In addition, the problem is viewed as a sequence of decisions, each decision leading to different subproblems; if a wrong decision is made, a suboptimal solution results, so all possible decisions need to be accounted for.

As an example of dynamic programming, consider the problem of constructing an optimal search pattern for probing an ordered sequence of elements. The problem is similar to searching an array—in the previous section we described binary search in which an interval in an array is repeatedly bisected until the search ends. Now, however, suppose we know the frequencies with which the search will seek various elements (both in the sequence and missing from it). For example, if we know that the last few elements in the sequence are frequently sought—binary search does not make use of this information—it might be more efficient to begin the search at the right end of the array, not in the middle. Specifically, we are given an ordered sequence $x_1 < x_2 < \cdots < x_n$ and associated frequencies of access $\beta_1, \beta_2, \ldots, \beta_n$, respectively; furthermore, we are given $\alpha_0, \alpha_1, \ldots, \alpha_n$ where α_i is the frequency with which the search will fail because the object sought, z, was missing from the sequence, $x_i < z < x_{i+1}$ (with the obvious meaning when $i = 0$ or $i = n$). What is the optimal order to search for an unknown element z? In fact, how should we describe the optimal search order?

We express a search order as a **binary search tree**, a diagram showing the sequence of probes made in every possible search. We place at the root of the tree the sequence element at which the first probe is made, say x_i; the left subtree of x_i is constructed recursively for the probes made when $z < x_i$ and the right subtree of x_i is constructed recursively for the probes made when $z > x_i$. We label each item in the tree with the frequency that the search ends at that item. Figure 1.3 shows a simple example. The search of sequence $x_1 < x_2 < x_3 < x_4 < x_5$ according to the tree of Fig. 1.3 is done by comparing the unknown element z with x_4 (the root); if $z = x_4$, the search ends. If $z < x_4$, z is compared with x_2 (the root of the left subtree); if $z = x_2$, the search ends. Otherwise, if $z < x_2$, z is compared with x_1 (the root of the left subtree of x_2); if $z = x_1$, the search ends. Otherwise, if $z < x_1$, the search ends unsuccessfully at the leaf labeled α_0. Other results of comparisons lead along other paths in the tree from the root downward. By its nature, a binary search tree is *lexicographic* in that for all nodes in the tree, the elements in the left subtree of the node are smaller and the elements in the right subtree of the node are larger than the node.

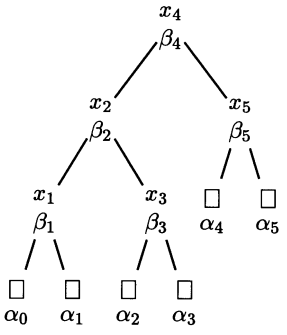


FIGURE 1.3 A binary search tree.

Because we are to find an optimal search pattern (tree), we want the cost of searching to be minimized. The cost of searching is measured by the *weighted path length* of the tree:

$$\sum_{i=1}^{n} \beta_i \times [1 + \text{ level } (\beta_i)] + \sum_{i=0}^{n} \alpha_i \times \text{ level } (\alpha_i) \ ,$$

defined formally as

$$W\left(\square\right) = 0 \ ,$$
$$W\left(T = \begin{array}{c} \bullet \\ T_l \quad T_r \end{array}\right) = W\left(T_l\right) + W\left(T_r\right) + \sum \alpha_i + \sum \beta_i \ ,$$

where the summations $\sum \alpha_i$ and $\sum \beta_i$ are over all α_i and β_i in T. Since there are exponentially many possible binary trees, finding the one with minimum weighted path length could, if done naïvely, take exponentially long.

The key observation we make is that a **principle of optimality** holds for the cost of binary search trees: subtrees of an optimal search tree must themselves be optimal. This observation means, for example, if the tree shown in Fig. 1.3 is optimal, then its left subtree must be the optimal tree for the problem of searching the sequence $x_1 < x_2 < x_3$ with frequencies $\beta_1, \beta_2, \beta_3$ and $\alpha_0, \alpha_1, \alpha_2, \alpha_3$. (If a subtree in Fig. 1.3 were *not* optimal, we could replace it with a better one, reducing the weighted path length of the entire tree because of the recursive definition of weighted path length.) In general terms, the principle of optimality states that subsolutions of an optimal solution must themselves be optimal.

The optimality principle, together with the recursive definition of weighted path length, means that we can express the construction of an optimal tree recursively. Let $C_{i,j}$, $0 \le i \le j \le n$, be the cost of an optimal tree over $x_{i+1} < x_{i+2} < \cdots < x_j$ with the associated frequencies $\beta_{i+1}, \beta_{i+2}, \ldots, \beta_j$ and $\alpha_i, \alpha_{i+1}, \ldots, \alpha_j$. Then,

$$\begin{aligned} C_{i,i} &= 0 \ , \\ C_{i,j} &= \min_{i<k\le j} \left(C_{i,k-1} + C_{k,j}\right) + W_{i,j} \ , \end{aligned}$$

where

$$\begin{aligned} W_{i,i} &= \alpha_i \ , \\ W_{i,j} &= W_{i,j-1} + \beta_j + \alpha_j \ . \end{aligned}$$

These two recurrence relations can be implemented directly as recursive functions to compute $C_{0,n}$, the cost of the optimal tree, leading to the following two functions:

```
 1     int W (int i, int j) {
 2       if (i == j)
 3         return alpha[j];
 4       else
 5         return W(i,j-1) + beta[j] + alpha[j];
 6     }
 7
 8     int C (int i, int j) {
 9       if (i == j)
10         return 0;
11      else {
12         int minCost = MAXINT;
13        int cost;
14         for (int k = i+1; k <= j; k++) {
15           cost = C(i,k-1) + C(k,j) + W(i,j);
16           if (cost < minCost)
17             minCost = cost;
18         }
19         return minCost;
20       }
21     }
```

These two functions correctly compute the cost of an optimal tree; the tree itself can be obtained by storing the values of `k` when `cost < minCost` in line 16.

However, the above functions are unnecessarily time consuming (requiring exponential time) because the same subproblems are solved repeatedly. For example, each call `W(i,j)` uses time $\Theta(j - i)$ and such calls are made repeatedly for the same values of `i` and `j`. We can make the process more efficient by caching the values of `W(i,j)` in an array as they are computed and using the cached values when possible:

```
 1     int w[n][n];
 2     for (int i = 0; i < n; i++)
 3       for (int j = 0; j < n; j++)
 4         w[i][j] = MAXINT;
 5
 6     int W (int i, int j) {
 7       if (w[i][j] == MAXINT)
 8         if (i == j)
 9           w[i][j] = alpha[j];
10         else
11           w[i][j] =  W(i,j-1) + beta[j] + alpha[j];
12       return w[i][j];
13     }
```

In the same way, we should cache the values of `C(i,j)` in an array as they are computed:

```
 1     int c[n][n];
 2     for (int i = 0; i < n; i++)
 3       for (int j = 0; j < n; j++)
 4         c[i][j] = MAXINT;
 5
 6     int C (int i, int j) {
```

```
 7       if (c[i][j] == MAXINT)
 8         if (i == j)
 9            c[i][j] = 0;
10         else {
11           int minCost = MAXINT;
12           int cost;
13           for (int k = i+1; k <= j; k++) {
14             cost = C(i,k-1) + C(k,j) + W(i,j);
15             if (cost < minCost)
16               minCost = cost;
17           }
18           c[i][j] = minCost;
19         }
20       return c[i][j];
21    }
```

The idea of caching the solutions to subproblems is crucial to making the algorithm efficient. In this case, the resulting computation requires time $\Theta(n^3)$; this is surprisingly efficient, considering that an optimal tree is being found from among exponentially many possible trees.

By studying the pattern in which the arrays `C` and `W` are filled in, we see that the main diagonal `c[i][i]` is filled in first, then the first upper super-diagonal `c[i][i+1]`, then the second upper super-diagonal `c[i][i+2]`, and so on until the upper right corner of the array is reached. Rewriting the code to do this directly, and adding an array `R[][]` to keep track of the roots of subtrees, we obtain

```
 1     int w[n][n];
 2     int R[n][n];
 3     int c[n][n];
 4
 5     // Fill in the main diagonal
 6     for (int i = 0; i < n; i++) {
 7       w[i][i] = alpha[i];
 8       R[i][i] = 0;
 9       c[i][i] = 0;
10     }
11
12     int minCost, cost;
13     for (int d = 1; d < n; d++)
14       // Fill in d-th upper super-diagonal
15       for (i = 0; i < n-d; i++) {
16         w[i][i+d] = w[i][i+d-1] + beta[i+d] + alpha[i+d];
17         R[i][i+d] = i+1;
18         c[i][i+d] = c[i][i] + c[i+1][i+d] + w[i][i+d];
19         for (int k = i+2; k <= i+d; k++) {
20           cost = c[i][k-1] + c[k][i+d] + w[i][i+d];
21           if (cost < c[i][i+d]) {
22             R[i][i+d] = k;
23             c[i][i+d] = cost;
24           }
25         }
26       }
```

which more clearly shows the $\Theta(n^3)$ behavior.

As a second example of dynamic programming, consider the **traveling salesman problem** in which a salesman must visit n cities, returning to his starting point, and is required to minimize the cost of the trip. The cost of going from city i to city j is $C_{i,j}$. To use dynamic programming we must specify an optimal tour in a recursive framework, with subproblems resembling the overall problem. Thus we define

$$T(i; j_1, j_2, \ldots, j_k) = \begin{cases} \text{cost of an optimal tour from city } i \text{ to city 1 that} \\ \text{goes through each of the cities } j_1, j_2, \ldots, j_k \\ \text{exactly once, in any order, and through no other} \\ \text{cities.} \end{cases}$$

The principle of optimality tells us that

$$T(i; j_1, j_2, \ldots, j_k) = \min_{1 \le m \le k} \left\{ C_{i,j_m} + T(j_m; j_1, j_2, \ldots, j_{m-1}, j_{m+1}, \ldots, j_k) \right\} ,$$

where, by definition,

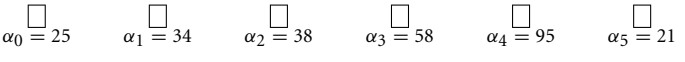
$$T(i; j) = C_{i,j} + C_{j,1} .$$

We can write a function T that directly implements the above recursive definition, but as in the optimal search tree problem, many subproblems would be solved repeatedly, leading to an algorithm requiring time $\Theta(n!)$. By caching the values $T(i; j_1, j_2, \ldots, j_k)$, we reduce the time required to $\Theta(n^2 2^n)$, still exponential, but considerably less than without caching.

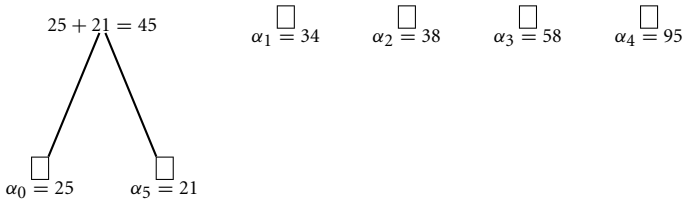
1.5 Greedy Heuristics

Optimization problems always have an objective function to be minimized or maximized, but it is not often clear what steps to take to reach the optimum value. For example, in the optimum binary search tree problem of the previous section, we used dynamic programming to examine systematically all possible trees; but perhaps there is a simple rule that leads directly to the best tree—say by choosing the largest β_i to be the root and then continuing recursively. Such an approach would be less time-consuming than the $\Theta(n^3)$ algorithm we gave, but it does not necessarily give an optimum tree (if we follow the rule of choosing the largest β_i to be the root, we get trees that are no better, on the average, than a randomly chosen trees). The problem with such an approach is that it makes decisions that are *locally optimum*, though perhaps not *globally optimum*. But, such a "greedy" sequence of locally optimum choices does lead to a globally optimum solution in some circumstances.

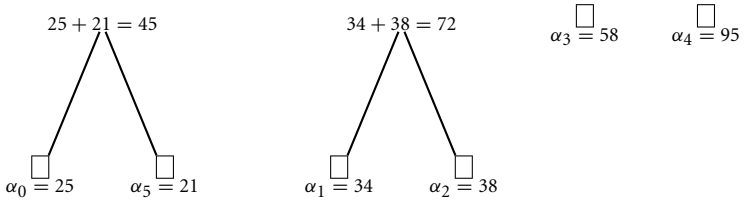
Suppose, for example, $\beta_i = 0$ for $1 \le i \le n$, and we remove the lexicographic requirement of the tree; the resulting problem is the determination of an optimal prefix code for $n + 1$ letters with frequencies $\alpha_0, \alpha_1, \ldots, \alpha_n$. Because we have removed the lexicographic restriction, the dynamic programming solution of the previous section no longer works, but the following simple greedy strategy yields an optimum tree: Repeatedly combine the two lowest-frequency items as the left and right subtrees of a newly created item whose frequency is the sum of the two frequencies combined. Here is an example of this construction; we start with five leaves with weights



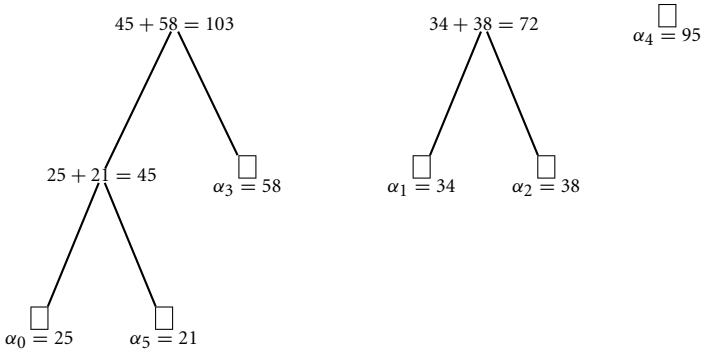
First, combine leaves $\alpha_0 = 25$ and $\alpha_5 = 21$ into a subtree of frequency $25 + 21 = 45$:



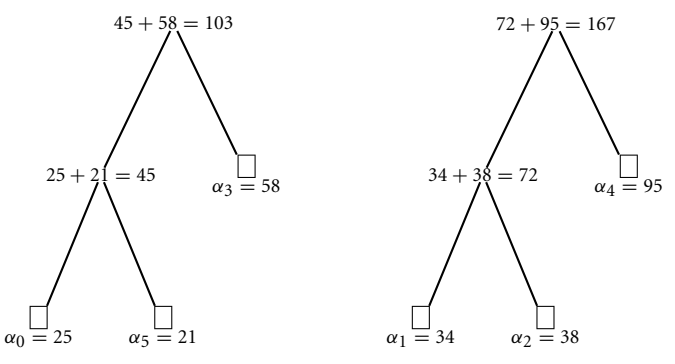
Then combine leaves $\alpha_1 = 34$ and $\alpha_2 = 38$ into a subtree of frequency $34 + 38 = 72$:



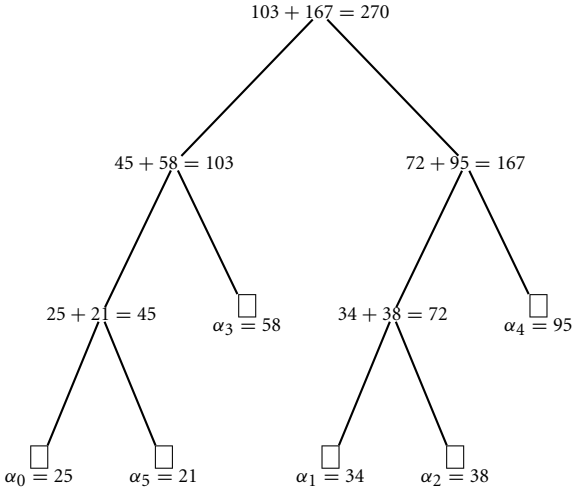
Next, combine the subtree of frequency $\alpha_0 + \alpha_5 = 45$ with $\alpha_3 = 58$:



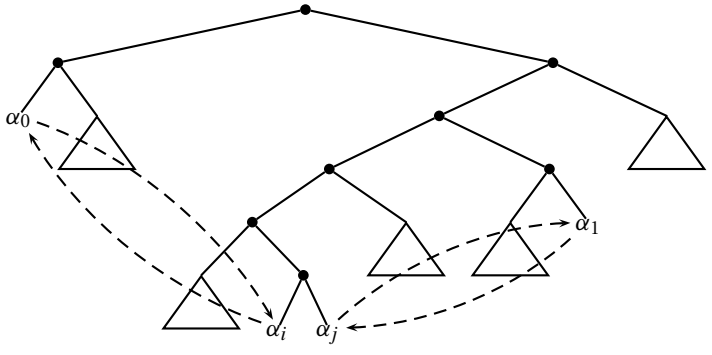
Then, combine the subtree of frequency $\alpha_1 + \alpha_2 = 72$ with $\alpha_4 = 95$:



Finally, combine the only two remaining subtrees:



How do we know that the above-outlined process leads to an optimum tree? The key to proving that the tree is optimum is to assume, by way of contradiction, that it is not optimum. In this case, the greedy strategy must have erred in one of its choices, so let us look at the *first* error this strategy made. Since all previous greedy choices were not errors, and hence lead to an optimum tree, we can assume that we have a sequence of frequencies $\alpha_0, \alpha_1, \ldots, \alpha_n$ such that the first greedy choice is erroneous—without loss of generality assume that α_0 and α_1 are two smallest frequencies, those combined erroneously by the greedy strategy. For this combination to be erroneous, there must be no optimum tree in which these two αs are siblings, so consider an optimum tree, the locations of α_0 and α_1, and the location of the two deepest leaves in the tree, α_i and α_j:



By interchanging the positions of α_0 and α_i and α_1 and α_j (as shown), we obtain a tree in which α_0 and α_1 are siblings. Because α_0 and α_1 are the two lowest frequencies (because they were the greedy algorithm's choice) $\alpha_0 \leq \alpha_i$ and $\alpha_1 \leq \alpha_j$, thus the weighted path length of the modified tree is no larger than before the modification since $\text{level}(\alpha_0) \geq \text{level}(\alpha_i)$, $\text{level}(\alpha_1) \geq \text{level}(\alpha_j)$ and hence

$$\text{level}\,(\alpha_i) \times \alpha_0 + \text{level}\,(\alpha_j) \times \alpha_1 \leq \text{level}\,(\alpha_0) \times \alpha_0 + \text{level}\,(\alpha_1) \times \alpha_1\,.$$

In other words, the first so-called mistake of the greedy algorithm was in fact not a mistake, since there is an optimum tree in which α_0 and α_1 are siblings. Thus we conclude that the greedy algorithm never makes a first mistake—that is, it never makes a mistake at all!

The greedy algorithm above is called *Huffman's algorithm*. If the subtrees are kept on a priority queue by cumulative frequency, the algorithm needs to insert the $n + 1$ leaf frequencies onto the queue, and the

repeatedly remove the two least elements on the queue, unite those to elements into a single subtree, and put that subtree back on the queue. This process continues until the queue contains a single item, the optimum tree. Reasonable implementations of priority queues will yield $O(n \log n)$ implementations of Huffman's greedy algorithm.

The idea of making greedy choices, facilitated with a priority queue, works to find optimum solutions to other problems too. For example, a **spanning tree** of a weighted, connected, undirected graph $G = (V, E)$ is a subset of $|V| - 1$ edges from E connecting all the vertices in G; a spanning tree is minimum if the sum of the weights of its edges is as small as possible. *Prim's algorithm* uses a sequence of greedy choices to determine a minimum spanning tree: Start with an arbitrary vertex $v \in V$ as the spanning-tree-to-be. Then, repeatedly add the cheapest edge connecting the spanning-tree-to-be to a vertex not yet in it. If the vertices not yet in the tree are stored in a priority queue implemented by a Fibonacci heap, the total time required by Prim's algorithm will be $O(|E| + |V| \log |V|)$. But why does the sequence of greedy choices lead to a minimum spanning tree?

Suppose Prim's algorithm does *not* result in a minimum spanning tree. As we did with Huffman's algorithm, we ask what the state of affairs must be when Prim's algorithm makes its first mistake; we will see that the assumption of a first mistake leads to a contradiction, proving the correctness of Prim's algorithm. Let the edges added to the spanning tree be, in the order added, $e_1, e_2, e_3, \ldots$, and let e_i be the first mistake. In other words, there is a minimum spanning tree $T_{\min}$ containing $e_1, e_2, \ldots, e_{i-1}$, but no minimum spanning tree containing $e_1, e_2, \ldots, e_i$. Imagine what happens if we add the edge e_i to $T_{\min}$: since $T_{\min}$ is a spanning tree, the addition of e_i causes a cycle containing e_i. Let $e_{\max}$ be the highest-cost edge on that cycle not among $e_1, e_2, \ldots, e_i$. There must be such an $e_{\max}$ because $e_1, e_2, \ldots, e_i$ are acyclic, since they are in the spanning tree constructed by Prim's algorithm. Moreover, because Prim's algorithm always makes a greedy choice—that is, chooses the lowest-cost available edge—the cost of e_i is no more than the cost of any edge available to Prim's algorithm when e_i is chosen; the cost of $e_{\max}$ is at least that of one of those unchosen edges, so it follows that the cost of e_i is no more than the cost of $e_{\max}$. In other words, the cost of the spanning tree $T_{\min} - \{e_{\max}\} \cup \{e_i\}$ is at most that of $T_{\min}$; that is, $T_{\min} - \{e_{\max}\} \cup \{e_i\}$ is also a minimum spanning tree, contradicting our assumption that the choice of e_i is the first mistake. Therefore, the spanning tree constructed by Prim's algorithm must be a minimum spanning tree.

We can apply the greedy heuristic to many optimization problems, and even if the results are not optimal, they are often quite good. For example, in the n-city traveling salesman problem, we can get near-optimal tours in time $O(n^2)$ when the intercity costs are symmetric ($C_{i,j} = C_{j,i}$ for all i and j) and satisfy the triangle inequality($C_{i,j} \leq C_{i,k} + C_{k,j}$ for all i, j, and k). The *closest insertion algorithm* starts with a "tour" consisting of a single, arbitrarily chosen city, and successively inserts the remaining cities to the tour, making a greedy choice about which city to insert next and where to insert it: the city chosen for insertion is the city not on the tour but closest to a city on the tour; the chosen city is inserted adjacent to the city on the tour to which it is closest.

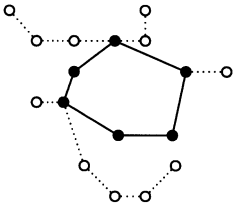
Given an $n \times n$ symmetric distance matrix C that satisfies the triangle inequality, let I_n of length $|I_n|$ be the "closest insertion tour" produced by the closest insertion heuristic and let O_n be an optimal tour of length $|O_n|$. Then

$$\frac{|I_n|}{|O_n|} < 2 .$$

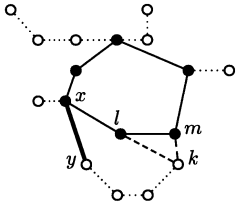
This bound is proved by an incremental form of the optimality proofs for greedy heuristics we have seen above: we ask not where the first error is, but by how much we are in error at each greedy insertion to the tour—we establish a correspondence between edges of the optimal tour O_n and cities inserted on the closest insertion tour. We show that at each insertion of a new city to the closest insertion tour, the additional length added by that insertion is at most twice the length of corresponding edge of the optimal tour O_n.

To establish the correspondence, imagine the closest insertion algorithm keeping track not only of the current tour, but also of a spider-like configuration including the edges of the current tour (the body of

the spider) and pieces of the optimal tour (the legs of the spider). We show the current tour in solid lines and the pieces of optimal tour as dotted lines:



Initially, the spider consists of the arbitrarily chosen city with which the closest insertion tour begins and the legs of the spider consist of all the edges of the optimal tour *except* for one edge eliminated arbitrarily. As each city is inserted into the closest insertion tour, the algorithm will delete from the spider-like configuration one of the dotted edges from the optimal tour. When city k is inserted between cities l and m, the edge deleted is the one attaching the spider to the leg that contains the city inserted (from city x to city y), shown here in bold:



Now,

$$C_{k,m} \leq C_{x,y} \ ,$$

because of the greedy choice to add city k to the tour and not city y. By the triangle inequality,

$$C_{l,k} \leq C_{l,m} + C_{m,k} \ ,$$

and by symmetry we can combine these two inequalities to get

$$C_{l,k} \leq C_{l,m} + C_{x,y} \ .$$

Adding this last inequality to the first one above,

$$C_{l,k} + C_{k,m} \leq C_{l,m} + 2C_{x,y} \ ,$$

that is,

$$C_{l,k} + C_{k,m} - C_{l,m} \leq 2C_{x,y} \ .$$

Thus adding city k between cities l and m adds no more to I_n than $2C_{x,y}$. Summing these incremental amounts over the cost of the entire algorithm tells us

$$|I_n| \leq 2\,|O_n| \ ,$$

as we claimed.

1.6 Lower Bounds

In Subsection "Sorting" and Section 1.3 we saw that we could sort faster than naïve $\Theta(n^2)$ worst-case behavior algorithms: we designed more sophisticated $\Theta(n \log n)$ worst-case algorithms. Can we do still better? No, $\Theta(n \log n)$ is a **lower bound** on sorting algorithms based on comparisons of the items being sorted. More precisely, let us consider only sorting algorithms described by decision boxes of the form

$$x_i : x_j$$

and outcome boxes such as

$$\boxed{x_1 < x_2 < x_3}$$

Such diagrams are called *decision trees*. Figure 1.4 shows a decision tree for sorting the three elements x_1, x_2, and x_3.

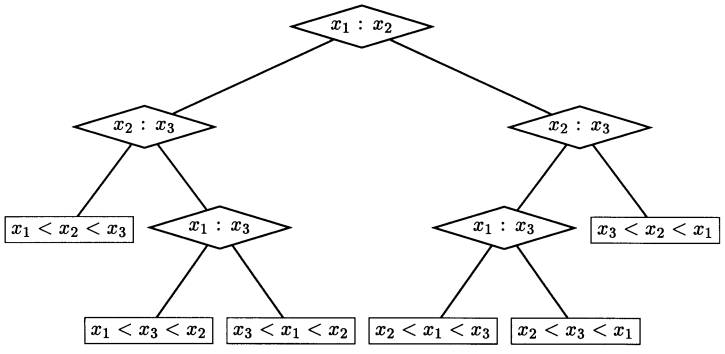


FIGURE 1.4 A decision tree for sorting the three elements x_1, x_2, and x_3.

Restricting ourselves to sorting algorithms represented by decision trees eliminates algorithms not based on comparisons of the elements, but it also appears to eliminate from consideration any of the common sorting algorithms, such as insertion sort, heapsort, and mergesort, all of which use index manipulations in loops, auxiliary variables, recursion, and so on. Furthermore, we have not allowed the algorithms to consider the possibility that some of the elements to be sorted may have equal values. These objections to modeling sorting algorithms on decision trees are serious, but can be countered by arguments that we have not been too restrictive.

For example, disallowing elements that are equal can be defended, because we certainly expect any sorting algorithm to work correctly in the special case that all of the elements are different; we are just examining an algorithm's behavior in this special case—a lower bound in a special case gives a lower bound on the general case. The objection that such normal programming techniques as auxiliary variables, loops, recursion, and so on are disallowed can be countered by the observation that *any* sorting algorithm based on comparisons of the elements can be stripped of its programming implementation to yield a decision tree. We expand all loops and all recursive calls, ignoring data moves and keeping track only of the comparisons between elements and nothing else. In this way, all common sorting algorithms can be described by decision trees.

We make an important observation about decision trees and the sorting algorithms represented as decision trees: *If a sorting algorithm correctly sorts all possible input sequences of n items, then the corresponding decision tree has n! outcome boxes.* This observation follows by examining the correspondence between permutations and outcome boxes. Since the decision tree arose by tracing through the algorithm for all

possible input sequences (that is, permutations), an outcome box must have occurred as the result of some input permutation or it would not be in the decision tree. Moreover, it is impossible that there are two different permutations corresponding to the same outcome box—such an algorithm cannot sort all input sequences correctly. Since there are $n!$ permutations of n elements, the decision tree has $n!$ leaves (outcome boxes).

To prove the $\Theta(n \log n)$ lower bound, define the *cost* of the ith leaf in the decision tree, $c(i)$, to be the number of element comparisons used by the algorithm when the input permutation causes the algorithm to terminate at the ith leaf. In other words, $c(i)$ is the *depth* of the ith leaf. This measure of cost ignores much of the work in the sorting process, but the overall work done will be proportional to the depth of the leaf at which the sorting algorithm terminates; because we are concerned only with lower bounds with in the Θ-notation, this analysis suffices.

Kraft's inequality tells us that for any tree with N leaves,

$$\sum_{i=1}^{N} \frac{1}{2^{c(i)}} \leq 1 \, . \tag{1.2}$$

We can prove this inequality by induction on the height of the tree: When the height is zero, there is one leaf of depth zero and the inequality is trivial. When the height is non-zero, the inequality applies inductively to the left and right subtrees; the edges from the root to these subtrees increases the depth of each leaf by one, so the sum over each of the two subtrees is 1/2 and the inequality follows.

We use Kraft's inequality by letting h be the height of a decision tree corresponding to a sorting algorithm applied to n items. Then h is the depth of the deepest leaf, that is, the worst-case number of comparisons of the algorithm: $h \geq c(i)$, for all i. Therefore,

$$\begin{aligned} \frac{N}{2^h} &= \sum_{i=1}^{N} \frac{1}{2^h} \\ &\leq \sum_{i=1}^{N} \frac{1}{2^{c(i)}} \\ &\leq 1 \, , \end{aligned}$$

and so

$$N \leq 2^h \, .$$

However, we saw that $N = n!$, so this last inequality can be rewritten as

$$2^h \geq n! \, .$$

But

$$n! \geq \left(\frac{n}{2}\right)^{n/2} \, ,$$

so that

$$h \geq \log_2 n! = \Theta(n \log n) \, ,$$

which is what we wanted to prove.

We can make an even stronger statement about sorting algorithms that can be modeled by decision trees: It is impossible to sort in *average time* better than $\Theta(n \log n)$, if each of the $n!$ input permutations is equally likely to be the input. The average number of decisions in this case is

$$\frac{1}{N} \sum_{i=1}^{N} c(i) \, .$$

Suppose this is less than $\log_2 N$; that is, suppose

$$\sum_{i=1}^{N} c(i) < N \log_2 N \ .$$

By the arithmetic/geometric mean inequality, we know that

$$\frac{1}{m} \sum_{i=1}^{m} u_i \geq \left(\prod_{i=1}^{m} u_i \right)^{1/m} \ . \tag{1.3}$$

Applying this inequality, we have

$$\begin{aligned} \sum_{i=1}^{N} \frac{1}{2^{c(i)}} &\geq N \left(\prod_{i=1}^{N} \frac{1}{2^{c(i)}} \right)^{1/N} \\ &= N \left(2^{-\sum_{i=1}^{N} c(i)} \right)^{1/N} \\ &> N \left(2^{-N \log_2 N} \right)^{1/N} , \end{aligned}$$

by assumption,

$$\begin{aligned} &= N \left(N^{-N} \right)^{1/N} \\ &= 1 , \end{aligned}$$

contradicting Kraft's inequality.

The lower bounds on sorting are called *information theoretic* lower bounds, because the rely on the amount of "information" contained in a single decision (comparison); in essence, the best a comparison can do is divide the set of possibilities into two equal parts. Such bounds also apply to many searching problems—for example, such arguments prove that binary search is, in a sense, optimal.

Information theoretic lower bounds do not always give useful results. Consider the **element uniqueness problem,** the problem of determining if there are any duplicate numbers in a set of n numbers, $x_1, x_2, \ldots, x_n$. Since there are only two possible outcomes, yes or no, the information theoretic lower bound says that a single comparison should be sufficient to answer the question. Indeed, that is true: Compare the product

$$\prod_{1 \leq i < j \leq n} \left(x_i - x_j \right) \tag{1.4}$$

to zero. If the product is non-zero, there are no duplicate numbers; if it is zero there are duplicates.

Of course, the cost of the one comparison is negligible compared to the cost of computing the product (1.4). It takes $\Theta(n^2)$ arithmetic operations to determine the product, but we are ignoring this dominant expense. The resulting lower bound is ridiculous.

To obtain a sensible lower bound for the element uniqueness problem, we define an **algebraic computation tree** for inputs $x_1, x_2, \ldots, x_n$ as a tree in which every leaf is either "yes" or "no." Every internal node either is a binary node (that is, with two children) based on a comparison of values computed in the ancestors of that binary node, or is a unary node (that is, with one child) that computes a value based on constants and values computed in the ancestors of that unary node, using the operations of addition, subtraction, multiplication, division, and square roots. An algebraic computation tree thus describes functions that take n numbers and compute a yes-or-no answer using intermediate algebraic results. The cost of an algebraic computation tree is its height.

By a complicated argument based on algebraic geometry, one can prove that any algebraic computation tree for the element uniqueness problem has depth at least $\Theta(n \log n)$. This is a much more sensible, satisfying lower bound on the problem. It follows from this lower bound that a simple sort-and-scan algorithm is essentially optimal for the element uniqueness problem.

1.7 Defining Terms

Algebraic computation tree: A tree combining simple algebraic operations with comparisons of values.

Amortized cost: The cost of an operation considered to be spread over a sequence of many operations.

Average-case cost: The sum of costs over all possible inputs divided by the number of possible inputs.

Binary search tree: A binary tree that is lexicographically arranged so that, for every node in the tree, the nodes to its left are smaller and those to its right are larger.

Binary search: Divide-and-conquer search of a sorted array in which the middle element of the current range is probed so as to split the range in half.

Divide-and-conquer: A paradigm of algorithm design in which a problem is solved by reducing it to subproblems of the same structure.

Dynamic programming: A paradigm of algorithm design in which an optimization problem is solved by a combination of caching subproblem solutions and appealing to the "principle of optimality."

Element uniqueness problem: The problem of determining if there are duplicates in a set of numbers.

Greedy heuristic: A paradigm of algorithm design in which an optimization problem is solved by making locally optimum decisions.

Heap: A tree in which parent–child relationships are consistently "less than" or "greater than."

Information theoretic bounds: Lower bounds based on the rate at which information can be accumulated.

Kraft's inequality: The statement that $\sum_{i=1}^{N} 2^{-c(i)} \leq 1$, where the sum is taken over the N leaves of a binary tree and $c(i)$ is the depth of leaf i.

Lower bound: A function (or growth rate) below which solving a problem is impossible.

Merge sort: A sorting algorithm based on repeated splitting and merging.

Principle of optimality: The observation, in some optimization problems, that components of a globally optimum solution must themselves be globally optimal.

Priority queue: A data structure that supports the operations of creation, insertion, minimum, deletion of the minimum, and (possibly) decreasing the value an element, deletion, or merge.

Recurrence relation: The specification of a sequence of values in terms of earlier values in the sequence.

Sorting: Rearranging a sequence into order.

Spanning tree: A connected, acyclic subgraph containing all of the vertices of a graph.

Traveling salesman problem: The problem of determining the optimal route through a set of cities, given the intercity travel costs.

Worst-case cost: The cost of an algorithm in the most pessimistic input possibility.

References

[1] Cormen, T.H., Leiserson, C.E., and Rivest, R.L., *Introduction to Algorithms,* McGraw-Hill, New York, 1990.

[2] Fredman, M.L. and Tarjan, R.E., "Fibonacci heaps and their use in improved network optimization problems," *J. ACM,* **34**, 596–615, 1987.

[3] Greene, D.H. and Knuth, D.E., *Mathematics for the Analysis of Algorithms,* 3rd ed., Birkhäuser, Boston, 1990.

[4] Knuth, D.E., *The Art of Computer Programming, Volume 1: Fundamental Algorithms,* 3rd ed., Addison-Wesley, Reading, MA, 1997.

[5] Knuth, D.E., *The Art of Computer Programming, Volume 2: Seminumerical Algorithms,* 3rd ed., Addison-Wesley, Reading, MA, 1997.

[6] Knuth, D.E., *The Art of Computer Programming, Volume 3: Sorting and Searching,* 2nd ed., Addison-Wesley, Reading, MA, 1997.

[7] Lueker, G.S., "Some techniques for solving recurrences," *Computing Surveys,* **12**, 419–436, 1980.

[8] Mehlhorn, K., *Data Structures and Algorithms 1: Sorting and Searching,* Springer-Verlag, Berlin, 1984.

[9] Reingold, E.M. and Hansen, W.J., *Data Structures in Pascal,* Little, Brown and Company, Boston, 1986.

[10] Reingold, E.M., Nievergelt, J., and Deo, N., *Combinatorial Algorithms, Theory and Practice,* Prentice-Hall, Englewood Cliffs, NJ, 1977.

[11] Rosencrantz, D.J., Stearns, R.E., and Lewis, P.M., "An analysis of several heuristics for the traveling salesman problem," *SIAM J. Comp.,* **6**, 563–581, 1977.

[12] Tarjan, R.E., *Data Structures and Network Algorithms,* Society of Industrial and Applied Mathematics, Philadelphia, PA, 1983.

Further Information

General discussions of the analysis of algorithms and data structures can be found in [1, 4], and [10]; [9] has a more elementary treatment. Both [3] and [7] contain detailed treatments of recurrences, especially in regard to the analysis of algorithms. Sorting and searching techniques are explored in depth in [5, 6] discusses algorithms for problems such as computing powers, evaluating polynomials, and multiplying large numbers. Reference [12] discusses many important graph algorithms, including several for finding minimum spanning trees. Our discussion of Fibonacci heaps is from [2]; our discussion of the heuristics for the traveling salesman problem is from [11]. A detailed discussion of the lower bound of the element-uniqueness problem is presented in [8, vol. 1, pp. 75–79], along with much other material on algebraic computation trees.

2

Searching

Ricardo Baeza-Yates
Universidad de Chile

Patricio V. Poblete
Universidad de Chile

2.1 Introduction

Searching is one of the main applications in computers as well as in other fields, including daily life. The basic problem consists in finding a given object in a set of objects of the same kind. Databases are perhaps the best example where searching is the main task involved, and also where its performance is crucial.

We use the *dictionary problem* as a generic example of searching for a key in a set of keys. Formally, we are given a set S of n distinct keys[1] $x_1, \ldots, x_n$, and we have to implement the following operations, for a given key x:

$$\begin{aligned} \texttt{Search}: &\quad x \in S? \\ \texttt{Insert}: &\quad S \leftarrow S \cup \{x\} \\ \texttt{Delete}: &\quad S \leftarrow S - \{x\} \end{aligned}$$

Although for simplicity we treat the set S as just a set of keys, in practice it would consist of a set of records, one of whose fields would be designated as the key. Extending the algorithms to cover this case is straightforward.

[1] We will not consider in detail the case of nondistinct keys. Most of the algorithms work in that case too, or can be extended without much effort, but the performance may not be the same, especially in degenerate cases.

Searches have always two possible outcomes. A search can be *successful* or *unsuccessful*, depending on whether the key was found or not in the set. We will use the letter U to denote the cost of an unsuccessful search, and S to denote the cost of a successful search. In particular, we will use the name U_n (respectively, S_n) to denote the random variable "cost of an unsuccessful (respectively, successful) search for a random element in a table built by random insertions." Unless otherwise noted, we assume that the elements to be accessed are chosen with uniform probability. The notations C'_n and C_n have been used in the literature to denote the expected values of U_n and S_n, respectively [22]. We use the notation $\mathbf{E}X$ to denote the expected value of the random variable X.

In this chapter we cover the most basic searching algorithms which work on fixed size arrays or tables and linked lists. They include techniques to search an array (unsorted or sorted), **self-organizing strategies** for arrays and lists, and hashing. In particular, hashing is a widely used method to implement dictionaries. We cover here the basic algorithms, and we provide pointers to the related literature. With the exception of hashing, we emphasize the SEARCH operation, because updates require $O(n)$ time. We also include a summary of other related searching problems.

2.2 Sequential Search

Consider the simplest problem: search for a given element in a set of n integers. If the numbers are given one by one (this is called an *on-line* problem) the obvious solution is to use sequential search. That is, we compare every element, and in the worst case we need n comparisons (either it is the last element or it is not present). Under the traditional RAM model, this algorithm is optimal. This is the algorithm used to search in an unsorted array storing n elements, and is advisable when n is small or when we do not have enough time or space to store the elements (for example in a very fast communication line). Clearly, $U_n = n$. If finding an element in any position has the same probability, then $\mathbf{E}S_n = \frac{n+1}{2}$.

Randomized Sequential Search

We can improve the worst case of sequential search in a probabilistic sense if the element belongs to the set (successful search) and we have all the elements in advance (*off-line* case). Consider the following **randomized algorithm.** We flip a coin. If it is a heads, we search the set from 1 to n. Otherwise, from n to 1. The worst case for each possibility is n comparisons. However, we have two algorithms and not only one. Suppose that the element we are looking for is in position i and that the coin is fair (that is, the probability of heads or tails is the same). So, the number of comparisons to find the element is i if it is heads, or $n - i + 1$ if it is tails. So, averaging over both algorithms (note that we are not averaging over all possible inputs), the expected worst case is

$$\frac{1}{2} \times i + \frac{1}{2} \times (n - i + 1) = \frac{n+1}{2}$$

which is independent of where the element is! This is better than n. In other words, an adversary would have to place the element in the middle position because he/she does not know which algorithm will be used.

Self-Organizing Heuristics

If the probability of retrieving each element is not the same, we can improve a successful search by ordering the elements by decreasing probability of access, either in an array or a linked list. Let p_i be the probability of accessing element i, and assume without loss of generality that $p_i > p_{i+1}$. Then, we have that the

optimal static order (OPT) has

$$\mathbf{E}S_n^{OPT} = \sum_{i=1}^{n} i\ p_i$$

However, most of the time we do not know the accessing probabilities and in practice they may change over time. For that reason, there are several heuristics to *reorganize* dynamically the order of the list. The most common ones are *move-to-front* (MF) where we promote the accessed element to the first place of the list, and *transpose* (T) where we advance the accessed element one place in the list (if it is not the first). These two heuristics are *memoryless* in the sense that they work only with the element currently accessed. MF is best suited for a linked list while T can also be applied to arrays. A good heuristic if access probabilities do not change much with time is the *count* (C) heuristic. In this case every element keeps a counter with the number of times it has been accessed and advances in the list one or more positions when its count is larger than previous elements in the list. The main disadvantage of C is that we need $O(n)$ extra space to store the counters if they fit in a word. Other more complex heuristics have been proposed, which are hybrids of the basic ones or/and use limited memory. They can also be extended to double-linked lists or more complex data structures as search trees.

Using these heuristics is advisable for small n, when space is severely limited, or when the performance obtained is good enough.[2] Evaluating how good a self-organizing strategy is with respect to the optimal order is not easily defined, as the order of the list is dynamic and not static. One possibility is to use the asymptotic expected successful search time, that is, the expected search time achieved by the algorithm after a very large sequence of independent accesses averaged over all possible initial configurations and sequences according to stable access probabilities. In this case, we have that

$$\mathbf{E}S_n^T \leq \mathbf{E}S_n^{MF} \leq \frac{\pi}{2}\mathbf{E}S_n^{OPT} \approx 1.57\mathbf{E}S_n^{OPT}$$

and $\mathbf{E}S_n^C = \mathbf{E}S_n^{OPT}$.

Another possible analysis is to use the worst-case search cost, but usually this is not fair because many times the worst-case situation does not repeat very often. A more realistic solution is to consider the **amortized cost**. That is, the average number of comparisons over a worst-case sequence of executions. Then, a costly single access can be amortized with cheaper accesses that follow after. In this case, starting with an empty list, we have

$$S^{MF} \leq 2S^{OPT}$$

and

$$S^{C} \leq 2S^{OPT}$$

while S^T can be as bad as $O(mS^{OPT})$ for m operations. If we consider a nonstatic optimal algorithm, that is, an algorithm that knows the sequence of accesses in advance and can rearrange the list with every access to minimize the search cost, then the results change. Under the assumption that the access cost function is convex, that is, if $f(i)$ is the cost of accessing the i-th element, then $f(i) - f(i-1) \geq f(i+1) - f(i)$. In this case we usually have $f(i) = i$, and then only MF satisfies the inequality

$$S^{MF} \leq 2S^{OPT}$$

for this new notion of optimal algorithm. In this case, T and C may cost $O(m)$ times the cost of the optimal algorithm for m operations. Another interesting measure is how fast a heuristic converges to the asymptotic behavior. For example, T converges more slowly than MF but it is more stable. However, MF it is more robust as seen in the amortized case.

---

[2] Also when linked lists are an internal component of other algorithms, like hashing with chaining, which is explained later.

2.3 Sorted Array Search

In the off-line case we can search faster if we allow some time to preprocess the set and the elements can be ordered. Certainly, if we sort the set (using $O(n \log n)$ comparisons in the worst-case) and we store it in an array, we can use the well-known binary search. Binary search uses divide and conquer to quickly discard half of the elements by comparing the searched key with the element in the middle of the array, and if not equal, following the search recursively either on the first half or the second half (if the searched key was smaller or larger, respectively). Using binary search we can solve the problem using at most $U_n = \lceil \log_2(n+1) \rceil$ comparisons. Therefore, if we do many searches we can amortize the cost of the initial sorting.

On average, a successful search is also $O(\log n)$. In practice we do not have three-way comparisons, so it is better to search recursively until we have discarded all but one element and then comparing for equality. Binary search is optimal for the RAM comparison model in the worst and the average case. However, by assuming more information about the set or changing the model, we can improve the average or the worst case, as shown in the next sections.

Parallel Binary Search

Suppose now that we change the model by having p processors with a shared memory. That is, we use a *parallel RAM* (PRAM) model. Can we speed up binary search? First, we have to define how the memory is accessed in a concurrent way. The most used model is CREW, that is, concurrent read, but exclusive write (otherwise it is difficult to know the final value of a memory cell after a writing operation). In a **CREW PRAM**, we can use the following simple parallel binary search. We divide the sorted set in $p+1$ segments (then, there are p internal segment boundaries). Processor i compares the key to the element stored in the i-th boundary and writes in a variable c_i a 0 if it is greater or a 1 if it is smaller (in case of equality the search ends). All the processors do this in parallel. After this step, there is an index j such that $c_j = 0$ and $c_{j+1} = 1$ (we assume that $c_0 = 0$ and $c_{p+1} = 1$), which indicates in what segment the key should be. Then, processor i compares c_i and c_{i+1} and if they are different writes the new boundaries where the search continues recursively (see Fig. 2.1). This step is also done in parallel (processor 1 and p take care of the extreme cases). When the segment is of size p or less, each processor compares one element and the search ends. Then, the worst-case number of parallel key comparisons is given by

$$U_n = 1 + U_{\frac{n}{p+1}}, \quad U_i = 1 \ (i \leq p)$$

which gives $U_n = \log_{p+1} n + O(p)$. That is, $U_n = O(\log n / \log(p+1))$. Note that for $p = 1$ we obtain the binary search result, as expected. It is possible to prove that is not possible to do it better. In the PRAM model, the optimal speed-up is when the work done by p processors is p times the work of the optimal sequential algorithm. In this case, the total work is $p \log n / \log p$, which is larger than $\log n$. In other words, searching in a sorted set cannot be solved with optimal speed-up. If we restrict the PRAM model also to exclusive reads (EREW), then $U_n = O(\log n - \log p)$, which is even worse. This is because, at every recursive step, if all the processors cannot read the new segment concurrently, we slow down all the process.

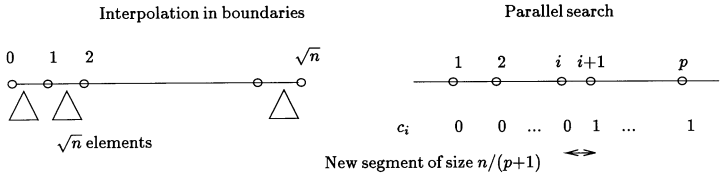


FIGURE 2.1 Binary interpolation (left) and parallel binary (right) search.

Interpolation Search

Assume now that the distribution of the n integers is uniform over a fixed range of $M >> n$ integers. Then, instead of using binary search to divide the set, we can linearly interpolate the position of the searched element with respect to the smallest and the largest element. In this way, on average, it is possible to prove that $O(\log\log n)$ comparisons are needed if the element is in the set. The proof is quite involved and mathematical [29], but there are some variations of interpolation search that have a simpler analysis. The main fact behind the $O(\log\log n)$ complexity is that when we do the first interpolation, with very high probability the searched element is at distance $O(\sqrt{n})$. So, the expected number of comparisons is given by the recurrence

$$\mathbf{E}S_n = a + \mathbf{E}S_{b\sqrt{n}}, \quad \mathbf{E}S_1 = 1$$

for some constants a and b, which gives $\mathbf{E}S_n = O(\log\log n)$. A simple variation is the following, which is called binary interpolation search [28]. Imagine that we divide the ordered set in $\sqrt{n}$ segments of size approximately $\sqrt{n}$. Then, we use interpolation search on the $\sqrt{n}+1$ keys that are segment boundaries (including the first and the last key as shown in Fig. 2.1, left) to find in what segment the key is. After we know the segment, we apply the same algorithm recursively in it. We can think that we have a $\sqrt{n}$-ary search tree where in each node we use interpolation search to find the right pointer. By a simple probabilistic analysis it is possible to show that on average less than 2.5 comparisons are needed to find in what segment is the key. So, we can use the previous recurrence with $a = 2.5$ and $b = 1$, obtaining less than $2.5\log_2\log_2 n$ comparisons on average.

2.4 Hashing

If the keys are drawn from a universe $U = \{0, \ldots, u-1\}$, where u is a reasonably small natural number, a simple solution is to use a table $T[0..u-1]$, indexed by the keys. Initially, all the table elements are initialized to a special value **empty**. When element x is inserted, the corresponding record is stored in the entry $T[x]$.

In the case when all we need to know is whether a given element is present or not, it is enough for $T[x]$ to take only two values: 0 (**empty**) and 1 (**not empty**), and the resulting data structure is called a *bit vector*.

Using this approach, all three basic operations (INSERT, SEARCH, DELETE) take time $\Theta(1)$ in the worst case.

When the size of the universe is much larger, as is the case for character strings, the same approach could still work in principle, as strings can be interpreted as (possibly very large) natural numbers, but the size of the table would make it impractical. A solution is to map the keys onto a relatively small integer range, using a function called a **hash function.**

The resulting data structure, called *hash tables*, makes it possible to use keys drawn from an arbitrarily large universe as "subscripts," much in the way small natural numbers are used as subscripts for a normal array. They are the basis for the implementation of the "associative arrays" available in some languages.

More formally, suppose we want to store our set of size n in a table of size m. (The ratio $\alpha = n/m$ is called the *load factor* of the table.) Assume we have a *hash function* h that maps each key $x \in U$ to an integer value $h(x) \in [0..m-1]$. The basic idea is to store key x in location $T[h(x)]$.

Typically, hash functions are chosen so that they generate "random looking" values. For example, the following is a function that usually works well:

$$h(x) = x \bmod m$$

where m is a prime number.

The preceding function assumes that x is an integer. In most practical applications, x is a character string instead. Strings are sequences of characters, each of which has an internal representation as a small

natural number (e.g., using the ASCII coding). If a string x can be written as $c_k c_{k-1} \ldots c_1 c_0$, where each c_i satisfies $0 \leq c_i < C$, then we can compute h as

$$h \leftarrow 0; \quad \textbf{for } i \textbf{ in } 0..k \textbf{ do } h \leftarrow (h * C + c_i) \textbf{ mod } m$$

There is one important problem that needs to be solved. As keys are inserted in the table, it is possible that we may have *collisions* between different keys hashing to the same table slot. If the hash function distributes the elements uniformly over the table, the number of collisions cannot be too large on the average (after all, the expected number of elements per slot is α), but the well known *birthday paradox* makes it very likely that there will be at least one collision, even for a lightly loaded table.

There are two basic methods for handling collisions in a hash table: **chaining** and **open addressing.**

Chaining

The simplest chaining method stores elements in the table as long as collisions do not occur. When there is a collision, the incoming key is stored in an *overflow area*, and the corresponding record is appended at the end of a linked list that stores all elements that hashed to that same location (see Fig. 2.2). The original hash table is then called the *primary area*. Figure 2.2 shows the result of inserting keys $A, B, \ldots, I$ in a

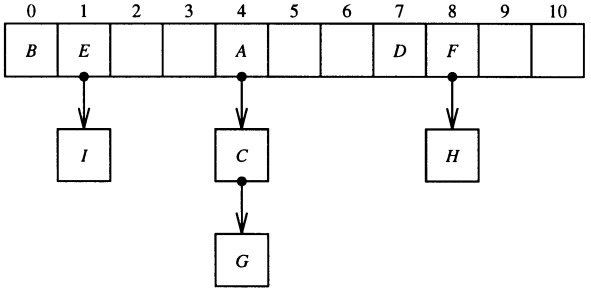


FIGURE 2.2 Hashing with separate chaining.

hash table using chaining to resolve collisions, with the following hash function:

x	A	B	C	D	E	F	G	H	I
$h(x)$	4	0	4	7	1	8	4	8	1

If the hash function maps elements uniformly, and if the elements are drawn at random from the universe, the expected values for these performance measures U_n and S_n are

$$\mathbf{E}U_n = e^{-\alpha} + \alpha + \Theta\left(\frac{1}{m}\right)$$

and

$$\mathbf{E}S_n = 1 + \frac{\alpha}{2} + \Theta\left(\frac{1}{m}\right)$$

Note that the search cost is basically independent of the number of elements, and that it depends on the load factor instead. By making the latter low enough, we can have hash tables with very efficient average search times.

The worst case, on the other hand, can be very bad: if all the keys happen to hash to the same location, the search cost is $\Theta(n)$. The probability of this happening is, of course, exceedingly small, so a more

realistic measure of the worst case may be the expected length of the longest chain in a table. This can be shown to be $\Theta\left(\frac{\log m}{\log\log m}\right)$ [16].

Deletions are handled simply by removing the appropriate element from the list. When the element happened to be in the primary area, the first remaining element in the chain must be promoted to the primary area.

The need for an overflow area can be eliminated by storing these records in table locations that happen to be empty. The resulting method, called *coalesced hashing*, has slightly larger search times, because of unrelated chains fusing accidentally, but it is still efficient even for a full table ($\alpha = 1$):

$$\begin{aligned} \mathbf{E}U_n &= 1 + \frac{1}{4}\left(e^{2\alpha} - 1 - 2\alpha\right) + \Theta\left(\frac{1}{m}\right) \\ \mathbf{E}S_n &= 1 + \frac{1}{8\alpha}\left(e^{2\alpha} - 1 - 2\alpha\right) + \frac{\alpha}{4} + \Theta\left(\frac{1}{m}\right) \end{aligned}$$

Deletions require some care, as simply declaring a given location **empty** may confuse subsequent searches. If the rest of the chain contains an element that hashes to the now empty location, it must be moved there, and the process must be repeated for the new vacant location, until the chain is exhausted. In practice, this is not as slow as it sounds, as chains are usually short.

The preceding method can be generalized by allocating an overflow area of a given size, and storing the colliding elements there as long as there is space. Once the overflow area (called the *cellar* in this method) becomes full, the empty slots in the primary area begin to be used. This data structure was studied by Vitter and Chen [35]. By appropriately tuning the relative sizes of the primary and of the overflow areas, this method can outperform the other chaining algorithms. Even at a load of 100%, an unsuccessful search requires only 1.79 probes.

Vitter and Chen's analysis of coalesced hashing is very detailed, and also very complex. An alternative approach to this problem has been used by Siegel [32] to obtain a much simpler analysis that leads to more detailed results.

Open Addressing

This is a family of methods that avoids the use of pointers, by computing a new hash value every time there is a collision.

Formally, this can be viewed as using a sequence of hash functions $h_0(x), h_1(x), \ldots$. An insertion probes that sequence of locations until finding an empty slot. Searches follow that same probe sequence, and are considered unsuccessful as soon as they hit an empty location.

The simplest way to generate the probe sequence is by first evaluating the hash function, and then scanning the table sequentially from that location (and wrapping around the end of the table). This is called *linear probing*, and is reasonably efficient if the load factor is not too high, but, as the table becomes full, it is too slow to be practical:

$$\begin{aligned} \mathbf{E}U_n &= \frac{1}{2}\left(1 + \frac{1}{1-\alpha}\right) + \Theta\left(\frac{1}{m}\right) \\ \mathbf{E}S_n &= \frac{1}{2}\left(1 + \frac{1}{(1-\alpha)^2}\right) + \Theta\left(\frac{1}{m}\right) \end{aligned}$$

Note that these formulae break down for $\alpha = 1$. For a full table, the unsuccessful and the successful search costs are $\Theta(m)$ and $\Theta(\sqrt{m})$, respectively.

A better method for generating the probe sequences is *double hashing*. In addition to the original hash function $h(x)$, a second hash function $s(x) \in [1..m-1]$ is used, to provide a "step size." The probe sequence is then generated as

$$h_0(x) = h(x); \quad h_{i+1}(x) = (h_i(x) + s(x)) \bmod m$$

Figure 2.3 shows the result of inserting keys $A, B, \ldots, I$ using the following hash functions:

x	A	B	C	D	E	F	G	H	I
$h(x)$	4	0	4	7	1	8	4	8	1
$s(x)$	5	1	4	2	5	3	9	2	9

Analyzing double hashing is quite hard [18, 26], and for this reason most mathematical analyses are instead done assuming one of two simplified models:

- *Uniform probing:* the locations are chosen at random from the set $[0..m-1]$, without replacement, or
- *Random probing:* the locations are chosen at random from the set $[0..m-1]$, with replacement.

0	1	2	3	4	5	6	7	8	9	10
B	E	G	F	A		I	D	C		H

FIGURE 2.3 Open addressing with double hashing.

For both models, it can be shown that

$$\begin{aligned} \mathbf{E}U_n &= \frac{1}{1-\alpha} + \Theta\left(\frac{1}{m}\right) \\ \mathbf{E}S_n &= \frac{1}{\alpha}\ln\frac{1}{1-\alpha} + \Theta\left(\frac{1}{m}\right) \end{aligned}$$

Again, for a full table the above expressions are useless, but we can prove that the search costs are $\Theta(m)$ and $\Theta(\log m)$, respectively.

Deletions cannot be done by simply erasing the given element, because searches would stop there and miss any element located beyond that point in its probe sequence. The solution of marking the location as "dead" (i.e., still occupied for the purposes of searching, but free for the purposes of insertion) works at the expense of deteriorating the search time.

An interesting property of collision resolution in *open addressing* hash tables is that when two keys collide (one incoming key and one that is already in the table), *either* of them may validly stay in that location, and the other one has to try its next probe location. The traditional insertion method does not use this degree of freedom, and simply assigns locations to the keys in a "first-come-first-served" (FCFS) fashion.

Several methods have been proposed, that make use of this flexibility to improve the performance of open addressing hash tables. Knuth and Amble [21] used it to resolve collisions in favor of the element with the smallest key, with the result that the table obtained is the same as if the keys had been inserted in increasing order. This implies that all keys encountered in a successful search are in increasing order, a fact that can be used to speed up unsuccessful searches.

If we restrict ourselves to methods that arbitrate collisions based only on the past history (i.e., no lookahead), it can be shown that the expected successful search cost does not depend on the rule used (assuming random probing). However, the *variance* of the successful search cost does depend on the method used, and can be decreased drastically with respect to that of the standard FCFS method.

A smaller variance is important because of at least two reasons. First, a method with a low variance becomes more predictable, and less subject to wide fluctuations in its response time. Second, and more important, the usual method of following the probe sequence sequentially may be improved by replacing it by an optimal search algorithm that probes the most likely location first, then the second most likely,

and so on. A reasonable approximation for this is a "mode-centered" search, that probes the most likely location first, and then moves away from it symmetrically.

Perhaps the simplest heuristic in this class is "last-come-first-served" (LCFS) [30], which does exactly the opposite from what the standard method does: in the case of a collision, the location is assigned to the incoming key.

For a full table (assuming random probing), the variance of the standard (FCFS) method is $\Theta(m)$. The LCFS heuristic reduces this to $\Theta(\log m)$.

Another heuristic, that is much more aggressive in trying to decrease inequalities between the search costs of individual elements is the "Robin Hood" (RH) method [9]. In the case of a collision, it awards the location to the element that has the largest retrieval cost. For a full table (assuming random probing), the variance of the cost of a successful search for Robin Hood hashing is ≤ 1.833, and using the optimal search strategy brings the expected retrieval cost down to ≤ 2.57 probes.

These variance reduction techniques can be applied also to linear probing. It can be shown [31] that, for a full table, both LCFS and RH decrease the variance from the $\Theta(m^{3/2})$ of the standard FCFS method to $\Theta(m)$. In the case of linear probing, it can be shown that, for any given set of keys, the Robin Hood arrangement minimizes the variance of the search time.

If we wish to decrease the expected search cost itself, and not just the variance, we must look ahead in the respective probe sequences of the keys involved in a collision. The simplest scheme would be to resolve the collision in favor of the key that would have to probe the most locations before finding an empty one. This idea can be applied recursively, and Brent [7] and Gonnet and Munro [15] used this to obtain methods that decreased the expected search cost to 2.4921 and to 2.13414 probes, respectively, for a full table.

Gonnet and Munro [15] considered also the possibility of moving keys *backward* in their probe sequences to find the *optimal* table arrangement for a given set of keys. This problem is mostly of theoretical interest, and there are actually two versions of it, depending on whether the average search cost or the maximum search cost are minimized. Simulation results show that the optimal average search cost for a full table is approximately 1.83 probes.

Choosing a Hash Function

The traditional choice of hash functions suffers from two problems. First, collisions are very likely to occur, and the method has to plan for them. Second, a malicious adversary, knowing the hash function, may generate a set of keys that will make the worst-case be $\Theta(n)$.

If the set of keys is known in advance, we may be able to find a *perfect hash function*, i.e., a hash function that produces no collisions for that set of keys. Many methods have been proposed for constructing perfect hash functions, beginning with the work of Fredman, Komlós and Szemerédi [14]. Mehlhorn [27] proved a matching upper and lower bound of $\Theta(n)$ bits for the program size of a perfect hash function.

Fox et al. [12, 13] provide algorithms for finding a *minimal* perfect hash function (i.e., for a full table), which runs in expected linear time on the number of keys involved. Their algorithms have been successfully used on sets sizes of the order of one million keys.

An approach to deal with the worst-case problem was introduced by Carter and Wegman [8]. They use a class of hash functions, and choose one function at random from the class for each run of the algorithm. In order for the method to work, the functions must be such that no pair of keys collide very often. Formally, a set $\mathcal{H}$ of hash functions is said to be *universal* if for each pair of distinct keys, the number of hash functions $h \in \mathcal{H}$ is exactly $|\mathcal{H}|/m$. This implies that for a randomly chosen h, the probability of a collision between x and y is $1/m$, the same as if h assigned truly random hash values for x and y. Cormen, Leiserson and Rivest [10] show that, if keys are composed of $r+1$ "bytes" $x_0, \ldots, x_r$, each less than m, and $a = \langle a_0, \ldots, a_r \rangle$ is a sequence of elements chosen at random from $[0..m-1]$, then the set of functions $h_a(x) = \Sigma_{0 \leq i \leq r} a_i x_i \bmod m$ is universal.

Hashing in Secondary Storage

All the hashing methods we have covered can be extended to secondary storage. In this setting, keys are usually stored in *buckets*, each holding a number of keys, and the hash function is used to select a bucket, not a particular key. Instead of the problem of collisions, we need to address the problem of bucket *overflow*. The analysis of the performance of these methods is notoriously harder than that of the main memory version, and few exact results exist [34].

However, for most practical applications, simply adapting the main memory methods is not enough, as they usually assume that the size of the hash table (m) is fixed in advance. Files need to be able to grow on demand, and also to shrink if we want to maintain an adequate memory utilization.

Several methods are known to implement *extendible* hashing (also called *dynamic* hash tables). The basic idea is to use an unbounded hash function $h(x) \geq 0$, but to use only its d rightmost bits, where d is chosen to keep overflow low or zero.

Fagin et al. [11] use the rightmost d bits from the hash function to access a *directory* of size 2^d, whose entries are pointers to the actual buckets holding the keys. Several directory entries may point to the same bucket.

Litwin [25] and Larson [23, 24] studied schemes that do not require a directory. Their methods work by gradually doubling the table size, scanning buckets from left to right. To do this, bucket splitting must be delayed until it is that bucket's turn to be split, and overflow records must be held temporarily using chaining or other similar method.

2.5 Related Searching Problems

Searching in an Unbounded Set

In most cases we search in a bounded set. We can also search in an *unbounded set*. Consider the following game: one person thinks about a positive number and another person has to guess it with questions of the type: *the number x is less than, equal to, or larger than the number that you are thinking?* This problem was considered in [4].

A first obvious solution is to use the sequence $1, 2, \ldots, n$ (that is, sequential search), using n questions. We can do better by using the "gambler" strategy. That is, we use the sequence $1, 2, 4, \ldots, 2^m$, until we have $2^m \geq n$. In the worst case we have $m = \lfloor \log n \rfloor + 1$. Next, we can use binary search in the interval $2^{m-1} + 1$ to 2^m to search for n, using in the worst case $m - 1$ questions. Hence, the total number of questions is $2m - 1 = 2\lfloor \log n \rfloor + 1$. This algorithm is depicted in Fig. 2.4. That is, only twice a binary search in a finite set of n elements. Can we do better? We can think that what we did is to search the exponent m using sequential search. So, we can use this algorithm, A_1, to search for m using $2\lfloor \log m \rfloor + 1$ questions, and then use binary search, with a total number of questions of $\log n + 2 \log \log n + O(1)$ questions. We could call this algorithm A_2.

In general, we can define algorithm A_k, which uses A_{k-1} to find m and then use binary search. The complexity of such algorithm is

$$S_n^k = \log n + \log \log n + \cdots + \log^{(k-1)} n + 2 \log^{(k)} n + O(1)$$

questions, where $\log^{(i)} n$ denotes log applied i times. Of course, if we could know the value of n in advance, there is an optimal value for k of $O(\log^* n)$,[3] because if k is too large, we go too far. However, we do not know n a priori!

[3] $\log^* n$ is the number of times that we have to apply the log function before we reach a value less than or equal to 0.

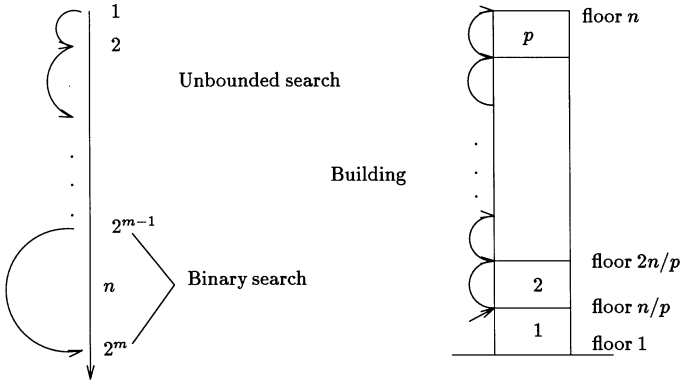


FIGURE 2.4 A_1 unbounded search (left) and the person problem (right).

Searching with Bounded Resources

Most of the time we assume that we can perform an unbounded number of questions when searching. However, in many real situations, we search with bounded resources. For example, gasoline when using a car. As an example, we use a variable cost searching problem, initially proposed in [5, Section 3.2], with some changes, but maintaining the same philosophy. Given a building of n floors and k persons, we want to answer the following problem: which is the largest floor from where a person can jump and does not break a leg? We assume that a person with a broken leg cannot jump again.[4]

Suppose that the answer is floor j. So, we have j floors that will give us a positive answer and $n - j$ floors that give us a negative answer. However, we can only afford up to k negative answers to solve the problem (in general $k < n - j$). So, we have a bounded resource: persons.

If we have just one person, the solution is easy, since we are forced to use sequential search to find j. Any other strategy does not work, because if the person fails, we do not solve the problem. If we have many persons (more precisely $k > \log n$), we can use binary search. In both cases, the solution is optimal in the worst case.

If we have two persons, a first solution would be to start using binary search with the first person, and then use the second sequentially in the remaining segment. In the worst case, the first person fails in the first jump, giving a $n/2$ jumps algorithm. The problem is that both persons do not perform the same amount of work. We can balance the work by using the following algorithm: the first person tries sequentially every n/p floors for a chosen p, that is n/p, $2n/p$, etc. When his/her leg breaks, the second person has a segment of approximately n/p floors to check (see Fig. 2.4). In the worst case the number of floors is p (first person) plus n/p (second person). So we have

$$U_n^2 = p + n/p + O(1)$$

Balancing the work, we have $p = n/p$, which implies $p = \sqrt{n}$, giving $U_n^2 = 2\sqrt{n} + O(1)$. Note that to succeed, any algorithm has to do sequential search in some segment with the last person.

We can generalize the above algorithm to k persons using the partitioning idea recursively. Every person except the last one partitions the remaining segment in p parts and the last person uses sequential search. In the worst case, every person (except the last one) has to perform p jumps. The last one does sequential search on a segment of size n/p^{k-1}. So, the total cost is approximately

$$U_n^k = (k-1)p + \frac{n}{p^{k-1}}$$

[4]This is a theoretical example, do not try to solve this problem in practice!

Balancing the work for every person, we must have $p = n/p^{k-1}$, obtaining $p = n^{1/k}$ (same as using calculus!). Then, the final cost is

$$U_n^k = kn^{1/k}$$

If we consider $k = \log_2 n$, we have

$$U_n^k = kn2^{\log_2(n^{1/k})} = \log_2 n 2^{\frac{\log_2 n}{k}} = 2\log_2 n$$

which is almost like binary search. In fact, taking care of the partition boundaries, and using an optimal partition (related to binomial trees), we can save k jumps, which gives the same as binary search. So we have a continuum from sequential to binary search as k grows.

We can mix the previous two cases to have unbounded search with limited resources. The solution mixes the two approaches already given and can be a nice exercise for interested readers.

Searching with Nonuniform Access Cost

In the traditional RAM model we assume that any memory access has the same cost. However, this is not true if we consider the memory hierarchy of a computer: registers, cache and main memory, secondary storage, etc. As an example of this case, we use the *hierarchical memory* model introduced in [1]. That is, the access cost to position x is given by a function $f(x)$. The traditional RAM model is when $f(x)$ is a constant function. Based in access times of current devices, possible values are $f(x) = \log x$ or $f(x) = x^\alpha$ with $0 < \alpha \leq 1$.

Given a set of n integers in a hierarchical memory, two problems are discussed. First, given a fixed order (sorted data), what is the optimal worst-case search algorithm. Second, what is the optimal ordering (implicit structure) of the data to minimize the worst-case search time. This ordering must be described using constant space.

In both cases, we want to have the n elements in n contiguous memory locations starting at some position and only using a constant amount of memory to describe the searching procedure. In our search problem, we consider only successful searches, with the probability of searching for each one of the n elements being the same.

Suppose that the elements are sorted. Let $S(i, j)$ be the optimal worst-case cost to search for an element which is between positions i and j of the memory. We can express the optimal worst-case cost as

$$S(i, j) = \min_{k=i,\ldots,j} \{f(k) + \max(S(i, k-1), S(k+1, j))\}$$

for $i \geq j$ or 0 otherwise. We are interested in $S(1, n)$. This recurrence can be solved using dynamic programming in $O(n^2)$ time. This problem was considered in [20], where it is shown that for logarithmic or polynomial $f(x)$, the optimal algorithm needs $O(f(n)\log n)$ comparisons. In particular, if $f(x) = x^\alpha$, a lower and upper bound of

$$\frac{n^\alpha \log n}{1+\alpha}$$

for the worst-case cost of searching is given in [20].

In our second problem, we can order the elements to minimize the searching cost. A first approach is to store the data as the implicit complete binary search tree induced by a binary search in the sorted data, such that the last level is compacted to the left (*left complete binary tree*). That is, we store the root of the tree in position 1 and in general the children of the element in position i in positions $2i$ and $2i+1$ like in a heap. Nevertheless, there are better addressing schemes that balance as much as possible every path of the search tree.

Searching with Partial Information

In this section we use a nonuniform cost model plus an unbounded domain. In addition the algorithm does not know all the information of the domain and learns about it while searching. In this case we are searching for an object in some space under the restriction that for each new "probe" we must pay costs proportional to the distance of the probe position relative to our current probe position and we wish to minimize this cost. This is meant to model the cost in real terms of a robot (or human) searching for an object when the mobile searcher must move about to find the object. It is also the case for many searching problems on secondary memory devices as disk and tapes. This another example of an **on-line algorithm.**

An on-line algorithm is called *c-competitive* if the solution to the problem related to the optimal solution when we have all the information at the beginning (off-line case) is bounded by

$$\frac{\text{Solution (on-line)}}{\text{Optimal (off-line)}} \leq c$$

Suppose that a person wants to find a bridge over a river. We can abstract this problem as finding some distinguished point on a line. Assume that the point is n (unknown) steps away along the line and that the person does not know how far away the point is. What is the minimum number of steps it must make to find the point, as a function of n?

The optimal way to find the point (up to lower order terms) is given by linear spiral search [3]: execute cycles of steps where the function determining the number of steps to walk before the i^{th} turn starting from the origin is 2^i for all $i \geq 1$. That is, we first walk one step to the left, we return to the origin, then two steps to the right, returning again to the origin, then four steps to the left, etc. The total distance walked is $2\Sigma_{i=1}^{\lfloor \log n \rfloor + 1} 2^i + n$, which is no more than 9 times the original distance. That is, this is a 9-competitive algorithm, and this constant cannot be improved.

2.6 Research Issues and Summary

Sequential and binary search are present in many forms in most programs, as they are basic tools for any data structure (either in main memory or in secondary storage). Hashing provides an efficient solution when we need good average search cost. We also covered several variants generalizing the model of computation (randomized, parallel, bounded resources) or the data model (unbounded, nonuniform access cost, and partial knowledge).

As for research issues, we list here some of the most important problems that still deserve more work. Regarding hashing, faster and practical algorithms to find perfect hashing functions are still needed. The hierarchical memory model has been extended to cover nonatomic accesses (that is, access by blocks) and other variations. This model still has several open problems. Searching with partial information led to research on more difficult problems such as motion planning.

2.7 Defining Terms

Amortized cost: Worst-case cost of a sequence of operations, averaged over the number of operations.

Chaining: A family of hashing algorithms that solve collisions by using pointers to link elements.

CREW PRAM: Computer model that has many processors sharing a memory where many can read at the same time, but only one can write at the same time in a given memory cell.

Hash function: Function that maps keys onto table locations, by performing arithmetic operations on the keys. Keys that hash to the same location are said to *collide*.

On-line algorithm: Algorithm that process the input sequentially.

Open addressing: A family of collision resolution strategies based on computing alternative hash locations for the colliding elements.

Randomized algorithm: Algorithm that makes some random (or pseudo-random) choices.

Self-organizing strategies: Heuristic that reorders a list of elements according to how the elements are accessed.

Universal hashing: A scheme that chooses randomly from a set of hash functions.

References

[1] Aggarwal, A., Alpern, B., Chandra, K., and Snir, M., A model for hierarchical memory. In *Proc. of the 19th Annual ACM Symp. of the Theory of Computing,* 305–314, New York, 1987.

[2] Baeza-Yates, R., Searching: An algorithmic tour. In *Encyclopedia of Computer Science and Technology,* Allen Kent and James G. Williams, Eds., 37, 331–359, Marcel Dekker, 1997.

[3] Baeza-Yates, R.A., Culberson, J., and Rawlins, G., Searching in the plane. *Information and Computation,* 106(2), 234–252, Oct 1993. Preliminary version titled "Searching with uncertainty" was presented at SWAT'88, Halmstad, Sweden, LNCS 318, 176–189.

[4] Bentley, J.L. and Yao, A.C-C., An almost optimal algorithm for unbounded searching. *Inf. Proc. Letters,* 5(3), 82–87, Aug. 1976.

[5] Bentley, J.L. and Brown, D.J., A general class of resource trade-offs. In *IEEE Foundations of Computer Science,* 21, 217–228, Syracuse, NY, Oct. 1980.

[6] Bentley, J.L. and McGeoch, C.C., Amortized analyses of self-organizing sequential search heuristics. *Communications of the ACM,* 28(4), 404–411, Apr. 1985.

[7] Brent, R.P., Reducing the retrieval time of scatter storage techniques. *Communications of the Association for Computing Machinery,* 16(2), 105–109, Feb. 1973.

[8] Carter, J.L. and Wegman, M.N., Universal classes of hash functions. *Journal of Computer and Systems Sciences,* 18(2), 135–154, Apr. 1979.

[9] Celis, P., Larson, P.-Å., and Munro, J.I., Robin Hood hashing. In *26th Annual Symposium on Foundations of Computer Science,* 281–288, Portland, OR, 1985.

[10] Cormen, T.H., Leiserson, C.E., and Rivest, R.L., *The Design and Analysis of Computer Algorithms,* Cambridge, MA, MIT Press, 1990.

[11] Fagin, R., Nievergelt, J., Pippenger, N., and Strong, H.R., Extendible hashing: A fast access method for dynamic files. *ACM Transactions on Database Systems,* 4(3), 315–344, Sep. 1979.

[12] Fox, E.A., Qi, F.C., Daoud, A.M., and Heath, L.S., Order-preserving minimal perfect hash functions and information retrieval. *ACM Transactions on Information Systems,* 9(3), 281–308, 1991.

[13] Fox, E.A., Heath, L.S., Chen, Q.F., and Daoud, A.M., Minimal perfect hash functions for large databases. *Communications of the Association for Computing Machinery,* 35(1), 105–121, Jan. 1992.

[14] Fredman, M.L., Komlós, J., and Szemerédi, E., Storing a sparse table with $O(1)$ worst case access time. *Journal of the Association for Computing Machinery,* 31(3), 538–544, Jul. 1984.

[15] Gonnet, G.H. and Munro, J.I., Efficient ordering of hash tables. *SIAM Journal on Computing,* 8(3), 463–478, Aug. 1979.

[16] Gonnet, G.H., Expected length of the longest probe sequence in hash code searching. *Journal of the Association for Computing Machinery,* 28(2), 289–304, Apr. 1981.

[17] Gonnet, G.H. and Baeza-Yates, R., *Handbook of Algorithms and Data Structures: in Pascal and C.* Addison-Wesley, Reading, MA, 2nd ed., 1991.

[18] Guibas, L.J. and Szemerédi, E., The analysis of double hashing. *Journal of Computer and System Sciences,* 16(2), 226–274, Apr. 1978.

[19] Hester, J.H. and Hirschberg, D.S., Self-organizing linear search. *ACM C. Surveys,* 17(3), 295–311, Sep. 1985.
[20] Knight, W.J., Search in a ordered array having variable probe cost. *SIAM J. of Computing,* 17(6), 1203–1214, Dec. 1988.
[21] Knuth, D.E. and Amble, O., Ordered hash tables. *The Computer Journal,* 17(5), 135–142, May 1974.
[22] Knuth, D.E., *The Art of Computer Programming, Sorting and Searching.* Addison-Wesley, Reading, MA, 2nd ed., 1975.
[23] Larson, P.-Å., Dynamic hashing. *BIT,* 18(2), 184–201, 1978.
[24] Larson, P.-Å., Performance analysis of linear hashing with partial expansions. *ACM Transactions on Database Systems,* 7(4), 566–587, Dec. 1982.
[25] Litwin, W., Virtual hashing: A dynamically changing hashing. In *Fourth International Conference on Very Large Data Bases,* Yao, S.B., Ed., 517–523, West Berlin, Germany, ACM Press, 1978.
[26] Lueker, G.S. and Molodowitch, M., More analysis of double hashing. *Combinatorica,* 13(1), 83–96, 1993.
[27] Mehlhorn, K., On the program size of perfect and universal hash functions. In *23rd Annual Symposium on Foundations of Computer Science,* 170–175, Chicago, IL, 1982.
[28] Perl, Y. and Reingold, E.M., Understanding the complexity for interpolation search. *Inf. Proc. Letters,* 6(6), 219–221, Dec. 1977.
[29] Perl, Y., Itai, A., and Avni, H., Interpolation search—a $\log \log n$ search. *C.ACM,* 21(7), 550–553, Jul. 1978.
[30] Poblete, P.V. and Munro, J.I., Last-come-first-served hashing. *Journal of Algorithms,* 10(2), 228–248, Jun. 1989.
[31] Poblete, P.V., Viola, A., and Munro, J.I., The analysis of a hashing scheme by the diagonal Poisson transform. *LNCS,* 855, 94–105, 1994.
[32] Siegel, A., On the statistical dependencies of coalesced hashing and their implications for both full and limited independence. In *6th ACM-SIAM Symposium on Discrete Algorithms (SODA),* 10–19, San Francisco, CA, 1995.
[33] Sleator, D.D. and Tarjan, R.E., Amortized efficiency of list update and paging rules. *Communications of the ACM,* 28(2), 202–208, Feb. 1985.
[34] Viola, A. and Poblete, P.V., The analysis of linear probing hashing with buckets. In *Fourth Annual European Symposium on Algorithms - ESA'96,* Diaz, J. and Serna, M., Eds., 221–233, Springer-Verlag, Barcelona, Sep. 1996.
[35] Vitter, J.S., *Analysis of Coalescing Hashing.* Ph.D. Thesis, Stanford University, Stanford, CA, Oct. 1980.

Further Information

Additional algorithms and references for the first four sections can be found in [17] and in many algorithms textbooks. More information on self-organizing heuristics can be found on Hester and Hirschberg's survey [19]. The amortized-case analysis is presented in [6, 33]. More information on searching with partial information is given in [3]. More information on searching nonatomic objects is covered in [2].

Many chapters of this handbook extend the material presented here. Another example of amortized analysis of algorithms is given in Chapter 1. Analysis of the average case is covered in Chapter 14. Randomized, on-line, and parallel algorithms are covered in Chapters 15, 10, and 47, respectively. Searching and updating more complex data structures is explained in Chapters 4 and 5. Searching for strings and subtrees is covered in Chapters 11 and 13, respectively. The use of hashing functions in cryptography is covered in Chapter 8.

3

Sorting and Order Statistics

Vladimir Estivill-Castro
University of Newcastle

3.1 Introduction

Sorting is the computational process of rearranging a given sequence of items from some total order into ascending or descending order. Because sorting is a task in the very core of Computer Science, efficient algorithms were developed early. The first practical and industrial applications of computers had many uses for sorting. It is still a very frequently occurring problem, often appearing as a preliminary step to some other computational tasks. A related application to sorting is computing *order statistics,* for example, finding the median, the smallest or the largest of a set of items. Although finding order statistics is immediate once the items are sorted, sorting can be avoided and faster algorithms have been designed for finding the kth largest element, the most practical of which is derived from the structure of a sorting method.

Sorting usually involves data consisting of records in one or several files. One or several fields of the records are used as the criteria for sorting (often a small part of the record) and are called the *key.* Usually, the objective of the sorting method is to rearrange the records so that their keys are arranged in numerical or alphabetical order.

In many applications of sorting, elementary sorting algorithms are the best alternative. There are several reasons for this. Sorting programs are often used once (or only a few times) rather than repeated many times, one after another. Simple methods are always suitable for small files, say less than a 100 elements. The increasing speeds in every time less expensive computers are enlarging the size for which basic methods are adequate. More advanced algorithms require more careful programming, and their correctness or efficiency is more fragile to thorough understanding of their mechanisms. Also, sophisticated methods may not take advantage of existing order in the input, which may already be sorted, while elementary methods usually do. Finally, elementary sorting algorithms usually have a very desirable property, named *stability;* that is, they preserve the relative order of items with equal keys. This is usually expected in applications generating reports from already sorted files, but with a different key. For example, long distance phone calls are usually recorded in a log in chronological order by the date and time the call

was made. When reporting bills to customers, the carrier sorts by customer name, but the result should preserve the chronological order of the calls made by each particular customer.

Advanced methods are the choice in applications involving a large number of items. Also, they can be used to build a robust, general-purpose sorting routine. Elementary methods should not be used for large, randomly permuted files. An illustrative trade-off between a sophisticated technique that results in generally better performance over the simplicity of the programming is the sorting of keys and pointers to records rather than the entire records. Once the keys are sorted, the pointers to the complete records are used in a simple pass over the file to rearrange the data in the desired order. To apply this technique we must construct the auxiliary sequence of keys and pointers (or in the case of long keys, the keys can be kept with the records as well). However, many exchanges and data moves are saved until the final destination of each record is known. In addition, less space is required if, as is common in practice, keys are only a small part of the data record. When keys are a quarter or more of their records, this technique is not worth the programming effort.

When the file to be sorted is small enough that all the data fits in main memory (usually an array of records or keys), the sorting process is called **internal sorting**. **External sorting** corresponds to the situation when the file to be sorted is so large that the data does not entirely fit in main memory; then, the sorting process must be performed with clever use of input/output operations, since these operations, account for most of the computational time. Currently, the sorting of files that are too large to be held in main memory is performed on disk drives, rather than tapes. The availability of large main memories and new technologies for disk drives have modified the models for external sorting [17, 20].

Sorting algorithms can also be divided into two large groups according to what they require about the data to perform the sorting. The first group is usually called *comparison-based.* Methods of this class only use the fact that the universe of keys is linearly ordered. That is, given two items, they only need to query what is the order between these two items. Because of this property, the implementation of **comparison-based algorithms** can be generic with respect to the data type of the keys, and a comparison routine can be supplied as a parameter to the sorting procedure. The second group of algorithms assumes further that keys are restricted to a certain domain or representation and use knowledge of this information to dissect subparts, bytes, or bits of the keys.

Sorting is also ideal for introducing issues regarding algorithmic complexity. For comparison-based algorithms, it is possible to precisely define an abstract model of computation (namely, *decision trees*) and show lower bounds on the number of comparisons any sorting method would require to sort a sequence with n items (in the worst case and in the average case). A comparison-based sorting algorithm that requires $O(n \log n)$ comparisons is said to be *optimal,* because this matches the $\Omega(n \log n)$ lower bounds. Thus, in theory, no other algorithm could be faster. The fact that algorithms that are not comparison-based can result in faster implementations illustrates the relevance of the model of computation with respect to theoretical claims of optimality and the effect that a stronger assumption on the data has for designing a faster algorithm [1, 14].

Sorting illustrates *randomization* (the fact that the algorithm makes a random choice). Randomization has practical value for sorting algorithms. In particular, it provides an easy protection for sophisticated algorithms from special input that may be simple (like almost sorted) but harmful to the efficiency of the method. The most notable example is the use of randomization for the selection of the pivot in *Quicksort.*

In what follows, we will assume that the goal is to sort into ascending order, since sorting in descending order is symmetrical, or can be achieved by sorting in ascending order and reversing the result (in linear time, which is usually affordable). Also, we make no distinction between the records to be sorted and their keys, assuming that some provision has been made for handling this as suggested before. Through our presentation the keys may not all be different, since one of the main applications of sorting is to bring together records with matching keys. We will present algorithms in pseudocode in the style introduced by Cormen et al. [5], or when more specific detail is convenient, we will use PASCAL code. When appropriate, we will indicate possible trade-offs between clarity and efficiency of the code. We believe that efficiency should not be pursued at the extreme, and certainly not above clarity. The costs of programming and

code maintenance are usually larger than slight efficiency gains by tricky coding. For example, there is a conceptually simple remedy to make every sorting routine **stable**. The idea is to precede it with the construction of new keys and sort according to lexicographical order of the new keys. The new key for the ith item is the pair (k_i, i), where k_i is the original sorting key. This requires the extra management of the composed keys and adds to the programming effort the risk of a faulty implementation. Today, this could probably be solved by simply choosing a competitive stable sort, perhaps at the expense of slightly more space or time.

3.2 Underlying Principles

Divide-and-conquer is a natural, top-down approach for the design of an algorithm for the abstract problem of sorting a sequence $X = \langle x_1, x_2, \ldots, x_n \rangle$ of n items. It consists of dividing the problem into smaller subproblems hoping that the solutions of the subproblems are easier to find and then composing the partial solutions into the solution of the original problem. A prototype of this idea is *Mergesort;* where the input sequence $X = \langle x_1, x_2, \ldots, x_n \rangle$ is split into two sequences $X_L = \langle x_1, x_2, \ldots, x_{\lfloor n/2 \rfloor} \rangle$ the left subsequence) and $X_R = \langle x_{\lfloor n/2 \rfloor + 1}, \ldots, x_n \rangle$ (the right subsequence). Finding solutions recursively for sequences of more than one item and terminating the recursion with sequences with only one item (since these are always sorted) provides a solution for the two subproblems. The overall solution is found by describing a method to merge two sorted sequences. In fact, internal sorting algorithms are variations of two forms of using divide-and-conquer:

Conquer form. Divide is simple (usually requiring constant time), conquer is sophisticated.

Divide form. Divide is sophisticated, conquer is simple (usually requiring constant time).

Again, *Mergesort* is an illustration of the conquer from. The core of the method is in the merging of the two sorted sequences, hence its name.

The prototype for the **divide form** is *Quicksort* [10]. Here, one item x_p (called the *pivot*) is selected from the input sequence X and its key is used to create two subproblems $X_{\leq}$ and $X_{\geq}$, where $X_{\leq}$ contains items in X with keys less or equal than the pivot's, while items in $X_{\geq}$ have keys larger or equal than the pivot's. Now, recursively applying the algorithm for $X_{\leq}$ and $X_{\geq}$ results in a global solution (with a trivial conquer step that places $X_{\leq}$ before $X_{\geq}$).

Sometimes we may want to conceptually simplify the **conquer form** by not dividing into two problems of roughly the same size, but rather divide X into $X' = \langle x_1, \ldots, x_{n-1} \rangle$ and $X'' = \langle x_n \rangle$. This is conceptually simpler in two ways. First, X'' is a trivial subproblem, because it has only one item and thus it is already sorted. Therefore, in a sense we have one subproblem less. Second, the merging of the solutions of X' and X'' is simpler than the merging of two sequences of almost equal length, because we just need to place x_n in its proper position amongst the sorted items from X'. Because the method is based upon inserting into an already sorted sequence, sorting algorithms with this idea are called **insertion sorts**.

Insertion sorts vary according to how the sorted list for the solution of X' is represented by a data structure that supports insertions. The simplest alternative is to have the sorted list stored in an array, and this method is named *Insertion Sort* or *Straight Insertion Sort* [12]. However, if the sorted list is represented by a *level-linked-tree* with a *finger,* the method has been named *A-sort* [14] or *Local Insertion Sort* [13].

The complementary simplification in the divide form makes one subproblem trivial by selecting a pivot so $X_{\leq}$ or $X_{\geq}$ consist of just one item. This is achieved if we select the pivot as the item with the smallest or with the largest key in the input. The algorithms under this scheme are called **selection sorts**, and they vary according to the data structured used to represent X so that repeated extraction of the maximum (minimum) key is efficient. This is typically the requirement of the *priority queue* abstract data type with the keys as the priorities. When the priority queue is implemented as an array and the smallest key is found

by scanning this array, the method is *Selection Sort.* However, if the priority queue is an array organized into a heap, then the method is called *Heapsort.*

Using divide-and-conquer does not necessarily mean that the division must be into two subproblems. It may divide into several subproblems. For example, if the keys can be manipulated with other operations besides comparisons, *Bucket Sort* uses an interpolation formula on the keys to partition the records between m buckets. The buckets are sets of records, which are usually implemented as *queues.* The queues are usually implemented as linked lists that allow insertion and removal in constant time in first-in first-out order, as the abstract data type queue requires. The buckets represent subproblems to be sorted recursively. Finally, all the buckets are concatenated together. *Shellsort* divides the problem of sorting X into several problems consisting of the interlaced subsequences of X that consist of items d positions apart. Thus, the first subproblem is $\langle x_1, x_{1+d}, x_{1+2d}, \ldots \rangle$ while the second subproblem is $\langle x_2, x_{2+d}, x_{2+2d}, \ldots \rangle$. *Shellsort* solves the subproblems by applying *Insertion Sort;* however, rather than using a **multiway merge** to combine these solutions, it reapplies itself to the whole input but with a smaller d. Careful selection of the sequence of values of d results in a very practical sorting method.

We have used divide-and-conquer to conceptually depict the landscape of sorting algorithms (refer to Fig. 3.1). Nevertheless, in practice, sorting algorithms are usually not implemented as recursive programs. Instead, a nonrecursive equivalent analog is implemented (although computations may be performed in different order). In applications, the nonrecursive version is more efficient since the administration of the recursion is avoided. For example, a straightforward nonrecursive version of *Mergesort* proceeds as follows. First, the pairs of lists $\langle x_{2i-1} \rangle$ and $\langle x_{2i} \rangle$ (for $i = 1, \ldots \lfloor n/2 \rfloor$) are merged to form sorted lists of length two. Next, the pairs of lists $\langle x_{4i-1}, x_{4i-2} \rangle$ and $\langle x_{4i-1}, x_{4i} \rangle$ (for $i = 1, \ldots \lfloor n/4 \rfloor$) are merged to form sorted lists of length four. The process builds sorted lists twice as long in each round until the input is sorted. Similarly, *Insertion Sort* has a practical iterative version illustrated in Fig. 3.2.

	Internal Sorting Algorithms	
	Conquer from	Divide form
comparison based	*Mergesort* *Insertion Sort*	*Quicksort* *Selection sort* *Heapsort* *Shellsort*
restricted universe		*Bucket Sort* *Radix Sorts*

FIGURE 3.1 The landscape of internal sorting algorithms.

```
INSERTION SORT(X, n);
1 X[0] ← −∞;
2 for j ← 2 to n
3     do i ← j − 1;
4        t ← X[j];
5        while t < X[i]
6              do X[i + 1] ← X[i];
7                 i ← i − 1;
8        X[i + 1] ← t; ▷ part of the for j loop
```

FIGURE 3.2 The sentinel version of *insertion sort.*

Placing a sorting method in the landscape provided by divide-and-conquer allows easy computation of its time requirements, at least under the O notation. For example, algorithms of the conquer form have a divide part that takes $O(1)$ time to derive two subproblems of roughly equal size. The solutions of subproblems may be combined in $O(n)$ time. This results in the following recurrence for the time $T(n)$

to solve a problem of size n:

$$T(n) = \begin{cases} 2T(\lceil n/2 \rceil) + O(n) + O(1) & \text{if } n > 1\,, \\ O(1) & n = 1\,. \end{cases} \tag{3.1}$$

It is not hard to see that each level of recursion takes linear time and that there are at most $O(\log n)$ levels (since n is roughly divided by 2 at each level). This results in $T(n) = O(n \log n)$ time overall. If the divide form splits into one problem of size $n-1$ and one trivial problem, the recurrence is as follows:

$$T(n) = \begin{cases} 2T(n-1) + O(1) + O(conquer(n-1)) & \text{if } n > 1\,, \\ O(1) & n = 1\,, \end{cases} \tag{3.2}$$

where $conquer(n-1)$ is the time required for the conquer step. It is not hard to see that there are $O(n)$ levels of recursion (since n is decremented by one at each level). Thus, the solution to the recurrence is $T(n) = O(n) + \Sigma_{i=1}^{n} O(conquer(i))$. In the case of *Insertion Sort,* the worst case for $conquer(i)$ is $O(i)$, for $i = 1, \ldots, n$. Thus, we have that *Insertion Sort* is $O(n^2)$. However, *Local Insertion Sort* assures $conquer(i) = O(\log i)$ and results in an algorithm that requires $O(n \log n)$ time. We will not purse this analysis any further, confident that the reader will be able to find enough information here or in the references to identify the time and space complexity of the algorithms presented, at least up to the O notation.

Naturally, one may ask why there are so many sorting algorithms if they all solve the same problem and fit a general framework. It turns out that, when implemented, each has different properties that makes them more suitable for different objectives.

First, comparison-based sorting algorithms are ranked by their theoretical performance in the comparison-based model of computation. Thus, an $O(n \log n)$ algorithm should always be preferred over an $O(n^2)$ algorithm if files are large. However, theoretical bounds may be for the worst case, or the expected case (where the analysis assumes that the keys are pairwise different and all possible permutations of items are equally likely). Thus, an $O(n^2)$ algorithm should be preferred over an $O(n \log n)$ algorithm if the file is small, or if we know that the file is already almost sorted. Particularly, if in such a case, the $O(n^2)$ algorithm turns into an $O(n)$ algorithm. For example, *Insertion Sort* requires exactly $Inv(X) + n - 1$ comparisons and $Inv(X) + 2n - 1$ data moves, where $Inv(X)$ is the number of *inversions* in a sequence $X = \langle x_1, x_2, \ldots, x_n \rangle$; that is, the number of pairs (i, j) where $i < j$ and $x_i > x_j$. On the other hand, if *Quicksort* is not carefully implemented, it may degenerate to $\Omega(n^2)$ performance for nearly sorted inputs.

If the theoretical complexities are equivalent, other aspects come into play. Naturally, the next criteria is the size of the constant hidden under the O notation (as well as the size of hidden minor terms when the file is small). These constants are affected by implementation aspects. The most significant are now listed.

- The relative costs of swaps, comparisons, and other operations in the computer at hand (and for the data types of the keys and records). Usually, swaps are more costly than comparisons, which in turn are more costly than other arithmetic operations; however, comparisons may be just as costly as swaps or an order of magnitude less costly, depending on the length of keys, records, and strategies to rearrange the records.
- The length of the machine code, so code remains in memory, under an operating system that administers paging or in the cache of the microprocessor.
- Similarly, the locality of references to data or the capacity to place frequently compared keys in a CPU register.

Finally, there may be restrictions that force the choice of one sorting method over another. These include limitations like the data structure holding the data may be a linked list instead of an array, or the space available may be seriously restricted. There may be need for stability, or the programming tool may lack recursion. In practice, a hybrid sort is usually the best answer.

3.3 State of the Art and Best Practices

Comparison-Based Internal Sorting

Insertion Sort

Figure 3.2 presented *Insertion Sort.* This algorithm uses sequential search to find the location, one item at a time, in a portion already sorted of the input array. It is mainly used to sort small arrays.

Besides being one of the simplest sorting algorithms, which results in simple code, it has many desirable properties. From the programming point of view, its loop is very short (usually taking advantage of memory management in the CPU cache or main memory), the key of the inserted element may be placed in a CPU register and access to data exhibits as much locality as it is perhaps possible. Also, if a minimum possible key value is known, a *sentinel* can be placed at the beginning of the array to simplify the inner loop, resulting in faster execution. Another alternative is to place the item being inserted, itself as a sentinel each time. From the applicability point of view, it is stable; recall this means records with equal keys remain in the same relative order after the sort. Its $\Theta(n^2)$ expected-case complexity and worst-case behavior makes it only suitable for small files, and thus, it is usually applied in *Shellsort* to sort the interleaved sequences. However, the fact that it is **adaptive** with respect to the measure of disorder Inv makes it suitable for almost sorted files with respect to this measure. Thus, it is commonly used to sort roughly sorted data produced by implementations of *Quicksort* that do not follow recursion calls once the subarray is small. This idea helps *Quicksort* implementations achieve better performance, since the administration of recursive calls for small files is more time consuming than using *Insertion Sort* to complete the sorting.

Insertion Sort also requires only constant space; that is, space for a few local variables (refer to Fig. 3.2). From the point of view of quadratic sorting algorithms, it is a clear winner. Investigations of theoretical interest have looked at comparison-based algorithms where space requirements are constant, and data moves are linear. Only in this case, *Insertion Sort* is not the answer, since *Selection Sort* achieves this with the equivalent number of comparisons. Thus, when records are very large and no provision is taken for avoiding the expensive data moves (by sorting a set of indices rather than the data directly), *Selection Sort* should be used.

Shellsort

One idea toward improving the performance of *Insertion Sort* is to observe than each element, when inserted into the sorted portion of the array, travels a distance equal to the number of elements to its left which are greater than itself (the number of elements inverted with it). However, this traveling is done in steps of just adjacent elements and not by exchanges between elements far apart. The idea behind *Shellsort* [12] is to use *Insertion Sort* to sort interleaved sequences formed by items d positions apart, thus allowing for exchanges as far as d positions apart. After this, elements far apart are closer to their final destinations, so d is reduced to allow exchanges of closer positions. To assure the output is sorted, the final value for d is one.

There are many proposals for the *increment sequence* [9]; the sequence of values of d. Some proposals are $\langle 2^k - 1, 2^{k-1} - 1, \ldots, 7, 3, 1\rangle$, $\langle 2^p 3^q, \ldots, 9, 8, 6, 4, 3, 2, 1\rangle$, and $\langle S_i, \ldots, 40, 13, 4, 1\rangle$ where $S_i = 3S_{i-1} + 1$. It is possible that better sequences exist; however, the improvement that they may produce in practice is almost not visible. *Shellsort* is guaranteed to be clearly below the quadratic behavior of *Insertion Sort.* The exact theoretical complexity remains elusive but large experiments conjecture $O(n(\log n)^2)$, $O(n^{1.25})$, and $O(n \log n \log\log n)$ comparisons are required for various increment sequences. Thus, it will certainly be much faster than quadratic algorithms, and for medium-size files it would remain competitive with $O(n \log n)$ algorithms.

From the programming point of view, *Shellsort* is simple to program. It is *Insertion Sort* inside the loop for the increment sequence. It is important not to use the version of *Insertion Sort* with sentinels, since for the rounds with large d many sentinels would be required. Not using a sentinel demands two exit points for the most inner loop. This can be handled with a clean use of a **goto**, but in languages which short-cut

logical connectives (typically C) this feature can be used to avoid gotos. Figure 3.3 shows pseudocode for *Shellsort.*

```
SHELLSORT(X, n);
1  d ← n ;
2  repeat
3        if d < 5
4        then d ← 1
5        else d ← (5 * d − 1) div 11;
6        for j ← d + 1 to n
7        do i ← j − d;
8            t ← X[j];
9            while t < X[i]
10           do X[i + d] ← X[i];
11               i ← i − d;
12               if i < d
13               then goto 14;
14           X[i + d] ← t; ▷ part of the for j loop
15 until d ≤ 1;
```

FIGURE 3.3 *Shellsort* with increment sequence $\langle \lfloor n\alpha \rfloor, \lfloor \lfloor n\alpha \rfloor \alpha \rfloor, \ldots, \rangle$ with $\alpha = 0.4545 < 5/11$.

Unfortunately, *Shellsort* loses some of the virtues of *Insertion Sort.* It is no longer stable, and its behavior in nearly sorted files is adaptive but not as marked as for *Insertion Sort.* However, its space requirements remain constant, and its coding is straightforward and usually results in a short program loop. It does not have a bad case and is a good candidate for a library sorting routine. The usual recommendation when facing a sorting problem is to first try out *Shellsort* because a correct implementation is easy to achieve. Only if this proves to be insufficient for the application at hand should a more sophisticated method be attempted.

Heapsort

The *priority queue* abstract data type is an object that allows the storage of items with a key indicating their priority as well as retrieval of the item with the largest key. Given a data structure for this abstract data type, a sorting method can be constructed as follows. Insert each data item in the priority queue with the sorting key as the priority key. Repeatedly extract the item with the largest key in the priority queue to obtain the items sorted in reverse order.

One immediate implementation of the priority queue is as an unsorted list (either as an array or as a linked list). Insertion in the priority queue is trivial; the item is just appended to the list. Extraction of the item with the largest key is achieved by scanning the list to find the largest item. If this implementation of a priority queue is used for sorting, the algorithm is called *Selection Sort.* Its time complexity is $\Theta(n^2)$, and as was already mentioned, its main virtue is that data moves are minimal.

A second implementation of a priority queue is to keep a list of the items sorted in descending order by their priorities. Now, extraction of the item with largest priority requires constant time, since it is known that the largest item is at the front of the list. However, inserting a new item into the priority queue implies scanning the sorted list for the position of the new item. Using this implementation of a priority queue for sorting we observe that we obtain *Insertion Sort* once more.

The above implementations of a queue offer constant time for either insertion or extraction of the item with maximum key in exchange for linear time for the other operation. Thus, in applications of priority queues where the balance of operations is uneven, they may result in efficient methods. However, for sorting, n items are inserted and n are extracted; thus, a balance is required between the insert and extract

operations. This is achieved by implementing the priority queue as a *heap* [1, 12] that shares the space with the array of data to be sorted. An array $A[1..n]$ satisfies the heap property if $A[\lfloor k/2 \rfloor] \geq A[k]$, for $2 \leq k \leq n$. In this case, $A[\lfloor k/2 \rfloor]$ is called the *parent* of $A[k]$ while $A[2k]$ and $A[2k+1]$ are called the *children* of $A[k]$. However, an item may have no children or only one child, in which case its called a *leaf.* The heap is constructed using all the elements in the array and is located in the lower part of the array. The sorted array is incrementally constructed from the item with largest key towards the element with smallest key. *Heapsort* operates in two phases. The first phase builds the heap using all the elements, and careful programming guarantees that this requires $O(n)$ time. The second phase repeatedly extracts the item with largest key from the heap. Since the heap shrinks by one element, the space created is used to place the element just extracted. Each of the n updates in the heap takes $O(\log i)$ comparisons, where i is the number of items currently in the heap. In fact, the second phase of *Heapsort* exchanges the first item of the array (the item with largest key) with the item in the last position of the heap, and sinks the new item at the top of the heap to reestablish the heap property.

Heapsort can be efficiently implemented around a procedure *sink* for repairing the heap property (refer to Fig. 3.4). Procedure *sink*$(k, limit)$ moves down the heap, if necessary, exchanging the item at position k with the largest of its two children, and stopping when the item at position k is no longer smaller than one of its children (or when $k > limit$); refer to Fig. 3.5. Observe that the loop in *sink* has two distinct exits, when item k has no children and when the heap property is reestablished. For our pseudocode, we have decided to avoid the use of gotos. However, the reader can refer to our use of a goto in Fig. 3.3 for an idea to construct an implementation that actually saves some data moves. Using procedure *sink,* the code for *Heapsort* is simple. From the applicability point of view, *Heapsort* has the disadvantage that it is not stable. However, it is guaranteed to execute in $O(n \log n)$ time in the worst case and no extra space.

```
HEAPSORT(X, n);
1 for i ← n div 2 downto 2
2 do SINK(i, n)
3 for i ← n downto 2
4 do SINK(1, i)
5      t ← X[1]
6      X[1] ← X[i]
7      X[i] ← t
```

FIGURE 3.4 The pseudocode for *Heapsort.*

```
SINK(k, limit);
1 while 2 * k ≤ limit
2 do j ← 2k
3    if j < limit
4    then ▷ two children
5           if X[j] < X[j + 1]
6           then j ← j + 1
7    if X[k] < X[j]
8    then t ← X[j]
9           X[j] ← X[k]
10          X[k] ← t
11          k ← j
12   else k ← limit + 1 ▷ force loop exit
```

FIGURE 3.5 The sinking of one item into a heap.

It is worth revising the analysis of *Heapsort.* First, let us look at *sink.* In each pass around its loop, *sink* at least doubles the value of k, and *sink* terminates when k reaches $limit$ (or before, if the heap property is reestablished earlier). Thus, in the worst case $sink$ requires $O(h)$ time where h is the height of the heap. The loop in lines 1 and 2 for *Heapsort* constitute the core of the first phase (in our code, the head construction is completed after the first execution of line 4). A call to *sink* is made for each node. The first $\lfloor n/2 \rfloor$ calls to *sink* are for heaps of height 1, the next $\lfloor n/4 \rfloor$ are for heaps of height 2, and so on. Summing over the heights, we have that the phase for building the heap requires

$$O\left(\sum_{i=1}^{\log n} i\,\frac{n}{2^i}\right) = O(n) \tag{3.3}$$

time. Now, the core of the second phase is the loop from line 3. These are n sinks plus a constant for the three assignments. The ith of the sinks is in a heap of height $O(\log i)$. Thus, this is $O(n \log n)$ time. We conclude that the first phase of *Heapsort* (building the priority queue) requires $O(n)$ time. This is useful when building a priority queue. The second phase, and, thus, the algorithm, requires $O(n \log n)$ time.

Heapsort does not use any extra storage, nor does it require a language supplying recursion. For some, it may be surprising that *Heapsort* destroys the order in an already sorted array to re-sort it. Thus, *Heapsort* does not take advantage of existing order in the input, but it compensates this with the fact that its running time has very small variance across the universe of permutations. Intuitively, items at the leaves of the heap have small keys, which makes the sinking usually travel down to a leaf. Thus, almost all of the n updates in the heap takes at least $\Omega(\log i)$ comparisons, making the number of comparisons vary very little from one input permutation to another. Although its average case performance may not be as good as *Quicksort,* it is rather simple to obtain an implementation that is robust. It is a very good choice for an internal sorting algorithm. Sorting by selection with an array having a heap property is also used for external sorting.

Quicksort

For many applications a more realistic measure of the time complexity of an algorithm is its expected time. In sorting, a classical example is *Quicksort* [10, 18], which has an optimal expected time complexity of $O(n \log n)$ under the decision tree model, while there are sequences that force it to perform $\Omega(n^2)$ operations (in other words, its worse-case time complexity is quadratic). If the worst-case sequences are very rare, or the algorithm exhibits a small variance around its expected case, then this type of algorithm is suitable in practice.

Several factors have made *Quicksort* a very popular choice for implementing a sorting routine. *Quicksort* is a simple divide-and-conquer concept, the partitioning can be done in a very short loop and is also conceptually simple, its memory requirements can be guaranteed to be only logarithmic on the size of the input, the pivot can be placed in a register and, most importantly, the expected number of comparisons is almost half of the worst-case optimal competitors, most notably *Heapsort.* In their presentation, many introductory courses on algorithms favor *Quicksort.* However, it is very easy to implement *Quicksort* in such a way that it seems correct and extremely efficient for many sorting situations. However, it may be hiding $O(n^2)$ behavior for a simple case (for example, sorting n equal keys). Users of such library routine will be satisfied with it initially, only to find out later that on something that seems a simple sorting task, the implementation is consuming too much time to finish the sort.

Fine tuning of *Quicksort* is a delicate issue [3]. Many of the improvements proposed may be compensated by reduced applicability of the method or more fragile and less clear code. Although the partitioning is conceptually simple, much care is required to avoid common pitfalls. Among these, we must assure that the selection and placement of the pivot maintains the assumption about the distribution of input sequences. That is, the partitioning must guarantee that all permutations of smaller sequences are equally likely when permutations of n items are equally likely. The partitioning must also handle extreme cases, which occur far more frequently in practice than the uniformity assumption for the theoretical analysis. These extreme

cases include the case in which the file is already sorted (either in ascending or descending order) and its subcase, the case in which the keys are all equal, as well as the case in which many keys replicated. One very common application of sorting is bringing together items with equal keys [3].

Recall that *Quicksort* is a prototype of divide-and-conquer with the core of the work performed during the divide phase. The standard *Quicksort* algorithm selects from a fixed location in the array a splitting element or *pivot* to partition a sequence in two parts. After partitioning, the items of the subparts are in correct order with respect to each other. Most partition schemes result in *Quicksort* not being stable. Figure 3.6 presents a version for partitioning that is correct and assures $O(n \log n)$ performance even if all keys are equal; it does not require sentinels and the indices i and j never go out of bounds from the subarray. The drawback of using fixed location pivots for the partition is when the input is sorted (in descending or ascending order). In these cases, the choice of pivot drives *Quicksort* to $O(n^2)$ performance. This is still the case for the routine presented here. However, we have accounted for repeated key values, so if there are e key values equal to the pivot's, then $\lceil e/2 \rceil$ end up in the right subfile. If all keys are always different, the partition can be redesigned so that it leaves the pivot in its correct position and out of further consideration. Figure 3.7 presents the global view of *Quicksort*. The second subfile is never empty (i.e., $p < r$), and thus, this *Quicksort* always terminates.

```
PARTITION(X, l, r)
1 pivot ← X[l]
2 i ← l − 1
3 j ← r + 1
4 while TRUE
5       do repeat j ← j − 1
6          until X[j] ≤ pivot
7          repeat i ← i + 1
8          until X[i] ≥ pivot
9          if i < j
10           then exchange X[i] ↔ X[j]
11           else return j
```

FIGURE 3.6 A simple and robust partitioning.

```
QUICKSORT(X, l, r)
1 if l < r
2   then split ← PARTITION(X, l, r)
3        QUICKSORT(X, l, split)
4        QUICKSORT(X, split + 1, r)
```

FIGURE 3.7 Pseudocode for *Quicksort*. An array is sorted with the call QUICKSORT $(X, 1, n)$.

The most popular variants to protect *Quicksort* from worst-case behavior are the following. The splitting item is selected as the median of a small sample, typically three items (the first, middle, and last element of the subarray). Many results show that this can reduce the expected average running time by about 5% to 10% (depending on the cost of comparisons and how many keys are different). This approach assures that a worst case happens with negligible low probability. For this method, the partitioning can accommodate the elements used in the sample so then no sentinels are required, but there is still the danger of many or all equal keys.

Another proposal delays selection of the splitting element; instead, a pair of elements that determine the range for the median is used. As the array is scanned, every time an element falls between the pair,

one of the values is updated to maintain the range as close to the median as possible. At the end of the partitioning two elements are in their final positions, dividing the interval. This method is fairly robust, but it enlarges the inner loop deteriorating performance, there is a subtle loss of randomness, and it also complicates the code significantly. Correctness for many equal keys remains a delicate issue.

Other methods are not truly comparison-based; for example, they use pivots that are arithmetic averages of the keys. These methods reduce the applicability of the routine and may loop forever on equal keys.

Randomness can be a useful tool in algorithm design, especially if some bias in input is suspected. A randomized version of *Quicksort* is practical because there are many ways in which the algorithm can proceed with good performance and only a few worst cases. Some authors find displeasing the need to use a pseudo-random generator for a problem as well studied as sorting. However, we find that the simple partitioning routine presented if Fig. 3.7 is robust to many of the aspects that make partitioning difficult to code and can remain simple and robust while handling worst case performance with the use of randomization. The randomized version of *Quicksort* is extremely solid and easy to code; refer to Fig. 3.8. Moreover, the inner loop of *Quicksort* remains extremely short; it is inside partition and consists of modifying an integer by 1 (increment or decrement, a very efficient operation in current hardware) and comparing a key with the key of the pivot (this value, along with the indexes i and j, can be placed in a register of the CPU). Also, access to the array exhibits a lot of locality of reference. By using the randomized version, the space requirements become $O(\log n)$ without the need to sort recursively the smallest of the two subfiles produced by the partitioning. If ever in practice the algorithm is taking too long, just halting it and running it again will provide a new seed with extremely high probability of reasonable performance.

```
ROUGHLY QUICKSORT(X, l, r)
1 while r − l > 10
2       do i ← RANDOM(l, r)
3          exchange X[i] ↔ X[l]
4          split ← PARTITION(X, l, r)
5          ROUGHLY QUICKSORT(X, l, split)
6          l ← split + 1
```

FIGURE 3.8 Randomized and tuned version of *Quicksort*.

Further improvements can now be made to tune up the code presented here (of course, sacrificing some simplicity). One of the recursive calls can be eliminated by **tail recursion removal**, and thus the time for half of the procedure calls is saved. Finally, it is not necessary to use a technique such as *Quicksort* itself to sort the very small subarrays produced in the final recursive calls. It is about 15% to 20% more efficient to sort small files of less than 10 items by a final call to *Insertion Sort* to complete the sorting. Figure 3.8 illustrates the tuned hybrid version of *Quicksort* that incorporates these improvements. To sort a file, first a call is made as ROUGHLY QUICKSORT$(X, 1, n)$ immediately followed by the call INSERTION SORT(X, n). Obviously both calls should be packed under a call for *Quicksort* to avoid accidentally forgetting to make both calls. However, for testing purposes, it is good practice to call them separately. Otherwise, we may receive the impression the implementation of the *Quicksort* part is correct, while *Insertion Sort* is actually doing the sorting.

It is worth revising the analysis of *Quicksort*. We will do this for the randomized version. This version has no bad inputs. For the same input, each run has different behavior. The analysis computes the expected time for each input, and it shows that this is $O(n \log n)$ time.

Suppose that $i \leftarrow$ RANDOM(l, r) returns an integer in $[l, \ldots, r]$ with uniform probability. That is, Prob$[i = k] = 1/(r - l + 1)$, for all $k \in [l \ldots r]$. Let $T_{RQ}(X)$ be the number of comparisons performed by *Randomized Quicksort* on an input X. Because the algorithm is randomized, $T_{RQ}(X)$ is a random variable, and we will be interested in its expected value. For the analysis, we evaluate the largest expected value of $T_{RQ}(X)$ over all inputs with n different key values. Thus, we estimate $E[T_{RQ}(n)] =$

$\max\{E[T_{RQ}(X)] \mid \|X\| = n\}$. The largest subproblem for a recursive call is $n-1$. Thus, the recursive form of the algorithm allows the following derivation, where c is a constant.

$$E\left[T_{RQ}(n)\right] = \sum_{i=1}^{n-1} \text{Prob}[i=k] E\left[\begin{array}{l}\sharp \text{ of comparisons when subproblems} \\ \text{are of sizes } i \text{ and } n-i\end{array}\right] + cn \tag{3.4}$$

$$\leq cn + \frac{1}{n}\sum_{i=1}^{n-1}\left(E\left[T_{RQ}(i)\right] + E\left[T_{RQ}(n-i)\right]\right) \tag{3.5}$$

$$= cn + \frac{2}{n}\sum_{i=1}^{n-1} E\left[T_{RQ}(i)\right]. \tag{3.6}$$

Since $E[T_{RQ}(0)] \leq b$ and $E[T_{RQ}(1)] \leq b$ for some constant b, then it is not hard to verify by induction that $E[T_{RQ}(n)] \leq kn\log_e n = O(n\log n)$, for all $n \geq 2$, which is the required result. Moreover, recurrence (3.6) can be solved exactly to obtain an expression for the constant hidden under the O notation.

Mergesort

Mergesort is not only a prototype of the conquer form in divide-and-conquer, as we saw earlier. *Mergesort* has two properties that can make it a better choice over *Heapsort* and *Quicksort* in many applications. The first of these properties is that *Mergesort* is naturally a stable sorting algorithm, while additional efforts are required to obtain stable versions of *Heapsort* and *Quicksort*. The second property is that access to the data is sequential; thus, data does not have to be in an array. This makes *Mergesort* an ideal method to sort linked lists. It is possible to use a divide form for obtaining a *Quicksort* version for lists that is also stable. However, the methods for protection against quadratic worst cases still make *Mergesort* a more fortunate choice. The advantages of *Mergesort* are not without cost. *Mergesort* requires $O(n)$ extra space (for another array or for the pointers in the linked list implementation). It is possible to implement *Mergesort* with constant space, but the gain hardly justifies the added programming effort.

Mergesort is based upon merging two sorted sequences of roughly the same length. Actually, *merging* is a very common special case of sorting, and it is interesting in its own right. Merging two ordered lists is achieved by repeatedly comparing the head elements and moving the one with the smaller key to the output list. Figure 3.9 shows PASCAL code for merging two linked lists. The PASCAL code for linked lists of *Mergesort* is shown in Fig. 3.10. It uses the function for merging of the previous figure. This implementation of *Mergesort* is more general that a sorting procedure for all the items in a linked list. It is a PASCAL function with two parameters, the head of the lists to be sorted and an integer n indicating how many items from the head should be included in the sort. The implementation returns as a result the head of the sorted portion and the head of the original list is a VAR parameter adjusted to point to the $(n+1)$th element of the original sequence. If n is larger or equal to the length of the list, the pointer returned includes the whole list and the VAR parameter is set to nil.

Restricted Universe Sorts

In this section we present algorithms that use other aspects about the keys to carry out the sorting. These algorithms were very popular at some point, and were the standard to sort punched cards. With the emergence of comparison-based sorting algorithms, which provided generality as well as elegant analyses and matching bounds, these algorithms lost popularity. However, their implementations can be much faster than comparison-based sorting algorithms. The choice between comparison-based methods and these types of algorithms may depend on the particular application. For a general sorting routine, many factors must be considered, and the criteria to determine which approach is best should not be limited to just running time. If the keys meet the conditions for using these methods, they are certainly a very

```
type
     list ↑ item;
     item =record k: keytype;
                  next : list;
            end;

function merge(X1,X2:list):list;
var head, tail, t: list;
begin
     head := nil;
     while X2 <> nil
          do   if X1 = nil (* reverse roles of X2 with X1 *)
               then begin X1:=X2; X2:= nil; end
               else begin
                         if X2↑.k > X1↑.k
                         then begin t:=X1; X1:=X1↑.next end;
                         else begin t:=X2; X2:=X2↑.next end;
                         t↑.next := nil;
                         if head = nil then head:=t;
                         else tail↑.next := t;
                         tail := t;
                    end;
     if head = nil then head :=X1;
     else tail↑.next :=X1;
     merge := head
end
```

FIGURE 3.9 PASCAL code for merging two linked lists.

```
function mergesort(VAR: head; n:integer): list;
var t: list;
begin
     if head = nil
     then mergesort = nil
     else if n> 1
          then mergesort:=merge( mergesort(head, n div 2), mergesort(head, (n+1) div 2) )
          else begin
                    t := head;
                    head := head↑.next;
                    t↑.next := nil;
                    mergesort := t;
               end
end
```

FIGURE 3.10 PASCAL code for a merge function that sorts the first n items of a linked list.

good alternative. In fact, today's technology has word lengths and memory sizes that make competitive many of the algorithms presented here. These algorithms were considered useful for only small restricted universes. Large restricted universes can be implemented with current memory sizes and current word sizes for many practical cases. Recent research has shown theoretical improvements on older versions of these methods [2].

Distribution Counting

A special situation of the sorting problem $X = \langle x_1, \ldots, x_n \rangle$ is the sorting of n distinct integers in the range $[1, m]$. If the value of $m = O(n)$, the fact that the x_i are distinct integers allows a very simple sorting method that runs in $O(n)$ time. Use a temporary array $T : [1, m]$ and place each x_i in $T[x_i]$. Scan T to collect the items in sorted order (where T was initialized to hold only 0).

This idea can be extended in many ways; the first is to handle the case when the integers are no longer distinct, and the resulting method is called *Distribution Counting.* The fundamental idea is to determine, for each x_i, its *rank.* The rank is the number of elements less or equal (but before x_i in X) than x_i. The

rank can be used to place x_i directly in its final position in the output array OUT. To compute the rank, we use the fact that the set of possible key values is the integers in $[1, m]$. We count the number E_k of values in X that equal k, for $k = 1, \ldots, m$. Arithmetic sums $\Sigma_{k=1}^{i} E_k$ can be used to find how many x_j are less or equal to x_i. Scanning through X, we can now find the destination of x_i, when x_i is reached. Figure 3.11 presents the pseudocode for the algorithm. Observe that two loops are till n and two till m. From this observation, the $O(n + m) = O(n)$ time complexity follows directly. The method has the disadvantage that extra space is required; however, in practice, we will use this method when m fits our available main memory, and in such cases, this extra space is not a problem.

```
DISTRIBUTION COUNTING(X, m, OUT)
1 for k ← 1 to m
2     do count[k] ← 0
3 for i ← 1 to n
4     do count[X_i] ← count[X_i] + 1
5 for k ← 2 to m
6     do count[k] ← count[k] + count[k − 1]
7 for i ← n downto 1
8     do OUT[count[X_i]] ← X_i
9        count[X_i] ← count[X_i]-1
```

FIGURE 3.11 Pseudocode for *Distribution Counting.*

A very appealing property of *Distribution Counting* is that it is a stable method. Observe that not a single comparison is required to sort. However, we need an array of size m, and the key values must fit the addressing space.

Bucket Sort

Bucket Sort is an extension to the idea of finding out where in the output array each x_i should be placed. However, the keys are not necessarily integers. We assume that we can apply an interpolation formula to the keys to obtain a new key in the real interval $[0, 1)$ which proportionally indicates where x_i should be relative to the smallest and largest possible keys. The interval $[0, 1)$ is partitioned into m equal-sized consecutive subintervals, each with an associated queue. The item x_i is placed in the queue q_j when the interpolation address from its key lies in the jth interval. The queues are sorted recursively, and then concatenated starting from the lower interval. The first-in last-out properties of the queues assure that, if the recursive sorting is stable, the overall sorting is stable. In particular, if *Bucket Sort* is called recursively, the method is stable. However, in practice, it is expected that queues will have very few items after one or two partitions. Thus, it is convenient to switch to an alternative stable sorting methods to sort the items in each bucket, most preferable *Insertion Sort.* For the insertion into the queues, an implementation that allows insertion in constant time should be used. Usually, linked lists with a pointer to their last item is the best alternative. This also assures that the catenation of the queues is efficient.

The method has an excellent average case time complexity, namely, it is linear (when $m = \Theta(n)$). However, the assumption is a uniform distribution of the interpolated keys in $[0, 1)$. In the worst scenario, the method may send every item into one bucket only, resulting in quadratic performance. The difficulty lies in finding the interpolation function. These functions work with large integers (like the maximum key) and must be carefully programmed to avoid integer overflow.

However, the method has been specialized so that k rounds of it sort k-tuples of integers in $[1, m]$ in $O(k(n + m))$ time, and also to sort strings of characters with excellent results [1]. Namely, strings of characters are sorted in $O(m + L)$ where L it the total length of the strings. In this cases, the alphabet of characters defines a restricted universe and the interpolation formula is just a displacement from the

```
BUCKET SORT(X)
1 for i ← 0 to m − 1
2     do Queue[i] ← empty
3 for i ← 1 to n
4     do insert x_i into Queue[⌊interpol(x_i)m⌋]
5 for i ← 0 to m − 1
6     do Sort(Queue[i])
7 for i ← 1 to m − 1
8     do Concatenate Queue[i] at the back of Queue[0]
9 return Queue[0]
```

FIGURE 3.12 The code for *Bucket Sort.*

smallest value. Moreover, the number of buckets can be made equal to the different values of the universe. These specializations are very similar to radix sorting, which we discuss next.

Radix Sort

Radix sort refers to a family of sorting methods where the keys are interpreted as representation in some base (usually a power of 2) or as strings over a given small alphabet. The radix sorts examine the digits of this representation in as many rounds as the length of the key to achieve the sorting. Thus, radix sorts perform several passes over the input, in each pass, performing decisions by one digit only.

The sorting can be done from the most significant digit toward the least significant digit or the other way around. The radix sort version that goes from most significant digit toward least significant digit is called *Top-Down Radix Sort*, *MSD Radix Sort*, or *Radix Exchange Sort* [9, 12, 19]. It resembles *Bucket Sort*, and from the perspective of divide-and-conquer is a method of the divide form. The most significant digit is used to split the items into groups. Next, the algorithm is applied recursively to the groups separatedly, with the first digit out of consideration. The sorted groups are collected by the order of increasing values of the splitting digit. Recursion is terminated by groups of size one. If we consider the level of recursion as rounds over the string of digits of the key, the algorithm keeps the invariant that after the ith pass, the input is sorted according to the first i digits of the keys.

The radix sort version that proceeds from the least significant digit toward the most significant digit is usually called *Bottom-Up Radix Sort*, *Straight Radix Sort*, *LSD Radix sort*, or just *Radix Sort* [9, 12, 19]. It could be considered as doing the activities of each round in different order. Split the items into groups according to the digit under consideration, group the items in order of increasing values of the splitting digit. Apply the algorithm recursively to all the items, but considering the next more significant digit. At first, it may not seem clear why this method is correct. It has a dual invariant to the *Top-Down Sort*; however, after the ith pass, the input is sorted according to the last i digits of the keys. Thus, for this *Bottom-Up* version to work, it is crucial that the insertion of items into their groups is made in the first-in first-out order of a queue. For the *Top-Down* version, this is only required to ensure stability. Both methods are stable though.

The *Top-Down* version has several advantages and disadvantages with respect to the *Bottom-Up* version. In the *Top-Down* version, the algorithm only examines the distinguishing prefixes, while the entire set of digits of all keys are examined by the *Bottom-Up* version. However, *Top-Down* needs space to keep track of the recursive calls generated, while the *Bottom-Up* version does not. If the input digits have a random distribution, then both versions of radix sort are very effective. However, in practice this assumption regarding the distribution is not the case. For example, if the digits are the bits of characters, the first leading bits of all lower case letters are the same in most character encoding schemes. Thus, *Top-Down Radix Sort* deteriorates with files with many equal keys (similar to *Bucket Sort*).

The *Bottom-Up* version is like a *Distribution Counting* on the digit that is being used. In fact, this is the easiest way to implement it. Thus, the digits can be processed more naturally as groups of digits (and allowing a larger array for the distribution counting). This is an advantage of the *Bottom-Up* version over the *Top-Down* version.

It should be pointed out that radix sorts can be considered linear in the size of the input, since each digit of the keys is examined only once. However, other variants of the analysis are possible; these include modifying the assumptions regarding the distribution of keys or according to considerations of the word size of the machine. Some authors think that the n keys require $\log n$ bits to be represented and stored in memory. From this perspective, radix sorts require $\log n$ passes with $\Omega(n)$ operations on them, still amounting to $O(n \log n)$ time. In any case, radix sorts, are reasonable methods for a sorting routine, or for a hybrid one. One hybrid method proposed by Sedgewick [19] consist of using the *Bottom-up* version of radix sort, but for the most significant half of the digits of the keys. This makes the file almost sorted, so the sort can be finished by *Insertion Sort.* The result is a linear sorting methods for most current word sizes on randomly distributed keys.

Order Statistics

The kth *order statistic* of a sequence of n items is the kth largest item. In particular, the smallest element is the first order statistic while the largest element is the nth order statistic. Finding the smallest or largest item in a sequence, can easily be achieved in linear time. For the smallest item, we just have to scan the sequence, remembering the smallest item seen so far. Obviously, we can also find the first and second statistic in linear time by the same procedure, just remembering the two smallest items seen so far. However, as soon as $\log n$ statistics are required, it is best to sort the sequence and retrieve any order statistic required directly.

A common request is to find jointly the smallest and largest items of a sequence of n item. Scanning through the sequence remembering the smallest and largest items seen so far requires that each new item be compared with what is being remembered; thus, $2n + O(1)$ comparisons are required. A better alternative for this case is to from $\lfloor n/2 \rfloor$ pairs of items, and perform the comparisons within pairs. We find the smallest item among the smaller items of the pairs, while the largest is found among the larger items of the pairs (a final comparison may be required for an element left out when n is odd). This results in $\lfloor 3n/2 \rfloor + O(1)$ comparison, which in some applications is worth the effort.

The fact that the smallest and largest items can be retrieved in $O(n)$ time without the need of sorting made the quest for linear algorithms for the kth order statistic a very interesting one for some time. Still, today there are many theoreticians researching the possibility of linear selection of the median (the $\lfloor n/2 \rfloor$th item) with a smaller constant factor. As a matter of fact, selection of the k largest item is another illustration of the use of average complexity to reflect a practical situation more accurately than worst-case analysis. The theoretical worst-case linear algorithms are so complex that very few authors dare to present pseudocode for them. This is perfectly justified, because nobody should implement worst-case algorithms in light of very efficient algorithms in the expected case, which are far easier conceptually as well as simpler in programming effort terms.

Lets consider divide-and-conquer approaches to finding the kth largest element. If we take the conquer form, as in *Mergesort*, it seems difficult to imagine how the kth largest item of the left subsequence and the kth largest item of the right subsequence relate to the kth largest item of the overall subsequence. However, if we take the divide form, as in *Quicksort*, we see that partitioning divides the input and conceptually splits by the correct rank of the pivot. If the position of the pivot is $i \geq k$, we only need to search for the kth largest element of the $X_{\leq}$ subsequence. Otherwise, we have found i items that we can remove from further consideration, since they are smaller than the kth largest. We just need to find the $k - i$ largest in the subsequence $X_{\geq}$. This approach to divide-and-conquer results in only one subproblem to be pursued recursively. The analysis results in an algorithm that requires $O(n)$ time in the expected case. Such a method requires, again, careful protection against the worst cases. Moreover, it is more likely that a file

that is being analyzed for its order statistics has being inadvertently sorted before, setting up a potential worst case for a selection method whose selection of the pivot is not adequate. In the algorithm presented in Fig. 3.13 we use the same partitioning algorithm as in the section for *Quicksort*; refer to Fig. 3.7.

```
SELECT(X, l, r, k)
1 if r = l
2    then return X[l]
3 i ← RANDOM(l, r)
4 exchange X[i] ↔ X[l]
5 split ← PARTITION(X, l, r)
6 if k ≤ split
7    then return SELECT(X, l, split, k)
8    else return SELECT(X, split + 1, r, k)
```

FIGURE 3.13 Randomized version for selection of the kth largest.

External Sorting

There are many situations where external sorting is required for the maintenance of a well-organized database. Files are often maintained in sorted order with respect to some attribute to facilitate searching and processing. External sorting is used not only to produce organized output, but also to efficiently implement complex operations such as a relational join. External sorting consists of two phases: a *run-creation phase* and a *merge phase*. During the first phase, the file to be sorted is divided into smaller sorted sequences called *initial runs* or *strings* [12]. These runs are created by bringing into main memory a fragment of the file. During the second phase, one or more activations of *multiway-merge* are used to combine the initial runs into a single run [16].

Currently, the sorting of files that are too large to be held in main memory is performed on disk drives [16]; see Fig. 3.14. Situations where only one disk drive is available are now uncommon, since this usually results into very slow sorting processes and complex algorithms while the problem can easily be solved with another disk drive (which are more affordable today). In each pass (one for run-creation and one or more for merging) the input file is read from the *IN* disk drive. The output of one pass is the input of the next, until a single run is formed; thus, the *IN* and *OUT* disks swap roles after each pass. While one of the input buffers, say I_i, $i \in \{0, f-1\}$, is being filled, the sorting process reads records from some of the other input buffers $I_0, \ldots, I_{i-1}, I_{i+1}, \ldots, I_{f-1}$. The output file of each pass is written using *double buffering*. While one of the output buffers, say O_i, $i \in \{0, 1\}$, is being filled, the other buffer O_{1-i} is being written to disk. The roles of O_i and O_{1-i} are interchanged when one buffer is full and the other is empty. In practice, several data records are read in a I/O operation forming a **physical block**, while the capacity of buffers defines the size of *logical blocks*. For the description of external sorting methods, the use of logical blocks is usually sufficient and we will just name them *blocks*.

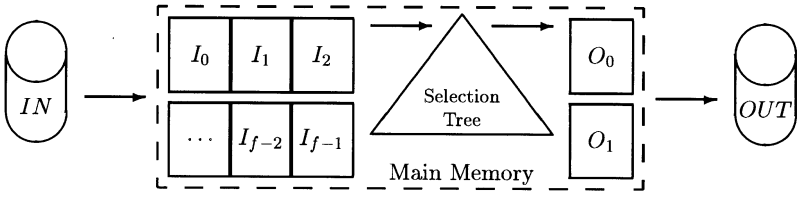


FIGURE 3.14 The model for a pass of external sorting.

During the run-creation phase the number f of input buffers is 2 and reading is sequential using double-buffering. During a merge pass, the next block to be read is normally from a different run and the disk arm must be repositioned. Thus, reading is normally not sequential. Writing during the merge is, however, faster than reading, since normally it is performed sequentially and no seeks are involved (except for the minimum seek for the next cylinder when the current cylinder is full). In each pass, the output is written sequentially to disk. Scattered writing during a pass in anticipation of saving seeks because of some sequential reading of the next pass has been shown to be counterproductive [20]. Thus, in the two-disk model (see Fig. 3.14) the writing during the merge will completely overlap with the reading and its time requirements are a minor concern. In contrast to the run-creation phase in which reading is sequential, merging may require a seek each time a data block is read.

Replacement selection usually produces runs that are larger than the available main memory; the larger the initial runs, the faster the overall sorting. Replacement selection allows full overlapping of I/O with sequential reading and writing of data, and it is standard for the run creation phase. The classic result on the performance of replacement selection establishes that, when all input files are assumed to be equally likely, the asymptotic expected length of the resulting runs is twice the size of available main memory [12]. Other researchers have modified replacement selection such that, asymptotically, the expected length of an initial run is more than twice the size of available main memory. These methods have received limited acceptance because they require more sophisticated I/O operations and prevent full overlapping; hence, the possible benefits hardly justify the added complexity of the methods. Similarly, any attempt to design a new run-creation method that profits from the existing order in the input file will almost certainly have inefficient overlapping of I/O operations. More recently, it has been mathematically confirmed that the lengths of the runs created by replacement selection increase as the order in the input file increases [6].

During the run-creation phase *Replacement Selection* consists of a *selection tree.* This structure is a binary tree where nodes hold the smaller of their two children. It is called selection tree because the item at the root of the tree holds the smallest key. By tracing the path up of the smallest key from its place at a leaf to the root, we have selected the smallest item among those in the leaves. If we replace the smallest item with another value at the corresponding leaf, we only are required to update the path to the root. Performing the comparisons along this path updates the root as the new smallest item. Selection trees are different from heaps (ordered to extract the item with smallest keys) in that selection trees have fixed size. During the selection phase, the selection tree is initialized with the first P elements of the input file (where P is the available internal memory). Repeatedly, the smallest item is removed from the selection tree and placed in the output stream, the next item from the input file is inserted in its place as a leaf in the selection tree. The name *Replacement Selection* comes from the fact that the new item from the input file replaces the item just selected to the output stream. To make certain that items enter and leave the selection tree in the proper order, the comparisons are not only with respect to the sorting keys, but also with respect to the current run being output. Thus, the selection tree uses lexicographically the composite keys (r, key), where r is the *run-number* of the item, and key is the sorting key. The run number of an item entering the selection tree is known by comparing it to the item which it is replacing. If it is smaller than the item just sent to the output stream, the run number is one more than the current run number; otherwise, it is the same run number as the current run.

The Merge

In the merging phase of external sorting, blocks from each run are read into main memory, and the records from each block are extracted and merged into a single run. Replacement selection (implemented with a *selection tree*) is also used as the process to merge several runs into one [12]. Here, however, each leaf is associated with each of the runs being merged. The *order* of the merge is the number of leaves in the selection tree. Because main memory is a critical resource here, items in the selection tree are not replicated, but rather a tree of losers is used [12].

There are many factors involved in the performance of disk drives. For example, larger main memories implies larger data blocks and the *block-transfer rate* is now significant with respect to *seek time* and *rotational latency.* Using a larger block size reduces the total number of reads (and seeks) and reduces the overhead of the merging phase. Now, a merge pass requires at least as many buffers as the order ω of the merge. On one hand, using only one buffer for each run, maximizes block size, and if we perform a seek for each block, it reduces the total number of seeks. However, we cannot overlap I/O. On the other hand, using more buffers, say, two buffers for each run, increases the overlap of I/O, but reduces the block size and increases the total number of seeks. Note that because the amount of main memory is fixed during the merge phase, the number of buffers is inversely proportional to their size.

Salzberg [17] found that, for almost all situations, the use of $f = 2\omega$ buffers assigned as pairs to each merging stream outperforms the use of ω fixed buffers. Recently, it has been shown that double buffering cannot take advantage of nearly-sorted data [7]. Double buffering does not guarantee full overlap of I/O during merging. When buffers are not fixed to a particular run, but can be reassigned to another run during the merge, they are called *floating buffers* [11]. Using twice as many floating buffers as the order of the merge provides maximum overlap of I/O [11]. Zheng and Larson [20] combined Knuth's [12] *forecasting* and floating-buffers techniques and proposed six to ten times as many floating input buffers as the order of the merge. In fact, techniques based on floating buffers not only ensure full overlap of I/O during merging, but also require less main memory. Moreover, it was also demonstrated that techniques based on floating buffers profit significantly from nearly-sorted files [7].

Floating Buffers

The *consumption sequence* is the particular order in which the merge consumes blocks from the runs being merged. This sequence can be precomputed by extracting the highest key (the last key) from each data block (during the previous pass) and sorting them. The time taken to compute the consumption sequence can be overlapped with the output of the last run and the necessary space for the subsidiary internal sort is also available then; thus, the entire consumption sequence can be computed during the previous pass with negligible overhead. The floating-buffers technique exploits the knowledge of the consumption sequence to speed up reading.

We illustrate double buffering and floating buffers with a merging example of four runs that are placed sequentially as shown in Fig. 3.15. Let $C = \langle C_1, C_2, \ldots, C_T \rangle$ be the consumption sequence, where C_i identifies a data block with respect to its location on the disk. For example, consider

$$C = \langle 1, 8, 13, 18, 2, 9, 14, 19, 3, 10, 4, 5, 15, 11, 12, 16, 17, 20, 6, 21, 7, 22, 23 \rangle .^{1}$$

Double buffering uses twice as many buffers as runs, and a seek is required each time a block is needed from a different run. Moreover, even when a block is needed from the same run, this may not be known at exactly the right time; therefore, the disk will continue to rotate and every read has rotational latency. In the example, double buffering reads one block from each run (with a seek in each case) and then it reads a second block from each run (again with a seek in each case). Next, the disk arm travels to block 3 to read a new block from the first run (one more seek). Afterward, the arm moves to block 10 to get a new block from the second run. Then, it moves to block 4 and reads block 4 and block 5, but a seek is not required for reading block 5 since the run-creation phase places blocks from the same run sequentially on the disk. In total, double buffering performs 19 seeks.

[1] Alternatively, the consumption sequence can be specified as $C = \langle c_1, c_2, \ldots, c_T \rangle$ where c_i is the run from which the ith block should be read. For this example, $C = \langle 1, 2, 3, 4, 1, 2, 3, 4, 1, 2, 1, 1, 3, 2, 2, 3, 3, 4, 1, 4, 1, 4, 4 \rangle$.

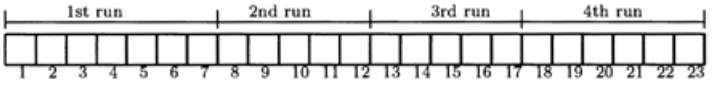


FIGURE 3.15 An example of four runs (written sequentially) on a disk.

We now consider the effect of using only seven buffers (of the same size as before) managed as floating buffers [11]. In this case, we use only three more buffers than the number of runs but we use knowledge of the consumption sequence. The seven buffers are used as follows: four buffers contain a block from each run that are currently being consumed by the merge, two buffers contain look-ahead data, and one buffer is used for reading new data. In the previous example, after the merging of the two data blocks 1 and 7, the buffers are as follows:

block 24	block 2	empty	block 8	block 15	block 23	block 16
buffer 1	buffer 2	buffer 3	buffer 4	buffer 5	buffer 6	buffer 7

Data from buffers 2, 4, 5, and 6 are consumed by the merge and are placed in an output buffer. At the same time, one data block is read into buffer 3. As soon as the merge needs a new block from run 3, it is already in buffer 7 and the merge releases buffer 5. Thus, the system enters a new state in which we merge buffers 2, 4, 6, and 7, and we read a new block into buffer 5. Figure 3.16 shows the merge when the blocks are read in the following order:

$$\langle \underline{1, 2, 8, 9}, \underline{13, 18}, \underline{14, 19}, \underline{3, 4, 5}, \underline{10, 11, 12}, \underline{15, 16, 17}, \underline{6, 7}, \underline{20, 21, 22, 23} \rangle \ . \tag{3.7}$$

The letter e denotes an empty buffer and b_i denotes a buffer that holds data block i. Reading activity is

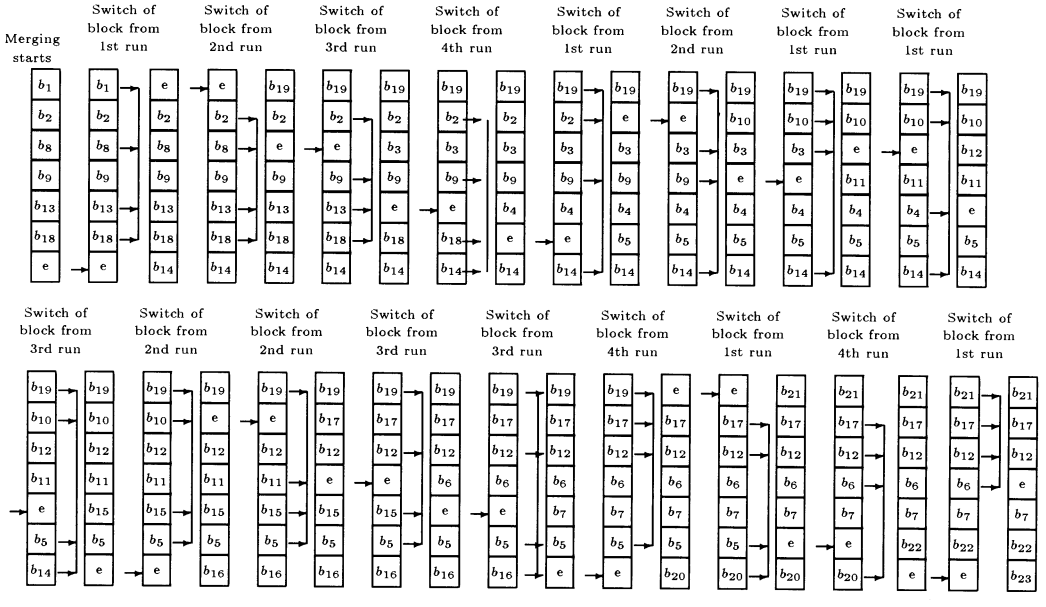


FIGURE 3.16 Merging with seven floating buffers.

indicated by an arrow into a buffer and merging activity is indicated by an arrow out of a buffer. The ordered sequence in which the blocks are read from the disk into main memory is called the *reading sequence.* A reading sequence is *feasible* if every time the merge needs a block, it is already in main memory and reading never has to wait because there is no buffer space available in main memory. Note that the consumption

sequence (with two or more floating buffers for each run) is always a feasible reading sequence [11]. In the example of Fig. 3.16, not only is the new reading sequence feasible and provides just-in-time blocks for the merge, but also it requires only 11 seeks and uses even less memory than double buffering!

Computing a Feasible Reading Sequence

In the previous subsection, we have illustrated that that floating buffers can save both main memory and the overhead due to seeks. There are, however, two important aspects in using floating buffers. First, floating buffers are effective when knowledge of the consumption sequence is used to compute reading sequences [20]. The consumption sequence is computed by the previous pass (and for the first pass, during the run-creation phase). The consumption sequence is the consumption order of data-blocks in the next pass. Thus, the buffer size for the next pass must be known by the previous pass. Overall, the buffer size and the number f of input floating buffers for each pass must be chosen before starting the sorting.

Before sorting we usually know the length $|X_I|$ of the file, and assuming it is in random order, we can expect, after run-creation, initial runs of twice the size P of the available main memory. That is, $E[Runs(X_O)] = |X_I|/2P$. Now, we can decide the number of merge passes (most commonly only one) and the order ω of these merge passes. Zheng and Larson [20] follow this approach and recommend the number f of floating buffers to be between 6ω and 10ω. Once the value of f is chosen, the buffer size is determined, and where to partition the input into data blocks is defined. The justification for this strategy is that current memory sizes allow it and an inaccurate estimate of the number of initial runs or their sizes seems not to affect performance [20]. It has been shown that if the input is nearly sorted, the fact that the technique just described may choose f much larger that 10ω does not affect floating buffers. Moreover, for nearly sorted files, reading during the merge becomes almost sequential, and over 80% of seeks can be avoided [7].

The second difficulty consists of computing feasible reading sequences that minimize the number of seeks. Zheng and Larson [20] have related the problem of finding the optimal feasible reading sequence to the traveling salesman problem; thus, research has concentrated on approximation algorithms.

To describe precisely the problem finding a feasible reading sequence with fewer seeks we will partition, into what we call *groups,* the blocks in a read sequence that are adjacent and belong to the same run. The groups are indicated by underlining in sequence (3.7). A seek is needed at the beginning of each group, because the disk head has to move to a different run. Inside a group, we read blocks from a single run sequentially, as placed by the run-creation phase or a previous merge pass. Note that there is no improvement in reading data blocks from the same run in different order than in the consumption sequence. For long groups, a seek may be required from one cylinder to the next, but such a seek takes minimum time because the reading is sequential. Moreover, in this case the need to read a block from the next cylinder is known early enough to avoid rotational latency. Thus, we want to minimize the number of groups while maintaining feasibility.

We now describe *group shifting,* an algorithm that computes a feasible reading sequence with fewer seeks. Group shifting starts with a consumption sequence $C = \langle C_1, \ldots, C_T \rangle$ as the initial feasible reading sequence. It scans the groups in the sequence twice. The first scan produces a feasible reading sequence that is the input for the next. A scan builds a new feasible reading sequence incrementally. The first ω groups of the new reading sequence are the first ω groups of the previous reading sequence because, for $i = 1, \ldots, \omega$, the optimal feasible reading sequence for the first i groups consists of the first i groups of the consumption sequence. In each scan, the groups of the previous sequence are analyzed in the order they appear. During the first scan an attempt is made to move each group in turn forward and catenate it with the previous group from the same run while preserving feasibility. A single group that results from the catenation of groups B_j and B_k is denoted by $(B_j B_k)$. During the second scan an attempt is made to move back the previous group from the same run under analysis, while preserving feasibility. We summarize the new algorithm in Fig. 3.17.

```
GROUP-SHIFTING (C = ⟨C_1, ..., C_T⟩ original consumption sequence for a merge pass of order ω).
1 N ← ⟨C_1, C_2, ..., C_ω⟩
2 b ← ω
3 while b < T
4 doLet B the (b + 1)th group in C and
5    let the sequence N be N = ⟨B_1, B_2, ..., B_p⟩,
6    where B_l is the last group in N from the same run as B.
7    if group B can be moved forward and joined to B_l preserving feasibility,
8    then N ← ⟨B_1, ..., B_{l-1}, (B_l B), B_{l+1}, ..., B_p⟩,
9    else  append B to N.
10 M ← first ω groups in N
11 b ← ω
12 while b < |N|
13 do Let B be the (b + 1)th group in N and
14    let the sequence M be M = ⟨B_1, B_2, ..., B_p⟩,
15    where B_l is the last group in M from the same run as B.
16    if group B_l can be shifted to the end of M and joined to group B preserving feasibility,
17    then M ← ⟨B_1, ..., B_{l-1}, B_{l+1}, ..., B_p, (B_l B)⟩,
18    else append B to M
19 return M
```

FIGURE 3.17 The algorithm for reducing seeks during the merging phase with floating buffers.

For an example of a forward move during the first scan, consider $b = 4$, $M^b = \langle 1, 8, 13, 18\rangle$, and group $b + 1$ is 2. Then, $M^{b+1} = \langle 1, 2, 8, 13, 28\rangle$. As an example of a backward move during the second scan, consider $b = 18$,

$M^b = \langle 1, 2, 8, 9, 13, 18, 14, 19, 3, 4, 5, 10, 11, 12, 15, 16, 17, 20, 21, 6, 7\rangle$, and group $b + 1$ is 22, 23. Moving 20, 21 over 6, 7 gives the optimal sequence of Fig. 3.16.

The algorithm uses the following fact to test that feasibility is preserved. Let $C = \langle C_1, \ldots, C_T\rangle$ be the consumption sequence for ω runs with T data blocks. A read sequence $R = \langle R_1, \ldots, R_T\rangle$ is feasible for $f > \omega + 1$ floating buffers if and only if, for all k such that $f \leq k \leq T$, we have $\{C_1, C_2, \ldots, C_{k-f+\omega}\} \subset \{R_1, R_2, \ldots, R_{k-1}\}$.

3.4 Research Issues and Summary

We now look at some of the research issues on sorting from the practical point of view. In the area of internal sorting, advances in data structures for the abstract data type *dictionary* or for the abstract data type *priority queue* may result in newer or alternative sorting algorithms. The implementation of dictionaries by variants of binary search trees where the items can easily (in linear time) be recovered in sorted order with an *in-order* traversal, results in an immediate sorting algorithm. We just insert the items to be sorted into the tree implementing the dictionary using the sorting keys as the dictionary keys. Later, we extract the sorted order from the tree. An insertion sort is obtained for each representation of the dictionary. Some interesting advances, at the theoretical level, but perhaps at the practical level, have been obtained by using data structures like *Fusion Trees* [8]. Although these algorithms are currently somewhat complicated and they make use of dissecting keys and careful packing of information in memory words, the increase in the word size of computers is making them practically feasible.

Another area of research is the more careful study of the alternatives offered by radix sorts. Careful analyses have emerged for these methods and they take into consideration the effect of non-uniform distributions. Moreover, simple combinations of *Top-Down* and *Bottom-up* have resulted in hybrid Radix Sorting algorithms with very good performance [2]. In practical experiments, Andersson and Nilsson observed that their proposed *Forward Radix Sort* defeats some of the best alternatives offered by the comparison-based approach.

A third area of research with possible practical implications is the area of adaptive sorting. When the sorting algorithm takes advantage of existing order in the input, the time taken by the algorithm to sort is

a smoothly growing function of the size of the sequence and the disorder in the sequence. In this case, we say that the algorithm is *adaptive* [14]. Adaptive sorting algorithms are attractive because nearly sorted sequences are common in practice [12, 14, 18]; thus, we have the possibility of improving on algorithms that are oblivious to the existing order in the input.

So far we presented *Insertion Sort* as an example of this type of algorithm. Adaptive algorithms have received attention for comparison-based sorting. Many theoretical algorithms have been found for many *measures of disorder* [6, 13]. However, from the practical point of view, these algorithms usually involve more machinery. This additional overhead is unappealing because of its programming effort. Thus, room remains for providing adaptive sorting algorithms that are simple for the practitioner. Adaptive algorithms currently in use are Cook and Kim's *CKsort* [4], *Natural Mergesort* [12], *Skipsort* [6], and *Splaysort* [15]. Some of these algorithms have been shown to be far more efficient in nearly sorted inputs for just a small overhead on randomly permuted files, but they have not received wide acceptance.

Finally, let us summarize the alternatives when facing a sorting problem. First we must determine if our situation is in the area of external sorting. A model with two disk drives is recommended in this case. Use replacement selection for run creation and for merging, using floating buffers during the second phase. It is possible to tune the sorting for just one pass during the second phase.

If the situation is internal sorting of a small file, then *Insertion Sort* does the job. If we are sorting integers, or character strings or some other restricted universe, then *Distribution Counting*, *Bucket Sort*, and *Radix Sort* are very good choices. If we are after a stable methods, **restricted universe sorts** are also good options. If we need something more general, the next level up is *Shellsort*, and finally the tuned versions of $O(n \log n)$ comparison-based sorting algorithms. If we have serious grounds to suspect the inputs are nearly sorted, we should consider adaptive algorithms. Whenever the sorting key is a small portion of the data records, we should try to avoid expensive data moves by sorting a file of keys and indexes. Always preserve a clear and simple code.

3.5 Defining Terms

Adaptive: A sorting algorithm that can take advantage of existing order in the input, reducing its requirements for computational resources as a function of the amount of disorder in the input.

Comparison-based algorithm: A sorting methods that uses comparisons, and nothing else about the sorting keys, to rearrange the input into ascending or descending order.

Conquer form: An instantiation of the divide-and-conquer paradigm for the structure of an algorithm where the bulk of the work is combining the solutions of subproblems into a solution for the original problem.

Divide form: An instantiation of the divide-and-conquer paradigm for the structure of an algorithm where the bulk of the work is dividing the problem into subproblems.

External sorting: The situation when the file to be sorted is too large to fit in main memory.

Insertion sort: The family of sorting algorithms where one item is analyzed at a time and inserted into a data structure holding a representation of a sorted list of previously analyzed items.

Internal sorting: The situation when the file to be sorted is small enough to fit in main memory.

Multiway merge: The mechanism by which ω sorted runs are merged into a single run. The input runs are usually organized in pairs and merged using the standard method for merging two sorted sequences. The results are paired again, and merged, until just one run is produced. The parameter ω is called the order of the merge.

Restricted universe sorts: Algorithms that operate on the basis that the keys are members of a restricted set of values. They may not require comparisons of keys to perform the sorting.

Selection sorts: The family of sorting algorithms where the data items are retrieved from a data structure, one item at a time, in sorted order.

Sorting arrays: The data to be sorted is placed in an array and access to individual items can be done randomly. The goal of the sorting is that the ascending order matches the order of indices in the array.

Sorting liked lists: The data to be sorted is a sequence represented as a linked list. The goal is to rearrange the pointers of the linked list so that the linked list has the data in sorted order.

Stable: A sorting algorithm where the relative order of items with equal keys in the input sequence is always preserved in the sorted output.

References

[1] Aho, A.V., Hopcroft, J.E., and Ullman, J.D., *The Design and Analysis of Computer Algorithms.* Addison-Wesley, Reading, MA, 1974.

[2] Andersson, A. and Nilsson, S., A new efficient radix sort. In *Proceedings of the 35th Annual Symposium of Foundations of Computer Science,* 714–721. IEEE Computer Society, 1994.

[3] Bentley. J. and McIlroy, M.D., Engineering a sort function. *Software – Practice and Experience,* 23(11), 1249–1265, 1993.

[4] Cook, C.R. and Kim, D.J., Best sorting algorithms for nearly sorted lists. *CACM,* 23, 620–624, 1980.

[5] Cormen, T.H., Leiserson, C.E., and Rivest, R.L., *Introduction to Algorithms.* MIT Press, Cambridge, MA, 1990.

[6] Estivill-Castro, V. and Wood, D., A survey of adaptive sorting algorithms. *Computing Surveys,* 24, 441–476, 1992.

[7] Estivill-Castro, V. and Wood, D., Foundations for faster external sorting. In *Fourteenth Conference on the Foundations of Software Technology and Theoretical Computer Science,* 414–425, Madras, India, 1994. Springer-Verlag LNCS 880.

[8] Fredman, M.L and Willard D.E., Blasting through the information theoretic barrier with fusion trees. In *Proceedings of the 22nd Annual ACM Symposium on Theory of Computing,* 1–7. ACM, ACM Press, 1990.

[9] Gonnet, G.H. and Baeza-Yates, R., *Handbook of Algorithms and Data Structures, 2nd ed.,* Addison-Wesley, Don Mills, Ontario, 1991.

[10] Hoare, C.A.R., Algorithm 64, Quicksort. *CACM,* 4(7), 321, 1961.

[11] Horowitz, E. and Sahni, S., *Fundamentals of Data Structures.* Computer Science Press, Woodland Hill, CA, 1976.

[12] Knuth, D.E., *The Art of Computer Programming, Vol.3: Sorting and Searching.* Addison-Wesley, Reading, MA, 1973.

[13] Mannila, H., Measures of presortedness and optimal sorting algorithms. *IEEE T. on Computers,* C-34, 318–325, 1985.

[14] Mehlhorn, K., *Data Structures and Algorithms, Vol. 1: Sorting and Searching.* EATCS Monographs on Theoretical Computer Science. Springer-Verlag, Berlin/Heidelberg, 1984.

[15] Moffat, A., Eddy, A., Petterson, O., Splaysort. Fast, versatile, practical. *Software—Practice and Experience,* 26(7):781–797, July, 1996.

[16] Salzberg, B., *File Structures: An Analytic Approach.* Prentice-Hall, Englewood Cliffs, NJ, 1988.

[17] Salzberg, B., Merging sorted runs using large main memory. *Acta Informatica,* 27, 195–215, 1989.

[18] Sedgewick, R., *Quicksort.* Garland, New York, 1980.

[19] Sedgewick, R., *Algorithms.* Addison-Wesley, Reading, MA, 2nd ed., 1983.

[20] Zheng, L.Q. and Larson, P.A., Speeding up external mergesort. *IEEE Transactions on Knowledge and Data Engineering,* 8(2), 322–332, 1996.

Further Information

For the detailed arguments that provide the theoretical lower bounds for comparison-based sorting algorithms the reader may wish to consult [1, 14]. These books also include a description of sorting strings by *Bucket Sort* in *time proportional to the total length of* the strings.

Sedgewick's books on algorithms [19] provide illustrative descriptions of *Radix Sorts.* Other interesting algorithms, for example, *Linear Probing Sort*, although they are more complicated to program, usually have very good performance in practice. They can be reviewed in Gonnet and Baeza's handbook [9].

We have omited here algorithms for external sorting with tapes, since this is now rare. However, the reader may consult classical sources [9, 12].

For more information on the advances in fusion trees and radix sort, the reader may wish to review the *Proceedings of the IEEE Symposium of Foundations of Computer Science* (FOCS) or the *ACM Symposium on the Theory of Computing.*

4

Basic Data Structures[1]

Roberto Tamassia
Brown University

Bryan Cantrill
Brown University

4.1 Introduction

The study of data structures, i.e., methods for organizing data that are suitable for computer processing, is one of the classic topics of computer science. At the hardware level, a computer views storage devices such as internal memory and disk as holders of elementary data units (bytes), each accessible through its address (an integer). When writing programs, instead of manipulating the data at the byte level, it is convenient to organize them into higher level entities, called *data structures.*

Containers, Elements, and Locators

Most data structures can be viewed as **containers** that store a collection of objects of a given type, called the *elements* of the container. Often a total order is defined among the elements (e.g., alphabetically ordered names, points in the plane ordered by x-coordinate). We assume that the elements of a container can be accessed by means of variables called **locators**. When an object is inserted into the container, a locator is

[1]The material in this chapter was previously published in *The Computer Science and Engineering Handbook,* Allen B. Tucker, Editor-in-Chief, CRC Press, Boca Raton, FL, 1997.

returned, which can be later used to access or delete the object. A locator is typically implemented with a pointer or an index into an array.

A data structure has an associated repertory of operations, classified into *queries,* which retrieve information on the data structure (e.g., return the number of elements, or test the presence of a given element), and *updates,* which modify the data structure (e.g., insertion and deletion of elements). The performance of a data structure is characterized by the space requirement and the time complexity of the operations in its repertory. The *amortized* time complexity of an operation is the average time over a suitably defined **sequence** of operations.

However, efficiency is not the only quality measure of a data structure. Simplicity and ease of implementation should be taken into account when choosing a data structure for solving a practical problem.

Abstract Data Types

Data structures are concrete implementations of **abstract data types** (ADTs). A *data type* is a collection of objects. A data type can be mathematically specified (e.g., real number, directed graph) or concretely specified within a programming language (e.g., `int` in C, `set` in Pascal). An ADT is a mathematically specified data type equipped with operations that can be performed on the objects. Object-oriented programming languages, such as C++, provide support for expressing ADTs by means of *classes.* ADTs specify the data stored and the operations to be performed on them.

Main Issues in the Study of Data Structures

The following issues are of foremost importance in the study of data structures.

Static vs. Dynamic A *static* data structure supports only queries, while a dynamic data structure supports also updates. A *dynamic* data structure is often more complicated than its static counterpart supporting the same repertory of queries. A *persistent* data structure (see, e.g., [9]) is a dynamic data structure that supports operations on past versions. There are many problems for which no efficient dynamic data structures are known. It has been observed that there are strong similarities among the classes of problems that are difficult to parallelize and those that are difficult to dynamize (see, e.g., [32]). Further investigations are needed to study the relationship between parallel and incremental complexity [26].

Implicit vs. Explicit Two fundamental data organization mechanisms are used in data structures. In an *explicit* data structure, pointers (i.e., memory addresses) are used to link the elements and access them (e.g., a singly linked list, where each element has a pointer to the next one). In an *implicit* data structure, mathematical relationships support the retrieval of elements (e.g., array representation of a **heap**, see "Time Complexity"). Explicit data structures must use additional space to store pointers. However, they are more flexible for complex problems. Most programming languages support pointers and basic implicit data structures, such as arrays.

Internal vs. External Memory In a typical computer, there are two levels of memory: internal memory (RAM) and external memory (disk). The internal memory is much faster than external memory but has much smaller capacity. Data structures designed to work for data that fit into internal memory may not perform well for large amounts of data that need to be stored in external memory. For large-scale problems, data structures need to be designed that take into account the two levels of memory [1]. For example, two-level indices such as B-trees [6] have been designed to efficiently search in large databases.

Space vs. Time Data structures often exhibit a trade-off between space and time complexity. For example, suppose we want to represent a set of integers in the range $[0, N]$ (e.g., for a set of social security numbers $N = 10^{10} - 1$) such that we can efficiently query whether a given

element is in the set, insert an element, or delete an element. Two possible data structures for this problem are an N-element bit-array (where the bit in position i indicates the presence of integer i in the set), and a balanced **search tree** (such as a 2-3 tree or a red-black tree). The bit-array has optimal time complexity, since it supports queries, insertions, and deletions in constant time. However, it uses space proportional to the size N of the range, irrespectively of the number of elements actually stored. The balanced search tree supports queries, insertions, and deletions in logarithmic time but uses optimal space proportional to the current number of elements stored.

Theory vs. Practice A large and ever-growing body of theoretical research on data structures is available, where the performance is measured in asymptotic terms ("big-Oh" notation). While asymptotic complexity analysis is an important mathematical subject, it does not completely capture the notion of efficiency of data structures in practical scenarios, where constant factors cannot be disregarded and the difficulty of implementation substantially affects design and maintenance costs. Experimental studies comparing the practical efficiency of data structures for specific classes of problems should be encouraged to bridge the gap between the theory and practice of data structures.

Fundamental Data Structures

The following four data structures are ubiquitously used in the description of discrete algorithms, and serve as basic building blocks for realizing more complex data structures. They are covered in detail in the textbooks listed in "Further Information" and in the additional references provided.

Sequence A sequence is a container that stores elements in a certain linear order, which is imposed by the operations performed. The basic operations supported are retrieving, inserting, and removing an element given its position. Special types of sequences include stacks and queues, where insertions and deletions can be done only at the head or tail of the sequence. The basic realization of sequences are by means of arrays and linked lists. Concatenable queues (see, e.g., [18]) support additional operations such as splitting and splicing, and determining the sequence containing a given element. In external memory, a sequence is typically associated with a file.

Priority Queue A **priority queue** is a container of elements from a totally ordered universe that supports the basic operations of inserting an element and retrieving/removing the largest element. A key application of priority queues is to sorting algorithms. A heap is an efficient realization of a priority queue that embeds the elements into the ancestor/descendant partial order of a binary tree. A heap also admits an implicit realization where the nodes of the tree are mapped into the elements of an array (see "Time Complexity"). Sophisticated variations of priority queues include min-max heaps, pagodas, deaps, binomial heaps, and Fibonacci heaps. The buffer tree is efficient external-memory realization of a priority queue.

Dictionary A **dictionary** is a container of elements from a totally ordered universe that supports the basic operations of inserting/deleting elements and searching for a given element. **Hash tables** provide an efficient implicit realization of a dictionary. Efficient explicit implementations include skip lists [31], tries, and balanced search trees (e.g., **AVL-trees**, red-black trees, 2-3 trees, 2-3-4 trees, weight-balanced trees, biased search trees, splay trees). The technique of fractional cascading [3] speeds up searching for the same element in a collection of dictionaries. In external memory, dictionaries are typically implemented as B-trees and their variations.

Union-Find A union-find data structure represents a collection disjoint sets and supports the two fundamental operations of merging two sets and finding the set containing a given element.

There is a simple and optimal union-find data structure (rooted tree with path compression) whose time complexity analysis is very difficult to analyze. See, e.g., [15].

Examples of fundamental data structures used in three major application domains are mentioned below.

Graphs and Networks adjacency matrix, adjacency lists, link-cut tree [34], dynamic expression tree [5], topology tree [14], SPQR-tree [8], sparsification tree [11]. See also, e.g., [12, 23, 35].

Text Processing string, suffix tree, Patricia tree. See, e.g., [16].

Geometry and Graphics binary space partition tree, chain tree, trapezoid tree, range tree, segment-tree, interval-tree, priority-search tree, hull-tree, quad-tree, R-tree, grid file, metablock tree. See, e.g., [4, 10, 13, 23, 27, 28, 30].

Organization of the Chapter

The rest of this chapter focuses on three fundamental abstract data types: sequences, priority queues, and dictionaries. Examples of efficient data structures and algorithms for implementing them are presented in detail in Sections 4.2, 4.3, and 4.4, respectively. Namely, we cover arrays, singly-and doubly-linked lists, heaps, search trees, (a, b)-**trees**, AVL-trees, **bucket arrays**, and hash tables.

4.2 Sequence

Introduction

A *sequence* is a container that stores elements in a certain order, which is imposed by the operations performed. The basic operations supported are:

- INSERTRANK: insert an element in a given position;
- REMOVE: remove an element.

Sequences are a basic form of data organization, and are typically used to realize and implement other data types and data structures.

Operations

Using locators (see "Containers, Elements, and Locators"), we can define a more complete repertory of operations for a sequence S:

SIZE(N) return the number of elements N of S;

HEAD(c) assign to c a locator to the first element of S; if S is empty, c is a null locator;

TAIL(c) assign to c a locator to the last element of S; if S is empty, a null locator is returned;

LOCATERANK(r, c) assign to c a locator to the r-th element of S; if $r < 1$ or $r > N$, where N is the size of S, c is a null locator;

PREV(c', c'') assign to c'' a locator to the element of S preceding the element with locator c'; if c' is the locator of the first element of S, c'' is a null locator;

NEXT(c', c'') assign to c'' a locator to the element of S following the element with locator c'; if c' is the locator of the last element of S, c'' is a null locator;

INSERTAFTER(e, c', c'') insert element e into S after the element with locator c', and return a locator c'' to e;

INSERTBEFORE(e, c', c'') insert element e into S before the element with locator c', and return a locator c'' to e;

INSERTHEAD(e, c) insert element e at the beginning of S, and return a locator c to e;

INSERTTAIL(e, c) insert element e at the end of S, and return a locator c to e;

INSERTRANK(e, r, c) insert element e in the r-th position of S; if $r < 1$ or $r > N + 1$, where N is the current size of S, c is a null locator;

REMOVE(c, e) remove from S and return element e with locator c;

MODIFY(c, e) replace with e the element with locator c.

Some of the above operations can be easily expressed by means of other operations of the repertory. For example, operations HEAD and TAIL can be easily expressed by means of LOCATERANK and SIZE.

Implementation with an Array

The simplest way to implement a sequence is to use a (one-dimensional) array, where the i-th element of the array stores the i-th element of the list, and to keep a variable that stores the size N of the sequence. With this implementation, accessing elements takes $O(1)$ time, while insertions and deletions take $O(N)$ time.

Table 4.1 shows the time complexity of the implementation of a sequence by means of an array.

TABLE 4.1 Performance of a Sequence Implemented with an Array

Operation	Time
SIZE	$O(1)$
HEAD	$O(1)$
TAIL	$O(1)$
LOCATERANK	$O(1)$
PREV	$O(1)$
NEXT	$O(1)$
INSERTAFTER	$O(N)$
INSERTBEFORE	$O(N)$
INSERTHEAD	$O(N)$
INSERTTAIL	$O(1)$
INSERTRANK	$O(N)$
REMOVE	$O(N)$
MODIFY	$O(1)$

Note: We denote with N the number of elements in the sequence at the time the operation is performed. The space complexity is $O(N)$.

Implementation with a Singly-Linked List

A sequence can also be implemented with a singly-linked list, where each element has a pointer to the next one. We also store the size of the sequence, and pointers to the first and last element of the sequence.

With this implementation, accessing elements takes $O(N)$ time, since we need to traverse the list, while some insertions and deletions take $O(1)$ time.

Table 4.2 shows the time complexity of the implementation of sequence by means of singly-linked list.

TABLE 4.2 Performance of a Sequence Implemented with a Singly-Linked List

Operation	Time
SIZE	$O(1)$
HEAD	$O(1)$
TAIL	$O(1)$
LOCATERANK	$O(N)$
PREV	$O(N)$
NEXT	$O(1)$
INSERTAFTER	$O(1)$
INSERTBEFORE	$O(N)$
INSERTHEAD	$O(1)$
INSERTTAIL	$O(1)$
INSERTRANK	$O(N)$
REMOVE	$O(N)$
MODIFY	$O(1)$

Note: We denote with N the number of elements in the sequence at the time the operation is performed. The space complexity is $O(N)$.

Implementation with a Doubly-Linked List

Better performance can be achieved, at the expense of using additional space, by implementing a sequence with a doubly-linked list, where each element has pointers to the next and previous elements. We also store the size of the sequence, and pointers to the first and last element of the sequence.

Table 4.3 shows the time complexity of the implementation of sequence by means of a doubly-linked list.

TABLE 4.3 Performance of a Sequence Implemented with a Doubly-Linked List

Operation	Time
SIZE	$O(1)$
HEAD	$O(1)$
TAIL	$O(1)$
LOCATERANK	$O(N)$
PREV	$O(1)$
NEXT	$O(1)$
INSERTAFTER	$O(1)$
INSERTBEFORE	$O(1)$
INSERTHEAD	$O(1)$
INSERTTAIL	$O(1)$
INSERTRANK	$O(N)$
REMOVE	$O(1)$
MODIFY	$O(1)$

Note: We denote with N the number of elements in the sequence at the time the operation is performed. The space complexity is $O(N)$.

4.3 Priority Queue

Introduction

A *priority queue* is a container of elements from a totally ordered universe that supports the following two basic operations:

- INSERT: insert an element into the priority queue;
- REMOVEMAX: remove the largest element from the priority queue.

Here are some simple applications of a priority queue:

Scheduling A scheduling system can store the tasks to be performed into a priority queue, and select the task with highest priority to be executed next.

Sorting To sort a set of N elements, we can insert them one at a time into a priority queue by means of N INSERT operations, and then retrieve them in decreasing order by means of N REMOVEMAX operations. This two-phase method is the paradigm of several popular sorting algorithms, including *Selection-Sort, Insertion-Sort,* and *Heap-Sort.*

Operations

Using locators, we can define a more complete repertory of operations for a priority queue Q:

SIZE(N) return the current number of elements N in Q;
MAX(c) return a locator c to the maximum element of Q;
INSERT(e, c) insert element e into Q and return a locator c to e;
REMOVE(c, e) remove from Q and return element e with locator c;
REMOVEMAX(e) remove from Q and return the maximum element e from Q;
MODIFY(c, e) replace with e the element with locator c.

Note that operation REMOVEMAX(e) is equivalent to MAX(c) followed by REMOVE(c, e).

Realization with a Sequence

We can realize a priority queue by reusing and extending the sequence abstract data type (see Section 4.2). Operations SIZE, MODIFY, and REMOVE correspond to the homonymous sequence operations.

Unsorted Sequence

We can realize INSERT by an INSERTHEAD or an INSERTTAIL, which means that the sequence is not kept sorted. Operation MAX can be performed by scanning the sequence with an iteration of NEXT operations, keeping track of the maximum element encountered. Finally, as observed above, operation REMOVEMAX is a combination of MAX and REMOVE. Table 4.4 shows the time complexity of this realization, assuming that the sequence is implemented with a doubly-linked list.

Sorted Sequence

An alternative implementation uses a sequence that is kept sorted. In this case, operation MAX corresponds to simply accessing the last element of the sequence. However, operation INSERT now requires scanning the sequence to find the appropriate position where to insert the new element. Table 4.5 shows the time complexity of this realization, assuming that the sequence is implemented with a doubly-linked list.

TABLE 4.4 Performance of a Priority Queue Realized by an Unsorted Sequence, Implemented with a Doubly-Linked List

Operation	Time
SIZE	$O(1)$
MAX	$O(N)$
INSERT	$O(1)$
REMOVE	$O(1)$
REMOVEMAX	$O(N)$
MODIFY	$O(1)$

Note: We Denote with N the number of elements in the priority queue at the time the operation is performed. The space complexity is $O(N)$.

TABLE 4.5 Performance of a Priority Queue Realized by a Sorted Sequence, Implemented with a Doubly-Linked List

Operation	Time
SIZE	$O(1)$
MAX	$O(1)$
INSERT	$O(N)$
REMOVE	$O(1)$
REMOVEMAX	$O(1)$
MODIFY	$O(N)$

Note: We denote with N the number of elements in the priority queue at the time the operation is performed. The space complexity is $O(N)$.

Realizing a priority queue with a sequence, sorted or unsorted, has the drawback that some operations require linear time in the worst case. Hence, this realization is not suitable in many applications where fast running times are sought for all the priority queue operations.

Sorting

For example, consider the sorting application (see "Introduction" in Section 4.3). We have a collection of N elements from a totally ordered universe, and we want to sort them using a priority queue Q. We assume that each element uses $O(1)$ space, and any two elements can be compared in $O(1)$ time. If we realize Q with an unsorted sequence, then the first phase (inserting the N elements into Q) takes $O(N)$ time. However the second phase (removing N times the maximum element) takes time:

$$O\left(\sum_{i=1}^{N} i\right) = O\left(N^2\right) .$$

Hence, the overall time complexity is $O(N^2)$. This sorting method is known as *Selection-Sort.*

However, if we realize the priority queue with a sorted sequence, then the first phase takes time:

$$O\left(\sum_{i=1}^{N} i\right) = O\left(N^2\right) ,$$

while the second phase takes time $O(N)$. Again, the overall time complexity is $O(N^2)$. This sorting method is known as *Insertion-Sort.*

Realization with a Heap

A more sophisticated realization of a priority queue uses a data structure called *heap.* A heap is a binary tree T whose internal nodes store each one element from a totally ordered universe, with the following properties (see Fig. 4.1):

Level Property: all the levels of T are full, except possibly for the bottommost level, which is left-filled;

Partial Order Property: let μ be a node of T distinct from the root, and let ν be the parent of μ; then the element stored at μ is less than or equal to the element stored at ν.

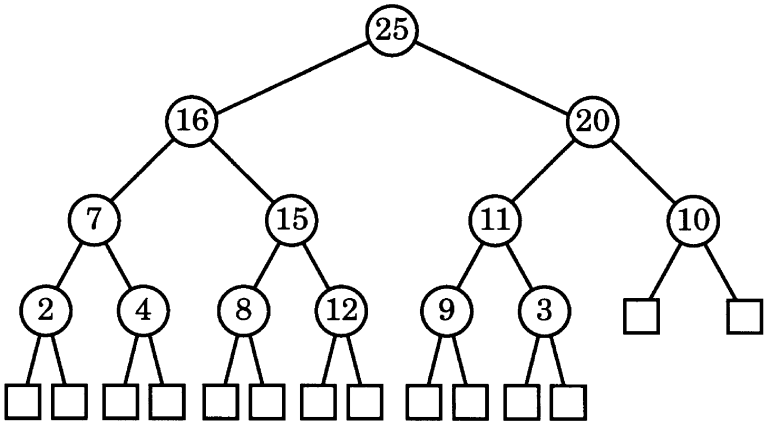


FIGURE 4.1 Example of a heap storing 13 elements.

The leaves of a heap do not store data and serve only as "placeholders." The level property implies that heap T is a minimum-height binary tree. More precisely, if T stores N elements and has height h, then each level i with $0 \leq i \leq h-2$ stores exactly 2^i elements, while level $h-1$ stores between 1 and 2^{h-1} elements. Note that level h contains only leaves. We have

$$2^{h-1} = 1 + \sum_{i=0}^{h-2} 2^i \leq N \leq \sum_{i=0}^{h-1} 2^i = 2^h - 1 \,,$$

from which we obtain

$$\log_2(N+1) \leq h \leq 1 + \log_2 N \,.$$

Now, we show how to perform the various priority queue operations by means of a heap T. We denote with $x(\mu)$ the element stored at an internal node μ of T. We denote with ρ the root of T. We call *last node* of T the rightmost internal node of the bottommost internal level of T.

By storing a counter that keeps track of the current number of elements, SIZE consists of simply returning the value of the counter. By the partial order property, the maximum element is stored at the root, and hence, operation MAX can be performed by accessing node ρ.

Operation INSERT

To insert an element e into T, we add a new internal node μ to T such that μ becomes the new last node of T, and set $x(\mu) = e$. This action ensures that the level property is satisfied, but may violate

the partial-order property. Hence, if $\mu \neq \rho$, we compare $x(\mu)$ with $x(\nu)$, where ν is the parent of μ. If $x(\mu) > x(\nu)$, then we need to restore the partial order property, which can be locally achieved by exchanging the elements stored at μ and ν. This causes the new element e to move up one level. Again, the partial order property may be violated, and we may have to continue moving up the new element e until no violation occurs. In the worst case, the new element e moves up to the root ρ of T by means of $O(\log N)$ exchanges. The upward movement of element e by means of exchanges is conventionally called *upheap.*

An example of a sequence of insertions into a heap is shown in Fig. 4.2.

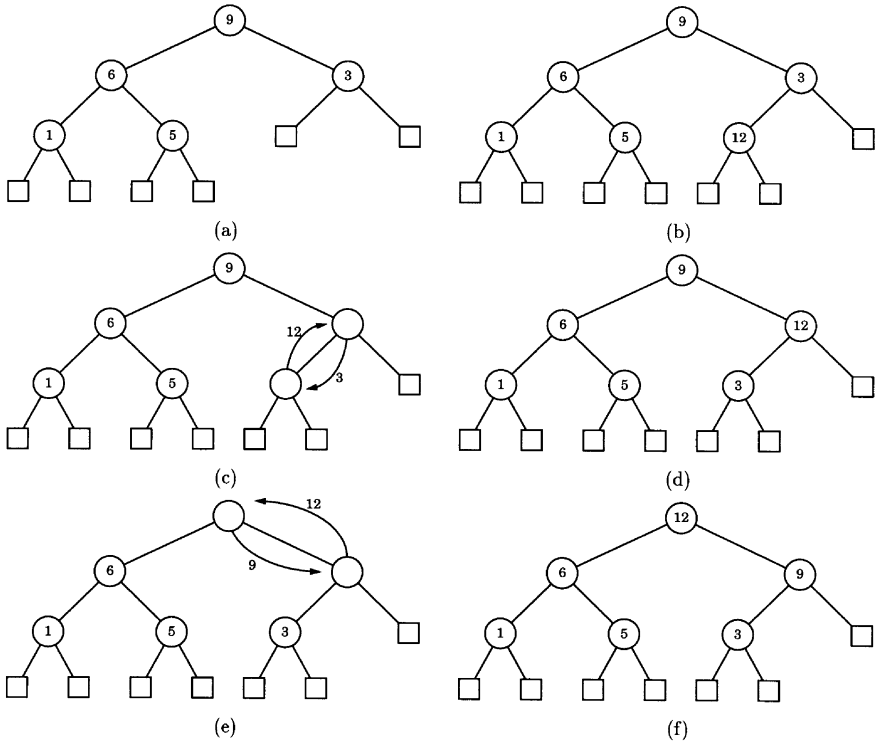


FIGURE 4.2 Example of insertion into a heap.

Operation REMOVEMAX

To remove the maximum element, we cannot simply delete the root of T, because this would disrupt the binary tree structure. Instead, we access the last node λ of T, copy its element e to the root by setting $x(\rho) = x(\lambda)$, and delete λ. We have preserved the level property, but we may have violated the partial order property. Hence, if ρ has at least one nonleaf child, we compare $x(\rho)$ with the maximum element $x(\sigma)$ stored at a child of ρ. If $x(\rho) < x(\sigma)$, then we need to restore the partial order property, which can be locally achieved by exchanging the elements stored at ρ and σ. Again, the partial order property may be violated, and we continue moving down element e until no violation occurs. In the worst case, element e moves down to the bottom internal level of T by means of $O(\log N)$ exchanges. The downward movement of element e by means of exchanges is conventionally called *downheap.*

An example of operation REMOVEMAX in a heap is shown in Fig. 4.3.

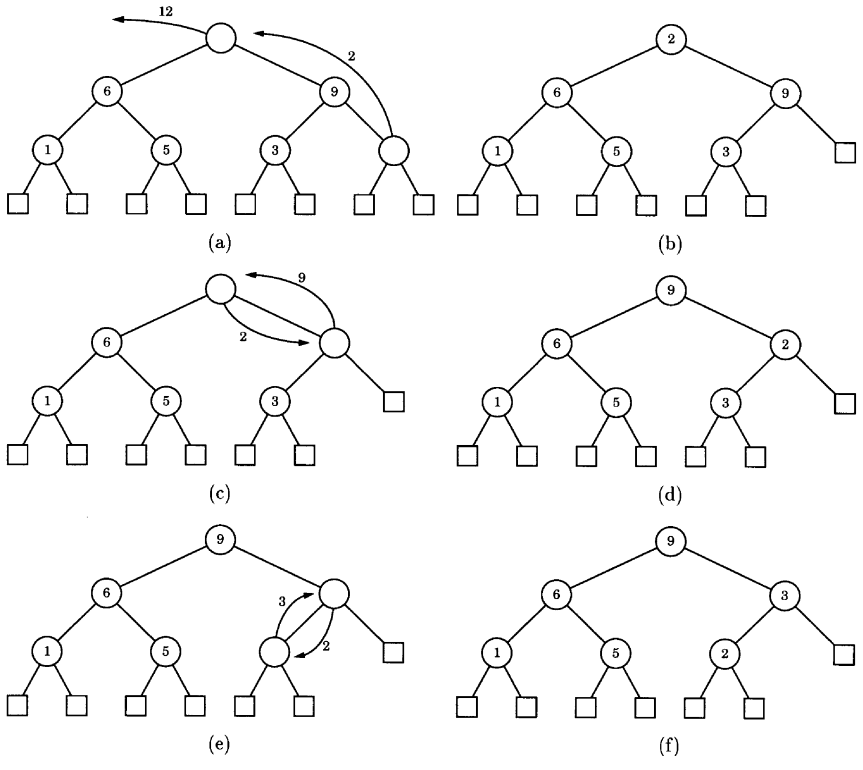


FIGURE 4.3 REMOVEMAX operation in a heap.

Operation REMOVE

To remove an arbitrary element of heap T, we cannot simply delete its node μ, because this would disrupt the binary tree structure. Instead, we proceed as before and delete the last node of T after copying to μ its element e. We have preserved the level property, but we may have violated the partial order property, which can be restored by performing either upheap or downheap.

Finally, after modifying an element of heap T, if the partial order property is violated, we just need to perform either upheap or downheap.

Time Complexity

Table 4.6 shows the time complexity of the realization of a priority queue by means of a heap. We assume that the heap is itself realized by a data structure for binary trees that supports $O(1)$-time access to the children and parent of a node. For instance, we can implement the heap explicitly with a linked structure (with pointers from a node to its parents and children), or implicitly with an array (where node i has children $2i$ and $2i + 1$).

Let N the number of elements in a priority queue Q realized with a heap T at the time an operation is performed. The time bounds of Table 4.6 are based on the following facts:

- In the worst case, the time complexity of upheap and downheap is proportional to the height of T.
- If we keep a pointer to the last node of T, we can update this pointer in time proportional to the height of T in operations INSERT, REMOVE, and REMOVEMAX, as illustrated in Fig. 4.4.
- The height of heap T is $O(\log N)$.

TABLE 4.6 Performance of a Priority Queue Realized by a Heap, Implemented with a Suitable Binary Tree Data Structure

Operation	Time
SIZE	$O(1)$
MAX	$O(1)$
INSERT	$O(\log N)$
REMOVE	$O(\log N)$
REMOVEMAX	$O(\log N)$
MODIFY	$O(\log N)$

Note: We denote with N the number of elements in the priority queue at the time the operation is performed. The space complexity is $O(N)$.

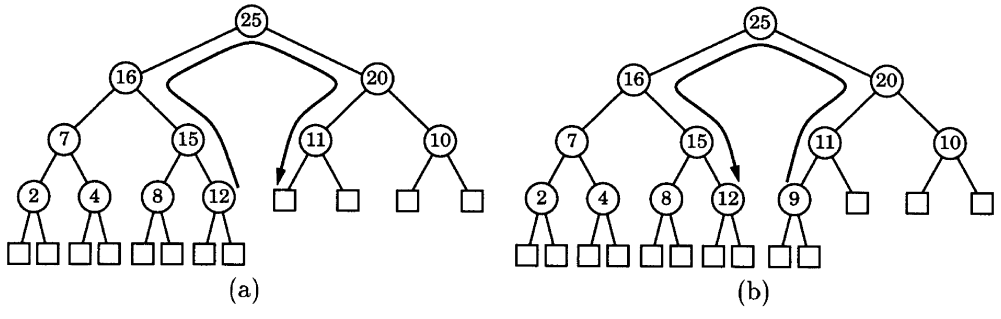


FIGURE 4.4 Update of the pointer to the last node: (a) INSERT; (b) REMOVE or REMOVEMAX.

The $O(N)$ space complexity bound for the heap is based on the following facts:

- The heap has $2N + 1$ nodes (N internal nodes and $N + 1$ leaves).
- Every node uses $O(1)$ space.
- In the array implementation, because of the level property the array elements used to store heap nodes are in the contiguous locations 1 through $2N - 1$.

Note that we can reduce the space requirement by a constant factor implementing the leaves of the heap with null objects, such that only the internal nodes have space associated with them.

Sorting

Realizing a priority queue with a heap has the advantage that all the operations take $O(\log N)$ time, where N is the number of elements in the priority queue at the time the operation is performed. For example, in the sorting application (see "Introduction" in Section 4.3), both the first phase (inserting the N elements) and the second phase (removing N times the maximum element) take time:

$$O\left(\sum_{i=1}^{N} \log i\right) = O(N \log N) .$$

Hence, sorting with a priority queue realized with a heap takes $O(N \log N)$ time. This sorting method is known as *Heap-Sort,* and its performance is considerably better than that of *Selection-Sort* and *Insertion-Sort* (see "Sorting" in Section 4.3), where the priority queue is realized as a sequence.

Realization with a Dictionary

A priority queue can be easily realized with a dictionary (see Section 4.4). Indeed, all the operations in the priority queue repertory are supported by a dictionary. To achieve $O(1)$ time for operation MAX, we can store the locator of the maximum element in a variable, and recompute it after an update operations. This realization of a priority queue with a dictionary has the same asymptotic complexity bounds as the realization with a heap, provided the dictionary is suitably implemented, e.g., with an (a, b)-tree (see "Realization with an (a, b)-Tree") or an AVL-tree (see "Realization with an AVL-Tree"). However, a heap is simpler to program than an (a, b)-tree or an AVL-tree.

4.4 Dictionary

A *dictionary* is a container of elements from a totally ordered universe that supports the following basic operations:

- FIND: search for an element;
- INSERT: insert an element;
- REMOVE: delete an element.

A major application of dictionaries are database systems.

Operations

In the most general setting, the elements stored in a dictionary are pairs (x, y), where x is the *key* giving the ordering of the elements, and y is the auxiliary information. For example, in a database storing student records, the key could be the student's last name, and the auxiliary information the student's transcript. It is convenient to augment the ordered universe of keys with two *special keys:* $+\infty$ and $-\infty$, and assume that each dictionary has, in addition to its *regular elements,* two *special elements,* with keys $+\infty$ and $-\infty$, respectively. For simplicity, we shall also assume that no two elements of a dictionary have the same key. An insertion of an element with the same key as that of an existing element will be rejected by returning a null locator.

Using locators (see "Containers, Elements, and Locators"), we can define a more complete repertory of operations for a dictionary D:

SIZE(N) return the number of regular elements N of D;

FIND(x, c) if D contains an element with key x, assign to c a locator to such an element, otherwise set c equal to a null locator;

LOCATEPREV(x, c) assign to c a locator to the element of D with the largest key less than or equal to x; if x is smaller than all the keys of the regular elements, c is a locator the special element with key $-\infty$; if $x = -\infty$, c is a null locator;

LOCATENEXT(x, c) assign to c a locator to the element of D with the smallest key greater than or equal to x; if x is larger than all the keys of the regular elements, c is a locator to the special element with key $+\infty$; if $x = +\infty$, c is a null locator;

LOCATERANK(r, c) assign to c a locator to the r-th element of D; if $r < 1$, c is a locator to the special element with key $-\infty$; if $r > N$, where N is the size of D, c is a locator to the special element with key $+\infty$;

PREV(c', c'') assign to c'' a locator to the element of D with the largest key less than that of the element with locator c'; if the key of the element with locator c' is smaller than all the keys of the regular elements, this operation returns a locator to the special element with key $-\infty$;

NEXT(c', c'') assign to c'' a locator to the element of D with the smallest key larger than that of the element with locator c'; if the key of the element with locator c' is larger than all the keys of the regular elements, this operation returns a locator to the special element with key $+\infty$;

MIN(c) assign to c a locator to the regular element of D with minimum key; if D has no regular elements, c is a null locator;

MAX(c) assign to c a locator to the regular element of D with maximum key; if D has no regular elements, c is null a locator;

INSERT(e, c) insert element e into D, and return a locator c to e; if there is already an element with the same key as e, this operation returns a null locator;

REMOVE(c, e) remove from D and return element e with locator c;

MODIFY(c, e) replace with e the element with locator c.

Some of the above operations can be easily expressed by means of other operations of the repertory. For example, operation FIND is a simple variation of LOCATEPREV or LOCATENEXT; MIN and MAX are special cases of LOCATERANK, or can be expressed by means of PREV and NEXT.

Realization with a Sequence

We can realize a dictionary by reusing and extending the sequence abstract data type (see Section 4.2). Operations SIZE, INSERT, and REMOVE correspond to the homonymous sequence operations.

Unsorted Sequence

We can realize INSERT by an INSERTHEAD or an INSERTTAIL, which means that the sequence is not kept sorted. Operation FIND(x, c) can be performed by scanning the sequence with an iteration of NEXT operations, until we either find an element with key x, or we reach the end of the sequence. Table 4.7 shows the time complexity of this realization, assuming that the sequence is implemented with a doubly-linked list.

TABLE 4.7 Performance of a Dictionary Realized by an Unsorted Sequence, Implemented with a Doubly-Linked List

Operation	Time
SIZE	$O(1)$
FIND	$O(N)$
LOCATEPREV	$O(N)$
LOCATENEXT	$O(N)$
LOCATERANK	$O(N)$
NEXT	$O(N)$
PREV	$O(N)$
MIN	$O(N)$
MAX	$O(N)$
INSERT	$O(1)$
REMOVE	$O(1)$
MODIFY	$O(1)$

Note: We denote with N the number of elements in the dictionary at the time the operation is performed.

Sorted Sequence

We can also use a sorted sequence to realize a dictionary. Operation INSERT now requires scanning the sequence to find the appropriate position where to insert the new element. However, in a FIND operation, we can stop scanning the sequence as soon as we find an element with a key larger than the search key. Table 4.8 shows the time complexity of this realization by a sorted sequence, assuming that the sequence is implemented with a doubly-linked list.

TABLE 4.8 Performance of a Dictionary Realized by a Sorted Sequence, Implemented with a Doubly-Linked List

Operation	Time
SIZE	$O(1)$
FIND	$O(N)$
LOCATEPREV	$O(N)$
LOCATENEXT	$O(N)$
LOCATERANK	$O(N)$
NEXT	$O(1)$
PREV	$O(1)$
MIN	$O(1)$
MAX	$O(1)$
INSERT	$O(N)$
REMOVE	$O(1)$
MODIFY	$O(N)$

Note: We denote with N the number of elements in the dictionary at the time the operation is performed. The space complexity is $O(N)$.

Sorted Array

We can obtain a different performance trade-off by implementing the sorted sequence by means of an array, which allows constant-time access to any element of the sequence given its position. Indeed, with this realization we can speed up operation FIND(x, c) using the *binary search* strategy, as follows. If the dictionary is empty, we are done. Otherwise, let N be the current number of elements in the dictionary. We compare the search key k with the key x_m of the middle element of the sequence, i.e., the element at position $\lfloor N/2 \rfloor$. If $x = x_m$, we have found the element. Else, we recursively search in the subsequence of the elements preceding the middle element if $x < x_m$, or following the middle element if $x > x_m$. At each recursive call, the number of elements of the subsequence being searched halves. Hence, the number of sequence elements accessed and the number of comparisons performed by binary search is $O(\log N)$. While searching takes $O(\log N)$ time, inserting or deleting elements now takes $O(N)$ time.

Table 4.9 shows the performance of a dictionary realized with a sorted sequence, implemented with an array.

Realization with a Search Tree

A *search tree* for elements of the type (x, y), where x is a key from a totally ordered universe, is a rooted ordered tree T such that

TABLE 4.9 Performance of a Dictionary Realized by a Sorted Sequence, Implemented with an Array

Operation	Time
SIZE	$O(1)$
FIND	$O(\log N)$
LOCATEPREV	$O(\log N)$
LOCATENEXT	$O(\log N)$
LOCATERANK	$O(1)$
NEXT	$O(1)$
PREV	$O(1)$
MIN	$O(1)$
MAX	$O(1)$
INSERT	$O(N)$
REMOVE	$O(N)$
MODIFY	$O(N)$

Note: We denote with N the number of elements in the dictionary at the time the operation is performed. The space complexity is $O(N)$.

- Each internal node of T has at least two children and stores a nonempty set of elements;
- A node μ of T with d children $\mu_1, \cdots, \mu_d$ stores $d-1$ elements $(x_1, y_1) \cdots (x_{d-1}, y_{d-1})$, where $x_1 \leq \cdots \leq x_{d-1}$;
- For each element (x, y) stored at a node in the subtree of T rooted at μ_i, we have $x_{i-1} \leq x \leq x_i$, where $x_0 = -\infty$ and $x_d = +\infty$.

In a search tree, each internal node stores a nonempty collection of keys, while the leaves do not store any key and serve only as "placeholders." An example of search tree is shown in Fig. 4.5(a). A special type of search tree is a **binary search tree**, where each internal node stores one key and has two children.

We will recursively describe the realization of a dictionary D by means of a search tree T, since we will use dictionaries to implement the nodes of T. Namely, an internal node μ of T with children $\mu_1, \cdots, \mu_d$ and elements $(x_1, y_1) \cdots (x_{d-1}, y_{d-1})$ is equipped with a dictionary $D(\mu)$ whose regular elements are the pairs $(x_i, (y_i, \mu_i))$, $i = 1, \cdots, d-1$ and whose special element with key $+\infty$ is $(+\infty, (\cdot, \mu_d))$. A regular element (x, y) stored in D is associated with a regular element $(x, (y, v))$ stored in a dictionary $D(\mu)$, for some node μ of T. See the example in Fig. 4.5(b).

Operation FIND

Operation FIND(x, c) on dictionary D is performed by means of the following recursive method for a node μ of T, where μ is initially the root of T [see Fig. 4.5(b)]. We execute LOCATENEXT(x, c') on dictionary $D(\mu)$ and let $(x', (y', v))$ be the element pointed by the returned locator c'. We have three cases:

- $x = x'$: we have found x and return locator c to (x', y');
- $x \neq x'$ and v is a leaf: we have determined that x is not in D and return a null locator c;
- $x \neq x'$ and v is an internal node: we set $\mu = v$ and recursively execute the method.

Operation INSERT

Operations LOCATEPREV, LOCATENEXT, and INSERT can be performed with small variations of the above method. For example, to perform operation INSERT(e, c), where $e = (x, y)$, we modify the

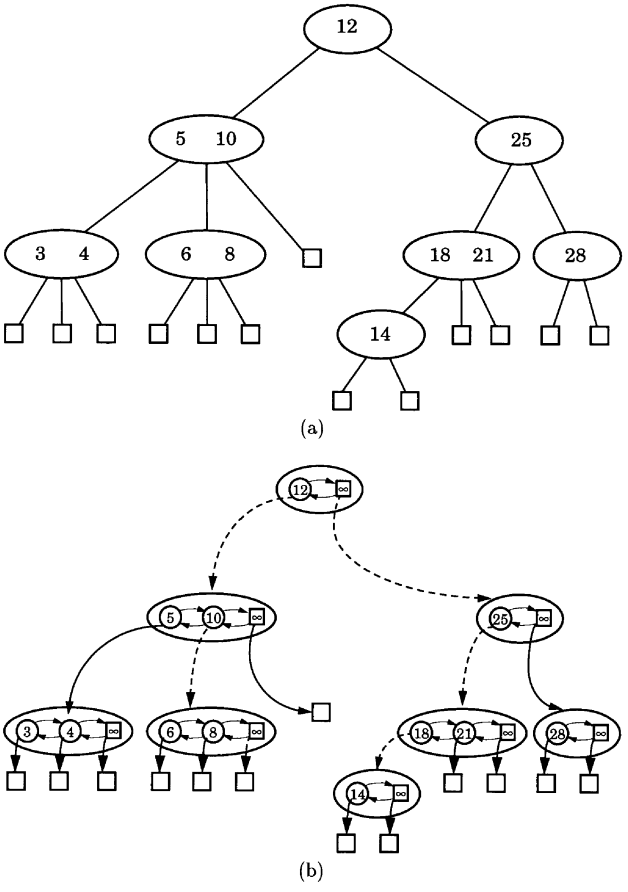


FIGURE 4.5 Realization of a dictionary by means of a search tree. (a) A search tree T. (b) Realization of the dictionaries at the nodes of T by means of sorted sequences. The search paths for elements 9 (unsuccessful search) and 14 (successful search) are shown with dashed lines.

above cases as follows (see Fig. 4.6):

- $x = x'$: an element with key x already exists, and we return a null locator;
- $x \neq x'$ and v is a leaf: we create a new leaf node λ, insert a new element $(x, (y, \lambda))$ into $D(\mu)$, and return a locator c to (x, y).
- $x \neq x'$ and v is an internal node: we set $\mu = v$ and recursively execute the method.

Note that new elements are inserted at the "bottom" of the search tree.

Operation REMOVE

Operation REMOVE(e, c) is more complex (see Fig. 4.7). Let the associated element of $e = (x, y)$ in T be $(x, (y, v))$, stored in dictionary $D(\mu)$ of node μ.

- If node v is a leaf, we simply delete element $(x, (y, v))$ from $D(\mu)$.
- Else (v is an internal node), we find the successor element $(x', (y', v'))$ of $(x, (y, v))$ in $D(\mu)$ with a NEXT operation in $D(\mu)$.
  - If v' is a leaf, we replace v' with v, i.e., change element $(x', (y', v'))$ to $(x', (y', v))$, and delete element $(x, (y, v))$ from $D(\mu)$.

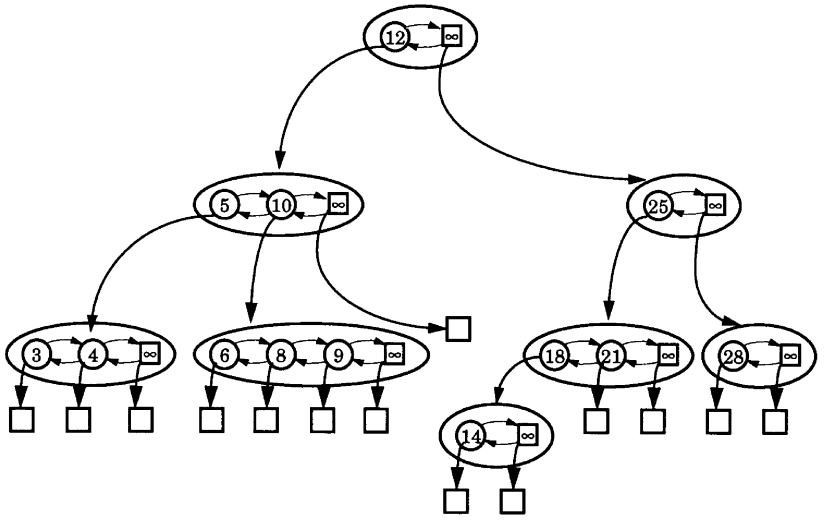


FIGURE 4.6 Insertion of element 9 into the search tree of Fig. 4.5.

– Else (v' is an internal node), while the leftmost child v'' of v' is not a leaf, we set $v' = v''$. Let $(x'', (y'', v''))$ be the first element of $D(v')$ (node v'' is a leaf). We replace $(x, (y, v))$ with $(x'', (y'', v))$ in $D(\mu)$ and delete $(x'', (y'', v''))$ from $D(v')$.

The above actions may cause dictionary $D(\mu)$ or $D(v')$ to become empty. If this happens, say for $D(\mu)$ and μ is not the root of T, we need to remove node μ. Let $(+\infty, (\cdot, \kappa))$ be the special element of $D(\mu)$ with key $+\infty$, and let $(z, (w, \mu))$ be the element pointing to μ in the parent node π of μ. We delete node μ and replace $(z, (w, \mu))$ with $(z, (w, \kappa))$ in $D(\pi)$.

Note that, if we start with an initially empty dictionary, a sequence of insertions and deletions performed with the above methods yields a search tree with a single node. In the next sections, we show how to avoid this behavior by imposing additional conditions on the structure of a search tree.

Realization with an (a, b)-Tree

An (a, b)*-tree,* where a and b are integer constants such that $2 \leq a \leq (b+1)/2$, is a search tree T with the following additional restrictions:

Level Property: all the levels of T are full, i.e., all the leaves are at the same depth;

Size Property: let μ be an internal node of T, and d be the number of children of μ; if μ is the root of T, then $d \geq 2$, else $a \leq d \leq b$.

The height of an (a, b) tree storing N elements is $O(\log_a N) = O(\log N)$. Indeed, in the worst case, the root has two children, and all the other internal nodes have a children.

The realization of a dictionary with an (a, b)-tree extends that with a search tree. Namely, the implementation of operations INSERT and REMOVE need to be modified in order to preserve the level and size properties. Also, we maintain the current size of the dictionary, and pointers to the minimum and maximum regular elements of the dictionary.

Insertion

The implementation of operation INSERT for search trees given in "Operation INSERT" adds a new element to the dictionary $D(\mu)$ of an existing node μ of T. Since the structure of the tree is not changed, the level property is satisfied. However, if $D(\mu)$ had the maximum allowed size $b-1$ before insertion

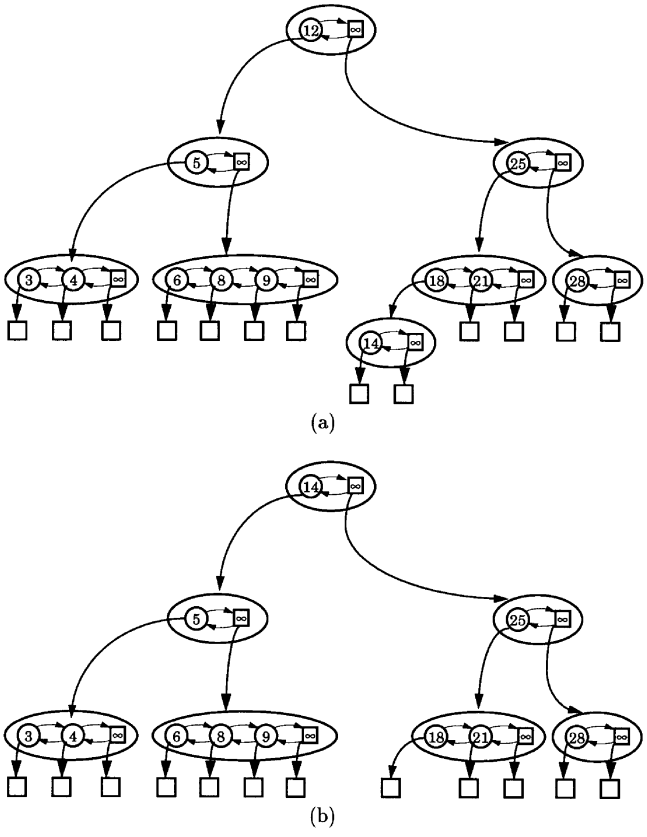


FIGURE 4.7 (a) Deletion of element 10 from the search tree of Fig. 4.6. (b) Deletion of element 12 from the search tree of part a.

(recall that the size of $D(\mu)$ is one less than the number of children of μ), the size property is violated at μ because $D(\mu)$ has now size b. To remedy this *overflow* situation, we perform the following *node-split* (see Fig. 4.8):

- Let the special element of $D(\mu)$ be $(+\infty, (\cdot, \mu_{b+1}))$. Find the median element of $D(\mu)$, i.e., the element $e_i = (x_i, (y_i, \mu_i))$ such that $i = \lceil (b+1)/2 \rceil$.
- Split $D(\mu)$ into:
  - dictionary D', containing the $\lceil (b-1)/2 \rceil$ regular elements $e_j = (x_j, (y_j, \mu_j))$, $j = 1 \cdots i-1$ and the special element $(+\infty, (\cdot, \mu_i))$;
  - element e; and
  - dictionary D'', containing the $\lfloor (b-1)/2 \rfloor$ regular elements $e_j = (x_j, (y_j, \mu_j))$, $j = i+1 \cdots b$ and the special element $(+\infty, (\cdot, \mu_{b+1}))$.
- Create a new tree node κ, and set $D(\kappa) = D'$. Hence, node κ has children $\mu_1 \cdots \mu_i$.
- Set $D(\mu) = D''$. Hence, node μ has children $\mu_{i+1} \cdots \mu_{b+1}$.
- If μ is the root of T, create a new node π with an empty dictionary $D(\pi)$. Else, let π be the parent of μ.
- Insert element $(x_i, (y_i, \kappa))$ into dictionary $D(\pi)$.

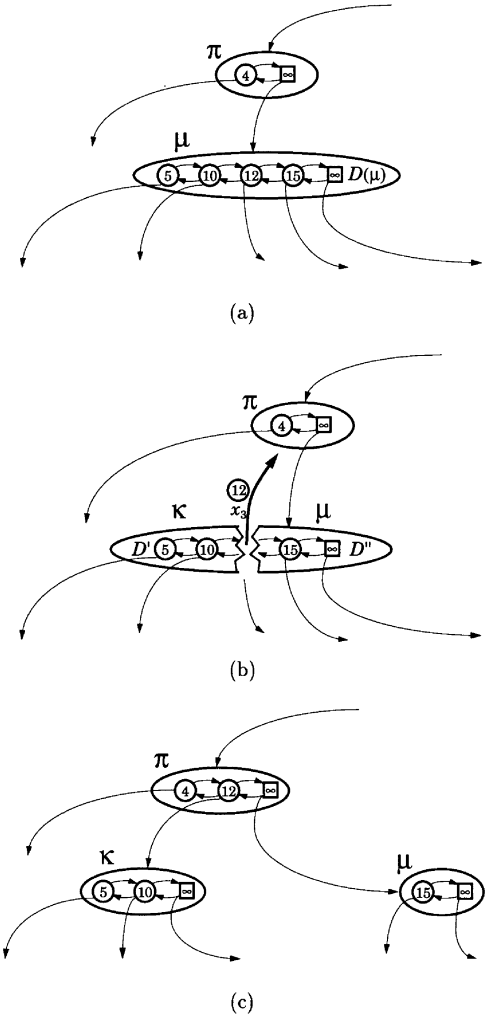


FIGURE 4.8 Example of node-split in a 2-4 tree: (a) initial configuration with an overflow at node μ; (b) split of the node μ into μ' and μ'' and insertion of the median element into the parent node π; (c) final configuration.

After a node-split, the level property is still verified. Also, the size property is verified for all the nodes of T, except possibly for node π. If π has $b + 1$ children, we repeat the node-split for $\mu = \pi$. Each time we perform a node-split, the possible violation of the size property appears at a higher level in the tree. This guarantees the termination of the algorithm for the INSERT operation. We omit the description of the simple method for updating the pointers to the minimum and maximum regular elements.

Deletion

The implementation of operation REMOVE for search trees given in the subsection "REMOVE" removes an element from the dictionary $D(\mu)$ of an existing node μ of T. Since the structure of the tree is not changed, the level property is satisfied. However, if μ is not the root, and $D(\mu)$ had the minimum allowed size $a - 1$ before deletion (recall that the size of the dictionary is one less than the number of children of the node), the size property is violated at μ because $D(\mu)$ has now size $a - 2$. To remedy this *underflow* situation, we perform the following *node-merge* (see Figs. 4.9 and 4.10):

- If μ has a right sibling, let μ'' be the right sibling of μ and $\mu' = \mu$; else, let μ' be the left sibling of μ and $\mu'' = \mu$. Let $(+\infty, (\cdot, \nu))$ be the special element of $D(\mu')$.
- Let π be the parent of μ' and μ''. Remove from $D(\pi)$ the regular element $(x, (y, \mu'))$ associated with μ'.
- Create a new dictionary D containing the regular elements of $D(\mu')$ and $D(\mu'')$, regular element $(x, (y, \nu))$, and the special element of $D(\mu'')$.
- Set $D(\mu'') = D$, and destroy node μ'.
- If μ'' has more than b children, perform a node-split at μ''.

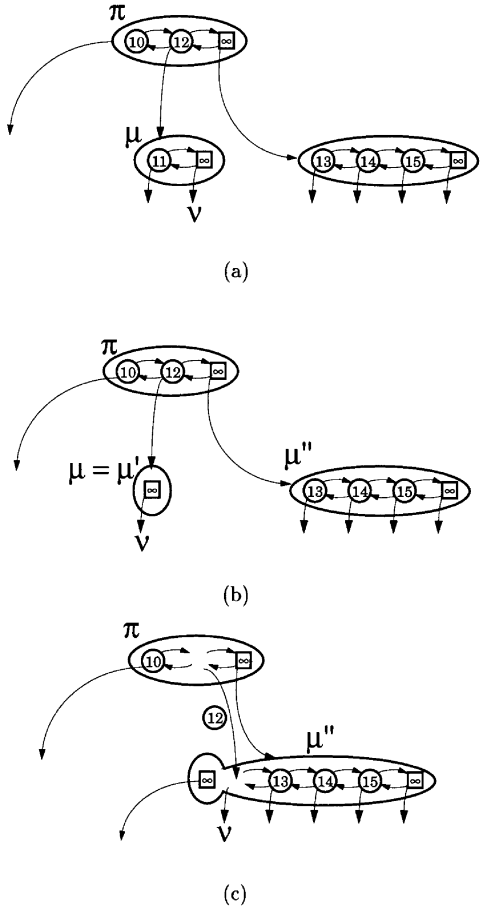


FIGURE 4.9 Example of node merge in a 2-4 tree: (a) initial configuration; (b) the removal of an element from dictionary $D(\mu)$ causes an underflow at node μ; (c) merging node $\mu = \mu'$ into its sibling μ''.

After a node-merge, the level property is still verified. Also, the size property is verified for all the nodes of T, except possibly for node π. If π is the root and has one child (and thus, an empty dictionary), we remove node π. If π is not the root and has fewer than $a - 1$ children, we repeat the node-merge for $\mu = \pi$. Each time we perform a node-merge, the possible violation of the size property appears at a higher level in the tree. This guarantees the termination of the algorithm for the REMOVE operation. We omit the description of the simple method for updating the pointers to the minimum and maximum regular elements.

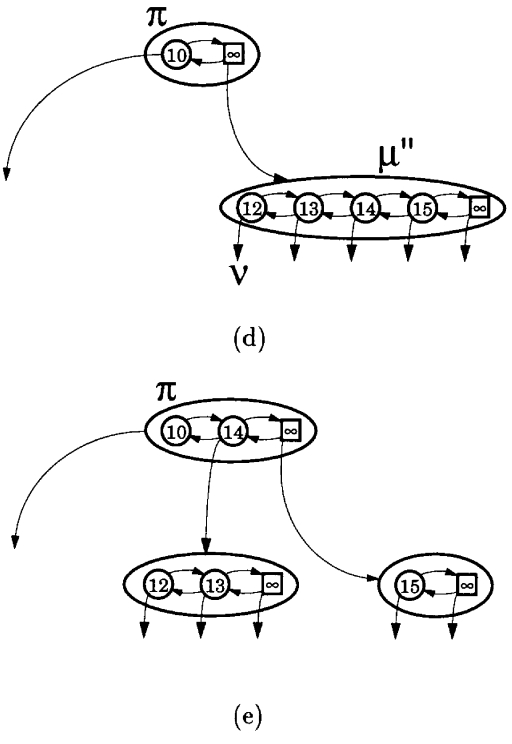


FIGURE 4.10 Example of node merge in a 2-4 tree: (d) overflow at node μ''; (e) final configuration after splitting node μ''.

Complexity

Let T be an (a, b)-tree storing N elements. The height of T is $O(\log_a N) = O(\log N)$. Each dictionary operation affects only the nodes along a root-to-leaf path. We assume that the dictionaries at the nodes of T are realized with sequences. Hence, processing a node takes $O(b) = O(1)$ time. We conclude that each operation takes $O(\log N)$ time.

Table 4.10 shows the performance of a dictionary realized with an (a, b)-tree.

Realization with an AVL-tree

An *AVL-tree* is a search tree T with the following additional restrictions:

Binary Property: T is a binary tree, i.e., every internal node has two children, (left and right child), and stores one key.

Height-Balance Property: For every internal node μ, the heights of the subtrees rooted at the children of μ differ at most by one.

An example of AVL-tree is shown in Fig. 4.11. The height of an AVL-tree storing N elements is $O(\log N)$. This can be shown as follows. Let N_h be the minimum number of elements stored in an AVL-tree of height h. We have $N_0 = 0$, $N_1 = 1$, and

$$N_h = 1 + N_{h-1} + N_{h-2}, \text{ for } h \geq 2 .$$

The above recurrence relation defines the well-known Fibonacci numbers. Hence, $N_h = \Omega(\phi^N)$, where $1 < \phi < 2$.

TABLE 4.10 Performance of a Dictionary Realized by an (a, b)-Tree

Operation	Time
SIZE	$O(1)$
FIND	$O(\log N)$
LOCATEPREV	$O(\log N)$
LOCATENEXT	$O(\log N)$
LOCATERANK	$O(\log N)$
NEXT	$O(\log N)$
PREV	$O(\log N)$
MIN	$O(1)$
MAX	$O(1)$
INSERT	$O(\log N)$
REMOVE	$O(\log N)$
MODIFY	$O(\log N)$

Note: We denote with N the number of elements in the dictionary at the time the operation is performed. The space complexity is $O(N)$.

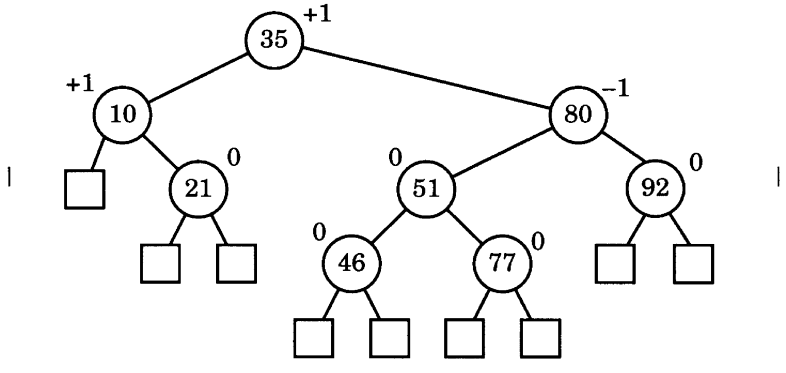


FIGURE 4.11 Example of AVL-tree storing 9 elements. The keys are shown inside the nodes, and the balance factors (see Section "Rebalancing") are shown next to the nodes.

The realization of a dictionary with an AVL-tree extends that with a search tree. Namely, the implementation of operations INSERT and REMOVE need to be modified in order to preserve the binary and height-balance properties after an insertion or deletion.

Insertion

The implementation of INSERT for search trees given in the subsection "Operation INSERT" adds the new element to an existing node. This violates the binary property, and hence, cannot be done in an AVL-tree. Hence, we modify the three cases of the INSERT algorithm for search trees as follows:

- $x = x'$: an element with key x already exists, and we return a null locator c;
- $x \neq x'$ and v is a leaf: we replace v with a new internal node κ with two leaf children, store element (x, y) in κ, and return a locator c to (x, y).
- $x \neq x'$ and v is an internal node: we set $\mu = v$ and recursively execute the method.

We have preserved the binary property. However, we may have violated the height-balance property, since the heights of some subtrees of T have increased by one. We say that a node is balanced if the

difference between the heights of its subtrees is -1, 0, or 1, and is unbalanced otherwise. The unbalanced nodes form a (possibly empty) subpath of the path from the new internal node κ to the root of T. See the example of Fig. 4.12.

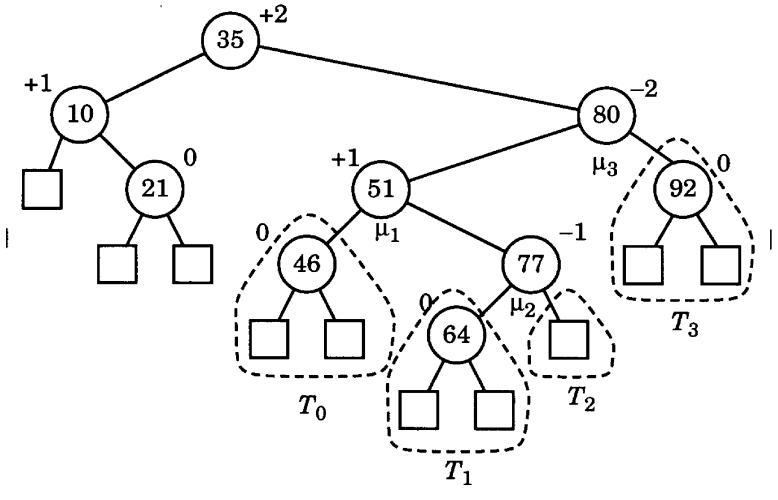


FIGURE 4.12 Insertion of an element with key 64 into the AVL-tree of Fig. 4.11. Note that two nodes (with balance factors $+2$ and -2) have become unbalanced. The dashed lines identify the subtrees that participate in the rebalancing, as illustrated in Fig. 4.14.

Rebalancing

To restore the height-balance property, we *rebalance* the lowest node μ that is unbalanced, as follows.

- Let μ' be the child of μ whose subtree has maximum height, and μ'' be the child of μ' whose subtree has maximum height.
- Let (μ_1, μ_2, μ_3) be the left-to-right ordering of nodes $\{\mu, \mu', \mu''\}$, and (T_0, T_1, T_2, T_3) be the left-to-right ordering of the four subtrees of $\{\mu, \mu', \mu''\}$ not rooted at a node in $\{\mu, \mu', \mu''\}$.
- replace the subtree rooted at μ with a new subtree rooted at μ_2, where μ_1 is the left child of μ_2 and has subtrees T_0 and T_1, and μ_3 is the right child of μ_2 and has subtrees T_2 and T_3.

Two examples of rebalancing are schematically shown in Fig. 4.14. Other symmetric configurations are possible. In Fig. 4.13, we show the rebalancing for the tree of Fig. 4.12.

Note that the rebalancing causes all the nodes in the subtree of μ_2 to become balanced. Also, the subtree rooted at μ_2 now has the same height as the subtree rooted at node μ before insertion. This causes all the previously unbalanced nodes to become balanced. To keep track of the nodes that become unbalanced, we can store at each node a *balance factor,* which is the difference of the heights of the left and right subtrees. A node becomes unbalanced when its balance factor becomes $+2$ or -2. It is easy to modify the algorithm for operation INSERT such that it maintains the balance factors of the nodes.

Deletion

The implementation of REMOVE for search trees given in "Realization with a Search Tree" preserves the binary property, but may cause the height-balance property to be violated. After deleting a node, there can be only one unbalanced node, on the path from the deleted node to the root of T.

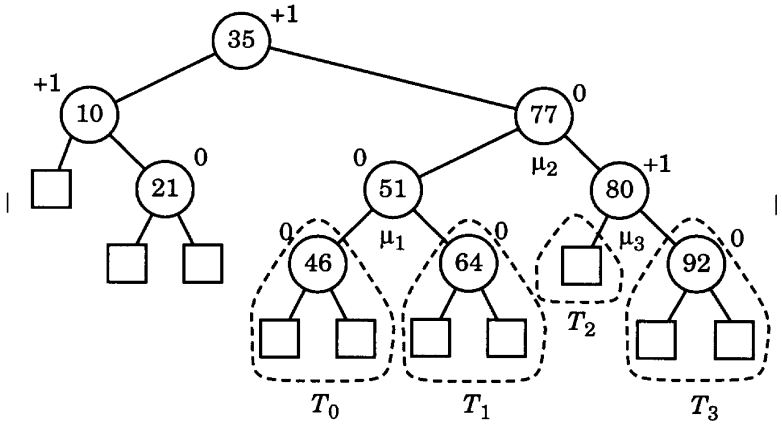


FIGURE 4.13 AVL-tree obtained by rebalancing the lowest unbalanced node in the tree of Fig. 4.11. Note that all the nodes are now balanced. The dashed lines identify the subtrees that participate in the rebalancing, as illustrated in Fig. 4.14.

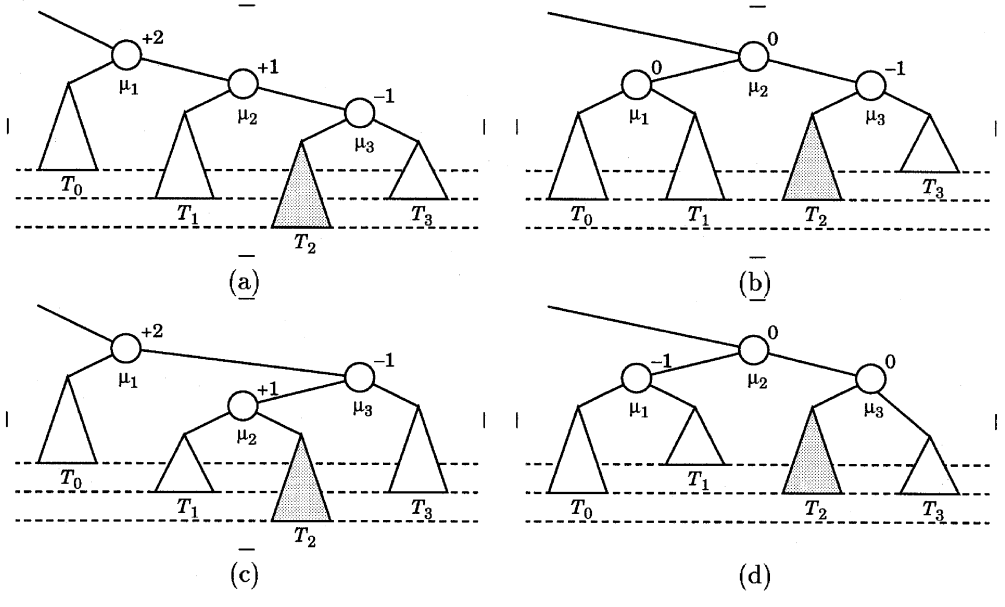


FIGURE 4.14 Schematic illustration of rebalancing a node in the INSERT algorithm for AVL-trees. The shaded subtree is the one where the new element was inserted. (a, b) Rebalancing by means of a "single rotation." (c, d) Rebalancing by means of a "double rotation."

To restore the height-balance property, we *rebalance* the unbalanced node using the above algorithm. Notice, however, that the choice of μ'' may not be unique, since the subtrees of μ' may have the same height. In this case, the height of the subtree rooted at μ_2 is the same as the height of the subtree rooted at μ before rebalancing, and we are done. If instead the subtrees of μ' do not have the same height, then the height of the subtree rooted at μ_2 is one less than the height of the subtree rooted at μ before rebalancing. This may cause an ancestor of μ_2 to become unbalanced, and we repeat the rebalancing step. Balance factors are used to keep track of the nodes that become unbalanced, and can be easily maintained by the REMOVE algorithm.

Complexity

Let T be an AVL-tree storing N elements. The height of T is $O(\log N)$. Each dictionary operation affects only the nodes along a root-to-leaf path. Rebalancing a node takes $O(1)$ time. We conclude that each operation takes $O(\log N)$ time.

Table 4.11 shows the performance of a dictionary realized with an AVL-tree.

TABLE 4.11 Performance of a Dictionary Realized by an AVL-Tree

Operation	Time
SIZE	$O(1)$
FIND	$O(\log N)$
LOCATEPREV	$O(\log N)$
LOCATENEXT	$O(\log N)$
LOCATERANK	$O(\log N)$
NEXT	$O(\log N)$
PREV	$O(\log N)$
MIN	$O(1)$
MAX	$O(1)$
INSERT	$O(\log N)$
REMOVE	$O(\log N)$
MODIFY	$O(\log N)$

Note: We Denote with N the number of elements in the dictionary at the time the operation is performed. The space complexity is $O(N)$.

Realization with a Hash Table

The previous realizations of a dictionary make no assumptions on the structure of the keys, and use comparisons between keys to guide the execution of the various operations.

Bucket Array

If the keys of a dictionary D are integers in the range $[1, M]$, we can implement D with a *bucket array* B. An element (x, y) of D is represented by setting $B[x] = y$. If an integer x is not in D, the location $B[x]$ stores a null value. In this implementation, we allocate a "bucket" for every possible element of D.

Table 4.12 shows the performance of a dictionary realized a bucket array.

The bucket array method can be extended to keys that are easily mapped to integers; e.g., three-letter airport codes can be mapped to the integers in the range $[1, 26^3]$.

Hashing

The bucket array method works well when the range of keys is small. However, it is inefficient when the range of keys is large. To overcome this problem, we can use a *hash function* h that maps the keys of the original dictionary D into integers in the range $[1, M]$, where M is a parameter of the hash function. Now, we can apply the bucket array method using the *hashed value* $h(x)$ of the keys. In general, a *collision* may happen, where two distinct keys x_1 and x_2 have the same hashed value, i.e., $x_1 \neq x_2$ and $h(x_1) = h(x_2)$. Hence, each bucket must be able to accommodate a collection of elements.

TABLE 4.12 Performance of a Dictionary Realized by Bucket Array

Operation	Time
SIZE	$O(1)$
FIND	$O(1)$
LOCATEPREV	$O(M)$
LOCATENEXT	$O(M)$
LOCATERANK	$O(M)$
NEXT	$O(M)$
PREV	$O(M)$
MIN	$O(M)$
MAX	$O(M)$
INSERT	$O(1)$
REMOVE	$O(1)$
MODIFY	$O(1)$

Note: The keys in the dictionary are integers in the range $[1, M]$. The space complexity is $O(M)$.

A hash table of size M for a function $h(x)$ is a bucket array B of size M (primary structure) whose entries are dictionaries (secondary structures), such that element (x, y) is stored in the dictionary $B[h(x)]$. For simplicity of programming, the dictionaries used as secondary structures are typically realized with sequences. An example of hash table is shown in Fig. 4.15.

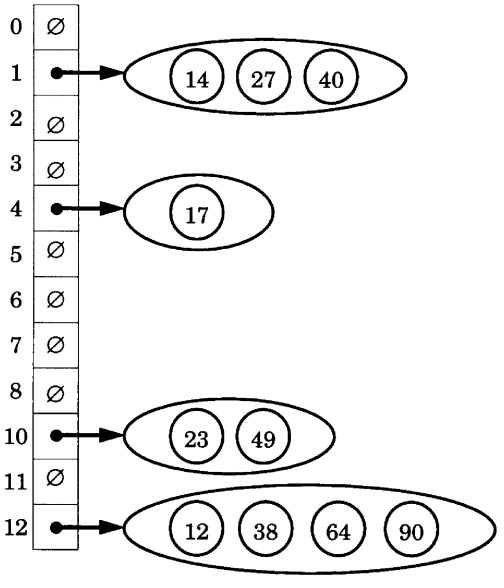


FIGURE 4.15 Example of hash table of size 13 storing 10 elements. The hash function is $h(x) = x \mod 13$.

If all the elements in the dictionary D collide, they are all stored in the same dictionary of the bucket array, and the performance of the hash table is the same as that of the kind of dictionary used as a secondary structures. At the other end of the spectrum, if no two elements of the dictionary D collide, they are stored

in distinct one-element dictionaries of the bucket array, and the performance of the hash table is the same as that of a bucket array.

A typical hash function for integer keys is $h(x) = x \bmod M$. The size M of the hash table is usually chosen as a prime number. An example of hash table is shown in Fig. 4.15.

It is interesting to analyze the performance of a hash table from a probabilistic viewpoint. If we assume that the hashed values of the keys are uniformly distributed in the range $[1, M]$, then each bucket holds on average N/M keys, where N is the size of the dictionary. Hence, when $N = O(M)$, the average size of the secondary data structures is $O(1)$.

Table 4.13 shows the performance of a dictionary realized a hash table. Both the worst-case and average time complexity in the above probabilistic model are indicated.

TABLE 4.13 Performance of a Dictionary Realized by a Hash Table of Size M

	Time	
Operation	**Worst-Case**	**Average**
SIZE	$O(1)$	$O(1)$
FIND	$O(N)$	$O(N/M)$
LOCATEPREV	$O(N+M)$	$O(N+M)$
LOCATENEXT	$O(N+M)$	$O(N+M)$
LOCATERANK	$O(N+M)$	$O(N+M)$
NEXT	$O(N+M)$	$O(N+M)$
PREV	$O(N+M)$	$O(N+M)$
MIN	$O(N+M)$	$O(N+M)$
MAX	$O(N+M)$	$O(N+M)$
INSERT	$O(1)$	$O(1)$
REMOVE	$O(1)$	$O(1)$
MODIFY	$O(1)$	$O(1)$

Note: We denote with N the number of elements in the dictionary at the time the operation is performed. The space complexity is $O(N+M)$. The average time complexity refers to a probabilistic model where the hashed values of the keys are uniformly distributed in the range $[1, M]$.

4.5 Defining Terms

(a, b)-tree: Search tree with additional properties (each node has between a and b children, and all the levels are full); see "Realization with an (a, b)-Tree."

Abstract data type: Mathematically specified data type equipped with operations that can be performed on the objects; see "Abstract Data Types."

AVL-tree: Binary search tree such that the subtrees of each node have heights that differ by at most one; see "Realization with an AVL-Tree."

Binary search tree: Search tree such that each internal node has two children; see "Realization with a Search Tree."

Bucket array: Implementation of a dictionary by means of an array indexed by the keys of the dictionary elements; see "Bucket Array."

Container: Abstract data type storing a collection of objects (elements); see "Containers, Elements, and Locators."

Dictionary: Container storing elements from a sorted universe supporting searches, insertions, and deletions; see Section 4.4.

Hash table: Implementation of a dictionary by means of a bucket array storing secondary dictionaries; see "Hashing."

Heap: Binary tree with additional properties storing the elements of a priority queue; see "Realization with a Heap."

Locator: Variable that allows to access an object stored in a container; see "Containers, Elements, and Locators."

Priority queue: Container storing elements from a sorted universe supporting finding the maximum element, insertions, and deletions; see Section 4.3.

Search tree: Rooted ordered tree with additional properties storing the elements of a dictionary; see "Realization with a Search Tree."

Sequence: Container storing object in a certain order, supporting insertions (in a given position) and deletions; see Section 4.2.

References

[1] Aggarwal, A. and Vitter, J.S., The input/output complexity of sorting and related problems. *Commun. ACM,* 31, 1116–1127, 1988.

[2] Aho, A.V., Hopcroft, J.E., and Ullman J.D., *Data Structures and Algorithms.* Addison-Wesley, Reading, MA, 1983.

[3] Chazelle, B. and Guibas, L.J., Fractional cascading: I. A data structuring technique. *Algorithmica,* 1, 133–162, 1986.

[4] Chiang, Y.-J. and Tamassia, R., Dynamic algorithms in computational geometry. *Proc. IEEE,* 80(9), 1412–1434, Sep 1992.

[5] Cohen, R.F. and Tamassia, R., Dynamic expression trees. *Algorithmica,* 13, 245–265, 1995.

[6] Comer, D., The ubiquitous B-tree. *ACM Comput. Surv.,* 11, 121–137, 1979.

[7] Cormen, T.H., Leiserson, C.E., and Rivest, R.L., *Introduction to Algorithms.* The MIT Press, Cambridge, MA, 1990.

[8] Di Battista, G. and Tamassia, R., On-line maintenance of triconnected components with SPQR-trees, *Algorithmica,* 15, 302–318, 1996.

[9] Driscoll, J.R., Sarnak, N., Sleator, D.D., and Tarjan, R.E., Making data structures persistent. *J. Comput. Syst. Sci.,* 38, 86–124, 1989.

[10] Edelsbrunner, H., *Algorithms in Combinatorial Geometry,* volume 10 of *EATCS Monographs on Theoretical Computer Science.* Springer-Verlag, Heidelberg, West Germany, 1987.

[11] Eppstein, D., Galil, Z., Italiano, G.F., and Nissenzweig, A., Sparsification: A technique for speeding up dynamic graph algorithms. In *Proc. 33rd. Annu. IEEE Sympos. Found. Comput. Sci.,* 60–69, 1992.

[12] Even, S., *Graph Algorithms.* Computer Science Press, Potomac, MD, 1979.

[13] Foley, J.D., van Dam, A., Feiner, S.K., and Hughes, J.F., *Computer Graphics: Principles and Practice.* Addison-Wesley, Reading, MA, 1990.

[14] Frederickson, G.N., A data structure for dynamically maintaining rooted trees. In *Proc. 4th ACM-SIAM Symp. Discrete Algorithms,* 175–184, 1993.

[15] Galil, Z. and Italiano, G.F., Data structures and algorithms for disjoint set union problems. *ACM Computing Surveys,* 23(3), 319–344, 1991.

[16] Gonnet, G.H. and Baeza-Yates, R., *Handbook of Algorithms and Data Structures.* Addison-Wesley, Reading, MA, 1991.
[17] Goodrich, M.T. and Tamassia, R., *Data Structures and Algorithms in Java.* John Wiley & Sons, New York, 1998.
[18] Hoffmann, K., Mehlhorn, K., Rosenstiehl, P., and Tarjan, R.E., Sorting Jordan sequences in linear time using level-linked search trees. *Inform. Control,* 68, 170–184, 1986.
[19] Horowitz, E., Sahni, S., and Metha, D., *Fundamentals of Data Structures in C + +*. Computer Science Press, 1995.
[20] Knuth, D.E., *Fundamental Algorithms,* volume 1 of *The Art of Computer Programming.* Addison-Wesley, Reading, MA, 1968.
[21] Knuth, D.E., *Sorting and Searching,* volume 3 of *The Art of Computer Programming.* Addison-Wesley, Reading, MA, 1973.
[22] Lewis, H.R. and Denenberg, L., *Data Structures and Their Algorithms.* Harper Collins, 1991.
[23] Mehlhorn, K., *Data Structures and Algorithms.* Volumes 1–3. Springer-Verlag, 1984.
[24] Mehlhorn, K. and Näer, S., LEDA; a platform for combinatorial and geometric computing. *CACM,* 38, 96–102, 1995.
http://www.mpi-sb.mpg.de/guide/staff/uhrig/leda.html.
[25] Mehlhorn, K. and Tsakalidis, A., Data structures. In *Algorithms and Complexity,* volume A of *Handbook of Theoretical Computer Science.* J. van Leeuwen, Ed., Elsevier, Amsterdam, 1990.
[26] Miltersen, P.B., Sairam, S., Vitter, J.S., and Tamassia, R., Complexity models for incremental computation. *Theoret. Comput. Sci.,* 130, 203–236, 1994.
[27] Nievergelt, J. and Hinrichs, K.H., *Algorithms and Data Structures: With Applications to Graphics and Geometry.* Prentice-Hall, Englewood Cliffs, NJ, 1993.
[28] O'Rourke, J., *Computational Geometry in C.* Cambridge University Press, 1994.
[29] Overmars, M.H., *The design of dynamic data structures,* volume 156 of *Lecture Notes in Computer Science.* Springer-Verlag, 1983.
[30] Preparata, F.P. and Shamos, M.I., *Computational Geometry: An Introduction.* Springer-Verlag, New York, 1985.
[31] Pugh, W., Skip lists: a probabilistic alternative to balanced trees. *Commun. ACM,* 35, 668–676, 1990.
[32] Reif, J.H., A topological approach to dynamic graph connectivity, *Inform. Process. Lett.,* 25, 65–70, 1987.
[33] Sedgewick, R., *Algorithms in C++*. Addison-Wesley, Reading, MA, 1992.
[34] Sleator, D.D. and Tarjan, R.E., A data structure for dynamic tress. *J. Comput. Syst. Sci.,* 26(3), 362–381, 1983.
[35] Tarjan, R.E., *Data Structures and Network Algorithms,* volume 44 of *CBMS-NSF Regional Conference Series in Applied Mathematics.* Society for Industrial Applied Matematics, 1983.
[36] Vitter, J.S. and Flajolet, P., Average-case analysis of algorithms and data structures. In *Algorithms and Complexity,* volume A of *Handbook of Theoretical Computer Science,* J. van Leeuwen, Ed., 431–524. Elsevier, Amsterdam, 1990.
[37] Wood, D., *Data Structures, Algorithms, and Performance.* Addison-Wesley, Reading, MA, 1993.

Further Information

Many textbooks and monographs have been written on data structures, e.g., [2, 7, 16, 17, 19, 20, 21, 22, 23, 27, 29, 30, 33, 35, 37].

Recent papers surveying the state-of-the art in data structures include [4, 15, 25, 36].

The LEDA project [24] aims at developing a C++ library of efficient and reliable implementations of sophisticated data structures.

5

Topics in Data Structures

Giuseppe F. Italiano
University Venezia, "Ca' Foscari" di Venezia

Rajeev Raman
King's College, London

5.1 Introduction

In this chapter, we describe advanced data structures and algorithmic techniques, mostly focusing our attention to two important problems: set union and persistence. We first describe set union data structures. Their discovery required a new set of techniques and tools that have proved useful in other areas as well. We survey algorithms and data structures for set union problems, and attempt to provide a unifying theoretical framework for this growing body of algorithmic tools. **Persistent data structures** maintain information about their past states and find uses in a diverse spectrum of applications. The body of work relating to persistent data structures brings together quite a surprising cocktail of techniques, from real-time computation to techniques from functional programming.

Set Union Data Structures

The *set union problem* consists of maintaining a collection of disjoint sets under an intermixed sequence of the following two kinds of operations:

union(A, B)*:* Combine the two sets A and B into a new set named A.
find(x)*:* Return the name of the set containing element x.

The operations are presented on line, namely each operation must be processed before the next one is known. Initially, the collection consists of n singleton sets $\{1\}, \{2\}, \ldots, \{n\}$, and the name of set $\{i\}$ is i, $1 \leq i \leq n$. Figure 5.1 illustrates an example of set union operations.

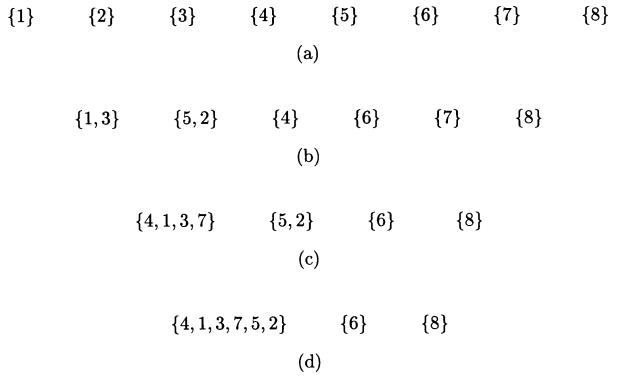


FIGURE 5.1 Examples of set union operations. (a) The initial collection of disjoint sets. (b) The disjoint sets of (a) after performing union(1, 3) and union(5, 2). (c) The disjoint sets of (b) after performing union(1, 7) followed by union(4, 1). (d) The disjoint sets of (c) after performing union(4, 5).

The set union problem has been widely studied, and finds application in a wide range of areas, including Fortran compilers [10, 38], property grammars [78, 79], computational geometry [49, 67, 68], finite state machines [4, 44], string algorithms [5, 48], logic programming and theorem proving [7, 8, 47, 95], and several combinatorial problems such as finding minimum spanning trees [4, 53], solving dynamic edge- and vertex-connectivity problems [98], computing least common ancestors in trees [3], solving off–line minimum problems [34, 45], finding dominators in graphs [83], and checking flow graph reducibility [82].

Several variants of set union have been introduced, in which the possibility of backtracking over the sequences of unions was taken into account [9, 39, 59, 63, 97]. This was motivated by problems arising in logic programming interpreter memory management [40, 60, 61, 96].

Persistent Data Structures

Data structures that one encounters in traditional algorithmic settings are *ephemeral*, i.e., if the data structure is updated then the previous state of the data structure is lost. A *persistent* data structure, on the other hand, preserves old versions of the data structure. Several kinds of persistence can be distinguished based upon what kind of access is allowed to old versions of the data structure. Accesses to a data structure can be of two kinds: *updates*, which change the information content of the data structure, and *queries*, which do not. For the sake of ease of presentation, we will assume that queries do not even change the internal representation of the data, i.e., read-only access to a data structure suffices to answer a query.

In the persistent setting we would like to maintain multiple *versions* of data structures. In addition to the arguments taken by its ephemeral counterparts, a persistent query or update operation takes as

an argument the version of the data structure to which the query or update refers. A persistent update also returns a *handle* to the new version created by the update. We distinguish between three kinds of persistence:

- A *partially* persistent data structure allows updates only to the latest version of the data structure. All versions of the data structure may be queried, however. Clearly, the versions of a partially persistent data structure exhibit a linear ordering as shown in Fig. 5.2(a).
- A *fully* persistent data structure allows all existing versions of the data structure to be queried or updated. However, an update may operate only on a single version at a time—for instance combining two or more old versions of the data structure to form a new one is not allowed. The versions of a fully persistent data structure form a tree, as shown in Fig. 5.2(b).
- A **purely functional language** is one that does not allow any *destructive* operation—one that overwrites data—such as the assignment operation. Purely functional languages are side-effect-free, i.e., invoking a function has no effect other than computing the value returned by the function. In particular, an update operation to a data structure implemented in a purely functional language returns a new data structure containing the updated values, while leaving the original data structure unchanged. Data structures implemented in purely functional languages are therefore persistent in the strongest possible sense, as they allow unrestricted access for both reading and updating all versions of the data structure.
  An example of a purely functional language is pure LISP [64]. Side-effect-free code can also be written in *functional* languages such as ML [70], most existing variants of LISP (e.g., Common LISP [80]) or Haskell [46], by eschewing the destructive operations supported by these languages.

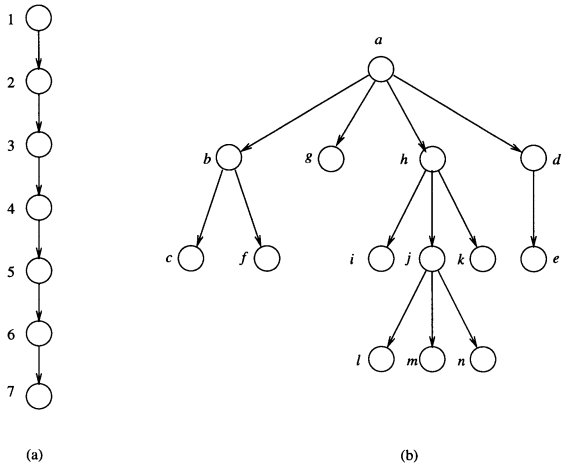


FIGURE 5.2 Structure of versions for (a) partial and (b) full persistence.

This section aims to cover a selection of the major results relating to the above forms of persistence. The body of work contains both *ad hoc* techniques for creating persistent data structures for particular problems, as well as general techniques to make ephemeral data structures persistent. Indeed, early work on persistence [17, 20, 30] focused almost exclusively on the former. Sarnak [75] and Driscoll et al. [28] were the first to offer very efficient general techniques for partial and full persistence. These and related results will form the bulk of the material in this chapter dealing with partial and full persistence. However, the prospect of obtaining still greater efficiency led to the further development of some *ad hoc* persistent

data structures [25, 26, 41]. The results on functional data structures will largely focus on implementations of individual data structures.

There has also been some research into data structures that support *backtrack* or *rollback* operations, whereby the data structure can be reset to some previous state. We do not cover these operations in this section, but we note that fully persistent data structures support backtracking (although sometimes not as efficiently as data structures designed especially for backtracking). Data structures with backtracking for the union–find problem are covered in Section 5.4.

Persistent data structures have numerous applications, including text, program and file editing and maintenance, computational geometry, tree pattern matching and inheritance in object-oriented programming languages. One elegant application of partially persistent search trees to the classical geometric problem of *planar point location* was given by Sarnak and Tarjan [76]. Suppose the Euclidean plane is divided into polygons by a collection of n line segments that intersect only at their endpoints (see Fig. 5.3), and we want to preprocess the collection of line segments so that, given a query point p, we can efficiently determine the polygon to which p belongs. Sarnak and Tarjan achieve this by combining the well-known *plane sweep* technique with a persistent data structure.

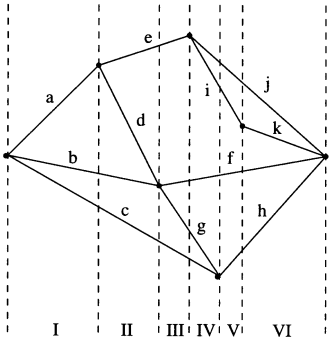


FIGURE 5.3 A planar subdivision.

Imagine moving an infinite vertical line (called the *sweep line*) from left to right across the plane, beginning at the leftmost endpoint of any line segment. As the sweep line moves, we maintain the line segments currently intersecting the sweep line in a balanced binary search tree, in order of their point of intersection with the sweep line (i.e., of two line segments, the one that intersects the sweep line at a higher location is considered smaller). Figure 5.4 shows the evolution of the search tree as the sweep line continues its progress from left to right. Note that the plane is divided into vertical *slabs*, within which the search tree does not change.

Given a query point p, we first locate the slab in which the x-coordinate of p lies. If we could remember what our search tree looked like while the sweep line was in this slab, we could query the search tree using the y-coordinate of p to find the two segments immediately above and below p in this slab; these line segments uniquely determine the polygon in which p lies. However, if we maintained the line segments in a partially persistent search tree as the sweep line moves from left to right, all incarnations of the search tree during this process are available for queries.

Sarnak and Tarjan show that it is possible to perform the preprocessing (which merely consists of building up the persistent tree) in $O(n \log n)$ time. The data structure uses $O(n)$ space and can be queried in $O(\log n)$ time, giving a simple optimal solution to the planar point location problem.

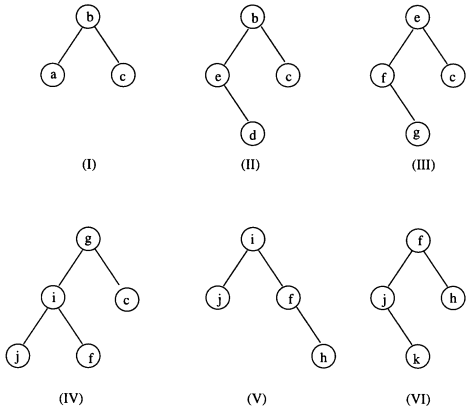


FIGURE 5.4 The evolution of the search tree during the plane sweep.

Models of Computation

Different models of computation have been developed for analyzing data structures. One model of computation is the **random access machine,** whose memory consists of an unbounded sequence of registers, each of which is capable of holding an integer. In this model, arithmetic operations are allowed to compute the address of a memory register. Usually, it is assumed that the size of a register is bounded by $O(\log n)$[1] bits, where n is the input problem size. A more formal definition of random access machines can be found in [4]. Another model of computation, known as the **cell probe model** of computation, was introduced by Yao [99]. In the cell probe, the cost of a computation is measured by the total number of memory accesses to a random access memory with $\lceil \log n \rceil$ bits cell size. All other computations are not accounted for and are considered to be free. Note that the cell probe model is more general than a random access machine, and thus, is more suitable for proving lower bounds. A third model of computation is the **pointer machine** [13, 54, 55, 77, 85]. Its storage consists of an unbounded collection of registers (or records) connected by pointers. Each register can contain an arbitrary amount of additional information but no arithmetic is allowed to compute the address of a register. The only possibility to access a register is by following pointers. This is the main difference between random access machines and pointer machines. Throughout this chapter, we use the terms *random-access algorithms*, *cell-probe algorithms*, and *pointer-based algorithms* to refer to algorithms respectively for random access machines, the cell probe model, and pointer machines.

Among pointer-based algorithms, two different classes were defined specifically for set union problems: *separable pointer algorithms* [85] and *nonseparable pointer algorithms* [69].

Separable pointer algorithms run on a pointer machine and satisfy the **separability** assumption as defined in [85] (see below). A separable pointer algorithm makes use of a linked data structure, namely a collection of records and pointers that can be thought of as a directed graph: each record is represented by a node and each pointer is represented by an edge in the graph. The algorithm solves the set union problem according to the following rules [14, 85]:

(i) The operations must be performed on line, i.e., each operation must be executed before the next one is known.

(ii) Each element of each set is a node of the data structure. There can be also additional (working) nodes.

[1] Throughout this chapter all logarithms are assumed to be to the base 2, unless explicitly otherwise specified.

(iii) (*Separability*). After each operation, the data structure can be partitioned into disjoint subgraphs such that each subgraph corresponds to exactly one current set. The name of the set occurs in exactly one node in the subgraph. No edge leads from one subgraph to another.

(iv) To perform find(x), the algorithm obtains the node v corresponding to element x and follows paths starting from v until it reaches the node which contains the name of the corresponding set.

(v) During any operation the algorithm may insert or delete any number of edges. The only restriction is that rule (iii) must hold after each operation.

The class of *nonseparable pointer algorithms* [69] does not require the separability assumption. The only requirement is that the number of edges leaving each node must be bounded by some constant $c > 0$. More formally, rule (iii) above is replaced by the following rule, while the other four rules are left unchanged:

(iii) There exists a constant $c > 0$ such that there are at most c edges leaving a node.

As we will see later on, often separable and nonseparable pointer-based algorithms admit quite different upper and lower bounds for the same problems.

5.2 The Set Union Problem

As defined in the previous section, the *set union problem* consists of performing a sequence of union and find operations, starting from a collection of n singleton sets $\{1\}, \{2\}, \ldots, \{n\}$. The initial name of set $\{i\}$ is i. As there are at most n items to be united, the number of unions in any sequence of operations is bounded above by $(n - 1)$. There are two invariants which hold at any time for the set union problem: first, the sets are always disjoint and define a partition of $\{1, 2, \ldots, n\}$; second, the name of each set corresponds to one of the items contained in the set itself. Both invariants are trivial consequences of the definition of union and find operations.

A different version of this problem considers the following operation in place of unions:

unite(A,B): Combine the two sets A and B into a new set, whose name is either A or B.

The only difference between union and unite is that unite allows the name of the new set to be arbitrarily chosen (e.g., at run time by the algorithm). This is not a significant restriction in many applications, where one is mostly concerned with testing whether two elements belong to the same set, no matter what the name of the set can be. However, some extensions of the set union problem have quite different time bounds depending on whether unions or unites are considered. In the following, we will deal with unions unless explicitly specified otherwise.

Amortized Time Complexity

In this section we describe algorithms for the set union problem [84, 89] giving the optimal amortized time complexity per operation. We only mention here that the amortized time is the running time per operation averaged over a worst-case sequence of operations, and refer the interested reader to [88] for a more detailed definition of amortized complexity. For the sake of completeness, we first survey some of the basic algorithms that have been proposed in the literature [4, 31, 38]. These are: the *quick-find*, the *weighted quick-find*, the *quick-union*, and the *weighted quick-union* algorithms. The quick-find algorithm performs find operations quickly, while the quick-union algorithm performs union operations quickly. Their weighted counterparts speed these computations up by introducing some weighting rules during union operations.

Most of these algorithms represent sets as rooted trees, following a technique introduced first by Galler and Fischer [38]. There is a tree for each disjoint set, and nodes of a tree correspond to elements of the

corresponding set. The name of the set is stored in the tree root. Each tree node has a pointer to its parent: in the following, we refer to $p(x)$ as the parent of node x.

The quick-find algorithm can be described as follows. Each set is represented by a tree of height 1. Elements of the set are the leaves of the tree. The root of the tree is a special node which contains the name of the set. Initially, singleton set $\{i\}$, $1 \leq i \leq n$, is represented by a tree of height 1 composed of one leaf and one root. To perform a union(A, B), all the leaves of the tree corresponding to B are made children of the root of the tree corresponding to A. The old root of B is deleted. This maintains the invariant that each tree is of height 1 and can be performed in $O(|B|)$ time, where $|B|$ denotes the total number of elements in set B. Since a set can have as many as $O(n)$ elements, this gives an $O(n)$ time complexity in the worst case for each union. To perform a find(x), return the name stored in the parent of x. Since all trees are maintained of height 1, the parent of x is a tree root. Consequently a find requires $O(1)$ time.

A more efficient variant attributed to McIlroy and Morris (see [4]) and known as weighted quick-find uses the freedom implicit in each union operation according to the following weighting rule.

> *Union by size:* Make the children of the root of the smaller tree point to the root of the larger, arbitrarily breaking a tie. This requires that the size of each tree is maintained throughout any sequence of operations.

Although this rule does not improve the worst-case time complexity of each operation, it improves to $O(\log n)$ the amortized bound of a union (see, e.g., [4]).

The quick-union algorithm [38] can be described as follows. Again, each set is represented by a tree. However, there are two main differences with the data structure used by the quick-find algorithm. The first is that now the height of a tree can be greater than 1. The second is that each node of each tree corresponds to an element of a set and therefore there is no need for special nodes. Once again, the root of each tree contains the name of the corresponding set. A union(A, B) is performed by making the tree root of set B child of the tree root of set A. A find(x) is performed by starting from the node x and by following the pointer to the parent until the tree root is reached. The name of the set stored in the tree root is then returned. As a result, the quick-union algorithm is able to support each union in $O(1)$ time and each find in $O(n)$ time.

This time bound can be improved by using the freedom implicit in each union operation, according to one of the following two union rules. This gives rise to two weighted quick-union algorithms.

> *Union by size:* Make the root of the smaller tree point to the root of the larger, arbitrarily breaking a tie. This requires maintaining the number of descendants for each node, in the following referred to as the *size* of a node, throughout all the sequence of operations.
>
> *Union by rank:* [89] Make the root of the shallower tree point to the root of the other, arbitrarily breaking a tie. This requires maintaining the height of the subtree rooted at each node, in the following referred to as the *rank* of a node, throughout all the sequences of operations.

After a union(A, B), the name of the new tree root is set to A. It can be easily proved (see, e.g., [89]) that the height of the trees achieved with either the "union by size" or the "union by rank" rule is never more than $\log n$. Thus, with either rule each union can be performed in $O(1)$ time and each find in $O(\log n)$ time.

A better amortized bound can be obtained if one of the following *compaction rules* is applied to the path examined during a find operation (see Fig. 5.5).

> *Path compression* [45]: Make every encountered node point to the tree root.
>
> *Path splitting* [93, 94]: Make every encountered node (except the last and the next to last) point to its grandparent.
>
> *Path halving* [93, 94]: Make every other encountered node (except the last and the next to last) point to its grandparent.

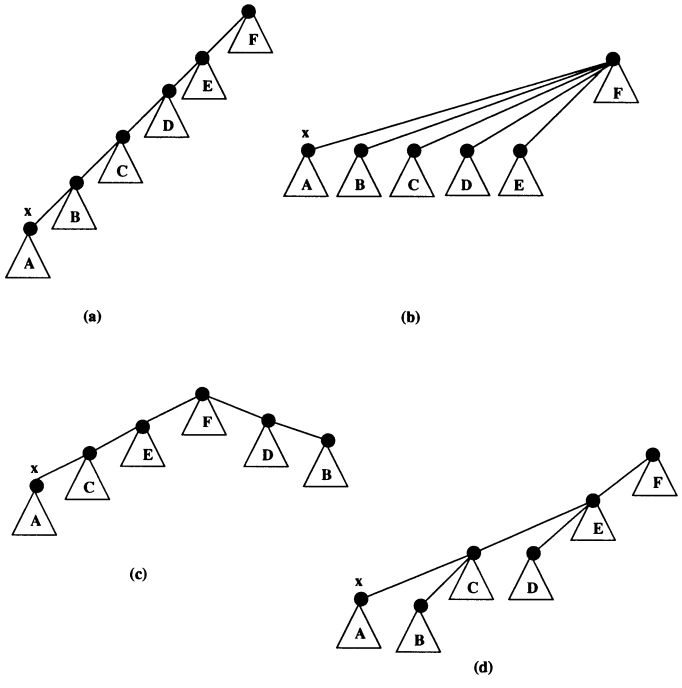


FIGURE 5.5 Illustrating path compaction techniques: (a) the tree before performing a find(x) operation; (b) path compression; (c) path splitting; (d) path halving.

Combining the two choices of a union rule and the three choices of a compaction rule, six possible algorithms are obtained. As shown in [89] they all have an $O(\alpha(m+n,n))$ amortized time complexity, where α is a very slowly growing function, a functional inverse of Ackermann's function [1].

THEOREM 5.1 **[89]** *The algorithms with either linking by size or linking by rank and either compression, splitting or halving run in* $O(n+m\alpha(m+n,n))$ *time on a sequence of at most* $(n-1)$ *unions and m finds.*

No better amortized bound is possible for separable and nonseparable pointer algorithms or in the cell probe model of computation [32, 56, 89].

THEOREM 5.2 **[32, 56, 89]** *Any pointer-based or cell-probe algorithm requires* $\Omega(n+m\alpha(m+n,n))$ *worst-case time for processing a sequence of* $(n-1)$ *unions and m finds.*

Single-Operation Worst-Case Time Complexity

The algorithms that use any union and any compaction rule have still single-operation worst-case time complexity $O(\log n)$ [89], since the trees created by any of the union rules can have height as large as $O(\log n)$. Blum [14] proposed a data structure for the set union problem that supports each union and find in $O(\log n/\log\log n)$ time in the worst case, and showed that this is the actual lower bound for separable pointer-based algorithms.

The data structure used to establish the upper bound is called *k–UF tree*. For any $k \geq 2$, a k–UF tree is a rooted tree such that: (i) the root has at least two children; (ii) each internal node has at least k children; and (iii) all the leaves are at the same level. As a consequence of this definition, the height of a k–UF tree

with n leaves is at most $\lceil \log_k n \rceil$. We refer to the root of a k–UF tree as *fat* if it has more than k children, and as *slim* otherwise. A k–UF tree is said to be *fat* if its root is fat, otherwise it is referred to as *slim*.

Disjoint sets can be represented by k–UF trees as follows. The elements of the set are stored in the leaves and the name of the set is stored in the root. Furthermore, the root also contains the height of the tree and a bit specifying whether it is fat or slim. A find(x) is performed as described in the previous section by starting from the leaf containing x and returning the name stored in the root. This can be accomplished in $O(\log_k n)$ worst-case time. A union(A, B) is performed by first accessing the roots r_A and r_B of the corresponding k–UF trees T_A and T_B. Blum assumed that his algorithm obtained in constant time r_A and r_B before performing a union(A, B). If this is not the case, r_A and r_B can be obtained by means of two finds (i.e., find(A) and find(B)), due to the property that the name of each set corresponds to one of the items contained in the set itself. We now show how to unite the two k–UF *trees* T_A and T_B. Assume without loss of generality that $height(T_B) \leq height(T_A)$. Let v be the node on the path from the leftmost leaf of T_A to r_A with the same height as T_B. Clearly, v can be located by following the leftmost path starting from the root r_A for exactly $height(T_A) - height(T_B)$ steps. When merging T_A and T_B, only three cases are possible, which give rise to three different types of unions.

Type 1: Root r_B is fat (i.e., has more than k children) and v is not the root of T_A. Then r_B is made a sibling of v.

Type 2: Root r_B is fat and v is fat and equal to r_A (the root of T_A). A new (slim) root r is created and both r_A and r_B are made children of r.

Type 3: This deals with the remaining cases, i.e., either root r_B is slim or $v = r_A$ is slim. If root r_B is slim, then all the children of r_B are made the rightmost children of v, and r_B is deleted. Otherwise, all the children of the slim node $v = r_A$ are made the rightmost children of r_B, and r_A is deleted.

THEOREM 5.3 **[14]** *k–UF trees can support each union and find in $O(\log n / \log\log n)$ time in the worst case. Their space complexity is $O(n)$.*

PROOF Each find can be performed in $O(\log_k n)$ time. Each union(A, B) can require at most $O(\log_k n)$ time to locate the nodes r_A, r_B and v as defined above. Both type 1 and type 2 unions can be performed in constant time, while type 3 unions require at most $O(k)$ time, due to the definition of a slim root. Choosing $k = \lceil \log n / \log\log n \rceil$ yields the claimed time bound. The space complexity derives from the fact that a k–UF tree with ℓ leaves has at most $(2\ell - 1)$ nodes. Thus, the forest of k–UF trees requires at most a total of $O(n)$ space to store all the disjoint sets.

Blum showed also that this bound is tight for the class of separable pointer algorithms, while Fredman and Saks [32] showed that the same lower bound holds in the cell probe model of computation.

THEOREM 5.4 **[14, 32]** *Every separable pointer or cell-probe algorithm for the disjoint set union problem has single-operation worst-case time complexity at least $\Omega(\log n / \log\log n)$.*

Special Linear Cases

The six algorithms using either union rule and either compaction rule as described in "Amortized Time Complexity" run in $O(n + m\alpha(m, n))$ time on a sequence of at most $(n - 1)$ union and m find operations. As stated in Theorem 5.2, no better amortized bound is possible for either pointer-based algorithms or in the cell probe model of computation. This does not exclude, however, that a better bound is possible for a special case of set union. Gabow and Tarjan [34] indeed proposed a random-access algorithm that

runs in linear time in the special case where the structure of the union operations is known in advance. Interestingly, Tarjan's lower bound for separable pointer algorithms applies also to this special case, and thus, the power of a random access machine seems necessary to achieve a linear-time algorithm. This result is of theoretical interest as well as being significant in many applications, such as scheduling problems, the off-line minimum problem, finding maximum matching on graphs, VLSI channel routing, finding nearest common ancestors in trees, and flow graph reducibility [34].

The problem can be formalized as follows. We are given a tree T containing n nodes which correspond to the initial n singleton sets. Denoting by $p(v)$ the parent of the node v in T, we have to perform a sequence of union and find operations such that each union can be only of the form union$(p(v), v)$. For such a reason, T is called the *static union tree* and the problem will be referred to as the *static tree set union*. Also the case in which the union tree can dynamically grow by means of new node insertions (referred to as *incremental tree set union*) can be solved in linear time.

THEOREM 5.5 **[34]** *If the knowledge about the union tree is available in advance, each union and find operation can be supported in $O(1)$ amortized time. The total space required is $O(n)$.*

The same algorithm given for the static tree set union can be extended to the incremental tree set union problem. For this problem, the union tree is not known in advance but is allowed to grow only one node at the time during the sequence of union and find operations. This has application in several algorithms for finding maximum matching in general graphs.

THEOREM 5.6 **[34]** *The algorithm for incremental tree set union runs in a total of $O(m + n)$ time and requires $O(n)$ preprocessing time and space.*

Loebl and Nešetřil [58] presented a linear-time algorithm for another special case of the set union problem. They considered sequences of unions and finds with a constraint on the subsequence of finds. Namely, the finds are listed in a *postorder* fashion, where a postorder is a linear ordering of the leaves induced by a drawing of the tree in the plane. In this framework, they proved that such sequences of union and find operations can be performed in linear time, thus, getting $O(1)$ amortized time per operation. A preliminary version of these results was reported in [58].

5.3 The Set Union Problem on Intervals

In this section, we describe efficient solutions to the *set union problem on intervals*, which can be defined as follows. Informally, we would like to maintain a partition of a list $\{1, 2, \ldots, n\}$ in adjacent intervals. A union operation joins two adjacent intervals, a find returns the name of the interval containing x and a split divides the interval containing x (at x itself). More formally, at any time we maintain a collection of disjoint sets A_i with the following properties. The A_i's, $1 \leq i \leq k$, are disjoint sets whose members are ordered by the relation $\leq$, and such that $\cup_{i=1}^{k} A_i = \{1, 2, \ldots, n\}$. Furthermore, every item in A_i is less than or equal to all the items in A_{i+1}, for $i = 1, 2, \ldots, n-1$. In other words, the intervals A_i partition the interval $[1, n]$. Set A_i is said to be adjacent to sets A_{i-1} and A_{i+1}. The set union problem on intervals consists of performing a sequence of the following three operations:

union(S_1, S_2, S)*:* Given the adjacent sets S_1 and S_2, combine them into a new set $S = S_1 \cup S_2$;
find(x)*:* Given the item x, return the name of the set containing x;
split(S, S_1, S_2, x)*:* Partition S into two sets $S_1 = \{a \in S | a < x\}$ and $S_2 = \{a \in S | a \geq x\}$.

Adopting the same terminology used in [69], we will refer to the set union problem on intervals as the *interval union-split-find problem*. After discussing this problem, we consider two special cases:

the *interval union-find problem* and the *interval split-find problem*, where only union-find and split-find operations are allowed, respectively. The interval union-split-find problem and its subproblems have applications in a wide range of areas, including problems in computational geometry such as dynamic segment intersection [49, 67, 68], shortest paths problems [6, 66], and the longest common subsequence problem [5, 48].

Interval Union-Split-Find

In this section we will describe optimal separable and nonseparable pointer algorithms for the interval union-split-find problem. The best separable algorithm for this problem runs in $O(\log n)$ worst-case time for each operation, while nonseparable pointer algorithms require only $O(\log \log n)$ worst-case time for each operation. In both cases, no better bound is possible.

The upper bound for separable pointer algorithms can be easily obtained by means of balanced trees [4, 21], while the lower bound was proved by Mehlhorn et al. [69].

THEOREM 5.7 **[69]** *For any separable pointer algorithm, both the worst-case per operation time complexity of the interval split-find problem and the amortized time complexity of the interval union-split-find problem are* $\Omega(\log n)$.

Turning to nonseparable pointer algorithms, the upper bound can be found in [52, 68, 91, 92]. In particular, van Emde Boas et al. [92] introduced a priority queue which supports among other operations *insert*, *delete*, and *successor* on a set with elements belonging to a fixed universe $S = \{1, 2, \ldots, n\}$. The time required by each of those operation is $O(\log \log n)$. Originally, the space was $O(n \log \log n)$ but later it was improved to $O(n)$. It is easy to show (see also [69]) that the above operations correspond respectively to union, split, and find, and therefore the following theorem holds.

THEOREM 5.8 **[91]** *Each union, find and split can be supported in* $O(\log \log n)$ *worst-case time. The space required is* $O(n)$.

We observe that the algorithm based on van Emde Boas' priority queue is inherently nonseparable. Mehlhorn et al. [69] proved that this is indeed the best possible bound that can be achieved by a nonseparable pointer algorithm:

THEOREM 5.9 **[69]** *For any nonseparable pointer algorithm, both the worst-case per operation time complexity of the interval split-find problem and the amortized time complexity of the interval union-split-find problem are* $\Omega(\log \log n)$.

Notice that Theorems 5.7 and 5.8 imply that for the interval union-split-find problem the separability assumption causes an exponential loss of efficiency.

Interval Union-Find

The interval union-find problem can be seen from two different perspectives: indeed it is a special case of the union-split-find problem, when no split operations are performed, and it is a restriction of the set union problem described in Section 5.2, where only adjacent intervals are allowed to be joined. Consequently, the $O(\alpha(m+n, n))$ amortized bound given in Theorem 5.1 and the $O(\log n / \log \log n)$ single-operation worst-case bound given in Theorem 5.3 trivially extend to interval union–find. Tarjan's proof of the $\Omega(\alpha(m+n, n))$ amortized lower bound for separable pointer algorithms also holds for the interval union–

find problem, while Blum and Rochow [15] have adapted Blum's original lower bound proof for separable pointer algorithms to interval union–find. Thus, the best bounds for separable pointer algorithms are achieved by employing the more general set union algorithms. On the other side, the interval union–find problem can be solved in $O(\log \log n)$ time per operation with the nonseparable algorithm of van Emde Boas [91], while Gabow and Tarjan used the data structure described in "Special Linear Cases" to obtain an $O(1)$ amortized time for interval union–find on a random access machine.

Interval Split–Find

According to Theorems 5.7, 5.8, and 5.9, the two algorithms given for the more general interval union-split-find problem, are still optimal for the single-operation worst-case time complexity of the interval split–find problem. As a result, each split and find operation can be supported in $\Theta(\log n)$ and in $\Theta(\log \log n)$ time, respectively, in the separable and nonseparable pointer machine model.

As shown by Hopcroft and Ullman [45], the amortized complexity of this problem can be reduced to $O(\log^* n)$, where $\log^* n$ is the iterated logarithm function.[2] Their algorithm works as follows. The basic data structure is a tree, for which each node at level i, $i \geq 1$, has at most $2^{f(i-1)}$ children, where $f(i) = f(i-1)2^{f(i-1)}$, for $i \geq 1$, and $f(0) = 1$. A node is said to be complete either if it is at level 0 or if it is at level $i \geq 1$ and has $2^{f(i-1)}$ children, all of which are complete. A node that is not complete is called incomplete. The invariant maintained for the data structure is that no node has more than two incomplete children. Moreover, the incomplete children (if any) will be leftmost and rightmost. As in the usual tree data structures for set union, the name of a set is stored in the tree root.

Initially, such a tree with n leaves is created. Its height is $O(\log^* n)$ and therefore a find(x) will require $O(\log^* n)$ time to return the name of the set. To perform a split(x), we start at the leaf corresponding to x and traverse the path to the root to partition the tree into two trees. It is possible to show that using this data structure, the amortized cost of a split is $O(\log^* n)$ [45]. This bound can be further improved to $O(\alpha(m, n))$ as shown by Gabow [33]. The algorithm used to establish this upper bound relies on a sophisticated partition of the items contained in each set.

THEOREM 5.10 **[33]** *There exists a data structure supporting a sequence of m find and split operations in $O(m\alpha(m, n))$ worst-case time. The space required is $O(n)$.*

La Poutré [56] proved that this bound is tight for (both separable and nonseparable) pointer-based algorithms.

THEOREM 5.11 **[56]** *Any pointer-based algorithm requires $\Omega(n + m\alpha(m, n))$ time to perform $(n - 1)$ split and m find operations.*

Using the power of a random access machine, Gabow and Tarjan were able to achieve $\Theta(1)$ amortized time for the interval split–find problem [34]. This bound is obtained by employing a slight variant of the data structure sketched in "Special Linear Cases."

---

[2] $\log^* n = \min\{i \mid \log^{[i]} n \leq 1\}$, where $\log^{[i]} n = \log \log^{[i-1]} n$ for $i > 0$ and $\log^{[0]} n = n$.

5.4 The Set Union Problem with Deunions

Mannila and Ukkonen [59] defined a generalization of the set union problem, which they called *set union with deunions*. In addition to union and find, the following operation is allowed.

deunion: Undo the most recently performed union operation not yet undone.

Motivations for studying this problem arise in logic programming, and more precisely in memory management of interpreters without function symbols [40, 60, 61, 96]. In Prolog, for example, variables of clauses correspond to the elements of the sets, unifications correspond to unions and backtracking corresponds to deunions [60].

Algorithms for Set Union with Deunions

The set union problem with deunions can be solved by a modification of Blum's data structure described in "Single-Operation Worst-Case Time Complexity." To facilitate deunions, we maintain a *union stack* that stores some bookkeeping information related to unions. Finds are performed as in "Single-Operation Worst-Case Time Complexity." Unions require some additional work to maintain the union stack. We now sketch which information is stored in the union stack. For sake of simplicity we do not take into account names of the sets (namely, we show how to handle unite rather than union operations): names can be easily maintained in some extra information stored in the union stack. Initially, the union stack is empty. When a type 1 union is performed, we proceed as in "Single-Operation Worst-Case Time Complexity" and then push onto the union stack a record containing a pointer to the old root r_B. Similarly, when a type 2 union is performed, we push onto the union stack a record containing a pointer to r_A and a pointer to r_B. Finally, when a type 3 union is performed, we push onto the union stack a pointer to the leftmost child of either r_B or r_A, depending on the two cases.

Deunions basically use the top stack record to invalidate the last union performed. Indeed, we pop the top record from the union stack, and check whether the union to be undone is of type 1, 2, or 3. For type 1 unions, we follow the pointer to r_B and delete the edge leaving this node, thus, restoring it as a root. For type 2 unions, we follow the pointers to r_A and r_B and delete the edges leaving these nodes and their parent. For type 3 unions, we follow the pointer to the node, and move it together with all its right sibling as a child of a new root.

It can be easily showed that this augmented version of Blum's data structure supports each union, find, and deunion in $O(\log n / \log\log n)$ time in the worst case, with an $O(n)$ space usage. This was proved to be a lower bound for separable pointer algorithms by Westbrook and Tarjan [97]:

THEOREM 5.12 **[97]** *Every separable pointer algorithm for the set union problem with deunions requires at least* $\Omega(\log n / \log\log n)$ *amortized time per operation.*

All of the union rules and path compaction techniques described in "Amortized Time Complexity" can be extended in order to deal with deunions using the same bookkeeping method (i.e., the union stack) described above. However, path compression with any one of the union rules leads to an $O(\log n)$ amortized algorithm, as it can be seen by first performing $(n-1)$ unions which build a binomial tree (as defined, for instance, in [89]) of depth $O(\log n)$ and then by repeatedly carrying out a find on the deepest leaf, a deunion, and a redo of that union. Westbrook and Tarjan [97] showed that using either one of the union rules combined with path splitting or path halving yield $O(\log n / \log\log n)$ amortized algorithms for the set union problem with deunions. We now describe their algorithms.

In the following, a union operation not yet undone will be referred to as *live*, and as *dead* otherwise. To handle deunions, again a *union stack* is maintained, which contains the roots made nonroots by live unions. Additionally, we maintain for each node x a *node stack* $P(x)$, which contains the pointers leaving

x created either by unions or by finds. During a path compaction caused by a find, the old pointer leaving x is left in $P(x)$ and each newly created pointer (x, y) is pushed onto $P(x)$. The bottommost pointer on these stacks is created by a union and will be referred to as a *union pointer*. The other pointers are created by the path compaction performed during the find operations and are called *find pointers*. Each of these pointers is associated with a unique union operation, the one whose undoing would invalidate the pointer. The pointer is said to be *live* if the associated union operation is live, and it is said to be *dead* otherwise.

Unions are performed as in the set union problem, except that for each union a new item is pushed onto the union stack, containing the tree root made nonroot and some bookkeeping information about the set name and either size or rank. To perform a deunion, the top element is popped from the union stack and the pointer leaving that node is deleted. The extra information stored in the union stack is used to maintain set names and either sizes or ranks.

There are actually two versions of these algorithms, depending on when dead pointers are removed from the data structure. *Eager algorithms* pop pointers from the node stacks as soon as they become dead (i.e., after a deunion operation). On the other hand, *lazy algorithms* remove dead pointers in a lazy fashion while performing subsequent union and find operations. Combined with the allowed union and compaction rules, this gives a total of eight algorithms. They all have the same time and space complexity, as the following theorem shows.

THEOREM 5.13 **[97]** *Either union by size or union by rank in combination with either path splitting or path halving gives both eager and lazy algorithms which run in $O(\log n / \log\log n)$ amortized time for operation. The space required by all these algorithms is $O(n)$.*

The Set Union Problem with Unlimited Backtracking

Other variants of the set union problem with deunions have been considered such as set union with arbitrary deunions [36, 63], set union with dynamic weighted backtracking [39], and set union with unlimited backtracking [9]. In this chapter, we will discuss only set union with unlimited backtracking and refer the interested readers to the references for the other problems.

As before, we denote a union not yet undone by live, and by dead otherwise. In the set union problem with unlimited backtracking, deunions are replaced by the following more general operation:

backtrack(i): Undo the last i live unions performed. i is assumed to be an integer, $i \geq 0$.

The name of this problem derives from the fact that the limitation that at most one union could be undone per operation is removed.

Note that this problem is more general than the set union problem with deunions, since a deunion can be simply implemented as backtrack(1). Furthermore, a backtrack(i) can be implemented by performing exactly i deunions. Hence, a sequence of m_1 unions, m_2 finds, and m_3 backtracks can be carried out by simply performing at most m_1 deunions instead of the backtracks. Applying either Westbrook and Tarjan's algorithms or Blum's modified algorithm to the sequence of union, find, and deunion operations, a total of $O((m_1 + m_2)\log n / \log\log n)$ worst-case running time will result. As a consequence, the set union problem with unlimited backtracking can be solved in $O(\log n / \log\log n)$ amortized time per operation. Since deunions are a special case of backtracks, this bound is tight for the class of separable pointer algorithms because of Theorem 5.12.

However, using either Westbrook and Tarjan's algorithms or Blum's augmented data structure, each backtrack(i) can require $\Omega(i \log n / \log\log n)$ in the worst case. Indeed, the worst-case time complexity of backtrack(i) is at least $\Omega(i)$ as long as one insists on deleting pointers as soon as they are invalidated by backtracking (as in the eager methods described in "Algorithms for Set Union with Deunions," since in this case at least one pointer must be removed for each erased union. This is clearly undesirable, since i can be as large as $(n - 1)$.

The following theorem holds for the set union with unlimited backtracking, when union operations are taken into account.

THEOREM 5.14 **[37]** *It is possible to perform each union, find and backtrack(i) in $O(\log n)$ time in the worst case. This bound is tight for nonseparable pointer algorithms.*

Apostolico et al. [9] showed that, when unites instead of unions are performed (i.e., when the name of the new set can be arbitrarily chosen by the algorithm), a better bound for separable pointer algorithms can be achieved:

THEOREM 5.15 **[9]** *There exists a data structure which supports each unite and find operation in $O(\log n / \log\log n)$ time, each backtrack in $O(1)$ time, and requires $O(n)$ space.*

No better bound is possible for any separable pointer algorithm or in the cell probe model of computation, as it can be shown by a trivial extension of Theorem 5.4.

5.5 Partial and Full Persistence

In this section we cover general techniques for partial and full persistence. The time complexities of these techniques will generally be expressed in terms of *slowdowns* with respect to the ephemeral query and update operations. The slowdowns will usually be functions of m, the number of versions. A slowdown of $T_q(m)$ for queries means, for example, that a persistent query to a version which is a data structure of size n is accomplished in time $O(T_q(m) \cdot Q(n))$ time, where $Q(n)$ is the running time of an ephemeral query operation on a data structure of size n.

Methods for Arbitrary Data Structures

The Fat Node Method

A very simple idea for making any data structure partially persistent is the *fat node* method, which works as follows. The m versions are numbered by integers from 1 (the first) to m (the last). We will take the convention that if a persistent query specifies version t, for some $1 \leq t \leq m$, then the query is answered according to the state of the data structure as it was after version t was created but before (if ever) version $t + 1$ was begun.

Each memory location μ in the ephemeral data structure can be associated with a set $C(\mu)$ containing pairs of the form $\langle t, v\rangle$, where v is a value and t is a version number, sometimes referred to as the *time stamp* of v. A pair $\langle t, v\rangle$ is present in $C(\mu)$ if and only if (a) memory location μ was modified while creating version t and (b) at the completion of version t, the location μ contained the value v. For every memory location μ in the ephemeral data structure, we associate an auxiliary data structure $A(\mu)$, which stores $C(\mu)$ ordered by time stamp.

In order to perform a persistent query in version t we simulate the operation of the ephemeral query algorithm. Whenever the ephemeral query algorithm attempts to read a memory location μ, we query $A(\mu)$ to determine the value of μ in version t. Let t^* be the largest time stamp in $C(\mu)$ which is less than or equal to t. Clearly, the required value is v^* where $\langle t^*, v^*\rangle \in C(\mu)$. Creating version $m + 1$ by modifying version m is also easy: if memory locations $\mu_1, \mu_2, \ldots$ were modified while creating version $m + 1$, and the values of these locations in version $m + 1$ were $v_1, v_2, \ldots$, we simply insert the pair $\langle m + 1, v_i\rangle$ to $A(\mu_i)$ for $i = 1, 2, \ldots$.

If we implement the auxiliary data structures as red-black trees [21] then it is possible to query $A(\mu)$ in $O(\log |C(\mu)|) = O(\log m)$ time and also to add a new pair to $A(\mu)$ in $O(1)$ amortized time (this is

possible because the new pair will always have a time stamp greater than or equal to any time stamp in $C(\mu)$). In fact, we can even obtain $O(1)$ worst-case slowdown for updates by using a data structure given in [57]. Note that each ephemeral memory modification performed during a persistent update also incurs a space cost of $O(1)$ (in general this is unavoidable). We thus obtain the following theorem.

THEOREM 5.16 **[28]** *Any data structure can be made partially persistent with slowdown $O(\log m)$ for queries and $O(1)$ for updates. The space cost is $O(1)$ for each ephemeral memory modification.*

The fat node method can be extended to full persistence with a little work. Again, we will take the convention that a persistent query on version t is answered according to the state of the data structure as it was after version t was created but before (if ever) it was modified to create any descendant version. Again, each memory location μ in the ephemeral data structure will be associated with a set $C(\mu)$ containing pairs of the form $\langle t, v\rangle$, where v is a value and t is a version (the timestamp). The rules specifying what pairs are stored in $C(\mu)$ are somewhat more complicated. The main difficulty is that the versions in full persistence are only partially ordered. In order to find out the value of a memory location μ in version t, we need to find the deepest ancestor of t in the version tree where μ was modified (this problem is similar to the inheritance problem for object-oriented languages).

One solution is to impose a total order on the versions by converting the version tree into a *version list*, which is simply a pre-order listing of the version tree. Whenever a new version is created, it is added to the version list immediately after its parent, thus inductively maintaining the pre-ordering of the list. We now compare any two versions as follows: the one which is further to the left in the version list is considered smaller.

For example, a version list corresponding to the tree in Fig. 5.6 is $[a, b, c, f, g, h, i, j, l, m, n, o, k, d, e]$, and by the linearization, version f is considered to be less than version m, and version j is considered to be less than version l.

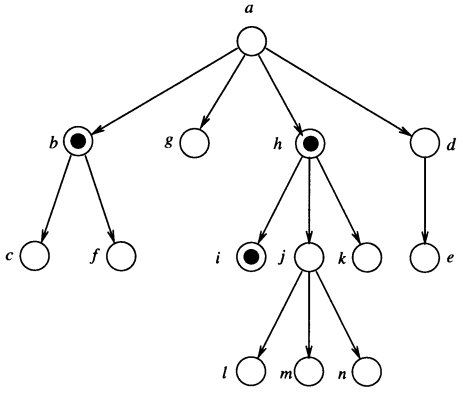


FIGURE 5.6 Navigating in full persistence: an example version tree.

Now consider a particular memory location π which was modified in versions b, h, and i of the data structure, with values B, H, and I being written to it in these versions. The following table shows the value of π in each version in the list (a $\perp$ means that no value has yet been written to π and hence its value may be undefined):

Version	a	b	c	f	g	h	i	j	l	m	n	o	k	d	e
Value	$\perp$	B	B	B	$\perp$	H	I	H	H	H	H	H	H	$\perp$	$\perp$

As can be seen in the above example, if π is modified in versions b, h and i, the version list is divided into

intervals containing respectively the sets $\{a\}, \{b, c, f\}, \{g\}, \{h\}, \{i\}, \{j, l, m, n, o, k\}, \{d, e\}$, such that for all versions in that interval, the value of π is the same. In general, the intervals of the version list for which the answer is the same will be different for different memory locations.

Hence, for each memory location μ, we define $C(\mu)$ to contains pairs of the form $\langle t, v\rangle$, where t is the leftmost version in its interval, and v is the value of μ in version t. Again, $C(\mu)$ is stored in an auxiliary data structure $A(\mu)$ ordered by time-stamp (the ordering among versions is as specified by the version list). In the example above, $C(\pi)$ would contain the following pairs:

$$\langle a, \perp\rangle, \langle b, B\rangle, \langle g, \perp\rangle, \langle h, H\rangle, \langle i, I\rangle, \langle j, H\rangle, \langle d, \perp\rangle .$$

In order to determine the value of some memory location μ in version t, we simply search among the pairs stored in $A(\mu)$, comparing versions, until we find the left endpoint of the interval to which t belongs; the associated value is the required answer.

How about updates? Let μ be any memory location, and firstly notice that if a new version is created in which μ is not modified, the value of μ in this new version will be the same as the value of μ in its parent, and the new version will be added to the version list right after its parent. This will simply enlarge the interval to which its parent belongs, and will also not change the left endpoint of the interval. Hence, if μ is not modified in some version, no change need be made to $A(\mu)$. On the other hand, adding a version where μ is modified creates a new interval containing only the new version, and in addition may split an existing interval into two. In general, if μ is modified in k different versions, $C(\mu)$ may contain up to $2k + 1$ pairs, and in each update, up to two new pairs may need to be inserted into $A(\mu)$. In the above example, if we create a new version p as a child of m and modify π to contain P in this version, then the interval $\{j, l, m, n, o, k\}$ splits into two intervals $\{j, l, m\}$ and $\{n, o, k\}$, and the new interval consisting only of $\{p\}$ is created. Hence, we would have to add the pairs $\langle n, H\rangle$ and $\langle p, P\rangle$ to $C(\pi)$.

Provided we can perform the comparison of two versions in constant time, and we store the pairs in say a red-black tree, we can perform a persistent query by simulating the ephemeral query algorithm, with a slowdown of $O(\log |C(\mu)|) = O(\log m)$, where m is the total number of versions. In the case of full persistence, updates also incur a slowdown of $O(\log m)$, and incur a $O(1)$ space cost per memory modification. Maintaining the version list so that two versions can be compared in constant time to determine which of the two is leftward is known as the *list order* problem, and has been studied in a series of papers [22, 90], culminating in an optimal data structure by Dietz and Sleator [24] which allows insertions and comparisons each in $O(1)$ worst-case time. We conclude:

THEOREM 5.17 **[28]** *Any data structure can be made fully persistent with slowdown $O(\log m)$ for both queries and updates. The space cost is $O(1)$ for each ephemeral memory modification.*

Faster Implementations of the Fat Node Method

For arbitrary data structures, the slowdown produced by the fat node method can be reduced by making use of the power of the RAM model. In the case of partial persistence, the versions are numbered with integers from 1 to m, where m is the number of versions, and special data structures for predecessor queries on integer sets may be used. For instance, the van Emde Boas data structure [91, 92] processes insertions, deletions, and predecessor queries on a set $S \subseteq \{1, \ldots, m\}$ in $O(\log \log m)$ time each. By using dynamic perfect hashing [27] to minimize space usage, the space required by this data structure can be reduced to linear in the size of the data structure, at the cost of making the updates run in $O(\log \log m)$ expected time. We thus obtain:

THEOREM 5.18 **[28, 27]** *Any data structure can be made partially persistent on a RAM with slowdown $O(\log \log m)$ for queries and expected slowdown $O(\log \log m)$ for updates. The space cost is $O(1)$ per ephemeral memory modification.*

At first sight it does not appear possible to use the same approach for full persistence because the versions are not integers. However, it turns out that algorithms for the list order problem work by assigning integer labels to the elements of the version list such that the labels increase monotonically from the beginning to the end of the list. Furthermore, these labels are guaranteed to be in the range $1..m^c$ where m is the number of versions and $c > 1$ is some constant. This means we can once again use the van Emde Boas data structure to search amongst the versions in $O(\log \log m)$ time. Unfortunately, each insertion into the version list may cause many of the integers to be relabeled, and making the changes to the appropriate auxiliary structures may prove expensive. Dietz [23] shows how to combine modifications to the list order algorithms together with standard bucketing techniques to obtain:

THEOREM 5.19 **[23]** *Any data structure can be made fully persistent on a RAM with slowdown $O(\log \log m)$ for queries and expected slowdown $O(\log \log m)$ for updates. The space cost is $O(1)$ per ephemeral memory modification.*

Methods for Linked Data Structures

The methods discussed above, while efficient, are not optimal and some of them are not simple to code. By placing some restrictions on the class of data structures which we want to make persistent, we can obtain some very simple and efficient algorithms for persistence. One such subclass of data structures is that of *linked* data structure.

A linked data structure is an abstraction of pointer-based data structures such as linked lists, search trees, etc. Informally, a linked data structure is composed of a collection of *nodes*, each with a finite number of named fields. Some of these fields are capable of holding an atomic piece of information, while others can hold a pointer to some node (or the value nil). For simplicity we assume the nodes are homogenous (i.e., of the same type) and that all access to the data structure is through a single designated *root* node. Any version of a linked data structure can be viewed as a directed graph, with vertices corresponding to nodes and edges corresponding to pointers.

Queries are abstracted away as *access operations* which consist of a series of *access steps*. The access algorithm has a collection of *accessed* nodes, which initially contains only the root. At each step, the algorithm either reads information from one of the accessed nodes or follows a non-nil pointer from one of the accessed nodes; the node so reached is then added to the set of accessed nodes. In actual data structures, of course, the information read by the query algorithm would be used to determine the pointers to follow as well as to compute an answer to return. Update operations are assumed to consist of an intermixed sequence of access steps as before and *update steps*. An update step either creates an explicitly initialized new node or writes a value to a field of some previously accessed node. We now discuss how one might implement persistent access and update operations.

Path Copying

A very simple but wasteful method for persistence is to copy the entire data structure after every update. *Path copying* is an optimization of this for linked data structures, which copies only "essential" nodes. Specifically, if an update modifies a version v by changing values in a set S of nodes, then it suffices to make copies of the nodes in S, together with all nodes that lie on a path from the root of version v to any node in S. The handle to the new version is simply a pointer to the new root. One advantage of this method is that traversing it is trivial: given a pointer to the root in some version, traversing it is done exactly as in the ephemeral case.

This method performs reasonably efficiently in the case of balanced search trees. Assuming that each node in the balanced search tree contains pointers only to its children, updates in balanced search trees such as AVL trees [2] and red-black trees [21] would cause only $O(\log n)$ nodes to be copied (these would be nodes either on the path from the root to the inserted or deleted item, or nodes adjacent to this path).

Note that this method does not work as well if the search tree only has an amortized $O(\log n)$ update cost, e.g., in the case of splay trees [87, p. 53 ff]. We therefore get the following theorem, which was independently noted by [74, 81].

THEOREM 5.20 *There is a fully persistent balanced search tree with persistent update and query times $O(\log n)$ and with space cost $O(\log n)$ per update, where n is the number of keys in the version of the data structure which is being updated or queried.*

Of course, for many other data structures, path copying may prove prohibitively expensive, and even in the case of balanced search trees, the space complexity is non-optimal, as red-black trees with lazy recoloring only modify $O(1)$ locations per update.

The Node Copying and Split Node Data Structures

An (ephemeral) *bounded-degree* linked data structure is one where the maximum *in-degree*, i.e., the maximum number of nodes that are pointing to any node, is bounded by a constant. Many, if not most, pointer-based data structures have this property, such as linked lists, search trees and so on (some of the data structures covered earlier in this chapter do not have this property). Driscoll et al. [28] showed that bounded-degree linked data structures could be made partially or fully persistent very efficiently, by means of the *node copying* and *split node* data structures respectively.

The source of inefficiency in the fat node data structure is searching among all the versions in the auxiliary data structure associated with an ephemeral node, as there is no bound on the number of such versions. The *node copying* data structure attempts to remedy this by replacing each fat node by a *collection* of "plump" nodes, each of which is capable of recording a bounded number of changes to an ephemeral node. Again, we assume that the versions are numbered with consecutive integers, starting from 1 (the first) to m (the last). Analogously to the fat node data structure, each ephemeral node x is associated with a set $C(x)$ of pairs $\langle t, r\rangle$, where t is a version number, and r is a record containing values for each of the fields of x. The set $C(x)$ is stored in a collection of plump nodes, each of which is capable of storing $2d+1$ pairs, where d is the bound on the in-degree of the ephemeral data structure.

The collection of plump nodes storing $C(x)$ is kept in a linked list $L(x)$. Let X be any plump node in $L(x)$ and let X' the next plump node in the list, if any. Let τ denote the smallest time stamp in X' if X' exists, and let $\tau = \infty$ otherwise. The list $L(x)$ is sorted by time stamp in the sense that all pairs in X are sorted by time stamp and all time stamps in X are smaller than τ. Each pair $\langle t, r\rangle$ in X is naturally associated with a *valid interval*, which is the half-open interval of versions beginning at t, up to, but not including the time stamp of the next pair in X, or τ if no such pair exists. The valid interval of X is simply the union of the valid intervals of the pairs stored in X. The following invariants always hold:

(i) For any pair $p = \langle t, r\rangle$ in $C(x)$, if a data field in r contains some value v then the value of the corresponding data field of ephemeral node x during the entire valid interval of p was also v. Furthermore, if a pointer field in r contains a pointer to a plump node in $L(y)$ or nil then the corresponding field in ephemeral node x pointed to ephemeral node y or contained nil, respectively, during the entire valid interval of p.

(ii) For any pair $p = \langle t, r\rangle$ in $C(x)$, if a pointer field in r points to a plump node Y, then the valid interval of p is contained in the valid interval of Y.

(iii) The handle of version t is a pointer to the (unique) plump node in $L(root)$ whose valid interval contains t.

A persistent access operation on version t is performed by a step-by-step simulation of the ephemeral access algorithm. For any ephemeral node x and version t, let $P(x, t)$ denote the plump node in $L(x)$ whose valid interval contains t. Since the valid intervals of the pairs in $C(x)$ are disjoint and partition

the interval $[1, \infty)$, this is well-defined. We ensure that if after some step, the ephemeral access algorithm would have accessed a set S of nodes, then the persistent access algorithm would have accessed the set of plump nodes $\{P(y, t) | y \in S\}$. This invariant holds initially, as the ephemeral algorithm would have accessed only $root$, and by (iv), the handle of version t points to $P(root, t)$.

If the ephemeral algorithm attempts to read a data field of an accessed node x then the persistent algorithm searches among the $O(1)$ pairs in $P(x, t)$ to find the pair whose valid interval contains t, and reads the value of the field from that pair. By (ii), this gives the correct value of the field. If the ephemeral algorithm follows a pointer from an accessed node x and reaches a node y, then the persistent algorithm searches among the $O(1)$ pairs in $P(x, t)$ to find the pair whose valid interval contains t, and follows the pointer specified in that pair. By invariants (i) and (ii) this pointer must point to $P(y, t)$. This proves the correctness of the simulation of the access operation.

Suppose during an ephemeral update operation on version m of the data structure, the ephemeral update operation writes some values into the fields of an ephemeral node x. Then the pair $\langle m + 1, r \rangle$ is added to $C(x)$, where r contains the field values of x at the end of the update operation. If the plump node $P(x, m)$ is not full then this pair is simply added to $P(x, m)$. Otherwise, a new plump node that contains only this pair is created and added to the end of $L(x)$. For all nodes y that pointed to x in version m, this could cause a violation of (ii). Hence, for all such y, we add a new pair $\langle m + 1, r' \rangle$ to $C(y)$, where r' is identical to the last record in $C(y)$ except that pointers to $P(x, m)$ are replaced by pointers to the new plump node. If this addition necessitates the creation of a new plump node in $L(y)$ then pointers to $P(m, y)$ are updated as above. A simple potential argument in [28] shows that not only does this process terminate, but the amortized space cost for each memory modification is $O(1)$. At the end of the process, a pointer to the last node in $L(root)$ is returned as the handle to version $m + 1$. Hence, we have that:

THEOREM 5.21 **[28]** *Any bounded-degree linked data structure can be made partially persistent with worst-case slowdown $O(1)$ for queries, amortized slowdown $O(1)$ for updates, and amortized space cost $O(1)$ per memory modification.*

Although we will not describe them in detail here, similar ideas were applied by Driscoll et al. in the *split node* data structure which can be used to make bounded-degree linked data structures fully persistent in the following time bounds:

THEOREM 5.22 **[28]** *Any bounded-degree linked data structure can be made fully persistent with worst-case slowdown $O(1)$ for queries, amortized slowdown $O(1)$ for updates, and amortized space cost $O(1)$ per memory modification.*

Driscoll et al. left open the issue of whether the time and space bounds for Theorems 5.21 and 5.22 could be made worst-case rather than amortized. Toward this end, they used a method called *displaced storage of changes* to give a fully persistent search tree with $O(\log n)$ worst-case query and update times and $O(1)$ amortized space per update, improving upon the time bounds of Theorem 5.20. This method relies heavily on the property of balanced search trees that there is a unique path from the root to any internal node, and it is not clear how to extract a general method for full persistence from it. A more direct assault on their open problem was made by [25], which showed that all bounds in Theorem 5.21 could be made worst-case on the RAM model. In the same paper it was also shown that the space cost could be made $O(1)$ worst-case on the pointer machine model, but the slowdown for updates remained $O(1)$ amortized. Subsequently, Brodal [11] fully resolved the open problem of Driscoll et al. for partial persistence by showing that all bounds in Theorem 5.21 could be made worst-case on the pointer machine model. For the case of full persistence it was shown in [26] how to achieve $O(\log \log m)$ worst-case slowdown for updates and queries and a worst-case space cost of $O(1)$ per memory modification, but the open problem

of Driscoll et al. remains only partially resolved in this case. It should be noted that the data structures of [11, 26] are not much more complicated than the original data structures of Driscoll et al.

5.6 Functional Data Structures

In this section we will consider the implementation of data structures in functional languages. Although implementation in a functional language automatically guarantees persistence, the central issue is maintaining the same level of efficiency as in the imperative setting.

The state-of-the-art regarding general methods is quickly summarized. The path-copying method described at the beginning of the previous section can easily be implemented in a functional setting. This means that balanced binary trees (without parent pointers) can be implemented in a functional language, with queries and updates taking $O(\log n)$ worst-case time, and with a suboptimal worst-case space bound of $\Theta(\log n)$. Using the functional implementation of search trees to implement a dictionary which will simulate the memory of any imperative program, it is possible to implement any data structure which uses a maximum of M memory locations in a functional language with a slowdown of $O(\log M)$ in the query and update times, and a space cost of $O(\log M)$ per memory modification.

Naturally, better bounds are obtained by considering specific data structuring problems, and we summarize the known results at the end of this section. First, though, we will focus on perhaps the most fundamental data structuring problem in this context, that of implementing *catenable lists*. A catenable list supports the following set of operations:

makelist(a): Creates a new list containing only the element a.

head(X): Returns the first element of list X. Gives an error if X is empty.

tail(X): Returns the list obtained by deleting the first element of list X without modifying X. Gives an error if X is empty.

catenate(X, Y): Returns the list obtained by appending list Y to list X, without modifying X or Y.

Driscoll et al. [29] were the first to study this problem, and efficient but nonoptimal solutions were proposed in [16, 29]. We will sketch two proofs of the following theorem, due to Kaplan and Tarjan [50] and Okasaki [71]:

THEOREM 5.23 *The above set of operations can be implemented in $O(1)$ time each.*

The result due to Kaplan and Tarjan is stronger in two respects. Firstly, the solution of [50] gives $O(1)$ worst-case time bounds for all operations, while Okasaki's only gives amortized time bounds. Also, Okasaki's result uses "memoization" which, technically speaking, is a side-effect, and hence, his solution is not purely functional. On the other hand, Okasaki's solution is extremely simple to code in most functional programming languages, and offers insight into how to make amortized data structures fully persistent efficiently. In general, this is difficult because in an amortized data structure, some operations in a sequence of operations may be expensive, even though the average cost is low. In the fully persistent setting, an adversary can repeatedly perform an expensive operation as often as desired, pushing the average cost of an operation close to the maximum cost of any operation.

We will briefly cover both these solutions, beginning with Okasaki's. In each case we will first consider a variant of the problem where the *catenate* operation is replaced by the operation *inject*(a, X) which adds a to the end of list X. Note that *inject*(a, X) is equivalent to *catenate*(*makelist*(a), X). Although this change simplifies the problem substantially (this variant was solved quite long ago [43]) we use it to elaborate upon the principles in a simple setting.

Implementation of Catenable Lists in Functional Languages

We begin by noting that adding an element a to the front of a list X, without changing X, can be done in $O(1)$ time. We will denote this operation by $a :: X$. However, adding an element to the end of X involves a destructive update. The standard solution is to store the list X as a pair of lists $\langle F, R\rangle$, with F representing an initial segment of X, and R representing the remainder of X, stored in reversed order. Furthermore, we maintain the invariant that $|F| \geq |R|$.

To implement an *inject* or *tail* operation, we first obtain the pair $\langle F', R'\rangle$, which equals $\langle F, a :: R\rangle$ or $\langle tail(F), R\rangle$, as the case may be. If $|F'| \geq |R'|$, we return $\langle F', R'\rangle$. Otherwise we return $\langle F' +\!\!+ reverse(R'), [\,]\rangle$, where $X +\!\!+ Y$ appends Y to X and $reverse(X)$ returns the reverse of list X. The functions $+\!\!+$ and *reverse* are defined as follows:

$$\begin{aligned} X +\!\!+ Y &= Y \text{ if } X = [\,], \\ &= head(X) :: (tail(X) +\!\!+ Y) \text{ otherwise}. \\ reverse(X) &= rev(X, [\,]), \text{ where:} \\ rev(X, Y) &= Y \text{ if } X = [\,], \\ &= rev(tail(X), head(X) :: Y) \text{ otherwise}. \end{aligned}$$

The running time of $X +\!\!+ Y$ is clearly $O(|X|)$, as is the running time of $reverse(X)$. Although the amortized cost of *inject* can be easily seen to be $O(1)$ in an ephemeral setting, the efficiency of this data structure may be much worse in a fully persistent setting, as discussed above.

If, however, the functional language supports *lazy evaluation* and *memoization* then this solution can be used as is. Lazy evaluation refers to delaying calculating the value of expressions as much as possible. If lazy evaluation is used, the expression $F' +\!\!+ reverse(R')$ is not evaluated until we try to determine its *head* or *tail*. Even then, the expression is not fully evaluated unless F' is empty, and the list $tail(F' +\!\!+ reverse(R'))$ remains represented internally as $tail(F') +\!\!+ reverse(R')$. Note that *reverse* cannot be computed incrementally like $+\!\!+$: once started, a call to *reverse* must run to completion before the first element in the reversed list is available. Memoization involves caching the result of a delayed computation the first time it is executed, so that the next time the same computation needs to be performed, it can be looked up rather than recomputed.

The amortized analysis uses a "debit" argument. Each element of a list is associated with a number of debits, which will be proportional to the amount of delayed work which must be done before this element can be accessed. Each operation can "discharge" $O(1)$ debits, i.e., when the delayed work is eventually done, a cost proportional to the number of debits discharged by an operation will be charged to this operation. The goal will be to prove that all debits on an element will have been discharged before it is accessed. However, once the work has been done, the result is memoized and any other thread of execution which require this result will simply use the memoized version at no extra cost. The debits satisfy the following invariant. For $i = 0, 1, \ldots$, let $d_i \geq 0$ denote the number of debits on the ith element of any list $\langle F, R\rangle$. Then:

$$\sum_{j=0}^{i} d_i \leq \min\{2i, |F| - |R|\}, \text{ for } i = 0, 1, \ldots$$

Note that the first (zeroth) element on the list always has zero debits on it, and so *head* only accesses elements whose debits have been paid. If no list reversal takes place during a *tail* operation, the value of $|F|$ goes down by one, as does the index of each remaining element in the list (i.e., the old $(i + 1)$st element will now be the new ith element). It suffices to pay of $O(1)$ debits at each of the first two locations in the list where the invariant is violated. A new element *inject*ed into list R has no delayed computation associated with it, and is give zero debits. The violations of the invariant caused by an *inject* where no list reversal occurs are handled as above. As a list reversal occurs only if $m = |F| = |R|$ before the operation which caused the reversal, the invariant implies that all debits on the front list have been paid off before

the reversal. Note that there are no debits on the rear list. After the reversal, one debit is placed on each element of the old front list (to pay for the delayed incremental $++$ operation) and $m+1$ debits are placed on the first element of the reversed list (to pay for the reversal), and zero on the remaining elements of the remaining elements of the reversed list, as there is no further delayed computation associated with them. It is easy to verify that the invariant is still satisfied after discharging $O(1)$ debits.

To add catenation to Okasaki's algorithm, a list is represented as a tree whose left-to-right pre-order traversal gives the list being represented. The children of a node are stored in a functional queue as described above. In order to perform $catenate(X, Y)$ the operation $link(X, Y)$ is performed, which adds root of the tree Y is added to the end of the child queue for the root of the tree X. The operation $tail(X)$ removes the root of the tree for X. If its children of the root are $X_1, \ldots, X_m$ then the new list is given by $link(X_1, link(X_2, \ldots, link(X_{m-1}, X_m)))$. By executing the *link* operations in a lazy fashion and using memoization, all operations can be made to run in $O(1)$ time.

Purely Functional Catenable Lists

In this section we will describe the techniques used by Kaplan and Tarjan to obtain a purely functional queue. The critical difference is that we cannot assume memoization in a purely functional setting. This appears to mean that the data structures once again have to support each operation in worst-case constant time. The main ideas used by Kaplan and Tarjan are those of *data-structural bootstrapping* and *recursive slowdown*. Data-structural bootstrapping was introduced by [29] and refers to allowing a data structure to use the same data structure as a recursive sub-structure.

Recursive slowdown can be viewed as running the recursive data structures "at a slower speed." We will now give a very simple illustration of recursive slowdown. Let a 2-queue be a data structure which allows the *tail* and *inject* operations, but holds a maximum of 2 elements. Note that the bound on the size means that all operations on a 2-queue can be can be trivially implemented in constant time, by copying the entire queue each time. A queue Q consists of three components: a front queue $f(Q)$, which is a 2-queue, a rear queue $r(Q)$, which is also a 2-queue, and a center queue $c(Q)$, which is a recursive queue, each element of which is a *pair* of elements of the top-level queue. We will ensure that at least one of $f(Q)$ is non-empty unless Q itself is empty.

The operations are handled as follows An *inject* adds an element to the end of $r(Q)$. If $r(Q)$ is full, then the two elements currently in $r(Q)$ are inserted as a pair into $c(Q)$ and the new element is inserted into $r(Q)$. Similarly, a *tail* operation attempts to remove the first element from $f(Q)$. If $f(Q)$ is empty then we extract the first pair from $c(Q)$, if $c(Q)$ is non-empty and place the second element from the pair into $f(Q)$, discarding the first element. If $c(Q)$ is also empty then we discard the first element from $r(Q)$.

The key to the complexity bound is that only every alternate *inject* or *tail* operation accesses $c(Q)$. Therefore, the recurrence giving the amortized running time $T(n)$ of operations on this data structure behaves roughly like $2T(n) = T(n/2) + k$ for some constant k. The term $T(n/2)$ represents the cost of performing an operation on $c(Q)$, since $c(Q)$ can contain at most $n/2$ pairs of elements, if n is the number of elements in Q as a whole. Rewriting this recurrence as $T(n) = \frac{1}{2}T(n/2) + k'$ and expanding gives that $T(n) = O(1)$ (even replacing $n/2$ by $n-1$ in the RHS gives $T(n) = O(1)$).

This data structure is not suitable for use in a persistent setting as a single operation may still take $\Theta(\log n)$ time. For example, if $r(Q)$, $r(c(Q))$, $r(c(c(Q)))\ldots$ each contain two elements, then a single *inject* at the top level would cause changes at all $\Theta(\log n)$ levels of recursion. This is analogous to carry propagation in binary numbers—if we define a binary number where for $i = 0, 1, \ldots$, the ith digit is 0 if $c^i(Q)$ contains one element and 1 if it contains two (the 0th digit is considered to be the least significant) then each *inject* can be viewed as adding 1 to this binary number. In the worst case, adding 1 to a binary number can take time proportional to the number of digits.

A different number system can alleviate this problem. Consider a number system where the ith digit still has weight 2^i, as in the binary system, but where digits can take the value 0, 1 or 2 [19]. Further, we require that any pair of 2's be separated by at least one 0 and that the rightmost digit, which is not a 1 is a 0. This

number system is *redundant*, i.e., a number can be represented in more than one way (the decimal number 4, for example, can be represented as either 100 or 020). Using this number system, we can increment a value by one in constant time by the following rules: (i) add one by changing the rightmost 0 to a 1, or by changing x 1 to $(x+1)$ 0; then (ii) fixing the rightmost 2 by changing x 2 to $(x+1)$ 0. Now we increase the capacity of $r(Q)$ to 3 elements, and let a queue containing i elements represent the digit $i-1$. We then perform an *inject* in $O(1)$ worst-case time by simulating the algorithm above for incrementing a counter. Using a similar idea to make *tail* run in $O(1)$ time, we can make all operations run in $O(1)$ time.

In the version of their data structure which supports catenation, Kaplan and Tarjan again let a queue be represented by three queues $f(Q)$, $c(Q)$, and $r(Q)$, where $f(Q)$ and $r(Q)$ are of constant size as before. The center queue $c(Q)$ in this case holds either (i) a queue of constant size containing at least two elements or (ii) a pair whose first element is a queue as in (i) and whose second element is a catenable queue. To execute *catenate*(X, Y), the general aim is to first try and combine $r(X)$ and $f(Y)$ into a single queue. When this is possible, a pair consisting of the resulting queue and $c(Y)$ is *inject*ed into $c(X)$. Otherwise, $r(X)$ is *inject*ed into $c(X)$ and the pair $\langle f(X), c(X)\rangle$ is also *inject*ed into $c(X)$. Details can be found in [50].

Other Data Structures

A *deque* is a list which allows single elements to be added or removed from the front or the rear of the list. Efficient persistent deques implemented in functional languages were studied in [18, 35, 42], with some of these supporting additional operations. A *catenable deque* allows all the operations above defined for a catenable list, but also allows deletion of a single element from the end of the list. Kaplan and Tarjan [50] have stated that their technique extends to give purely functional catenable deques with constant worst-case time per operation. Other data structures which can be implemented in functional languages include finger search trees [51] and worst-case optimal priority queues [12]. (See [72, 73] for yet more examples.)

5.7 Research Issues and Summary

In this chapter we have described the most efficient known algorithms for set union and persistency.

Most of the set union algorithms we have described are optimal with respect to a certain model of computation (e.g., pointer machines with or without the separability assumption, random access machines). There are still several open problems in all the models of computation we have considered. First, there are no lower bounds for some of the set union problems on intervals: for instance, for nonseparable pointer algorithms we are only aware of the trivial lower bound for interval union–find. This problem requires $\Theta(1)$ amortized time on a random access machine as shown by Gabow and Tarjan [34]. Second, it is still open whether in the amortized and the single operation worst-case complexity of the set union problems with deunions or backtracking can be improved for nonseparable pointer algorithms or in the cell probe model of computation.

5.8 Defining Terms

Cell probe model: Model of computation where the cost of a computation is measured by the total number of memory accesses to a random access memory with $\lceil \log n \rceil$ bits cell size. All other computations are not accounted for and are considered to be free.

Persistent data structure: A data structure that preserves its old versions. Partially persistent data structures allow updates to their latest version only, while all versions of the data structure may be queried. Fully persistent data structures allow all their existing versions to be queried or updated.

Pointer machine: Model of computation whose storage consists of an unbounded collection of registers (or records) connected by pointers. Each register can contain an arbitrary amount of additional information, but no arithmetic is allowed to compute the address of a register. The only possibility to access a register is by following pointers.

Purely functional language: A language that does not allow any destructive operation—one which overwrites data—such as the assignment operation. Purely functional languages are side-effect-free, i.e., invoking a function has no effect other than computing the value returned by the function.

Random access machine: Model of computation whose memory consists of an unbounded sequence of registers, each of which is capable of holding an integer. In this model, arithmetic operations are allowed to compute the address of a memory register.

Separability: Assumption that defines two different classes of pointer-based algorithms for set union problems. An algorithm is separable if after each operation, its data structures can be partitioned into disjoint subgraphs so that each subgraph corresponds to exactly one current set, and no edge leads from one subgraph to another.

Acknowledgments

The work of the first author was supported in part by the Commission of the European Communities under project no. 20244 (ALCOM-IT) and by a research grant from University of Venice "Ca' Foscari." The work of the second author was supported in part by a Nuffield Foundation Award for Newly-Appointed Lecturers in the Sciences.

References

[1] Ackermann, W., Zum Hilbertshen Aufbau der reelen Zahlen, *Math. Ann.*, 99, 118–133, 1928.

[2] Adel'son-Vel'skii, G.M, and Landis, E.M., An algorithm for the organization of information, *Dokl. Akad. Nauk SSSR,* 146, 263–266, (in Russian), 1962.

[3] Aho, A.V., Hopcroft, J.E., and Ullman, J.D., On computing least common ancestors in trees, *Proc. 5th Annual ACM Symposium on Theory of Computing,* 253–265, 1973.

[4] Aho, A.V., Hopcroft, J.E., and Ullman, J.D., *The Design and Analysis of Computer Algorithms,* Addison-Wesley, Reading, MA, 1974.

[5] Aho, A.V., Hopcroft, J.E., and Ullman, J.D., *Data Structures and Algorithms,* Addison-Wesley, Reading, MA, 1983.

[6] Ahuja, R.K., Mehlhorn, K., Orlin, J.B., and Tarjan, R.E., Faster algorithms for the shortest path problem, *J. Assoc. Comput. Mach.,* 37, 213–223, 1990.

[7] Aït-Kaci, H., An algebraic semantics approach to the effective resolution of type equations, *Theoret. Comput. Sci.,* 45, 1986.

[8] Aït-Kaci, H. and Nasr, R., LOGIN: A logic programming language with built-in inheritance, *J. Logic Program.,* 3, 1986.

[9] Apostolico, A., Italiano, G.F., Gambosi, G., and Talamo, M., The set union problem with unlimited backtracking, *SIAM J. Computing,* 23, 50–70, 1994.

[10] Arden, B.W., Galler, B.A., and Graham, R.M., An algorithm for equivalence declarations, *Comm. ACM,* 4, 310–314, 1961.

[11] Brodal, G.S., Partially persistent data structures of bounded degree with constant update time, Technical Report BRICS RS-94-35, BRICS, Department of Computer Science, University of Aarhus, 1994.
[12] Brodal, G.S. and Okasaki, C., Optimal purely functional priority queues, *J. Functional Programming,* to appear, 1996.
[13] Ben-Amram, A.M. and Galil, Z., On pointers versus addresses, *J. Assoc. Comput. Mach.,* 39, 617–648, 1992.
[14] Blum, N., On the single operation worst-case time complexity of the disjoint set union problem, *SIAM J. Comput.,* 15, 1021–1024, 1986.
[15] Blum, N. and Rochow, H., A lower bound on the single-operation worst-case time complexity of the union–find problem on intervals, *Inform. Proc. Lett.,* 51, 57–60, 1994.
[16] Buchsbaum, A.L. and Tarjan, R.E., Confluently persistent deques via data-structural bootstrapping, *J. Algorithms,* 18, 513–547, 1995.
[17] Chazelle, B., How to search in history, *Information and Control,* 64, 77–99, 1985.
[18] Chuang, T-R. and Goldberg, B., Real-time deques, multihead Turing machines, and purely functional programming, *Proceedings of the Conference of Functional Programming and Computer Architecture,* 289–298, 1992.
[19] Clancy, M.J. and Knuth, D.E., A programming and problem-solving seminar, Technical Report STAN-CS-77-606, Stanford University, 1977.
[20] Cole, R., Searching and storing similar lists, *J. Algorithms,* 7, 202–220, 1986.
[21] Cormen, T.H., Leiserson, C.E., and Rivest, R.L., *Introduction to Algorithms,* MIT Press, Cambridge, MA, 1990.
[22] Dietz, P.F., Maintaining order in a linked list, *Proc. 14th Annual ACM Symposium on Theory of Computing,* 122–127, 1982.
[23] Dietz, P.F., Fully persistent arrays, *Proc. Workshop on Algorithms and Data Structures (WADS '89), Lecture Notes in Computer Science,* 382, Springer-Verlag, Berlin, 67–74, 1989.
[24] Dietz, P.F. and Sleator, D.D., Two algorithms for maintaining order in a list, *Proc. 19th Annual ACM Symposium on Theory of Computing,* 365–372, 1987.
[25] Dietz, P.F. and Raman, R., Persistence, amortization and randomization, *Proc. 2nd Annual ACM-SIAM Symposium on Discrete Algorithms,* 77–87, 1991.
[26] Dietz, P.F. and Raman, R., Persistence, amortization and parallelization: On some combinatorial games and their applications, *Proc. Workshop on Algorithms and Data Structures (WADS '93), Lecture Notes in Computer Science,* 709, Springer-Verlag, Berlin, 289–301, 1993.
[27] Dietzfelbinger, M., Karlin, A., Mehlhorn, K., Meyer auf der Heide, F., Rohnhert, H., and Tarjan, R.E., Dynamic perfect hashing: upper and lower bounds, *Proc. 29th Annual IEEE Conference on the Foundations of Computer Science,* 1988, 524–531. The application to partial persistence was mentioned in the talk. A revised version of the paper appeared in *SIAM J. Computing,* 23, 738–761, 1994.
[28] Driscoll, J.R., Sarnak, N., Sleator, D.D., and Tarjan, R.E., Making data structures persistent, *J. Computer Systems Sci.,* 38, 86–124, 1989.
[29] Driscoll, J.R., Sleator, D.D.K., and Tarjan, R.E., Fully persistent lists with catenation, *J. ACM,* 41, 943–959, 1994.
[30] Dobkin, D.P. and Munro, J.I., Efficient uses of the past, *J. Algorithms,* 6, 455–465, 1985.
[31] Fischer, M.J., Efficiency of equivalence algorithms, in *Complexity of Computer Computations,* R.E. Miller and J.W. Thatcher, Eds., Plenum Press, New York, 153–168.
[32] Fredman, M.L. and Saks, M.E., The cell probe complexity of dynamic data structures, *Proc. 21st Annual ACM Symposium on Theory of Computing,* 345–354, 1989.
[33] Gabow, H.N., A scaling algorithm for weighted matching on general graphs, *Proc. 26th Annual Symposium on Foundations of Computer Science,* 90–100, 1985.

[34] Gabow, H.N. and Tarjan, R.E., A linear time algorithm for a special case of disjoint set union, *J. Comput. Sys. Sci.,* 30, 209–221, 1985.
[35] Gajewska, H. and Tarjan, R.E., Deques with heap order, *Information Processing Letters,* 22, 197–200, 1986.
[36] Galil, Z. and Italiano, G.F., A note on set union with arbitrary deunions, *Information Processing Letters,* 37, 331–335, 1991.
[37] Galil, Z. and Italiano, G.F., Data structures and algorithms for disjoint set union problems, *ACM Computing Surveys,* 23, 319–344, 1991.
[38] Galler, B.A. and Fischer, M., An improved equivalence algorithm, *Comm. ACM,* 7, 301–303, 1964.
[39] Gambosi, G., Italiano, G.F., and Talamo, M., Worst-case analysis of the set union problem with extended backtracking, *Theoret. Comput. Sci.,* 68, 57–70, 1989.
[40] Hogger, G.J., *Introduction to Logic Programming,* Academic Press, 1984.
[41] Italiano, G.F. and Sarnak, N., Fully persistent data structures for disjoint set union problems, *Proc. Workshop on Algorithms and Data Structures (WADS '91), Lecture Notes in Computer Science,* 519, Springer-Verlag, Berlin, 449–460, 1991.
[42] Hood, R., *The Efficient Implementation of Very-High-Level Programming Language Constructs,* Ph.D. Thesis, Cornell University, 1982.
[43] Hood, R. and Melville, R., Real-time operations in pure Lisp, *Information Processing Letters,* 13, 50–53, 1981.
[44] Hopcroft, J.E. and Karp, R.M., An algorithm for testing the equivalence of finite automata, TR-71-114, Dept. of Computer Science, Cornell University, Ithaca, NY, 1971.
[45] Hopcroft, J.E. and Ullman, J.D., Set merging algorithms, *SIAM J. Comput.,* 2, 294–303, 1973.
[46] Hudak, P., Jones, S.P., Wadler, P., Boutel, B., Fairbairn, J., Fasel, J., Guzman, M.M., Hammond, K., Hughes, J., Johnsson, T., Kieburtz, D., Nikhil, R., Partain, W., and Peterson, J., Report on the functional programming language Haskell, version 1.2, *SIGPLAN Notices,* 27, 1992.
[47] Huet, G., *Resolutions d'equations dans les langages d'ordre* 1, 2, . . . ω, Ph.D. Dissertation, Univ. de Paris VII, France, 1976.
[48] Hunt, J.W. and Szymanski, T.G., A fast algorithm for computing longest common subsequences, *Comm. Assoc. Comput. Mach.,* 20, 350–353, 1977.
[49] Imai, T. and Asano, T., Dynamic segment intersection with applications, *J. Algorithms,* 8, 1–18, 1987.
[50] Kaplan, H. and Tarjan, R.E., Persistent lists with catenation via recursive slow-down, *Proc. 27th Annual ACM Symposium on the Theory of Computing,* 93–102, 1995.
[51] Kaplan, H. Tarjan, R.E., Purely functional representations of catenable sorted lists, *Proc. 28th Annual ACM Symposium on the Theory of Computing,* 202–211, 1996.
[52] Karlsson, R.G., Algorithms in a restricted universe, Technical Report CS-84-50, Department of Computer Science, University of Waterloo, 1984.
[53] Kerschenbaum, A. and van Slyke, R., Computing minimum spanning trees efficiently, *Proc. 25th Annual Conf. of the ACM,* 518–527, 1972.
[54] Knuth, D.E., *The Art of Computer Programming,* Vol. 1: *Fundamental Algorithms.* Addison-Wesley, Reading, MA, 1968.
[55] Kolmogorv, A.N., On the notion of algorithm, *Uspehi Mat. Nauk.,* 8, 175–176, 1953.
[56] La Poutré, J.A., Lower bounds for the union–find and the split–find problem on pointer machines, *Proc. 22nd Annual ACM Symposium on Theory of Computing,* 34–44, 1990.
[57] Levcopolous, C. and Overmars, M.H., A balanced search tree with $O(1)$ worst-case update time, *Acta Informatica,* 26, 269–278, 1988.
[58] Loebl, M. and Nešetřil, J., Linearity and unprovability of set union problem strategies, *Proc. 20th Annual ACM Symposium on Theory of Computing,* 360–366, 1988.

[59] Mannila, H. and Ukkonen, E., The set union problem with backtracking, *Proc. 13th International Colloquium on Automata, Languages and Programming* (ICALP 86), *Lecture Notes in Computer Science,* 226, Springer-Verlag, Berlin, 236–243, 1986.
[60] Mannila, M. and Ukkonen, E., On the complexity of unification sequences, *Proc. 3rd International Conference on Logic Programming, Lecture Notes in Computer Science,* 225, Springer-Verlag, Berlin, 122–133, 1986.
[61] Mannila, H. and Ukkonen, E., Timestamped term representation for implementing Prolog, *Proc. 3rd IEEE Conference on Logic Programming,* 159–167, 1986.
[62] Mannila, H. and Ukkonen, E., Space-time optimal algorithms for the set union problem with backtracking. Technical Report C-1987-80, Department of Computer Science, University of Helsinki, Finland.
[63] Mannila, H. and Ukkonen, E., Time parameter and arbitrary deunions in the set union problem, *Proc. 1st Scandinavian Workshop on Algorithm Theory* (SWAT 88), *Lecture Notes in Computer Science,* 318, Springer-Verlag, Berlin, 34–42, 1988.
[64] McCarthy, J., Recursive functions of symbolic expressions and their computation by machine, *Commun. ACM,* 7, 184–195, 1960.
[65] Mehlhorn, K., *Data Structures and Algorithms,* Vol. 1: *Sorting and Searching,* Springer-Verlag, Berlin, 1984.
[66] Mehlhorn, K., *Data Structures and Algorithms,* Vol. 2: *Graph Algorithms and NP-Completeness,* Springer-Verlag, Berlin, 1984.
[67] Mehlhorn, K., *Data Structures and Algorithms,* Vol. 3: *Multidimensional Searching and Computational Geometry,* Springer-Verlag, Berlin, 1984.
[68] Mehlhorn, K. and Näher, S., Dynamic fractional cascading, *Algorithmica* 5, 215–241, 1990.
[69] Mehlhorn, K., Näher, S., and Alt, H., A lower bound for the complexity of the union–split–find problem, *SIAM J. Comput.,* 17, 1093–1102, 1990.
[70] Milner, R., Tofte, M., and Harper, R., *The Definition of Standard ML,* MIT Press, Cambridge, MA, 1990.
[71] Okasaki, C., Amortization, lazy evaluation, and persistence: Lists with catenation via lazy linking, *Proc. 36th Annual Symposium on Foundations of Computer Science,* 646-654, 1995.
[72] Okasaki, C., Functional Data Structures, in *Advanced Functional Programming, Lecture Notes in Computer Science,* 1129, Springer-Verlag, Berlin, 67–74, 1996.
[73] Okasaki, C., The role of lazy evaluation in amortized data structures, *Proc. 1996 ACM SIGPLAN International Conference on Functional Programming,* 62–72, 1996.
[74] Reps, T., Titelbaum, T., and Demers, A., Incremental context-dependent analysis for language-based editors, *ACM Transactions on Programming Languages and Systems,* 5, 449–477, 1983.
[75] Sarnak, N., *Persistent Data Structures,* Ph.D. Thesis, Department of Computer Science, New York University, 1986.
[76] Sarnak, N. and Tarjan, R.E., Planar point location using persistent search trees, *Commun. ACM,* 28, 669–679, 1986.
[77] Schönage, A., Storage modification machines, *SIAM J. Comput.,* 9, 490–508, 1980.
[78] Stearns, R.E. and Lewis, P.M., Property grammars and table machines, *Information and Control,* 14, 524–549, 1969.
[79] Stearns, R.E. and Rosenkrantz, P.M., Table machine simulation, *Conf. Rec. IEEE 10th Annual Symp. on Switching and Automata Theory,* 118–128, 1969.
[80] Steele Jr., G.L., *Common Lisp: The Language,* Digital Press, Bedford, MA, 1984.
[81] Swart, G.F., Efficient algorithms for computing geometric intersections, Technical Report 85-01-02, Department of Computer Science, University of Washington, Seattle, WA, 1985.
[82] Tarjan, R.E., Testing flow graph reducibility, *Proc. 5th Annual ACM Symp. on Theory of Computing,* 96–107, 1973.

[83] Tarjan, R.E., Finding dominators in directed graphs, *SIAM J. Comput.,* 3, 62–89, 1974.
[84] Tarjan, R.E., Efficiency of a good but not linear set union algorithm, *J. Assoc. Comput. Mach.,* 22, 215–225, 1975.
[85] Tarjan, R.E., A class of algorithms which require nonlinear time to maintain disjoint sets, *J. Comput. Sys. Sci.,* 18, 110–127, 1979.
[86] Tarjan, R.E., Application of path compression on balanced trees, *J. Assoc. Comput. Mach.,* 26, 690–715, 1979.
[87] Tarjan, R.E., *Data Structures and Network Algorithms,* SIAM, Philadelphia, PA, 1983.
[88] Tarjan, R.E., Amortized computational complexity, *SIAM J. Alg. Disc. Meth.,* 6, 306–318, 1985.
[89] Tarjan, R.E. and van Leeuwen, J., Worst-case analysis of set union algorithms, *J. Assoc. Comput. Mach.,* 31, 245–281, 1984.
[90] Tsakalidis, A.K., Maintaining order in a generalized linked list, *Acta Informatica,* 21, 101–112, 1984.
[91] van Emde Boas, P., Preserving order in a forest in less than logarithmic time and linear space, *Inform. Processing Lett.,* 6, 80–82, 1977.
[92] van Emde Boas, P., Kaas, R., and Zijlstra, E., Design and implementation of an efficient priority queue, *Math. Systems Theory,* 10, 99–127, 1977.
[93] van Leeuwen, J. and van der Weide, T., Alternative path compression techniques, Technical Report RUU-CS-77-3, Department of Computer Science, University of Utrecht, Utrecht, The Netherlands, 1977.
[94] van der Weide, T., *Data Structures: an Axiomatic Approach and the Use of Binomial Trees in Developing and Analyzing Algorithms,* Mathematisch Centrum, Amsterdam, The Netherlands, 1980.
[95] Vitter, J.S. and Simons, R.A., New classes for parallel complexity: A study of unification and other complete problems for P, *IEEE Trans. Comput. C-35.* 1989.
[96] Warren, D.H.D. and Pereira, L.M., Prolog—the language and its implementation compared with LISP, *ACM SIGPLAN Notices,* 12, 109–115, 1977.
[97] Westbrook, J. and Tarjan, R.E., Amortized analysis of algorithms for set union with backtracking, *SIAM J. Comput.,* 18, 1–11, 1989.
[98] Westbrook, J. and Tarjan, R.E., Maintaining bridge-connected and biconnected components on-line, *Algorithmica,* 7, 433–464, 1992.
[99] Yao, A.C., Should tables be sorted? *J. Assoc. Comput. Mach.,* 28, 615–628, 1981.

Further Information

Research on advanced algorithms and data structures is published in many computer science journals, including *Algorithmica, Journal of ACM, Journal of Algorithms,* and *SIAM Journal on Computing.* Work on data structures is published also in the proceedings of general theoretical computer science conferences, such as the "ACM Symposium on Theory of Computing (STOC)," and the "IEEE Symposium on Foundations of Computer Science (FOCS)." More specialized conferences devoted exclusively to algorithms are the "ACM–SIAM Symposium on Discrete Algorithms (SODA)" and the "European Symposium on Algorithms (ESA)." Online bibliographies for many of these conferences and journals can be found on the World Wide Web.

Galil and Italiano [37] provide useful summaries on the state of the art in set union data structures. A in-depth study of implementing data structures in functional languages is given in [72].

6

Basic Graph Algorithms

Samir Khuller
University of Maryland

Balaji Raghavachari
University of Texas at Dallas

6.1 Introduction

Graphs provide a powerful tool to model objects and relationships between objects. The study of graphs dates back to the 18th century, when Euler defined the *Königsberg bridge* problem, and since then has been pursued by many researchers. Graphs can be used to model problems in many areas such as transportation, scheduling, networks, robotics, VLSI design, compilers, mathematical biology and software engineering. Many optimization problems from these and other diverse areas can be phrased in graph theoretic terms, leading to algorithmic questions about graphs.

Graphs are defined by a set of vertices and a set of edges, where each edge connects two vertices. Graphs are further classified into directed and undirected graphs, depending on whether their edges are directed

or not. An important subclass of directed graphs that arises in many applications, such as precedence constrained scheduling problems, are **directed acyclic graphs** (DAG). Interesting subclasses of undirected graphs include **trees, bipartite graphs,** and **planar graphs.**

In this chapter, we focus on a few basic problems and algorithms dealing with graphs. Other chapters in this handbook provide details on specific algorithmic techniques and problem areas dealing with graphs, e.g., randomized algorithms (Chapter 15), combinatorial algorithms (Chapter 7), dynamic graph algorithms (Chapter 8), graph drawing (Chapter 9), and approximation algorithms (Chapter 34). Pointers into the literature are provided for various algorithmic results about graphs that are not covered in depth in this chapter.

6.2 Preliminaries

An undirected *graph* $G = (V, E)$ is defined as a set V of *vertices* and a set E of *edges.* An edge $e = (u, v)$ is an unordered pair of vertices. A *directed graph* is defined similarly, except that its edges are ordered pairs of vertices, i.e., for a directed graph, $E \subseteq V \times V$. The terms *nodes* and vertices are used interchangeably. In this chapter, it is assumed that the graph has neither self loops — edges of the form (v, v) — nor multiple edges connecting two given vertices. The number of vertices of a graph, $|V|$, is often denoted by n. A graph is a **sparse graph** if $|E| \ll |V|^2$.

Bipartite graphs form a subclass of graphs and are defined as follows. A graph $G = (V, E)$ is bipartite if the vertex set V can be partitioned into two sets X and Y such that $E \subseteq X \times Y$. In other words, each edge of G connects a vertex in X with a vertex in Y. Such a graph is denoted by $G = (X, Y, E)$. Since bipartite graphs occur commonly in practice, often algorithms are designed specially for them. *Planar graphs* are graphs that can be drawn in the plane without any two edges crossing each other. Let K_n be the complete graph on n vertices, and $K_{x,y}$ be the complete bipartite graph with x and y vertices in either side of the bipartite graph respectively. A *homeomorph* of a graph is obtained by subdividing an edge by adding new vertices.

A vertex w is *adjacent* to another vertex v if $(v, w) \in E$. An edge (v, w) is said to be *incident* to vertices v and w. The *neighbors* of a vertex v are all vertices $w \in V$ such that $(v, w) \in E$. The number of edges incident to a vertex is called its **degree.** For a directed graph, if (v, w) is an edge, then we say that the edge goes from v to w. The *out-degree* of a vertex v is the number of edges from v to other vertices. The *in-degree* of v is the number of edges from other vertices to v.

A **path** $p = [v_0, v_1, \ldots, v_k]$ from v_0 to v_k is a sequence of vertices such that (v_i, v_{i+1}) is an edge in the graph for $0 \leq i < k$. Any edge may be used only once in a path. An *intermediate vertex* (or internal vertex) on a path $P[u, v]$, a path from u to v, is a vertex incident to the path, other than u and v. A path is *simple* if all of its internal vertices are distinct. A **cycle** is a path whose end vertices are the same, i.e., $v_0 = v_k$. A **walk** $w = [v_0, v_1, \ldots, v_k]$ from v_0 to v_k is a sequence of vertices such that (v_i, v_{i+1}) is an edge in the graph for $0 \leq i < k$. A *closed walk* is one in which $v_0 = v_k$. A graph is said to be *connected* if there is a path between every pair of vertices. A directed graph is said to be **strongly connected** if there is a path between every pair of vertices in each direction. An acyclic, undirected graph is a **forest,** and a *tree* is a connected forest. A *maximal forest* F of a graph G is a forest of G such that the addition of any other edge of G to F introduces a cycle. A directed graph that does not have any cycles is known as a *directed acyclic graph* (DAG). Consider a binary relation C between the vertices of an undirected graph G such that for any two vertices u and v, uCv if and only if there is a path in G between u and v. C is an equivalence relation, and it partitions the vertices of G into equivalence classes, known as the *connected components* of G.

Graphs may have weights associated with edges or vertices. In the case of edge-weighted graphs (edge weights denoting lengths), the *distance* between two vertices is the length of a shortest path between them, where the length of a path is defined as the sum of the weights of its edges. The *diameter* of a graph is the maximum of the distance between all pairs of vertices.

There are two convenient ways of representing graphs on computers. In the *adjacency list* representation, each vertex has a linked list; there is one entry in the list for each of its adjacent vertices. The graph is thus, represented as an array of linked lists, one list for each vertex. This representation uses $O(|V| + |E|)$ storage, which is good for sparse graphs. Such a storage scheme allows one to scan all vertices adjacent to a given vertex in time proportional to the degree of the vertex. In the *adjacency matrix* representation, an $n \times n$ array is used to represent the graph. The $[i, j]$ entry of this array is 1 if the graph has an edge between vertices i and j, and 0 otherwise. This representation permits one to test if there is an edge between any pair of vertices in constant time. Both these representation schemes extend naturally to represent directed graphs. For all algorithms in this chapter except the all-pairs shortest paths problem, it is assumed that the given graph is represented by an adjacency list.

Section 6.3 discusses various tree traversal algorithms. Sections 6.4 and 6.5 discuss depth-first and breadth-first search techniques. Section 6.6 discusses the single source shortest-path problem. Section 6.7 discusses **minimum spanning trees.** Section 6.8 discusses some traversal problems in graphs. Section 6.9 discusses various topics such as **planar graphs,** graph coloring, light approximate shortest path trees and network decomposition, and Section 6.10 concludes with some pointers to current research on graph algorithms.

6.3 Tree Traversals

A tree is *rooted* if one of its vertices is designated as the root vertex and all edges of the tree are oriented (directed) to point away from the root. In a rooted tree, there is a directed path from the root to any vertex in the tree. For any directed edge (u, v) in a rooted tree, u is v's *parent* and v is u's *child.* The *descendants* of a vertex w are all vertices in the tree (including w) that are reachable by directed paths starting at w. The *ancestors* of a vertex w are those vertices for which w is a descendant. Vertices that have no children are called **leaves.** A *binary tree* is a special case of a rooted tree in which each node has at most two children, namely the left child and the right child. The trees rooted at the two children of a node are called the *left subtree* and *right subtree.*

In this section we study techniques for processing the vertices of a given binary tree in various orders. It is assumed that each vertex of the binary tree is represented by a record that contains fields to hold attributes of that vertex and two special fields *left* and *right* that point to its left and right subtree respectively. Given a pointer to a record, the notation used for accessing its fields is similar to that used in the *C* programming language.

The three major tree traversal techniques are *preorder, inorder,* and *postorder.* These techniques are used as procedures in many tree algorithms where the vertices of the tree have to be processed in a specific order. In a preorder traversal, the root of any subtree has to be processed *before* any of its descendants. In a postorder traversal, the root of any subtree has to be processed *after* all of its descendants. In an inorder traversal, the root of a subtree is processed after all vertices in its left subtree have been processed, but before any of the vertices in its right subtree are processed. Preorder and postorder traversals generalize to arbitrary rooted trees. The algorithm below shows how postorder traversal of a binary tree can be used to count the number of descendants of each node and store the value in that node. The algorithm runs in linear time in the size of the tree.

```
PostOrder (T)
1  if T ≠ nil then
2     lc ← PostOrder(T → left).
3     rc ← PostOrder(T → right).
4     T → desc ← lc + rc + 1.
5     return (T → desc).
```

```
6   else
7       return 0.
8   end-if
end-proc
```

6.4 Depth-First Search

Depth-first search (DFS) is a fundamental graph searching technique developed by Hopcroft and Tarjan [16, 27]. Similar graph searching techniques were given earlier by Tremaux [8]. The structure of DFS enables efficient algorithms for many other graph problems such as biconnectivity, triconnectivity, and planarity [8].

The algorithm first initializes all vertices of the graph as being unvisited. Processing of the graph starts from an arbitrary vertex, known as the *root* vertex. Each vertex is processed when it is first discovered (also referred to as *visiting* a vertex). It is first marked as visited, and its adjacency list is then scanned for unvisited vertices. Each time an unvisited vertex is discovered, it is processed recursively by DFS. After a node's entire adjacency list has been explored, that instance of the DFS procedure returns. This procedure eventually visits all vertices that are in the same connected component of the root vertex. Once DFS terminates, if there are still any unvisited vertices left in the graph, one of them is chosen as the root and the same procedure is repeated.

The set of edges that led to the discovery of new vertices forms a maximal forest of the graph, known as the **DFS forest.** The algorithm keeps track of this forest using parent pointers; an array element $p[v]$ stores the parent of vertex v in the tree. In each connected component, only the root vertex has a *nil* parent in the DFS tree.

The DFS Algorithm

DFS is illustrated using an algorithm that assigns labels to vertices such that vertices in the same component receive the same label, a useful preprocessing step in many problems. Each time the algorithm processes a new component, it numbers its vertices with a new label.

```
DFS-Connected-Component (G)
1   c ← 0.
2   for all vertices v in G do
3       visited[v] ← false.
4       finished[v] ← false.
5       p[v] ← nil.
6   end-for
7   for all vertices v in G do
8       if not visited[v] then
9           c ← c + 1.
10          DFS (v, c).
11      end-if
12  end-for
end-proc
```

```
DFS (v, c)
1   visited[v] ← true.
2   component[v] ← c.
3   for all vertices w in adj[v] do
4      if not visited[w] then
5         p[w] ← v.
6         DFS (w, c).
7      end-if
8   end-for
9   finished[v] ← true.
end-proc
```

Sample Execution

Figure 6.1 shows a graph having two connected components. DFS started execution at vertex a, and the DFS forest is shown on the right. DFS visited the vertices b, d, c, e, and f, in that order. It then continued with vertices g, h, and i. In each case, the recursive call returned when the vertex has no more unvisited neighbors. Edges (d, a), (c, a), (f, d), and (i, g) are called *back edges,* and these edges do not belong to the DFS forest.

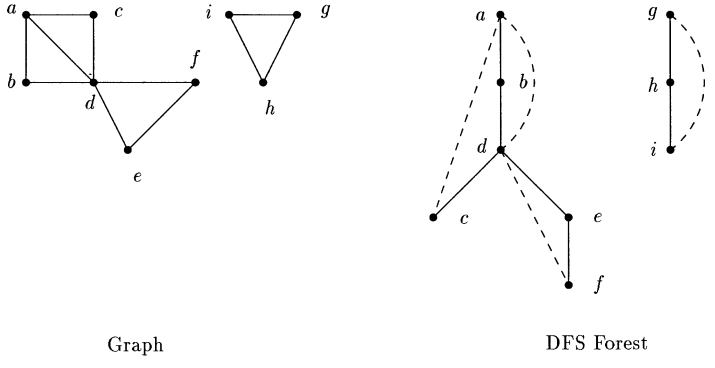


FIGURE 6.1 Sample execution of DFS on a graph having two connected components.

Analysis

A vertex v is processed as soon as it is encountered, and therefore at the start of DFS (v), *visited*$[v]$ is *false*. Since *visited*$[v]$ is set to *true* as soon as DFS starts execution, each vertex is visited exactly once. Depth-first search processes each edge of the graph exactly twice, once from each of its incident vertices. Since the algorithm spends constant time processing each edge of G, it runs in $O(|V| + |E|)$ time.

Classification of Edges

In the following discussion, there is no loss of generality in assuming that the input graph is connected. For a rooted DFS tree, vertices u and v are said to be *related,* if either u is an ancestor of v, or vice versa.

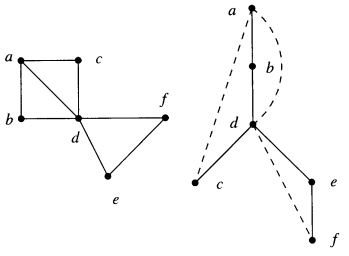
DFS is useful due to the special nature by which the edges of the graph may be classified with respect to a DFS tree. Note that the DFS tree is not unique, and which edges are added to the tree depends on

the order in which edges are explored while executing DFS. Edges of the DFS tree are known as *tree edges.* All other edges of the graph are known as *back edges,* and it can be shown that for any edge (u, v), u and v must be related. The graph does *not* have any *cross edges* — edges that connect two vertices that are unrelated.

Articulation Vertices and Biconnected Components

One of the many applications of depth-first search is to decompose a graph into its biconnected components. In this section, it is assumed that the graph is connected. An **articulation vertex** (also known as **cut vertex**) is a vertex whose deletion along with its incident edges breaks up the remaining graph into two or more disconnected pieces. A graph is called *biconnected* if it has no articulation vertices. A *biconnected component* of a **connected graph** is a maximal subset of edges such that the corresponding induced subgraph is biconnected. Each edge of the graph belongs to exactly one biconnected component. Biconnected components can have cut vertices in common.

The graph in Fig. 6.2 has two biconnected components, formed by the edge sets $\{(a, b), (a, c), (a, d), (b, d), (c, d)\}$ and $\{(d, e), (d, f), (e, f)\}$. There is a single cut vertex d and it is shared by both biconnected components.



Vertex	dis	fin	$low(v)$
a	1	12	1
b	2	11	1
c	4	5	1
d	3	10	1
e	6	9	3
f	7	8	3

FIGURE 6.2 Identifying cut vertices.

We now discuss a linear-time algorithm, developed by Hopcroft and Tarjan [16, 27], to identify the cut vertices and biconnected components of a connected graph. The algorithm uses the global variable *time* that is incremented every time a new vertex is visited or when DFS finishes visiting a vertex. Time is initially 0, and is $2|V|$ when the algorithm finally terminates. The algorithm records the value of *time* when a variable v is first visited in the array location *dis[v]* and the value of *time* when *DFS(v)* completes execution in *fin[v]*. We refer to *dis[v]* and *fin[v]* as the *discovery time* and *finish time* of vertex v, respectively.

Let T be a DFS tree of the given graph G. The notion of $low(v)$ of a vertex v with respect to T is defined as follows.

$$low(v) = \min(dis[v], dis[w] : (u, w) \text{ is a back edge for some descendant } u \text{ of } v)$$

$low(v)$ of a vertex is the discovery number of the vertex closest to the root that can be reached from v by following zero or more tree edges downward, and at most one back edge upward. It captures how far high the subtree of T rooted at v can reach by using at most one back edge. Figure 6.2 shows an example of a graph, a DFS tree of the graph and a table listing the values of *dis, fin,* and *low* of each vertex corresponding to that DFS tree.

Let T be the DFS tree generated by the algorithm, and let r be its root vertex. First, r is a cut vertex if and only if it has two or more children. This follows from the fact that there are no cross edges with respect to a DFS tree. Therefore the removal of r from G disconnects the remaining graph into as many

components as the number of children of r. The *low* values of vertices can be used to find cut vertices that are non-root vertices in the DFS tree. Let $v \neq r$ be a vertex in G. The following theorem characterizes precisely when v is a cut vertex in G.

THEOREM 6.1 *Let T be a DFS tree of a connected graph G, and let v be a non-root vertex of T. Vertex v is a cut vertex of G if and only if there is a child w of v in T with $low(w) \geq dis[v]$.*

Computing *low* values of a vertex and identifying all the biconnected components of a graph can be done efficiently with a single depth-first search scan. The algorithm uses a stack of edges. When an edge is encountered for the first time it is pushed into the stack irrespective of whether it is a tree edge or a back edge. Each time a cut vertex v is identified because $low(w) \geq dis[v]$ (as in Theorem 6.1), the stack contains the edges of the biconnected component as a contiguous block, with the edge (v, w) at the bottom of this block. The algorithm pops the edges of this biconnected component from the stack, and sets $cut[v]$ to *true* to indicate that v is a cut vertex.

```
BICONNECTED COMPONENTS (G)
1  time ← 0.
2  MAKEEMPTYSTACK (S).
3  for each u ∈ V do
4     visited[u] ← false.
5     cut[u] ← false.
6     p[u] ← nil.
7  end-for
8  Let v be an arbitrary vertex, DFS(v).
end-proc

DFS (v)
1  visited[v] ← true.
2  time ← time + 1.
3  dis[v] ← time.
4  low[v] ← dis[v].
5  for all vertices w in adj[v] do
6     if not visited[w] then
7        PUSH (S, (v, w)).
8        p[w] ← v.
9        DFS(w).
10       if (low[w] ≥ dis[v]) then
11          if (dis[v] ≠ 1) then cut[v] ← true.        (* v is not the root *)
12          else if (dis[w] > 2) then cut[v] ← true. (* v is root, and has at least 2 children *)
13          end-if
14          OUTPUTCOMP(v, w).
15       end-if
16       low[v] ← min(low[v], low[w]).
17    else if (p[v] ≠ w and dis[w] < dis[v]) then
18       PUSH (S, (v, w)).
19       low[v] ← min(low[v], dis[w]).
20    end-if
21 end-for
```

```
22  time ← time + 1.
23  fin[v] ← time.
end-proc
```

```
OUTPUTCOMP(v, w)
1   PRINT ("New Biconnected Component Found").
2   repeat
3      e ← POP (S).
4      PRINT (e).
5   until (e = (v, w)).
end-proc
```

In the example shown in Fig. 6.2 when DFS(e) finishes execution and returns control to DFS(d), the algorithm discovers that d is a cut vertex because $low(e) \geq dis[d]$. At this time, the stack contains the edges (d, f), (e, f), and (d, e) at the top of the stack, which are output as one biconnected component.

Remarks: The notion of biconnectivity can be generalized to higher connectivities. A graph is said to be k-connected, if there is no subset of $(k-1)$ vertices whose removal will disconnect the graph. For example, a graph is triconnected if it does not have any separating pairs of vertices — pairs of vertices whose removal disconnects the graph. A linear time algorithm for testing whether a given graph is triconnected was given by Hopcroft and Tarjan [15]. An $O(|V|^2)$ algorithm for testing if a graph is k-connected for any constant k was given by Nagamochi and Ibaraki [25]. One can also define a corresponding notion of edge-connectivity, where edges are deleted from a graph rather than vertices. Galil and Italiano [11] showed how to reduce edge connectivity to vertex connectivity.

Directed Depth-First Search

The DFS algorithm extends naturally to directed graphs. Each vertex stores an adjacency list of its outgoing edges. During the processing of a vertex, the algorithm first marks the vertex as visited, and then scans its adjacency list for unvisited neighbors. Each time an unvisited vertex is discovered, it is processed recursively. Apart from tree edges and back edges (from vertices to their ancestors in the tree), directed graphs may also have *forward edges* (from vertices to their descendants) and *cross edges* (between unrelated vertices). There may be a cross edge (u, v) in the graph only if u is visited after the procedure call "DFS (v)" has completed execution. The following algorithm implements DFS in a directed graph. For each vertex v, the algorithm computes the discovery time of v ($dis[v]$) and the time at which DFS(v) finishes execution ($fin[v]$). In addition, each edge of the graph is classified as either (i) tree edge, or (ii) back edge, or (iii) forward edge, or (iv) cross edge, with respect to the depth-first forest generated.

```
DIRECTED DFS (G)
1   for all vertices v in G do
2      visited[v] ← false.
3      finished[v] ← false.
4      p[v] ← nil.
5   end-for
6   time ← 0.
```

```
7   for all vertices v in G do
8       if not visited[v] then
9           DFS (v).
10      end-if
11      end-for
end-proc
```

```
DFS (v)
1   visited[v] ← true.
2   time ← time + 1.
3   dis[v] ← time.
4   for all vertices w in adj[j] do
5       if not visited[w] then
6           p[w] ← v.
7           PRINT ("Edge from" v "to" w "is a Tree edge").
8           DFS (w).
9       else if not finished[w] then
10          PRINT ("Edge from" v "to" w "is a Back edge").
11      else if dis[v] < dis[w] then
12          PRINT ("Edge from" v "to" w "is a Forward edge").
13      else
14          PRINT ("Edge from" v "to" w "is a Cross edge").
15      end-if
16  end-for
17  finished[v] ← true.
18  time ← time + 1.
19  fin[v] ← time.
end-proc
```

Sample Execution

A sample execution of the directed DFS algorithm is shown in Fig. 6.3. DFS was started at vertex a, and the DFS forest is shown on the right. DFS visits vertices b, d, f, and c, in that order. DFS then returns and continues with e, and then g. From g, vertices h and i are visited in that order. Observe that (d, a) and (i, g) are back-edges. Edges (c, d), (e, d), and (e, f) are cross edges. There is a single forward edge (g, i).

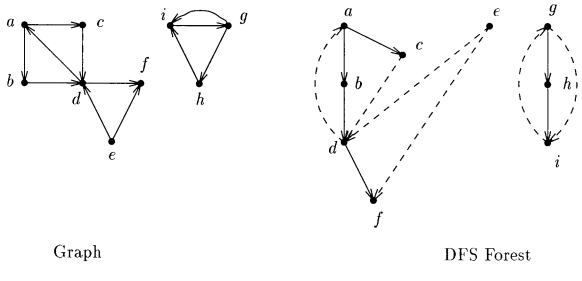


FIGURE 6.3 Sample execution of DFS on a directed graph.

Applications of DFS

Strong Connectivity: Directed DFS is used to design a linear-time algorithm that classifies the edges of a given directed graph into its **strongly-connected** components — maximal subgraphs that have directed paths connecting any pair of vertices in them. The algorithm itself involves running DFS twice, once on the original graph, and then a second time on G^R, which is the graph obtained by reversing the direction of all edges in G. During the second DFS, the algorithm identifies all the strongly connected components. The proof is somewhat subtle, and the reader is referred to [7] for details. Cormen et al. [7] credit Kosaraju and Sharir for this algorithm. The original algorithm due to Tarjan [27] is more complicated.

Directed Acyclic Graphs: Checking if a graph is acyclic can be done in linear time using DFS. A graph has a cycle if and only if there exists a back edge relative to its depth-first search forest. A directed graph that does not have any cycles is known as a **directed acyclic graph** (DAG). DAGs are useful in modeling precedence constraints in scheduling problems, where nodes denote jobs/tasks, and a directed edge from u to v denotes the constraint that job u must be completed before job v can begin execution. Many problems on DAGs can be solved efficiently using dynamic programming (see Chapter 1).

Topological Order: A useful concept in DAGs is that of a *topological order:* a linear ordering of the vertices that is consistent with the partial order defined by its edges. In other words, the vertices can be labeled with distinct integers in the range $[1 \ldots |V|]$ such that if there is a directed edge from a vertex labeled i to a vertex labeled j, then $i < j$. Topological sort has applications in diverse areas such as project management, scheduling and circuit evaluation.

The vertices of a given DAG can be ordered topologically in linear time by a suitable modification of the DFS algorithm. It can be shown that ordering vertices by decreasing finish times (as computed by DFS) is a valid topological order. The DFS algorithm is modified as follows. A counter is initialized to $|V|$. As each vertex is marked finished, the counter value is assigned as its topological number, and the counter is decremented. Since there are no back edges in a DAG, for all edges (u, v), v will be marked finished before u. Thus, the topological number of v will be higher than that of u.

The execution of the algorithm is illustrated with an example in Fig. 6.4. Along with each vertex, we show the discovery and finish times, respectively. Vertices are given decreasing topological numbers as they are marked finished. Vertex f finishes first and gets a topological number of 9 ($|V|$); d finishes next and gets numbered 8, and so on. The topological order found by the DFS is $g, h, i, a, b, e, c, d, f$, which is the reverse of the finishing order. Note that a given graph may have many valid topological ordering of the vertices.

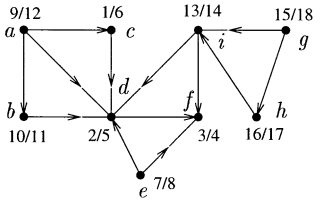


FIGURE 6.4 Example for topological sort. Order in which vertices finish: $f, d, c, e, b, a, i, h, g$.

Other topological ordering algorithms work by identifying and deleting vertices of in-degree zero (i.e., vertices with no incoming edges) recursively. With some care, this algorithm can be implemented in linear time as well.

Longest Path: In project scheduling, a DAG is used to model precedence constraints between tasks. A longest path in this graph is known as a *critical path* and its length is the least time that it takes to complete the project. The problem of computing the longest path in an arbitrary graph is NP-hard. However, longest paths in a DAG can be computed in linear time by using DFS. This method can be generalized to

the case when vertices have weights denoting duration of tasks.

The algorithm processes the vertices in reverse topological order. Let $P(v)$ denote the length of a longest path coming out of vertex v. When vertex v is processed, the algorithm computes the length of a longest path in the graph that starts at v.

$$P(v) = 1 + \max_{(v,w)\in E} P(w) .$$

Since we are processing vertices in reverse topological order, w is processed before v, if (v, w) is an edge, and thus, $P(w)$ is computed before $P(v)$.

6.5 Breadth-First Search

Breadth-first search is another natural way of searching a graph. The search starts at a *root* vertex r. Vertices are added to a queue as they are discovered, and processed in FIFO (first-in first-out) order.

Initially, all vertices are marked as unvisited, and the queue consists of only the root vertex. The algorithm repeatedly removes the vertex at the front of the queue, and scans its neighbors in the graph. Any neighbor that is unvisited is added to the end of the queue. This process is repeated until the queue is empty. All vertices in the same connected component as the root vertex are scanned and the algorithm outputs a spanning tree of this component. This tree, known as a breadth-first tree, is made up of the edges that led to the discovery of new vertices. The algorithm labels each vertex v by $d[v]$, the distance (length of a shortest path) from the root vertex to v, and stores the BFS tree in the array p, using parent-pointers. Vertices can be partitioned into levels based on their distance from the root. Observe that edges not in the BFS tree always go either between vertices in the same level, or between vertices in adjacent levels. This property is often useful.

The BFS Algorithm

```
BFS-DISTANCE (G, r)
1   MAKEEMPTYQUEUE (Q).
2   for all vertices v in G do
3      visited[v] ← false.
4      d[v] ← ∞.
5      p[v] ← nil.
6   end-for
7   visited[r] ← true.
8   d[r] ← 0.
9   ENQUEUE (Q, r).
10  while not EMPTY (Q) do
11     v ← DEQUEUE (Q).
12     for all vertices w in adj[v] do
13        if not visited[w] then
14           visited[w] ← true.
15           p[w] ← v.
16           d[w] ← d[v] + 1.
17           ENQUEUE (w, Q).
18        end-if
19     end-for
20  end-while
end-proc
```

Sample Execution

Figure 6.5 shows a connected graph on which BFS was run with vertex a as the root. When a is processed, vertices b, d, and c are added to the queue. When b is processed nothing is done since all its neighbors have been visited. When d is processed, e and f are added to the queue. Finally c, e, and f are processed.

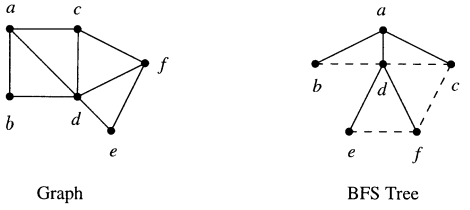


FIGURE 6.5 Sample execution of BFS on a graph.

Analysis

There is no loss of generality in assuming that the graph G is connected, since the algorithm can be repeated in each connected component, similar to the DFS algorithm. The algorithm processes each vertex exactly once, and each edge exactly twice. It spends a constant amount of time in processing each edge. Hence, the algorithm runs in $O(|V| + |E|)$ time.

Bipartite Graphs

A simple algorithm based on BFS can be designed to check if a given graph is bipartite: run BFS on each connected component of the graph, starting from an arbitrary vertex in each component as the root. The algorithm partitions the vertex set into the sets X and Y as follows. For a vertex v, if $d[v]$ is odd, then it inserts v into X. Otherwise $d[v]$ is even and it inserts v into Y. Now check to see if there is an edge in the graph that connects two vertices in the same set (X or Y). If the graph contains an edge between two vertices of the same set, say X, then we conclude that the graph is not bipartite, since the graph contains an odd-length cycle; otherwise the algorithm has partitioned the vertex set into X and Y and all edges of the graph connect a vertex in X with a vertex in Y, and therefore by definition, the graph is bipartite. (Note that it is known that a graph is bipartite if and only if it does not have a cycle of odd length.)

6.6 Single-Source Shortest Paths

A natural problem that often arises in practice is to compute the shortest paths from a specified node r to all other nodes in a graph. BFS solves this problem if all edges in the graph have the same length. Consider the more general case when each edge is given an arbitrary, nonnegative length. In this case, the length of a path is defined to be the sum of the lengths of its edges. The *distance* between two nodes is the length of a shortest path between them. The objective of the shortest path problem is to compute the distance from r to each vertex v in the graph, and a path of that length from r to v. The output is a tree, known as the *shortest path tree,* rooted at r. For any vertex v in the graph, the unique path from r to v in this tree is a shortest path from r to v in the input graph.

Dijkstra's Algorithm

Dijkstra's algorithm provides an efficient solution to the shortest path problem. For each vertex v, the algorithm maintains an upper bound of the distance from the root to vertex v in $d[v]$; initially $d[v]$ is set to infinity for all vertices except the root, which has d-value equal to zero. The algorithm maintains a set S of vertices with the property that for each vertex $v \in S$, $d[v]$ is the length of a shortest path from the root to v. For each vertex u in $V - S$, the algorithm maintains $d[u]$ to be the length of a shortest path from the root to u that goes entirely within S, except for the last edge. It selects a vertex u in $V - S$ with minimum $d[u]$ and adds it to S, and updates the distance estimates to the other vertices in $V - S$. In this update step it checks to see if there is a shorter path to any vertex in $V - S$ from the root that goes through u. Only the distance estimates of vertices that are adjacent to u need to be updated in this step. Since the primary operation is the selection of a vertex with minimum distance estimate, a priority queue is used to maintain the d-values of vertices (for more information about priority queues, see Chapter 4). The priority queue should be able to handle the DECREASEKEY operation to update the d-value in each iteration. The algorithm below implements Dijkstra's algorithm.

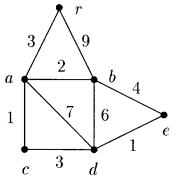
```
DIJKSTRA-SHORTEST PATHS (G, r)
1   for all vertices v in G do
2      visited[v] ← false.
3      d[v] ← ∞.
4      p[v] ← nil.
5   end-for
6   d[r] ← 0.
7   BUILDPQ (H, d).
8   while not EMPTY (H) do
9         u ← DELETEMIN (H).
10        visited[u] ← true.
11        for all vertices v in adj[u] do
12           RELAX (u, v).
13        end-for
14  end-while
end-proc
```

```
RELAX (u, v)
1   if not visited[v] and d[v] > d[u] + w(u, v) then
2      d[v] ← d[u] + w(u, v).
3      p[v] ← u.
4      DECREASEKEY (H, v, d[v]).
5   end-if
end-proc
```

Sample Execution

Figure 6.6 shows a sample execution of the algorithm. The column titled "Iter" specifies the number of iterations that the algorithm has executed through the while loop in Step 8. In iteration 0 the initial values of the distance estimates are ∞. In each subsequent line of the table, the column marked u shows the vertex that was chosen in Step 9 of the algorithm, and the other columns show the change to the distance estimates at the end of that iteration of the while loop. In the first iteration, vertex r was chosen, after

that a was chosen since it had the minimum distance label among the unvisited vertices, and so on. The distance labels of the unvisited neighbors of the visited vertex are updated in each iteration.



Iter	u	$d[a]$	$d[b]$	$d[c]$	$d[d]$	$d[e]$
0	-	∞	∞	∞	∞	∞
1	r	3	9	∞	∞	∞
2	a	3	5	4	10	∞
3	c	3	5	4	7	∞
4	b	3	5	4	7	9
5	d	3	5	4	7	8
6	e	3	5	4	7	8

FIGURE 6.6 Dijkstra's shortest path algorithm.

Analysis

The running time of the algorithm depends on the data structure that is used to implement the priority queue H. The algorithm performs $|V|$ DELETEMIN operations and at most $|E|$ DECREASEKEY operations. If a binary heap is used to find the records of any given vertex, each of these operations run in $O(\log |V|)$ time. There is no loss of generality in assuming that the graph is connected. Hence, the algorithm runs in $O(|E| \log |V|)$. If a Fibonacci heap [10] is used to implement the priority queue, the running time of the algorithm is $O(|E| + |V| \log |V|)$. Even though the Fibonacci heap gives the best asymptotic running time, the binary heap implementation is likely to give better running times for most practical instances.

Extensions

Dijkstra's algorithm can be generalized to solve several problems that are related to the shortest path problem. For example, in the bottleneck shortest path problem, the objective is to find, for each vertex v, a path from the root to v in which the length of the longest edge in that path is minimized. A small change to Dijkstra's algorithm (replacing the operation $+$ in RELAX by max) solves this problem. Other problems that can be solved by suitably modifying Dijkstra's algorithm include the following:

- Finding most reliable paths from the root to every vertex in a graph where each edge is given a probability of failure (independent of the other edges).
- Finding the fastest way to get from a given point in a city to a specified location using public transportation, given the train/bus schedules.

Bellman–Ford Algorithm

The shortest path algorithm described above directly generalizes to directed graphs, but it does not work if the graph has edges of negative length. For graphs that have edges of negative length, but no cycles of negative length, there is a different algorithm solves due to Bellman and Ford that solves the single source shortest paths problem in $O(|V||E|)$ time.

In a single scan of the edges, the RELAX operation is executed on each edge. The scan is then repeated $|V| - 1$ times. No special data structures are required to implement this algorithm, and the proof relies on the fact that a shortest path is simple and contains at most $|V| - 1$ edges.

This problem also finds applications in finding a feasible solution to a system of linear equations of a special form that arises in real-time applications: each equation specifies a bound on the difference between two variables. Each constraint is modeled by an edge in a suitably defined directed graph. Shortest paths

from the root of this graph capture feasible solutions to the system of equations (for more information, see [7, Chapter 25.5]).

The All-Pairs Shortest Paths Problem

Consider the problem of computing a shortest path between every pair of vertices in a directed graph with edge lengths. The problem can be solved in $O(|V|^3)$ time, even when some edges have negative lengths, as long as the graph has no negative length cycles. Let the lengths of the edges be stored in a matrix A; the array entry $A[i, j]$ stores the length of the edge from i to j. If there is no edge from i to j, then $A[i, j] = \infty$; also $A[i, i]$ is set to 0 for all i. A dynamic programming algorithm to solve the problem is discussed in this section. The algorithm is due to Floyd and builds on the work of Warshall.

Define $P_k[u, v]$ to be a shortest path from u to v that is restricted to using intermediate vertices only from the set $\{1, \ldots, k\}$. Let $D_k[u, v]$ be the length of $P_k[u, v]$. Note that $P_0[u, v] = (u, v)$ since the path is not allowed to use *any* intermediate vertices, and therefore $D_0[u, v] = A[u, v]$. Since there are no negative length cycles, there is no loss of generality in assuming that shortest paths are simple.

The structure of shortest paths leads to the following recursive formulation of P_k. Consider $P_k[i, j]$ for $k > 0$. Either vertex k is on this path or not. If $P_k[i, j]$ does not pass through k, then the path uses only vertices from the set $\{1, \ldots, k-1\}$ as intermediate vertices, and is therefore the same as $P_{k-1}[i, j]$. If k is a vertex on the path $P_k[i, j]$, then it passes through k exactly once because the path is simple. Moreover, the subpath from i to k in $P_k[i, j]$ is a shortest path from i to k that uses intermediate vertices from the set $\{1, \ldots, k-1\}$, as does the subpath from k to j in $P_k[i, j]$. Thus, the path $P_k[i, j]$ is the union of $P_{k-1}[i, k]$ and $P_{k-1}[k, j]$. The above discussion leads to the following recursive formulation of D_k:

$$D_k[i, j] = \begin{cases} \min\left(D_{k-1}[i, j], D_{k-1}[i, k] + D_{k-1}[k, j]\right) & \text{if } k > 0 \\ A[i, j] & \text{if } k = 0 \end{cases}$$

Finally, since $P_n[i, j]$ is allowed to go through any vertex in the graph, $D_n[i, j]$ is the length of a shortest path from i to j in the graph.

In the algorithm described below, a matrix D is used to store distances. It might appear at first glance that to compute the distance matrix D_k from D_{k-1}, different arrays must be used for them. However, it can be shown that in the kth iteration, the entries in the kth row and column do not change, and thus, the same space can be reused.

```
FLOYD-SHORTEST-PATH (G)
1   for i = 1 to |V| do
2      for j = 1 to |V| do
3         D[i, j] ← A[i, j]
4      end-for
5   end-for
6   for k = 1 to |V| do
7      for i = 1 to |V| do
8         for j = 1 to |V| do
9            D[i, j] ← min(D[i, j], D[i, k] + D[k, j]).
10        end-for
11     end-for
12  end for
end-proc
```

6.7 Minimum Spanning Trees

The following fundamental problem arises in network design. A set of sites need to be connected by a network. This problem has a natural formulation in graph-theoretic terms. Each site is represented by a vertex. Edges between vertices represent a potential link connecting the corresponding nodes. Each edge is given a non-negative cost corresponding to the cost of constructing that link. A tree is a minimal network that connects a set of nodes. The cost of a tree is the sum of the costs of its edges. A minimum-cost tree connecting the nodes of a given graph is called a minimum-cost spanning tree, or simply a **minimum spanning tree.**

The problem of computing a minimum spanning tree (MST) arises in many areas, and as a subproblem in combinatorial and geometric problems. MSTs can be computed efficiently using algorithms that are greedy in nature, and there are several different algorithms for finding an MST. One of the first algorithms was due to Boruvka. Two algorithms, popularly known as Prim's algorithm and Kruskal's algorithm, are described here.

We first describe some rules that characterize edges belonging to a minimum spanning tree. The various algorithms are based on applying these rules in different orders. Tarjan [28] uses colors to describe these rules. Initially, all edges are uncolored. When an edge is colored *blue* it is marked for inclusion in the MST. When an edge is colored *red* it is marked to be excluded from the MST. The algorithms maintain the property that there is an MST containing all the blue edges but none of the red edges.

A *cut* is a partitioning of the vertex set into two subsets S and $V - S$. An edge *crosses* the cut if it connects a vertex $x \in S$ to a vertex $y \in V - S$.

(**Blue rule**) Find a cut that is not crossed by any blue edge and color a minimum weight edge that crosses the cut to be blue.

(**Red rule**) Find a simple cycle containing no red edges and color a maximum weight edge on that cycle to be red.

The proofs that these rules work can be found in [28].

Prim's Algorithm

Prim's algorithm for finding an MST of a given graph is one of the oldest algorithms to solve the problem. The basic idea is to start from a single vertex and gradually "grow" a tree, which eventually spans the entire graph. At each step, the algorithm has a tree of blue edges that covers a set S of vertices. The blue rule is applied by picking the cut $S, V - S$. This may be used to extend the tree to include a vertex that is currently not in the tree. The algorithm selects a minimum-cost edge from the edges crossing the cut and adds it to the current tree (implicitly coloring the edge blue), thereby adding another vertex to S.

As in the case of Dijkstra's algorithm, each vertex $u \in V - S$ can attach itself to only one vertex in the tree so that the current solution maintained by the algorithm is always a tree. Since the algorithm always chooses a minimum-cost edge, it needs to maintain a minimum-cost edge that connects u to some vertex in S as the candidate edge for including u in the tree. A priority queue of vertices is used to select a vertex in $V - S$ that is incident to a minimum-cost candidate edge.

```
PRIM-MST (G, r)
1  for all vertices v in G do
2     visited[v] ← false.
3     d[v] ← ∞.
4     p[v] ← nil.
5  end-for
6  d[r] ← 0.
7  BUILDPQ (H, d).
```

```
8   while not Empty(H) do
9      u ← DeleteMin (H).
10     visited[u] ← true.
11     for all vertices v in adj[u] do
12        if not visited[v] and d[v] > w(u, v) then
13           d[v] ← w(u, v).
14           p[v] ← u.
15           DecreaseKey (H, v, d[v]).
16        end-if
17     end-for
18  end-while
end-proc
```

Analysis

First observe the similarity between Prim's and Dijkstra's algorithms. Both algorithms start building the tree from a single vertex and grow it by adding one vertex at a time. The only difference is the rule for deciding when the current label is updated for vertices outside the tree. Both algorithms have the same structure and therefore have similar running times. Prim's algorithm runs in $O(|E| \log |V|)$ time if the priority queue is implemented using binary heaps, and it runs in $O(|E| + |V| \log |V|)$ if the priority queue is implemented using Fibonacci heaps.

Kruskal's Algorithm

Kruskal's algorithm for finding an MST of a given graph is another classical algorithm for the problem, and is also greedy in nature. Unlike Prim's algorithm which grows a single tree, Kruskal's algorithm grows a forest. First the edges of the graph are sorted in nondecreasing order of their costs. The algorithm starts with an empty forest. The edges of the graph are scanned in sorted order, and if the addition of the current edge does not generate a cycle in the current forest, it is added to the forest. The main test at each step is: does the current edge connect two vertices in the same connected component of the current forest? Eventually the algorithm adds $n - 1$ edges to generate a spanning tree in the graph.

The following discussion explains the correctness of the algorithm based on the two rules described earlier. Suppose that as the algorithm progresses, the edges chosen by the algorithm are colored blue and the ones that it rejects are colored red. When an edge is considered and it forms a cycle with previously chosen edges, this is a cycle with no red edges. Since the algorithm considers the edges in nondecreasing order of weight, the last edge is the heaviest edge in the cycle and therefore it can be colored red by the red rule. If an edge connects two blue trees T_1 and T_2, then it is a lightest edge crossing the cut T_1 and $V - T_1$, because any other edge crossing the cut has not been considered yet and is therefore no lighter. Therefore it can be colored blue by the blue rule.

The main data structure needed to implement the algorithm is to maintain connected components. An abstract version of this problem is known as the union–find problem for collection of disjoint sets (Chapter 8). Efficient algorithms are known for this problem, where an arbitrary sequence of Union and Find operations can be implemented to run in almost linear time (for more information, see [7, 28]).

```
Kruskal-MST(G)
1   T ← φ.
2   for all vertices v in G do
3      p[v] ← v.
```

```
4   end-for
5   Sort the edges of G by nondecreasing order of costs.
6   for all edges e = (u, v) in G in sorted order do
7      if FIND (u) ≠ FIND (v) then
8         T ← T ∪ (u, v).
9         UNION (u, v).
10     end-if
11  end-for
end-proc
```

Analysis

The running time of the algorithm is dominated by Step 5 of the algorithm in which the edges of the graph are sorted by nondecreasing order of their costs. This takes $O(|E| \log |E|)$ (which is also $O(|E| \log |V|)$) time using an efficient sorting algorithm such as heap sort. Kruskal's algorithm runs faster in the following special cases: if the edges are presorted, if the edge costs are within a small range, or if the number of different edge costs is bounded. In all these cases, the edges can be sorted in linear time, and Kruskal's algorithm runs in the near-linear time of $O(|E|\alpha(|E|, |V|))$, where $\alpha(m, n)$ is the inverse Ackermann function [28].

Boruvka's Algorithm

Boruvka's algorithm also grows many trees simultaneously. Initially there are $|V|$ trees, where each vertex forms its own tree. At each stage the algorithm keeps a collection of blue trees (i.e., trees built using only blue edges). For convenience, assume that all edge weights are distinct. If two edges have the same weight, they may be ordered arbitrarily. Each tree selects a minimum cost edge that connects it to some other tree and colors it blue. At the end of this parallel coloring step, each tree merges with a collection of other trees. The number of trees decreases by at least a factor of 2 in each step, and therefore after $\log |V|$ iterations there is exactly one tree. In practice, many trees merge in a single step and the algorithm converges much faster. Each step can be implemented in $O(|E|)$ time, and hence, the algorithm runs in $O(|E| \log |V|)$. For the special case of planar graphs, the above algorithm actually runs in $O(|V|)$ time.

Almost linear-time deterministic algorithms for the MST problem in undirected graphs are known [5, 10]. Recently, Karger, Klein and Tarjan [18] showed that they can combine the approach of Boruvka's algorithm with a random sampling approach to obtain a randomized algorithm with an expected running time of $O(|E|)$. Their algorithm also needs to use as a subroutine a procedure to verify that a proposed tree is indeed an MST [20, 21]. The equivalent of minimum spanning trees in directed graphs are known as minimum **branchings** and are discussed in Chapter 7.

6.8 Tour and Traversal Problems

There are many applications for finding certain kinds of paths and tours in graphs. We briefly discuss some of the basic problems.

The **traveling salesman problem** (TSP) is that of finding a shortest tour that visits all the vertices of a given graph with weights on the edges. It has received considerable attention in the literature [22]. The problem is known to be computationally intractable (NP-hard). Several heuristics are known to solve practical instances. Considerable progress has also been made in finding optimal solutions for graphs with a few thousand vertices.

One of the first graph-theoretic problems to be studied, the **Euler tour problem** asks for the existence of a closed walk in a given connected graph that traverses each edge exactly once. Euler proved that such

a closed walk exists if and only if each vertex has even degree [12]. Such a graph is known as an **Eulerian graph.** Given an Eulerian graph, an Euler tour in it can be computed using an algorithm similar to DFS in linear time.

Given an edge-weighted graph, the **Chinese postman problem** is that of finding a shortest closed walk that traverses each edge at least once. Although the problem sounds very similar to the TSP problem, it can be solved optimally in polynomial time [1].

6.9 Assorted Topics

Planar Graphs

A graph is called *planar* if it can be drawn on the plane without any of its edges crossing each other. A *planar embedding* is a drawing of a planar graph on the plane with no crossing edges. An embedded planar graph is known as a *plane graph.* A *face* of a plane graph is a connected region of the plane surrounded by edges of the planar graph. The unbounded face is referred to as the *exterior face.* Euler's formula captures a fundamental property of planar graphs by relating the number of edges, the number of vertices and the number of faces of a plane graph: $|F| - |E| + |V| = 2$. One of the consequences of this formula is that a simple planar graph has at most $O(|V|)$ edges.

Extensive work has been done on the study of planar graphs and a recent book has been devoted to the subject [26]. A fundamental problem in this area is deciding whether a given graph is planar, and if so, finding a planar embedding for it. Kuratowski gave necessary and sufficient conditions for when a graph is planar, by showing that a graph is planar if and only if it has no subgraph that is a homeomorph of K_5 or $K_{3,3}$. Hopcroft and Tarjan [17] gave a linear time algorithm to test if a graph is planar, and if it is, to find a planar embedding for the graph.

A *balanced separator* is a subset of vertices that disconnects the graph in such a way, that the resulting components each have at most a constant fraction of the number of vertices of the original graph. Balanced separators are useful in designing "divide-and-conquer" algorithms for graph problems, such as graph layout problems (Chapter 23). Such algorithms are possible when one is guaranteed to find separators that have very few vertices relative to the graph. Lipton and Tarjan [24] proved that every planar graph on $|V|$ vertices has a separator of size at most $\sqrt{8|V|}$, whose deletion breaks the graph into two or more disconnected graphs, each of which has at most $\frac{2}{3}|V|$ vertices. Using the property that planar graphs have small separators, Frederickson [9] has given faster shortest path algorithms for planar graphs. Recently, this was improved to a linear time algorithm by Henzinger et al. [13].

Graph Coloring

A coloring of a graph is an assignment of colors to the vertices, so that any two adjacent vertices have distinct colors. Traditionally, the colors are not given names, but represented by positive integers. The *vertex coloring* problem is the following: given a graph, to color its vertices using the fewest number of colors (known as the *chromatic number* of the graph). This was one of the first problems that were shown to be intractable (NP-hard). Recently it has been shown that even the problem of approximating the chromatic number of the graph within any reasonable factor is intractable. But, the coloring problem needs to be solved in practice (such as in the channel assignment problem in cellular networks), and heuristics are used to generate solutions. We discuss a commonly used greedy heuristic below: the vertices of the graph are colored sequentially in an arbitrary order. When a vertex is being processed, the color assigned to it is the smallest positive number that is not used by any of its neighbors that have been processed earlier. This scheme guarantees that if the degree of a vertex is Δ, then its color is at most $\Delta + 1$. There are special classes of graphs, such as planar graphs, in which the vertices can be carefully ordered in such a way that the number of colors used is small. For example, the vertices of a planar graph can be ordered such that

every vertex has at most 5 neighbors that appear earlier in the list. By coloring its vertices in that order yields a 6-coloring. There is a different algorithm that colors any planar graph using only 4 colors.

Light Approximate Shortest Path Trees

To broadcast information from a specified vertex r to all vertices of G, one may wish to send the information along a shortest path tree in order to reduce the time taken by the message to reach the nodes (i.e., minimizing *delay*). Though the shortest path tree may minimize delays, it may be a much costlier network to construct and considerably heavier than a minimum spanning tree, which leads to the question of whether there are trees that are light (like an MST) and yet capture distances like a shortest path tree. In this section, we consider the problem of computing a light subgraph that approximates a shortest path tree rooted at r.

Let T_{min} be a minimum spanning tree of G. For any vertex v, let $d(r, v)$ be the length of a shortest path from r to v in G. Let $\alpha > 1$ and $\beta > 1$ be arbitrary constants. An (α, β)-*light approximate shortest path tree* $\Big((\alpha, \beta)\text{-LAST}\Big)$ of G is a spanning tree T of G with the property that the distance from the root to any vertex v in T is at most $\alpha \cdot d(r, v)$ and the weight of T is at most β times the weight of T_{min}.

Awerbuch, Baratz and Peleg [3], motivated by applications in broadcast-network design, made a fundamental contribution by showing that every graph has a *shallow-light* tree — a tree whose *diameter* is at most a constant times the diameter of G and whose total weight is at most a constant times the weight of a minimum spanning tree. Cong et al. [6] studied the same problem and showed that the problem has applications in VLSI-circuit design; they improved the approximation ratios obtained in [3] and also studied variations of the problem such as bounding the *radius* of the tree instead of the diameter.

Khuller, Raghavachari and Young [19] modified the shallow-light tree algorithm and showed that the distance from the root to each vertex can be approximated within a constant factor. Their algorithm also runs in linear time if a minimum spanning tree and a shortest path tree are provided. The algorithm computes an $(\alpha, 1 + \frac{2}{\alpha - 1})$-LAST.

The basic idea is as follows: initialize a subgraph H to be a minimum spanning tree T_{min}. The vertices are processed in a preorder traversal of T_{min}. When a vertex v is processed, its distance from r in H is compared to $\alpha \cdot d(r, v)$. If the distance exceeds the required threshold, then the algorithm adds to H a shortest path in G from r to v. When all the vertices have been processed, the distance in H from r to any vertex v meets its distance requirement. A shortest path tree in H is returned by the algorithm as the required LAST.

Network Decomposition

The problem of decomposing a graph into clusters, each of which has low diameter, has applications in distributed computing. Awerbuch [2] introduced an elegant algorithm for computing low diameter clusters, with the property that there are few inter-cluster edges (assuming that edges going between clusters are not counted multiply). This construction was further refined by Awerbuch and Peleg [4], and they showed that a graph can be decomposed into clusters of diameter $O(r \log |V|)$ with the property that each r neighborhood of a vertex belongs to some cluster. (An r neighborhood of a vertex is the set of nodes whose distance from the vertex is at most r.) In addition, each vertex belongs to at most $2 \log |V|$ clusters. Using a similar approach Linial and Saks [23] showed that a graph can be decomposed into $O(\log |V|)$ clusters, with the property that each connected component in a cluster has $O(\log |V|)$ diameter. These techniques have found several applications in the computation of approximate shortest paths, and in other distributed computing problems.

The basic idea behind these methods is to perform an "expanding BFS." The algorithm selects an arbitrary vertex, and executes BFS with that vertex as the root. The algorithm continues the search layer by layer, ensuring that the number of vertices in a layer is at least as large as the number of vertices currently

in that BFS tree. Since the tree expands rapidly, this procedure generates a low diameter BFS tree (cluster). If the algorithm comes across a layer in which the number of nodes is not big enough, it *rejects* that layer and stops growing that tree. The set of nodes in the layer that was not added to the BFS tree that was being grown is guaranteed to be small. The algorithm continues by selecting a new vertex that was not chosen in any cluster and repeats the above procedure.

6.10 Research Issues and Summary

We have illustrated some of the fundamental techniques that are useful for manipulating graphs. These basic algorithms are used as tools in the design of algorithms for graphs. The problems studied in this chapter included representation of graphs, tree traversal techniques, search techniques for graphs, shortest path problems, minimum spanning trees, and tour problems on graphs.

Current research on graph algorithms focuses on dynamic algorithms, graph layout and drawing, and approximation algorithms. More information about these areas can be found in Chapter 8, Chapter 9, and Chapter 34. The methods illustrated in our chapter find use in the solution of almost any graph problem.

The *graph isomorphism* problem is an old problem in this area. The input to this problem is two graphs and the problem is to decide whether the two graphs are isomorphic, i.e., whether the rows and columns of the adjacency matrix of one of the graphs can be permuted so that it is identical to the adjacency matrix of the other graph. This problem is neither known to be polynomial-time solvable nor known to be NP-hard. This is in contrast to the *subgraph isomorphism* problem in which the problem is to decide whether there is a subgraph of the first graph that is isomorphic to the second graph. The subgraph isomorphism is known to be NP-complete. Special instances of the **graph isomorphism problem** are known to be polynomially solvable, such as when the graphs are planar, or more generally of bounded genus. For more information on the isomorphism problem, see Hoffman [14].

Another open problem is whether there exists a *deterministic* linear time algorithm for computing a minimum spanning tree. Near-linear time deterministic algorithms using Fibonacci heaps have been known for finding an MST. The newly discovered probabilistic algorithm uses random sampling to find an MST in expected linear time. Much of the recent research in this area is focusing on the design of approximation algorithms for NP-hard problems.

6.11 Defining Terms

Articulation vertex/Cut vertex: A vertex whose deletion disconnects a graph into two or more connected components.

Biconnected graph: A graph that has no articulation/cut vertices.

Bipartite graph: A graph in which the vertex set can be partitioned into two sets X and Y, such that each edge connects a node in X with a node in Y.

Branching: A rooted spanning tree in a directed graph, such that the root has a path in the tree to each vertex.

Chinese postman problem: Find a minimum length tour that traverses each edge at least once.

Connected graph: A graph in which there is a path between each pair of vertices.

Cycle: A path in which the start and end vertices of the path are identical.

Degree: The number of edges incident to a vertex in a graph.

DFS forest: A rooted forest formed by depth-first search.

Directed acyclic graph: A directed graph with no cycles.

Euler tour problem: Asks for a traversal of the edges that visits each edge exactly once.

Eulerian graph: A graph that has an Euler tour.

Forest: An acyclic graph.

Graph isomorphism problem: Deciding if two given graphs are isomorphic to each other.

Leaves: Vertices of degree one in a tree.

Minimum spanning tree: A spanning tree of minimum total weight.

Path: An ordered list of distinct edges, $\{e_i = (u_i, v_i) | i = 1, \ldots, k\}$, such that for any two consecutive edges e_i and e_{i+1}, $v_i = u_{i+1}$.

Planar graph: A graph that can be drawn on the plane without any of its edges crossing each other.

Sparse graph: A graph in which $|E| \ll |V|^2$.

Strongly connected graph: A directed graph in which there is a directed path between each ordered pair of vertices.

Topological order: A numbering of the vertices of a DAG such that every edge in the graph that goes from a vertex numbered i to a vertex numbered j satisfies $i < j$.

Traveling salesman problem: Asks for a minimum length tour of a graph that visits all the vertices exactly once.

Tree: A connected forest.

Walk: A path in which edges may be repeated.

Acknowledgments

Samir Khuller's research was supported by NSF Research Initiation Award CCR-9307462 and NSF CAREER Award CCR-9501355. Balaji Raghavachari's research was supported by NSF Research Initiation Award CCR-9409625.

References

[1] Ahuja, R.K, Magnanti, T.L., and Orlin, J.B., *Network Flows.* Prentice Hall, 1993.

[2] Awerbuch, B., Complexity of network synchronization. *J. Assoc. Comput. Mach.,* 32(4), 804–823, 1985.

[3] Awerbuch, B., Baratz, A., and Peleg, D., Cost-sensitive analysis of communication protocols. In *Proceedings of the Ninth Annual ACM Symposium on Principles of Distributed Computing,* 177–187, Quebec City, Quebec, Canada, 22–24, Aug. 1990.

[4] Awerbuch, B. and Peleg, D., Sparse partitions. In *Proceedings of the 31st Annual Symposium on Foundations of Computer Science,* 503–513, St. Louis, MO, 22–24, Oct 1990.

[5] Chazelle, B., A faster deterministic algorithm for minimum spanning trees. In *Proceedings of the 38th Annual Symposium on Foundations of Computer Science,* 22–31, Miami, FL, 20–22, Oct 1997.

[6] Cong, J., Kahng, A.B., Robins, G., Sarrafzadeh, M., and Wong, C.K., Provably good performance-driven global routing. *IEEE Transactions on CAD,* 739–752, 1992.

[7] Cormen, T.H., Leiserson, C.E., and Rivest, R.L., *Introduction to Algorithms.* The MIT Press, Cambridge, MA, 1989.

[8] Even, S., *Graph Algorithms.* Computer Science Press, Rockville, MD, 1979.

[9] Frederickson, G.N., Fast algorithms for shortest paths in planar graphs with applications. *SIAM J. Comput.,* 16(6), 1004–1022, 1987.

[10] Fredman, M.L. and Tarjan, R.E., Fibonacci heaps and their uses in improved network optimization algorithms. *J. Assoc. Comput. Mach.,* 34(3), 596–615, 1987.

[11] Galil, Z. and Italiano, G., Reducing edge connectivity to vertex connectivity. *SIGACT News,* 22(1), 57–61, 1991.

[12] Gibbons, A.M., *Algorithmic Graph Theory.* Cambridge University Press, New York, 1985.
[13] Henzinger, M.R., Klein, P.N., Rao, S., and Subramanian, S., Faster shortest-path algorithms for planar graphs. *J. Computer Syst. Sci.,* 55(1), 3–23, 1997.
[14] Hoffman, C.M., *Group-Theoretic Algorithms and Graph Isomorphism.* Lecture notes in Computer Science #136, Springer-Verlag, Berlin, 1982.
[15] Hopcroft, J.E. and Tarjan, R.E., Dividing a graph into triconnected components. *SIAM J. Comput.,* 2(3), 135–158, 1973.
[16] Hopcroft, J.E. and Tarjan, R.E., Efficient algorithms for graph manipulation. *Communications of the ACM,* 16, 372–378, 1973.
[17] Hopcroft, J.E. and Tarjan, R.E., Efficient planarity testing. *J. Assoc. Comput. Mach.,* 21(4), 549–568, 1974.
[18] Karger, D.R., Klein, P.N., and Tarjan, R.E., A randomized linear-time algorithm to find minimum spanning trees. *J. Assoc. Comput. Mach.,* 42(2), 321–328, 1995.
[19] Khuller, S., Raghavachari, B., and Young, N., Balancing minimum spanning trees and shortest-path trees. *Algorithmica,* 14(4), 305–321, 1995.
[20] King, V., A simpler minimum spanning tree verification algorithm. *Algorithmica,* 18(2), 263–270, 1997.
[21] Komlós, J., Linear verification for spanning trees. *Combinatorica,* 5, 57–65, 1985.
[22] Lawler, E.l., Lenstra, J.K., Rinnooy Kan, A.H.G., and Shmoys, D.B., *The Traveling Salesman Problem: A Guided Tour of Combinatorial Optimization.* Wiley, New York, 1985.
[23] Linial, M. and Saks, M., Low diameter graph decompositions. *Combinatorica,* 13(4), 441–454, 1993.
[24] Lipton, R. and Tarjan, R.E., A separator theorem for planar graphs. *SIAM J. Appl. Math.,* 36, 177–189, 1979.
[25] Nagamochi, H. and Ibaraki, T., Linear time algorithms for finding sparse k-connected spanning subgraph of a k-connected graph. *Algorithmica,* 7(5/6), 583–596, 1992.
[26] Nishizeki, T. and Chiba, N., *Planar Graphs: Theory and Algorithms.* North-Holland, Amsterdam, 1989.
[27] Tarjan, R.E., Depth-first search and linear graph algorithms. *SIAM J. Comput.,* 1(2), 146–160, Jun 1972.
[28] Tarjan, R.E., *Data Structures and Network Algorithms.* Society for Industrial and Applied Mathematics, Philadelphia, PA, 1983.

Further Information

The area of graph algorithms continues to be a very active field of research. There are several journals and conferences that discuss advances in the field. Here we name a partial list of some of the important meetings: "ACM Symposium on Theory of Computing (STOC)," "IEEE Conference on Foundations of Computer Science (FOCS)," "ACM-SIAM Symposium on Discrete Algorithms (SODA)," "International Colloquium on Automata, Languages and Programming (ICALP)," and "European Symposium on Algorithms (ESA)." There are many other regional algorithms/theory conferences that carry research papers on graph algorithms. The journals that carry articles on current research in graph algorithms are *Journal of the ACM, SIAM Journal on Computing, SIAM Journal on Discrete Mathematics, Journal of Algorithms, Algorithmica, Journal of Computer and System Sciences, Information and Computation, Information Processing Letters,* and *Theoretical Computer Science.* To find more details about some of the graph algorithms described in this chapter we refer the reader to the books by Cormen, Leiserson and Rivest [7], Even [8], Gibbons [12], and Tarjan [28].

7

Advanced Combinatorial Algorithms

Samir Khuller
University of Maryland

Balaji Raghavachari
University of Texas at Dallas

7.1 Introduction

The optimization of a given objective, while working with limited resources is a fundamental problem that occurs in all walks of life, and is especially important in computer science and operations research. Problems in discrete optimization vary widely in their complexity, and efficient solutions are derived for many of these problems by studying their combinatorial structure and understanding their fundamental properties. In this chapter, we study several problems and advanced algorithmic techniques for solving them. One of the basic topics in this field is the study of **network flow** and related optimization problems; these problems occur in various disciplines, and provide a fundamental framework for solving problems. For example, the problem of efficiently moving entities, such as bits, people, or products, from one place

to another in an underlying network, can be modeled as a network flow problem. Network flow finds applications in many other areas such as **matching,** scheduling, and connectivity problems in networks. The problem plays a central role in the fields of operations research and computer science, and considerable emphasis has been placed on the design of efficient algorithms for solving it.

The network flow problem is usually formulated on directed graphs (which are also known as *networks*). A fundamental problem is the **maximum flow** problem, usually referred to as the *max-flow* problem. The input to this problem is a directed graph $G = (V, E)$, a nonnegative **capacity** function $u : E \mapsto \Re^+$ that specifies the *capacity* of each arc, a source vertex $s \in V$, and a sink vertex $t \in V$. The problem captures the situation when a commodity is being produced at node s, and needs to be shipped to node t, through the network. The objective is to send as many units of flow as possible from s to t, while satisfying flow conservation constraints at all intermediate nodes and capacity constraints on the edges. The problem will be defined formally later.

Many practical combinatorial problems such as the **assignment problem,** and the problem of finding the susceptibility of networks to failures due to faulty links or nodes, are special instances of the max-flow problem. There are several variations and generalizations of the max-flow problem including the vertex capacitated max-flow problem, the minimum-cost max-flow problem, the minimum-cost circulation problem, and the multicommodity flow problem.

Section 7.2 discusses the matching problem. The single commodity maximum flow problem is introduced in Section 7.3. The minimum-cut problem is discussed in Section 7.4. Section 7.5 discusses the min-cost flow problem. The multicommodity flow problem is discussed in Section 7.6. Section 7.7 introduces the problem of computing optimal **branchings.** Section 7.8 discusses the problem of coloring the edges and vertices of a graph. Section 7.9 discusses approximation algorithms for NP-hard problems. At the end, references to current research in graph algorithms are provided.

7.2 The Matching Problem

An entire book [23] has been devoted to the study of various aspects of the matching problem, ranging from necessary and sufficient conditions for the existence of perfect matchings to algorithms for solving the matching problem. Many of the basic algorithms studied in Chapter 6 play an important role in developing various implementations for network flow and matching algorithms.

First the *matching* problem, which is a special case of the max-flow problem is introduced. Then the *assignment problem,* a generalization of the matching problem, is studied.

The maximum matching problem is discussed in detail only for bipartite graphs. The same principles are used to design efficient algorithms to solve the matching problem in arbitrary graphs. The algorithms for general graphs are complex due to the presence of odd-length cycles called **blossoms,** and the reader is referred to [26, Chapter 10], or [29, Chapter 9] for a detailed treatment of how blossoms are handled.

Matching Problem Definitions

Given a graph $G = (V, E)$, a *matching* M is a subset of the edges such that no two edges in M share a common vertex. In other words, the problem is that of finding a set of independent edges, that have no incident vertices in common. The cardinality of M is usually referred to as its *size.*

The following terms are defined with respect to a matching M. The edges in M are called *matched edges* and edges not in M are called *free edges.* Likewise, a vertex is a *matched vertex* if it is incident to a matched edge. A *free vertex* is one that is not matched. The *mate* of a matched vertex v is its neighbor w that is at the other end of the matched edge incident to v. A matching is called *perfect* if all vertices of the graph are matched in it. (When the number of vertices is odd, we permit one vertex to remain unmatched.) The objective of the maximum matching problem is to maximize $|M|$, the size of the matching. If the edges of the graph have weights, then the *weight* of a matching is defined to be the sum of the weights of the

edges in the matching. A path $p = [v_1, v_2, \ldots, v_k]$ is called an *alternating path* if the edges (v_{2j-1}, v_{2j}), $j = 1, 2, \ldots$ are free, and the edges (v_{2j}, v_{2j+1}), $j = 1, 2, \ldots$ are matched. An **augmenting path** $p = [v_1, v_2, \ldots, v_k]$ is an alternating path in which both v_1 and v_k are free vertices. Observe that an augmenting path is defined with respect to a specific matching. The symmetric difference of a matching M and an augmenting path P, $M \oplus P$, is defined to be $(M - P) \cup (P - M)$. It can be shown that $M \oplus P$ is also a matching. Figure 7.1 shows an augmenting path $p = [a, b, c, d, g, h]$ with respect to the given matching. The symmetric difference operation can be also be used as above between two matchings. In this case, the resulting graph consists of a collection of paths and cycles with alternate edges from each matching.

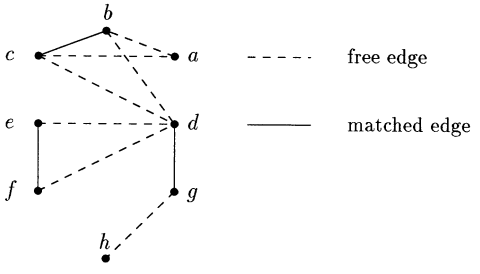


FIGURE 7.1 An augmenting path p with respect to a matching.

Applications of Matching

Matchings lie at the heart of many optimization problems and the problem has many applications: assigning workers to jobs, assigning a collection of jobs with precedence constraints to two processors such that the total execution time is minimized, determining the structure of chemical bonds in Chemistry, matching moving objects based on a sequence of snapshots, and localization of objects in space after obtaining information from multiple sensors (see [1]).

Matchings and Augmenting Paths

The following theorem gives necessary and sufficient conditions for the existence of a perfect matching in a bipartite graph.

THEOREM 7.1 **(Hall's Theorem)** *A bipartite graph $G = (X, Y, E)$ with $|X| = |Y|$ has a perfect matching if and only if $\forall S \subseteq X, |N(S)| \geq |S|$, where $N(S) \subseteq Y$ is the set of vertices that are neighbors of some vertex in S.*

Although the above theorem captures exactly the conditions under which a given bipartite graph has a perfect matching, it does not lead to an algorithm for finding perfect matchings directly. The following lemma shows how an augmenting path with respect to a given matching can be used to increase the size of a matching. An efficient algorithm will be described later that uses augmenting paths to construct a maximum matching incrementally.

LEMMA 7.1 Let P be the edges on an augmenting path $p = [v_1, \ldots, v_k]$ with respect to a matching M. Then $M' = M \oplus P$ is a matching of cardinality $|M| + 1$.

PROOF Since P is an augmenting path, both v_1 and v_k are free vertices in M. The number of free edges in P is one more than the number of matched edges in it. The symmetric difference operator replaces the matched edges of M in P by the free edges in P. Hence, the size of the resulting matching, $|M'|$, is one more than $|M|$.

The following theorem provides a necessary and sufficient condition for a given matching M to be a maximum matching.

THEOREM 7.2 *A matching M in a graph G is a maximum matching if and only if there is no augmenting path in G with respect to M.*

PROOF If there is an augmenting path with respect to M, then M cannot be a maximum matching, since by Lemma 7.1 there is a matching whose size is larger than that of M. To prove the converse we show that if there is no augmenting path with respect to M then M is a maximum matching. Suppose that there is a matching M' such that $|M'| > |M|$. Consider the subgraph of G induced by the edges $M \oplus M'$. Each vertex in this subgraph has degree at most two, since each node has at most one edge from each matching incident to it. Hence, each connected component of this subgraph is either a path or a simple cycle. For each cycle, the number of edges of M is the same as the number of edges of M'. Since $|M'| > |M|$, one of the paths must have more edges from M' than from M. This path is an augmenting path in G with respect to the matching M, contradicting the assumption that there were no augmenting paths with respect to M.

Bipartite Matching Algorithm

High-Level Description: The algorithm starts with the empty matching $M = \emptyset$, and augments the matching in phases. In each phase, an augmenting path with respect to the current matching M is found, and it is used to increase the size of the matching. An augmenting path, if one exists, can be found in $O(|E|)$ time, using a procedure similar to breadth-first search.

The search for an augmenting path proceeds from the free vertices. At each step when a vertex in X is processed, all its unvisited neighbors are also searched. When a matched vertex in Y is considered, only its matched neighbor is searched. This search proceeds along a subgraph referred to as the *Hungarian tree.*

The algorithm uses a queue Q to hold vertices that are yet to be processed. Initially, all free vertices in X are placed in the queue. The vertices are removed one by one from the queue and processed as follows. In turn, when vertex v is removed from the queue, the edges incident to it are scanned. If it has a neighbor in the vertex set Y that is free, then the search for an augmenting path is successful; procedure AUGMENT is called to update the matching, and the algorithm proceeds to its next phase. Otherwise, add the mates of all the matched neighbors of v to the queue if they have never been added to the queue, and continue the search for an augmenting path. If the algorithm empties the queue without finding an augmenting path, its current matching is a maximum matching and it terminates.

The main data structure that the algorithm uses are the arrays *mate* and *free.* The array *mate* is used to represent the current matching. For a matched vertex $v \in G$, $mate[v]$ denotes the matched neighbor of vertex v. For $v \in X$, $free[v]$ is a vertex in Y that is adjacent to v and is free. If no such vertex exists then $free[v] = 0$. The set A stores a set of directed edges (v, v') such that there is an alternating path of two edges from v to v'. This will be used in the search for augmenting paths from free vertices, while extending the alternating paths. When we add a vertex v' to the queue, we set $label[v']$ to v if we came to v' from v, since we need this information to augment on the alternating path we eventually find.

BIPARTITE MATCHING ($G = (X, Y, E)$)
1 **for** all vertices v in G **do**

```
2     mate[v] ← 0.
3   end-for
4   done ← false.
5   while not done do
6     INITIALIZE.
7     MAKEEMPTYQUEUE (Q).
8     for all vertices x ∈ X do        (* add unmatched vertices to Q *)
9       if mate[x] = 0 then
10        PUSH (Q, x).
11        label[x] ← 0.
12      end-if
13    end-for
14    found ← false.
15    while not found and not EMPTY (Q) do
16      x ← POP (Q).
17      if free[x] ≠ 0 then        (* found augmenting path *)
18        AUGMENT (x).
19        found ← true.
20      else        (* extend alternating paths from x *)
21        for all edges (x, x′) ∈ A do
22          if label[x′] = 0 then        (* x′ not already in Q *)
23            label[x′] ← x.
24            PUSH (Q, x′).
25          end-if
26        end-for
27      end-if
28      if EMPTY (Q) then
29        done ← true.
30      end-if
31    end-while
32  end-while
end-proc
```

```
INITIALIZE
1   for all vertices x ∈ X do
2     free[x] ← 0.
3   end-for
4   for all edges (x, y) ∈ E do
5     if mate[y] = 0 then  free[x] ← y
6     else if mate[y] ≠ x then  A ← A ∪ (x, mate[y]).
7     end-if
8   end-for
end-proc
```

```
AUGMENT (x)
1   if label[x] = 0 then
2     mate[x] ← free[x].
```

```
3       mate[free[x]] ← x
4    else
5       free[label[x]] ← mate[x]
6       mate[x] ← free[x]
7       mate[free[x]] ← x
8       AUGMENT (label[x])
9    end-if
end-proc
```

Sample Execution

Figure 7.2 shows a sample execution of the matching algorithm. We start with a partial matching and show the structure of the resulting Hungarian tree. In this example, the search starts from the free vertex b. We add c and e to Q. After we explore c, we add d to Q, and then f and a. Since $free[a] = u$, we stop since an augmenting path from vertex b to vertex u is found by the algorithm.

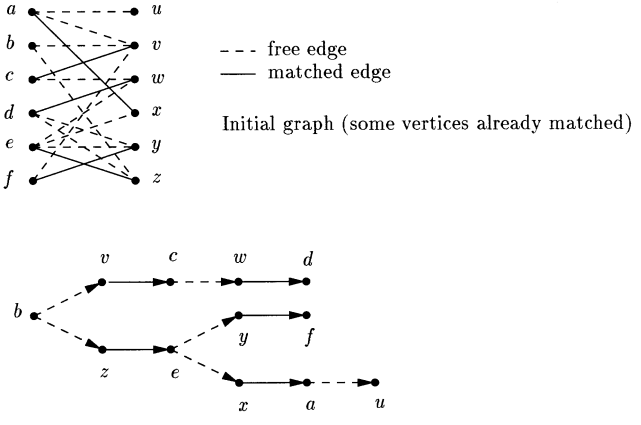


FIGURE 7.2 Sample execution of matching algorithm.

Analysis

If there are augmenting paths with respect to the current matching, the algorithm will find at least one of them. Hence, when the algorithm terminates, the graph has no augmenting paths with respect to the current matching and the current matching is optimal. Each iteration of the main while loop of the algorithm runs in $O(|E|)$ time. The construction of the auxiliary graph A and computation of the array $free$ also take $O(|E|)$ time. In each iteration, the size of the matching increases by one and thus, there are at most $\min(|X|, |Y|)$ iterations of the while loop. Therefore the algorithm solves the matching problem for bipartite graphs in time $O(\min(|X|, |Y|)|E|)$. Hopcroft and Karp (see [26]) showed how to improve the running time by finding a maximal set of disjoint augmenting paths in a single phase in $O(|E|)$ time. They also proved that the algorithm runs in only $O(\sqrt{|V|})$ phases, yielding a worst-case running time of $O(\sqrt{|V|}|E|)$.

The Matching Problem in General Graphs

The techniques used to solve the matching problem in bipartite graphs do not extend directly to non-bipartite graphs. The notion of augmenting paths and their relation to maximum matchings (Theorem 7.2) remain the same. Therefore the natural algorithm of starting with an empty matching and increasing its size repeatedly with an augmenting path until no augmenting paths exist in the graph still works. But the problem of finding augmenting paths in non-bipartite graphs is harder. The main trouble is due to odd length cycles known as *blossoms* that appear along alternating paths explored by the algorithm as it is looking for augmenting paths. We illustrate this difficulty with an example in Fig. 7.3. The search for an augmenting path from an unmatched vertex such as e, could go through the following sequence of vertices $[e, b, g, d, h, g, b, a]$. Even though the augmenting path satisfies the "local" conditions for being an augmenting path, it is not a valid augmenting path since it is not simple. The reason for this is that the odd length cycle (g, d, h, g) causes the path to "fold" on itself—a problem that does not arise in the bipartite case. In fact, the matching does contain a valid augmenting path $[e, f, c, d, h, g, b, a]$. In fact, not all odd cycles cause this problem, but odd cycles that are as dense in matched edges as possible, i.e., it depends on the current matching. By "shrinking" blossoms to single nodes, we can get rid of them [26]. Subsequent work focused on efficient implementation of this method.

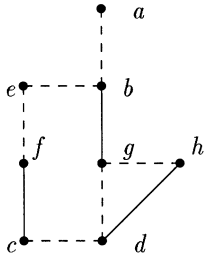


FIGURE 7.3 Difficulty in dealing with blossoms.

Edmonds (see [26]) gave the first polynomial-time algorithm for solving the maximum matching problem in general graphs. The current fastest algorithm for this problem is due to Micali and Vazirani [24] and their algorithm runs in $O(|E|\sqrt{|V|})$ steps, which is the same bound obtained by the Hopcroft–Karp algorithm for finding a maximum matching in bipartite graphs.

Assignment Problem

We now introduce the *assignment problem* — that of finding a maximum weight matching in a given bipartite graph in which edges are given nonnegative weights. There is no loss of generality in assuming that the graph is a complete bipartite graph, since zero-weight edges may be added between pairs of vertices that are nonadjacent in the original graph without affecting the weight of a maximum-weight matching. The minimization version of the weighted version is the problem of finding a *minimum-weight perfect matching* in a complete bipartite graph. Both versions of the weighted matching problem are equivalent and we sketch below how to reduce the minimum-weight perfect matching to maximum-weight matching. Choose a constant W that is larger than the weight of any edge, and assign each edge a new weight of $w'(e) = W - w(e)$. Observe that maximum-weight matchings with the new weight function are minimum-weight perfect matchings with the original weights.

In this section, we restrict our attention to the study of the maximum-weight matching problem for bipartite graphs. Similar techniques have been used to solve the maximum-weight matching problem in arbitrary graphs (see [22, 26]).

The input is a complete bipartite graph $G = (X, Y, X \times Y)$ and each edge e has a nonnegative weight of $w(e)$. The following algorithm is known as the *Hungarian method* (see [1, 23, 26]). The method can be viewed as a primal-dual algorithm in the framework of linear programming [26]. No knowledge of linear programming is assumed here.

A *feasible vertex-labeling* ℓ is defined to be a mapping from the set of vertices in G to the real numbers such that for each edge (x_i, y_j) the following condition holds:

$$\ell(x_i) + \ell(y_j) \geq w(x_i, y_j) .$$

The following can be verified to be a feasible vertex labeling. For each vertex $y_j \in Y$, set $\ell(y_j)$ to be 0, and for each vertex $x_i \in X$, set $\ell(x_i)$ to be the maximum weight of an edge incident to x_i:

$$\begin{aligned} \ell(y_j) &= 0 , \\ \ell(x_i) &= \max_j w(x_i, y_j) . \end{aligned}$$

The *equality subgraph*, G_ℓ, is defined to be the spanning subgraph of G which includes all vertices of G but only those edges (x_i, y_j) which have weights such that

$$\ell(x_i) + \ell(y_j) = w(x_i, y_j) .$$

The connection between equality subgraphs and maximum-weighted matchings is established by the following theorem.

THEOREM 7.3 *If the equality subgraph, G_ℓ, has a perfect matching, M^*, then M^* is a maximum weight matching in G.*

PROOF Let M^* be a perfect matching in G_ℓ. By definition,

$$w(M^*) = \sum_{e \in M^*} w(e) = \sum_{v \in X \cup Y} \ell(v) .$$

Let M be any perfect matching in G. Then

$$w(M) = \sum_{e \in M} w(e) \leq \sum_{v \in X \cup Y} \ell(v) = w(M^*) .$$

Hence, M^* is a maximum weight perfect matching.

High-Level Description: The above theorem is the basis of the following algorithm for finding a maximum weight matching in a complete bipartite graph. The algorithm starts with a feasible labeling, then computes the equality subgraph and a maximum cardinality matching in this subgraph. If the matching found is perfect, by Theorem 7.3, the matching must be a maximum weight matching and the algorithm returns it as its output. Otherwise the matching is not perfect, and more edges need to be added to the equality subgraph by *revising* the vertex labels. The revision should ensure that edges from the current matching do not leave the equality subgraph. After more edges are added to the equality subgraph, the algorithm grows the Hungarian trees further. Either the size of the matching increases because an augmenting path is found, or a new vertex is added to the Hungarian tree. In the former case, the current phase terminates and the algorithm starts a new phase since the matching size has increased. In the latter case, new nodes are added to the Hungarian tree. In $|X|$ phases, the tree includes all the nodes, and therefore there are at most $|X|$ phases before the size of the matching increases.

We now describe in more detail how the labels are updated and which edges are added to the equality subgraph. Suppose M is a maximum matching found by the algorithm. Hungarian trees are grown from all the free vertices in X. Vertices of X (including the free vertices) that are encountered in the search are added to a set S and vertices of Y that are encountered in the search are added to a set T. Let $\overline{S} = X - S$ and $\overline{T} = Y - T$. Figure 7.4 illustrates the structure of the sets S and T. Matched edges are shown in bold; the other edges are the edges in G_ℓ. Observe that there are no edges in the equality subgraph from S to $\overline{T}$, even though there may be edges from T to $\overline{S}$. The algorithm now revises the labels as follows. Decrease all the labels of vertices in S by a quantity δ (to be determined later) and increase the labels of the vertices in T by δ. This ensures that edges in the matching continue to stay in the equality subgraph. Edges in G (not in G_ℓ) that go from vertices in S to vertices in $\overline{T}$ are candidate edges to enter the equality subgraph, since one label is decreasing and the other is unchanged. The algorithm chooses δ to be the smallest value such that some edge of $G - G_\ell$ enters the equality subgraph. Suppose this edge goes from $x \in S$ to $y \in \overline{T}$. If y is free then an augmenting path has been found. On the other hand if y is matched, the Hungarian tree is grown by moving y to T and its matched neighbor to S and the process of revising labels is continued.

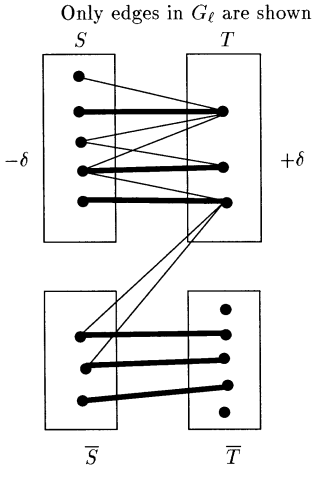


FIGURE 7.4 Sets S and T as maintained by the algorithm.

Weighted Edge-Cover Problem

There are many applications of weighted matchings. One useful application is the problem of finding a *minimum-weight edge cover* of a graph. Given a graph $G = (V, E)$ with weights on the edges, a set of edges $C \subseteq E$ forms an *edge cover* of G if every vertex in V is incident to at least one edge in C. The weight of the cover C is the sum of the weights of its edges. The objective is to find an edge cover C of minimum weight. The problem can be reduced to the minimum-weight perfect matching problem as follows: create an identical copy $G' = (V', E')$ of graph G, except the weights of edges in E' are all 0. Add an edge from each $v \in V$ to $v' \in V'$ with weight $w_{\min}(v) = \min_{x \in N(v)} w(v, x)$. The final graph H is the union of G and G', together with the edges connecting the two copies of each vertex; H contains $2|V|$ vertices and $2|E| + |V|$ edges. There exist other reductions from minimum-weight edge-cover to minimum-weight perfect matching; the reduction outlined above has the advantage that it creates a graph with $O(|V|)$ vertices and $O(|E|)$ edges. The minimum weight perfect matching problem may be solved using the techniques described earlier for the bipartite case, but the algorithm is more complex [22, 26].

THEOREM 7.4 *The weight of a minimum-weight perfect matching in H is equal to the weight of a minimum-weight edge cover in G.*

PROOF Consider a minimum-weight edge cover C in G. There is no loss of generality in assuming that a minimum-weight edge cover has no path of three edges, since the middle edge can be removed from the cover, thus, reducing its weight further. Hence, the edge cover C is a union of "stars" (trees of height 1). For each star that has a vertex of degree more than one, we can match one of the leaf nodes to its copy in G', with weight at most the weight of the edge incident on the leaf vertex, thus, reducing the degree of the star. We repeat this until each vertex has degree at most one in the edge cover, i.e., it is a matching. In H, select this matching, once in each copy of G. Observe that the cost of the matching within the second copy G' is 0. Thus, given C, we can find a perfect matching in H whose weight is no larger.

To argue the converse, we now show how to construct a cover C from a given perfect matching M in H. For each edge $(v, v') \in M$, add to C a least weight edge incident to v in G. Also include in C any edge of G that was matched in M. It can be verified that C is an edge cover whose weight equals the weight of M.

7.3 The Network Flow Problem

A number of polynomial time flow algorithms have been developed over the last two decades. The reader is referred to the books by Ahuja et al. [1] and Tarjan [29] for a detailed account of the historical development of the various flow methods. An excellent survey on network flow algorithms has been written by Goldberg et al. [13]. The book by Cormen et al. [5] describes the preflow push method, and to complement their coverage an implementation of the **blocking flow** technique of Malhotra et al. (see [26]) is discussed here.

Network Flow Problem Definitions

Flow network: A flow network $G = (V, E)$ is a directed graph with two specially marked nodes, namely, the source s, and the sink t. A *capacity* function $u : E \mapsto \Re^+$ maps edges to positive real numbers.
Max-flow problem: A flow function $f : E \mapsto \Re$ maps edges to real numbers. For an edge $e = (v, w)$, $f(v, w)$ refers to the flow on edge e, which is also called the net flow from vertex v to vertex w. This notation is extended to sets of vertices as follows: If X and Y are sets of vertices then $f(X, Y)$ is defined to be $\Sigma_{x \in X} \Sigma_{y \in Y} f(x, y)$. A flow function is required to satisfy the following constraints:

- (Capacity constraint) For all edges e, $f(e) \leq u(e)$.
- (Skew symmetry constraint) For an edge $e = (v, w)$, $f(v, w) = -f(w, v)$.
- (Flow conservation) For all vertices $v \in V - \{s, t\}$, $\Sigma_{w \in V} f(v, w) = 0$.

The capacity constraint states that the total flow on an edge does not exceed its capacity. The skew symmetry condition states that the flow on an edge is the negative of the flow in the reverse direction. The flow conservation constraint states that the total net flow out of any vertex other than the source and sink is zero.

The *value* of the flow is defined to be the net flow out of the source vertex:

$$|f| = \sum_{v \in V} f(s, v) \; .$$

In the *maximum flow problem* the objective is to find a flow function that satisfies the above three constraints, and also maximizes the total flow value $|f|$.

Remarks: This formulation of the network flow problem is powerful enough to capture generalizations where there are many sources and sinks (single commodity flow), and where both vertices and edges have

capacity constraints. To reduce multiple sources to a single source, we add a new source vertex with edges connecting it to the original source vertices. To reduce multiple sinks to a single sink, we add a new sink vertex with edges from the original sinks to the new sink. It is easy to reduce the vertex capacity problem to edge capacities by "splitting" a vertex into two vertices, and making all the incoming edges come into the first vertex and the outgoing edges come out of the second vertex. We then add an edge between them with the capacity of the corresponding vertex, so that the entire flow through the vertex is forced through this edge. The problem for undirected graphs can be solved by treating each undirected edge as two directed edges with the same capacity as the original undirected edge.

First the notion of cuts is defined, and then the max-flow min-cut theorem is introduced. We then introduce residual networks, layered networks and the concept of blocking flows. We then show how to reduce the max-flow problem to the computation of a sequence of blocking flows. Finally, an efficient algorithm for finding a blocking flow is described.

A *cut* is a partitioning of the vertex set into two subsets S and $V - S$. An edge *crosses* the cut if it connects a vertex $x \in S$ to a vertex $y \in V - S$. An s-t **cut** of the graph is a partitioning of the vertex set V into two sets S and $T = V - S$ such that $s \in S$ and $t \in T$. If f is a flow then the net flow across the cut is defined as $f(S, T)$. The capacity of the cut is defined as $u(S, T) = \Sigma_{x \in X} \Sigma_{y \in Y} u(x, y)$. The net flow across a cut may include negative net flows between vertices, but the capacity of the cut includes only non-negative values, i.e., only the capacities of edges from S to T. The net flow across a cut includes negative flows between vertices (flow from T to S), but the capacity of the cut includes only the capacities of edges from S to T.

Using the flow conservation principle, it can be shown that the net flow across an s-t cut is exactly the flow value $|f|$. By the capacity constraint, the flow across the cut cannot exceed the capacity of the cut. Thus, the value of the maximum flow is no greater than the capacity of a minimum s-t cut. The *max-flow min-cut theorem* states that the two quantities are actually equal. In other words, if f^* is a maximum flow, then there is some cut $(X, \overline{X})$ with $s \in X$ and $t \in \overline{X}$, such that $|f^*| = u(X, \overline{X})$. The reader is referred to [5, 29] for further details.

The *residual capacity* of an edge (v, w) with respect to a flow f is defined as follows:

$$u'(v, w) = u(v, w) - f(v, w) .$$

The quantity $u'(v, w)$ is the number of additional units of flow that can be pushed from v to w without violating the capacity constraints. An edge e is *saturated* if $u(e) = f(e)$, i.e., if its residual capacity $u'(e) = 0$. The residual graph, $G_R(f)$, for a flow f, is the graph with vertex set V, source and sink s and t, respectively, and those edges (v, w) for which $u'(v, w) > 0$. Figure 7.5 shows a network on the left with a given flow function. Each edge is labeled with two values: the flow through that edge and its capacity. The figure on the right depicts the residual network corresponding to this flow.

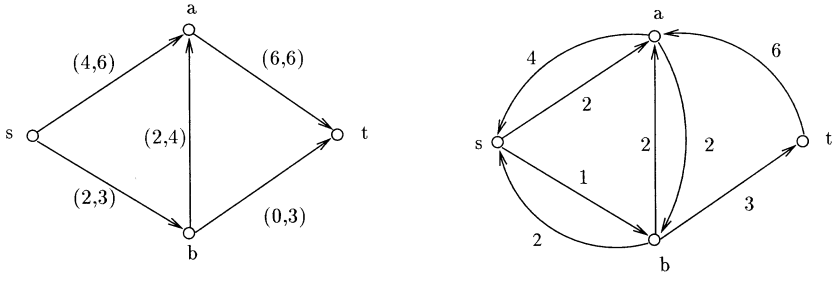


FIGURE 7.5 A network with a given flow function, and its residual network. A label of (f, c) for an edge indicates a flow of f and capacity of c.

An *augmenting path* for f is a path P from s to t in $G_R(f)$. The residual capacity of P, denoted by $u'(P)$, is the minimum value of $u'(v, w)$ over all edges (v, w) in the path P. The flow can be increased by $u'(P)$, by increasing the flow on each edge of P by this amount. Whenever $f(v, w)$ is changed, $f(w, v)$ is also correspondingly changed to maintain skew symmetry.

Most flow algorithms are based on the concept of augmenting paths pioneered by Ford and Fulkerson [8]. They start with an initial zero flow and augment the flow in stages. In each stage, a residual graph $G_R(f)$ with respect to the current flow function f is constructed, and an augmenting path in $G_R(f)$ is found, thus, increasing the value of the flow. Flow is increased along this path until an edge in this path is saturated. The algorithms iteratively keep increasing the flow until there are no more augmenting paths in $G_R(f)$, and return the final flow f as their output. Edmonds and Karp [6] suggested two possible improvements to this algorithm to make it run in polynomial time. The first was to choose shortest possible augmenting paths, where the length of the path is simply the number of edges on the path. This method can be improved using the approach due to Dinitz (see [29]) in which a sequence of blocking flows is found. The second strategy was to select a path which can be used to push the maximum amount of flow. The number of iterations of this method is bounded by $O(|E| \log C)$ where C is the largest edge capacity. In each iteration, we need to find a path on which we can push the maximum amount of flow. This can be done by suitably modifying Dijkstra's algorithm (Chapter 6).

The following lemma is fundamental in understanding the basic strategy behind these algorithms.

LEMMA 7.2 Let f be a flow and f^* be a maximum flow in G. The value of a maximum flow in the residual graph $G_R(f)$ is $|f^*| - |f|$.

PROOF Let f' be any flow in $G_R(f)$. Define $f + f'$ to be the flow $f(v, w) + f'(v, w)$ for each edge (v, w). Observe that $f + f'$ is a feasible flow in G of value $|f| + |f'|$. Since f^* is the maximum flow possible in G, $|f'| \leq |f^*| - |f|$. Similarly define $f^* - f$ to be a flow in $G_R(f)$ defined by $f^*(v, w) - f(v, w)$ in each edge (v, w), and this is a feasible flow in $G_R(f)$ of value $|f^*| - |f|$, and it is a maximum flow in $G_R(f)$.

Blocking Flow: A flow f is a *blocking flow* if every path from s to t in G contains a saturated edge. Blocking flows are also known as maximal flows. Figure 7.6 depicts a blocking flow. Any path from s to t contains at least one saturated edge. For example, in the path $[s, a, t]$, the edge (a, t) is saturated. It is important to note that a blocking flow is not necessarily a maximum flow. There may be augmenting paths that increase the flow on some edges and decrease the flow on other edges (by increasing the flow in the reverse direction). The example in Fig. 7.6 contains an augmenting path $[s, a, b, t]$, which can be used to increase the flow by 3 units.

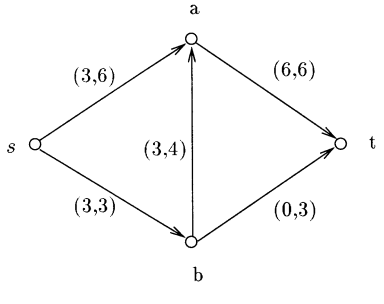


FIGURE 7.6 Example of blocking flow — all paths from s to t are saturated. A label of (f, c) for an edge indicates a flow of f and capacity of c.

Layered Networks: Let $G_R(f)$ be the residual graph with respect to a flow f. The level of a vertex v is the length of a shortest path from s to v in $G_R(f)$. The level graph L for f is the subgraph of $G_R(f)$ containing vertices reachable from s and only the edges (v, w) such that $level(w) = 1 + level(v)$. L contains all shortest length augmenting paths and can be constructed in $O(|E|)$ time.

The maximum flow algorithm proposed by Dinitz starts with the zero flow, and iteratively increases the flow by augmenting it with a blocking flow in $G_R(f)$ until t is not reachable from s in $G_R(f)$. At each step the current flow is replaced by the sum of the current flow and the blocking flow. This algorithm terminates in $|V| - 1$ iterations, since in each iteration the shortest distance from s to t in the residual graph increases. The shortest path from s to t is at most $|V| - 1$, and this gives an upper bound on the number of iterations of the algorithm.

An algorithm to find a blocking flow that runs in $O(|V|^2)$ time is described here (see [26] for details), and this yields an $O(|V|^3)$ max-flow algorithm. There are a number of $O(|V|^2)$ blocking flow algorithms available, some of which are described in [29].

Blocking Flows

Dinitz's algorithm finds a blocking flow as follows. The main step is to find paths from the source to the sink and push as much flow as possible on each path, and thus, saturate an edge in each path. It takes $O(|V|)$ time to compute the amount of flow that can be pushed on a path. Since there are $|E|$ edges, this yields an upper bound of $O(|V||E|)$ steps on the running time of the algorithm. The following algorithm shows how to find a blocking flow more efficiently.

Malhotra–Kumar–Maheshwari Blocking Flow Algorithm: The algorithm has a current flow function f and its corresponding residual graph $G_R(f)$. Define for each node $v \in G_R(f)$, a quantity $tp[v]$ that specifies its maximum throughput, i.e., either the sum of the capacities of the incoming arcs or the sum of the capacities of the outgoing arcs, whichever is smaller. The quantity $tp[v]$ represents the maximum flow that could pass through v in any feasible blocking flow in the residual graph. Vertices with zero throughput are deleted from $G_R(f)$.

The algorithm selects a vertex x with least throughput. It then greedily pushes a flow of $tp[x]$ from x toward t, level by level in the layered residual graph. This can be done by creating a queue which initially contains x, which is assigned the task of pushing $tp[x]$ out of it. In each step, the vertex v at the front of the queue is removed, and the arcs going out of v are scanned one at a time, and as much flow as possible is pushed out of them until v's allocated flow has been pushed out. For each arc (v, w) that the algorithm pushed flow through, it updates the residual capacity of the arc (v, w) and places w on a queue (if it is not already there) and increments the net incoming flow into w. Also $tp[v]$ is reduced by the amount of flow that was sent through it now. The flow finally reaches t, and the algorithm never comes across a vertex that has incoming flow that exceeds its outgoing capacity since x was chosen as a vertex with least throughput. The above idea is repeated to pull a flow of $tp[x]$ from the source s to x. Combining the two steps yields a flow of $tp[x]$ from s to t in the residual network that goes through x. The flow f is augmented by this amount. Vertex x is deleted from the residual graph, along with any other vertices that have zero throughput.

The above procedure is repeated until all vertices are deleted from the residual graph. The algorithm has a blocking flow at this stage since at least one vertex is saturated in every path from s to t. In the above algorithm, whenever an edge is saturated it may be deleted from the residual graph. Since the algorithm uses a greedy strategy to pump flows, at most $O(|E|)$ time is spent when an edge is saturated. When finding flow paths to push $tp[x]$, there are at most $|V|$ times (once for each vertex) when the algorithm pushes a flow that does not saturate the corresponding edge. After this step, x is deleted from the residual graph. Hence, the above algorithm to compute blocking flows terminates in $O(|E| + |V|^2) = O(|V|^2)$ steps.

Karzanov (see [29]) used the concept of preflows to develop an $O(|V|^2)$ time algorithm for developing blocking flows as well. This method was then adapted by Goldberg and Tarjan (see [5]), who proposed

a preflow-push method for computing a maximum flow in a network. A preflow is a flow that satisfies the capacity constraints, but not flow conservation. Any node in the network is allowed to have more flow coming into it than there is flowing out. The algorithm tries to move the excess flows towards the sinks without violating capacity constraints. The algorithm finally terminates with a feasible flow that is a max-flow. It finds a max-flow in $O(|V||E| \log \frac{|V|^2}{|E|})$ time.

7.4 The Min-Cut Problem

Two problems are studied in this section. The first is the s-t min-cut problem: given a directed graph with capacities on the edges and two special vertices s and t, the problem is to find a cut of minimum capacity that separates s from t. The value of a min-cut's capacity can be computed by finding the value of the maximum flow from s to t in the network. Recall that the max-flow min-cut theorem shows that the two quantities are equal. We now show how to find the sets S and T that provides an s-t min-cut from a given s-t max-flow.

The second problem is to find a smallest cut in a given graph G. In this problem on undirected graphs, the vertex set must be partitioned into two sets such that the total weight of the edges crossing the cut is minimized. This problem can be solved by enumerating over all $\{s, t\}$ pairs of vertices, and finding a s-t min-cut for each pair. This procedure is rather inefficient. However, Hao and Orlin [15] showed that this problem can be solved in the same order of running time as taken by a single max-flow computation. Stoer and Wagner [27] have given a simple and elegant algorithm for computing minimum cuts based on a technique due to Nagamochi and Ibaraki [25]. Recently, Karger [19] has given a randomized algorithm that runs in almost linear time for computing minimum cuts without using network-flow methods.

Finding an s-t Min-Cut

Let f^* be a maximum flow from s to t in the graph. Recall that $G_R(f^*)$ is the residual graph corresponding to f^*. Let S be the set of all vertices that are reachable from s in $G_R(f^*)$, i.e., vertices to which s has a directed path. Let T be all the remaining vertices of G. By definition $s \in S$. Since f^* is a max-flow from s to t, $t \in T$. Otherwise the flow can be increased by pushing flow along this path from s to t in $G_R(f^*)$. All the edges from vertices in S to vertices in T are saturated and form a minimum cut.

Finding All-Pair Min-Cuts

For an undirected graph G with $|V|$ vertices, Gomory and Hu (see [1]) showed that the flow values between each of the $|V|(|V|-1)/2$ pairs of vertices of G can be computed by solving only $|V|-1$ max-flow computations. Furthermore, they showed that the flow values can be represented by a weighted tree T on $|V|$ nodes. Each node in the tree represents one of the vertices of the given graph. For any pair of nodes (x, y), the maximum flow value from x to y (and hence, the x-y min-cut) in G is equal to the weight of the minimum-weight edge on the unique path in T between x and y.

Applications of Network Flows (and Min Cuts)

There are numerous applications of the maximum flow algorithm in scheduling problems of various kinds. It is used in open-pit mining, vehicle routing, etc. See [1] for further details.

Finding a minimum-weight vertex cover in bipartite graphs: A *vertex cover* of a graph is a set of vertices that is incident to all its edges. The weight of a vertex cover is the sum of the weights of its vertices. Finding a vertex cover of minimum weight is NP-hard for arbitrary graphs (see Section 7.9 for more details). For bipartite graphs, the problem is solvable efficiently using the maximum flow algorithm.

Given a bipartite graph $G = (X, Y, E)$ with weights on the vertices, we need to find a subset C of

vertices of minimum total weight, so that for each edge, at least one of its end vertices is in C. The weight of a vertex v is denoted by $w(v)$. Construct a flow network N from G as follows: add a new source vertex s, and for each $x \in X$, add a directed edge (s, x), with capacity $w(x)$. Add a new sink vertex t, and for each $y \in Y$, add a directed edge (y, t), with capacity $w(y)$. The edges in E are given infinite capacity and they are oriented from X to Y. Let C be a subset of vertices in G. We use the notation X_C to denote $X \cap C$ and Y_C to denote $Y \cap C$. Let $\overline{X_C} = X - X_C$ and $\overline{Y_C} = Y - Y_C$.

If C is a vertex cover of G, then there are no edges in G connecting $\overline{X_C}$ and $\overline{Y_C}$, since such edges would not be incident to C. A cut in the flow network N, whose capacity is the same as the weight of the vertex cover can be constructed as follows: $S = \{s\} \cup \overline{X_C} \cup Y_C$ and $T = \{t\} \cup X_C \cup \overline{Y_C}$. Each vertex cover of G gives rise to a cut of same weight. Similarly, any cut of finite capacity in N corresponds to a vertex cover of G of same weight. Hence, the minimum-weight s-t cut of N corresponds to a minimum-weight vertex cover of G.

Maximum-weight closed subset in a partial order: Consider a *directed acyclic graph* (DAG) $G = (V, E)$. Each vertex v of G is given a weight $w(v)$, that may be positive or negative. A subset of vertices $S \subseteq V$ is said to be *closed* in G if for every $s \in S$, the predecessors of s are also in S. The predecessors of a vertex s are vertices that have directed paths to s. We are interested in computing a closed subset S of maximum weight. This problem occurs for example in open-pit mining, and it can be solved efficiently by reducing it to computing the minimum cut in the following network N:

- The vertex set of $N = V \cup \{s, t\}$, where s and t are two new vertices.
- For each vertex v of negative weight, add the edge (s, v) with capacity $|w(v)|$ to N.
- For each vertex v of positive weight, add the edge (v, t) with capacity $w(v)$ to N.
- All edges of G are added to N, and each of these edges has infinite capacity.

Consider a minimum s-t cut in N. Let S be the positive weight vertices whose edges to the sink t are not in the min-cut. It can be shown that the union of the set S and its predecessors is a maximum-weight closed subset.

7.5 Minimum-Cost Flows

We now study one of the most important problems in combinatorial optimization, namely the minimum cost network flow problem.

We will outline some of the basic ideas behind the problem, and the reader can find a wealth of information in [1] about efficient algorithms for this problem.

The min-cost flow problem can be viewed as a generalization of the max-flow problem, the shortest path problem and the minimum weight perfect matching problem. To reduce the shortest path problem to the min-cost flow problem, notice that if we have to ship one unit of flow, we will ship it on the shortest path. We reduce the minimum weight perfect matching problem on a bipartite graph $G = (X, Y, X \times Y)$, to min-cost flow as follows. Introduce a special source s and sink t, and add unit capacity edges from s to vertices in X, and from vertices in Y to t. Compute a flow of value $|X|$ to obtain a minimum weight perfect matching.

Min-Cost Flow Problem Definitions

The flow network $G = (V, E)$ is a directed graph. There is a *capacity* function $u : E \mapsto \Re^+$ that maps edges to positive real numbers and a cost function $c : E \rightarrow R$. The cost function specifies the cost of shipping one unit of flow through the edge. Associated with each vertex v, there is a supply $b(v)$. If $b(v) > 0$ then v is a supply node, and if $b(v) < 0$ then it is a demand node. It is assumed that $\Sigma_{v \in V} b(v) = 0$; if the supply exceeds the demand we can add an artificial sink to absorb the excess flow.

A flow function is required to satisfy the following constraints:

- (Capacity Constraint) For all edges e, $f(e) \leq u(e)$.
- (Skew Symmetry Constraint) For an edge $e = (v, w)$, $f(v, w) = -f(w, v)$.
- (Flow Conservation) For all vertices $v \in V$, $\Sigma_{w \in V} f(v, w) = b(v)$.

We wish to find a flow function that minimizes the cost of the flow.

$$\min z(f) = \sum_{e \in E} c(e) \cdot f(e) .$$

Our algorithm again uses the concept of residual networks. As before, the residual network $G_R(f)$ is defined with respect to a specific flow function. The only difference is the following: if there is a flow $f(v, w)$ on edge $e = (v, w)$ then as before, its capacity in the residual network is $u(e) - f(e)$ and its residual cost is $c(e)$. The reverse edge (w, v) has residual capacity $f(e)$, but its residual cost is $-c(e)$. Note that sending a unit of flow on the reverse edge actually reduces the original amount of flow that was sent, and hence, the residual cost is negative. As before, only edges with strictly positive residual capacity are retained in the residual network.

Before studying any specific algorithm to solve the problem, we note that we can always find a feasible solution (not necessarily optimal), if one exists, by finding a max-flow, and ignoring costs. To see this, add two new vertices, a source vertex s and a sink vertex t. Add edges from s to all vertices v with $b(v) > 0$ of capacity $b(v)$, and edges from all vertices w with $b(w) < 0$ to t of capacity $|b(w)|$. Find a max-flow of value $\Sigma_{b(v)>0} b(v)$. If such flow does not exist then the flow problem is not feasible.

The following theorem characterizes optimal solutions.

THEOREM 7.5 *A feasible solution f is an optimal solution if and only if the residual network $G_R(f)$ has no directed cycles of negative cost.*

We use the above theorem to develop an algorithm for finding a min-cost flow. As indicated earlier, a feasible flow can be found by the techniques developed in the previous section. To improve the cost of the solution, we identify negative-cost cycles in the residual graph and push as much flow around them as possible and reduce the cost of the flow. We repeat this until there are no negative cycles left, and we have found a min-cost flow. A negative-cost cycle may be found by using an algorithm like the Bellman–Ford algorithm (Chapter 6), which takes $O(|E||V|)$ time.

An important issue that affects the running time of the algorithm is the choice of a negative cost cycle. For fast convergence, one should select a cycle that decreases the cost as much as possible, but finding such a cycle in NP-hard. Goldberg and Tarjan [12] showed that selecting a cycle with minimum mean-cost (the ratio of the cost of cycle to the number of arcs on the cycle) yields a polynomial time algorithm. There is a parametric search algorithm [1] to find a min-mean cycle that works as follows. Let μ^* be the average edge-weight of a min-mean cycle. Then μ^* is the largest value such that reducing the weight of every edge by μ^* does not introduce a negative-weight cycle into the graph. One can search for the value of μ^* using binary search: given a guess μ, decrease the weight of each edge by μ and test if the graph has no negative weight cycles. If the smallest cycle has zero weight, it is a min-mean cycle. If all cycles are positive, we have to increase our guess μ. If the graph has a negative weight cycle, we need to decrease our guess μ. Karp (see [1]) has given an algorithm based on dynamic programming for this problem that runs in $O(|E||V|)$ time.

The first polynomial-time algorithm for the min-cost flow problem was given by Edmonds and Karp [6]. Their idea is based on scaling capacities and the running time of their algorithm is $O(|E| \log U(|E| + |V| \log |V|))$, where U is the largest capacity. The first strongly polynomial algorithm (whose running time is only a function of $|E|$ and $|V|$) for the problem was given by Tardos [28].

7.6 The Multi-Commodity Flow Problem

A natural generalization of the max-flow problem is the **multi-commodity flow** problem. In this problem, there are a number of commodities $1, 2, \ldots, k$ that have to be shipped through a flow network from source vertices $s_1, s_2, \ldots, s_k$ to sink vertices $t_1, t_2, \ldots, t_k$, respectively. The amount of demand for commodity i is specified by d_i. Each edge and each vertex has a nonnegative capacity that specifies the total maximum flow of all commodities flowing through that edge or vertex. A flow is *feasible* if all the demands are routed from their respective sources to their respective sinks while satisfying flow conservation constraints at the nodes and capacity constraints on the edges and vertices. A flow is an ϵ*-feasible* flow for a positive real number ϵ if it satisfies $\frac{d_i}{1+\epsilon}$ of demand i. There are several variations of this problem as noted below:

In the simplest version of the *multi-commodity flow problem,* the goal is to decide if there exists a feasible flow that routes all the demands of the k commodities through the network. The flow corresponding to each commodity is allowed to be split arbitrarily (i.e., even fractionally) and routed through multiple paths. The **concurrent flow** problem is identical to the multi-commodity flow problem when the input instance has a feasible flow that satisfies all the demands. If the input instance is not feasible, then the objective is to find the smallest fraction ϵ for which there is an ϵ-feasible flow. In the *minimum-cost multi-commodity flow problem,* each edge has an associated cost for each unit of flow through it. The objective is to find a minimum-cost solution for routing all the demands through the network. All of the above problems can be formulated as linear programs, and therefore have polynomial-time algorithms using either the ellipsoid algorithm or the interior point method [20].

A lot of research has been done on finding approximately optimal multicommodity flows using more efficient algorithms. There are a number of papers that provide approximation algorithms for the multi-commodity flow problem. For a detailed survey of these results see [16, Chapter 5].

In this section, we will discuss a recent paper that introduced a new algorithm for solving multi-commodity flow problems. Awerbuch and Leighton [2] gave a simple and elegant algorithm for the concurrent flow problem. Their algorithm is based on the "liquid-flow paradigm." Flow is pushed from the sources on edges based on the "pressure difference" between the end vertices. The algorithm does not use any global properties of the graph (such as shortest paths) and uses only local changes based on the current flow. Due to its nature, the algorithm can be implemented to run in a distributed network since all decisions made by the algorithm are local in nature. They extended their algorithms to dynamic networks, where edge capacities are allowed to vary [3]. The best implementation of this algorithm runs in $O(k|V|^2|E|\epsilon^{-3}\ln^3(|E|/\epsilon))$ time.

In the *integer multi-commodity flow* problem, the capacities and flows are restricted to be integers. Unlike the single-commodity flow problem, for problems with integral capacities and demands, the existence of a feasible fractional solution to the multi-commodity flow problem does not guarantee a feasible integral solution. An extra constraint that may be imposed on this problem is to restrict each commodity to be sent along a single path. In other words, this constraint does not allow one to split the demand of a commodity into smaller parts and route them along independent paths (see the recent paper by Kleinberg [21] for approximation algorithms for this problem). Such constraints are common in problems that arise in telecommunication systems. All variations of the **integer multi-commodity flow** problem are NP-hard.

Local Control Algorithm

In this section, we describe Awerbuch and Leighton's local control algorithm that finds an approximate solution for the concurrent flow problem.

First, the problem is converted to the *continuous flow* problem. In this version of the problem, d_i units of commodity i is added to the source vertex s_i in each phase. Each vertex has queues to store the commodities that arrive at that node. The node tries to push the commodities stored in the queues towards the sink nodes. As the commodities arrive at the sinks, they are removed from the network. A continuous flow problem is *stable* if the total amount of all the commodities in the network at any point in time is

bounded (i.e., independent of the number of phases completed). The following theorem establishes a tight relationship between the multi-commodity flow problem (also known as the *static* problem) and its continuous version.

THEOREM 7.6 *A stable algorithm for the continuous flow problem can be used to generate an ϵ-feasible solution for the static concurrent flow problem.*

PROOF Let R be the number of phases of the stable algorithm for the continuous flow problem at which $\frac{1}{1+\epsilon}$ fraction of each of the commodities that have been injected into the network have been routed to their sinks. We know that R exists since the algorithm is stable. The average behavior of the continuous flow problem in these R phases generates an ϵ-feasible flow for the static problem. In other words, the flow of a commodity through a given edge is the average flow of that commodity through that edge over R iterations, and this flow is ϵ-feasible.

In order to get an approximation algorithm using the fluid-flow paradigm, there are two important challenges. First, it must be shown that the algorithm is stable, i.e., that the queue sizes are bounded. Second, the number of phases that it takes the algorithm to reach "steady state" (specified by R in the above theorem) must be minimized. Note that the running time of the approximation algorithm is R times the time it takes to run one phase of the continuous flow algorithm.

Each vertex v maintains one queue per commodity for each edge to which it is incident. Initially all the queues are empty. In each phase of the algorithm, commodities are added at the source nodes. The algorithm works in four steps:

1. Add $(1+\epsilon)d_i$ units of commodity i at s_i. The commodity is spread equally to all the queues at that vertex.
2. Push the flow across each edge so as to balance the queues as much as possible. If Δ_i is the discrepancy of commodity i in the queues at either end of edge e, then the flow f_i crossing the edge in that step is chosen so as to maximize $\Sigma_i f_i(\Delta_i - f_i)/d_i^2$ without exceeding the capacity constraints.
3. Remove the commodities that have reached their sinks.
4. Balance the amount of each commodity in each of the queues at each vertex.

The above 4 steps are repeated until $\frac{1}{1+\epsilon}$ fraction of the commodities that have been injected into the system have reached their destination.

It can be shown that the algorithm is stable and the number of phases is $O(|E|^2 k^{1.5} L\epsilon^{-3})$, where $|E|$ is the number of edges, k is the number of commodities, and $L \leq |V|$ is the maximum path length of any flow. The proof of stability and the bound on the number of rounds are not included here. The actual algorithm has a few other technical details that are not mentioned here, and the reader is referred to the original paper for further details [2].

7.7 Minimum Weight Branchings

A natural analog of a spanning tree in a directed graph is a *branching* (also called an *arborescence*). For a directed graph G and a vertex r, a branching rooted at r is an acyclic subgraph of G in which each vertex but r has exactly one outgoing edge, and there is a directed path from any vertex to r. This is also sometimes called an *in-branching*. By replacing "outgoing" in the above definition of a branching to "incoming," we get an *out-branching*. An *optimal branching* of an edge-weighted graph is a branching of minimum total weight. Unlike the minimum spanning tree problem, an optimal branching cannot be

computed using a greedy algorithm. Edmonds gave the first polynomial time algorithm to find optimal branchings (see [11]).

Let $G = (V, E)$ be an arbitrary graph, let r be the root of G, and let $w(e)$ be the weight of edge e. Consider the problem of computing an optimal branching rooted at r. In the following discussion, assume that all edges of the graph have a nonnegative weight.

Two key ideas are discussed below that can be converted into a polynomial time algorithm for computing optimal branchings. First, for any vertex $v \neq r$ in G, suppose that all outgoing edges of v are of positive weight. Let $\epsilon > 0$ be a number that is less than or equal to the weight of any outgoing edge from v. Suppose the weight of every outgoing edge from v is decreased by ϵ. Observe that since any branching has exactly one edge out of v, its weight decreases by exactly ϵ. Therefore an optimal branching of G with the original weights is also an optimal branching of G with the new weights. In other words, decreasing the weight of the outgoing edges of a vertex uniformly, leaves optimal branchings invariant. Second, suppose there is a cycle C in G consisting only of edges of zero weight. Suppose the vertices of C are combined into a single vertex, and the resulting multigraph is made into a simple graph by replacing each multiple edge by a single edge whose weight is the smallest among them, and discarding self loops. Let G_C be the resulting graph. It can be shown that the weights of optimal branchings of G and G_C are the same. An optimal branching of G_C can also be converted into an optimal branching of G by adding sufficiently many edges from C without introducing cycles. The above two ideas can be used to design a recursive algorithm for finding an optimal branching. The fastest implementation of a branching algorithm is due to Gabow et al. [9].

7.8 Coloring Problems

Vertex Coloring

A **vertex coloring** of a graph G is a coloring of the vertices of G such that no two adjacent vertices of G receive the same color. The objective of the problem is to find a coloring that uses as few colors as possible. The minimum number of colors needed to color the vertices of a graph is known as its **chromatic number.** The register allocation problem in compiler design and the map coloring problem are instances of the vertex coloring problem.

The problem of deciding if the chromatic number of a given graph is at most a given integer k is NP-complete. The problem is NP-complete even for fixed $k \geq 3$. For $k = 2$, the problem is to decide if the given graph is bipartite, and this can be solved in linear time using depth-first or breadth-first search. The vertex coloring problem in general graphs is a notoriously hard problem and it has been shown to be intractable even for approximating within a factor of n^{ϵ} for some constant $\epsilon > 0$ [16, Chapter 10]. A greedy coloring of a graph yields a coloring that uses at most $\Delta + 1$ colors, where Δ is the maximal degree of the graph. Unless the graph is an odd cycle or the complete graph, it can be colored with Δ colors (known as Brooks theorem) [11]. A celebrated result on graph coloring is that every planar graph is 4-colorable. However, checking if a planar graph is 3-colorable is NP-complete [10]. For more information on recent results on approximating the chromatic number, see [18].

Edge Coloring

The **edge coloring** problem is similar to the vertex coloring problem. In this problem, the goal is to color the edges of a given graph using the fewest colors such that no two edges incident to a common vertex are assigned the same color. The problem finds applications in assigning classroom to courses and in scheduling problems. The minimum number of colors needed to color a graph is known as its **chromatic index.** Since any two incident edges must receive distinct colors, the chromatic index of a graph with maximal degree Δ is at least Δ. In a remarkable theorem, Vizing (see [11]) has shown that every graph can be edge-colored using at most $\Delta + 1$ colors. Deciding whether the edges of a given graph can be colored

using Δ colors is NP-complete (see [16]). Special classes of graphs such as bipartite graphs and planar graphs of large maximal degree are known to be Δ-edge-colorable.

7.9 Approximation Algorithms for Hard Problems

Many graph problems are known to be NP-complete (see Sections A.1 and A.2 of [10]). The area of approximation algorithms explores intractable (NP-hard) optimization problems and tries to obtain polynomial-time algorithms that generate feasible solutions that are close to optimal. In this section, a few fundamental NP-hard optimization problems in graphs that can be approximated well are discussed. For more information about approximating these and other problems, see the chapter on approximation algorithms (Chapter 34) and a book on approximation algorithms edited by Hochbaum [16].

In the following discussion $G = (V, E)$ is an arbitrary undirected graph and k is a positive integer.

Vertex cover: a set of vertices $S \subset V$ such that every edge in E is incident to at least one vertex of S. The minimum vertex cover problem is that of computing a vertex cover of minimum cardinality. If the vertices of G have weights associated with them, a minimum-weight vertex cover is a vertex cover of minimum total weight. Using the primal-dual method of linear programming, one can obtain a 2-approximation.

Dominating set: a set of vertices $S \subset V$ such that every vertex in the graph is either in S or adjacent to some vertex in S. There are several versions of this problem such as the total dominating set (every vertex in G must have a neighbor in S, irrespective of whether it is in S or not), the connected dominating set (induced graph of S must be connected) and the independent dominating set (induced graph of S must be empty). The minimum dominating set problem is to compute a minimum cardinality dominating set. The problems can be generalized to the weighted case suitably. All but the independent dominating set problem can be approximated within a factor of $O(\log n)$.

Steiner tree problem: a tree of minimum total weight that connects a set of terminal vertices S in a graph $G = (V, E)$. There is a 2-approximation algorithm for the general problem. There are better algorithms when the graph is defined in Euclidean space. For more information, see [16, 17].

Minimum k-connected graph: a graph G is k-vertex-connected, or simply *k-connected,* if the removal of up to $k - 1$ vertices, along with their incident edges leaves the remaining graph connected. It is *k-edge-connected* if the removal of up to $k - 1$ edges does not disconnect the graph. In the minimum k-connected graph problem, we need to compute a k-connected spanning subgraph of G of minimum cardinality. The problem can be posed in the context of vertex or edge connectivities, and with edge weights. The edge-connectivity problems can be approximated within a factor of 2 from optimal, and a factor smaller than 2 when the edges do not have weights. The unweighted k-vertex-connected subgraph problem can be approximated within a factor of $1 + 2/k$, and the corresponding weighted problem can be approximated within a factor of $O(\log k)$. When the edges satisfy the triangle inequality, the vertex connectivity problem can be approximated within a factor of about $2 + 2/k$. These problems find applications in fault-tolerant network-design. For more information on approximation algorithms for connectivity problems, see [16, Chapter 6].

Degree constrained spanning tree: a spanning tree of maximal degree k. This problem is a generalization of the traveling salesman path problem. There is a polynomial time approximation algorithm that returns a spanning tree of maximal degree at most $k + 1$ if G has a spanning tree of degree k.

Max-cut: a partition of the vertex set into (V_1, V_2) such that the number of edges in $E \cap (V_1 \times V_2)$ (edges between a vertex in V_1 and a vertex in V_2) is maximized. It is easy to compute a partition in which at least half the edges of the graph cross the cut. This is a 2-approximation. Semi-definite programming techniques can be used to derive an approximation algorithm with a performance guarantee of about 1.15.

7.10 Research Issues and Summary

Some of the recent research efforts in graph algorithms have been in the areas of dynamic algorithms, graph layout and drawing, and approximation algorithms. Detailed references about these areas may be found in Chapter 8, Chapter 9, and Chapter 34, respectively. Many of the methods illustrated in our chapter find use in the solution of almost any optimization problem. For approximation algorithms see the edited book by Hochbaum [16], and more information in the area of graph layout and drawing can be found in an annotated bibliography by Di Battista et al. [4].

A recent exciting result of Karger's [19] is a randomized near-linear time algorithm for computing minimum cuts. Finding a deterministic algorithm with similar bounds is a major open problem.

Computing a maximum flow in $O(|E||V|)$ time in general graphs is still open. Several recent algorithms achieve these bounds for certain graph densities. A recent result by Goldberg and Rao [14] breaks this barrier, by paying an extra $\log U$ factor in the running time, when the edge capacities are in the range $[1 \ldots U]$. Finding a maximum matching in time better than $O(|E|\sqrt{|V|})$ or obtaining non trivial lower bounds are open problems.

7.11 Defining Terms

Assignment problem: The problem of finding a matching of maximum (or minimum) weight in an edge-weighted graph.

Augmenting path: A path used to augment (increase) the size of a matching or a flow.

Blocking flow: A flow function in which any directed path from s to t contains a saturated edge.

Blossoms: Odd length cycles that appear during the course of the matching algorithm on general graphs.

Branching: A spanning tree in a rooted graph, such that each vertex has a path to the root (also known as in-branching). An out-branching is a rooted spanning tree in which the root has a path to every vertex in the graph.

Capacity: The maximum amount of flow that is allowed to be sent through an edge or a vertex.

Chromatic index: The minimum number of colors with which the edges of a graph can be colored.

Chromatic number: The minimum number of colors needed to color the vertices of a graph.

Concurrent flow: A multi-commodity flow in which the same fraction of the demand of each commodity is satisfied.

Edge coloring: An assignment of colors to the edges of a graph such that no two edges incident to a common vertex receive the same color.

Integer multi-commodity flow: A multi-commodity flow in which the flow through each edge of each commodity is an integral value. The term is also used to capture the multi-commodity flow problem in which each demand is routed along a single path.

Matching: A subgraph in which every vertex has degree at most one.

Maximum flow: The maximum amount of feasible flow that can be sent from a source vertex to a sink vertex in a given network.

Multi-commodity flow: A network flow problem involving multiple commodities, in which each commodity has an associated demand and source-sink pairs.

Network flow: An assignment of flow values to the edges of a graph that satisfies flow conservation, skew symmetry and capacity constraints.

***s*-*t* cut:** A partitioning of the vertex set into S and T such that $s \in S$ and $t \in T$.

Vertex coloring: An assignment of colors to the vertices of a graph such that no two adjacent vertices receive the same color.

Acknowledgments

Samir Khuller's research was supported by NSF Research Initiation Award CCR-9307462 and NSF CAREER Award CCR-9501355. Balaji Raghavachari's research was supported by NSF Research Initiation Award CCR-9409625.

References

[1] Ahuja, R.K., Magnanti, T.L., and Orlin, J.B., *Network Flows.* Prentice Hall, 1993.

[2] Awerbuch, B. and Leighton, T., A simple local-control approximation algorithm for multicommodity flow. In *Proceedings of the 34th Annual Symposium on Foundations of Computer Science,* 459–468, Palo Alto, CA, 3–5, Nov. 1993.

[3] Awerbuch, B. and Leighton, T., Improved approximation algorithms for the multi-commodity flow problem and local competitive routing in dynamic networks. In *Proceedings of the Twenty-Sixth Annual ACM Symposium on Theory of Computing,* 487–496, Montréal, Québec, Canada, 23–25, May 1994.

[4] Battista, G.Di, Eades, P., Tamassia, R., and Tollis, I.G., Annotated bibliography on graph drawing algorithms. *Computational Geometry: Theory and Applications,* 4, 235–282, 1994.

[5] Cormen, T.H., Leiserson, C.E., and Rivest, R.L., *Introduction to algorithms.* MIT Press, Cambridge, MA, 1989.

[6] Edmonds, J. and Karp, R.M., Theoretical improvements in algorithmic efficiency for network flow problems. *J. Assoc. Comput. Mach.,* 19, 248–264, 1972.

[7] Even, S., *Graph Algorithms.* Computer Science Press, Rockville, MD, 1979.

[8] Ford, L.R. and Fulkerson, D.R., *Flows in Networks.* Princeton University Press, Princeton, NJ, 1962.

[9] Gabow, H.N., Galil, Z., Spencer, T., and Tarjan, R.E., Efficient algorithms for finding minimum spanning trees in undirected and directed graphs. *Combinatorica,* 6, 109–122, 1986.

[10] Garey, M.R. and Johnson, D.S., *Computers and Intractability: a Guide to the Theory of NP-Completeness.* Freeman, San Francisco, CA, 1979.

[11] Gibbons, A.M., *Algorithmic Graph Theory.* Cambridge University Press, New York, 1985.

[12] Goldberg, A.V. and Tarjan, R.E., Finding minimum-cost circulations by canceling negative cycles. *J. Assoc. Comput. Mach.,* 36(4), 873–886, Oct. 1989.

[13] Goldberg, A.V., Tardos, É., and Tarjan, R.E., Network flow algorithms. In *Algorithms and Combinatorics Volume 9: Flows, Paths and VLSI Layout,* B. Korte, L. Lovász, H.J. Prömel, and A. Schrijver, Eds., 101–164, Springer-Verlag, 1990.

[14] Goldberg, A. and Rao, S., Beyond the flow decomposition barrier. In *Proceedings of the 38th Annual Symposium on Foundations of Computer Science,* 2–11, Miami Beach, FL, 20–22, Oct. 1997.

[15] Hao, J. and Orlin, J.B., A faster algorithm for finding the minimum cut in a directed graph. *Journal of Algorithms,* 17(3), 424–446, Nov. 1994.

[16] Hochbaum, D.S., Ed. *Approximation Algorithms for NP-Hard Problems.* PWS Publishing, Boston, MA, 1996.

[17] Hwang, F., Richards, D.S., and Winter, P., *The Steiner Tree Problem.* North-Holland, 1992.

[18] Karger, D.R., Motwani, R., and Sudan, M., Approximate graph coloring by semidefinite programming. In *Proceedings of the 35th Annual Symposium on Foundations of Computer Science,* 2–13, Santa Fe, NM, 20–22, Nov. 1994.

[19] Karger, D.R., Minimum cuts in near-linear time. In *Proceedings of the Twenty-Eighth Annual ACM Symposium on Theory of Computing,* 56–63, Philadelphia, PA, 22–24, May 1996.
[20] Karloff, H., *Linear Programming.* Birkhäuser, Boston, MA, 1991.
[21] Kleinberg, J. M., Single-source unsplittable flow. In *Proceedings of the 37th Annual Symposium on Foundations of Computer Science,* 68–77, Burlington, VT, 14–16, Oct. 1996.
[22] Lawler, E.L., *Combinatorial Optimization: Networks and Matroids.* Holt, Rinehart and Winston, New York, 1976.
[23] Lovász, L. and Plummer, M.D., *Matching Theory.* Elsevier Science Publishers B.V., New York, 1986.
[24] Micali, S. and Vazirani, V.V., An $O(\sqrt{|V|} \cdot |E|)$ algorithm for finding maximum matching in general graphs. In *Proceedings of the 21st Annual Symposium on Foundations of Computer Science,* 17–27, Syracuse, NY, 13–15, Oct. 1980.
[25] Nagamochi, H. and Ibaraki, T., Computing edge-connectivity in multi-graphs and capacitated graphs. *SIAM J. Disc. Math.,* 5, 54–66, 1992.
[26] Papadimitriou, C.H. and Steiglitz, K., *Combinatorial Optimization: Algorithms and Complexity.* Prentice Hall, Englewood Cliffs, NJ, 1982.
[27] Stoer, M., and Wagner, F., A simple min-cut algorithm. *J. Assoc. Comput. Mach.,* 44(4), 585–590, 1997.
[28] Tardos, É., A strongly polynomial minimum cost circulation algorithm. *Combinatorica,* 5, 247–255, 1985.
[29] Tarjan, R.E., *Data Structures and Network Algorithms.* Society for Industrial and Applied Mathematics, Philadelphia, PA, 1983.

Further Information

The area of graph algorithms continues to be a very active field of research. There are several journals and conferences that discuss advances in the field. Here we name a partial list of some of the important meetings: "ACM Symposium on Theory of Computing (STOC)," "IEEE Conference on Foundations of Computer Science (FOCS)," "ACM-SIAM Symposium on Discrete Algorithms (SODA)," "International Colloquium on Automata, Languages and Programming (ICALP)," and the "European Symposium on Algorithms (ESA)." There are many other regional algorithms/theory conferences that carry research papers on graph algorithms. The journals that carry articles on current research in graph algorithms are *Journal of the ACM, SIAM Journal on Computing, SIAM Journal on Discrete Mathematics, Journal of Algorithms, Algorithmica, Journal of Computer and System Sciences, Information and Computation, Information Processing Letters,* and *Theoretical Computer Science.*

To find more details about some of the graph algorithms described in this chapter we refer the reader to the books by Cormen, Leiserson and Rivest [5], Even [7], Gibbons [11], and Tarjan [29]. Ahuja et al. [1] have provided a comprehensive book on the network flow problem. The survey chapter by Goldberg et al. [13] provides an excellent survey of various min-cost flow and generalized flow algorithms. A detailed survey describing various approaches for solving the matching and flow problems can be found in Tarjan's book [29]. Papadimitriou and Steiglitz [26] discuss the solution of many combinatorial optimization problems using a primal-dual framework.

8

Dynamic Graph Algorithms

David Eppstein
University of California at Irvine

Zvi Galil
Columbia University

Giuseppe F. Italiano
University Venezia,
"Ca' Foscari" di Venezia

8.1 Introduction

In many applications of graph algorithms, including communication networks, graphics, assembly planning, and VLSI design, graphs are subject to discrete changes, such as additions or deletions of edges or vertices. In the last decade there has been a growing interest in such dynamically changing graphs, and a whole body of algorithms and data structures for dynamic graphs has been discovered. This chapter is intended as an overview of this field.

In a typical dynamic graph problem one would like to answer queries on graphs that are undergoing a sequence of updates, for instance, insertions and deletions of edges and vertices. The goal of a dynamic graph algorithm is to update efficiently the solution of a problem after dynamic changes, rather than having to recompute it from scratch each time. Given their powerful versatility, it is not surprising that dynamic algorithms and dynamic data structures are often more difficult to design and analyze than their static counterparts.

We can classify dynamic graph problems according to the types of updates allowed. A problem is said to be *fully dynamic* if the update operations include unrestricted insertions and deletions of edges. A problem is called *partially dynamic* if only one type of update, either insertions or deletions, is allowed. If only insertions are allowed, the problem is called *incremental*; if only deletions are allowed it is called *decremental*.

In this chapter, we focus our attention to dynamic algorithms for undirected graphs, which were studied more extensively than dynamic algorithms for directed graphs. Indeed, designing efficient fully dynamic data structures for directed graphs has turned out to be an extremely difficult task. Most of the efficient data structures available for directed graphs are partially dynamic [3, 27, 28, 33, 45], and only preliminary results are available for fully dynamic problems [22]. For this reason, an alternative viewpoint that has been

proposed is to measure the complexity of a dynamic algorithm as a function of the *output change* [15, 36]. The main dynamic problems considered on directed graphs include shortest paths and transitive closure. For lack of space, we do not include in this chapter dynamic algorithms for planar graphs, which have received considerable attention in recent years [4, 5, 9, 10, 11, 18, 19, 26, 30, 38, 40, 41, 42], and focus our attention to general undirected graphs only.

The remainder of the chapter is organized as follows. In Section 8.2 we give some preliminary definitions and a little terminology. Dynamic tree problems are considered in Section 8.3, while in Section 8.4 we describe partially dynamic algorithms for undirected graphs. Fully dynamic algorithms for undirected graphs are described in Section 8.5. Finally, in Section 8.6 we describe some open problems.

8.2 Preliminary Definitions

In this section, we recall some basic graph-theoretic terminology from Chapter 6. Given an undirected graph G, the *minimum spanning forest of G* is the subgraph of minimum total weight that has the same connected components as the original graph. Whenever G is connected, this forest consists of a unique tree, and we refer to this tree as a minimum spanning tree of G. Note that a minimum spanning forest, or a minimum spanning tree is not necessarily unique. It is well known that a minimum spanning forest can be computed by either of two dual greedy algorithms, based on the following properties:

> *Cut Property:* Add edges one at a time to the spanning forest until it spans the graph. At each step, find a cut in the graph that contains no edges of the current forest, and add the edge with lowest weight crossing the cut.
>
> *Cycle Property:* Remove edges one at a time from the graph until only a forest is left. At each step, find a cycle in the remaining graph and remove the edge with the highest weight in the cycle.

Given an undirected graph $G = (V, E)$, and an integer $k \geq 2$, a pair of vertices $\langle u, v \rangle$ is said to be *k-edge-connected* if the removal of any $(k-1)$ edges in G leaves u and v connected. This is an equivalence relationship, and we denote it by $\equiv_k$, i.e., if a pair of vertices $\langle x, y \rangle$ is k-edge-connected we write $x \equiv_k y$. The vertices of a graph G are partitioned by this relationship into equivalence classes called *k-edge-connected components.* G is said to be *k-edge-connected* if the removal of any $(k-1)$ edges leaves G connected. As a result of these definitions, G is k-edge-connected if and only any two vertices of G are k-edge-connected. An edge set $E' \subseteq E$ is an *edge-cut for vertices x and y* if the removal of all the edges in E' disconnects G into two graphs, one containing x and the other containing y. An edge set $E' \subseteq E$ is an *edge-cut for G* if the removal of all the edges in E' disconnects G into two graphs. An edge-cut E' for G [for x and y, respectively] is *minimal* if removing any edge from E' reconnects G [x and y, respectively]. The cardinality of an edge-cut E', denoted by $|E'|$, is given by the number of edges in E'. An edge-cut E' for G [for x and y, respectively] is said to be a *minimum cardinality edge-cut* or in short a *connectivity edge-cut* if there is no other edge-cut E'' for G [for x and y, respectively] such that $|E''| < |E'|$. Connectivity edge-cuts are of course minimal edge-cuts. Note that G is k-edge-connected if and only if a connectivity edge-cut for G contains at least k edges, and $x \equiv_k y$ if and only if a connectivity edge-cut for x and y contains at least k edges. A connectivity edge-cut of cardinality 1 is called a *bridge.*

8.3 Dynamic Problems on Trees

This section presents data structures that maintain properties of a dynamically changing forest of trees. The basic operations that we will consider are edge insertions and edge deletions. Many properties of dynamically changing trees have been considered in the literature. The basic property is tree membership: namely, while the forest of trees is dynamically changing, we would like to know at any time which tree

contains a given vertex, or whether two vertices are in the same tree. Dynamic tree membership is a special case of dynamic connectivity in undirected graphs, and indeed we will later use some of the data structures developed here for trees to solve the more general problem on graphs. The partially dynamic tree membership problem can be solved by means of the set union data structures described in Chapter 5. We will thus, focus on the fully dynamic tree membership problem. Other properties that have been considered are finding the parent of a vertex, or finding the least common ancestor of two vertices [39]. When costs are associated either to vertices or to edges, one could also ask what is the minimum (or maximum) cost in a given path.

The fully dynamic tree membership problem consists of maintaining a forest of unrooted trees under insertion of edges (which merge two trees into one), deletion of edges (which split one tree into two), and membership queries. Typical queries return the name of the tree containing a given vertex, or ask whether two vertices are in a same tree. Most of the solutions presented here will root each tree arbitrarily at one of its vertices; by keeping extra information at the root (such as the name of the tree), membership queries are equivalent to finding the root of the tree containing the given vertex.

There are two fully dynamic data structures for this problem: the dynamic trees of Sleator and Tarjan [39], and the **topology trees** of Frederickson [12]. Both data structures follow the common idea of partitioning a tree into a set of vertex-disjoint paths. However, they are very different in how this partition is chosen, and in the data structures they use to represent the paths inside the partition. Indeed, Sleator and Tarjan [39] use a simple partition of the trees based upon a careful choice of sophisticated data structures to represent paths. On the contrary, Frederickson [12] uses a more sophisticated partition that is based upon the topology of the tree; this implies more complicated algorithms but simpler data structures for representing paths.

The dynamic trees of Sleator and Tarjan [39] are able to maintain a collection of rooted trees, each of whose vertices has a real-valued cost, under an arbitrary sequence of the following operations:

maketree(v)*:* initialize a new tree consisting of single vertex v with cost zero.

findroot(v)*:* return the root of the tree containing vertex v.

findcost(v)*:* return a vertex of minimum cost in the path from v to findroot(v).

addcost(v, δ)*:* add the real number δ to the cost of every vertex in the path from v to findroot(v).

link(v, w)*:* Merge the trees containing vertices v and w by inserting edge (v, w). This operation assumes that v and w are in different trees and that v is a tree root.

cut(v)*:* Delete the edge leaving v, thus, splitting into two the tree containing vertex v. This operation assumes that v is not a tree root.

evert(v)*:* Make v the root of its tree.

THEOREM 8.1 **[39]** *Each of the above operations can be supported in* $O(\log n)$ *worst-case time.*

We do not give the details of the method here, and refer the interested reader to reference [39]. Rather than the dynamic trees of Sleator and Tarjan, we describe in more detail the topology trees of Frederickson. The reason for this choice is that the topology trees are often used as building blocks by many dynamic graph algorithms.

Topology Trees

In this section we consider trees with maximum vertex degree 3. This is without loss of generality, as we can convert any tree T to a tree T' with maximum vertex degree 3 by means of a standard transformation [20], which will be defined in a more general sense for graphs in "Clustering and Topology Trees." Let T be a tree of maximum degree 3 to be maintained dynamically. Before defining formally a topology tree, we need a little terminology. A *vertex cluster* with respect to T is a set of vertices that induces a connected

subgraph on T. The *cardinality* of a cluster is the number of vertices in it. An edge is *incident* to a cluster if exactly one of its endpoints is inside the cluster. Two clusters are *adjacent* if there is a tree edge that is incident to both. A *boundary vertex* of a cluster is a vertex that is adjacent in T to some vertex not in the cluster. The *external degree* of a cluster is the number of tree edges incident to it. A *restricted partition* of T is a partition of its vertex set V into vertex clusters such that

(1) Each cluster of external degree 3 is of cardinality 1.

(2) Each cluster of external degree less than 3 is of cardinality at most 2.

(3) No two adjacent clusters can be combined and still satisfy the above.

There can be several restricted partitions for a given tree T, based upon different choices of the vertices to be unioned. Because of (3), the restricted partition implements a cluster-forming scheme according to a locally greedy heuristic, which does not always obtain the minimum number of clusters. However, this greedy method has the advantage of requiring only local adjustments during updates. Thus, although not optimal from the viewpoint of number of clusters generated, this partition is well suited for dynamic operations. To illustrate this point clearly, we sketch how to update the clusters of a restricted partition when the underlying tree is subject to updates.

Assume we want to delete an edge e from T. First, removing e splits T into two trees, say T_1 and T_2. T_1 and T_2 inherit all of the clusters of T, possibly with the following exceptions. If e is entirely contained in a cluster, this cluster is no longer connected and therefore must be split. After the split, we must check whether each of the two resulting clusters is adjacent to a cluster of tree degree at most 2, and if these two adjacent clusters together have at most 2 vertices. If so, we combine these two clusters in order to maintain condition (3) above. If e is between two clusters, then no split is needed. However, since the tree degree of the clusters containing the endpoints of e has been decreased, we must check if each cluster should be combined with an adjacent cluster, again because of condition (3). This completes the updates needed for the deletion of e.

Next, we show how to combine two trees T_1 and T_2 into one tree T by adding an edge f. Again, T inherits the clusters of T_1 and T_2 with the only following exceptions. If f increases the tree degree of a cluster from 1 to 2, in order to preserve condition (3) we must again check whether this cluster must be combined with the cluster newly adjacent to it. If f increases the tree degree of a cluster containing more than one vertex from 2 to 3, condition (1) is violated and we have to split the cluster. For each cluster formed after the split, we check whether it must be combined with an adjacent cluster.

A *restricted multilevel partition* consists of a collection of restricted partitions of V satisfying the following:

(1) The clusters at level 0 (known as *basic clusters*) contain one vertex each.

(2) The clusters at level $\ell \geq 1$ form a restricted partition with respect to the tree obtained after shrinking all the clusters at level $\ell - 1$.

(3) There is exactly one vertex cluster at the topmost level.

From this definition, it follows that any cluster at level $\ell \geq 1$ is either (a) the union of two adjacent clusters of level $(\ell - 1)$ such that the external degree of one cluster is 1 or the external degree of both is 2, or (b) one cluster of level $(\ell - 1)$, if rule (a) does not apply. Thus, any cluster obtained by unioning two clusters at a lower level, has always tree degree at most 2. This implies that at any level a cluster of tree degree 3 consists always of a single vertex. Once again, several multilevel partitions are possible. But each restricted multilevel partition has the nice property of having only logarithmic depth, as implied by the following lemma of Frederickson [13].

LEMMA 8.1 **[13]** For any $\ell \geq 0$, the number of clusters at level $\ell + 1$ is at most 5/6 times the number of clusters at level ℓ.

The *topology tree* is a hierarchical representation of T. Each level of the topology tree partitions the vertices of T into connected subsets called *clusters*. More precisely, given a restricted multilevel partition for T, a *topology tree* for T is a tree satisfying the following:

(1) A topology tree node at level ℓ represents a vertex cluster at level ℓ in the restricted multilevel partition.
(2) A node at level $\ell \geq 1$ has at most two children, representing the vertex clusters at level $\ell - 1$ whose union gives the vertex cluster the node represents.

Since by Lemma 8.1 a restricted multilevel partition reduces the number of clusters at each level by a constant fraction, the height of the topology tree is $O(\log n)$. We now sketch how to update a topology tree during edge insertions or deletions. If there are updates in the spanning tree T, the restricted multilevel partition of T may be required to change. The changes in the topology tree are caused by the changes in the restricted multilevel partition it represents: at each level of the topology tree, we apply few locally greedy adjustments similar to the ones described before for one-level restricted partitions. As shown in [13], the topology tree update actually consists of two subtasks. First, a constant number of basic clusters (corresponding to leaves in the topology tree) have to be examined and possibly updated. Second, the changes in these basic clusters percolate up in the topology tree, possibly causing vertex clusters in the multilevel partition to be regrouped in different ways. This is handled by rebuilding portions of the topology tree in a bottom-up fashion, and involves a constant amount of work to be done on at most $O(\log n)$ topology tree nodes.

LEMMA 8.2 **[12]** The update of a topology tree because of an edge insertion or deletion can be supported in time $O(\log n)$.

8.4 Partially Dynamic Problems on Undirected Graphs

For undirected graphs, most of the **partially dynamic problems** considered are incremental, namely they support edge insertions only. We first show how to maintain a minimum spanning forest when the underlying graph G is subject to insertions of edges. We represent each tree in the minimum spanning forest as a dynamic tree (as defined in Section 8.3). Let $e = (x, y)$ be the edge to be inserted, and let F be a minimum spanning forest of G before inserting e. If the endpoints of e are in two different connected components of G, then the new forest F' will be $F \cup \{e\}$. Note that all this can be done in $O(\log n)$ worst-case time with the dynamic trees: we find the root of x, the root of y, and then link the two trees. Otherwise, x and y are in the same tree of the forest F, and inserting e in F introduces one cycle λ. To compute the new spanning forest F' we apply the cycle property in λ as follows. Let T be the tree of F containing both x and y: perform an evert on T and root it at x. Next, find the maximum cost edge f in the path from y to the root. If the cost of e is greater than the cost of f, then $F' = F$. Otherwise, swap e and f. Note that again this implies a constant number of operations to be executed on a forest of dynamic trees and therefore can be accomplished in $O(\log n)$ worst-case time.

THEOREM 8.2 *A minimum spanning forest of a graph G subject to edge insertions only can be maintained in $O(\log n)$ worst-case time per operation, where n is the number of vertices in G.*

The previous theorem gives implicitly an incremental algorithm for maintaining the connected components of a graph, since a minimum spanning forest of G is trivially a spanning forest of G (we assume that each edge has cost 1). A query on whether two vertices x and y are in a same connected component can be answered in $O(\log n)$ time by finding and comparing the roots of x and y. A better bound can

be achieved by representing the trees in the forest using the set union data structures of Chapter 5. A query on whether two vertices x and y are connected can be answered by checking whether *find*(x) equals *find*(y). When inserting edge $e = (x, y)$, we first perform $A \leftarrow$ *find*(x) and $B \leftarrow$ *find*(y). If $A = B$, x and y were already connected and the insertion of e does not change the connected components of G. If $A \neq B$ then A and B have to be merged into a new connected component: we do this by executing a *union*(A, B). Since each operation can be implemented by a constant number of set union operations, we have the following theorem.

THEOREM 8.3 *The connected components a graph G subject to edge insertions only can be maintained in $O(\alpha(q, n))$ amortized time per query or update operation, where q is the total number of queries and n is the number of vertices in G.*

Set union data structures can be used for the partially dynamic maintenance of other graph properties, such as bipartiteness, and edge and vertex connectivity. For lack of space, we describe here only how to solve the partially dynamic 2-edge connectivity problem. This problem consists of maintaining a graph G under an intermixed sequence of operations of the following kinds.

> *Same2EdgeBlock*(u, v)*:* Return *true* if vertices u and v are in the same 2-edge-connected component. Return *false* otherwise.
>
> *InsertEdge*(x, y)*:* Insert a new edge between the two vertices x and y.

Westbrook and Tarjan [44] presented one algorithm that runs in a total of $O(q\alpha(q, n))$ time, where q is the total number of *Same2EdgeBlock* and *InsertEdge* operations, and n is the number of vertices. For sake of simplicity, we describe in details only the algorithm that operates on connected graphs, and refer the interested reader to [44] for the full details of the method on general unconnected graphs.

For any vertex x in G denote by $C_{2E}(x)$ the 2-edge-connected component of G containing x. The main data structure maintained by the algorithm is the bridge-block tree of G, which is defined as follows. Nodes in this tree correspond to 2-edge-connected components of G, and any two nodes are connected by an edge if and only if there is a bridge connecting the corresponding 2-edge-connected components. We assume that the bridge-block tree is rooted arbitrarily at one node, and we denote by *parent*(σ) the parent of a tree node σ in the bridge-block tree.

Besides maintaining the bridge-block tree of G, we maintain the actual 2-edge-connected components of G as disjoint sets, subject to *union* and *find* operations. For this, we use any of the fast set-union data structures described in Chapter 5, which are able to process any sequence of q *union* and *find* operations on a collection of n elements in $O(q\alpha(q, n))$ worst-case time.

We define the name of each disjoint set in the set-union data structure as a pointer to the tree node associated to the corresponding 2-edge-connected component. Thus, we can assume that for each vertex x in G, *find*$(x) = C_{2E}(x)$. With this data structure, a *Same2EdgeBlock*(x, y) can be simply performed by checking whether *find*(x)=*find*(y). The *union* operations will be used to update efficiently the 2-edge-connected components during *InsertEdge* operations. The effect of an *InsertEdge*(u, v) on the bridge-block tree depends on whether u and v are in the same 2-edge-connected component or in different 2-edge-connected components. Let $C_{2E}(u)$ and $C_{2E}(v)$ be the 2-edge-connected components containing u and v, respectively, before inserting the edge (u, v).

We now describe the changes caused in the bridge-block tree by the insertion of edge (u, v). If $C_{2E}(u) = C_{2E}(v)$, the bridge-block tree is unaffected by the new edge and hence, no change is needed in the data structure. If $C_{2E}(u) \neq C_{2E}(v)$, the inserted edge creates a new cycle that reduces the number of 2-edge-connected components and bridges of G. This cycle consists of (u, v) plus all the edges in the tree path between node $C_{2E}(u)$ and node $C_{2E}(v)$. All the nodes in this tree path must be replaced by a single node, and every node previously adjacent to one of the nodes in the tree path becomes adjacent to the new single node. This process is called *path condensation.*

To implement *InsertEdge* operations, the bridge-block tree is maintained as a data structure that supports the following primitives:

> *FindPath*(σ_1, σ_2)*:* Given two tree nodes σ_1 and σ_2, return the tree path between σ_1 and σ_2. If $\sigma_1 = \sigma_2 = \sigma$, return σ.
>
> *CondensePath*(π)*:* Perform path condensation on the tree path π, unioning the corresponding disjoint sets associated with the encountered tree nodes. Return the modified bridge-block tree. Note that a *CondensePath* does nothing if π is an empty path consisting of a single node.

A *FindPath*(σ_1, σ_2) can be implemented as follows: we proceed from u and from v toward the tree root, alternating one step at the time, until the paths traced from u and from v intersect at their nearest common ancestor. Note that this returns the nearest common ancestor of u and v, and the tree path π between u and v. The number of steps required to return π is at most $2|\pi|$. Since they are performed by following parent pointers, we call these *parent steps.* The path π is returned as a list of nodes (not in order along the path), with the nearest common ancestor at the end. Let $\widetilde{\sigma}$ be the parent of the nearest common ancestor. To perform a *CondensePath*(π), π is condensed into a single node σ, which is made child of $\widetilde{\sigma}$. All the disjoint sets associated with nodes in π are unioned. Since we perform a union at each step, we call these *union steps.*

LEMMA 8.3 In any sequence of operations, there are at most $O(n)$ parent and union steps.

PROOF Suppose a path π is being condensed. To generate π, $O(|\pi|)$ parent steps are required, and $|\pi| - 1$ nodes are condensed. Since after $(n-1)$ condensations the graph becomes 2-edge-connected, there can be at most $O(n)$ parent steps and $(n-1)$ union steps.

With the help of these primitives, we support an *InsertEdge*(u, v) as follows:

```
InsertEdge(u, v) begin
    return CondensePath(FindPath(find(u), find(v)));
end
```

THEOREM 8.4 **[44]** *Given an initially connected graph $G_0 = (V_0, E_0)$ and $O(|E_0|)$ preprocessing time, a sequence of q* InsertEdge *and* Same2EdgeBlock *operations can be processed in $O(q\alpha(q, n))$ time.*

PROOF The 2-edge-connected components and the bridge-block tree of G_0 can be found in $O(|E_0|)$ using the algorithm of Tarjan [43]. By Lemma 8.3, the total number of parent and union steps is $O(n)$, giving a total of $O(q)$ finds and $O(n)$ unions in the set union data structure.

To complete this section, we mention that most of the incremental algorithms proposed in the literature for edge and vertex connectivity (see, e.g.,[5, 6, 17, 29, 31, 32, 44]) follow the approach outlined here for 2-edge connectivity. Namely, connectivity queries are answered by maintaining a tree that reflects the structure of the connectivity cuts of the graph. When a new edge (u, v) is added to the graph, some connectivity cuts may be invalidated; these connectivity cuts can be easily found since they all lie in a tree path, whose endpoints correspond to u and v. After locating the path, some form of path condensation takes place. To support efficiently path condensation, set-union based data structures are used. The tree structure of the $(k-1)$-cuts is fairly complicated, however, as k increases, and it requires sophisticated data structures and a quite delicate analysis.

8.5 Fully Dynamic Problems on Undirected Graphs

This section describes fully dynamic algorithms for undirected graphs. These algorithms maintain efficiently some property of a graph that is undergoing structural changes defined by insertion and deletion of edges, and/or edge cost updates. For instance, the fully dynamic minimum spanning tree problem consists of maintaining a minimum spanning forest of a graph during the above operations. The typical updates for a fully dynamic problem will therefore be inserting a new edge, and deleting an existing edge. To check the graph property throughout a sequence of these updates, the algorithms must prepared to answer queries on the graph property. Thus, a fully dynamic connectivity algorithm must be able to insert edges, delete edges, and answer a query on whether the graph is connected, or whether two vertices are connected. Similarly, a fully dynamic k-edge connectivity algorithm must be able to insert edges, delete edges, and answer a query on whether the graph is k-edge-connected, or whether two vertices are k-edge-connected. The goal of a dynamic algorithm is to minimize the amount of recomputation required after each update. All of the fully dynamic algorithms that we describe in this section are able to dynamically maintain the graph property at a cost (per update operation) that is significantly smaller than the cost of recomputing the graph property from scratch. Many of the algorithms proposed in the literature use the same general techniques, and so we begin by describing these techniques. All of these techniques use some form of graph decomposition, and partition either the vertices or the edges of the graph to be maintained.

The first technique we present is the *clustering* technique of Frederickson [12], which is based upon partitioning the graph into a suitable collection of connected subgraphs called *clusters,* such that each update involves only a small number of such clusters. The dynamic trees of Sleator and Tarjan [39] can be used to maintain information about the edges of a tree; clusters dually keep track of the edges that are not part of some given spanning tree, by grouping them according to which clusters they connect. Typically, this decomposition is applied recursively and the information about the subgraphs is combined with the topology trees described in Section 8.3. A refinement of the clustering technique appears in the idea of *ambivalent data structures* [13], in which edges can belong to multiple groups, only one of which is actually selected depending on the topology of the given spanning tree.

Another technique we describe is **sparsification** by Eppstein et al. [8]. This is a divide-and-conquer technique that can be used to reduce the dependence on the number of edges in a graph, so that the time bounds for maintaining some property of the graph match the times for computing in sparse graphs. Roughly speaking, sparsification works as follows. Let $\mathcal{A}$ be an algorithm that maintains some property of a dynamic graph G with m edges and n vertices in time $T(n, m)$. Sparsification maintains a proper decomposition of G into small subgraphs, with $O(n)$ edges each. In this decomposition, each update involves applying algorithm $\mathcal{A}$ to few small subgraphs of G, resulting into an improved $T(n, n)$ time bound per update. Thus, throughout a sequence of operations, sparsification makes a graph looks sparse (i.e., with only $O(n)$ edges): hence, the reason for its name. Sparsification works on top of a given algorithm, and need not to know the internal details of this algorithm. Consequently, it can be applied orthogonally to other data structuring techniques; we will actually see a number of situations in which both clustering and sparsification can be combined to produce an efficient dynamic graph algorithm.

The previous two techniques allows one to design efficient deterministic algorithms. The last technique we present in this section is due to Henzinger and King [21], and it is a combination of a suitable graph decomposition and randomization. We now sketch how this decomposition is defined. Let G be a graph whose spanning forest has to be maintained dynamically. The edges of G are partitioned into $O(\log n)$ levels: the lower levels contain tightly-connected portions of G (i.e., dense edge cuts), while the higher levels contain loosely-connected portions of G (i.e., sparse cuts). For each level i, a spanning forest for the graph defined by all the edges in levels i or below is maintained. If a spanning forest edge e is deleted at some level i, random sampling is used to quickly find a replacement for e at that level. If random sampling succeeds, the forest is reconnected at level i. If random sampling fails, the edges that can replace e in level i form with high probability a sparse cut. These edges are moved to level $(i + 1)$ and the same procedure is applied recursively on level $(i + 1)$.

One particular dynamic graph problem that has been thoroughly investigated is the maintenance of a minimum spanning forest. This is an important problem on its own, but it has also impact on other problems as well. Indeed the data structures and techniques developed for dynamic minimum spanning forests have found applications also in other areas, such as dynamic edge and vertex connectivity [8, 13, 16, 24, 37, 38]. Thus, we will focus our attention to the fully dynamic maintenance of minimum spanning trees.

Clustering and Topology Trees

Let $G = (V, E)$ be a graph, with a designated spanning tree S. *Clustering* is a method of partitioning the vertex set V, into connected subtrees in S, so that each subtree is only adjacent to a few other subtrees. *Topology trees* are a representation of the tree S using a different tree with logarithmic height, formed by recursive clustering, and are a generalization of the topology trees described in Section 8.3. *2-dimensional topology trees* are formed from pairs of nodes in a topology tree, and allow us to maintain information about the edges in $E - S$. Fully dynamic algorithms based only on a single level of clustering obtain typically time bounds of $O(m^{2/3})$ (see for instance [16, 37]). When the partition can be applied recursively, better $O(m^{1/2})$ time bounds can be achieved with the help of topology trees (see for instance [12, 13]). Along with their applications to dynamic graph algorithms, topology trees can be used in situations in which a dynamic tree is part of a static graph; for instance, topology trees can be used to speed up the execution of pivots in the network simplex algorithm for minimum cost circulations [7].

Before defining clustering and topology trees, we describe a standard graph transformation that we use throughout. We convert the graph G into a graph with maximum vertex degree 3 [20]: Suppose $v \in V$ has degree $d(v) > 3$, and is adjacent to vertices $u_1, u_2, \ldots, u_d$. In the transformed graph, v is replaced by a cycle of d *dashed* edges: namely, we substitute v by d vertices $v_1, v_2, \ldots, v_d$. For each edge (v, u) of the original graph, in position i among the list of edges adjacent to v and position j among the edges adjacent to u, we create an *actual* edge (v_i, u_j). We also create *dashed* edges (v_i, v_{i+1}) for $1 \leq i \leq d - 1$, and a *dashed* edge (v_d, v_1) to close the loop. We call these the *dashed edges of* v. As a result of this transformation, the graph keeps its actual edges, and has an additional $O(m)$ dashed edges. Note that this transformation affects the problems we would like to solve. However for most of the problems to which this technique is applied (such as minimum spanning forests, connectivity and 2-edge connectivity), either the two solutions are identical (such as in the case of connectivity and 2-edge connectivity), or we can easily compute the solution in the original graph once we know the solution in the transformed graph (such as in the case of minimum spanning forests). In other cases, although this computation is not easy (such in the case of biconnectivity), it can be handled with some little extra effort.

Throughout the sequence of updates, any cluster-based dynamic graph algorithm maintains a spanning tree T of G. For each vertex v, T contains all of the dashed edges of v except one. The only one dashed edge of v that does not belong to T can be arbitrarily chosen. We now generalize the notion of restricted partition given in Section 8.3. Let $z > 0$ be an integer, to be fixed later on. A *restricted partition of order* z of G is a partition of its vertex set V into $O(m/z)$ vertex clusters such that:

(1) Each set in the partition yields a vertex cluster of external degree at most 3.

(2) Each cluster of external degree 3 is of cardinality 1.

(3) Each cluster of external degree less than 3 is of cardinality less than or equal to z.

(4) No two adjacent clusters can be combined and still satisfy the above.

Note that since any cluster has maximum external degree 3, it can have at most three boundary vertices. A restricted partition of order z can be found in linear time as shown in [13]. We now discuss how to update the clusters of a restricted partition of order z when the underlying graph is subject to updates. The basic update we will consider is a *swap*: a *swap* (e, f) in a spanning tree T replaces a tree edge e by a non-tree edge f, yielding a new spanning tree. This is a basic update operation, since each edge insertion,

edge deletion, or edge cost change causes at most one swap in a spanning tree: at most one edge is added to the tree and one edge removed. When a swap (e, f) is performed, we do the following. First, removing e splits T into two trees, say T_1 and T_2. T_1 and T_2 inherit all of the clusters of T, possibly with the following exceptions. If e is entirely contained in a cluster, this cluster is no longer connected and therefore must be split. After the split, we must check whether each of the two resulting clusters can be merged with neighboring clusters in order to maintain condition (4) above. If e is between two clusters, then no split is needed. However, since the tree degree of the clusters containing the endpoints of e has been decreased, we must check if each cluster should be combined with an adjacent cluster, again because of condition (4). This completes the updates needed for the deletion of e.

Next, we show how to combine T_1 and T_2 into one tree T by adding edge f. Again, T inherits the clusters of T_1 and T_2 with the only following exceptions. If f increases the tree degree of a cluster from 1 to 2, in order to preserve condition (4) we must again check whether this cluster must be combined with the cluster newly adjacent to it. If f increases the tree degree of a cluster containing more than one vertex from 2 to 3, condition (2) is violated and we do the following. Let w', w'', and w''' be the endpoints of the three tree edges incident on this cluster. Let x be the common vertex on tree paths between w', w'', and w'''. Make x into a cluster (of tree degree 3) by itself, and take the remaining parts of the old cluster as new clusters. For each cluster so formed, we check whether it must be combined with an adjacent cluster.

LEMMA 8.4 The time required to update a restricted partition of order z because of a swap is $O(z)$.

PROOF As described before, at most a constant number of vertex clusters is changed, deleted or created during a swap. Each cluster that is modified in some way, has at most z vertices and edges, and therefore can be handled in time $O(z)$.

The notion of *restricted multilevel partition* of "Topology Trees" can be generalized as follows:

(1) The clusters at level 0 (known as *basic clusters*) form a restricted partition of order z.
(2) The clusters at level $\ell \geq 1$ form a restricted partition of order 2 with respect to the tree obtained after shrinking all the clusters at level $\ell - 1$.
(3) There is exactly one vertex cluster at the topmost level.

We now list some properties of a multilevel restricted partition. First, any basic vertex cluster of tree degree 3 consists of a single vertex. Second, any cluster obtained by unioning two clusters at a lower level, has always tree degree at most 2. These two properties imply that any nonbasic cluster of tree degree 3 also consists of a single vertex, and all of its incident edges will be tree edges. Furthermore, there are no non-tree edges having an endpoint in a cluster of tree degree 3. Finally, the restricted multilevel partition has the nice property of having only logarithmic depth. Indeed Frederickson [13] shows that each level of the topology tree has a number of nodes which is a constant fraction of the previous level, from which the following lemma follows.

LEMMA 8.5 **[12, 13]** The number of levels in a restricted multilevel partition is $\Theta(\log n)$.

The *topology tree* is a hierarchical representation of G based on T. Each level of the topology tree partitions the vertices of G into connected subsets called *clusters.* More precisely, given a restricted multilevel partition for T, a *topology tree* for T is a tree satisfying the following:

(1) A topology tree node at level ℓ represents a vertex cluster at level ℓ in the restricted multilevel partition.

(2) A node at level $\ell \geq 1$ has at most two children, representing the vertex clusters at level $\ell - 1$ whose union gives the vertex cluster the node represents.

As shown in [13], the update of a topology tree because of an edge swap in T consists of two subtasks. First, a constant number of basic clusters (corresponding to leaves in the topology tree) have to be examined and possibly updated. As shown in Lemma 8.4, this can be supported in $O(z)$ time. Second, the changes in these basic clusters percolate up in the topology tree, possibly causing vertex clusters in the multilevel partition to be regrouped in different ways. This is handled by rebuilding portions of the topology tree in a bottom-up fashion, and involves a constant amount of work to be done on at most $O(\log n)$ topology tree nodes. This yields the following lemma.

LEMMA 8.6 **[12, 13]** The update of a topology tree because of an edge swap can be supported in time $O(z + \log n)$.

We now give the definition of *2-dimensional topology tree.* This is somehow involved, and we only sketch it here, referring the reader to [12, 13] for the full details. For every pair of nodes V_α and V_β at the same level in the topology tree there is a node labeled $V_\alpha \times V_\beta$ in the 2-dimensional topology tree. Let E_T be the tree edges of G (i.e., the edges in the spanning tree T): node $V_\alpha \times V_\beta$ represents all the non-tree edges of G (i.e., the edges of $E - E_T$) having one endpoint in V_α and the other in V_β. The root of the 2-dimensional topology tree is labeled $V \times V$ and represents all the non-tree edges of G. If a node is labeled $V_\alpha \times V_\beta$, and V_α has children V_{α_i}, $1 \leq i \leq p$, and V_β has children V_{β_j}, $1 \leq j \leq q$, in the topology tree, then $V_\alpha \times V_\beta$ has children $V_{\alpha_i} \times V_{\beta_j}$, $1 \leq i \leq p$, $1 \leq j \leq q$, in the 2-dimensional topology tree.

Note that a 2-dimensional topology tree corresponds roughly to having $O(m/z)$ topology trees, one for each basic cluster in the restricted multilevel partition. As previously described, updating the basic clusters because of an edge swap would require a total of $O(z)$ time, and then updating these $O(m/z)$ topology trees would require a total of $O((m/z)\log n)$ time. This yields a total of $O(z + (m/z)\log n)$ time. The computational saving of a 2-dimensional topology tree is that it can be updated during a swap in its corresponding topology tree in $O(m/z)$ time only [13]. The crucial point of this analysis is that only $O(m/z)$ nodes in the 2-dimensional topology tree need to be looked at and eventually updated during a swap, and this can be done in constant time per node.

LEMMA 8.7 **[12, 13]** The update of a 2-dimensional topology tree because of an edge swap in the corresponding topology tree can be supported in time $O(m/z)$.

Note that the bound in Lemma 8.7 does not take into account the time needed to update the topology tree upon which the 2-dimensional topology tree is based. If this is taken into account, the total time required to perform a swap becomes $O(z + (m/z))$. Typical algorithms will balance this bound by choosing $z = \Theta(m^{1/2})$ to get an $O(m^{1/2})$ total time bound, as shown in the following theorem.

THEOREM 8.5 **[12]** *The minimum spanning forest of an undirected graph can be maintained in time $O(m^{1/2})$ per update, where m is the current number of edges in the graph.*

PROOF We maintain a restricted multilevel partition of order z, and the corresponding topology tree and 2-dimensional topology tree as described before. We augment the 2-dimensional topology tree as follows. Each leaf $V_i \times V_j$ of the 2-dimensional topology tree stores the set $E_{i,j}$ of edges having one endpoint in V_i and the other in V_j, as well as the minimum cost edge in this set. This information is stored in a heap-like fashion: internal nodes of the 2-dimensional topology tree have the minimum of the values of their children. This additional information required constant time per node to be maintained.

Consequently, as in Lemma 8.7 the update of this augmented 2-dimensional topology tree because of a swap can be done in $O(m/z)$ time.

Whenever a new edge is inserted or a non-tree edge has its cost decreased, we can find a replacement edge in time $O(\log n)$ with the dynamic trees of Sleator on Tarjan, as described in Section 8.4. Whenever an edge is deleted, or a tree edge has its cost increased, we can find a replacement edge as follows. Let e be the edge that has been deleted or increased. We first split the 2-dimensional topology tree at e in $O(z+m/z)$ time by Lemma 8.7. Suppose this splits the corresponding topology tree into two trees, whose roots are the clusters V_α and V_β, with V_β having no fewer levels than V_α. To find a possible replacement edge for e, we examine the values at the nodes $V_\alpha \times V_\gamma$ for all possible V_γ in the 2-dimensional topology tree, and take the minimum. It takes $O(m/z)$ time to find and examine these nodes.

This yields a total of $O(z+(m/z))$ time for each update. Choosing $z = \lceil m^{1/2} \rceil$ gives an $O(m^{1/2})$ bound. However, m is changing because of insertions and deletions. When the value of z changes because of m, there will be at least $m^{1/2}$ update before z advances to the next value up or down in the same directions. Since there are at most $O(m/z)$ basic clusters that need to be adjusted, we can adjust a constant number of clusters during each update.

Sparsification

The techniques described in the previous section allow us to obtain an $O(m^{1/2})$ time bound for the fully dynamic maintenance of a minimum spanning forest, connectivity and 2-edge connectivity [12, 13]. The type of clustering used is very problem-dependent, however, and makes these techniques unsuitable to be used as a black box. Namely, whenever we want to apply such techniques to solve a certain problem, we must devise a proper partition of the graph into clusters. Furthermore, we must get into the low-level details of the data structures employed. For instance, we have to make sure that the problem can be represented as a small set of alternatives at each node of the 2-dimensional topology tree, and show how we can select efficiently among those alternatives, or select the information we are interested more directly from the topology tree.

In this section we describe a general technique for designing dynamic graph algorithms, due to Eppstein et al. [8], which is called *sparsification.* This technique can be used to speed up many fully dynamic graph algorithms. Roughly speaking, when the technique is applicable it speeds up a $T(n, m)$ time bound for a graph with n vertices and m edges to $T(n, O(n))$; i.e., to the time needed if the graph were sparse. For instance if $T(n, m) = O(m^{1/2})$, we get a better bound of $O(n^{1/2})$. Sparsification applies to a wide variety of dynamic graph problems, including minimum spanning forests, edge and vertex connectivity. Moreover, it is a general technique and can be used as a black box (without having to know the internal details) in order to dynamize graph algorithms.

The technique itself is quite simple. Let G be a graph with m edges and n vertices. We partition the edges of G into a collection of $O(m/n)$ sparse subgraphs, i.e., subgraphs with n vertices and $O(n)$ edges. The information relevant for each subgraph can be summarized in an even sparser subgraph, which is called a *sparse certificate.* We merge **certificates** in pairs, producing larger subgraphs that are made sparse by again computing their certificate. The result is a balanced binary tree in which each node is represented by a sparse certificate. Each update involves $\log(m/n)$[1] graphs with $O(n)$ edges each, instead of one graph with m edges. With some extra care, the $O(\log(m/n))$ overhead term can be eliminated.

We describe two variants of the sparsification technique. We use the first variant in situations where no previous fully dynamic algorithm was known. We use a static algorithm to recompute a sparse certificate in each tree node affected by an edge update. If the certificates can be found in time $O(m+n)$, this variant gives time bounds of $O(n)$ per update. In the second variant, we maintain certificates using a dynamic

[1] Throughout $\log x$ stands for $\max(1, \log_2 x)$, so $\log(m/n)$ is never smaller than 1 even if $m < 2n$.

data structure. For this to work, we need a *stability* property of our certificates to ensure that a small change in the input graph does not lead to a large change in the certificates. This variant transforms time bounds of the form $O(m^p)$ into $O(n^p)$. We start by describing an abstract version of sparsification. The technique is based on the concept of a *certificate*:

DEFINITION 8.1 For any graph property $\mathcal{P}$, and graph G, a *certificate* for G is a graph G' such that G has property $\mathcal{P}$ if and only if G' has the property.

Nagamochi and Ibaraki [35] use a similar concept, however they require G' to be a subgraph of G. We do not need this restriction. However, this allows trivial certificates: G' could be chosen from two graphs of constant complexity, one with property $\mathcal{P}$ and one without it.

DEFINITION 8.2 For any graph property $\mathcal{P}$, and graph G, a *strong certificate* for G is a graph G' on the same vertex set such that, for any H, $G \cup H$ has property $\mathcal{P}$ if and only if $G' \cup H$ has the property.

In all our uses of this definition, G and H will have the same vertex set and disjoint edge sets. A strong certificate need not be a subgraph of G, but it must have a structure closely related to that of G. The following facts follow immediately from Definition 8.2.

FACT 8.1 *Let G' be a strong certificate of property $\mathcal{P}$ for graph G, and let G'' be a strong certificate for G'. Then G'' is a strong certificate for G.*

FACT 8.2 *Let G' and H' be strong certificates of $\mathcal{P}$ for G and H. Then $G' \cup H'$ is a strong certificate for $G \cup H$.*

A property is said to have *sparse certificates* if there is some constant c such that for every graph G on an n-vertex set, we can find a strong certificate for G with at most cn edges.

The other key ingredient is a *sparsification tree.* We start with a partition of the vertices of the graph, as follows: we split the vertices evenly in two halves, and recursively partition each half. Thus, we end up with a complete binary tree in which nodes at distance i from the root have $n/2^i$ vertices. We then use the structure of this tree to partition the edges of the graph. For any two nodes α and β of the vertex partition tree at the same level i, containing vertex sets V_α and V_β, we create a node $E_{\alpha\beta}$ in the edge partition tree, containing all edges in $V_\alpha \times V_\beta$. The parent of $E_{\alpha\beta}$ is $E_{\gamma\delta}$, where γ and δ are the parents of α and β, respectively, in the vertex partition tree. Each node $E_{\alpha\beta}$ in the edge partition tree has either three or four children (three if $\alpha = \beta$, four otherwise). We use a slightly modified version of this edge partition tree as our sparsification tree. The modification is that we only construct those nodes $E_{\alpha\beta}$ for which there is at least one edge in $V_\alpha \times V_\beta$. If a new edge is inserted new nodes are created as necessary, and if an edge is deleted those nodes for which it was the only edge are deleted.

LEMMA 8.8 In the sparsification tree described above, each node $E_{\alpha\beta}$ at level i contains edges inducing a graph with at most $n/2^{i-1}$ vertices.

PROOF There can be at most $n/2^i$ vertices in each of V_α and V_β.

We say a time bound $T(n)$ is *well behaved* if for some $c < 1$, $T(n/2) < cT(n)$. We assume well-behavedness to eliminate strange situations in which a time bound fluctuates wildly with n. All polynomials are well behaved. Polylogarithms and other slowly growing functions are not well behaved, but since

sparsification typically causes little improvement for such functions we will in general assume all time bounds to be well behaved.

THEOREM 8.6 **[8]** *Let $\mathcal{P}$ be a property for which we can find sparse certificates in time $f(n, m)$ for some well-behaved f, and such that we can construct a data structure for testing property $\mathcal{P}$ in time $g(n, m)$ which can answer queries in time $q(n, m)$. Then there is a fully dynamic data structure for testing whether a graph has property $\mathcal{P}$, for which edge insertions and deletions can be performed in time $O(f(n, O(n))) + g(n, O(n))$, and for which the query time is $q(n, O(n))$.*

PROOF We maintain a sparse certificate for the graph corresponding to each node of the sparsification tree. The certificate at a given node is found by forming the union of the certificates at the three or four child nodes, and running the sparse certificate algorithm on this union. As shown in Lemmas 8.1 and 8.2, the certificate of a union of certificates is itself a certificate of the union, so this gives a sparse certificate for the subgraph at the node. Each certificate at level i can be computed in time $f(n/2^{i-1}, O(n/2^i))$. Each update will change the certificates of at most one node at each level of the tree. The time to recompute certificates at each such node adds in a geometric series to $f(n, O(n))$. This process results in a sparse certificate for the whole graph at the root of the tree. We update the data structure for property $\mathcal{P}$, on the graph formed by the sparse certificate at the root of the tree, in time $g(n, O(n))$. The total time per update is thus $O(f(n, O(n))) + g(n, cn)$.

This technique is very effective at producing dynamic graph data structures for a multitude of problems, in which the update time is $O(n \log^{O(1)} n)$ instead of the static time bounds of $O(m + n \log^{O(1)} n)$. To achieve sublinear update times, we further refine our sparsification idea.

DEFINITION 8.3 Let A be a function mapping graphs to strong certificates. Then A is *stable* if it has the following two properties:

(1) For any graphs G and H, $A(G \cup H) = A(A(G) \cup H)$.
(2) For any graph G and edge e in G, $A(G - e)$ differs from $A(G)$ by $O(1)$ edges.

Informally, we refer to a certificate as stable if it is the certificate produced by a stable mapping. The certificate consisting of the whole graph is stable, but not sparse.

THEOREM 8.7 **[8]** *Let $\mathcal{P}$ be a property for which stable sparse certificates can be maintained in time $f(n, m)$ per update, where f is well behaved, and for which there is a data structure for property $\mathcal{P}$ with update time $g(n, m)$ and query time $q(n, m)$. Then $\mathcal{P}$ can be maintained in time $O(f(n, O(n))) + g(n, O(n))$ per update, with query time $q(n, O(n))$.*

PROOF As before, we use the sparsification tree described above. After each update, we propagate the changes up the sparsification tree, using the data structure for maintaining certificates. We then update the data structure for property $\mathcal{P}$, which is defined on the graph formed by the sparse certificate at the tree root.

At each node of the tree, we maintain a stable certificate on the graph formed as the union of the certificates in the three or four child nodes. The first part of the definition of stability implies that this certificate will also be a stable certificate that could have been selected by the mapping A starting on the subgraph of all edges in groups descending from the node. The second part of the definition of stability

then bounds the number of changes in the certificate by some constant s, since the subgraph is changing only by a single edge. Thus, at each level of the sparsification tree there is a constant amount of change.

When we perform an update, we find these s changes at each successive level of the sparsification tree, using the data structure for stable certificates. We perform at most s data structure operations, one for each change in the certificate at the next lower level. Each operation produces at most s changes to be made to the certificate at the present level, so we would expect a total of s^2 changes. However, we can cancel many of these changes since as described above the net effect of the update will be at most s changes in the certificate.

In order to prevent the number of data structure operations from becoming larger and larger at higher levels of the sparsification tree, we perform this cancellation before passing the changes in the certificate up to the next level of the tree. Cancellation can be detected by leaving a marker on each edge, to keep track of whether it is in or out of the certificate. Only after all s^2 changes have been processed do we pass the at most s uncancelled changes up to the next level.

Each change takes time $f(n, O(n))$, and the times to change each certificate then add in a geometric series to give the stated bound.

Theorem 8.6 can be used to dynamize static algorithms, while Theorem 8.7 can be used to speed up existing fully dynamic algorithms. In order to apply effectively Theorem 8.6 we only need to *compute* efficiently *sparse* certificates, while for Theorem 8.7 we need to *maintain* efficiently *stable sparse* certificates. Indeed stability plays an important role in the proof of Theorem 8.7. In each level of the update path in the sparsification tree we compute s^2 changes resulting from the s changes in the previous level, and then by stability obtain only s changes after eliminating repetitions and canceling changes that require no update. Although in most of the applications we consider stability can be used directly in a much simpler way, we describe it in this way here for sake of generality.

We next describe the $O(n^{1/2})$ algorithm for the fully dynamic maintenance of a minimum spanning forest given by Eppstein et al. [8] based on sparsification. A minimum spanning forest is not a graph property, since it is a subgraph rather than a Boolean function. However sparsification still applies to this problem. Alternately, sparsification maintains any property defined on the minimum spanning trees of graphs. The data structure introduced in this section will also be an important subroutine in some results described later. We need the following analogue of strong certificates for minimum spanning trees:

LEMMA 8.9 Let T be a minimum spanning forest of graph G. Then for any H there is some minimum spanning forest of $G \cup H$ which does not use any edges in $G - T$.

PROOF If we use the cycle property on graph $G \cup H$, we can eliminate first any cycle in G (removing all edges in $G - T$) before dealing with cycles involving edges in H.

Thus, we can take the strong certificate of any minimum spanning forest property to be the minimum spanning forest itself. Minimum spanning forests also have a well-known property which, together with Lemma 8.9, proves that they satisfy the definition of stability:

LEMMA 8.10 Let T be a minimum spanning forest of graph G, and let e be an edge of T. Then either $T - e$ is a minimum spanning forest of $G - e$, or there is a minimum spanning forest of the form $T - e + f$ for some edge f.

If we modify the weights of the edges so that no two are equal, we can guarantee that there will be exactly one minimum spanning forest. For each vertex v in the graph let $i(v)$ be an identifying number chosen as an integer between 0 and $n - 1$. Let ϵ be the minimum difference between any two distinct weights of the

graph. Then for any edge $e = (u, v)$ with $i(u) < i(v)$ we replace $w(e)$ by $w(e) + \epsilon i(u)/n + \epsilon i(v)/n^2$. The resulting MSF will also be a minimum spanning forest for the unmodified weights, since for any two edges originally having distinct weights the ordering between those weights will be unchanged. This modification need not be performed explicitly—the only operations our algorithm performs on edge weights are comparisons of pairs of weights, and this can be done by combining the original weights with the numbers of the vertices involved taken in lexicographic order. The mapping from graphs to unique minimum spanning forests is stable, since part (1) of the definition of stability follows from Lemma 8.9, and part (2) follows from Lemma 8.10.

We use Frederickson's algorithm of Theorem 8.5 that states that minimum spanning trees can be maintained in time $O(m^{1/2})$. We improve this bound by combining Frederickson's algorithm with sparsification: we apply the stable sparsification technique of Theorem 8.7, with $f(n, m) = g(n, m) = O(m^{1/2})$ by Theorem 8.5.

THEOREM 8.8 **[8]** *The minimum spanning forest of an undirected graph can be maintained in time* $O(n^{1/2})$ *per update.*

The dynamic spanning tree algorithms described so far produce fully dynamic connectivity algorithms with the same time bounds. Indeed, the basic question of connectivity can be quickly determined from a minimum spanning forest. However, higher forms of connectivity are not so easy. For edge connectivity, sparsification can be applied using a dynamic minimum spanning forest algorithm, and provides efficient algorithms: 2-edge connectivity can be solved in $O(n^{1/2})$ time per update, 3-edge connectivity can be solved in $O(n^{2/3})$ time per update, and for any higher k, k-edge connectivity can be solved in $O(n \log n)$ time per update [8]. Vertex connectivity is not so easy: for $2 \leq k \leq 4$, there are algorithms with times ranging from $O(n^{1/2} \log^2 n)$ to $O(n\alpha(n))$ per update [8, 24, 38].

We end this section by mentioning that Henzinger and King have recently improved the update bound of Theorem 8.8 from $O(n^{1/2})$ to $O(n^{1/3} \log n)$. We refer the interested reader to [23] for the details.

Randomized Algorithms

All the previous techniques yield efficient deterministic algorithms for fully dynamic problems. Recently, Henzinger and King [21] proposed a new approach that, exploiting the power of randomization, is able to achieve faster update times for some problems. For the fully dynamic connectivity problem, the randomized technique of Henzinger and King yields an expected amortized update time of $O(\log^3 n)$ for a sequence of at least m_0 updates, where m_0 is the number of edges in the initial graph, and a query time of $O(\log n)$. It needs $\Theta(m + n \log n)$ space. We now sketch the main ideas behind this technique. The interested reader is referred to Chapter 15 for basic definitions on randomized algorithms.

Let $G = (V, E)$ be a graph to be maintained dynamically, and let F be a spanning forest of G. We call edges in F *tree edges,* and edges in $E \setminus F$ *non-tree edges.* First, we describe a data structure which stores all trees in the spanning forest F. This data structure is based on Euler tours, and allows one to obtain logarithmic updates and queries within the forest. Next, we show how to keep the forest F spanning throughout a sequence of updates. The Euler tour data structure for a tree T is simple, and consists of storing the tree vertices according to an Euler tour of T.

Each time a vertex v is visited in the Euler tour, we call this an *occurrence* of v and we denote it by $o(v)$. A vertex of degree Δ has exactly Δ occurrences, except for the root which has $(\Delta + 1)$ occurrences. Furthermore, each edge is visited exactly twice. Given an n-nodes tree T, we encode it with the sequence of $2n - 1$ symbols produced by procedure ET. This encoding is referred to as $ET(T)$. We now analyze how to update $ET(T)$ when T is subject to dynamic edge operations.

If an edge $e = (u, v)$ is deleted from T, denote by T_u and T_v the two trees obtained after the deletion, with $u \in T_u$ and $v \in T_v$. Let $o(u_1)$, $o(v_1)$, $o(u_2)$ and $o(v_2)$ be the occurrences of u and v encountered

during the visit of (u, v). Without loss of generality assume that $o(u_1) < o(v_1) < o(v_2) < o(u_2)$ so that $ET(T) = \alpha o(u_1)\beta o(v_1)\gamma o(v_2)\delta o(u_2)\epsilon$. $ET(T_u)$ and $ET(T_v)$ can be easily computed from $ET(T)$, as $ET(T_v) = o(v_1)\gamma o(v_2)$, and $ET(T_v) = \alpha o(u_1)\beta\delta o(u_2)\epsilon$. To change the root of T from r to another vertex s, we do the following. Let $ET(T) = o(r)\alpha o(s_1)\beta$, where $o(s_1)$ is any occurrence of s. Then, the new encoding will be $o(s_1)\beta\alpha o(s)$, where $o(s)$ is a newly created occurrence of s that is added at the end of the new sequence. If two trees T_1 and T_2 are joined in a new tree T because of a new edge $e = (u, v)$, with $u \in T_1$ and $v \in T_2$, we first reroot T_2 at v. Now, given $ET(T_1) = \alpha o(u_1)\beta$ and the computed encoding $ET(T_2) = o(v_1)\gamma o(v_2)$, we compute $ET(T) = \alpha o(u_1)o(v_1)\gamma o(v_2)o(u)\beta$, where $o(u)$ is a newly created occurrence of vertex u.

Note that all the above primitives require the following operations: (i) splicing out an interval from a sequence, (ii) inserting an interval into a sequence, (iii) inserting or (iv) deleting a single occurrence from a sequence. If the sequence $ET(T)$ is stored in a balanced search tree of degree d (i.e., a balanced d-ary search tree), then one may insert or splice an interval, or insert or delete an occurrence in time $O(d \log n / \log d)$, while maintaining the balance of the tree. It can be checked in $O(\log n/d)$ whether two elements are in the same tree, or whether one element precedes the other in the ordering. The balanced d-ary search tree that stores $ET(T)$ is referred to as the *ET(T)-tree.*

We augment ET-trees to store non-tree edges as follows. For each occurrence of vertex $v \in T$, we arbitrarily select one occurrence to be the *active occurrence* of v. The list of non-tree edges incident to v is stored in the active occurrence of v: each node in the $ET(T)$-tree contains the number of non-tree edges and active occurrences stored in its subtree; thus, the root of the $ET(T)$-tree contains the weight and size of T.

Using these data structures, we can implement the following operations on a collection of trees:

tree(x): return a pointer to $ET(T)$, where T is the tree containing vertex x.

non_tree_edges(T): return the list of non-tree edges incident to T.

sample_n_test(T): select one non-tree edge incident to T at random, where an edge with both endpoints in T is picked with probability $2/w(T)$, and an edge with only endpoint in T is picked with probability $1/w(T)$. Test whether the edge has exactly one endpoint in T, and if so return this edge.

insert_tree(e): join by edge e the two trees containing its endpoints. This operation assumes that the two endpoints of e are in two different trees of the forest.

delete_tree(e): remove e from the tree that contains it. This operation assumes that e is a tree edge.

insert_non_tree(e): insert a non-tree edge e. This operation assumes that the two endpoints of e are in a same tree.

delete_non_tree(e): remove the edge e. This operation assumes that e is a non-tree edge.

Using a balanced binary search tree for representing $ET(T)$, yields the following running times for the above operations: *sample_n_test(T), insert_tree(e), delete_tree(e), insert_non_tree(e),* and *delete_non_tree(e)* in $O(\log n)$ time, and *non_tree_edges(T)* in $O(w(T) \log n)$ time.

We now turn to the problem of keeping the forest of G spanning throughout a sequence of updates. Note that the hard operation is a deletion of a tree edge: indeed, as shown in Section 8.4 a spanning forest is easily maintained throughout edge insertions, and deleting a non-tree edge does not change the forest. Let $e = (u, v)$ be a tree edge of the forest F, and let T_e be the tree of F containing edge e. Let T_u and T_v the two trees obtained from T after the deletion of e, such that T_u contains u and T_v contains v. When e is deleted, T_u and T_v can be reconnected if and only if there is a non-tree edge in G with one endpoint in T_u and one endpoint in T_v. We call such an edge a *replacement edge for e*. In other words, if there is a replacement edge for e, T is reconnected via this replacement edge; otherwise, the deletion of e disconnects T into T_u and T_v. The set of all the replacement edges for e (i.e., all the possible edges reconnecting T_u and T_v), is called the *candidate set of e*.

One main idea behind the technique of Henzinger and King is the following: when e is deleted, use random sampling among the non-tree edges incident to T_e in order to find quickly whether there exists a replacement edge for e. Using the Euler tour data structure, a single random edge adjacent to T_e can be selected and tested whether it reconnects T_e in logarithmic time. The goal is an update time of $O(\log^3 n)$, so we can afford a number of sampled edges of $O(\log^2 n)$. However, the candidate set of e might only be a small fraction of all non-tree edges which are adjacent to T. In this case it is unlikely to find a replacement edge for e among the sampled edges. If we found no candidate among the sampled edges, we check explicitly all the non-tree edges adjacent to T. After random sampling has failed to produce a replacement edge, we need to perform this check explicitly, otherwise we would not be guaranteed to provide correct answers to the queries. Since there might be a lot of edges which are adjacent to T, this explicit check could be an expensive operation, so it should be made a low probability event for the randomized algorithm. This is not yet true, however, since deleting all edges in a relatively small candidate set, reinserting them, deleting them again, and so on will almost surely produce many of those unfortunate events.

The second main idea prevents this undesirable behavior: we maintain an edge decomposition of the current graph G into $O(\log n)$ edge disjoint subgraphs $G_i = (V, E_i)$. These subgraphs are hierarchically ordered. Each i corresponds to a *level*. For each level i, there is a forest F_i such that the union $\cup_{i \leq k} F_i$ is a spanning forest of $\cup_{i \leq k} G_i$; in particular the union F of all F_i is a spanning forest of G. A *spanning forest at level i* is a tree in $\cup_{j \leq i} F_j$. The *weight* $w(T)$ of a spanning tree T at level i is the number of pairs (e', v) such that e' is a non-tree edge in G_i adjacent to the node v in T. If T_1 and T_2 are the two trees resulting from the deletion of e, we sample edges adjacent to the tree with the smaller weight. If sampling is unsuccessful due to a candidate set which is non-empty but relatively small, then the two pieces of the tree which was split are reconnected on the next higher level using one candidate, and all other candidate edges are copied to that level. The idea is to have sparse cuts on high levels and dense cuts on low levels. Non-tree edges always belong to the lowest level where their endpoints are connected or a higher level, and we always start sampling at the level of the deleted tree edge. After moving the candidates one level up, they are normally no longer a small fraction of all adjacent non-tree edges at the new level. If the candidate set on one level is empty, we try to sample on the next higher level. There is one more case to mention: if sampling was unsuccessful despite the fact that the candidate set was big enough, which means that we had bad luck in the sampling, we do not move the candidates to the next level, since this event has a small probability and does not happen very frequently. We present the pseudocode for *replace(u,v,i),* which is called after the deletion of the forest edge $e = (u, v)$ on level i:
replace(u,v,i)

1. **Let** T_u and T_v be the spanning trees at level i containing u and v, respectively. **Let** T be the tree with smaller weight among T_u and T_v. Ties are broken arbitrarily.
2. **If** $w(T) > \log^2 n$ **then**
   (a) **Repeat** sample_n_test(T) for at most $16 \log^2 n$ times. Stop if a replacement edge e is found.
   (b) **If** a replacement edge e is found **then do** delete_non_tree(e), insert_tree(e), and **return.**
3. (a) **Let** S be the set of edges with exactly one endpoint in T.
   (b) **If** $|S| \geq w(T)/(8 \log n)$ **then**
   Select one $e \in S$, delete_non_tree(e), and insert_tree(e).
   (c) **Else if** $0 < |S| < w(T)/(8 \log n)$ **then**
   Delete one edge e from S, delete_non_tree(e), and insert_tree(e) in level $(i + 1)$.
   Forall $e' \in S$ **do** delete_non_tree(e') and insert_non_tree(e') in level $(i + 1)$.
   (d) **Else if** $i < l$ **then** replace$(u, v, i + 1)$.

Note that edges may migrate from level 1 to level l, one level at the time. However, an upper bound of $O(\log n)$ for the number of levels is guaranteed, if there are only deletions of edges. This can be proved as follows. For any i, let m_i be the number of edges ever in level i.

LEMMA 8.11 For any level i, and for all smaller trees T_1 on level i, $\Sigma w(T_1) \leq 2m_i \log n$.

PROOF Let T be a tree which is split into two trees: if an endpoint of an edge is contained in the smaller split tree, the weight of the tree containing the endpoint is at least halved. Thus, each endpoint of a non-tree edge is incident to a small tree at most $\log n$ times in a given level i and the lemma follows.

LEMMA 8.12 For any level i, $m_i \leq m/4^{i-1}$.

PROOF We proceed by induction on i. The lemma trivially holds for $i = 1$. Assume it holds for $(i-1)$. When summed over all small trees T_1 on level $(i-1)$, at most $\Sigma w(T_1)/(8 \log n)$ edges are added to level i. By Lemma 8.11, $\Sigma w(T_1) \leq 2m_{i-1} \log n$, where m_{i-1} is the number of edges ever in level $(i-1)$. The lemma now follows from the induction step.

The following is an easy corollary of Lemma 8.12:

COROLLARY 8.1 The sum over all levels of the total number of edges is $\Sigma_i m_i = O(m)$.

Lemma 8.12 gives immediately a bound on the number of levels:

LEMMA 8.13 The number of levels is at most $l = \lceil \log m - \log \log n \rceil + 1$.

PROOF Since edges are never moved to a higher level from a level with less than $2 \log n$ edges, Lemma 8.12 implies that all edges of G are contained in some E_i, $i \leq \lceil \log m - \log \log n \rceil + 1$.

We are now ready to describe an algorithm for maintaining a spanning forest of a graph G subject to edge deletions. Initially, we compute a spanning forest F of G, compute $ET(T)$ for each tree in the forest, and select active occurrences for each vertex. For each i, the spanning forest at level i is initialized to F. Then, we insert all the non-tree edges with the proper active occurrences into level 1, and compute the number of non-tree edges in the subtree of each node of the binary search tree. This requires $O(m+n)$ times to find the spanning forest and initialize level 1, plus $O(n)$ for each subsequent level to initialize the spanning forest at that level. To check whether two vertices x and y are connected, we test if *tree(x)=tree(y)* on the last level. This can be done in time $O(\log n)$. To update the data structure after the deletion of an edge $e = (u, v)$, we do the following. If e is a non-tree edge, it is enough to perform a *delete_non_tree(e)* in the level where e appears. If e is a tree edge, let i be the level where e first appears. We do a *delete_tree(e)* at level j, for $j \geq i$, and then call *replace(u, v, i)*. This yields the following bounds.

THEOREM 8.9 **[21]** *Let G be a graph with m_0 edges and n vertices subject to edge deletions only. A spanning forest of G can be maintained in $O(\log^3 n)$ expected amortized time per deletion, if there are at least $\Omega(m_0)$ deletions. The time per query is $O(\log n)$.*

PROOF The bound for queries follows from the above argument. Let e be an edge to be deleted. If e

is a non-tree edge, its deletion can be taken care of in $O(\log n)$ time via a *delete_non_tree* primitive.

If e is a tree edge, let T_1 and T_2 the two trees created by the deletion of e, $w(T_1) \leq w(T_2)$. During the sampling phase we spend exactly $O(\log^3 n)$ time, as the cost of *sample_n_test* is $O(\log n)$ and we repeat this for at most $16 \log^2 n$ times.

If the sampling is not successful, collecting and testing all the non-tree edges incident to T_1 implies a total cost of $O(w(T_1) \log n)$. We now distinguish two cases. If we are unlucky in the sampling, $|S| \geq w(T_1)/(8 \log n)$: this happens with probability at most $(1 - 1/(8 \log n))^{16 \log^2 n} = O(1/n^2)$ and thus, contributes an expected cost of $O(\log n)$ per operation. If the cut S is sparse, $|S| < w(T_1)/(8 \log n)$, and we move the candidate set for e one level higher. Throughout the sequence of deletions, the cost incurred at level i for this case is $\Sigma w(T_1) \log n$, where the sum is taken over the small trees T_1 on level i. By Lemma 8.13 and Corollary 8.1 this gives a total cost of $O(m_0 \log^2 n)$.

In all cases where a replacement edge is found, $O(\log n)$ tree operations are performed in the different levels, contributing a cost of $O(\log^2 n)$. Hence, each tree edge deletion contributes a total of $O(\log^3 n)$ expected amortized cost towards the sequence of updates.

If there are also insertions, however, the analysis in Theorem 8.9 does not carry through, as the upper bound on the number of levels in the graph decomposition is no longer guaranteed. To achieve the same bound, there have to be periodical rebuilds of parts of the data structure. This is done as follows. We let the maximum number of levels to be $l = \lceil 2 \log n \rceil$. When an edge $e = (u, v)$ is inserted into G, we add it to the last level l. If u and v were not previously connected, then we do this via a *tree_insert,* otherwise we perform a *non_tree_insert.* In order to prevent the number of levels to grow behind their upper bound, a rebuild of the data structure is executed periodically. A *rebuild at level* i, $i \geq 1$, is carried out by moving all the tree and non-tree edges in level j, $j > i$, back to level i. Also, for each $j > i$ all the tree edges in level i are inserted into level j. Note that after a rebuild at level i, $E_j = \emptyset$ and $F_j = F_i$ for all $j > i$, i.e., there are no non-tree edges above level i, and the spanning trees on level $j \geq i$ span G.

The crucial point of this method is deciding when to apply a rebuild at level i. This is done as follows. We keep a counter K that counts the number of edge insertions modulo $2^{\lceil 2 \log n \rceil}$ since the start of the algorithm. Let $K_1, K_2, \ldots, K_l$ be the binary representation of K, with K_1 being the most significant bit: we perform a rebuild at level i each time the bit K_i flips to 1. This implies that the last level is rebuilt every other insertion, level $(l - 1)$ every four insertions. In general, a rebuild at level i occurs every 2^{l-i+1} insertions. We now show that the rebuilds contribute a total cost of $O(\log^3 n)$ toward a sequence of insertions and deletions of edges.

For the sake of completeness, assume that at the beginning the initialization of the data structures is considered a rebuild at level 1. Given a level i, we define an *epoch for* i to be any period of time starting right after a rebuild at level j, $j \leq i$, and ending right after the next rebuild at level j', $j' \leq i$. Namely, an epoch for i is the period between two consecutive rebuilds below or at level i: an epoch for i starts either at the start of the algorithm or right after some bit K_j, $j \leq i$, flips to 1, and ends with the next such flip. It can be easily seen that each epoch for i occurs every 2^{l-i} insertions, for any $1 \leq i \leq l$. There are two types of epochs for i, depending on whether it is bit K_i or a bit K_j, $j > i$, that flips to 1: an *empty* epoch for i starts right after a rebuild at j, $j < i$, and a *full* epoch for i starts right after a rebuild at i. At the time of the initialization, a *full* epoch for 1 starts, while for $i \geq 2$, *empty* epochs for i starts. The difference between these two types of epochs is the following. When an *empty* epoch for i starts, all the edges at level i have been moved to some level j, $j < i$, and consequently E_i is empty. On the contrary, when a *full* epoch for i starts, all the edges at level k, $k > i$, have been moved to level i, and thus, $E_i \neq \emptyset$.

An important point in the analysis is that for any $i \geq 2$, any epoch for $(i - 1)$ consists of two parts, one corresponding to an *empty* epoch for i followed by another corresponding to a *full* epoch for i. This happens because a flip to 1 of a bit K_j, $j \leq 2$, must be followed by a flip to 1 of K_i before another bit $K_{j'}$, $j' \leq i$ flips again to 1. Thus, when each epoch for $(i - 1)$ starts, E_i is empty. Define m'_i to be the number of edges ever in level i during an epoch for i. The following lemma is the analogous of Lemma 8.11.

LEMMA 8.14 For any level i, and for all smaller trees T_1 on level i searched during an epoch for i, $\Sigma w(T_1) \leq 2m'_i \log n$.

LEMMA 8.15 $m'_i < n^2/2^{i-1}$.

PROOF To prove the lemma, it is enough to bound the number of edges that are moved to level i during any one epoch for $(i-1)$, as $E_i = \emptyset$ at the start of each epoch for $(i-1)$ and each epoch for i is contained in one epoch for $(i-1)$. Consider an edge e that is in level i during one epoch for $(i-1)$. There are only two possibilities: either e was passed up from level $(i-1)$ because of an edge deletion, or e was moved down during a rebuild at i. Assume that e was moved down during a rebuild at i: since right after the rebuild at i the second part of the epoch for $(i-1)$ (i.e., the *full* epoch for i) starts, e was moved back to level i still during the *empty* epoch for i. Note that E_k, for $k \geq i$, was empty at the beginning of the epoch for $(i-1)$; consequently, either e was passed up from E_{i-1} to E_i or e was inserted into G during the *empty* epoch for i. In summary, denoting by a_{i-1} the maximum number of edges passed up from E_{i-1} to E_i during one epoch for $(i-1)$, and by b_i the number of edges inserted into G during one epoch for i, we have that $m'_i \leq a_{i-1} + b_i$. By definition of epoch, the number of edges inserted during an epoch for i is $b_i = 2^{l-i}$. It remains for us to bound a_{i-1}. Applying the same argument as in the proof of Lemma 8.12, using this time Lemma 8.14, yields that $a_{i-1} \leq m'_{i-1}/4$. Substituting for a_{i-1} and b_i yields $m'_i \leq m'_{i-1}/4 + 2^{l-i}$, with $m'_1 \leq n^2$. Since $l = \lceil 2 \log n \rceil$, $m'_i < n^2/2^{i-1}$.

Lemma 8.15 implies that $m'_l \leq 2$, and thus, edges never need to be passed to a level higher than l.

COROLLARY 8.2 All edges of G are contained in some level E_i, $i \leq \lceil 2 \log n \rceil$.

We are now ready to analyze the running time of the entire algorithm.

THEOREM 8.10 **[21]** *Let G be a graph with m_0 edges and n vertices subject to edge insertions and deletions. A spanning forest of G can be maintained in $O(\log^3 n)$ expected amortized time per update, if there are at least $\Omega(m_0)$ updates. The time per query is $O(\log n)$.*

PROOF There are two main differences with the algorithm for deletions only described in Theorem 8.9. The first is that now the actual cost of an insertion has to be taken into account (i.e., the cost of operation *move_edges*). The second difference is that the argument that a total of $O(m_i \log n)$ edges are examined throughout the course of the algorithm when sparse cuts are moved one level higher must be modified to take into account epochs.

The cost of executing *move_edges(i)* is the cost of moving each non-tree and tree edge from E_j to E_i, for all $j > i$, plus the cost of updating all the forests F_k, $i \leq k < j$. The number of edges moved into level i by a *move_edges(i)* is $\Sigma_{j>i} m'_j$, which by Lemma 8.15, is never greater than $n^2/2^{i-1}$. Since moving one edge costs $O(\log n)$, the total cost incurred by a *move_edges(i)* operation is $O(n^2 \log n/2^i)$. Note that during an epoch for i, at most one *move_edges(i)* can be performed, since that will end the current epoch and start a new one.

Inserting a tree edge into a given level costs $O(\log n)$. Since a tree edge is never passed up during edge deletions, it can be added only once to a given level. This yields a total of $O(\log^2 n)$ per tree edge.

We now analyze the cost of collecting and testing the edges from all smaller trees T_1 on level i during an epoch for i (when sparse cuts for level i are moved to level $(i+1)$). Fix a level $i \leq l$. If $i = l$, since there are $O(1)$ edges in E_l at any given time, the total cost for collecting and testing on level l will be

$O(\log n)$. If $i < l$, the cost of collecting and testing edges on all small trees on level i during an epoch for i is $O(2m'_i \log n \times \log n)$ because of Lemma 8.14. By Lemma 8.15, this is $O(n^2 \log^2 n/2^i)$.

In summary, each update contributes $O(\log^3 n)$ per operation plus $O(n^2 \log^2 n/2^i)$ per each epoch for i, $1 \leq i \leq l$. To amortized the latter bound against insertions, during each insertion we distribute $\Theta(\log^2 n)$ credits per level. This sums up to $\Theta(\log^3 n)$ credits per insertion. An epoch for i occurs every $2^{l-i} = \Theta(n^2/2^i)$ insertions, at which time level i has accumulated $\Theta(n^2 \log^2 n/2^i)$ credits to pay for the cost of *move_edges(i)* and the cost of collecting and testing edges on all small trees.

We end this section by mentioning that Henzinger and Thorup have recently improved the update bound of Theorem 8.10 from $O(\log^3 n)$ to $O(\log^2 n)$. We refer the interested reader to [25] for the details.

8.6 Research Issues and Summary

In this chapter we have described the most efficient known algorithms for maintaining dynamic graphs. Experimental comparison of some of the dynamic connectivity algorithms has recently been performed by Alberts et al. [1], who showed that in the average case for sufficiently random inputs, a simple sparsification tree based on edge subdivision performs as well as the vertex subdivision method we described in Theorem 8.6. Furthermore, they compared this simplified sparsification algorithm (having a worst-case update bound of $O(n \log(m/n))$) with the randomized method of Henzinger and King (having an expected amortized bound of $O(\log^3 n)$). The sparsification method worked well for small update sequences, but the other method was faster on longer sequences. In subsequent work, Amato et al. [2] conducted an empirical study of some dynamic minimum spanning trees algorithms. In particular, they showed that a variant of the algorithm of Frederickson (based on one-level clustering) performs well both for random and nonrandom inputs. Sparsification on top of this variant of Frederickson yields better algorithms for nonrandom inputs.

There are still several open questions. For some of the fully dynamic problems described in this chapter, such as connectivity and 2-connectivity, polylogarithmic randomized algorithms are available. However, the deterministic bounds for the same problems have higher running times. Are there any faster deterministic algorithms for these problems? Furthermore, no randomized algorithm is known for the fully dynamic maintenance of a minimum spanning tree, and the fastest algorithm requires $O(n^{1/2})$ time per update. Is there any faster randomized algorithm for this? Very little is known about lower bounds for **fully dynamic graph problems.** The only nontrivial lower bounds known are the $\Omega(\log n/\log\log n)$ lower bounds of Fredman and Henzinger [14] for the cell-probe model of computation. While the randomized algorithms described in this chapter are close to these lower bounds, there is still a big gap for the deterministic algorithms. Can the gap between upper and lower bounds be tightened in this case? Furthermore, can we prove nontrivial lower bounds for other fully dynamic problems as well? Can we allow other update operations besides edge insertion and deletion? Usually, isolated vertices can be inserted and deleted in the same times as edge insertion and deletion. Is there some way of allowing rapid deletion of vertices that may still be connected to many edges?

8.7 Defining Terms

Certificate: For any graph property $\mathcal{P}$, and graph G, a certificate for G is a graph G' such that G has property $\mathcal{P}$ if and only if G' has the property.

Fully dynamic graph problem: Problem where the update operations include unrestricted insertions and deletions of edges.

Partially dynamic graph problem: Problem where the update operations include either edge insertions (*incremental*) or edge deletions (*decremental*).

Sparsification: Technique for designing dynamic graph algorithms, which when applicable transform a time bound of $T(n, m)$ into $O(T(n, n))$, where m is the number of edges, and n is the number of vertices of the given graph.

Topology tree: Tree that describes a balanced decomposition of another tree, according to its topology.

Acknowledgments

The first author was supported in part by NSF Grant CCR-9258355 and matching funds from Xerox Corp. The work of the second author was supported in part by NSF Grant CCR-9316209 and the CISE Institutional Infrastructure Grant CDA-9024735. The third author was supported in part by the ESPRIT LTR Project No. 20244 (ALCOM-IT) and by a research grant from University of Venice "Ca' Foscari."

References

[1] Alberts, D., Cattaneo, G., and Italiano, G.F., An empirical study of dynamic graph algorithms. *ACM Journal on Experimental Algorithmics,* vol. 2, 1997.

[2] Amato, G., Cattaneo, G., and Italiano, G.F., Experimental analysis of dynamic minimum spanning tree algorithms. In *Proc. 8th ACM-SIAM Annual Symp. on Discrete Algorithms* (SODA 97), New Orleans, LA, 314–323, 5-7 Jan 1997.

[3] Ausiello, G., Italiano, G.F., Marchetti-Spaccamela, A., and Nanni, U., Incremental algorithms for minimal length paths. *J. Algorithms,* 12, 615–638, 1991.

[4] Di Battista, G. and Tamassia, R., Incremental planarity testing. In *Proc. 30th IEEE Symp. Foundations of Computer Science,* 436–441, 1989.

[5] Di Battista, G. and Tamassia, R., On-line graph algorithms with SPQR-trees. In *Proc. 17th Int. Colloquium on Automata, Languages and Programming,* 598–611. Lecture Notes in Computer Science 443, Springer-Verlag, Berlin, 1990.

[6] Dinitz, E.A., Maintaining the 4-edge-connected components of a graph on-line. In *Proc. 2nd Israel Symp. Theory of Computing and Systems,* 88–99, 1993.

[7] Eppstein, D., Clustering for faster network simplex pivots. In *Proc. 5th ACM-SIAM Symp. Discrete Algorithms,* 160–166, 1994.

[8] Eppstein, D., Galil, Z., Italiano, G.F., and Nissenzweig, A., Sparsification—A technique for speeding up dynamic graph algorithms. To appear in *J. of Assoc. Comput. Mach.,* (1997). See also *Proc. 33rd IEEE Symp. Foundations of Computer Science,* 60–69, 1992.

[9] Eppstein, D., Galil, Z., Italiano, G.F., and Spencer, T.H., Separator based sparsification I: Planarity testing and minimum spanning trees. *Journal of Computer and System Science,* Special issue of STOC 93, 52(1), 3–27, 1996.

[10] Eppstein, D., Galil, Z., Italiano, G.F., and Spencer, T.H., Separator based sparsification II: Edge and vertex connectivity. To appear in *SIAM Journal on Computing.*

[11] Eppstein, D., Italiano, G.F., Tamassia, R., Tarjan, R.E., Westbrook, J., and Yung, M., Maintenance of a minimum spanning forest in a dynamic plane graph. *J. Algorithms,* 13, 33–54, 1992.

[12] Frederickson, G.N., Data structures for on-line updating of minimum spanning trees. *SIAM J. Comput.,* 14, 781–798, 1985.

[13] Frederickson, G.N., Ambivalent data structures for dynamic 2-edge-connectivity and k smallest spanning trees. *SIAM J. Comput.,* 26, 484–538, 1997.

[14] Fredman, M.L. and Henzinger, M.R., Lower bounds for fully dynamic connectivity problems in graphs. *Algorithmica.* To appear.

[15] Frigioni, D., Marchetti-Spaccamela, A., and Nanni, A., Fully dynamic output bounded single source shortest path problems. In *Proc. 7th ACM-SIAM Symp. Discrete Algorithms,* 212–221, 1996.

[16] Galil, Z. and Italiano, G.F., Fully dynamic algorithms for 2-edge-connectivity. *SIAM J. Comput.,* 21, 1047–1069, 1992.

[17] Galil, Z. and Italiano, G.F., Maintaining the 3-edge-connected components of a graph on-line. *SIAM J. Comput.,* 22, 11–28, 1993.

[18] Galil, Z., Italiano, G.F., and Sarnak, N., Fully dynamic planarity testing. In *Proc. 24th ACM Symp. Theory of Computing,* 495–506, 1992.

[19] Giammarresi, D. and Italiano, G.F., Decremental 2- and 3-connectivity on planar graphs. *Algorithmica,* 16(3), 263–287, 1996.

[20] Harary, F., *Graph Theory.* Addison-Wesley, Reading, MA, 1969.

[21] Henzinger, M.R. and King, V., Randomized dynamic graph algorithms with polylogarithmic time per operation. In *Proc. 27th Symp. on Theory of Computing,* 519–527, 1995.

[22] Henzinger, M.R. and King, V., Fully dynamic biconnectivity and transitive closure. In *Proc. Proc. 36th IEEE Symp. Foundations of Computer Science,* 664–672, 1995.

[23] Henzinger, M.R. and King, V., Maintaining minimum spanning trees in dynamic graphs. In *Proc. 24th Int. Coll. Automata, Languages, and Programming,* 594–604. Lecture Notes in Computer Science 1256, Springer-Verlag, Berlin, 1997.

[24] Henzinger, M.R. and La Poutré, J.A., Certificates and fast algorithms for biconnectivity in fully dynamic graphs. In *Proc. 3rd European Symp. on Algorithms,* 171–184. Lecture Notes in Computer Science 979, Springer-Verlag, Berlin, 1995.

[25] Henzinger, M.R. and Thorup, M., Improved sampling with applications to dynamic graph algorithms. In *Proc. 23rd Int. Colloquium on Automata, Languages and Programming,* 1996.

[26] Hershberger, J., Rauch, M., and Suri, S., Data structures for two-edge connectivity in planar graphs. *Theor. Comp. Sci.,* 130, 139–161, 1994.

[27] Italiano, G.F., Amortized efficiency of a path retrieval data structure. *Theor. Comput. Sci.,* 48, 273–281, 1986.

[28] Italiano, G.F., Finding paths and deleting edges in directed acyclic graphs. *Inform. Proc. Lett.,* 28, 5–11, 1988.

[29] Kanevsky, A., Tamassia, R., Di Battista, G., and Chen, J., On-line maintenance of the four-connected components of a graph. In *Proc. 32nd IEEE Symp. Foundations of Computer Science,* 793–801, 1991.

[30] Klein, P.N. and Sairam, S., Fully dynamic approximation schemes for shortest path problems in planar graphs. In *Proc. 3rd Worksh. Algorithms and Data Structures,* 442–451. Lecture Notes in Computer Science 709, Springer-Verlag, Berlin, 1993.

[31] La Poutré, J.A., Maintenance of 2- and 3-connected components of graphs, part II: 2- and 3-edge-connected components and 2-vertex-connected components. Technical Report ALCOM-91-145, Department of Computer Science, Utrecht University, 1991.

[32] La Poutré, J.A., Maintenance of triconnected components of graphs. In *Proc. 19th Int. Colloquium on Automata, Languages and Programming,* 354–365. Lecture Notes in Computer Science 623, Springer-Verlag, Berlin, 1992.

[33] La Poutré, J.A. and van Leeuwen, J., Maintenance of transitive closure and transitive reduction of graphs. In *Proc. Workshop on Graph-Theoretic Concepts in Computer Science,* 106–120. Lecture Notes in Computer Science 314, Springer-Verlag, Berlin, 1988.

[34] La Poutré, J.A., van Leeuwen, J., and Overmars, M.H., Maintenance of 2- and 3-connected components of graphs, part I: 2- and 3-edge-connected components. *Discrete Mathematics,* 114, 329–359, 1993.

[35] Nagamochi, H. and Ibaraki, T., Linear time algorithms for finding a sparse k-connected spanning subgraph of a k-connected graph. *Algorithmmica,* 7, 583–596, 1992.

[36] Ramalingam, G., *Bounded Incremental Compilation.* Ph.D. Thesis, Department of Computer Science, University of Wisconsin at Madison, Aug. 1993.
[37] Rauch, M., Fully dynamic biconnectivity in graphs. In *Proc. 33rd IEEE Symp. Foundations of Computer Science,* 50–59, 1992.
[38] Rauch, M., Improved data structures for fully dynamic biconnectivity. In *Proc. 26th Symp. Theory of Computing,* 1994.
[39] Sleator, D.D. and Tarjan, R.E., A data structure for dynamic trees. *J. Comp. Syst. Sci.,* 24, 362–381, 1983.
[40] Tamassia, R., A dynamic data structure for planar graph embedding. In *Proc. 15th Int. Colloquium on Automata, Languages and Programming,* 576–590. Lecture Notes in Computer Science 317, Springer-Verlag, Berlin, 1988.
[41] Tamassia, R. and Preparata, F.P., Dynamic maintenance of planar digraphs, with applications. *Algorithmica,* 5, 509–527, 1990.
[42] Tamassia, R. and Tollis, I.G., Dynamic reachability in planar digraphs with one source and one sink. *Theor. Comput. Sci.,* 119, 331–344, 1993.
[43] Tarjan, R.E., Depth-first search and linear graph algorithms. *SIAM J. Comput.,* 1, 146–160, 1972.
[44] Westbrook, J. and Tarjan, R.E., Maintaining bridge-connected and biconnected components on-line. *Algorithmica,* 7, 433–464, 1992.
[45] Yellin, D.M., Speeding up dynamic transitive closure for bounded degree graphs. *Acta Informatica,* 30, 369–384, 1993.

Further Information

Research on dynamic graph algorithms is published in many computer science journals, including *Algorithmica, Journal of ACM, Journal of Algorithms, Journal of Computer and System Science, SIAM Journal on Computing,* and *Theoretical Computer Science.* Work on this area is published also in the proceedings of general theoretical computer science conferences, such as the *ACM Symposium on Theory of Computing* (STOC), the *IEEE Symposium on Foundations of Computer Science* (FOCS), and the *International Colloquium on Automata, Languages and Programming* (ICALP). More specialized conferences devoted exclusively to algorithms are the *ACM–SIAM Symposium on Discrete Algorithms* (SODA), and the *European Symposium on Algorithms* (ESA).

9

Graph Drawing Algorithms

Peter Eades
The University of Newcastle

Petra Mutzel
Max-Planck-Institute fur Informatik

9.1 Introduction

Graphs are commonly used in Computer Science to model relational structures such as programs, databases, and data structures. For example:

- Petri nets are used extensively to model communications protocols.
- Call graphs of programs are often used in CASE tools; an example is Fig. 9.1(a).
- Object-oriented design techniques use a variety of graphs; one such example is the class diagram in Fig. 9.1(b).
- Data flow graphs are used widely in software engineering.

One of the critical problems in using such models is that the graph must be drawn in a way that illuminates the information in the application. A good **graph drawing** gives a clear understanding of a structural model; a bad drawing is simply misleading. For example, a graph of a computer network is pictured in Fig. 9.2(a); this drawing is easy to follow. A different drawing of the same graph in Fig. 9.2(b); this is much more difficult to follow.

A *graph drawing algorithm* takes a graph and produces a drawing of it. The *graph drawing problem* is to find graph drawing algorithms that produce good drawings. To make the problem more precise, we define a *graph* $G = (V, E)$ to consist of a set V of *vertices* and a set E of *edges,* that is, unordered pairs of vertices. A *drawing* of G assigns a location (in two or three dimensions) to every vertex of G and a simple curve c_{uv} to every edge (u, v) of G such that the endpoints of c_{uv} are the locations of u and v. Notation and terminology of graph theory is given in [4].

The remainder of this chapter gives an overview of graph drawing (Section 9.2) and details some methods for straight-line ("Straight Line Drawings" and "Straight-Line Drawings in Three Dimensions") and orthogonal ("Orthogonal Drawings" and "Three Dimensional Orthogonal Grid

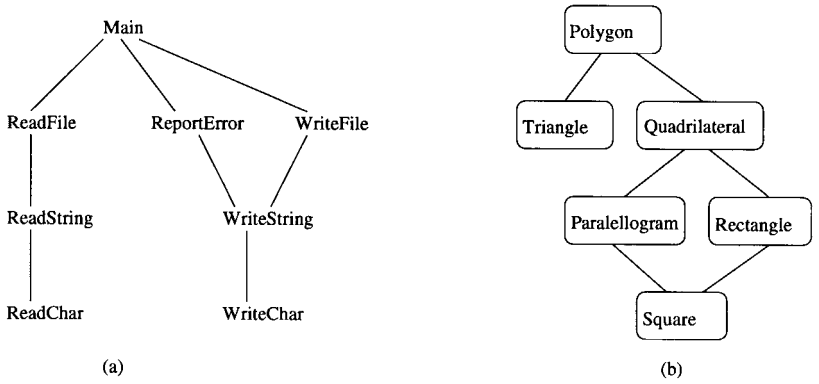


Figure 9.1 Two graphs.

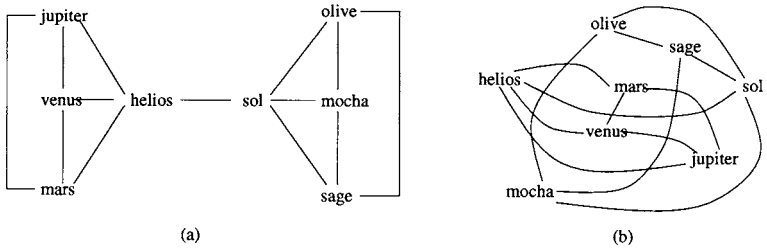


Figure 9.2 Two drawings of the same graph.

Drawings") drawings in two (Section 9.3) and three (Section 9.4) dimensions. A list of research problems is given in Section 9.5, and a review of the defining terms is given in Section 9.6.

9.2 Overview

In this section we provide a brief overview of the graph drawing problem. "Drawing Conventions" outlines the main conventions for drawing graphs, and "Aesthetic Criteria" lists the most common "aesthetic criteria," or optimization goals, of graph drawing. A brief survey of some significant graph drawing algorithms is given in "Drawing Methods."

Drawing Conventions

Drawing conventions for graphs differ from one application area to another. Some of the common conventions are listed below.

- Many graph drawing methods output a *grid* drawing: the location of each vertex has integer coordinates. Some **grid drawings** appear in Fig. 9.3. The algorithms in Sections 9.3 and 9.4 produce grid drawings.
- In a *polyline* drawing, the curve representing each edge is a polyline, that is, a chain of line segments. Polyline drawings are in Fig. 9.3(b), (c), and (d). If each polyline is just a line segment, then the drawing is a *straight-line* drawing, as in Fig. 9.3(c). Algorithms for creating **straight-line drawings** are given in "Straight Line Drawings" and "Straight-Line Drawings in Three Dimensions."
- In an *orthogonal* drawing, each edge is a polyline composed of straight line segments parallel to one of the coordinate axes. **Orthogonal drawings** are used in many application areas because horizontal and vertical line segments are easy to follow.

Figures 9.3(b) and (d) are orthogonal drawings. Orthogonal drawing algorithms are described in "Orthogonal Drawings" and "Three Dimensional Orthogonal Grid Drawings."

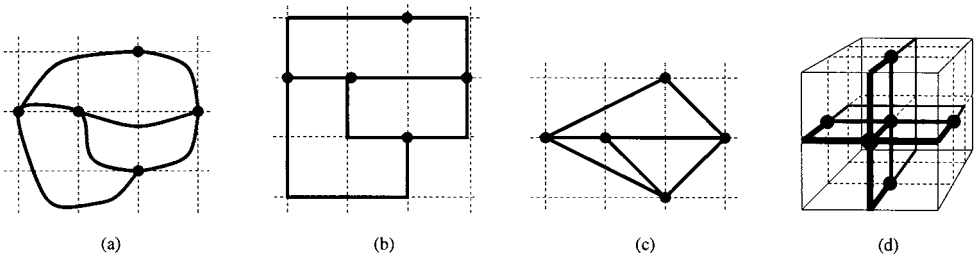


Figure 9.3 Examples of drawing conventions: (a) grid drawing (b) orthogonal grid drawing (c) straight-line drawing (d) three-dimensional orthogonal drawing.

Aesthetic Criteria

The main requirement of a graph drawing method is that the output should be *readable*; that is, it should be easy to understand, easy to remember, and it should illuminate rather than obscure the application. Of course it is difficult to model readability precisely, since it varies from one application to another, and from one human to another; these variations mean that there are many graph drawing problems. The problems can be classified roughly according to the specific optimization goals which they try to achieve. These goals are called *aesthetic criteria*; a list of some such criteria is below.

- Minimization of the number of **edge crossings** is an aesthetic that is important in many application areas. The drawing in Fig. 9.2(a) has no edge crossings, and Fig. 9.2(b) has ten.
  A graph that can be drawn in the plane with no edge crossings is called a **planar graph.** Methods for drawing planar graphs are described in Section 9.3.
  In general, visualization systems must deal with graphs which are not planar. To exploit the theory of planar graphs and methods for drawing planar graphs, it is necessary to *planarize* nonplanar graphs, that is, to transform them to planar graphs. A **planarization** technique is described in "Planarization."
- The *vertex resolution* of a drawing is the minimum distance between a pair of vertices. For a given screen size, we would like to have a drawing with maximum resolution.
  In most cases, the drawing is a grid drawing; this guarantees that the drawing has vertex resolution at least one. To ensure adequate vertex resolution, we try to keep the *size* of the grid drawing bounded. If a two-dimensional grid drawing lies within an isothetic rectangle of width w and height h, then the vertex resolution for a unit square screen is at least $\max(1/w, 1/h)$. All the methods described in Sections 9.3 and 9.4 give grid drawings with polynomially bounded size.
- *Bends*: In polyline drawings it is easier to follow edges with fewer bends. Thus, many graph drawing methods aim to minimize, or at least bound, the number of edge bends. In the drawing in Fig. 9.3(b), there are 6 edge bends; the maximum number of bends on an edge is two. Algorithms for straight-line drawings (no bends at all) are given in "Straight Line Drawings" and "Straight-Line Drawings in Three Dimensions."

Very few graphs have an orthogonal drawing with no bends, but there are a number of methods which aim to keep the number of bends in orthogonal drawings small. A method for creating 2-dimensional planar orthogonal drawings of a planar graph (with a fixed embedding) with a minimum number of bends is described in "Orthogonal Drawings." A method for creating orthogonal drawings in 3 dimensions with a bound on the number of bends in each edge is described in "Three Dimensional Orthogonal Grid Drawings."

Drawing Methods

In Sections 9.3 and 9.4, we describe some two- and three-dimensional graph drawing methods in detail. However, there are many different approaches to the graph drawing problem besides those described in Sections 9.3 and 9.4. Here we briefly overview some of the more significant methods.

Force-Directed Methods

Force-directed methods draw a physical analogy between the layout problem and a system of forces defined on drawings of graphs. For example, vertices may be replaced with bodies that repel each other, and edges have been replaced with Hooke's law springs. In general, these methods have two parts:

The *model*: this is a "force system" defined by the vertices and edges of the graph. It is a physical model for a graph. The model may be defined in terms of "energy" rather than a system of forces; the force system is just the derivative of the energy system.

The *algorithm*: this is a technique for finding an equilibrium state of the force system, that is, a position for each vertex such that the total force on every vertex is zero. This state defines a drawing of the graph. If the model is stated as an energy system then the algorithm is usually stated as a technique for finding a configuration with locally minimal energy.

Force-directed algorithms are easy to understand and easy to implement, and thus, they have become quite popular. They are fairly successful with "tree-like" graphs [see Fig. 9.4(a)]. They work in both two and three dimensions. A comparison of the many variations of the basic idea appears in [6].

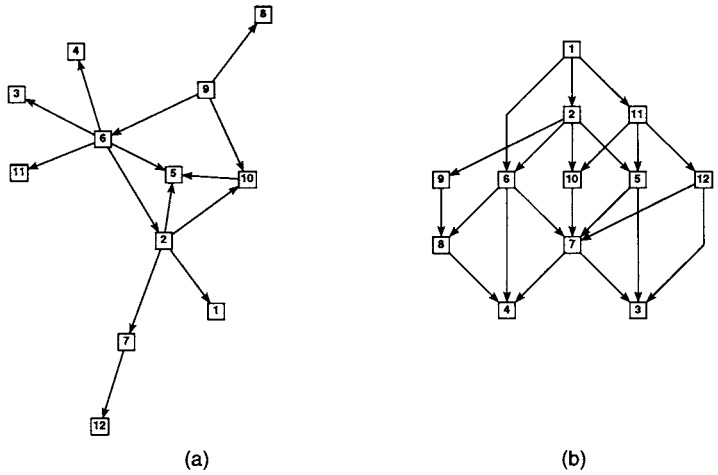


Figure 9.4 (a) A force-directed drawing and (b) a hierarchical drawing.

Hierarchical Methods

Hierarchical methods are suitable for directed graphs, especially where the graph has very few directed cycles.

Suppose that $G = (V, E)$ is an acyclic directed graph. A *layering* of G is a partition of V into subsets $L_1, L_2, \ldots, L_h$, such that if $(u, v) \in E$ where $u \in L_i$ and $v \in L_j$ then $i > j$. An acyclic directed graph with a layering is a *hierarchical graph.*

Hierarchical methods convert a directed graph into a hierarchical graph, and draw the hierarchical graph such that layer L_i lies on the horizontal line $y = i$. A sample drawing is in Fig. 9.4(b). The layers in this graph are $L_1 = \{3, 4\}$, $L_2 = \{7, 8\}$, $L_3 = \{5, 6, 9, 10, 12\}$, $L_4 = \{2, 11\}$, and $L_5 = \{1\}$. The aims of these methods include the following.

(a) Represent the "flow" of the graph from top to bottom of the page. This implies that most arcs should point downward.

(b) Ensure that the graph drawing fits the page; that is, it is not too high (not too many layers) and not too wide (not too many vertices in each layer).

(c) Ensure that arcs are not too long. This implies that the y extent $|i - j|$ of an arc (u, v) with $u \in L_i$ and $v \in L_j$ should be minimized.

(d) Reduce the number of edge crossings.

(e) Ensure that arcs are as straight as possible.

There are many hierarchical methods, but the following four steps are common to most.

1. Directed cycles are removed by temporarily reversing some of the arcs. The arcs that are reversed will appear pointing upward in the final drawing and thus, the number of arcs reversed should be small to achieve (a) above.
2. The set of vertices is partitioned into layers. The partition aims to achieve (b) and (c) above.
3. Vertices within each layer are permuted so that the overall number of crossings is small.
4. Vertices are positioned in each layer so that edges which span more than one layer are as straight as possible.

Each step involves heuristics for NP-hard optimization problems. A detailed description of hierarchical methods appears in [20]. An empirical comparison of various hierarchical drawing methods appears in [15].

Tree Drawing Methods

Rooted trees are special hierarchical graphs, and hierarchical methods apply. We can assign vertices at depth k in the tree to layer $h - k$, where h is the maximum depth. The convention that layer i is drawn on the horizontal line $y = i$ helps the viewer to see the hierarchy represented by the tree. However, for trees there are some simpler approaches; here we outline one such method.

Note that the edge crossing problem is trivial for trees. The main challenge is to create a drawing with width small enough to fit the page. The *Reingold–Tilford algorithm* [37] is a simple heuristic method designed to reduce width. Suppose that T is an oriented binary rooted tree. We denote the left subtree by T_L and the right subtree by T_R. Draw subtrees T_L and T_R recursively on separate sheets of paper. Move the drawings of T_L and T_R toward each other as far as possible without the two drawings touching. Now center the root of T between its children. A drawing computed with the Reingold–Tilford algorithm is shown in Fig. 9.5(a).

The Reingold–Tilford method can be extended to nonbinary trees. However, for trees with many children per vertex it is difficult to reduce width while retaining the layering convention (that is, placing vertices of depth k in the tree at a vertical distance of k below the root). The *tip-over* convention reduces width by "tipping over" some of the subtree drawings, and is effective

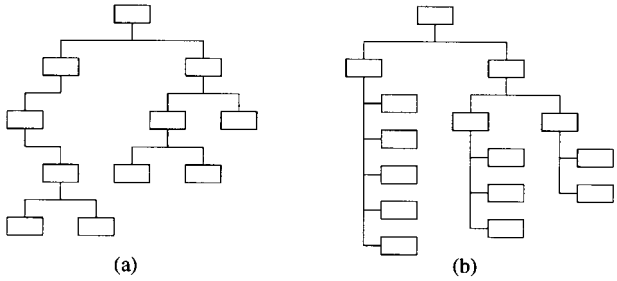


Figure 9.5 (a) Tree drawn with the Reingold–Tilford algorithm and (b) tip-over drawing of a tree.

for trees with large degrees. A sample tip-over drawing is shown in Fig. 9.5(b); algorithms for producing tip-over drawings are given in [19].

9.3 Graph Drawing in Two Dimensions

A great deal of the long history of graph theory has been devoted to the study of representations of graphs in the plane. The concepts and results developed over centuries by graph theorists have proved very useful in recent visualization and VLSI layout applications. This section describes some techniques used in finding good drawings in two dimensions.

A representation of a planar graph without edge crossings in the plane is called a *plane representation* or a *planar embedding.* The graph is then called a *plane* graph. A planar embedding divides the plane into regions, which we call *faces.* The unbounded face is called the *outer face.* A *planar embedding* can be represented by a *planar map,* that is a cyclic ordered list of the edges bounding each face.

Many graphs are not planar, and the first subsection of this section describes methods for transforming a nonplanar graph into a planar graph. The following subsections give two approaches to drawing the planar graph so obtained.

Planarization

Given a nonplanar graph, we want to find a drawing with a minimum number of edge crossings. This problem is NP-hard [24]. A natural approach to the problem is to first determine a maximum planar subgraph, and then insert the remaining edges, while trying to keep a minimal number of crossings.

The first step, finding a maximum planar subgraph, also involves an NP-hard problem [33]. We can find acceptable solutions using a *branch and cut* method, described in this section.

The second step, inserting the "nonplanar" edges, can be done by the following algorithm. We construct a plane representation G_P of the maximum planar subgraph output from the first step; see [34]. Next we construct the geometric dual graph $G_P^* = (V^*, E^*)$ of G_P. For each face f in G_P we introduce a vertex $v_f^* \in V^*$, and for each edge $e \in E$ we introduce an edge $e^* \in E^*$ between the vertices v_f^* and $v_{f'}^*$, where f and f' denote the two faces adjacent to e in G_P [see Fig. 9.6(a)].

Suppose that we already added a new edge $e = (v, w)$ to G_P. The edge e introduces a crossing to G_P, whenever it crosses a border edge between two faces in G_P. Hence, the number of crossings is exactly the number of border edges we cross. Thus, minimizing the number of crossings is equivalent to minimize the corresponding number of edges in the geometric dual on a path from a face in G_P adjacent to v to a face in G_P adjacent to w. We have to solve $|\delta(v)||\delta(w)|$ shortest path problems from vertices $v^* \in A^*(v)$, $w^* \in A^*(w)$ in G_P^*, where $A^*(v)$ and $A^*(w)$ contain all vertices corresponding to all the faces f_v and f_w that are adjacent to v or w in G_P. By adding two super-vertices v_0^* and w_0^* to G^* and edges to all vertices in $A^*(v)$ and $A^*(w)$, the number of shortest path problems can be reduced to one.

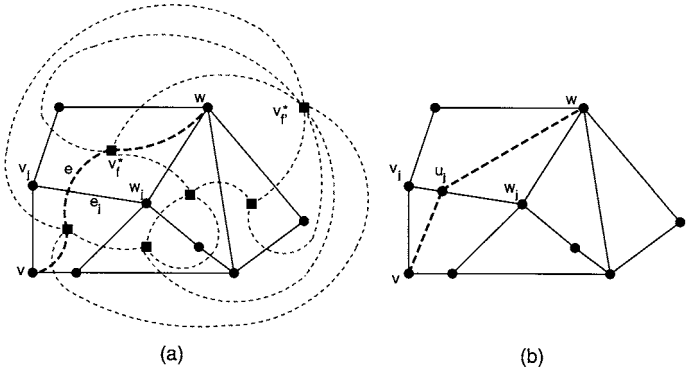


Figure 9.6 Introducing a new edge e to a plane graph G_P: (a) the geometric dual graph G_P^* with the edge e, (b) substituting the new crossing.

If the maximum planar subgraph contains all edges but one, we get the minimal number of crossings with respect to the plane representation G_P. In order to be able to add further edges, we substitute all the new crossings between each pair of edges, say $e = (v, w)$, and $e_j = (v_j, w_j)$, by a new vertex u_j adjacent to the end vertices v, w, v_j, and w_j [see Fig. 9.6(b)]. Again, we have a planar graph and can repeat the above process. Note that this approach is not an exact algorithm for crossing minimization, but experience has shown that it is a good heuristic in the case that the number of inserted edges is small. When all the edges are reinserted, we can use planar graph drawing algorithms as described in the following sections.

Next we consider the problem of determining the maximum planar subgraph. Given a (nonplanar) weighted graph with edge weights c_e for $e \in E$, the *maximum (weight) planar subgraph problem* is to delete a set of edges F to obtain a planar subgraph $G' = (V, E \setminus F)$ such that the sum of all edge weights $\Sigma_{e \in E \setminus F} c_e$ of G' is maximum. In the unweighted case, where $c_e = 1$ for all edges $e \in E$, the problem consists of finding the minimum number of edges whose deletion from a nonplanar graph gives a planar subgraph.

An efficient way to solve a given maximum planar subgraph problem in practice is to use polyhedral combinatorics, a subfield of combinatorial optimization that aims at describing relaxations of combinatorial optimization problems as linear programs and solving these with special purpose methods. In the following we describe this technique and the corresponding algorithm, which leads to quite good and in many cases provably optimal solutions for moderately sized sparse graphs (i.e., up to 100 edges) and very dense graphs.

Integer Programming Formulation

Suppose that a graph $G = (V, E)$ with edge weights c_e for all $e \in E$ is given. Let $\mathcal{P}_\mathcal{G}$ be the set of all planar subgraphs of G. For each planar subgraph $G[P] = (V', P) \in \mathcal{P}_\mathcal{G}$ induced by the edge set $P \subseteq E$, we define its *characteristic vector* $\chi^P \in \mathrm{R}^E$ by setting $\chi_e^P = 1$ if $e \in P$ and $\chi_e^P = 0$ if $e \notin P$. This yields a 1–1-correspondence between the planar subgraphs and certain $\{0, 1\}$-vectors in R^E. The *planar subgraph polytope* $\mathcal{PLS}(G)$ of G is defined as the convex hull over all characteristic vectors of planar subgraphs of G:

$$\mathcal{PLS}(G) := \mathrm{conv}\left\{\chi^P \in \mathrm{R}^E \mid G[P] \in \mathcal{P}_G\right\} \ .$$

The problem of finding a planar subgraph $G[P]$ of G with weight $c(P)$ as large as possible can be written as the linear program

$$\max\left\{c^T x \mid x \in \mathcal{PLS}(G)\right\} \ ,$$

since the vertices of the polytope $\mathcal{PLS}(G)$ are exactly the characteristic vectors of the planar subgraphs of G. In order to apply linear programming techniques to solve this linear program one

has to represent $\mathcal{PLS}(G)$ as the solution of an inequality system. Due to the NP-hardness of our problem, we cannot expect to be able to find a full description of $\mathcal{PLS}(G)$ by linear inequalities. Nevertheless, a partial description of the facial structure of $\mathcal{PLS}(G)$ by linear inequalities is useful in practice, because such a description defines a relaxation of the original problem. Such relaxations can be solved within a branch and bound framework via cutting plane techniques and linear programming in order to produce tight bounds.

In an irredundant description of $\mathcal{PLS}(G)$ by linear inequalities only facet-defining inequalities are present. An inequality $c^T x \leq c_0$ is *facet-defining* for a polytope $\mathcal{P}$ if it is *valid* for $\mathcal{P}$, i.e., $\mathcal{P} \subseteq \{x \in \mathrm{R}^{|E|} \mid c^T x \leq c_0\}$, and there exist $\dim(P)$ affinely independent points in $\mathcal{P}$ satisfying the inequality with equality. Geometrically, a facet-defining inequality corresponds to a face of maximal dimension of $\mathcal{P}$. For efficiency, also in a partial description by inequalities, we concentrate on those valid inequalities for $\mathcal{PLS}(G)$ which are facet-defining.

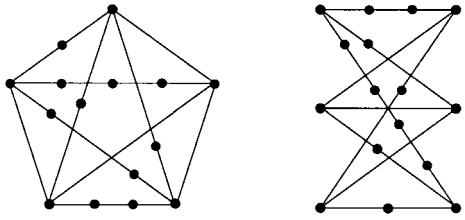


Figure 9.7 Subdivisions of K_5 and $K_{3,3}$.

By Kuratowski's Theorem every nonplanar graph contains a subdivision of K_5 or $K_{3,3}$, (see Fig. 9.7) and it follows that the minimal nonplanar graphs are exactly K_5, $K_{3,3}$, and their subdivisions. Let us call the minimal nonplanar subgraphs Kuratowski subgraphs. A Kuratowski subgraph gives rise to a Kuratowski inequality which requires that at least one edge has to be removed from it. We use the notation $x(K) = \Sigma_{e \in K} x_e$ for $K \subseteq E$. We have the following integer programming formulation for the maximum planar subgraph problem for a graph $G = (V, E)$ with weights c_e associated with every edge:

$$
\begin{array}{lll}
\text{maximize} & c^T x & \\
\text{subject to} & x(K) \leq |K| - 1 & \text{for all } K \subseteq E, K \text{ induces a Kuratowski subgraph} \\
& 0 \leq x_e \leq 1 & \text{for all } e \in E \\
& x_e \text{ integral} & \text{for all } e \in E\ .
\end{array}
$$

The following theorem states that all the above inequalities are necessary in the minimal description of $\mathcal{PLS}(G)$.

THEOREM 9.1 *[30] The dimension of the $\mathcal{PLS}$-polytope of $G = (V, E)$ is $|E|$, so it is full-dimensional. For all edges $e \in E$, the inequalities $x_e \geq 0$ and $x_e \leq 1$ define facets of $\mathcal{PLS}(G)$. Moreover, for all minimal nonplanar subgraphs $G' = (V', F)$ of $G = (V, E)$ the Kuratowski inequality $x(F) \leq |F| - 1$ defines a facet of $\mathcal{PLS}(G)$.*

Another class of inequalities plays an important role in the theory of planar graphs. These can be obtained from Euler's formula for the relationships of vertices, edges and faces in a plane connected graph, as in the following lemma.

LEMMA 9.1 (Euler inequalities) Suppose that $G = (V, E)$ and $G' = (V', E')$ is the subgraph of G induced by $V' \subseteq V$. Then $x(E') \leq 3|V'| - 6$ is valid for $\mathcal{PLS}(G)$. If G' contains no triangles, then the inequality intensifies to $x(E') \leq 2|V'| - 4$.

If the graph $G = (V, E)$ is dense then the Euler inequality may yield a facet; this is the case for $G = K_n$, resp. $G = K_{m,n}$. Further facet-defining inequalities for $\mathcal{PLS}(G)$ that generalize the Kuratowski inequalities are known (see [30]).

The Branch and Cut Algorithm

In the following we outline a cutting plane algorithm using facet-defining inequalities for $\mathcal{PLS}(G)$. The method uses a sequence of "relaxations" of the maximum planar subgraph problem, each solved by linear programming. A relaxation of a combinatorial maximization problem P is another maximization problem P_0, whose set of feasible solutions properly contains all feasible solutions of the original problem. The objective function of P_0 is an extension of the objective function of P. Hence, the value of the optimum solution of the relaxation of our maximum planar subgraph problem is at least as high as the optimum value of the original problem. Consequently, a solution of a relaxation of the maximum planar subgraph problem yields an upper bound.

A possible relaxation is obtained by dropping the integrality constraints in the integer programming formulation for the maximum planar subgraph problem. In general the number of Kuratowski subgraphs in a given graph is too big to be able to enumerate them all. But we do not have to add all of them. In every iteration of our cutting plane algorithm, we only add a few inequalities. We start with an initial linear program containing as constraints only the trivial inequalities and the Euler inequality for G. Hence, we optimize over a polytope containing the planar subgraph polytope $\mathcal{PLS}(G)$.

If the solution vector $\bar{x}$ of this linear program is integral, and it is a characteristic vector of a planar graph, then we know that we have found the optimum solution to our original maximum planar subgraph problem. This is true since we have solved a relaxation of the original problem P, for which we know that a solution of P cannot be better. If either integrality or planarity conditions are not satisfied, then we try to find inequalities which are violated by our current solution vector $\bar{x}$ and which are valid for all planar subgraphs of G. Geometrically speaking, we try to find an inequality defining a hyperplane that cuts off $\bar{x}$; this is the reason why these hyperplanes are called "cutting planes."

The cutting planes can be of type Kuratowski, Euler, or any other facet-defining or valid inequalities [30]. We add the inequality corresponding to such cutting planes to our current linear program. We solve the linear program and proceed in the same way, until we either find an optimum solution or we do not find any violated inequalities for $\bar{x}$. In the latter case, a solution value $z = c^T\bar{x}$ is obtained from the solution vector $\bar{x}$, which is not an incidence vector of a planar graph. A flow diagram of a cutting plane algorithm is shown in Fig. 9.8.

If we fail to solve the maximum planar subgraph problem to optimality, we can switch to branch and bound. That is, we branch by setting a variable to zero or one, and create two new subproblems. In each branching node, exactly the same procedure can be used. Note that all valid inequalities for $\mathcal{PLS}(G)$ are globally valid in the whole branch and bound tree.

The main question arising within every cutting plane method, namely, how to find those cutting planes, is called the *separation problem with respect to* $\mathcal{D}$:

> Given a class of inequalities $\mathcal{D}$, and a vector $\bar{x}$, either find an inequality in this class that is violated by $\bar{x}$, or prove that no such inequality exists.

Unfortunately, the separation problem for the Kuratowski inequalities is still unsolved. A heuristic separation procedure based on finding Kuratowski subgraphs may be used (see [30]). This may find violated inequalities, but in the case that it cannot find any, it is unable to prove that no violated inequality exists.

After a linear program has been solved, we try to exploit the solution to produce a feasible solution. This can be done by a simple greedy heuristic in which the edges are subsequently added, in increasing LP-value order, while they do not destroy planarity.

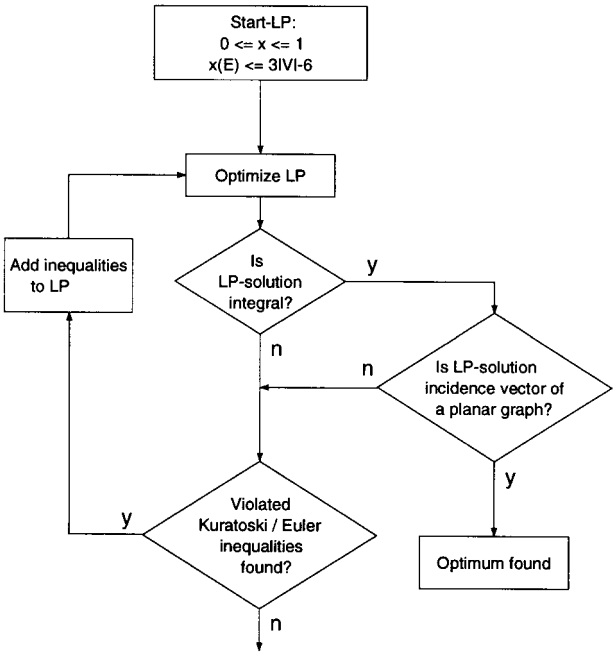


Figure 9.8 Flow diagram of a cutting plane algorithm.

Since during the computation of the branch and cut algorithm a sequence of increasing lower bounds l and decreasing upper bounds u is produced, we can stop the computation at any time with a quality guaranteed solution that deviates by at most a factor of $\frac{u-l}{u}$ from the optimum. Alternatively, we can run the branch and cut algorithm with a prespecified time limit.

Remarks

Computational results for the branch and cut approach are encouraging. For the graph shown in Fig. 9.9, the method took less than one second to find the maximum planar subgraph. In most graphs with up to 70 edges, the branch and cut algorithm provides the optimum solution within a few seconds.

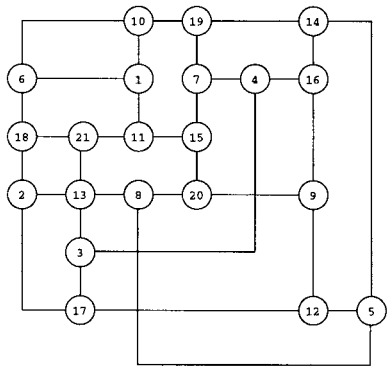


Figure 9.9 Orthogonal layout based on a maximum planar subgraph.

There are many other approaches to the maximum planar subgraph problem; see, for example, [7, 26].

Straight Line Drawings

In this section we consider the problem of constructing straight-line planar drawings of planar graphs. This problem predates the applications in information visualization and was considered by a number of mathematicians [22, 40, 42]. The problem with these early approaches was that they offered poor resolution; that is, they placed vertices exponentially close together. The breakthrough came with the following theorem from [13].

THEOREM 9.2 *Every n-vertex planar graph has a straight-line planar grid drawing which is contained within a rectangle of dimensions $O(n) \times O(n)$.*

The purpose of this section is to outline a constructive proof of Theorem 9.2. Roughly speaking, it proceeds as follows.

1. Dummy edges are added to the graph to ensure that it is a triangulated planar graph, that is, each face is a triangle.
2. The vertices are ordered in a certain way defined below.
3. The vertices are placed one at a time on the grid in the order defined at step 2.

The first step is not difficult. First we find a planar embedding, and then add edges to every face until it becomes a triangle. We present steps 2 and 3 in the following two subsections.

Computing the Ordering

Let $G = (V, E)$ be a triangulated plane graph with $n > 3$ vertices, with vertices u, v, and w on the outer face. Suppose that the vertices of V are ordered $v_1, v_2, \ldots, v_n$. Denote the subgraph induced by $v_1, v_2, \ldots, v_\ell$ by G_ℓ, and the outer face of G_ℓ by C_ℓ. The ordering $v_1 = u, v_2 = v, \ldots, v_n = w$ is a *canonical ordering* if for $3 \leq \ell \leq n-1$, G_ℓ is 2-connected, C_ℓ is a cycle containing the edge (v_1, v_2), $v_{\ell+1}$ is in outer face of G_ℓ, and $v_{\ell+1}$ has the least two neighbors in G_ℓ, and the neighbors are consecutive on the path $C_\ell - (v_1, v_2)$.

LEMMA 9.2 Every triangulated plane graph has a canonical ordering.

PROOF The proof proceeds by reverse induction. The outer face of G is the triangle v_1, v_2, v_n. Since G is triangulated, the neighbors of v_n form a cycle which is the boundary of the outer face of $G - \{v_n\} = G_{n-1}$. Thus, the lemma holds for $\ell = n - 1$.

Suppose that $i \leq n - 2$, and assume that $v_n, v_{n-1}, \ldots v_{i+2}$ have been chosen so that G_{i+1} and C_{i+1} satisfy the requirements of the lemma. We need to choose a vertex w as v_{i+1} on C_{i+1} so that w is not incident with a chord of C_{i+1}. It is not difficult to show that such a vertex exists.

The Drawing Algorithm

The algorithm for drawing the graph begins by drawing vertices v_1, v_2, and v_3 at locations $(0, 0)$, $(2, 0)$, and $(1, 1)$, respectively. Then the vertices $v_4, \ldots v_n$ are placed one at a time, increasing in y coordinate. After v_k has been placed, we have a drawing of the subgraph G_k. Suppose that the outer face C_k of the drawing of G_k consists of the edge (v_1, v_2), and a path $P_k = (v_1 = w_1, w_2, \ldots, w_m = v_2)$; then the drawing is constructed to satisfy the following three properties:

1. The vertices v_1 and v_2 are located at $(0, 0)$ and $(2k - 4, 0)$, respectively.
2. The path P_k is monotonically increasing in the x direction; that is, $x(w_1) < x(w_2) < \cdots < x(w_m)$.

3. For each $i \in \{1, 2, \ldots, m-1\}$, the edge (w_i, w_{i+1}) has slope either $+1$ or -1.

Such a drawing is illustrated in Fig. 9.10.

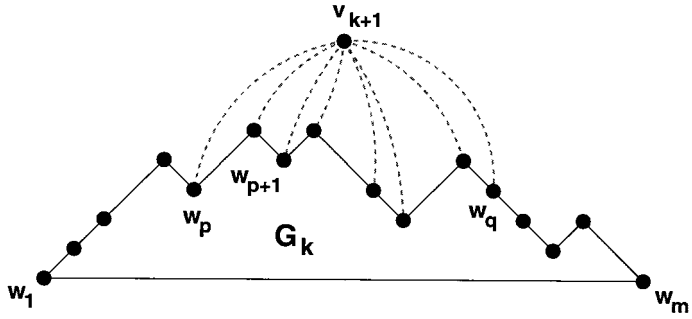


Figure 9.10 The subgraph G_k with v_{k+1} placed at $\mu(p, q)$.

We proceed by induction to show how a drawing with the properties may be computed. The drawing of G_3 satisfies the three properties. Now suppose that $k \geq 3$, and we have a drawing of G_k which satisfies the three properties. The canonical ordering of the vertices implies that the neighbors of v_{k+1} in G_k occur consecutively on the path P_k; suppose that they are $w_p, w_{p+1}, \ldots, w_q$. Note that the intersection of the line of slope $+1$ through w_p with the line of slope -1 through w_q is at a grid point $\mu(p, q)$ (since the Manhattan distance between two vertices is even). If we placed v_{k+1} at $\mu(p, q)$, then the resulting drawing would have no edge crossings, but perhaps the edge (w_p, v_{k+1}) overlaps with the edge (w_p, w_{p+1}), as in Fig. 9.10. This can be repaired by moving all $w_{p+1}, w_{p+2}, \ldots, w_m$ one unit to the right. It is also possible that the edge (v_{k+1}, w_q) overlaps the edge (w_{q-1}, w_q), so we move all the vertices $w_q, w_{q+1}, \ldots w_m$ a further unit to the right. This ensures that the newly added edges do not overlap with the edges of G_k. Now, we can place v_{k+1} safely at $\mu(p, q)$ as shown in Fig. 9.11 (note that $\mu(p, q)$ is still a grid point). The three required properties clearly hold.

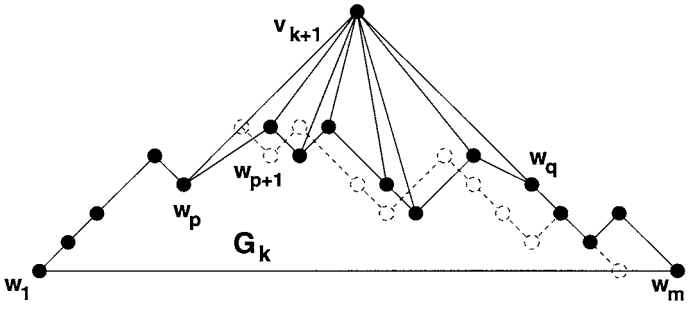


Figure 9.11 The subgraph G_k with v_{k+1} placed at $\mu(p, q)$ after moving the vertices $w_{p+1}, \ldots, w_{q-1}$ one unit and the vertices $w_q, \ldots, w_m$ two units to the right.

But there is a problem with merely moving the vertices on P_k: after the vertices $w_p, w_{p+1}, \ldots, w_m$ are moved to the right, the drawing of G_k may have edge crossings. We must use a more comprehensive strategy to repair the drawing of G_k; we must move even more vertices. Roughly speaking, when moving vertex w_i to the right, we will also move the vertices to the right below w_i. To this end we define a sequence of sets $M_k(w_1), M_k(w_2), \ldots, M_k(w_m)$ below. For a given sequence $\alpha(w_1), \alpha(w_2), \ldots, \alpha(w_m)$ of nonnegative integers, the vertices of G_k are moved succes-

sively as follows: first, all vertices of $M_k(w_1)$ are moved by $\alpha(w_1)$, then all vertices of $M_k(w_2)$ are moved by $\alpha(w_2)$, and so on.

The sets M_k are defined recursively as follows: $M_3(v_1) = \{v_1, v_2, v_3\}$, $M_3(v_2) = \{v_2, v_3\}$, $M_3(v_3) = \{v_3\}$. To compute M_{k+1} from M_k, note that if v_{k+1} is adjacent to $w_p, w_{p+1}, \ldots, w_q$, then P_{k+1} is $(w_1, w_2, \ldots, w_p, v_{k+1}, w_q, \ldots, w_m)$. For each vertex u on this path we must define $M_{k+1}(u)$. Roughly speaking, we add v_k to $M_k(w_i)$ if w_i is left of v_{k+1}, and do not alter $M_k(w_i)$ otherwise. More precisely, for $1 \leq i \leq p$, we define $M_{k+1}(w_i)$ to be $M_k(w_i) \cup \{v_{k+1}\}$, and $M_{k+1}(v_{k+1})$ is $M_k(w_{p+1}) \cup \{v_{k+1}\}$. For $q \leq j \leq M$, we define $M_{k+1}(w_j)$ to be $M_k(w_j)$. It is not difficult to show that the sets satisfy the following three properties.

(a) $w_j \in M_k(w_i)$ if and only if $i \leq j$.
(b) $M_k(w_m) \subset M_k(w_{m-1}) \subset \cdots \subset M_k(w_1)$.
(c) Suppose that $\alpha(w_1), \alpha(w_2), \ldots, \alpha(w_m)$ is a sequence of nonnegative integers and we apply algorithm *Move* to G_k; then the drawing of G_k remains planar.

These properties guarantee planarity.

Remarks

The proof of Theorem 9.2 constitutes an algorithm that can be implemented in linear time [10]. The area of the drawing that is produced is $2n - 4 \times n - 2$; this is asymptotically optimal [41], but the constants can be reduced to $n - 2 \times n - 2$ [8]. A sample drawing appears in Fig. 9.12.

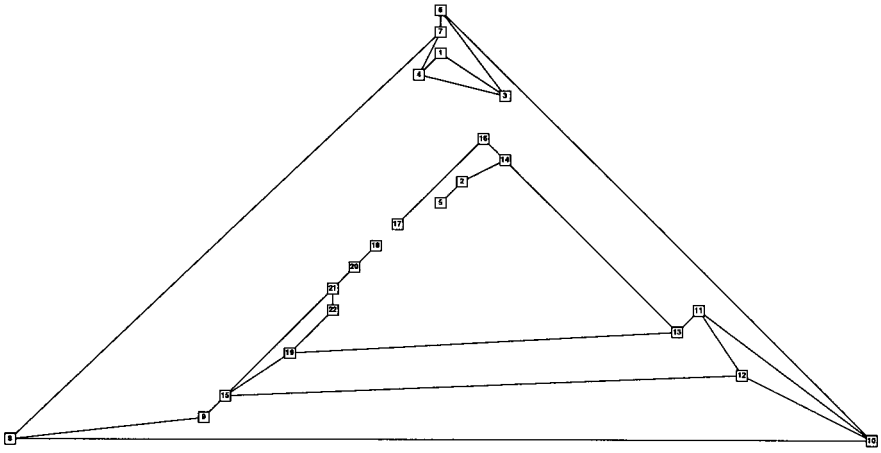


Figure 9.12 Sparse graph drawn with straight line edges.

There are some problems with the method. If the input graph is relatively sparse, then the number of "dummy" edges that must be added can be significant. These dummy edges have considerable influence over the final shape of the drawing but do not occur in the final drawing; the result may appear strange (see Fig. 9.12). Using a different ordering, the algorithm also works for 3-connected and even for biconnected planar graphs [27, 31]. This reduces the number of dummy edges considerably.

Although the drawings produced have relatively good vertex resolution, they are not very readable, because of two reasons: the angles between incident edges can get quite small, and the area is still too big in practice. These two significant problems can be overcome by allowing bends in the edges. Kant [31] modifies the algorithm described above to obtain a "mixed model algorithm;" this algorithm constructs a polyline drawing of a 3-connected planar graph such that the size of the minimal angle is at least $\frac{1}{d}\pi$, where d is the maximal degree of a vertex. Figure 9.13

shows the same graph as in Fig. 9.12 drawn with a modification of the mixed model algorithm for biconnected planar graphs by [27]. The grid size of the drawing in Fig. 9.12 is 38×19, whereas the size for the mixed model drawing is 10×8.

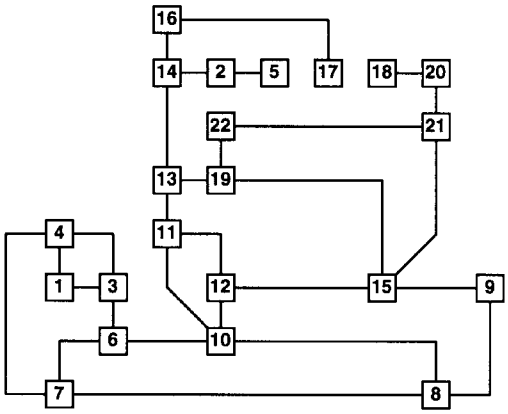


Figure 9.13 Sparse graph drawn with the mixed model algorithm.

Orthogonal Drawings

In polyline drawing, one aesthetic is to minimize the total number of bends. Generating an orthogonal drawing of a planar graph with the minimum number of bends is a NP-hard problem [25]. However, for the restricted problem where a fixed planar embedding is part of the input, there is a polynomial time algorithm [39]. In this subsection we will describe the transformation of this restricted bend minimization problem into a network flow problem.

Mathematical Preliminaries

Intuitively, the *orthogonal representation* of an orthogonal planar drawing of a graph describes its "shape;" it does not specify the length of the line-segments but determines the number of bends of each edge. The algorithm described in this section takes as input a planar embedding, represented as a planar map (that is a cyclic ordered list of the edges bounding each face). It produces as output an orthogonal representation with the minimum number of bends. From this, an actual drawing can be constructed via well-known compaction algorithms (see, e.g.,[29] or Chapter 23).

Formally, an orthogonal representation is a function H which assigns an ordered list $H(f)$ to each face f. Each element of $H(f)$ has the form (e, s, a), where e is an edge adjacent to f, s is a binary string, and $a \in \{90, 180, 270, 360\}$. If r is an element in $H(f)$ then $e[r]$, $s[r]$, and $a[r]$ denote the corresponding entries, that is, $r = (e[r], s[r], a[r])$. The list is ordered so that the edges $e[r]$ appear in clockwise order around the face f; thus, H consists of a planar map of the graph extended by the s- and a-arrays. The binary string $s[r]$ describes the shape of the edge $e[r]$, in the following way. The k-th bit in $s[r]$ represents the k-th bend while traversing the edge in face f in clockwise order: the entry is 0 if the bend produces a 90 degree angle (on the right hand side) and 1 otherwise. An edge without bend is represented by the zero-string ϵ. The number $a[r]$ represents the angle between the last line segment of edge $e[r]$ and the first line segment of edge $e[r']$, where r' follows r in the clockwise order around f. An orthogonal representation of

Fig. 9.14 is:

$$\begin{aligned} H(f_1) &= ((e_1, \varepsilon, 270), (e_5, 11, 90), (e_4, \varepsilon, 270), (e_2, 1011, 90)) \\ H(f_2) &= ((e_1, \varepsilon, 90), (e_6, \varepsilon, 180), (e_5, 00, 90)) \\ H(f_3) &= ((e_2, 0010, 90), (e_4, \varepsilon, 90), (e_6, \varepsilon, 90), (e_3, 0, 360), (e_3, 1, 90)) \ . \end{aligned}$$

The number of bends $B(H)$ of an orthogonal representation is

$$B(H) = \frac{1}{2} \sum_{f} \sum_{r \in H(f)} |s[r]| \ ,$$

where $|s|$ is the length of string s. Geometrical observations lead to the following lemma.

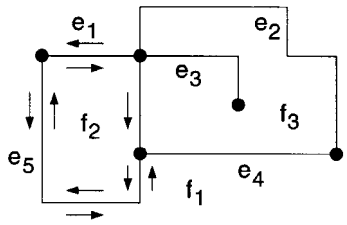


Figure 9.14 A graph and its orthogonal representation.

LEMMA 9.3 The function H is an orthogonal representation of an orthogonal planar drawing of a graph if and only if the properties (P1) to (P4) below are satisfied.

(P1) There is a plane graph with maximal degree four, whose planar map corresponds to the e-arrays in $H(f)$.

(P2) Suppose that r and r' are two elements in the orthogonal representation representing the same edge, that is, $e[r] = e[r']$. Then reversing the ordering of the elements in the string $s[r]$ and complementing each bit gives the string $s[r']$.

(P3) Suppose that ZEROS(s) denotes the number of 0's in string s, ONES(s) the number of 1's in string s, and

$$ROT(r) = \text{ZEROS}(s[r]) - \text{ONES}(s[r]) + (2 - a[r]/90) \ .$$

Then the faces described by H build rectilinear polygons, that is, polygons in which the lines are horizontal and vertical, if and only if

$$\sum_{r \in H(f)} ROT(r) = \begin{cases} -4 & \text{if } f \text{ is the outer face} \\ +4 & \text{otherwise} \end{cases} \ .$$

(P4) For all vertices v in the plane graph represented by H the following holds: the sum of the angles between adjacent edges to v given in the a-arrays of the corresponding elements in the lists $H(f)$ is 360.

The Transformation into a Network Flow Problem

Suppose that $G = (V, E)$ is a plane graph and P is a planar map, that is, for each face f of the embedding, $P(f)$ is a list of the edges on f in clockwise order. Denote the set of faces of G

defined by P by F. Let $\hat{V}$ denote the set of vertices of G with degree at most three. We define a network $N(P) = (S, A, u, c, b)$ with vertex set S, arc set A, capacity u, cost c and demand/supply b, as follows. The vertex set S consists of the vertices in $\hat{V}$ and the set F of faces. The arc set A is made up of two sets A_V and A_F, as follows:

- A_V contains the arcs (v, f), $v \in \hat{V}$, $f \in F$, for all vertices v that are endpoints of edges in $P(f)$. Each arc in A_V has infinite capacity and zero cost.
- A_F contains two arcs (f, g) and (g, f) for each pair $\{f, g\}$ of faces for which $P(f)$ and $P(g)$ share an arc. Moreover, A_F contains a self-loop (f, f) for each face f containing a bridge. Arcs in A_F have infinite capacity and unit cost.

The vertices $v \in \hat{V}$ supply $b(v) = 4 - \deg(v) > 0$ units of flow. The vertices $f \in F$ of the inner faces of size at most three ($|P(f)| \leq 3$), are supply vertices of value $b(f) = 4 - |P(f)| > 0$. The inner faces of size at least five and the outer face are demand vertices of value

$$b(f) = \begin{cases} -(|P(f)| + 4) & \text{if f is outer face} \\ -(|P(f)| - 4) & \text{otherwise .} \end{cases}$$

A graph and its network is shown in Fig. 9.15.

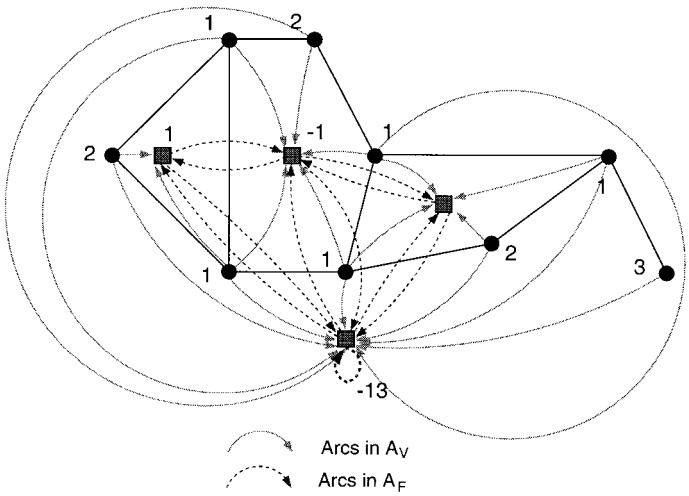


Figure 9.15 A graph and its network. Nodes are labeled with their supply and demand.

The value of the flow through the network is $z(P) = \Sigma_{s:b(s)>0} b(s) = -\Sigma_{t:b(t)<0} b(t)$.

The transformation defined above seems to be quite complex, but it can be intuitively interpreted as follows. Each unit of flow represents an angle of 90 degrees. For arcs in A_V the flow $x(v, f) + 1$ represents the sum of the angles between elements r in face f whose edges $e[r]$ are incident to v. The flow $x(f, g)$ in the arcs A_F represents the number of bends along those edges that separate the faces f and g and that create 90 degree angles in face f. The flow conservation rule on vertices $\hat{V}$ forces the sum of the angles around the vertices to be 360 degrees (see property (P4)). The conservation rule for vertices F arising from faces in P forces the created faces to be rectilinear polygons (see (P3)). Most importantly, the cost of a given flow in $N(P)$ is equal to the number of bends in the constructed orthogonal representation.

LEMMA 9.4 For any orthogonal planar drawing Π of a planar graph with given planar map P there exists an integer flow x in $N(P)$ with value $z(P)$ with cost equal to the number of bends in Π.

PROOF Let H be an orthogonal representation of G. The flow x in $N(P)$ can be constructed in the following way: for every arc (v, f) in A_v, the flow $x(v, f)$ arises from the a-entries in the elements in $H(f)$:

$$x(v,f) = \sum_{r \in R(v,f)} (a[r]/90 - 1), \text{ where}$$

$R(v, f) = \{r \in H(f) \mid e[r]$ and $e[r']$ are adjacent to vertex v and r precedes edge r' in $H(f)\}$.

For every arc (f, g) in A_F the flow is defined by the s-arrays of the elements in $H(f)$ associated with (f, g):

$$x(f,g) = \sum_{r \in R(f,g)} \text{ZEROS}(s[r]) \ ,$$

where $R(f, g) = \{r \in H(f) \mid e[r] \in P(f) \cap P(g)\}$. The flow $x(f, g)$ represents the angles of 90 degrees within region f along those edges separating f and g. In the case that $f = g$, we can see that $x(f, f)$ is equal to the number of bends along the bridges of f.

We have to show that the function f is, indeed, a flow with cost equal to the number of bends in H. Let us show that the conservation rule is satisfied at f. For vertices $v \in \hat{V}$ [using (P4)] we have

$$\sum_f x(v,f) - b(v) = \sum_f \sum_{r \in R(v,f)} (a[r]/90 - 1) - (4 - \deg(v)) = 0 \ .$$

For vertices $f \in F$ [using (P2) and (P3)] we have

$$\begin{aligned} \sum_g x(f,g) - \sum_s x(s,f) - b(f) &= \sum_g x(f,g) - \sum_g x(g,f) - \sum_v x(v,f) + |P(f)| \pm 4 \\ &= \sum_{r \in H(f)} (\text{ZEROS}(s[r]) - \text{ONES}(s[r]) - a[r]/90 + 1) \\ &\quad + |P(f)| \pm 4 \\ &= 0 \ . \end{aligned}$$

The costs of the flow are

$$\text{COST}(x) = \sum_{a \in A} c_a x_a = \sum_{(f,g) \in A_F} x(f,g) = \frac{1}{2} \sum_f \sum_{r \in H(f)} |s[r]| = B(H) \ ,$$

which is the number of bends in H.

LEMMA 9.5 For any integer flow x in $N(P)$ there exists an orthogonal planar drawing of a planar graph G with planar map P and with number of bends is equal to the cost of the flow x. The corresponding orthogonal representation can be constructed from x.

PROOF We construct the orthogonal representation as follows: the edge lists $H(f)$ are given by the planar map. For every arc (v, f) in A_V, let $r_1, \ldots, r_n$ be the elements in $R(v, f)$. We set $a[r_1] = 90(x(v, f) + 1)$, and $a[r_i] = 90$ for $2 \leq i \leq n$. For every pair (f, g) and (g, f) in A_F, let $r_1, \ldots, r_n$ be the elements in $R(f, g)$ and let $r'_1, \ldots, r'_n$ be the corresponding elements in $R(g, f)$. If $f = g$, then r_i and r'_i are the pairs of elements in $H(f)$ associated with both sides of the same bridge. We set

$$\begin{aligned} s[r_1] &= \begin{cases} 0^{x(f,g)} & \text{if } f = g \\ 0^{x(f,g)} 1^{x(g,f)} & \text{otherwise} \end{cases} \ , \\ s[r'_1] &= \begin{cases} 1^{x(g,f)} & \text{if } f = g \\ 0^{x(g,f)} 1^{x(f,g)} & \text{otherwise} \end{cases} \ , \end{aligned}$$

and $s[r_i] = s[r'_i] = \epsilon$ for $2 \leq i \leq n$.

The properties (P1) to (P4) are satisfied by our defined e, s- and a-arrays. The number of bends in the orthogonal representation is given by

$$\begin{aligned} B(H) &= \frac{1}{2}\sum_{f}\sum_{r\in H(f)} |s[r]| \\ &= \frac{1}{2}\sum_{(f,g)\in A_F} (x(f,g) + x(g,f)) \\ &= \sum_{a\in A} c(a)x(a) \\ &= \text{COST}(x) \ . \end{aligned}$$

In a network with integral costs and capacities there always exists a min-cost flow with integer flow values. Hence, we have the following theorem.

THEOREM 9.3 *Let P be a planar map of a planar graph G with maximal degree 4. The minimal number of bends in an orthogonal planar drawing with planar map P is equal to the cost of a min-cost flow in $N(P)$ with value $z(P)$. The orthogonal representation H of any orthogonal planar drawing of G can be constructed from the integer-valued min-cost flow in $N(P)$.*

The algorithm for constructing an orthogonal drawing is as follows. First construct the network $N(P)$ in time $O(|S|+|A|)$. A min-cost flow can be computed in time $O(n^2 \log n)$ (see Chapter 7 for the theory of network algorithms). The orthogonal representation can be constructed easily from the given flow. The length of the line segments can be generated by any compaction algorithm (see [29] or Chapter 23).

Remarks

The drawings produced by the algorithm described above look very pleasant (see Fig. 9.9). One drawback of the algorithm is that it is restricted to planar graphs with maximal degree four. In fact the methods described in this section can be extended to planar graphs of high degree ([2, 23, 32]). Figure 9.16 shows a drawing of a graph of maximum degree 6; this was obtained using the planarization above as well as the method of [32].

There are also orthogonal drawing methods for graphs which are not planar. These methods take no account of the number of crossings but try to keep the number of bends and the drawing area small [3, 36]. An experimental comparison of various orthogonal graph drawing algorithms is given in [18].

9.4 Graph Drawing in Three Dimensions

Three dimensional scientific visualization is now commonplace. Research into three-dimensional graph drawing has begun [38]. Most of this work has proceeded in an *ad hoc* manner; here we describe two fundamental algorithms. Both produce grid drawings with no edge crossings. The first, from [11], simply produces a straight-line drawing of an n vertex graph in a box of dimensions $O(n) \times O(n) \times O(n)$. The second, from [21], takes a graph of maximum degree at most 6 as input and produces an orthogonal grid drawing with at most 3 bends per edge.

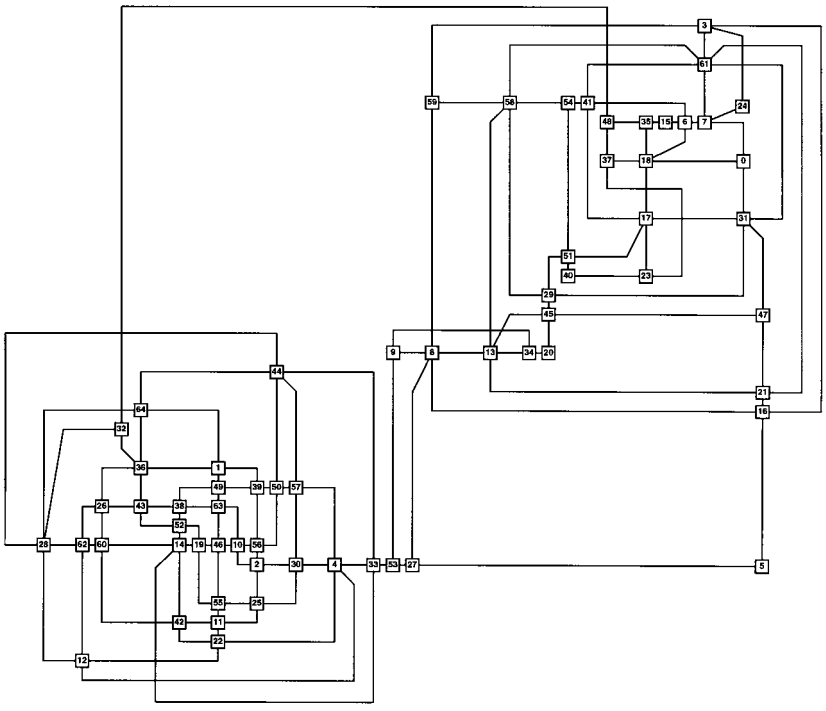


Figure 9.16 A graph drawn using network flow techniques.

Straight-Line Drawings in Three Dimensions

In this section we show that every graph with n vertices admits a straight-line crossing-free drawing with dimensions $n \times 2n \times 2n$.

Firstly note that a larger straight-line crossing-free drawing can be obtained using the *moment curve* $M(t) = (t, t^2, t^3)$. This curve has the well known property that no four points on the curve are coplanar. If the vertices of G are $v_1, v_2, \ldots, v_n$ and we place vertex v_t at $M(t)$, then no two edges of G are coplanar and thus, cannot cross. This drawing uses a large volume $(n \times n^2 \times n^3)$; next we show how to reduce this volume.

The method uses a prime number p with $n \leq p < 2n$. The existence of such a prime is guaranteed by Bertram's Theorem, and one can be found with a simple prime number sieve. The main observation is that we can use the *modulo p moment curve*

$$M_p(t) = \big(t \bmod p, t^2 \bmod p, t^3 \bmod p\big)$$

instead of the usual moment curve. We place vertex v_t at $M_p(t)$. For $t = 1, 2, \ldots, n$, $0 < t^k \bmod p < p < 2n$; thus, the volume of this drawing is bounded by $n \times 2n \times 2n$. It is a simple exercise to show that four distinct vertices cannot share a plane. Thus, we have the following theorem.

THEOREM 9.4 *Every graph with n vertices has a three-dimensional straight-line crossing-free grid drawing graph with volume $n \times 2n \times 2n$.*

Note that the volume bound of Theorem 9.4 is asymptotically optimal. Suppose that a drawing of the complete graph on n vertices lies strictly between the planes $x = 0$ and $x = \frac{n}{4}$. Then for some integer m, the plane $x = m$ contains at least 5 vertices. The straight-line edges between these 5 vertices must also lie in the plane $x = m$. But the subgraph induced by the 5 vertices is complete and thus, nonplanar. We can deduce the following theorem.

THEOREM 9.5 *Suppose that D is a three-dimensional straight-line crossing-free grid drawing of the complete graph G on n vertices, and D uses volume $X \times Y \times Z$. Then each of X, Y, Z is $\Omega(n)$.*

Three-Dimensional Orthogonal Grid Drawings

A graph that has a three-dimensional orthogonal drawing with no edge crossings clearly has maximum degree 6. The edge set of such a graph has a useful decomposition into three sets of cycles, as described by the following lemma.

LEMMA 9.6 Suppose that $G = (V, E)$ is a graph of maximum degree at most 6. Then there is an orientation $G' = (V, E')$ of G with subgraphs $C_{red} = (V_{red}, E_{red})$, $C_{blue} = (V_{blue}, E_{blue})$, and $C_{green} = (V_{green}, E_{green})$, such that

- $E' = E_{red} \cup E_{blue} \cup E_{green}$, and
- $E_\alpha \cap E_\beta = \emptyset$ for $\alpha, \beta \in \{red, blue, green\}$.
- Each vertex of C_α has in-degree one and out-degree one in C_α, for $\alpha \in \{red, blue, green\}$.

The decomposition can be obtained in $O(n^{1.5})$ time.

PROOF The existence of the orientation and the coloring follows from the fact that every regular graph of even degree is *2-factorable* [28]. The coloring can be found in time $O(n^{1.5})$ by using a matching algorithm for bipartite graphs.

Each of the subgraphs C_{red}, C_{blue}, and C_{green} in Lemma 9.6 consists of a set of disjoint directed cycles; we call these the *red cycles, blue cycles,* and *green cycles,* respectively. These cycles are used in the proof of the following theorem.

THEOREM 9.6 *Suppose that G is a graph of maximum degree at most 6. Then there is an orthogonal grid drawing of G with at most 3 bends per edge. The drawing can be computed in time $O(n^{1.5})$.*

The remainder of this section describes the proof of Theorem 9.6. The construction uses the preprocessing of the input graph $G = (V, E)$ as described in Lemma 9.6. Thus, we assume that G is a directed graph and the subgraphs C_{red}, C_{blue}, and C_{green} have been computed.

The vertices are listed in an arbitrary order $v_1, v_2, \ldots, v_n$ and v_i is placed at the location $p_i = (3i, 3i, 3i)$. If $1 \le i < j \le n$ then the isothetic cube with p_i and p_j at opposite corners is denoted by $C(i, j)$. The cube is illustrated in Fig. 9.17(a).

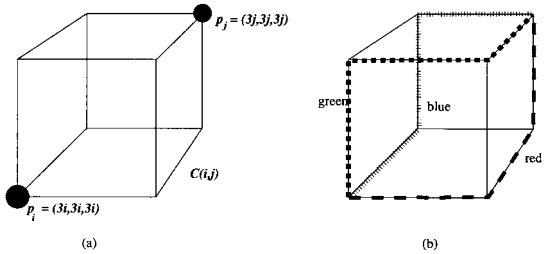


Figure 9.17 The cube $C(i, j)$.

Three disjoint routes can be defined along the edges of the cube $C(i,j)$ as illustrated in Fig. 9.17(b); these can be colored in three colors corresponding to the three colors on the arcs of G, as follows.

- *Red*: $p_i = (3i, 3i, 3i) \rightarrow (3j, 3i, 3i) \rightarrow (3j, 3j, 3i) \rightarrow (3j, 3j, 3j) = p_j$
- *Blue*: $p_i = (3i, 3i, 3i) \rightarrow (3i, 3j, 3i) \rightarrow (3i, 3j, 3j) \rightarrow (3j, 3j, 3j) = p_j$
- *Green*: $p_i = (3i, 3i, 3i) \rightarrow (3i, 3i, 3j) \rightarrow (3j, 3i, 3j) \rightarrow (3j, 3j, 3j) = p_j$

Note that each of these routes has only two bends. This coloring scheme for routes of edges of the cubes will be used to define routes for the edges of the graph. If the arc (v_i, v_j) of the graph has color $\alpha \in \{red,\ blue,\ green\}$, then it will approximately follow the route of $C(i,j)$ of color α.

Before defining the precise routes for the arcs, some intuitive feel for the reasons that they do not cross may be helpful. The way in which edges of different cubes intersect with each other is significant. The following lemma is easy to prove.

LEMMA 9.7 Suppose that $i < j$ and $k < \ell$, $i < k$, and $j \neq \ell$. Let s be a point on an edge of $C(i,j)$, and let t be a point on an edge of $C(k,\ell)$. Then the Euclidean distance between s and t is at least 3.

Lemma 9.7 implies that if the edges of two cubes $C(i,j)$ and $C(k,\ell)$ intersect then either $j = k$, $i = k$, or $j = \ell$. Intuitively, this means that if $(v_i, v_j) \in A$ is routed along edges of $C(i,j)$ and $(v_k, v_\ell) \in A$ is routed along edges of $C(k,\ell)$, and these routes intersect, then the edge (v_i, v_j) shares an endpoint with the edge (v_k, v_ℓ). Further, note from Lemma 9.7 that even if the route of $(v_i, v_j) \in A$ is "offset a little" from the edges of $C(i,j)$ and the route of $(v_k, v_\ell) \in A$ is "offset a little" from the edges of $C(k,\ell)$, and these routes intersect, then they are incident to a common vertex.

Next we define the precise routes for the red edges of G; the routes for other colors are similar. Basically, if (v_i, v_j) is a red edge then it follows the red route around the cube $C(i,j)$. In most cases it follows this route exactly. However, in some cases it must follow a path that is close to the red route on $C(i,j)$, but offset a little to avoid crossing another red edge.

The red edges form a set of disjoint cycles, and we shall route each cycle separately. Suppose that $(w_0, w_1, \ldots, w_{k-1})$ is a red cycle. For each vertex w_c on this cycle, there is an integer $f(w_c)$ such that $w_c = v_{f(w_c)}$, that is, w_c has been placed at $(3f(w_c), 3f(w_c), 3f(w_c))$. The function f orders the vertices of the cycle by their distance from the origin. An arc (w_c, w_{c+1}) on the cycle falls into one of four categories, defined by the order imposed by f (here the subscript addition is modulo k):

- *Normal increasing arcs*: $f(w_c) < f(w_{c+1})$ and $f(w_{c+1}) < f(w_{c+2})$.
- *Normal decreasing arcs*: $f(w_c) > f(w_{c+1})$ and $f(w_{c+1}) > f(w_{c+2})$.
- *Special arcs entering a local minimum*: $f(w_c) > f(w_{c+1})$ and $f(w_{c+1}) < f(w_{c+2})$.
- *Special arcs entering a local maximum*: $f(w_c) < f(w_{c+1})$ and $f(w_{c+1}) > f(w_{c+2})$.

A normal red arc (w_c, w_{c+1}) (increasing or decreasing) is routed exactly along the red path of edges of the cube $C(f(w_c), f(w_{c+1}))$. Note that each normal arc has two bends. The special arcs are routed near the red path; for two of the cube edges, they are offset by a distance of one. Each has three bends. If $a = 3f(w_c)$ and $b = 3f(w_{c+1})$, then the routes for special red arcs are:

- *Special arcs entering a local minimum* [Fig. 9.18(a)]: $(a,a,a) \rightarrow (a,a,b-1) \rightarrow (a,b,b-1) \rightarrow (b,b,b-1) \rightarrow (b,b,b)$.
- *Special arcs entering a local maximum* [Fig. 9.18(b)]: $(a,a,a) \rightarrow (b+1,a,a) \rightarrow (b+1,b,a) \rightarrow (b+1,b,b) \rightarrow (b,b,b)$.

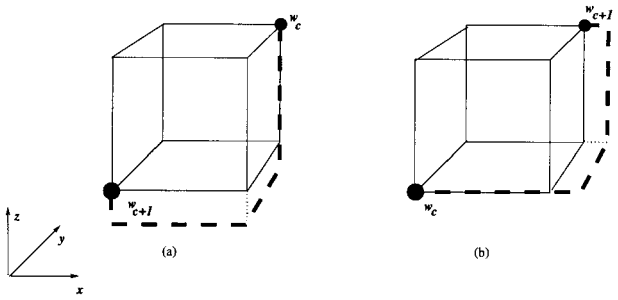


Figure 9.18 (a) Local minimum route, (b) local maximum route.

Now suppose that two arcs of different colors cross. Using Lemma 9.7 one can see that the crossing is between arcs which share a vertex, and it must be inside a $2 \times 2 \times 2$ box centered the shared vertex. Note a red arc always enters or leaves a vertex in a plane parallel to the xz plane, while a blue arc always enters or leaves a vertex in a plane parallel to the xy plane; thus, a red–blue crossing is impossible. Continuing this argument, we can deduce that two arcs of different colors cannot cross.

Next suppose that two arcs of the same color cross. We can assume that these arcs do not share both endpoints, since we will only draw one of a pair of multiple edges in the final drawing. Thus, by Lemma 9.7, the arcs share precisely one vertex. Lemma 9.6 ensures that one arc is entering a shared vertex, and one is leaving the shared vertex; suppose that the arcs are (v_i, v_j) and (v_j, v_k). There are essentially two cases to consider: $i < j < k$, and $j < i < k$. In the first case, the cubes $C(i, j)$ and $C(j, k)$ intersect only at the location of v_j, as in Fig. 9.19(a). Here (v_i, v_j) is a normal increasing arc, and (v_j, v_k) is either a normal increasing arc or special arc entering a local maximum. It is easy to see that no crossing is possible. In the second case, $C(j, i)$ and $C(j, k)$ overlap as in Fig. 9.19(b); (v_i, v_j) is a special arc entering a local minimum and (v_j, v_k) is either a normal increasing arc or a special arc entering a local maximum. Again, it is easy to observe that their routes are disjoint.

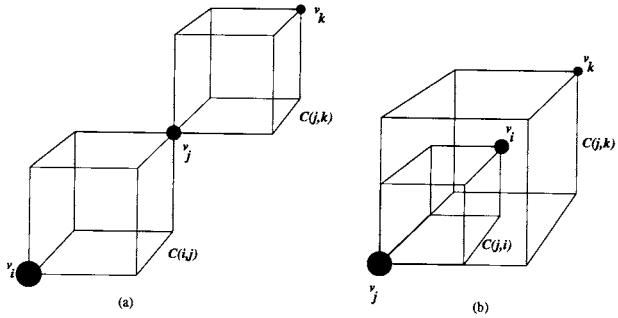


Figure 9.19 (a) Cubes intersect only at a vertex. (b) Cubes overlap.

9.5 Research Issues and Summary

The field of graph drawing is still developing and there are many fundamental questions to be resolved. Here we list a few.

- In the exact planarization algorithm, it is essential to solve the separation problem for the Kuratowski inequalities for nonintegral vectors $\bar{x}$ (see "Planarization"). It is not known whether this problem is NP-hard or not.

- The grid size for planar straight-line drawings is bounded below; there is an n-vertex plane graph G such that, for any straight-line grid drawing of G, each dimension of the grid is at least $\lfloor \frac{2(n-1)}{3} \rfloor$ even if the other dimension is allowed to be unbounded [9, 13]. It is conjectured that this lower bound is almost achievable; more precisely, that every planar graph with n vertices has a 2-dimensional straight-line drawing on a grid of size $\lceil \frac{2n}{3} \rceil \times \lceil \frac{2n}{3} \rceil$.
- There are many algorithms for creating planar orthogonal grid drawings in 2 dimensions using area $O(n^2)$ and a constant number of bends per edge. For binary trees, this result can be improved: $O(n \log(n))$ area with no bends at all [12]. However, it is not known whether every binary tree has an orthogonal planar drawing using linear area and no bends. This question is open even for three dimensions.
- The trade-off between volume and numbers of bends in three dimensions needs further exploration. Does every graph of maximum degree 6 admit an orthogonal drawing with at most 2 bends per edge? Does every graph of maximum degree 6 admit an orthogonal drawing with less than 7 bends per edge and volume $O(\sqrt{n}) \times O(\sqrt{n}) \times O(\sqrt{n})$?

9.6 Defining Terms

Edge crossing: Two nonincident edges *cross* in a graph drawing if their geometric representations intersect. The number of crossings in a graph drawing is the number of pairs of edges which cross.

Graph drawing: A geometric representation of a graph is a *graph drawing.* Usually, the representation is in either 2- or 3-dimensional Euclidean space, where a vertex v is represented by a point $p(v)$ and an edge (u, v) is represented by a simple curve whose endpoints are $p(u)$ and $p(v)$.

Grid drawing: A *grid drawing* is a graph drawing in which each vertex is represented by a point with integer coordinates.

Orthogonal drawing: An *orthogonal drawing* is a graph drawing in which each edge is represented by a polyline, each segment of which is parallel to a coordinate axis.

Planar graph: A graph drawing is *planar* if it has no edge crossings. A graph is *planar* if it has a planar drawing.

Planarization: Informally, *planarization* is the process of transforming a graph into a planar graph. More precisely, the transformation involves either removing edges (*planarization by edge removal*), or replacing pairs of nonincident edges by 4-stars, as in Fig. 9.20 (*planarization by adding crossing vertices*). In both cases, the aim of planarization is to make the number of operations (either removing edges or replacing pairs of nonincident edges by 4-stars) as few as possible.

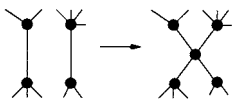


Figure 9.20 Planarization by crossing vertices.

Straight-line drawing: A *straight-line drawing* is a graph drawing in which each edge is represented by a straight line segment.

References

[1] Alberts, D., Gutwenger, C., Mutzel, P., and N‰her, S., AGD–Library: A library of algorithms for graph drawing. In *WAE '97 (Proc. on the Workshop on Algorithm Engineering),* http://www.dsi.unive.it/~wae97/ Italiano, G.F. and Orlando, S., Eds., Venice, Italy, Sep. 1997, 11–13, 1997.

[2] Batini, C., Nardelli, E., and Tamassia, R., A layout algorithm for data-flow diagrams. *IEEE Trans. Softw. Eng.,* SE-12(4), 538–546, 1986.

[3] Biedl, T. and Kant, G., A better heuristic for orthogonal graph drawings. In *Proc. 2nd Annu. European Sympos. Algorithms (ESA '94),* volume 855 of *Lecture Notes in Computer Science,* 24–35. Springer-Verlag, 1994.

[4] Bondy, J.A. and Murty, U.S.R., *Graph Theory with Applications.* North-Holland, New York, NY, 1976.

[5] Brandenburg, F.J., Ed., *Graph Drawing (Proc. GD '95),* volume 1027 of *Lecture Notes in Computer Science.* Springer-Verlag, 1996.

[6] Brandenburg, F.J., Himsolt, M., and Rohrer, C., An experimental comparison of force-directed and randomized graph drawing algorithms. In *Graph Drawing (Proc. GD '95),* volume 1027 of *Lecture Notes in Computer Science,* 76–87. Brandenburg, F.J., Ed., Springer-Verlag, 1996.

[7] Cai, J., Han, X., and Tarjan, R.E., An $O(m \log n)$-time algorithm for the maximal planar subgraph problem. *SIAM J. Comput.,* 22, 1142–1162, 1993.

[8] Chrobak, M. and Kant, G., Convex grid drawings of 3-connected planar graphs. *International Journal on Computational Geometry and Applications,* 7(3), 211–224, 1997.

[9] Chrobak, M. and Nakao, S., Minimum width grid drawings of planar graphs. In *Graph Drawing (Proc. GD '94),* volume 894 of *Lecture Notes in Computer Science,* 104–110. Tamassia, R. and Tollis, I.G., Eds., Springer-Verlag, 1995.

[10] Chrobak, M. and Payne, T.H., A linear time algorithm for drawing a planar graph on a grid. *Information Processing Letters,* 54, 241–246, 1995.

[11] Cohen, R.F., Eades, P., Lin, T., and Ruskey, F., Three-dimensional graph drawing. *Algorithmica,* 17(2), 199–208, 1996.

[12] Crescenzi, P., Battista, G.Di, and Piperno, A., A note on optimal area algorithms for upward drawings of binary trees. *Comput. Geom. Theory Appl.,* 2, 187–200, 1992.

[13] De Fraysseix, H., Pach, J., and Pollack, R., How to draw a planar graph on a grid. *Combinatorica,* 10(1), 41–51, 1990.

[14] Di Battista, G., Ed., *Graph Drawing (Proc. GD '97).* Volume 1353 of *Lecture Notes in Computer Science.* Springer-Verlag, 1997.

[15] Di Battista, G., Garg, A., Liotta, G., Parise, A., Tamassia, R., Tassinari, E., Vargiu, F., and Vismara, L., Drawing directed acyclic graphs: an experimental study. In *Graph Drawing (Proc. GD '96),* volume 1190 of *Lecture Notes in Computer Science,* 76–91. North, S., Ed., Springer-Verlag, 1997.

[16] Di Battista, G., Eades, P., Tamassia, R., and Tollis, I., *Graph Drawing: Algorithms for Geometric Representations of Graphs.* Prentice Hall, Englewood Cliffs, NJ, 1998. (to appear).

[17] Di Battista, G., Eades, P., Tamassia, R., and Tollis, I.G., Algorithms for drawing graphs: an annotated bibliography. *Comput. Geom. Theory Appl.,* 4, 235–282, 1994.

[18] Di Battista, G., Garg, A., Liotta, G., Tamassia, R., Tassinari, E., and Vargiu, F., An experimental comparison of three graph drawing algorithms. In *Proc. 11th Annu. ACM Sympos. Comput. Geom.,* 306–315, 1995.

[19] Eades, P., Lin, T., and Lin, X., Two tree drawing conventions. *International Journal of Computational Geometry and Applications,* 3(2), 133–153, 1993.

[20] Eades, P. and Sugiyama, K., How to draw a directed graph. *Journal of Information Processing,* 424–437, 1991.

[21] Eades, P., Symvonis, A., and Whitesides, S., Two algorithms for three dimensional orthogonal graph drawing. In *Graph Drawing,* (Proc. gD '96), volume 1190 of *Lecture Notes in Computer Science,* North, S., Ed., 139–154, Springer-Veralg, 1996.

[22] Fary, I., On straight lines representation of planar graphs. *Acta Sci. Math. Szeged.,* 11, 229–233, 1948.

[23] Fßmeier, U. and Kaufmann, M., Drawing high degree graphs with low bend numbers. In *Graph Drawing (Proc. GD '95),* volume 1027 of *Lecture Notes in Computer Science,* 254–266. Brandenburg, F.J., Ed., Springer-Verlag, 1996.

[24] Garey, M.R. and Johnson, D.S., Crossing number is NP-complete. *SIAM J. Algebraic Discrete Methods,* 4(3), 312–316, 1983.

[25] Garg, A. and Tamassia, R., On the computational complexity of upward and rectilinear planarity testing. In *Graph Drawing (Proc. GD '94),* volume 894 of *Lecture Notes in Computer Science,* 286–297. Tamassia, R. and Tollis, I.G., Eds., Springer-Verlag, 1995.

[26] Goldschmidt, O. and Takvorian, A., An efficient graph planarization two-phase heuristic. *Networks,* 24(2), 69–73, 1994.

[27] Gutwenger, C. and Mutzel, P., Grid embedding of biconnected planar graphs. Technical Report, Max-Planck-Institut Informatik, Saarbrcken, Germany, 1998.

[28] Holton, D. and Sheehan, J., *The Petersen Graph,* volume 7 of *Australian Math. Soc. Lecture Notes Series.* Cambridge UP, 1993.

[29] Hsueh, M.Y., *Symbolic Layout and Compaction of Integrated Circuits.* Ph.D. Thesis, University of California at Berkeley, 1979.

[30] Jnger, M. and Mutzel, P., Maximum planar subgraphs and nice embeddings: Practical layout tools. *Algorithmica, Special Issue on Graph Drawing,* 16(1), 33–59, 1996.

[31] Kant, G., Drawing planar graphs nicely using the *lmc*-ordering. In *Proc. 33th Ann. IEEE Symp. on Found. of Comp. Sci.,* 101–110, Pittsburgh, PA, 1992.

[32] Klau, G. and Mutzel, P., Quasi-orthogonal drawing of planar graphs. Technical Report, Max-Planck-Institut Informatik, MPI-I-98-1-013, Saarbrcken, Germany, 1998.

[33] Liu, P.C. and Geldmacher, R.C., On the deletion of nonplanar edges of a graph. In *Proc. 10th. S-E Conf. on Comb., Graph Theory, and Comp.,* 727–738, Boca Raton, FL, 1977.

[34] Mehlhorn, K. and Mutzel, P., On the embedding phase of the Hopcroft and Tarjan planarity testing algorithm. *Algorithmica,* 16(2), 233–242, 1996.

[35] North, S., Ed., *Graph Drawing (Proc. GD '96),* volume 1190 of *Lecture Notes in Computer Science.* Springer-Verlag, 1997.

[36] Papakostas, A. and Tollis, I.G., Improved algorithms and bounds for orthogonal drawings. In *Graph Drawing (Proc. GD '94),* volume 894 of *Lecture Notes in Computer Science,* 40–51. Tamassia, R. and Tollis, I.G., Eds., Springer-Verlag, 1995.

[37] Reingold, E. and Tilford, J., Tidier drawing of trees. *IEEE Trans. Softw. Eng.,* SE-7(2), 223–228, 1981.

[38] Robertson, G.G., Mackinlay, J.D., and Card, S.K., Cone trees: Animated 3D visualizations of hierarchical information. In *Proc. ACM Conf. on Human Factors in Computing Systems (CHI),* 189–193, 1991.

[39] Tamassia, R., On embedding a graph in the grid with the minimum number of bends. *SIAM J. Comput.,* 16(3), 421–444, 1987.

[40] Tutte, W.T., How to draw a graph. *Proceedings London Mathematical Society,* 13(3), 743–768, 1963.

[41] Valiant, L., Universality considerations in VLSI circuits. *IEEE Trans. Comput.,* C-30(2), 135–140, 1981.

[42] Wagner, K., Bemerkungen zum Vierfarbenproblem. *Jahresbericht der Deutschen Mathematiker Vereinigung,* 46, 26–32, 1936.

Further Information

A comprehensive bibliography of research published up to 1994 appears in [17]. A first book on graph drawing will appear in [16]. The proceedings of the annual international Graph Drawing Symposia are published in the *Lecture Notes in Computer Science* series by Springer; see, for example, [14, 5, 35]. There exist various software packages for graph drawing, see, for example, [1].

10

On-line Algorithms: Competitive Analysis and Beyond

Steven Phillips
AT&T Labs – Research

Jeffery Westbrook
AT&T Labs – Research

10.1 Introduction

The field of **on-line algorithms** addresses problems in which an algorithm is handicapped by lack of knowledge of the future. An on-line algorithm receives a sequence of inputs, and must process each input in turn, without detailed knowledge of later inputs. Typically the on-line algorithm is managing a system with persistent state, and the changes made to the state affect the cost of processing future inputs.

An example is the following idealization of load balancing on a multiprocessor computer. A sequence of jobs appear on-line, each having a known size. Each job must be irrevocably assigned on arrival to one of n machines. The load on a machine is the sum of the sizes of jobs assigned to it. We would prefer the assignments to balance the load on the machines, and in particular to minimize the maximum machine load. Graham showed that the algorithm that assigns each job to the machine that is currently least loaded achieves a maximum load that is never more than $2 - 1/n$ times greater than the best possible assignment of the jobs [38]. This is an example of what is now called **competitive analysis**. In competitive analysis, an on-line algorithm is evaluated by comparing its performance to the best that could have been achieved if all the inputs had been known in advance.

On-line algorithms have become an active topic of research. Competitive analysis has been applied to

a multitude of **on-line problems,** and the notion of competitive analysis has been refined in a number of ways. This chapter gives an overview of the scope of research in on-line algorithms. The reader will be introduced to the major topics and problems that have been studied to date, as well as a variety of methods and models that can be used to analyze and solve on-line problems.

We start with a study of the ski rental problem in Section 10.2. This simple problem arises in a number of contexts, and it provides a natural setting in which to start exploring the design and analysis of on-line algorithms. The basic models and methods of Section 10.2 are formalized in Section 10.3, and a review of some more advanced models is given in Section 10.4, using paging, an archetypal on-line problem, as the backdrop. Section 10.5 describes some general purpose algorithms that are applicable to large classes of on-line problems. Lastly, Section 10.6 is a selective compendium of topics in on-line algorithms that is intended to serve as a starting point for the investigation of old or new on-line problems.

10.2 The Ski Rental Problem

We begin with a classic example of an on-line problem, "ski rental," first posed in this form by Larry Rudolph. Suppose you decide to learn to ski. After each trip, you will make an irrevocable decision whether to stop skiing or continue learning, and you have no idea in advance what your decision will be. Skiing is an equipment-intensive sport, of course, and before each trip you have two options: rent the equipment at x dollars per day, or buy the equipment for a grand total of y dollars. For convenience we assume that $y = cx$ for some integer $c > 1$. Before each trip to the mountain you have to decide whether to rent or buy. You would like to spend as little money on equipment as possible. Buying equipment before even taking one lesson could be a terrible waste, if you decide to stop after the first trip. On the other hand, if you take many trips then at some point it will have been cheaper to buy than rent. At what point should you stop renting and buy?

The ski rental problem is relevant not only to the management of sporting equipment. It is applicable in a wide variety of resource allocation problems. For example, consider power management in a laptop computer. A typical laptop powers down the hard drive when it is not in use, because a running hard drive consumes battery power. It takes a significant amount of power and time, however, to start the hard drive running again once it has been powered down. If the user of the laptop doesn't use the hard drive for a while, how long should the laptop wait before powering it down? This is just the ski rental problem—running the drive is renting and powering down is buying (since a one-time cost is incurred when the drive is restarted).

A second application is spinning vs. blocking at a lock in a parallel program [45]. If a thread blocks at the lock, it does not consume valuable computing cycles while waiting for some other thread to free the lock. However, a context switch is needed to restart the thread when the lock is freed, costing some cycles. If instead the thread spins, waiting for the lock to be freed, then it can resume running as soon as the lock is freed, but it consumes cycles while it waits. The problem of deciding how long to spin before blocking is just ski rental — spinning is renting and blocking is buying. Other applications include snoopy caching [46] and IP-over-ATM networks [55].

Let's head back to the slopes. There is some number t of ski trips that you will take before stopping (or moving on to that great powder basin in the sky). Suppose you are told t in advance. Then it is easy to decide whether to rent or buy. If $tx \leq y$, you should rent, and otherwise you should buy right at the start. This is the **off-line** ski rental problem. The solution to the off-line problem is called the **optimal solution** and the cost of the optimal solution is called the **optimal cost.** The optimal cost is tx for $t \leq c$ and y for $t > c$.

In the *on-line* problem, the rent-or-buy decision must be made prior to each trip, without knowledge of t. All that can be determined after i trips is that $t \geq i$. Consider the following strategy: rent until $c = y/x$ trips have occurred, and then buy if a $(c + 1)$st trip happens. How well does this strategy do? If $t \leq c$, then it is optimal— the minimum possible amount is spent. Suppose $t > c$. Then the cost is exactly twice

the optimal cost. Therefore, although this strategy can be optimal in some situations, in the worst case it incurs a cost twice the optimal. This worst-case ratio between the cost incurred by the on-line strategy and the optimal cost is called the **competitive ratio**.

One may ask, is there a better strategy given the rules of the game? A strategy is simply a value k: the number of times to rent before buying. The on-line cost using such a strategy is tx for $t \leq k$ and $kx + y$ for $t > k$. Clearly, there is no value of k that is guaranteed to achieve the optimal cost in all cases. Indeed, any k is nonoptimal for the case $t = k + 1$, since $kx + y \geq (k + 2)x > (k + 1)x = tx$. This is typical of on-line problems. Without knowledge of the future, there is no on-line algorithm that is always optimal.

Furthermore, it is not hard to see that no strategy can have a competitive ratio that is lower than 2. The worst-case ratio between the on-line cost and the optimal cost is

$$\max\left\{\frac{kx+y}{tx}, \frac{kx+y}{y}\right\}.$$

If $k = 0$, then for $t = 1$ the first ratio is y/x, which is at least 2 by assumption. Otherwise, if $kx \leq y$, then the ratio is at least 2 when $t = k$ (the first ratio in the max), and if $kx > y$ the ratio is at least 2 when $t > k$ (the second ratio in the max).

A skier who abides by the rules of the game might be termed a cautious skier. She can be sure that she never pays more than twice what she had to. However, skiing is hardly a sport that attracts cautious people, so we'll change the rules to add an element of daring. (First we had the "green circle" model, next comes the "blue square" model.)

You are now allowed a **randomized** purchasing strategy. Your decision of whether to buy or rent might be made by a coin flip, or based on your horoscope. Your goal is now to minimize the *expected* ratio to the off-line cost, rather than the worst-case ratio.

Let strategy A_i be to rent $i - 1$ times, then buy on the ith ski trip. A randomized strategy can be summarized by a probability distribution π, where π_i is the probability you use algorithm A_i. Now assume we want the expected ratio to be at most α. Then if we ski exactly t times, as we noted earlier the optimal cost is tx for $t \leq c$ and y for $t > c$. We must therefore choose π to satisfy

$$\begin{aligned}
\mathbf{E}[\text{on-line cost}] &\leq \alpha t x \quad t \leq c\,, \\
\mathbf{E}[\text{on-line cost}] &\leq \alpha y \quad t > c\,.
\end{aligned}$$

If we make these inequalities equalities and solve the resulting equations, we get

$$\pi_i = \begin{cases} \frac{\alpha-1}{c}\left(\frac{c}{c-1}\right)^i & i = 1 \ldots c \\ 0 & \text{otherwise}\,. \end{cases} \tag{10.1}$$

Finally to solve for α we use $\sum \pi_i = 1$, giving

$$\alpha = \frac{1}{\left(1 + \frac{1}{c-1}\right)^c - 1} + 1\,. \tag{10.2}$$

It isn't hard to show that no better competitive ratio is achievable. While this no longer seems like simple financial advice to give to a first-time skier, the solution starts to make sense in a situation of hyperinflation in ski prices. As $c \to \infty$, so renting becomes infinitesimally cheap compared to buying, the competitive ratio tends to

$$\frac{e}{e-1} \approx 1.58\,.$$

This limit is best thought of in terms of *continuous ski rental.* You embark on a ski trip of unknown duration. You may rent skis, but rather than renting in discrete increments, you are charged for the exact

length of time you keep the skis. An any point, if you are still skiing you can choose to buy, incurring a large cost but ending the rental charges. For some applications of ski rental this continuous model is more suitable, for example in the laptop power-management application, where the power consumed by running the disk during one system clock tick is far less than by powering down and restarting the disk.

That completes the "blue square" model, and we are now ready for the "black diamond" model, namely *repeated ski rental.* Say you first learn to ski. Then you decide to try snowboarding. Then in the summer you take up waterskiing. Each time you have a new rental problem, but you have new information. For example, if you quickly tired of snowboarding, the same may be true for water skiing, so you would do better renting. The same issue arises in various applications. In laptop power-management, the system needs to decide over and over when to shut down the disk, and it can observe the user's disk-access behavior to achieve better performance. In a parallel program, a thread might encounter very similar behavior each time it reaches a particular lock, and that behavior can be used to make a more well-informed decision of how long to spin before blocking [45]. In particular, the thread's behavior might be modeled by a probability distribution for the time the thread must wait for the lock. If the thread chooses the amount of time it spins before blocking, the probability distribution can be used to derive the thread's expected cost (in wasted cycles). The best option for the thread is to choose the amount of time it spins so as to minimize this expected cost. Repeated ski rental has also been applied to IP-over-ATM networks [49]. In addition to having good theoretical properties, the algorithms in [45, 49] are shown empirically to have better performance than previous algorithms.

10.3 On-Line Adversaries and the Competitive Ratio

As suggested in the previous section, many algorithms for on-line problems can be studied by *competitive analysis.* Competitive analysis determines for an on-line algorithm the maximum over all possible inputs of the ratio of the cost incurred by the algorithm on that input to the minimum cost possible on that input. Sleator and Tarjan popularized the use of this performance measure in their seminal paper on sequential search and paging [66]. Their use of the performance ratio, though not new (having appeared in earlier bin-packing and scheduling work, for example [38]), triggered the proverbial avalanche, and competitive analysis has since been applied to an immense range of on-line problems.

In general, let F denote a family $\{I_1, I_2, \ldots\}$ of instances of an on-line problem. We require that an instance be finite, containing a bounded number of requests. Let $A(I)$ denote the cost of algorithm A on instance $I \in F$, and let $\mathrm{OPT}(I)$ denote the optimum cost to solve instance I. Then A is c_F*-competitive* if, for all $I \in F$,

$$A(I) \leq c_F \mathrm{OPT}(I) + b_F \ , \tag{10.3}$$

where c_F and b_F are constant with respect to I. They are in general functions of the family, F.

For example, in the ski rental problem, an instance is simply the number t, and the family of instances is the natural numbers. The competitive ratio of the green circle strategy is 2, because Eq. (10.3) holds with $c_F = 2$ and $b_F = 0$.

As an example of a more complicated situation, consider the *on-line Steiner tree problem,* defined as follows. An instance consists of a connected, weighted graph $G = (V, E)$ of n vertices and m edges, and a set $R \subseteq V$ of *required* vertices. A Steiner tree on R is a minimum-weight subtree of G that contains all the vertices in R. In the on-line problem, the set R is revealed on-line. Initially the subtree is empty. As each node is revealed, enough additional edges must be added to the tree so that the new node is connected to all previously revealed nodes. Once an edge has been added, it cannot be removed. The on-line Steiner tree problem arises in several settings, including facility location, planning of telecommunication networks, and as a subproblem of file allocation and load-balancing problems in networks.

Let $F(n, m)$ denote the family of instances of the on-line Steiner tree problem on n-node, m-edge weighted graphs, where the edge weights can be arbitrary. The "greedy algorithm," which always connects a new node to the previous tree in the cheapest possible way, is $O(\log n)$-competitive with $b_F = 0$ [41].

Furthermore, if b_F is required to be 0, then every on-line algorithm for the Steiner problem has a competitive ratio that is $\Omega(\log n)$ [41]. Here the competitive ratio is a function of the family of graphs, but independent of the weights. On the other hand, if we define $F(n, m, B)$ to be the family of n-node, m-edge graphs with all edge weights positive and total edge weight bounded by B, then the greedy algorithm is 0-competitive with $b_F = B$, since the weight of any solution is no more than the total edge weight.

The point of this discussion is that when we talk about the competitive ratio of a given algorithm, we will have Eq. (10.3) in mind, but the precise details of the model (the definition of the family) and the restrictions we place on the additive constant b_F may have a rather profound effect on the competitive ratio that we end up with for the algorithm.

Randomized Algorithms

A **randomized algorithm** is an algorithm that has access to an oracle that generates random bits when requested. It can use these random bits to "flip coins" and make different decisions based on the random bits. The competitive ratio is defined with respect to expected cost. Randomized algorithm A is c_F-*competitive* if, for all $I \in F$,

$$\mathbf{E}[A(I)] \leq c_F \mathrm{OPT}(I) + b_F \ . \tag{10.4}$$

The expectation is with respect to the sequence of random choices that A makes.

Adversaries

It is often useful to think of an on-line problem as a game between two players. One player, the **adversary,** generates requests on-line according to some specified mechanism and is charged a cost for its selections according to some specified rule. For example, it can be charged the optimum cost to satisfy the complete sequence of requests. The other player is the algorithm, which responds to requests by making a decision, and incurs some total cost over the course of the game. The adversary's goal is to maximize the ratio of the algorithm's cost to the adversary's cost, while the on-line algorithm's goal is to minimize it.

The basic definition of competitiveness for nonrandomized (deterministic) algorithms, given at the start of Section 10.3, corresponds to an adversary that has complete knowledge of the algorithm, that is allowed to generate any request sequence, and that is charged the minimum (off-line) cost to perform any request sequence it generates. Such an adversary can first decide how many requests it wants to generate, and then internally simulate the algorithm on each possible sequence of that length to find the sequence that maximizes the ratio between the on-line algorithm's cost and the adversary's cost. This kind of adversary is called a *deterministic off-line* adversary.

The definition of competitiveness for randomized algorithms given in "Randomized Algorithms" corresponds to an adversary that has complete knowledge of the algorithm but does not know what random bits the algorithm will have. It can generate any request sequence, and is charged the off-line cost. The adversary can internally simulate the algorithm with all possible strings of random bits to find a sequence that maximizes the expected ratio of costs. Such an adversary is called a *randomized oblivious* adversary (because it is oblivious to the actual random choice made by the on-line algorithm). Rather more subtle analyses of randomized algorithms are possible with when the adversary is allowed to know the random bits. We will return to this in a later section.

Extending the Notion of Competitiveness

Our plan is to measure the quality of an on-line algorithm by its competitive ratio. But is this always the best measure? For example, a person who was really considering learning to ski might not be convinced that the deterministic (green circle) and randomized (blue square) algorithms for ski rental are truly the best ones, despite our argument that they are optimal according to the measure of competitive ratio. Such

a person might complain that the competitive ratio is not always the right way to measure quality. For example, one might rather want to minimize the maximum cost incurred (in which case you should always buy right away). A number of other approaches to measuring the quality of an on-line strategy can be found in [53]. The competitive ratio is the only one used in computer science, however, and it seems to be flexible and useful, so we shall limit our attention to this measure.

Another possible complaint about competitive analysis is that the assumption of no knowledge about the future is overly pessimistic. For example, it may be unreasonable to assume that the fact that one has already made i ski trips gives no information about whether or not one will make the $(i+1)$st trip. In real life, people are not truly arbitrary. For some people, the more one has gone, the more likely one is to go again. Other people will try anything once but nothing twice, and so on. An on-line algorithm might be able to take advantage of these characteristics to reduce the overall cost. But the basic definition of competitive analysis does not take this into account. It presupposes a worst-case adversary, which is allowed to generate request in a completely capricious fashion. Such an analysis provides no way to differentiate a ski-rental strategy that tries to take advantage of observed characteristics of the request sequence, and one that always does the same thing (such as buying after y/x trips). In the worst case, both will have a competitive ratio no better than 2. In fact, an algorithm that does well for "reasonable" people might have a competitive ratio higher than 2.

We can account for this kind of situation within the basic framework of competitive analysis, however, by limiting the way that the adversary chooses request sequences. For example, in the "blue square" version of ski rental, the adversary might choose t according to some probability distribution D. Instance I occurs with probability p specified by D.

The definition of competitive ratio is extended to include probability distributions over the family of instances in the following way. An algorithm is c_F competitive on distribution D over a family of instances F if

$$\mathbf{E}[A(I) - c_F \cdot \mathrm{OPT}(I)] \leq b_F \,. \tag{10.5}$$

Here the expectation is over the distribution D on instances in F. By the linearity of expectations, this is equivalent to

$$\mathbf{E}[A(I)] \leq c_F \cdot \mathbf{E}[\mathrm{OPT}(I)] + b_F \,. \tag{10.6}$$

As an example, consider a version of ski rental in which the probability that t trips are taken is $e^{-t} \cdot \frac{e-1}{e}$ (the quantity $\frac{e-1}{e}$ is a normalizing constant). This models the case where one is likely to get bored with skiing (or become irreparably injured) as time goes on. The expected number of trips in this situation is $\frac{1}{e-1}$. One on-line strategy in this situation is to always rent, leading to an expected cost of $\frac{x}{e-1}$. This strategy turns out to be optimal, although we do not prove it here. The optimum off-line strategy for a given t is as in Section 10.2, and the exact closed-form expression for the expected off-line cost can be computed fairly easily. The adversary can maximize the ratio of the on-line to off-line cost by setting $y = x$. In this case the competitive ratio is $\frac{e}{e-1}$.

The reader may recall from Section 10.2 that the value $\frac{e}{e-1}$ is also the optimal competitive ratio that can be achieved by a randomized algorithm against a worst-case adversary. This is neither a coincidence nor a simple trick perpetrated by the authors of this article, but rather a deep principle of on-line analysis known as *Yao's minimax theorem* [73]. This theorem is actually an adaptation of the famous minimax theorem of game theory [71]. It states that the best ratio achievable by a deterministic algorithm against any distribution is exactly the same as the best ratio achievable by a randomized algorithm against a worst-case adversary.

More formally, for a given on-line problem let F_n denote the family of input instances of size at most n. Let $\mathcal{D}_n$ denote the set of all probability distributions over the instances in F_n. Let $\mathcal{A}_n$ denote the set of deterministic on-line algorithms for instances of size n. We think of a deterministic on-line algorithm as being a decision tree; a node at depth i in the tree represents a decision about how to respond to the ith request. Finally, let $\mathcal{R}_n$ denote the set of probability distributions on the algorithms in $\mathcal{A}_n$. A randomized

on-line algorithm is a simply a particular probability distribution $R \in \mathcal{R}_n$ on the deterministic algorithms in $\mathcal{A}$. Yao's minimax principle says

$$\lim_{n\to\infty} \left\{ \max_{D\in\mathcal{D}_n} \min_{A\in\mathcal{A}_n} \frac{\mathbf{E}_{I:D}[A(I)]}{\mathbf{E}_{I:D}[\mathrm{OPT}(I)]} \right\} = \lim_{n\to\infty} \left\{ \min_{R\in\mathcal{R}_n} \max_{I\in F_n} \frac{\mathbf{E}_{A:R}[A(I)]}{\mathbf{E}_{A:R}[\mathrm{OPT}(I)]} \right\} . \tag{10.7}$$

The notation $\mathbf{E}_{I:D}$ indicates that the expectation is over instances $I \in F_n$ according to the probability distribution D. Similarly, $\mathbf{E}_{A:R}$ indicates the expectation over algorithms $A \in \mathcal{A}_n$ according to the probability distribution R. Several variants of this minimax principle have been used in the analysis of on-line algorithms; see for example [20].

10.4 Paging: A Classic On-Line Problem

Paging is the archetypal problem in on-line algorithms, the problem that has been the testing ground for most new methods of analysis of on-line algorithms.

Consider a hierarchical storage system consisting of a small, fast memory, that can store k memory pages, and a large slow device with room for n pages, with $n \gg k$. The classic instantiation is a virtual memory system, with the fast memory constructed of RAM and the slow device being a hard disk. (There are other instantiations, see for example [49, 55].) When a program references a memory location in a page that isn't in memory, a *page fault* occurs, and the program is blocked and must wait while the page is loaded into memory. Some page must be evicted from memory to make room for the page being loaded, and a *paging algorithm* chooses which page to evict. The aim of the paging algorithm is to minimize the number of page faults caused by the sequence of memory references.

The first application of competitive analysis to paging occurred in the seminal CACM paper of Sleator and Tarjan [66]. They analyzed the performance of the least recently used (LRU) rule, an algorithm widely regarded as very effective in practice. They proved (generalizations of) the following two theorems, whose proofs we provide as examples of the genre.

THEOREM 10.1 *LRU has competitive ratio k.*

PROOF Partition the page reference sequence σ into $\sigma_0, \sigma_1, \ldots, \sigma_i$, such that LRU faults exactly k times in $\sigma_1, \ldots, \sigma_i$ and at most k times in σ_0. For $1 \le j \le i$, let p_j be the last page referenced in σ_{j-1}. During σ_j, LRU cannot fault twice on any page, and it cannot fault on p_j. Therefore σ_j contains references to k different pages, all different from p_j, so the optimal algorithm must fault at least once in σ_j. This suffices to prove the theorem.

THEOREM 10.2 *No deterministic on-line algorithm has a competitive ratio less than k.*

PROOF Let $\mathcal{A}$ be a deterministic algorithm, and consider the case $n = k + 1$. Let σ be the sequence constructed by always referencing the page that is *not* in $\mathcal{A}$'s fast memory. $\mathcal{A}$ faults on each reference, whereas the optimum algorithm, which always evicts the page whose next reference is furthest in the future [14], faults at most once every k references.

A better competitive ratio is achievable through the use of randomization. Fiat et al. [30] give a simple randomized algorithm that has competitive ratio $2\mathcal{H}_k$, where $\mathcal{H}_k = \sum_{1\le i\le k} 1/k$ is the k harmonic number, and is $\Theta(\log k)$. A more complex algorithm has competitive ratio $\mathcal{H}_k$ [58], matching a lower bound proved by Fiat et al.

These tight bounds for deterministic and randomized competitive ratios are pleasing, particularly in the case of an empirically good algorithm such as LRU. There are, however, some limitations with these results. First, other nonrandomized algorithms that behave worse than LRU in practice, such as first-in first-out (FIFO), also have a competitive ratio of k. Second, on traces taken from program executions, the performance ratio of LRU is much less than k, or even $\log k$, typically close to 2 [74].

Borodin et al. added some realism by modeling *locality of reference,* the property of page reference sequences that makes hierarchical storage useful at all. They modeled locality using an access graph, where the reference sequence is restricted to be a walk through the graph. In addition to analyzing LRU in this model, they considered how to use advance knowledge of the access graph, an approach continued by Irani et al. [43] and Fiat and Karlin [29]. Fiat and Mendel [31] show how to optimally use the access graph, even when it is not known in advance but must be learnt as the requests arrive. In addition, they need only store the most recently seen part of the access graph, rather than the entirety.

As discussed in "Extending the Notion of Competitiveness," an access graph is a way to limit the power of the adversary, restricting the possible reference sequences so that the results obtained have more bearing on paging in practice. A alternative approach was taken in the earliest theoretical treatments of paging in which reference sequences were assumed to be generated by various probability distributions. In one study, each page reference was chosen according to a fixed probability distribution over the set of pages [33], while others did probabilistic analyses of LRU in which page references were chosen according to LRU stack-depth (where the ith most recently accessed page has stack-depth i) [52, 64]. Results in these papers are specific to the particular probabilistic model of reference sequences, however, and we'd like more general-purpose results.

A step in this direction was the Markov paging model of Karlin et al. [47], in which the locality of reference inherent in a program is modeled by a Markov chain. Each node in the chain corresponds to a page and each transition into a state generates a reference to that page. Hence, a reference sequence is generated by the probabilistic transitions of the chain. Karlin et al. found an algorithm whose expected page fault rate on any Markov chain is within a constant factor of the best possible for that chain. This approach was extended by Lund et al. [55], who show that for *any* distribution over page reference sequences, there is a natural algorithm that gets within a constant factor of the best possible expected fault rate for the distribution, which needs only to know, for pages i and j in the fast memory, the probability that i will be referenced before j.

Koutsoupias and Papadimitriou [51] introduced two other alternative analysis techniques: the diffuse adversary model and comparative analysis. In the first technique, reference sequences are generated by some distribution $\mathcal{D}$ from a set Δ. The on-line algorithm can take advantage of knowing Δ, but doesn't know which $\mathcal{D}$ will be used, and its expected fault rate is compared to the expected optimal fault rate on sequences from $\mathcal{D}$. In comparative analysis, rather than comparing on-line algorithms to the optimal algorithm, we compare two arbitrary classes of algorithms. Koutsoupias and Papadimitriou use the diffuse adversary mode to exhibit a simple family Δ for which LRU is the best on-line algorithm, and comparative analysis to investigate the value of lookahead in paging, comparing algorithms with lookahead to those without.

Torng [70] includes the time to access fast memory, in addition to the time incurred during a page fault, in the cost of servicing a page reference sequence. This has a number of attractive consequences: lookahead provably helps, and on reference sequences exhibiting a natural notion of locality of reference, some algorithms (including LRU) achieve a *constant* competitive ratio. Another model for analyzing on-line algorithms, and algorithms for paging in particular, is Young's loose competitiveness [75].

10.5 General Models for On-line Problems

In this section we review some of the general models and tools that have been developed for on-line problems. If a given on-line problem can be phrased in terms of one of these models, standard algorithms

are available to solve the problem. In this section the adversaries are assumed to be worst-case.

Most of the models we discuss are defined over *metric spaces.* A metric space, M, is a pair (P, δ) where P is a set (finite or infinite) and δ is a distance function from $P \times P$ to $\mathcal{R}$. A typical example of a metric space is the plane with Euclidean distance. The function δ must satisfy several conditions:

1. $\delta(a, a) = 0 \; \forall \, a \in P$.
2. $\delta(a, b) \geq 0 \; \forall \, a, b \in P$.
3. $\delta(a, b) = \delta(b, a) \; \forall \, a, b \in P$.
4. $\delta(a, b) + \delta(b, c) \geq \delta(a, c) \; \forall \, a, b, c \in P$.

The k-Server Model

The k-server model, defined in [57], had its genesis in paging problems. Let M be a metric space and let S be a set of k *servers.* The servers are always located on k not necessarily distinct points $s_1, \ldots, s_k$ in M. An instance of the k-server problem consists of a metric space M, the initial positions of the servers, and a sequence σ of requests, which are revealed one-at-a-time. A request is simply a point r in M. The servers must satisfy the following condition.

> Let r be the most recently revealed request point. There must be at least one server, i, such that $s_i \equiv r$.

After request r is revealed, the set of servers are moved as necessary so that at least one server is located at the request point. Let s_i' denotes the position of server i after being moved. The cost of moving the servers is $\sum_i \delta(s_i, s_i')$. The total cost of servicing σ is the sum over requests of the movement costs.

Example: Paging. The paging problem is modeled as a k-server problem in the following way. There is one point in P for each unique page of memory. There is one server for each cache location. Define $\delta(a, b) = 1$ for any pair $a, b \in P$. If server i is located at point p, then page p is in cache location i. If no server is located at point p then the corresponding page is out of cache. Moving server i from a to b corresponds to ejecting page a and loading b into slot i.

When Manasse et al. [57] first proposed the k-server model, they conjectured that there exists a deterministic algorithm that has competitive ratio of k on any instance of the k-server problem (not just a paging instance). This became the rather famous "k-server conjecture." For deterministic algorithms, k is the best possible ratio. This follows from the lower bound on the competitive ratio of paging algorithms.

So far, the truth of the k-server conjecture has not been resolved, but a good candidate for a deterministic algorithm with ratio k is the *work-function algorithm,* developed independently by several researchers [21, 24, 57]. The work-function algorithm keeps track of the optimal off-line costs and tries to make moves that keep its cost close to the optimal cost. Given a sequence of m tasks, the optimal off-line cost can be computed with a simple dynamic programming algorithm. Let $g_i(X)$ denote the minimum cost to perform tasks 1 to i and end in configuration X, where $X = (s_1, \ldots, x_k)$. Then $g_0(X) = \delta(X_0, X)$, where X_0 is the start configuration, and

$$g_i(X) = \min_{X'} \left\{ g_{i-1}\left(X'\right) + \delta\left(X, X'\right) \right\} . \tag{10.8}$$

The optimal cost to process the sequence is $\min_X g_m(X)$.

The function $g_i(\cdot)$ is an example of a *work function.* A work function is a function mapping the configuration space to the positive reals, with the property that the value of the work function at point X lower-bounds the optimal cost to process requests 1 through i and end at position X.

Work Function Algorithm. Let r be the tth request, and let X_{t-1} be the configuration of the on-line servers when r arrives. Let X_t be the configuration X, $r \in X$, that minimizes $g_t(X) + \delta(X_{t-1}, X)$.

Koutsopias and Papadimtriou [50] showed that the work-function algorithm is $2k - 1$ competitive, but the proof is beyond the scope of this survey. Note that the work-function algorithm requires space $\Theta(n^k)$ space, which is quite prohibitive for large k.

Sometimes it is useful to compute the optimal off-line strategy. For example, it is useful in the design of a prefetching strategy for a data-independent computation flow, as might occur in a large matrix application such as Gaussian elimination or an eigenvalue calculation. Chrobak et al. [23] showed how to reduce finite instances of the off-line k-server problem to an instance of the network flow problem. If there are n points in the metric space, and there are m requests, the flow instance consists of a network of size $O(k(m + n))$. Using a standard network flow algorithm [68] a solution to the **off-line problem** can be found in time $O(k^2(m + n)^2 \log(k(m + n)))$.

Metrical Task Systems

Borodin et al. [21] proposed a more general model of on-line computation called *metrical task systems.* A metrical task system consists of a set of n states S and a distance function δ so that (S, δ) is a metric space. At any time, the system must be in exactly one of the n states. A sequence σ of *tasks* is revealed, one task at a time. Task t_i is an arbitrary cost vector, $(c_{i1}, \ldots c_{in})$, that maps states to costs. Value c_{ij} is a nonnegative real number. If the system is in state j at the time task i is processed, the cost of the task is then c_{ij}. The cost vectors can be entirely arbitrary and need have no relation to each other or the state transition function. If the function δ is not symmetric but still obeys the triangle inequality then (S, δ) is just called a *task system.*

The notion of *lookahead* arises in metrical task systems. Informally, an on-line problem has *lookahead* k if an on-line algorithm is allowed to know the next k requests when making decisions about how to change state. More formally, an on-line problem modeled as a metrical task system proceeds in a series of rounds. At the start of a round, the on-line algorithm is shown the next k tasks. The on-line algorithm can then change the state of the system at cost $\delta(s, s')$, where s and s' are the old and new states, respectively. Finally, the on-line algorithm "accepts" the next task t, at cost $c_{ts'}$. This ends the round. The value of k may be a function of the number of states n, but it cannot be a function of the number of tasks. An off-line algorithm, which can see the entire sequence of tasks, is said to have *unbounded* lookahead.

Example: sequential search. This problem models searching for an item in a linked list containing n items. After each search, the list can be rearranged to move more commonly accessed items forward. The subsection "Data Structure Problems" gives more details. The metrical task system (S, δ) has one state for each of the $n!$ permutations of the list. The cost of moving between two states is given by the number of inversions between the two lists: it is not hard to show that one list can be rearranged to any other with a number of exchanges equal to the number of inversions between source and target list. An access to element x is modeled by a task t such that c_{ts} is the position of item x in permutation s. This model has lookahead 0.

Example: paging. There are $\binom{m}{k}$ states, one for each possible subset of k pages in cache out of m total pages. The cost of moving between states is equal to the number of pages they differ in. An access to page x is modeled by a task t such that c_{ts} is 0 if s contains x in cache and $+\infty$ otherwise. The paging problem has lookahead 1. This reduction can be extended to model the k-server problem as a metrical task system by letting the cost to change state be the minimum distance servers must move to make the change.

It is evident that the task system model is very general and encompasses a large range of on-line problems. Borodin, Linial and Saks gave a general-purpose algorithm for the model.

The Borodin–Linial–Saks algorithm. This algorithm for task systems is a straightforward extension of the work function algorithm of "The k Server Model," although in fact the BLS algorithm

appeared first. As before, we keep track of the optimal off-line cost with a function $g_i(s)$, which denotes the minimum cost to perform tasks 1 to i and end in state s. This function can be computed as in the section on k-servers. Let r be the tth request, and let s_{t-1} be the current state when r arrives. Move to the state s_t that minimizes $g_t(s_t) + \delta(s_t, s_{t-1}) + c_{ts_t}$.

When δ is not symmetric, it is important that $\delta(s_t, s_{t-1})$ be used in the minimization, rather than $\delta(s_{t-1}, s_t)$. The formulation given above is different from that in [21] but the two are essentially equivalent.

Borodin, Linial, and Saks prove that this algorithm is $2n-1$-competitive on any metrical task system and $O(n^2)$-competitive on any task system that is not metrical. Furthermore, they show that no deterministic algorithm is better than $2n-1$-competitive. This result is rather depressing, since the number of states could be very large, as in the sequential search example above. There the number of states is $k!$, where k is length of the list. On the other hand, there are practical problems such as multiprocessor page migration (see "Data Management in Networks") in which the number of states is much smaller and plenty of processing time is available, and the BLS algorithm is certainly applicable in these situations.

Like the work function algorithm, the BLS algorithm requires lookahead 1. The paging example shows that there cannot be a general algorithm with lookahead 0 that has bounded competitiveness. Let $\delta(a, b) = 0$ for all states a, b. If the on-line algorithm is in state i, an adversary simply creates a task that has cost 1 in state i and cost 0 in all other states. The off-line cost is always 0 while the cost to the on-line algorithm can be made arbitrarily high by creating a sufficiently long sequence.

Recently Bartal et al. [11] found a randomized algorithm for metrical task systems that achieves a competitive ratio that is polylogarithmic in n. The algorithm is rather complicated, however, and lies beyond the scope of this section.

Request-Answer Games

The most general model of on-line problems that appears in the literature is *request-answer* games [15]. This model exploits the relationship between on-line algorithms and game theory hinted at in "Adversaries." An on-line problem can be regarded as a game between two players, the algorithm and the adversary. At each round of the game, the adversary makes a move, which is a request plus a list of legal responses (answers) and associated costs. The algorithm then selects one of the answers to respond with. Two types of adversaries have been considered: oblivious adversaries, which do not know what responses the algorithm gives and must decide on the next move in the dark, an adaptive adversaries, which are given complete information about the algorithm's responses. The distinction is only relevant when the algorithm is randomized. If the algorithm is deterministic, an adversary can simulate the algorithm and so know exactly what moves are made.

The adversary is charged a cost for the sequence of requests it generates. The cost can be assessed in two ways. The adversary may be charged the minimum cost to answer the requests using the given answers. This is the off-line adversary. Alternatively, the adversary may have an associated on-line algorithm that it must choose in advance. The cost incurred by this associated on-line algorithm is the cost charged to the adversary. This is the on-line adversary. Again, the distinction is moot if the algorithm is deterministic, because the adversary can simulate the algorithm, choose a sequence of requests, and then select an "on-line" algorithm that just happens to exactly minimize the cost on that particular sequence.

The algorithm wins against a particular adversary if the cost incurred by the algorithm is no more than a constant c times the cost incurred by the adversary. In this case we say the algorithm is c-competitive against the adversary. Because the request-answer model is so general, there cannot be an algorithm that is always competitive against all adversaries.

10.6 The Trail Map: A Selective Guide to On-line Problems

In recent years there has been a flurry of work on on-line problems. Competitive analysis has been applied to a mountain of old and new problems. This section contains a guide to some of the major areas that have received attention.

Data Structure Problems

Most dynamic data structures can be studied as on-line decision problems. So far, however, only two have received much attention.

In the *sequential search* or *list-update* problem, a dictionary of keys is stored in a linked-list data structure. The list may be searched for a particular key by scanning the list in order from front to back, stopping the scan as soon as the item is found. After each search, the list can be reordered by swapping adjacent pairs of elements. By moving items that are frequently searched for to the front of the list, the overall search time can be decreased, but the total cost of a request sequence is the sum of the search time for elements plus the cost of the swaps. Swaps that move the searched-for item forward are free. A popular optimization is the "move-to-front" rule: after a search succeeds, the found element is moved to the front of the list.

Early work on sequential search (see for example [63]) studied the performance of the move-to-front heuristic against an adversary that generates requests using a fixed probability distribution on the list elements. The adversary is charged the cost incurred by a static list that is optimally sorted for the given distribution (i.e., the item with highest probability of access is at the front, then the second highest probability, and so on). Chung et al. [25] showed that move-to-front never incurs a cost greater than $\pi/2$ times the cost incurred by the optimal static list, and Gonnet et al. [37] showed a distribution on which move-to-front did this badly.

A sea-change occurred in 1985, starting with Bentley and McGeoch [16] and followed shortly by Sleator and Tarjan's seminal paper on competitive analysis [66], which gave the first analysis of an on-line algorithm against the optimal off-line adversary. Sleator and Tarjan showed that move-to-front is 2-competitive. It is not hard to see that this is the best possible competitive ratio for a deterministic algorithm by considering a sequence in which the adversary always accesses the last element in the on-line algorithm's list. Since the early 1990s research has concentrated on randomized algorithms. The best known algorithm is 1.6-competitive against an oblivious adversary [3], and a lower bound of 1.5 is known for all randomized algorithms [69].

Another important data structure for storing and searching is the dynamic binary tree. An item is searched for using the standard binary tree search algorithm. After each search, the tree may be reorganized by doing a series of *rotations.* Let (u, v) be a tree edge such that v is the right child u. A rotation of edge (u, v) makes u the new left child of v, the old left child of v becomes the new right child of u, and v replaces u as the left or right child of the old parent of u. Rotations are a fundamental method of reorganizing binary trees, and more information can be found in a standard introductory algorithms textbook such as Cormen et al. [26]. The goal of the rotation is to reduce the search time, by moving frequently accessed items nearer to the root of the search tree.

Any standard balanced search tree such as AVL-trees or red/black trees (see [26]) ensures that the cost of an individual search is $O(\log n)$ and that $O(\log n)$ rotations are done per search. Since the optimal cost is at least 1 per search, balanced trees are trivially $O(\log n)$-competitive. Although there has been considerable work on the problem, no one has yet produced a reorganization algorithm that is provably better than $O(\log n)$-competitive. Conversely, no lower bound greater than 1 is known. It is known, however, that of all dynamic binary search tree algorithms that have so far been proposed, only one can be better than $O(\log n)$ competitive. That candidate is the splay tree data structure of Sleator and Tarjan [67]. Sleator and Tarjan showed that splay trees are $O(\log n)$-competitive, and conjectured that they are in fact $O(1)$-competitive. This conjecture has been shown true or nearly true in certain special cases but the competitiveness of splay trees remains open in the general case, and is one of the most interesting open problems in the theory of

data structures and on-line algorithms. See the survey on on-line data structure problems by Albers and Westbrook in [32] (also available at http://www.research.att.com/~jeffw/data-survey.ps)

Network Admission Control

The explosive growth of the Internet and World Wide Web, and the promise of revolutionary services combining voice, video, and data communications in asynchronous transfer mode (ATM) networks provide a fertile source of on-line problems. The main focus has been on admission control and routing problems in ATM networks, starting with Garay and Gopal [34] and Garay et al. [35], who studied preemptive call control on a single link or a line network. Awerbuch et al. [9] applied approximation techniques for multicommodity flow problems to give an elegant analysis of the following problem: calls are offered to a network on-line, each call having a bandwidth requirement, source and destination nodes and duration. The goal is to accept or reject calls, routing accepted calls subject to edge capacity constraints, so as to maximize the *throughput,* defined as the average over time of the aggregate bandwidth of accepted calls. Subject to a restriction that no call bandwidth is more than $1/\log \rho$ times the minimum edge capacity, they give an algorithm that is $\log \rho$-competitive, which is best possible, within constant factors, for deterministic algorithms. Here ρ is the product of the number of nodes in the network and the ratio of the maximum to minimum job duration. The algorithm essentially uses shortest-path routing with a load-dependent cost metric. More specifically, the cost of an edge is exponential in the load (fraction of capacity used), the cost of a path is the sum of its edge costs, and a call, if admitted, is routed on a least cost path. Load-dependent routing has been used in practice as an important component of established networks; Gawlick et al. [36] show that the exponential cost model gives good results in a practical setting.

A number of variants of competitive routing and admission control have been developed, for example where tight bounds are achieved for tree networks, arrays and hypercubes, where the objective is to minimize congestion rather than throughput. A good survey of these results is given by Plotkin [61].

Data Management in Networks

The problem of managing data in a network of computers can be abstracted as follows. A collection of n computers is connected by some kind of connection network. The network is modeled as an undirected graph whose nodes are computers and whose edges correspond to direct connects between computers. Each edge has a weight, which is a measure of the cost to send a message across the corresponding connection. (The cost may model delay, dollar cost per message, or any other actual cost measure.) The cost to send a message of size q bytes, $q \geq 1$, from machine i to machine j is modeled as $q \cdot \delta(i, j)$, where $\delta(i, j)$ is the total weight of the shortest path connecting i and j.

Each computer has some amount of local storage and supports a collection of local processes (users, or programs). A large collection of data, which we call the "file," is to be read or written concurrently by the processes. The file is partitioned into D records; a *request* at machine i is either a read or a write of an individual record. There may be one or more complete copies of the file stored in the network, each stored at a single machine. If machine i generates a read request, that request is satisfied at zero cost if the machine is storing a copy of the file. Otherwise, a message must be sent through the network to the nearest machine j storing a copy of the file. The requested record is then sent back, at cost $q\delta(i, j)$ for some constant q. For convenience, we assume $q = 1$. A write request is slightly more complicated. If there is a single copy in the network, then the write request is handled the same way as a read request. If there are multiple copies, all copies must be kept consistent. Hence, if a machine generates a write request, all machines containing a copy of the file must be notified. The message cost is lower-bounded by the weight of a minimum Steiner tree containing the requesting machine and all machines with a copy. This can be achieved (within constant factors) if the machines can store and forward messages intelligently (i.e., each machine transmits messages only to its adjacent neighbors in the Steiner tree. (Computing a Steiner tree is NP-hard, of course, but although we use a Steiner tree to model costs, in practice a Steiner tree can be

approximated within a factor of 2 by a minimum spanning tree, which is fast to compute.)

The overall goal is to minimize total message cost. As the request patterns change, it may be advantageous to make new copies of the data file. For example, a machine that is generating many reads can benefit from keeping a copy locally, although the addition of a new copy will increase the message cost of subsequent writes. Similarly, if there are many write requests compared to reads, it may be useful to decrease the number of copies. A copy of the file can be placed at machine i by transmitting the D records to i from the nearest machine j with a copy. The cost is $D \cdot \delta(i, j)$. A copy can be discarded from a machine at no cost. Of course, there must always be one copy of the file in the network.

Although this model is idealized it captures much of the main difficulties in handling files in networks. The static problem of assigning a fixed set of copies has been extensively studied. For the static problem a probabilistic distribution is typically assumed and then the expected cost per unit time is calculated for various cost models. Dowdy and Foster give a survey of these results [27].

The on-line problem has been studied in various settings. Two interesting restrictions are *replication,* in which there are only read requests, and *migration,* in which the file can be moved but there is never more than one copy. Migration and replication were the earliest of this class of problem to be studied in an on-line setting [18]. The general problem, which allows for reads, writes, and multiple copies, is called *file allocation.*

Consider replication in a network of two machines connected by an edge. Initially there is one copy, at machine a. Now suppose b begins to generate read requests. When should a copy be placed at b? A little reflection reveals that this is exactly the ski rental problem, with the rent cost 1 and the buy cost D. Hence, there is a two-competitive deterministic algorithm and an $e/(e-1)$-competitive randomized algorithm. For general networks, however, the problem becomes much harder. The off-line adversary, having decided what machines will need copies in the end, will achieve minimum cost by replicating all copies at the start along a minimum Steiner tree connecting the machines. The on-line algorithm must essentially generate on-line an approximation to this minimum Steiner tree. The competitive ratio for this problem is $\Theta(\log n)$ [41].

On-line migration has the advantage of avoiding the problem of maintaining consistency and is popular in designs of virtual shared memory. There are several deterministic and randomized algorithms, all with low constant competitive ratios, in the range 2–5. See Bartal et al. [12] for further details.

For file allocation, deterministic and randomized algorithms are known that are $O(\log n)$-competitive (the best possible up to constant factors). If the network is a bus or a tree, small constant competitive ratios can be achieved, in the range 2–3. In general networks, there is also a problem with simply keeping track of the nearest copy of the file. If the cost of this data tracking problem is included, the best-known competitive ratio increases to $O(\log^2 n)$. See Lund et al. [56] for more information.

There are various related problems, such as providing fault tolerance [72] or handling bounded storage at each machine [13]. A completely unexplored problem is when there is a cost per unit storage.

Robot Searching and Navigation

In this class of problems, an automaton is placed in an unknown environment, and asked to find its way to a certain location, or to find an object hidden in the environment. It discovers the nature of the environment only by exploring. In general, the goal is to limit the amount of time-consuming exploration required to reach the goal.

In the most abstract setting, the environment consists of a directed graph. The robot is started at some node in the graph, and asked to reach a particular other node. Each node can be uniquely identified by the robot, and when at a node, the robot can see and identify the set of incident edges. But it does not know where an edge leads until it has crossed the edge.

As an example, we can state the following problem, due in its original form to Chrobak and Larmore. A downhill skier, having inadvertently missed his turn and plunged off a cliff, finds himself in the afterlife, still wearing his skis. The afterlife consists of a completely flat, foggy, snowy 1-dimensional space, in which

the skier is standing at Cartesian coordinate zero. A supernatural being appears, states that a door into Heaven is located somewhere in the one-dimensional space, and then disappears again. What strategy should the skier adopt to find the doorway with the minimum amount of tedious cross-country poling? (Because of the fog, the skier has to actually ski right up to the doorway to find it.) The optimal strategy is to head right to the doorway, but the skier does not know in which of the two directions the doorway is located. But, if the skier adopts the strategy of moving in one direction for 1 time unit, then returning to the start, then going in the other direction for two time units, then returning, then going in the first direction for four time units, and so forth, the skier will eventually find the doorway having traveled no more than 9 times the optimal distance. Hence, this strategy is 9-competitive.

More geometrical papers concern a robot traveling in a two-dimensional plane that contains obstacles. An obstacle is a closed two-dimensional curve (i.e., a square or circle). The robot must find a path to the target (the location searched for) that avoids all the obstacles. Generally it is assumed that the robot knows its initial position and the location of the target, in Cartesian coordinates. An obstacle is discovered only when the robot runs into it.

In the model introduced by Blum et al. [19] it is assumed that the robot is told the dimensions of an obstacle once it encounters the obstacle, and that the robot has perfect position information. A navigation algorithm is called $c(n)$-competitive if the length of the path traveled by the robot is no more than $c(n)$ times the length of the shortest obstacle-avoiding path. In brief, it is known that if the obstacles are axis-parallel simple rectangles then there are $O(\sqrt{n})$-competitive navigation algorithms (and slightly better randomized algorithms [17]). If the obstacles can be concave, however, then even if the sides of obstacles are axis-parallel, no algorithm is better than $\Omega(n)$-competitive.

A somewhat different model and complexity measure has been adopted by the theoretical robotics community [54]. Here the obstacles can be bounded by arbitrary curves, and the shape of the obstacle must be discovered by traversing its boundary. The distance traveled by the robot is compared against the sum, P, of the perimeters of the obstacles. It is shown that in the worst-case any algorithm must travel distance at least P. How close the robot can come to that bound depends on how much knowledge the robot has about its position. With complete, exact knowledge of Cartesian coordinates there is an algorithm that never travels more than $2P$ [54]. With only partial position information (perhaps more interesting in practice) the robot multiplicative factor varies from 3 to $O(\log n / \log \log n)$. For references to more recent work see Angluin et al. [4].

Graph Theory

One can create an on-line version of almost any graph-theoretic problem by requiring that the graph be revealed in an on-line fashion. This can mean that the vertex set is known initially and edges of the graph are added one-at-a-time, or that nothing is known initially, and at each step a vertex plus all edges to already revealed nodes are given. In addition, one can allow node or vertex deletions in an on-line fashion. On-line algorithms for graph problems can often serve as simple approximation algorithms for problems that are hard to solve optimally, such as the Steiner tree problem in networks.

In the on-line matching problem, at each step a vertex is revealed, along with all its edges to previously revealed vertices. The new vertex can be matched to some previously revealed vertex, or it can be left unmatched in the expectation that some vertex to be revealed subsequently can be matched to the current vertex. Once the choice has been made, however, it cannot be unmade. Node and edges are never deleted. The matching constructed on-line is compared to the maximum matching in the full graph. No constant is allowed in the competitive ratio. One can also ask to construct a matching of maximum weight, when each edge has a weight. For unweighted matching [48], low constant competitive ratios are achievable. For example, the simple greedy strategy, which matches each vertex as it arrives, if possible, is 2-competitive. Weighted matching [44] is harder.

A generalized version of the on-line Steiner tree problem (see Section 10.3) in which nodes must be connected with multiple paths, is examined in [7]. The on-line traveling salesman problem is examined in [6].

Substantial research has been done on coloring a graph on-line (see for example [40]). The nodes and incident edges are revealed one at a time, and as each node is revealed it must be colored so that no adjacent vertex has the same color. The goal is to minimize the number of colors.

Scheduling and Load Balancing

Many scheduling and load-balancing problems are essentially on-line in nature. Scheduling problems arise, for example, in a computer operating system when users present tasks to be run on the system, and the tasks must be scheduled without knowledge of future task arrivals. An example of a load balancing problem is when a program running on a parallel machine spawns processes in an unpredictable way, and to optimize the program's performance, the processes must be partitioned on-line among the machine's processors.

On-line scheduling and load balancing is an active field of study. Representative topics are on-line scheduling of sequential jobs on parallel machines to minimize the makespan (maximum completion time) [65], scheduling parallel jobs on parallel machines [28], load balancing when each job can be handled only by a subset of the machines and we wish to minimize the maximum load [10] or the L_p norm of the machines [8], preemptively scheduling jobs in a single processor to minimize the flow time, or average waiting time of jobs [59], scheduling to minimize the average waiting time of precedence-constrained jobs [39]. Work in this area has had an impact on virtual-circuit routing (e.g., [5]) and algorithms for maximum-flow [60].

To give a sense of the typical approach to on-line scheduling and load balancing problems, we present a single problem in some detail. Consider a system consisting of m identical machines. A set of jobs is presented to the system, and a scheduler must assign each job to a machine, so as to minimize the maximum load. Here the load of a machine is the sum of the weights of jobs assigned to it. In perhaps the first on-line treatment of a scheduling problem, Graham observed that the *list processing* algorithm, which assigns each job in turn to the machine whose current load is smallest, has a competitive ratio of $2 - 1/m$.

The argument is follows. Suppose a collection of jobs has been assigned to the machines using Graham's algorithm. Let i be the most loaded machine, j be the job that was last assigned to machine i, let s be its size, and let w be the load on machine i just before job j was assigned. The algorithm's maximum load is $s + w$. Let Opt be the maximum load of the optimal algorithm. Clearly,

$$\text{Opt} \geq s \; . \tag{10.9}$$

In addition, the total weight of all jobs must be at least $mw + s$, because at the time job j was assigned to machine i, all other machines must have had load at least w.

Therefore

$$\text{Opt} \geq w + s/m \; . \tag{10.10}$$

Combining these inequalities gives

$$(2 - 1/m)\text{Opt} \geq w + s \; , \tag{10.11}$$

which proves that list processing is $2 - 1/m$ competitive. This bound is shown to be tight by the job sequence consisting of $m(m - 1)$ jobs of size 1 followed by a single job of size m.

Recently attention has been focused on the problem of determine the best-possible competitive ratio for this problem (see [2], and the references therein). Currently the best known bounds are due to Albers, who demonstrates an algorithm with competitive ratio at most 1.923, and proves that no algorithm can be better than 1.852-competitive.

Finance

A natural application area for on-line methods is finance. The worst-case nature of competitive analysis offers an appealing alternative to methods of mathematical finance which are based on detailed probabilistic models approximating the behavior of economic variables. On the other hand, the worst-case nature of competitive analysis is also problematic: if the markets are truly adversarial, the best investment strategy might be never to invest (and instead spend it all skiing). Therefore research in this area has focused on creating on-line models that, while remaining worst-case, allow some realistic constraints to be included. Raghavan's statistical adversary model [22, 62] allows an adversary to pick any input sequence (for example a sequence of daily share prices) as long as the sequence exhibits a particular statistical property (for example, a mean value within some range). Al-Binali [1] introduces a competitive risk-reward framework, in which an algorithm uses a particular forecast (for example, that interest rates will rise 1% within 6 months). This framework distinguishes between the restricted competitive ratio, which is the competitive ratio on inputs for which the forecast is correct, and the unrestricted competitive ratio. Let OC be the best possible unrestricted competitive ratio. The reward of the algorithm is the ratio of its restricted competitive ratio to OC, while the risk is the ratio of the algorithm's unrestricted competitive ratio to OC. The risk measures how badly the algorithm does when the forecast is wrong, the reward quantifies the benefit when the forecast is right.

10.7 Research Issues and Summary

As we have suggested in our Trail Map, on-line problems are ubiquitous in computer science. There is much research into on-line algorithms that is called by other names, and an important research goal is to further integrate competitive analysis into these areas.

The foundational questions in the study of on-line algorithms concern the right model and right measure of goodness. Designing algorithms to have good competitive ratios is a useful exercise that brings additional insight into the problem at hand. Since the standard competitive ratio is a worst-case measure, however, it may sometimes be too pessimistic, and variants of the standard competitive analysis have been proposed to address this issue. An important avenue of research is to provide a unifying framework for these variants, and to better understand when each variant it is likely to be a good predictor of actual performance.

Such research is both empirical and theoretical. As described in Section 10.4, the paging problem has become a proving ground both for empirical studies and for attempts to refine competitive analysis to provide a measure that better distinguishes between algorithms that have substantially different behavior in practice, and to capture situations in which worst-case adversaries are very unlikely. A skier doesn't have to use that same pair of skis on all terrain; she can buy different equipment for powder, moguls, ice, or slush.

The classic open problem in competitive analysis is resolving the "k-server conjecture" (see Section 10.5). Although substantial progress has been made on this problem, a tantalizing gap still remains.

10.8 Defining Terms

Adversary: The input sequence can be thought of as being generated by an adversary that uses information about the past moves of the on-line algorithm to choose inputs that maximize the ratio between the cost to the algorithm and the optimal cost.

Competitive analysis: A performance analysis in which an on-line algorithm is evaluated by comparing its performance to the best that could have been achieved if all the inputs had been known in advance.

Competitive ratio: The worst-case ratio between the cost incurred by an on-line algorithm and the optimal cost.

Off-line problem: A decision problem in which an algorithm is given the entire sequence of inputs in advance.

On-line algorithm: An algorithm that solves an on-line problem.

On-line problem: A problem in which an algorithm receives a sequence of inputs, and must process each input in turn, without detailed knowledge of future inputs.

Optimal cost: The minimum cost to process an input sequence.

Randomized algorithm: An algorithm that uses random bits to make decisions.

References

[1] al Binali, S., The competitive analysis of risk taking with application to online trading. In *36th Annual Symposium on Foundations of Computer Science,* 1997.

[2] Albers, S., Better bounds for online scheduling. In *Proceedings of the 28th Annual ACM Symposium on Theory of Computing,* 130–139, El Paso, TX, 1997.

[3] Albers, S. and Mitzenmacher, M., Average case analyses of list update algorithms, with applications to data compression. In *Proc. of the 23rd International Colloquium on Automata, Languages and Programming, Springer Lecture Notes in Computer Science, Volume 1099,* 514–525, 1996.

[4] Angluin, D., Westbrook, J., and Zhu, W., Robot navigation with range queries. In *Proc. 28th ACM Symposium on the Theory of Computing,* 469–478, 1996.

[5] Aspnes, J., Azar, Y., Fiat, A., Plotkin, S., and Waarts, O., On-line load balancing with applications to machine scheduling and virtual circuit routing. In *Proc. 25th ACM Symposium on the Theory of Computing,* 623–631, 1993.

[6] Ausiello, G., Feuerstein, E., Leonardi, S., Stougie, L., and Talamo, M., Competitive algorithms for the on-line traveling salesman problem. In *Proceedings of International Workshop on Algorithms and Data Structures,* volume 955 of *Lecture Notes in Computer Science,* 1995.

[7] Awerbuch, B., Azar, Y., and Bartal, Y., On-line generalized steiner problem. In *Proc. of 7th ACM-SIAM Symposium on Discrete Algorithms,* 68–74, 1996.

[8] Awerbuch, B., Azar, Y., Grove, E.F., Kao, M.-Y., Krishnan, P. and Vitter, J.S., Load balancing in the L_p norm. In *36th Annual Symposium on Foundations of Computer Science,* 383–391, Milwaukee, WI, IEEE, 1995.

[9] Awerbuch, B., Azar, Y., and Plotkin, S., Throughput-competitive online routing. In *34th IEEE Symposium on Foundations of Computer Science.* 32–40, 1993.

[10] Azar, Y., Broder, A., and Karlin, A., On-line load balancing. In *Proc. 33rd IEEE Symposium on Foundations of Computer Science,* 218–225. To appear in *Theoretical Computer Science,* 1992.

[11] Bartal, Y., Blum, A., Burch, C., and Tomkins, A., A polylog(n)-competitive algorithm for metrical task systems. In *Proc. 29th ACM Symposium on Theory of Computing,* 711–719, 1997a.

[12] Bartal, Y., Charikar, M., and Indyk, P., On page migration and other related task systems. In *Proceedings of the 8th Annual ACM-SIAM Symposium on Discrete Algorithms,* 43–52, New Orleans, LA, 1997b.

[13] Bartal, Y., Fiat, A., and Rabani, Y., Competitive algorithms for distributed data management. In *Proc. of the 24th Symposium on Theory of Computation,* 39–48, 1992.

[14] Belady, L., A study of replacement algorithms for virtual storage computers. *IBM Systems Journal,* 5, 78–101, 1966.

[15] Ben-David, S., Borodin, A., Karp, R., Tardos, G., and Widgerson, A., On the power of randomization in on-line algorithms. In *Proc. 22nd Symposium on Theory of Algorithms,* 379–386, 1990.

[16] Bentley, J.L. and McGeoch, C.C., Amortized analyses of self-organizing sequential search heuristics. *Communications of ACM,* 28(4), 404–411, 1985.
[17] Berman, P., Blum, A., Fiat, A., Karloff, H., Rosén, A. and Saks, M., Randomized robot navigation algorithms. In *Proc. 7th ACM-SIAM Symp. on Discrete Algorithms,* 75–84, 1996.
[18] Black, D.L. and Sleator, D.D., Competitive algorithms for replication and migration problems. Technical Report CMU-CS-89-201, Department of Computer Science, Carnegie-Mellon University, 1989.
[19] Blum, A., Raghavan, P., and Schieber, B., Navigating in unfamiliar geometric terrain. In *Proc. 23rd STOC,* 494–504, 1991.
[20] Borodin, A. and El-Yaniv, R., *Online Algorithms and Competitive Analysis.* Cambridge University Press, 1998.
[21] Borodin, A., Linial, N., and Saks, M., An optimal online algorithm for metrical task systems. In *Proc. 19th Annual ACM Symposium on Theory of Computing,* 373–382, 1987.
[22] Chou, A., Cooperstock, J., El-Yaniv, R., Klugerman, M. and Leighton, T., The statistical adversary allows optimal money-making trading strategies. In *Proceedings of the 6th Annual ACM-SIAM Symposium on Discrete Algorithms,* 1995.
[23] Chrobak, M., Karloff, H., Payne, T.H., and Vishwanathan, S., New results on server problems. *SIAM Journal on Discrete Mathematics,* 4, 172–181, 1991. Also in Proceedings of the 1st Annual ACM-SIAM Symposium on Discrete Algorithms, San Francisco, 1990, 291-300.
[24] Chrobak, M. and Larmore, L.L., The server problem and on-line games. In *DIMACS Series in Discrete Mathematics and Theoretical Computer Science,* volume 7, 11–64, 1992.
[25] Chung, F.R.K., Hajela, D.J., and Seymour, P.D., Self-organizing sequential search and Hilbert's inequality. In *Proc. 17th Annual Symposium on the Theory of Computing,* 217–223, 1985.
[26] Cormen, T., Leiserson, C., and Rivest, R., *Introduction to Algorithms.* McGraw-Hill, New York, 1990.
[27] Dowdy, L.W. and Foster, D.V., Comparative models of the file assignment problem. *ACM Computing Surveys,* 14(2), 287–313, 1982.
[28] Feldmann, A., Kao, M.Y., Sgall, J., and Teng, S.H., Optimal online scheduling of parallel jobs with dependencies. In *Proc. 25th ACM Symposium on Theory of Computing,* 642–651. ACM, 1993.
[29] Fiat, A. and Karlin, A., Randomized and multipointer paging with locality of reference. In *Proc. 27th ACM Symposium on Theory of Computing,* 626–634, 1995.
[30] Fiat, A., Karp, R., Luby, M., McGeoch, L.A., Sleator, D. and Young, N., Competitive paging algorithms. *Journal of Algorithms,* 12, 685–699, 1991.
[31] Fiat, A. and Mendel, M., Truly online paging with locality of reference. In *Proc. 38th Symposium on Foundations of Computer Science (FOCS),* 1997.
[32] Fiat, A. and Woeginger, G., *Survey Papers on Online Algorithms and Competitive Analysis.* To appear, 1998.
[33] Franaszek, P. and Wagner, T.J., Some distribution-free aspects of paging performance. *Journal of the ACM,* 21, 31–39, 1974.
[34] Garay, J. and Gopal, I., Call preemption in communication networks. In *Proc. Infocom,* 1992.
[35] Garay, J., Gopal, I., Kutten, S., Mansour, Y., and Yung, M., Efficient online call control algorithms. In *Proc. 2nd Israel Symposium on Theory of Computing and Systems,* 285–293, 1993.
[36] Gawlick, R., Kamath, A., Plotkin, S., and Ramakrishnan, K., Routing and admission control of virtual circuits in general topology networks. Technical Report BL011212-940819-19TM, AT&T Bell Laboratories, 1994.
[37] Gonnet, G.H., Munro, J.I., and Suwanda, H., Towards self-organizing linear search. In *Proc. 19th Annual IEEE Symposium on Foundations of Computer Science,* 169–174, 1979.
[38] Graham, R.L., Bounds for certain multiprocessing anomalies. *Bell System Technical Journal,* 45, 1563–1581, 1966.

[39] Hall, L., Shmoys, D., and Wein, J., Scheduling to minimize average completion time: Off-line and on-line algorithms. In *Proc. of 7th ACM-SIAM Symposium on Discrete Algorithms,* 142–151, 1996.

[40] Halldórsson, M.M., Parallel and on-line graph coloring algorithms. In *Proc. 3rd Int. Symp. on Algorithms and Computation,* 61–70. Lecture Notes in Computer Science, Springer-Verlag, 1992.

[41] Imase, M. and Waxman, B.M., Dynamic Steiner tree problem. *SIAM J. Discrete Math.,* 4, 369–384, 1991.

[42] Irani, S. and Karlin, A., Online computation. In Hochbaum, D., Ed., *Approximation Algorithms,* 521–564. PWS, New York, 1996.

[43] Irani, S., Karlin, A., and Phillips, S., Strongly competitive algorithms for paging with locality of reference. In *3rd Annual ACM-SIAM Symposium on Discrete Algorithms,* 228–236, 1992.

[44] Kalyanasundaram, B. and Pruhs, K., Online weighted matching. *Journal of Algorithms,* 14, 478–488, 1993. Preliminary version appeared in SODA '91.

[45] Karlin, A., Li, K., Manasse, M., and Owicki, S., Empirical studies of competitive spinning for shared memory multiprocessors. In *Proc. 13th ACM Symposium on Operating Systems Principles,* 1991.

[46] Karlin, A., Manasse, M., Rudolph, L., and Sleator, D., Competitive snoopy caching. *Algorithmica,* 3(1), 79–119, 1988.

[47] Karlin, A., Phillips, S., and Raghavan, P., Markov paging. In *Proc. 33rd IEEE Symposium on Foundations of Computer Science,* 208–217, 1992.

[48] Karp, R.M., Vazirani, U.V., and Vazirani, V.V., An optimal algorithm for on-line bipartite matching. In *Proceedings of the 22nd Annual ACM Symposium on Theory of Computing,* 352–358, Baltimore, MD, 1990.

[49] Keshav, S., Lund, C., Phillips, S., Reingold, N., and Saran, H., An empirical evaluation of virtual circuit holding time policies in IP-over-ATM networks. *IEEE Journal on Selected Areas in Communications,* 13(8), 1371–1382, 1995.

[50] Koutsoupias, E. and Papadimitriou, C., On the k-server conjecture. In *Proc. 25th Symposium on Theory of Computing,* 507–511, 1994a.

[51] Koutsoupias, E. and Papadimitriou, C.H., Beyond competitive analysis. In *35th Annual Symposium on Foundations of Computer Science,* 394–400, Santa Fe, New Mexico. IEEE, 1994b.

[52] Lewis, P. and Shedler, G., Empirically derived models for sequences of page exceptions. *IBM J. Res. and Develop.,* 17, 86–100, 1973.

[53] Luce, R.D. and Raiffa, H., *Games and Decisions.* John Wiley & Sons, 1957.

[54] Lumelsky, V.J. and Stepanov, A.A., Path-planning strategies for a point mobile automaton moving amidst unknown obstacles of arbitrary shape. *Algorithmica,* 2, 403–430, 1987.

[55] Lund, C., Phillips, S., and Reingold, N., IP over connection-oriented networks and distributional paging. In *35th IEEE Symposium on Foundations of Computer Science,* 424–435, 1994a.

[56] Lund, C., Reingold, N., Westbrook, J., and Yan, D., On-line distributed data management. In *Proceedings of the 2nd Annual European Symposium on Algorithms, ESA '94,* volume 855 of *Lecture Notes in Computer Science,* 202–214, Utrecht, The Netherlands. Springer-Verlag, 1994b.

[57] Manasse, M., McGeoch, L.A., and Sleator, D., Competitive algorithms for online problems. In *Proc. 20th Annual ACM Symposium on Theory of Computing,* 322–333, 1988.

[58] McGeoch, L. and Sleator, D., A strongly competitive randomized paging algorithm. *J. Algorithms,* 6, 816–825, 1991.

[59] Motwani, R., Phillips, S., and Torng, E., Non-clairvoyant scheduling. In *Proc. 4th Annual ACM-SIAM Symposium on Discrete Algorithms,* 422–431, 1993. Also to appear in *Theoretical Computer Science,* Special Issue on Dynamic and On-Line Algorithms.

[60] Phillips, S. and Westbrook, J., Online load balancing and network flow. In *Proceedings of the Twenty-Fifth Annual ACM Symposium on Theory of Computing,* 402–411, San Diego, CA, 1993.

[61] Plotkin, S., Competitive routing of virtual circuits in atm networks. *IEEE J. Selected Areas in Comm.,* 1128–1136. Special issue on Advances in the Fundamentals of Networking, 1995.
[62] Raghavan, P., A statistical adversary for on-line algorithms. DIMACS *Series in Discrete Mathematics and Theoretical Computer Science,* 7, 79–83, 1992.
[63] Rivest, R., On self-organizing sequential search heuristics. *Communication of the ACM,* 19, 63–67, 1976.
[64] Shedler, G. and Tung, C., Locality in page reference strings. *Sicomp,* 1, 218–241, 1972.
[65] Shmoys, D.B., Wein, J., and Williamson, D.P., Scheduling parallel machines on-line. In McGeoch, L.A. and Sleator, D.D., Eds., *On-Line Algorithms,* volume 7 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science,* 163–166. AMS/ACM, 1991.
[66] Sleator, D. and Tarjan, R.E., Amortized efficiency of list update and paging rules. *Communications of the ACM,* 28, 202–208, 1985a.
[67] Sleator, D.D. and Tarjan, R.E., Self-adjusting binary search trees. *J. Assoc. Comput. Mach.,* 32, 652–686, 1985b.
[68] Tarjan, R.E., *Data Structures and Network Algorithms.* SIAM, Philadelphia, 1983.
[69] Teia, B., A lower bound for randomized list update algorithms. *Information Processing Letters,* 47, 5–9, 1993.
[70] Torng, E., A unified analysis of paging and caching. In *36th IEEE Symposium on Foundations of Computer Science,* 194–203, 1995.
[71] von Neumann, J. and Morgenstern, O., *Theory of Games and Economic Behavior.* Princeton University Press, 1947.
[72] Westbrook, J. and Zuck, L., Adaptive algorithms for paso systems. In *Proc. ACM Symp. on Principles of Distributed Computing (PODC 94),* 264–273, 1994.
[73] Yao, A.C., Probabilistic computations: Towards a unified measure of complexity. In *Proc. 12th ACM Symposium on Theory of Computing,* 1980.
[74] Young, N., The k-server dual and loose competitiveness for paging. *Algorithmica.* To appear. Rewritten version of "On-line caching as cache size varies," in *The 2nd Annual ACM-SIAM Symposium on Discrete Algorithms,* 241–250, 1991.
[75] Young, N.E., The k-server dual and loose competitiveness for paging. *Algorithmica,* 11(6), 525–541, 1994.

Further Information

Irani and Karlin [42] provide a short survey of the field from a more theoretical viewpoint, while the book of Borodin and El-Yaniv [20] is an extensive treatment of on-line algorithms. A collection of surveys of various subfields has been edited by Fiat and Woeginger [32]. Marek Chrobak and John Noga have collected a bibliography of papers on on-line problems, available at

http://www.cs.ucr.edu/~marek/pubs/online.bi

We have extended that bibliography with recent papers and our revised version is available at

http://www.research.att.com/~jeffw/online.bib

Current research into on-line problems and algorithms is published in a number of conferences and journals. Currently there is a lag time of several years between journal submission and final publication, so conferences provide the forum for fast dissemination of cutting-edge research. A partial list of larger conferences includes the ACM Symposium on Theory of Computing (STOC), the IEEE Conference on Foundations of Computer Science (FOCS), the ACM-SIAM Symposium on Discrete Algorithms (SODA), the International Colloquium on Automata, Languages, and Programming (ICALP), and the European

Symposium on Algorithms (ESA). There are also numerous regional conferences. Announcements about conferences can generally be found in the *Communications of the ACM* or *IEEE Spectrum.*

A partial list of journals includes *Journal of the ACM, SIAM Journal on Computing, Journal of Algorithms, Algorithmica, Information Processing Letters,* and *Theoretical Computer Science.*

11

Pattern Matching in Strings

Maxime Crochemore
Université de Marne-la-Vallée

Christophe Hancart
Université de Rouen

11.1 Introduction

The present chapter describes a few standard algorithms used for processing **texts.** They apply, for example, to the manipulation of texts (text editors), to the storage of textual data (text compression), and to data retrieval systems. The algorithms of the chapter are interesting in different respects. First, they are basic components used in the implementations of practical software. Second, they introduce programming methods that serve as paradigms in other fields of computer science (system or software design). Third, they play an important role in theoretical computer science by providing challenging problems.

Although data are stored variously, text remains the main form of exchanging information. This is particularly evident in literature or linguistics where data are composed of huge corpora and dictionaries. This applies as well to computer science where a large amount of data are stored in linear files. And this is also the case in molecular biology where biological molecules can often be approximated as sequences of nucleotides or aminoacids. Moreover, the quantity of available data in these fields tends to double every 18 months. This is the reason why algorithms should be efficient even if the speed of computers increases regularly.

The manipulation of texts involves several problems among which are **pattern** matching, approximate pattern matching, comparing strings, and text compression. The first problem is partially treated in the present chapter, in that we consider only one-dimensional objects. Extensions of the methods to higher dimensional objects and solutions to the second problem appear in Chapter 13. The third problem includes the comparison of molecular sequences, and is developed in the corresponding chapter. Finally, Chapter 12 is devoted to text compression.

Pattern matching is the problem of locating a collection of objects (the pattern) inside raw text. This is the opposite of the data base approach in which texts are structured in fields which themselves are searched by keywords. In this chapter, texts and elements of patterns are strings, which are finite sequences

of symbols over a finite alphabet. Methods for searching patterns described by general regular expressions derive from standard parsing techniques (see the chapter on formal grammars and languages). We focus our attention to the case where the pattern represents a finite set of strings. Although the latter case is a specialization of the former case, it can be solved with more efficient algorithms.

Solutions to pattern matching in strings divide in two families. In the first one, the pattern is fixed. This situation occurs for example in text editors for the "search" and "substitute" commands, and in telecommunications for checking tokens. In the second family of solutions, the text is considered as fixed while the pattern is variable. This applies to dictionaries and to data bases of molecular sequences, for example, and to full-text data bases in general.

The efficiency of algorithms is evaluated by their worst-case running times and the amount of memory space they require. In almost all cases, these are the most objective and consistent criteria to appreciate the efficiency of algorithms. A more realistic measure is to consider the expected running time of programs. But such a computation is machine dependent and, moreover, it is based on an average analysis that is often unreachable. This is partly due to the fact that texts are hard to modelize in a probabilistic framework and that computations are impracticable in pertinent models. So, most average-case analyses are on random texts.

The alphabet, the finite set of symbols, is denoted by Σ, and the whole set of strings over Σ by Σ^*. The length of a string u is denoted by $|u|$; it is the length of the underlying finite sequence of symbols. The concatenation of two strings u and v is denoted by uv. A string v is said to be a **factor** (also called segment, substring, etc.) of a string u if u can be written in the form $u'vu''$ where $u', u'' \in \Sigma^*$; if $i = |u'|$ and $j = |u'v| - 1$, we say that the factor v starts at position i and ends at position j in u; the factor v is also denoted by $u[i\,..\,j]$. The symbol at position i in a string u, that is the $i+1$-th symbol of u, is denoted by $u[i]$; we consider implicitly that $u = u[0\,..\,|u| - 1]$.

11.2 Matching Fixed Patterns

We consider in this section the two cases where the pattern represents a fixed string or a fixed dictionary (a finite set of strings). Algorithms search for and locate all the **occurrences** of the pattern in any text.

In the string-matching problem, the first case, it is convenient to consider that the text is examined through a **window.** The window delimits a factor of the text and has usually the length of the pattern. It slides along the text from left to right. During the search, it is periodically shifted according to rules that are specific to each algorithm. When the window is at a certain position on the text, the algorithm checks whether the pattern occurs there or not by comparing some symbols in the window with the corresponding aligned symbols of the pattern; if there is a whole match, the position is reported. During this scan operation, the algorithm acquires from the text information that is often used to determine the length of the next shift of the window. Some part of the gathered information can also be memorized in order to save time during the next scan operation.

In the dictionary-matching problem, the second case, methods are based on the use of automata, or related data structures.

The Brute Force Algorithm

The simplest implementation of the sliding window mechanism is the brute force algorithm. The strategy consists here in uniformly sliding the window one position to the right after each scan operation. As far as scans are correctly implemented, this obviously leads to a correct algorithm.

We give below the pseudocode of the corresponding procedure. The inputs are a nonempty string x, its length m (thus $m \geq 1$), a string y, and its length n. The variable p in the procedure corresponds to the current left position of the window on the text. It is understood that the string-to-string comparison in line 2 has to be processed symbol per symbol according to a given order.

```
BRUTE-FORCE-MATCHER(x, m, y, n)
1  for p from 0 up to n − m
2      loop if y[p .. p + m − 1] = x
3                then report p
```

The time complexity of the brute force algorithm is $O(m \times n)$ in the worst case (for instance when $a^{m-1}b$ is searched in a^n for any two symbol $a, b \in \Sigma$ satisfying $a \neq b$ if we assume that the rightmost symbol in the window is compared last). But its behavior is linear in n when searching in random texts.

The Karp–Rabin Algorithm

Hashing provides a simple method for avoiding a quadratic number of symbol comparisons in most practical situations. Instead of checking at each position p of the window on the text whether the pattern occurs here or not, it seems to be more efficient to check only if the factor of the text delimited by the window, namely $y[p \,..\, p + m - 1]$, "looks like" x. In order to check the resemblance between the two strings, a hash function is used. But, to be helpful for the string-matching problem, the hash function should be highly discriminating for strings. According to the running times of the algorithms, the function should also have the following properties:

- To be efficiently computable;
- To provide an easy computation of the value associated with the next factor from the value associated with the current factor.

The last point is met when symbols of alphabet Σ are assimilated with integers and when the hash function, say h, is defined for each string $u \in \Sigma^*$ by

$$h(u) = \left(\sum_{i=0}^{|u|-1} u[i] \times d^{|u|-1-i} \right) \bmod q \ ,$$

where q and d are two constants. Then, for each string $v \in \Sigma^*$, for each symbols $a', a'' \in \Sigma$, $h(va'')$ is computed from $h(a'v)$ by the formula

$$h\left(va''\right) = \left(\left(h\left(a'v\right) - a' \times d^{|v|} \right) \times d + a'' \right) \bmod q \ .$$

During the search for pattern x, it is enough to compare the value $h(x)$ with the hash value associated with each factor of length m of text y. If the two values are equal, that is, in case of collision, it is still necessary to check whether the factor is equal to x or not by symbol comparisons.

The underlying string-matching algorithm, which is denoted as the Karp–Rabin algorithm, is implemented below as the procedure KARP-RABIN-MATCHER. In the procedure, the values $d^{m-1} \bmod q$, $h(x)$, and $h(y[0 \,..\, m-2])$ are first precomputed, and stored respectively in the variables r, s, and t (lines 1–7). The value of t is then recomputed at each step of the search phase (lines 8–12). It is assumed that the value of symbols ranges from 0 to $c - 1$; the quantity $(c - 1) \times q$ is added in line 8 to provide correct computations on positive integers.

KARP-RABIN-MATCHER(x, m, y, n)
1 $r \leftarrow 1$
2 $s \leftarrow x[0] \bmod q$
3 $t \leftarrow 0$
4 **for** i from 1 up to $m-1$
5 **loop** $r \leftarrow (r \times d) \bmod q$
6 $s \leftarrow (s \times d + x[i]) \bmod q$
7 $t \leftarrow (t \times d + y[i-1]) \bmod q$
8 **for** p from 0 up to $n-m$
9 **loop** $t \leftarrow (t \times d + y[p+m-1]) \bmod q$
10 **if** $t = s$ and $y[p\,..\,p+m-1] = x$
11 **then** report p
12 $t \leftarrow ((c-1) \times q + t - y[p] \times r) \bmod q$

Convenient values for d are powers of 2; in this case, all the products by d can be computed as shifts on integers. The value of q is generally a large prime (such that the quantities $(q-1) \times d + c - 1$ and $c \times q - 1$ do not cause overflows), but it can also be the value of the implicit modulus supported by integer operations. An illustration of the behavior of the algorithm is given in Fig. 11.1.

p	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
$y[p]$	n	o	␣	d	e	f	e	n	s	e	␣	f	o	r	␣	s	e	n	s	e
$h(y[p\,..\,p+4])$	8	8	6	28	9	18	28	26	22	12	17	24	16	0	1	9	—	—	—	—

FIGURE 11.1 An illustration of the behavior of the Karp–Rabin algorithm when searching for the pattern $x =$ `sense` in the text $y =$ `no␣defense␣for␣sense`. Here, symbols are assimilated with their ASCII codes (hence $c = 256$), and the values of q and d are set, respectively, to 31 and 2. This is valid for example when the maximal integer is $2^{16} - 1$. The value of $h(x)$ is $(115 \times 16 + 101 \times 8 + 110 \times 4 + 115 \times 2 + 101) \bmod 31 = 9$. Since only $h(y[4\,..\,8])$ and $h(y[15\,..\,19])$ among the defined values of $h(y[p\,..\,p+4])$ are equal to $h(x)$, two string-to-string comparisons against x are performed.

The worst case complexity of the above string-matching algorithm is quadratic, as it is for the brute force algorithm, but its expected running time is $O(m+n)$ if parameters q and d are adequate.

The Knuth–Morris–Pratt Algorithm

This section presents the first discovered linear-time string-matching algorithm. Its design follows a tight analysis of a version of the brute force algorithm in which the string-to-string comparison proceeds from left to right. The brute force algorithm wastes the information gathered during the scan of the text. On the contrary, the Knuth–Morris–Pratt algorithm stores the information with two purposes. First, it is used to improve the length of shifts. Second, there is no backward scan of the text.

Consider a given position p of the window on the text. Assume that a mismatch occurs between symbols $y[p+i]$ and $x[i]$ for some i, $0 \le i < m$ (an illustration is given in Fig. 11.2). Thus, we have $y[p\,..\,p+i-1] = x[0\,..\,i-1]$ and $y[p+i] \neq x[i]$. With regard to the information given by $x[0\,..\,i-1]$, interesting shifts are necessarily connected with the **borders** of $x[0\,..\,i-1]$. (A border of a string u is a factor of u that is both a **prefix** and a **suffix** of u). Among the borders of $x[0\,..\,i-1]$, the longest proper border followed by a symbol different from $x[i]$ is the best possible candidate, subject to the existence of such of a border. (A factor v of a string u is said to be a **proper** factor of u if u and v are not identical, that

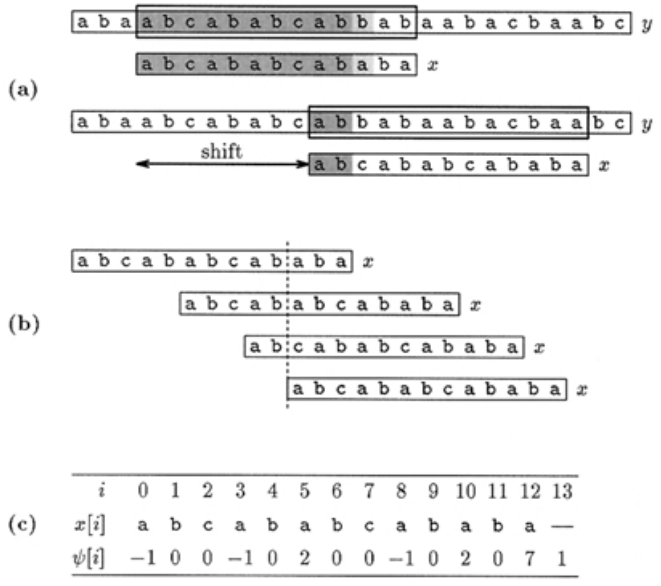


FIGURE 11.2 An illustration of the shift in the Knuth–Morris–Pratt algorithm when searching for the pattern $x = \texttt{abcababcababa}$. **(a)** The window on the text y is at position 3. A mismatch occurs at position 10 on x. The matching symbols are shown darkly shaded, and the current analyzed symbols lightly shaded. Avoiding both a backtrack on the text and an immediate mismatch leads to shift the window 8 positions to the right. The string-to-string comparison resumes at position 2 on the pattern. **(b)** The current shift is the consequence of an analysis of the list of the proper borders of $x[0\,..\,9]$ and of the symbol which follow them in x. The prefixes of x that are borders of $x[0\,..\,9] = \texttt{abcabacbab}$ are right-aligned along the discontinuous vertical line. String $x[0\,..\,4] = \texttt{abcab}$ is a border of $x[0\,..\,9]$, but is followed by symbol $\texttt{a}$ which is identical to $x[10]$. String $x[0\,..\,1]$ is the expected border, since it is followed by symbol $\texttt{c}$. **(c)** The values of the function ψ for pattern x.

is, if $|v| < |u|$.) This introduces the function ψ defined for each $i \in \{0, 1, \ldots, m-1\}$ by

$$\psi[i] = \max\{k \mid (0 \le k < i,\ x[i-k\,..\,i-1] = x[0\,..\,k-1],\ x[k] \ne x[i]) \text{ or } (k = -1)\}\ .$$

Then, after a shift of length $i - \psi[i]$, the symbol comparisons can resume with $y[p+i]$ against $x[\psi[i]]$ in the case where $\psi[i] \ge 0$, and $y[p+i+1]$ against $x[0]$ otherwise. Doing so, we miss no occurrence of x in y, and avoid a backtrack on the text. The previous statement is still valid when no mismatch occurs, that is when $i = m$, if we consider for a moment the string $x\$$ instead of x, where $\$$ is a symbol of alphabet Σ occurring nowhere in x. This amounts to completing the definition of function ψ by setting

$$\psi[m] = \max\{k \mid 0 \le k < m,\ x[m-k\,..\,m-1] = x[0\,..\,k-1]\}\ .$$

The Knuth–Morris–Pratt string-matching algorithm is given in pseudocode below as the procedure KNUTH-MORRIS-PRATT-MATCHER. The values of function ψ are first computed by the function BETTER-PREFIX-FUNCTION given after. The value of the variable j is equal to $p+i$ in the remainder of the code (the search phase of the algorithm strictly speaking); this simplifies the code, and points out the sequential processing of the text. Observe that the preprocessing phase applies a similar method to the pattern itself, as if $y = x[1\,..\,m-1]$.

```
KNUTH-MORRIS-PRATT-MATCHER(x, m, y, n)
1  ψ ← BETTER-PREFIX-FUNCTION (x, m)
2  i ← 0
3  for j from 0 up to n − 1
4     loop while i ≥ 0 and y[j] ≠ x[i]
5              loop i ← ψ[i]
6        i ← i + 1
7        i i = m
8              then report j + 1 − m
9                   i ← ψ[m]
```

```
BETTER-PREFIX-FUNCTION(x, m)
1  ψ[0] ← −1
2  i ← 0
3  for j from 1 up to m − 1
4     loop if x[j] = x[i]
5              then ψ[j] ← ψ[i]
6              else ψ[j] ← i
7                   loop i ← ψ[i]
8                      while i ≥ 0 and x[j] ≠ x[i]
9           i ← ψ + 1
10 ψ[m] ← i
11 return ψ
```

The algorithm has a worst-case running time in $O(m + n)$, and requires $O(m)$ extra-space to store function ψ. The linear running time results from the fact that the number of symbol comparisons performed during the preprocessing phase and the search phase is less than $2m$ and $2n$, respectively. All the previous bounds are independent of the size of the alphabet.

The Boyer–Moore Algorithm

The Boyer–Moore algorithm is considered as the most efficient string-matching algorithm in usual applications. A simplified version of it, or the entire algorithm, is often implemented in text editors for the "search" and "substitute" commands.

The scan operation proceeds from right to left in the window on the text, instead of left to right as in the Knuth–Morris–Pratt algorithm. In case of a mismatch, the algorithm uses two functions to shift the window. These two shift functions are called the better-factor shift function and the bad-symbol shift function. In the two next paragraphs, we explain the goal of the two functions and we give procedures to precompute their values.

We first explain the aim of the better-factor shift function. Let p be the current (left) position of the window on the text. Assume that a mismatch occurs between symbols $y[p + i]$ and $x[i]$ for some i, $0 \leq i < m$ (an illustration is given in Fig. 11.3). Then, we have $y[p + i] \neq x[i]$ and $y[p + i + 1 \,..\, p + m - 1] = x[i + 1 \,..\, m - 1]$. The better-factor shift consists in aligning the factor $y[p + i + 1 \,..\, p + m - 1]$ with its rightmost occurrence $x[k + 1 \,..\, m - 1 - i + k]$ in x preceded by a symbol $x[k]$ different from $x[i]$ to avoid an immediate mismatch. If no such factor exists, the shift consists in aligning the longest suffix of $y[p + i + 1 \,..\, p + m - 1]$ with a matching prefix of x. The better-factor shift function β is defined by

$$\begin{aligned} \beta[i] \quad = \quad & \min\{i - k \mid (0 \leq k < i,\ x[k + 1 \,..\, m - 1 - i + k] = x[i + 1 \cdot\cdot m - 1],\ x[k] \neq x[i]) \\ & \text{or } (i - m \leq k < 0,\ x = x[i - k \,..\, m - 1]x[m - i + k \,..\, m - 1])\} \end{aligned}$$

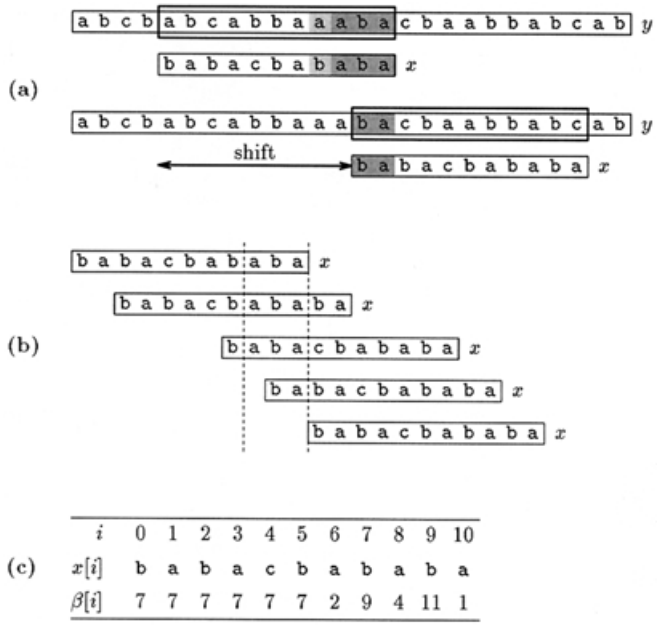


FIGURE 11.3 An illustration of the better-factor shift in the Boyer–Moore algorithm when searching for the pattern $x = \texttt{babacbababa}$. **(a)** The window on the text is at position 4. The string-to-string comparison, which proceeds from right to left, stops with a mismatch at position 7 on x. The window is shifted 9 positions to the right to avoid an immediate mismatch. **(b)** Indeed, the string $x[8\,..\,10] = \texttt{aba}$ is repeated three times in x, but is preceded each time by symbol $x[7] = \texttt{b}$. The expected matching factor in x is then the prefix $\texttt{ba}$ of x. The factors of x identical with $\texttt{aba}$ and the prefixes of x ending with a suffix of $\texttt{aba}$ are right-aligned along the rightmost discontinuous vertical line. **(c)** The values of the shift function β for pattern x.

for each $i \in \{0, 1, \ldots, m-1\}$. The value $\beta[i]$ is then exactly the length of the shift induced by the better-factor shift. The values of function β are computed by the function given below as the function BETTER-FACTOR-FUNCTION. An auxiliary table, namely f, is used; it is an analogue of the function ψ used in the Knuth–Morris–Pratt algorithm, but defined this time for the reverse pattern; it is indexed from 0 to $m-1$. The running time of the function BETTER-FACTOR-FUNCTION is $O(m)$.

```
BETTER-FACTOR-FUNCTION(x, m)
1   for j from 0 up to m − 1
2       loop β[j] ← 0
3   i ← m
4   for j from m − 1 down to 0
5       loop f[j] ← i + 1
6           while i < m and x[j] ≠ x[i]
7               loop if β[i] = 0
8                       then β[i] ← i − j
9                    i ← f[i] − 1
10          i ← i − 1
11  for j from 0 up to m − 1
12      loop if β[j] = 0
13              then β[j] ← i + 1
14          if j = i
15              then i ← f[i] − 1
16  return β
```

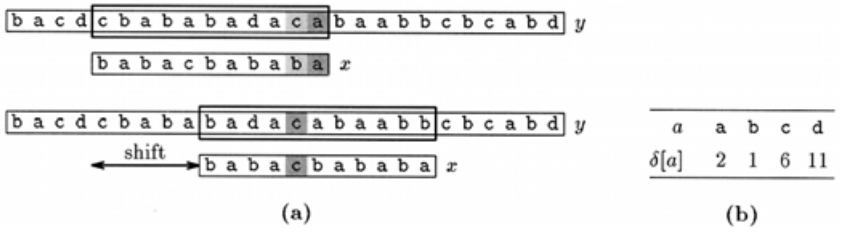


FIGURE 11.4 An illustration of the bad-symbol shift in the Boyer–Moore algorithm when searching for the pattern $x = \texttt{babacbababa}$. **(a)** The window on the text is at position 4. The string-to-string comparison stops with a mismatch at position 9 on x. Considering only this position and the unexpected symbol occurring at this position, namely symbol $y[13] = \texttt{c}$, leads to shift the window 5 positions to the right. Notice that if the unexpected symbol were `a` or `d`, the applied shift would have been 1 and 10, respectively. **(b)** The values of the table δ for pattern x when alphabet Σ is reduced to { `a`, `b`, `c`, `d` }.

We now come to the aim of the bad-symbol shift function (Fig. 11.4 shows an illustration). Consider again the text symbol $y[p+i]$ that causes a mismatch. Assume first that this symbol occurs in $x[0\,..\,m-2]$. Then, let k be the position of the rightmost occurrence of $y[p+i]$ in $x[0\,..\,m-2]$. The window can be shifted $i-k$ positions to the right if $k < i$, and only one position otherwise, without missing an occurrence of x in y. Assume now that symbol $y[p+i]$ does not occur in x. Then, no occurrence of x in y can overlap the position $p+i$ on the text, and thus, the window can be shifted $i+1$ positions to the right. Let δ be the table indexed on alphabet Σ, and defined for each symbol $a \in \Sigma$ by

$$\delta[a] = \min\{m\} \cup \{m-1-j \mid 0 \le j < m-1,\ x[j] = a\}\,.$$

According to the above discussion, the bad-symbol shift for the unexpected text symbol a aligned with the symbol at position i on the pattern is the value

$$\gamma[a, i] = \max\{\delta[a] + i - m + 1,\ 1\}\,,$$

which defines the bad-symbol shift function γ on $\Sigma \times \{0, 1, \ldots, m-1\}$. We give now the code of the function LAST-OCCURRENCE-FUNCTION that computes table δ. Its running time is $O(m + \text{card}\Sigma)$.

```
LAST-OCCURRENCE-FUNCTION(x, m)
1  for each a ∈ Σ
2     loop δ[a] ← m
3  for j from 0 up to m − 2
4     loop δ[x[j]] ← m − 1 − j
5  return δ
```

The shift applied in the Boyer–Moore algorithm in case of a mismatch is the maximum between the better-factor shift and the bad-symbol shift. In case of a whole match, the shift applied to the window is m minus the length of the longest proper border of x, that is also the value $\beta[0]$ (this value is indeed what is called "the period" of the pattern). The code of the entire algorithm is given below.

```
BOYER-MOORE-MATCHER(x, m, y, n)
1  β ← BETTER-FACTOR-FUNCTION(x, m)
2  δ ← LAST-OCCURRENCE-FUNCTION(x, m)
3  p ← 0
4  while p ≤ n − m
5     loop i ← m − 1
6          while i ≥ 0 and y[p + i] = x[i]
7             loop i ← i − 1
8          if i ≥ 0
9             then p ← p + max{β[i], δ[y[p + i]] + i − m + 1}
10            else report p
11                 p ← pβ[0]
```

The worst-case running time of the algorithm is quadratic. It is surprising however that, when used to search only for the first occurrence of the pattern, the algorithm runs in linear time. Slight modifications of the strategy yield linear-time algorithms. When searching for $a^{m-1}b$ in a^n with $a, b \in \Sigma$ and $a \neq b$, the algorithm considers only $\lfloor n/m \rfloor$ symbols of the text. This bound is the absolute minimum for any string-matching algorithm in the model where only the pattern is preprocessed. Indeed, the algorithm is expected to be extremely fast on large alphabets (relative to the length of the pattern).

Practical String-Matching Algorithms

The bad-symbol shift function introduced in the Boyer–Moore algorithm is not very efficient for small alphabets, but when the alphabet is large compared with the length of the pattern (as it is often the case with the ASCII table and ordinary searches made under a text editor), it becomes very useful. Using only the corresponding table produces some efficient algorithms for practical searches. We describe one of these algorithms below.

Consider a position p of the window on the text, and assume that the symbols $y[p+m-1]$ and $x[m-1]$ are identical. If $x[m-1]$ does not occur in the prefix $x[0\,..\,m-2]$ of x, the window can be shifted m positions to the right after the string-to-string comparison between $y[p\,..\,p+m-2]$ and $x[0\,..\,m-2]$ is performed. Otherwise, let k be the position of the rightmost occurrence of $x[m-1]$ in $x[0\,..\,m-2]$; the window can be shifted $m-1-k$ positions to the right. This shows that $\delta[y[p+m-1]]$ is also a valid shift in the case where $y[p+m-1] = x[m-1]$. The underlying algorithm is the Horspool algorithm.

The pseudocode of the Horspool algorithm is given below. To prevent two references to the rightmost symbol in the window at each scan and shift operation, table δ is slightly modified: $\delta[x[m-1]]$ contains the sentinel value 0, after its previous value is saved in variable t. The value of the variable j is the value of the expression $p+m-1$ in the discussion above.

```
HORSPOOL-MATCHER(x, m, y, n)
1  δ ← LAST-OCCURRENCE-FUNCTION(x, m)
2  t ← δ[x[m − 1]]
3  δ[x[m − 1]] ← 0
4  j ← m − 1
5  while j < n
6     loop s ← δ[y[j]]
7          if s ≠ 0
8             then j ← j + s
9             else if y[j − m + 1 .. j − 1] = x[0 .. m − 2]
10                    then report j − m + 1
11                 j ← j + t
```

Just like the brute force algorithm, the Horspool algorithm has a quadratic worst-case time complexity. But its behavior may be at least as good as the behavior of the Boyer–Moore algorithm in practice because the Horspool algorithm is simpler. An example showing the behavior of both algorithms is given in Fig. 11.5.

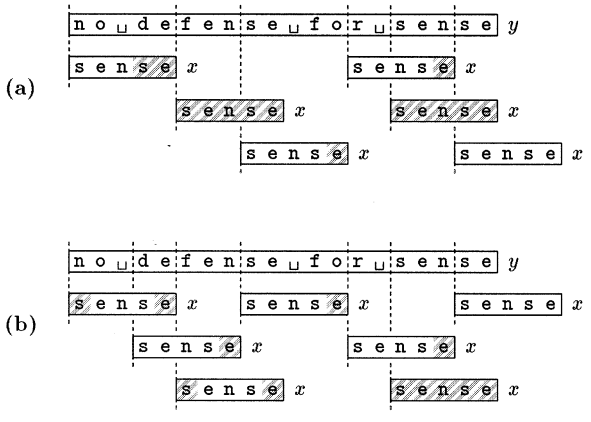


FIGURE 11.5 An illustration of the behavior of two fast string-matching algorithms when searching for the pattern $x = \texttt{sense}$ in the text $y = \texttt{no␣defense␣for␣sense}$. The successive positions of the window on the text are suggested by the alignments of x with the corresponding factors of y. The symbols of x considered during each scan operation are shown hachured. **(a)** Behavior of the Boyer–Moore algorithm. The first and second shifts result from the better-shift function, the third and fourth from the bad-symbol function, and the fifth from a shift of the length of x minus the length of its longest proper border (the period of x). **(b)** Behavior of the Horspool algorithm. We assume here that the four leftmost symbols in the window are compared with the symbols of $x[0\,..\,3]$ from left to right.

The Aho–Corasick Algorithm

The UNIX operating system provides standard text-file facilities. Among them is the series of `grep` commands that locate patterns in files. We describe in this section the Aho–Corasick algorithm underlying an implementation of the `fgrep` command of UNIX. It searches files for a finite and fixed set of strings (the dictionary), and can for instance output lines containing at least one of the strings.

If we are interested in searching for all occurrences of all strings of a dictionary, a first solution consists in repeating some string-matching algorithm for each string. Considering a dictionary X containing k strings and a text y, the search runs in that case in time $O(m + n \times k)$, where m is the sum of the length of the strings in X, and n the length of y. But this solution is not efficient, since text y has to be read k times. The solution described in this section provides both a sequential read of the text and a total running time which is $O(m + n)$ on a fixed alphabet. The algorithm can be viewed as a direct extension of a weaker version of the Knuth–Morris–Pratt algorithm.

The search is done with the help of an automaton that stores the situations encountered during the process. At a given position on the text, the current state is identified with the set of pattern prefixes ending here. The state represents all the factors of the pattern that can possibly lead to occurrences. Among the factors, the longest contains all the information necessary to continue the search. So, the search is realized with an automaton, denoted by $\mathcal{D}(X)$, of which states are in one-to-one correspondence with the prefixes of X. Implementing completely the transition function of $\mathcal{D}(X)$ would required a size $O(m \times \text{card}\Sigma)$. Instead of that, the Aho–Corasick algorithm requires only $O(m)$ space. To get this space complexity, a part of the transition function is made explicit in the data, and the other part is computed with the help of a

failure function. For the first part, we assume that for any input (p, a), the function denoted by TARGET returns some state q if the triple (p, a, q) is an edge in the data, and the value NIL otherwise. The second part uses the failure function *fail,* which is an analogue of the function ψ used in the Knuth–Morris–Pratt algorithm. But this time, the function is defined on the set of states, and for each state p different from the initial state,

$$fail[p] \quad = \quad \text{the state identified with the longest proper suffix of the prefix identified with } p \text{ that is also a prefix of a string of } X .$$

The aim the failure function is to defer the computation of a transition from the current state, say p, to the computation of the transition from the state $fail[p]$ with the same input symbol, say a, when no edge from p labeled by symbol a is in the data; the initial state, which is identified with the empty string, is the default state for the statement. We give below the pseudocode of the function NEXT-STATE that computes the transitions in the representation. The initial state is denoted by i.

```
NEXT-STATE(p, a, i)
1  while p ≠ NIL and TARGET(p, a)= NIL
2     loop p ← fail[p]
3  if p ≠ NIL
4     then q ← TARGET(p, a)
5     else  q ← i
6  return q
```

The preprocessing phase of the Aho–Corasick algorithm builds the explicit part of $\mathcal{D}(X)$ including function *fail.* It is divided itself into two phases.

The first phase of the preprocessing phase consists in building a sub-automaton of $\mathcal{D}(X)$. It is the **trie** of X (the digital tree in which branches spell the strings of X and edges are labeled by symbols) having as initial state the root of the trie and as terminal states the nodes corresponding to strings of X (an example is given in Fig. 11.6). It differs from $\mathcal{D}(X)$ in two points:

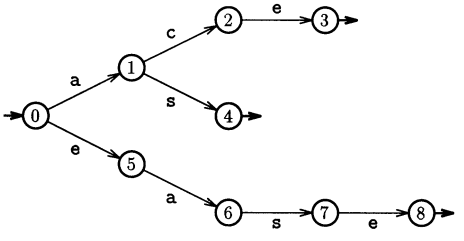


FIGURE 11.6 The trie-like automaton of the pattern $X = \{\texttt{ace}, \texttt{as}, \texttt{ease}\}$. The initial state is distinguished by a thick ingoing arrow, each terminal state by a thick outgoing arrow. The states are numbered from 0 to 8, according to the order in which they are created by the construction statement described in the present section. State 0 is identified with the empty string, state 1 with `a`, state 2 with `ac`, state 3 with `ace`, and so on. The automaton accepts the language X.

- It contains only the forward edges;
- It accepts only the set X.

(An edge (p, a, q) in the automaton is said to be forward if the prefix identified with q is in the form ua where u is the prefix corresponding to p.) The function given below as the function TRIE-LIKE-AUTOMATON computes the automaton corresponding to the trie of X by returning its initial state. The terminal mark of each state r is managed through the attribute $terminal[r]$; the mark is either TRUE or FALSE depending on whether state r is terminal or not. We assume that the function NEW-STATE creates and returns a new state, and that the procedure MAKE-EDGE adds a given new edge to the data.

```
TRIE-LIKE-AUTOMATON(X)
1  i ← NEW-STATE
2  terminal[i] ← FALSE
3  for string x from first to last string of X
4     loop p ← i
5          for symbol a from first to last symbol of x
6             loop q ← TARGET(p, a)
7                  if q = NIL
8                     then q ← NEW-STATE
9                          terminal[q] ← FALSE
10                         MAKE-EDGE(p, a, q)
11                 p ← q
12       terminal[p] ← TRUE
13 return i
```

The second step of the preprocessing phase consists mainly in precomputing the failure function. This is done by a breadth-first traversal of the trie-like automaton. The corresponding pseudocode is given below as the procedure MAKE-FAILURE-FUNCTION.

```
MAKE-FAILURE-FUNCTION(i)
1  fail[i] ← NIL
2  θ ← EMPTY-QUEUE
3  ENQUEUE(θ, i)
4  while not QUEUE-IS-EMPTY(θ)
5     loop p ← DEQUEUE(θ)
6          for each symbol a such that TARGET(p, a) ≠ NIL
7             loop q ← TARGET(p, a)
8                  fail[q] ← NEXT-STATE(fail[p], a, i)
9                  if terminal[fail[q]]
10                    then terminal[q] ← TRUE
11                 ENQUEUE (θ, q)
```

During the computation, some states can be made terminal. This occurs when the state is identified with a prefix that ends with a string of X (an illustration is given in Fig. 11.7).

The complete dictionary-matching algorithm, implemented in the pseudocode below as the procedure AHO-CORASICK-MATCHER, starts with the two steps of the preprocessing; the search follows, which simulates automaton $\mathcal{D}(X)$. It is understood that the empty string does not belong to X.

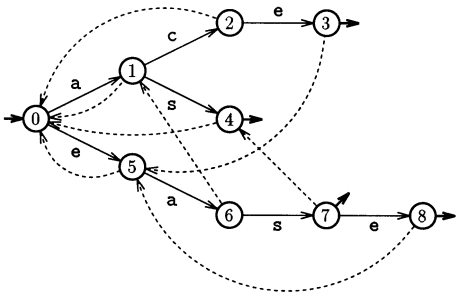


FIGURE 11.7 The explicit part of the automaton $\mathcal{D}(X)$ of the pattern $X = \{\texttt{ace}, \texttt{as}, \texttt{ease}\}$. Compared to the trie-like automaton of X displayed in Fig. 11.6, state 7 has been made terminal; this is because the corresponding prefix, namely `eas`, ends with the string `as` that is in X. The failure function *fail* is depicted with discontinuous non-labeled directed edges.

```
AHO-CORASICK-MATCHER(X, y)
1  i ← TRIE-LIKE-AUTOMATON(X)
2  MAKE-FAILURE-FUNCTION(i)
3  p ← i
4  for symbol a from first to last symbol of y
5     loop p ← NEXT-STATE(p, a, i)
6          if terminal[p]
7             then report an occurrence
```

The total number of tests "TARGET$(p, a) =$ NIL" performed by function NEXT-STATE during its calls by procedure MAKE-FAILURE-FUNCTION and during its calls by the search phase of the algorithm are bounded by $2m$ and $2n$, respectively, similarly as the bounds of comparisons in the Knuth–Morris–Pratt algorithm. Using a total order on the alphabet, the running time of function TARGET is both $O(\log k)$ and $O(\log \text{card}\Sigma)$, since the maximum number of edges outgoing a state in the data representing automaton $\mathcal{D}(X)$ is bounded both by k and by $\text{card}\Sigma$. Thus, the entire algorithm runs in time $O(m+n)$ on a fixed alphabet, and in time $O((m+n) \times \log \min\{k, \text{card}\Sigma\})$ in the general case. The algorithm requires $O(m)$ extra space to store the data and to implement the queue used during the breadth-first traversal executed in procedure MAKE-FAILURE-FUNCTION.

Let us discuss the question of reporting occurrences of pattern X (line 7 of procedure AHO-CORASICK-MATCHER). The simplest way of doing it is to report the ending positions of occurrences. This remains to output the value of the position of the current symbol in the text. A second possibility is to report the whole set of strings in X ending at the current position. To do so, the attribute *terminal* has to be transformed. First, for a state r, *terminal*$[r]$ is the set of the string of X that are suffixes of the string corresponding to r. Second, to avoid a quadratic behavior, sets are manipulated by their identifiers only.

Small Patterns

For most text-searching problems, the length of the pattern is small, and is no more than the word size. Representing the state of the search as an integer, and using binary operations to compute the transitions from state to state give some efficient algorithms easy to implement. We present below one algorithm of this class.

Let X be a dictionary of k strings, and m be the sum of the length of the strings in X. Now, consider the automaton $\mathcal{N}(X)$ obtained from the k straightforward deterministic automata accepting the k strings by

- Merging the k initial states into one initial state, say i,

- Adding the edges in the form (i, a, i), for each symbol $a \in \Sigma$.

The automaton $\mathcal{N}(X)$ is nondeterministic, and it accepts the language $\Sigma^* X$ (an example is given in Fig. 11.8).

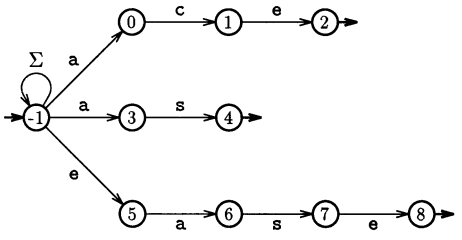


FIGURE 11.8 A straightforward nondeterministic automaton with only one initial state that accepts the language $\Sigma^* X$ for pattern $X = \{\texttt{ace}, \texttt{as}, \texttt{ease}\}$. The edge labeled by Σ denotes the cardΣ edges labeled by the cardΣ distinct symbols in Σ.

The search for occurrences of strings in X is performed with a simulation of the deterministic automaton recognizing $\Sigma^* X$. Indeed, the determinization of $\mathcal{N}(X)$ is not performed, but is just simulated via the subset construction: at a given time, the automaton is not in a given state, but in a set of states. This subset is recomputed whenever necessary in the execution of the search.

Let us number the states of $\mathcal{N}(X)$ from -1 to $m-1$ using a preorder tree walk (this is the case for the example given in Fig. 11.8). Let us code the subsets of the set of states of $\mathcal{N}(X)$ minus the initial state by an integer using the following convention: state number j is in the subset if and only if the bit at position j of the binary code (starting at position 0) of the integer is 1. Now, let v be the binary value corresponding to the current subset, let a be the current input symbol, and let v' be the binary value corresponding to the next subset. It is then easy to verify that v' is computed from v from following rule: the jth bit in v' is 1 if and only if

- Either there is an edge labeled by a from the initial state to state number j,
- Either there is an edge from state number $j-1$ to state number j and the bit at position $j-1$ in v is 1.

Consider now a binary value f defined for each $j, 0 \leq j \leq m-1$, by:

$$\text{the } j\text{th bit of } f \text{ is 1 if and only if}$$
$$\text{there exists an edge labeled by } a \text{ from initial state to state number } j \ ,$$

and the binary value table σ indexed on alphabet Σ, and defined for each symbol $a \in \Sigma$ and each bit-position $j, 0 \leq j \leq m-1$, by:

$$\text{the } j\text{th bit of } \sigma[a] \text{ is 1 if and only if}$$
$$\text{number } j \text{ corresponds to a target state of some edge labeled by } a \ .$$

Then, v, v' and a satisfy the relation

$$v' = (2v \vee f) \wedge \sigma[a] \ ,$$

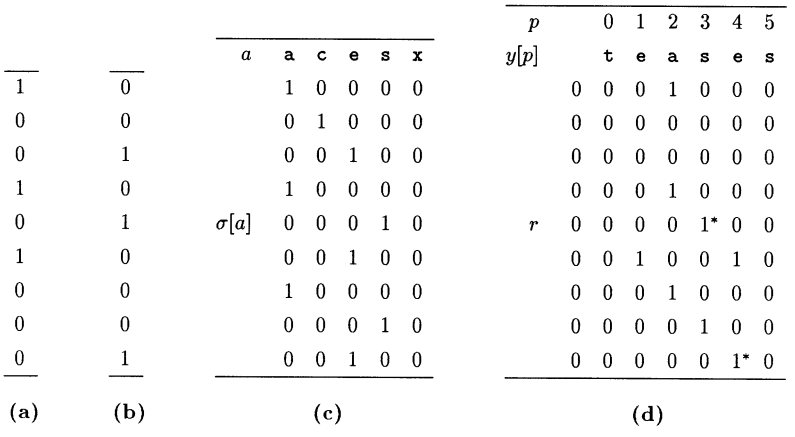
denoting respectively by $\vee$ and $\wedge$ the binary operations "or" and "and." It only remains to be able to test whether one of the states represented by v' is a terminal state or not. Let t be a binary value such that, for each $j, 0 \leq j \leq m-1$,

$$\text{the } j\text{th bit of } t \text{ is 1 if and only if state number } j \text{ is a terminal state.}$$

The corresponding test is then

$$v' \wedge t \neq 0 \,.$$

An example is given in Fig. 11.9.



		a	a	c	e	s	x
1	0		1	0	0	0	0
0	0		0	1	0	0	0
0	1		0	0	1	0	0
1	0		1	0	0	0	0
0	1	$\sigma[a]$	0	0	0	1	0
1	0		0	0	1	0	0
0	0		1	0	0	0	0
0	0		0	0	0	1	0
0	1		0	0	1	0	0

p		0	1	2	3	4	5
$y[p]$		t	e	a	s	e	s
	0	0	0	1	0	0	0
	0	0	0	0	0	0	0
	0	0	0	0	0	0	0
	0	0	0	1	0	0	0
r	0	0	0	0	1*	0	0
	0	0	1	0	0	1	0
	0	0	0	1	0	0	0
	0	0	0	0	1	0	0
	0	0	0	0	0	1*	0

(a) **(b)** **(c)** **(d)**

FIGURE 11.9 An illustration of the behavior of the TINY-MATCHER algorithm when searching for the pattern $X = \{\texttt{ace}, \texttt{as}, \texttt{ease}\}$ in the text $y = \texttt{teases}$. **(a)** The binary value f, which codes potential transitions from the initial state. Bits of a given binary value are written downward, bit at position 0 right at the top. **(b)** The binary value t, which codes terminal states. **(c)** The values of the table σ for pattern X. Symbol x means a symbol that does not appear in the strings in X. **(d)** The successive binary values of the integer r corresponding to the subset of states reached during the search. Asterisked bits indicate that a terminal state is reached, which means that a string in X ends at the current position on the text.

We give now the code of the function TINY-AUTOMATON that computes binary values f and t, and table σ. Its running time is $O(m + \text{card}\Sigma)$.

```
TINY-AUTOMATON(X)
1   f ← 0
2   t ← 0
3   for each a ∈ Σ
4       loop σ[a]0
5   r ← 1
6   for string x from first to last string of X
7       loop f ← f ∨ r
8           for symbol a from first to last symbol of x
9               loop σ[a] ← σ[a] ∨ r
10                  r′ ← r
11                  r ← 2r
12          t ← t ∨ r′
13  return (f, t, σ)
```

The code of the entire algorithm is given below.

```
TINY-MATCHER(X, y)
1  (f, t, σ) ← TINY-AUTOMATON(X)
2  r ← f
3  for symbol a from first to last symbol of y
4     loop r ← (2r ∨ f) ∧ σ[a]
5          if r ∧ t ≠ 0
6             then report an occurrence
```

The total-running time of the algorithm is $O(m + n + \mathrm{card}\Sigma)$, where n is the length of y. It requires $O(\mathrm{card}\Sigma)$ extra space to store table σ.

The technique developed in the current section can easily be generalized, for example in allowing do not care symbols.

11.3 Indexing Texts

This section deals with the pattern-matching problem applied to fixed texts. Solutions consist in building an index on the text that speeds up further searches. The indexes that we consider here are data structures that contain all the suffixes and therefore all the factors of the text. Two types of structures are presented: **suffix trees** and suffix automata. They are both compact representations of suffixes in the sense that their sizes are linear in the length of the text, although the sum of lengths of suffixes of a string is quadratic. Moreover, their constructions take linear time on fixed alphabets. On an arbitrary finite alphabet Σ, assumed to be ordered, a $\log \mathrm{card}\Sigma$ factor has to be added to almost all running times given in the following. This corresponds to the branching operation involved in the respective data structures.

Indexes are powerful tools that have many applications. Here is a nonexhaustive list of them, assuming an index on the text y.

- Membership: testing if a string x occurs in y.
- Occurrence number: producing the number of occurrences of a string x in y.
- List of positions: analogue of the string-matching problem of Section 11.2.
- Longest repeated factor: locating the longest factor of y occurring at least twice in y.
- Longest common factor: finding a longest string that occurs both in a string x and in y.

Solutions to some of these problems are first considered with suffix trees, then with suffix automata. Suffix arrays is another data structure that provides solution running slightly slower. They are shortly described at the end of the section.

Suffix Trees

The suffix tree $\mathcal{T}(y)$ of a nonempty string y of length n is a data structure containing all the suffixes of y. In order to simplify the statement, it is assumed that y ends with a special symbol of the alphabet occurring nowhere else in y (this special symbol is denoted by \$ in the examples). The suffix tree of y is a trie which satisfies the following properties:

- The branches from the root to the external nodes spell the nonempty suffixes of y, and each external node is marked by the position of the occurrence of the corresponding suffix in y;
- The internal nodes have at least two successors, except if y is a one-length string;
- The edges outgoing an internal node are labeled by factors starting with different symbols;
- Any string that labels an edge is represented by the couple of integers corresponding to its position in y and its length.

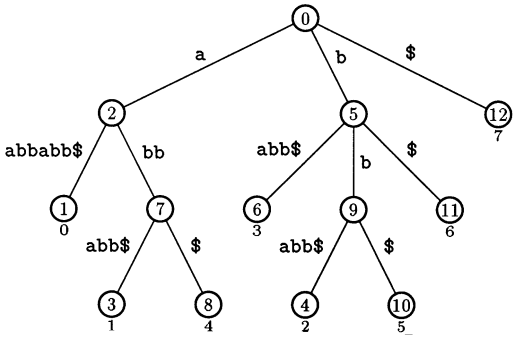


FIGURE 11.10 The suffix tree $\mathcal{T}(y)$ of the string $y = \texttt{aabbabb\$}$. The nodes are numbered from 0 to 12, according to the order in which they are created by the construction algorithm described in the present section. Each of the eight external nodes of the trie is marked by the position of the occurrence of the corresponding suffix in y. Hence, the branch (0, 5, 9, 4), running from the root to an external node, spells the string `bbabb$`, which is the suffix of y starting at position 2.

(An example of suffix tree is displayed in Fig. 11.10.) The special symbol at the end of y avoids marking nodes, and implies that $\mathcal{T}(y)$ has exactly n external nodes. The other properties then imply that the total size of $\mathcal{T}(y)$ is $O(n)$, which makes it possible to design a linear-time construction of the data structure. The algorithm described in the following and implemented by the procedure SUFFIX-TREE given further has this time complexity.

The construction algorithm works as follows. It inserts the nonempty suffixes $y[i\,..\,n-1]$, $0 \leq i < n$, of y in the data structure from the longest to the shortest suffix. In order to explain how this is performed, we introduce the two notations

$$h_i = \text{the longest prefix of } y[i\,..\,n-1] \text{ that is a prefix of some stricly longest suffix of } y\ ,$$

and

$$t_i = \text{the string } w \text{ such that } y[i\,..\,n-1] \text{ is identical with } h_i w\ ,$$

defined for each $i \in \{1, \ldots, n-1\}$. The strategy to insert the suffixes is precisely based on these definitions. Initially, the data structure contains only the string y. Then, the insertion of the string $y[i\,..\,n-1]$, $1 \leq i < n$, proceeds in two steps:

- first, the "head" in the data structure, that is, the node h corresponding to string h_i, is located, possibly breaking an edge;
- second, a node called the "tail," say t, is created, added as successor of node h, and the edge from h to t is labeled with string t_i.

The second step of the insertion is clearly performed in constant time. Thus, finding the head is critical for the overall performance of the construction algorithm. A brute-force method to find the head consists in spelling the current suffix $y[i\,..\,n-1]$ from the root of the trie, giving an $O(|h_i|)$ time complexity for the insertion at step i, and an $O(n^2)$ running time to build the suffix tree $\mathcal{T}(y)$. Adding "short-circuit" links leads to an overall $O(n)$ time complexity, although there is no guarantee that the insertion at any step i is realized in constant time.

Observe that in any suffix tree, if the string corresponding to a given internal node p in the data structure is in the form au with $a \in \Sigma$ and $u \in \Sigma^*$, then there exists an unique internal node corresponding to the

string u. From this remark are defined the suffix links by

$$\begin{aligned} link[p] \quad = \quad & \text{the node } q \text{ corresponding to the string } u \\ & \text{when } p \text{ corresponds to the string } au \text{ for some symbol } a \in \Sigma \end{aligned}$$

for each internal node p that is different from the root. The links are useful when computing h_i from h_{i-1} because of the property: if h_{i-1} is in the form aw for some symbol $a \in \Sigma$ and some string $w \in \Sigma^*$, then w is a prefix of h_i.

We explain in three following paragraphs how the suffix links help to find the successive heads efficiently. We consider a step i in the algorithm assuming that $i \geq 1$. We denote by g the node that corresponds to the string h_{i-1}. The aim is both to insert $y[i\,..\,n-1]$ and to find the node h corresponding to the string h_i. We first study the most general case of the insertion of the suffix $y[i\,..\,n-1]$. Particular cases are studied after.

We assume in the present case that the predecessor of g in the data structure, say g', is both defined and different from the root. Then h_{i-1} is in the form auv where $a \in \Sigma$, $u, v \in \Sigma^*$, au corresponds to the node g', and v labels the edge from g' to g. Since the string uv is a prefix of h_i, it can be fully spelled from the root. Moreover, the spelling operation of uv from the root can be short-circuited by spelling only the string v from the node $link[g']$. The node q reached at the end of the spelling operation (possibly breaking the last partially taken down edge) is then exactly the node $link[g]$. It remains to spell the string t_{i-1} from q for completely inserting the string $y[i\,..\,n-1]$. The spelling stops on the expected node h (possibly breaking again an edge) which becomes the new head in the data structure. The suffix of t_{i-1} that has not been spelled so far, is exactly the string t_i. (An example for the whole previous statement is given in Fig. 11.11.)

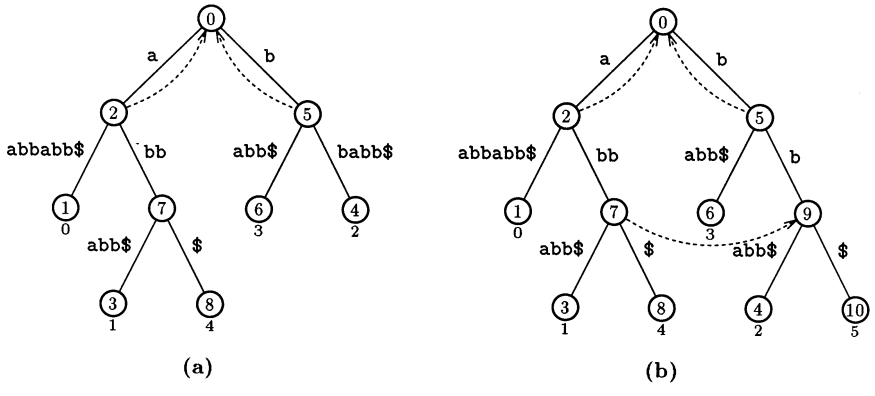


FIGURE 11.11 During the construction of the suffix tree $\mathcal{T}(y)$ of the string $y = \texttt{aabbabb\$}$, the step 5, that is, the insertion of the suffix $\texttt{bb\$}$. The defined suffix link are depicted with discontinuous nonlabeled directed edges. **(a)** Initially, the head in the data structure is node 7, and its suffix link is not yet defined. The predecessor of node 7, node 2, is different from the root, and the factor of y that is spelled from the root to node 7, namely $h_4 = \texttt{abb}$, is in the form auv, where $a \in \Sigma$, $u \in \Sigma^*$, and v is the string of Σ^* labeling the edge from node 2 to node 7. Here, $a = \texttt{a}$, u is the empty string, and $v = \texttt{bb}$. Then, the string $uv = \texttt{bb}$ is spelled from the node linked with node 2, that is, from node 0; the spelling operation stops on the edge from node 5 to node 4; this edge is broken, which creates node 9. Node 9 is linked to node 7. The string $t_4 = \$$ is spelled from node 9; the spelling operation stops on node 9, which becomes the new head in the data structure. **(b)** Node 10 is created, added as successor of node 9, and the edge from node 9 to node 10 is labeled by the string \$, remainder of the last spelling operation.

The second case is when g is a (direct) successor of the root. The string h_{i-1} is then in the form au where $a \in \Sigma$ and $u \in \Sigma^*$. Similarly to the above case, the string u can be fully spelled from the root. The spelling of u gives a node q, which is then linked with g. Afterwards, the string t_{i-1} is spelled from q.

The last case is when g is the root itself. The string t_{i-1} minus its first symbol has to be spelled from the root. Which ends the study of all the possible cases that can arise.

The important aspect of the algorithm is the use of two different implementations for the two spelling operations pointed out above. The first one, given in the pseudocode below as the function FAST-FIND, deals with the situation where we know in advance that a given factor $y[j\,..\,j+k-1]$ of y can be fully spelled from a given node p of the trie. It is then sufficient to scan only the first symbols of the labels of the encountered nodes, which justifies the name of the function. The second implementation of the spelling operation spells a factor $y[j\,..\,j+k-1]$ of y from a given node p too, but, this time, the spelling is performed symbol by symbol. The corresponding function is implemented after as the function SLOW-FIND. Before giving the pseudocode of the functions, we precise the notations used in the following.

- For any input (y, p, j), the function SUCCESSOR-BY-ONE-SYMBOL returns the node q such that q is a successor of the node p and the first symbol of the label of the edge from p to q is $y[j]$; if such a node q does not exist, it returns NIL.
- For any input (p, q), the function LABEL returns the two integers that represents the label of the edge from the node p to the node q.
- The function NEW-NODE creates and returns a new node.
- For any input (p, j, k, q, ℓ), the function NEW-BREAKING-NODE creates and returns the node r breaking the edge $(p, y[j\,..\,j+k-1], q)$ at the position ℓ in the label $y[j\,..\,j+k-1]$. (Which gives the two edges $(p, y[j\,..\,j+\ell-1], r)$ and $(r, y[j+\ell\,..\,j+k-1], q)$.)

Function FAST-FIND returns a couple of nodes such that the second one is the node reached by the spelling, and the first one is its predecessor.

```
FAST-FIND(y, p, j, k)
1   p' ←NIL
2   while k > 0
3      loop p' ← p
4           q ← SUCCESSOR-BY-ONE-SYMBOL (y, p, j)
5           (r, s) ← LABEL(p, q)
6           if s ≤ k
7              then p ← q
8                   j ← j + s
9                   k ← k − s
10             else p ← NEW-BREAKING-NODE (p, r, s, q, k)
11                  k ← 0
12  return (p', p)
```

Compared to function FAST-FIND, function SLOW-FIND considers an extra-input that is the predecessor of node p (denoted by p'). It considers in addition two extra-outputs that are the position and the length of the factor that remains to be spelled.

```
SLOW-FIND(y, p', p, j, k)
1  b ← FALSE
2  loop   q ← SUCCESSOR-BY-ONE-SYMBOL(y, p, j)
3         if q = NIL
4            then b ← TRUE
5            else (r, s) ← LABEL (p, q)
6                 ℓ ← 1
7                 while ℓ < s and y[j + ℓ] = y[r + ℓ]
8                       loop ℓ ← ℓ + 1
9                 j ← j + ℓ
10                k ← k + ℓ
11                p' ← p
12                if ℓ = s
13                   then p ← q
14                   else p ← NEW-BREAKING-NODE(p, r, s, q, ℓ)
15                        b ← TRUE
16   while b = FALSE
17 return (p', p, j, k)
```

The complete construction algorithm is implemented as the function SUFFIX-TREE given below. The function returns the root of the constructed suffix-tree. Memorizing systematically the predecessors h' and q' of the nodes h and q avoids considering doubly linked tries. The name of the attribute which marks the positions of the external nodes is made explicit.

```
SUFFIX-TREE(y, n)
1  p ← NEW-NODE
2  h' ←NIL
3  h ← p
4  r ← −1
5  s ← n + 1
6  for i from 0 up to n − 1
7    loop if h' = NIL
8            then (h', h, r, s) ← SLOW-FIND(y, NIL,p, r + 1, s − 1)
9            else  (j, k) ← LABEL (h', h)
10                 if h' = p
11                      then(q', q) ← FAST-FIND (y, p, j + 1, k − 1)
12                      else (q', q) ← FAST-FIND (y, link[h'], j, k)
13                 link[h] ← q
14                 (h', h, r, s) ← SLOW-FIND (y, q', q, r, s)
15         t ← NEW-NODE
16         MAKE-EDGE (h, (r, s), t)
17         position[t] ← i
18 return p
```

The algorithm runs in time $O(n)$ (more precisely $O(n \times \log \operatorname{card}\Sigma)$ if we take into account the branching in the data structure). Indeed, the instruction at line 4 in function FAST-FIND is performed less than $2n$ times, and the number of symbol comparisons done at line 7 in function SLOW-FIND is less than n.

Once the suffix tree of y is build, some operations can be performed rapidly. We describe four applications in the following. Let x be a string of length m.

Testing whether x occurs in y or not can be solved in time $O(m)$ by spelling x from the root of the trie symbol by symbol. If the operation succeeds, x occurs in y. Otherwise, we get the longest prefix of x occurring in y.

Producing the number of occurrences of x in y starts identically by spelling x. Assume that x occurs actually in y. Let p be the node at the extremity of the last taken down edge, or be the root itself if x is empty. The expected number, say k, is then exactly the number of external nodes of the sub-trie of root p. This number can be computed by traversing the sub-trie. Since each internal node of the sub-trie has at least two successors, the total size of the sub-trie is $O(k)$, and the traversal of the sub-trie is performed in time $O(k)$ (independently of Σ). The method can be improved by precomputing in time $O(n)$ (independently of Σ) all the values associated with each internal node; the whole operation is then performed in time $O(m)$, whatever is the number of occurrences of x.

The method for reporting the list of positions of x in y proceeds in the same way. The running time needed by the operation is $O(m)$ to locate x in the trie, plus $O(k)$ to report each of the positions associated with the k external nodes.

Finding the longest repeated factor of y remains to compute the "deepest" internal node of the trie, that is, the internal node corresponding to a longest possible factor in y. This is performed in time $O(n)$.

Suffix Automata

The **suffix automaton** $\mathcal{S}(y)$ of a string y is the minimal deterministic automaton recognizing Suff(y), that is, the set of suffixes of y. This automaton is minimal among all the deterministic automata recognizing the same language, which implies that it is not necessarily complete. An example is given in Fig. 11.12.

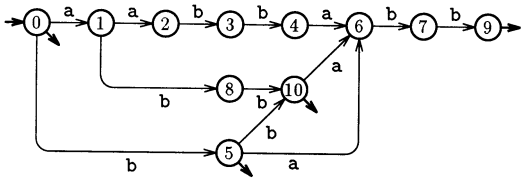


FIGURE 11.12 The suffix automaton $\mathcal{S}(y)$ of the string $y =$ `aabbabb`. The states are numbered from 0 to 10, according to the order in which they are created by the construction algorithm described in the present section. The initial state is state 0, terminal states are states 0, 5, 9, and 10. This automaton is the minimal deterministic automaton accepting the language of the suffixes of y.

The main point about suffix automata is that their size is asymptotically linear in the length of the string. More precisely, given a string y of length n, the number of states of $\mathcal{S}(y)$ is equal to $n+1$ when $n \leq 2$, and is bounded by $n+1$ and $2n-1$ otherwise; as to the number of edges, it is equal to $n+1$ when $n \leq 1$, it is 2 or 3 when $n = 2$, and it bounded by n and $3n-4$ otherwise.

The construction of the suffix automaton of a string y of length n can be performed in time $O(n)$, or, more precisely, in time $O(n \times \log \mathrm{card}\Sigma)$ on an arbitrary alphabet Σ. It makes use of a failure function *fail* defines on the states of $\mathcal{S}(y)$. The set of states of $\mathcal{S}(y)$ identifies with the quotient sets

$$u^{-1}\mathrm{Suff}(y) = \left\{v \in \Sigma^* \mid uv \in \mathrm{Suff}(y)\right\}$$

for the strings u in the whole set of factors of y. One may observe that two sets in the form $u^{-1}\mathrm{Suff}(y)$ are either disjoint or comparable. This allows to set

$$\mathit{fail}[p] = \text{ the smallest quotient set stricly containing the quotient set identified with } p\ ,$$

for each state p of the automaton different from the initial state of the automaton. The function given below as the function SUFFIX-AUTOMATON builds the suffix automaton of y, and returns the initial state, say i, of the automaton. The construction is on-line, which means that at each step of the construction, just after processing a prefix y' of y, the suffix automaton $\mathcal{S}(y')$ is build. Denoting by t the state without outgoing edge in the automaton $\mathcal{S}(y')$, terminal states of $\mathcal{S}(y')$ are implicitly known by the "suffix path" of t, that is, the list of the states

$$t, \mathit{fail}[t], \mathit{fail}[\mathit{fail}[t]], \ldots, i \ .$$

The algorithm uses the function *length* defined for each state p of $\mathcal{S}(y)$ by

$$\mathit{length}[p] = \text{ the length of the longest string spelled from } i \text{ to } p \ .$$

```
SUFFIX-AUTOMATON(y)
1  i ←NEW-STATE
2  terminal[i] ←FALSE
3  length[i] ← 0
4  fail[i] ← NIL
5  t ← i
6  for symbol a from first to last symbol of y
7     loop t ← SUFFIX-AUTOMATON-EXTENSION(i, t, a)
8  p ← t
9  loop terminal[p] ← TRUE
10       p ← fail[p]
11    while p ≠ NIL
12 return i
```

The on-line construction is based on the function SUFFIX-AUTOMATON-EXTENSION that is implemented below. The latter function processes the next symbol, say a, of the string y. If y' is the prefix of y preceding a, it transforms the suffix automaton $\mathcal{S}(y')$ already build into the suffix automaton $\mathcal{S}(y'a)$.

```
SUFFIX-AUTOMATON-EXTENSION(i, t, a)
1  t' ← t
2  t ← NEW-STATE
3  terminal[t] ← FALSE
4  length[t] ← length[t'] + 1
5  p ← t'
6  loop MAKE-EDGE(p, a, t)
7       p ← fail[p]
8    while p ≠ NIL and TARGET(p, a) = NIL
9  if p = NIL
10   then fail[t] ← i
11   else q ← TARGET(p, a)
12        if length[q] = length[p] + 1
13             then fail[t] ← q
14             else r ← NEW-STATE
15                  terminal[r] ← FALSE
16                  for each letter b such that TARGET(q, b) ≠ NIL
17                       loop MAKE-EDGE(r, b, TARGET(q, b))
18                  length[r] ← length[p] + 1
```

```
19                          fail[r] ← fail[q]
20                          fail[q] ← r
21                          fail[t] ← r
22                          loop CANCEL-EDGE(p, a TARGET(p, a))
23                              MAKE-EDGE (p, a, r)
24                              p ← fail[p]
25                            while p ≠ NIL and TARGET(p, a) = q
26  return t
```

We illustrate the behavior of function SUFFIX-AUTOMATON-EXTENSION in Fig. 11.13.

With the suffix automaton $\mathcal{S}(y)$ of y, several operations can be solved efficiently. We describe three of them. Let x be a string of length m.

Membership test solves in time $O(m)$ by spelling x from the initial state of the automaton. If the entire string is spelled, x occurs in y. Otherwise we get the longest prefix of x occurring in y.

Computing the number k of occurrences of x in y (assuming that x is a factor of y) starts similarly. Let p be the state reached after the spelling of x from the initial state. Then k is exactly the number of terminal states accessible from p. The number k associated with each state p can be precomputing in time $O(n)$ (independently of the alphabet) by a depth-first traversal of the graph underlying the automaton. The query for x is then performed in time $O(m)$, whatever is k.

The base of an algorithm for computing a longest factor common to x and y is implemented in the procedure ENDING-FACTORS-MATCHER given below. This procedure reports at each position in y the length of the longest factor of x ending here. It can obviously be used for string matching. It works as the procedure AHO-CORASICK-MATCHER in the use of the failure function. The running time of the search phase of the procedure is $O(m)$.

```
ENDING-FACTORS-MATCHER(y, x)
1   i ← SUFFIX-AUTOMATON(y)
2   ℓ ← 0
3   p ← i
4   for symbol a from first to last symbol of x
5     loop if TARGET(p, a) ≠ NIL
6           then ℓ ← ℓ + 1
7                p ← TARGET(p, a)
8           else loop  p ← fail[p]
9             while p ≠ NIL and TARGET(p, a) ≠ NIL
10            if p = NIL
11                    then ℓ ← 0
12                         p ← i
13                    else ℓ ← length[p] + 1
14                         p ← TARGET (p, a)
15          report ℓ
```

Retaining a largest value of the variable ℓ in the procedure (instead of reporting all values) solves the longest common factor problem.

Suffix Arrays

There is a clever an rather simple way to deal with all suffixes of a text of length n: to arrange their list in increasing lexicographic order to be able to perform binary searches on them. The implementation of this idea leads to a data structure called suffix array. It is an efficient representation in the sense that

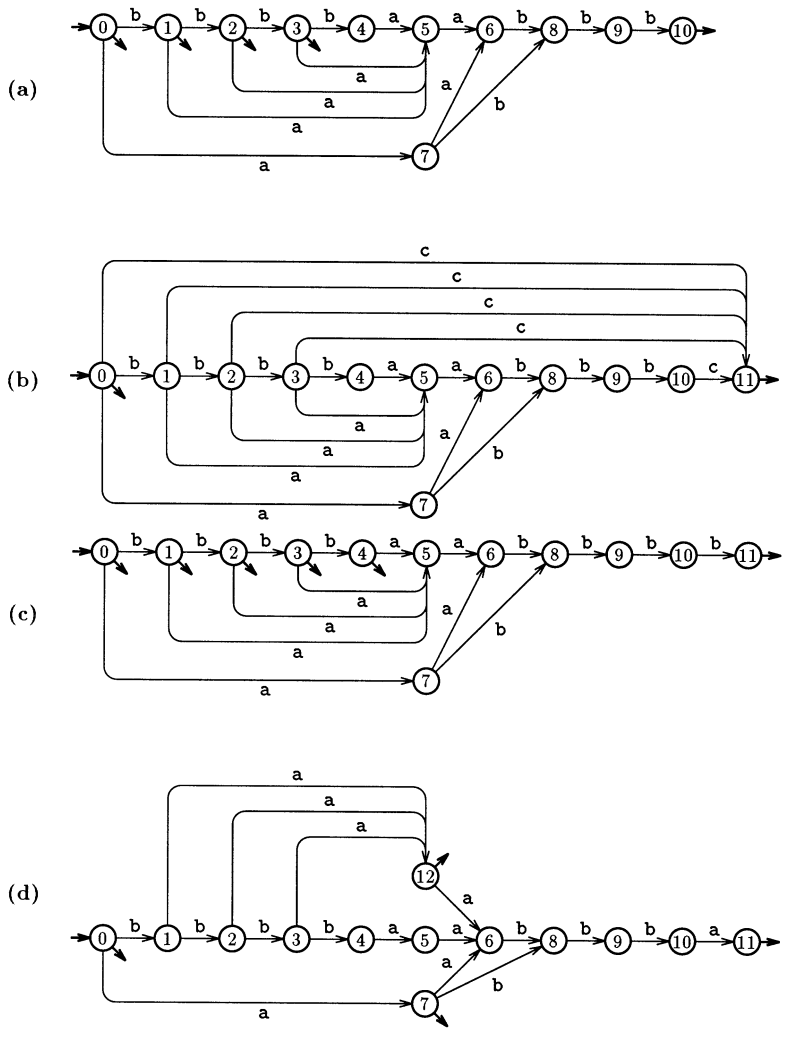


FIGURE 11.13 An illustration of the behavior of function SUFFIX-AUTOMATON-EXTENSION. The function transforms the suffix automaton $\mathcal{S}(y')$ of a string y' in the suffix automaton $\mathcal{S}(y'a)$ for any given symbol a (the terminal states being implicitly known). Let us consider that $y' =$ `bbbbaabbb`, and let us examine three possible cases according to a, namely $a =$ `c`, $a =$ `b`, and $a =$ `a`. **(a)** The automaton $\mathcal{S}$(`bbbbaabbb`). The state denoted by t' is state 10, and the suffix path of t' is the list of the states 10, 3, 2, 1, and 0. During the execution of the first loop of the function, state p runs through a part of the suffix path of t'. At the same time, edges labeled by a are created from p the newly created state $t = 11$, unless such an edge already exists in which case the loop stops. **(b)** If $a =$ `c`, the execution stops with an undefined value for p. The edges labeled by `c` start at terminal states, and the failure of t is the initial state. **(c)** If $a =$ `b`, the loop stops on state $p = 3$, because an edge labeled by `b` is defined on it. The condition at line 12 of function SUFFIX-AUTOMATON-EXTENSION is satisfied, which means that the edge labeled by a from p is not a short-circuit. In this case, the state ending the previous edge is the failure of t. **(d)** Finally, when $a =$ `a`, the loop stops on state $p = 3$ for the same reason, but the edge labeled by a from p is a short-circuit. The string `bbba` is a suffix of the (newly considered) string `bbbbaabbba`, but `bbbba` is not. Since these two strings reach state 5, this state is duplicated into a new state $r = 12$ that becomes terminal. Suffixes `bba` and `ba` are redirected to this new state. The failure of t is r.

- It has $O(n)$ size,
- It can be constructed in $O(n \log n)$ time,
- It allows the computation of the membership test of a string of length m in the text in time $O(m + \log n)$.

So, the time required to construct and use the structure is slightly greater than that needed to compute the suffix tree. But suffix arrays have two advantages:

- Their constructions is rather simple. It is even commonly admitted that, in practice, it behaves better than the construction of suffix trees.
- It consists of two linear size arrays which, in practice again, take little memory space.

11.4 Research Issues and Summary

String searching by hashing was introduced by Harrison [23], and later fully analyzed in [26]

The first linear-time string-matching algorithm is due to Knuth et al. [27]. It can be proved that, during the search, the delay, that is, the number of times a symbol of the text is compared to symbols of the pattern, is less than $\lfloor \log_\Phi(m+1) \rfloor$, where Φ is the golden ratio $(1+\sqrt{5})/2$. Simon [32] gives a similar algorithm but with a delay bounded by the size of the alphabet (of the pattern). Hancart [22] proves that the delay of Simon's algorithm is less than $1 + \lfloor \log_2 m \rfloor$. This paper also proves that this is optimal among algorithms processing the text with a one-symbol buffer. The bound becomes $O(\log \min\{1 + \lfloor \log_2 m \rfloor, \text{card}\Sigma\})$ using an ordering on the alphabet Σ, which is not a restriction in practice.

Galil [19] gives a general criterion to transform string-matching algorithms that work sequentially on the text into real-time algorithms.

The Boyer–Moore algorithm was designed in [7] The version given in this chapter follows [27]. This paper contains the first proof on the linearity of the algorithm when restricted to the search of the first occurrence of the pattern. Cole [9] proves that the maximum number of symbol comparisons is bounded by $3n$ for nonperiodic patterns, and that this bound is tight.

Knuth et al. [27] considers a variant of the Boyer–Moore algorithm in which all previous matches inside the current window are memorized. Each window configuration becomes the state of what is called the Boyer–Moore automaton. It is still unknown whether the maximum number of states of the automaton is polynomial or not.

Several variants of the Boyer–Moore algorithm avoid the quadratic behavior when searching for all occurrences of the pattern. Among the most efficient in terms of the number of symbol comparisons are the algorithm of Apostolico and Giancarlo (1986), Turbo-BM algorithm by Crochemore et al. [16] (the two previous algorithms are analyzed in [28], and the algorithm of Colussi [11].

The Horspool algorithm is from [24]. The paper contains practical aspects of string matching that are developed in [25].

The optimal bound on the expected time complexity of string matching is $O(\frac{\log m}{m} n)$ (see [27] and the paper of Yao [38]).

String matching can be solved by linear-time algorithms requiring only a constant amount of memory in addition to the pattern and the (window on the) text. This can be proved by different techniques presented in [15]. The most recent solution is by Gąsieniec, Plandowski, and Rytter [18].

Cole et al. [10] shows that, in the worst case, any string-matching algorithm working with symbol comparisons makes at least $n + \frac{9}{4m}(n-m)$ comparisons during its search phase. Some string-matching algorithms make less than $2n$ comparisons. The presently known upper bound on the problem is $n + \frac{8}{3(m+1)}(n-m)$, but with a quadratic-time preprocessing phase (see Cole et al. [10]). With a linear-time preprocessing phase, the current upper bounds are $\frac{4}{3}n - \frac{1}{3}m$ and $n + \frac{4\log m + 2}{m}(n-m)$ (see, respectively,

[20] and [8]). Except in a few cases (patterns of length 3 for example), lower and upper bounds do not meet. So, the problem of the exact complexity of string matching is open.

The Aho–Corasick algorithm is from [2]. Commentz-Walter [12] has designed an extension of the Boyer–Moore algorithm that solves the dictionary-matching problem. It is fully described in [1].

Ideas of "Small Patterns" are from [4] and [36]. An implementation of the method given in "Small Patterns" is the `agrep` command of UNIX [36].

The suffix-tree construction of "Suffix Trees" is from [29]. An on-line version is by Ukkonen [34]. A previous algorithm by Weiner [35] relates suffix trees to a data structure close to suffix automata.

The construction of suffix automata, also described as direct acyclic word graphs and often denoted by the acronym DAWG, is from Blumer et al. [5] and from [14]. An application to data retrieval by the mean of inverted files is described in Blumer et al. [6].

The alternative data structure for indexes given in "Suffix Arrays" is by [30].

11.5 Defining Terms

Border: A string v is a border of a string u if v is both a prefix and a suffix of u. String v is said to be the border of u if it is the longest proper border of u.

Factor: A string v is a factor of a string u if $u = u'vu''$ for some strings u' and u''.

Occurrence: A string v occurs in a string u if v is a factor of u.

Pattern: A finite number of strings that are searched for in texts.

Prefix: A string v is a prefix of a string u if $u = vu''$ for some string u''.

Proper: Qualifies a factor of a string that is not equal to the string itself.

Suffix: A string v is a suffix of a string u if $u = u'v$ for some string u'.

Suffix tree: Trie containing all the suffixes of a string.

Suffix automaton: Smallest automaton accepting the suffixes of a string.

Text: A stream of symbols that is searched for occurrences of patterns.

Trie: Digital tree, tree in which edges are labeled by symbols or strings.

Window: Factor of the text that is aligned with the pattern.

References

[1] Aho, A.V., Algorithms for finding patterns in strings. In *Handbook of Theoretical Computer Science,* J. van Leeuwen, Ed., vol. A, chap. 5, 255–300. Elsevier, Amsterdam, 1990.

[2] Aho, A.V. and Corasick, M.J., Efficient string matching: an aid to bibliographic search. *Comm. ACM,* 18, 333–340, 1975.

[3] Baase, S., *Computer Algorithms — Introduction to Design and Analysis.* Addison-Wesley, Reading, MA, 1988.

[4] Baeza-Yates, R. and Gonnet, G.H., A new approach to text searching. *Comm. ACM,* 35, 74–82, 1992.

[5] Blumer, A., Blumer, J., Ehrenfeucht, A., Haussler, D., Chen, M.T., and Seiferas, J., The smallest automaton recognizing the subwords of a text. *Theoret. Comput. Sci.,* 40, 31–55, 1985.

[6] Blumer, A., Blumer, J., Ehrenfeucht, A., Haussler, D., and McConnel, R., Complete inverted files for efficient text retrieval and analysis. *J. ACM,* 34, 578–595, 1987.

[7] Boyer, R.S. and Moore, J.S., A fast string searching algorithm. *Comm. ACM,* 20, 762–772, 1977.

[8] Breslauer, D. and Galil, Z., Efficient comparison based string matching. *J. Complexity,* 9, 339–365, 1993.

[9] Cole, R., Tight bounds on the complexity of the Boyer-Moore pattern matching algorithm. *SIAM J. Comput.,* 23, 1075–1091, 1994.
[10] Cole, R., Hariharan, R., Zwick, U., and Paterson, M.S., Tighter lower bounds on the exact complexity of string matching. *SIAM J. Comput.,* 24, 30–45, 1995.
[11] Colussi, L., Fastest pattern matching in strings. *J. Algorithms,* 16, 163–189, 1994.
[12] Commentz-Walter, B., A string-matching algorithm fast on the average, in *Automata, Languages and Programming,* Lecture Notes in Computer Science, Springer-Verlag, Berlin, 1979, 118–132.
[13] Cormen, T.H., Leiserson, C.E., and Rivest, R.L., *Introduction to Algorithms.* MIT Press, 1990.
[14] Crochemore, M., Transducers and repetitions. *Theoret. Comput. Sci.,* 45, 63–86, 1986.
[15] Crochemore, M. and Rytter, W., *Text Algorithms.* Oxford University Press, 1994.
[16] Crochemore, M., Czumaj, A., Gąsieniec, L., Jarominek, S., Lecroq, T., Plandowski, W., and Rytter, W., Speeding up two string-matching algorithms, in *9th Annual Symposium on Theoretical Aspects of Computer Science,* Finkel, A. and Jantzen, M., Eds., Springer Verlag, Berlin, 1992, 589–600.
[17] Frakes, W.B. and Baeza-Yates, R. *Information Retrieval: Data Structures and Algorithms.* Prentice-Hall, Englewood Cliffs, NJ, 1992.
[18] Gąsieniec, L., Plandowski, W., and Rytter, W., The zooming method: a recursive approach to time-space efficient string-matching. *TCS,* 147, (1–2), 19–30, 1995.
[19] Galil, Z., String matching in real time. *J. ACM,* 28, 134–149, 1981.
[20] Galil, Z. and Giancarlo, R., On the exact complexity of string matching: upper bounds. *SIAM J. Comput.,* 21, 407–437, 1992.
[21] Gonnet, G.H. and Baeza-Yates, R.A., *Handbook of Algorithms and Data Structures.* Addison-Wesley, Reading, MA, 1991.
[22] Hancart, C., On Simon's string searching algorithm. *Inf. Process. Lett.,* 47, 95–99, 1993.
[23] Harrison, M.C., Implementation of the substring test by hashing, *Comm. ACM,* 14(2), 777-779, 1971.
[24] Horspool, R.N., Practical fast searching in strings. *Software — Practice and Experience,* 10, 501–506, 1980.
[25] Hume, A. and Sunday, D.M., Fast string searching. *Software — Practice and Experience,* 21, 1221–1248, 1991.
[26] Karp, R.M. and Rabin, M.O., Efficient randomized pattern-matching algorithms. *IBM J. Res. Dev.,* 31, 249–260, 1987.
[27] Knuth, D.E., Morris, J.H., Jr., and Pratt, V.R., Fast pattern matching in strings. *SIAM J. Comput.,* 6, 323–350, 1977.
[28] Lecroq, T., Experimental results on string-matching algorithms. *Software — Practice and Experience,* 25, 727–765, 1995.
[29] McCreight, E.M., A space-economical suffix tree construction algorithm. *J. Algorithms,* 23, 262–272, 1976.
[30] Manber, U. and Myers, G., Suffix arrays: a new method for on-line string searches. *SIAM J. Comput.,* 22, 935–948, 1993.
[31] Sedgewick, R., *Algorithms.* Addison-Wesley, Reading, MA, 1988.
[32] Simon, I., String matching algorithms and automata. In *First American Workshop on String Processing,* R. Baeza-Yates and N. Ziviani, Eds., 151–157. Universidade Federal de Minas Gerais, 1993.
[33] Stephen, G.A., *String Searching Algorithms.* World Scientific Press, 1994.
[34] Ukkonen, E., Constructing suffix trees on-line in linear time, *IFIP '92,* 484–492, 1992.
[35] Weiner, P., Linear pattern-matching algorithms, in *Proc. 14th IEEE Annual Symposium on Switching and Automata Theory,* Institute of Electrial and Electronics Engineers, New York, 1973, 1–11.

[36] Wu, S. and Manber, U., Agrep — A fast approximate pattern-matching tool. *Usenix Winter 1992 Technical Conference,* 153–162, 1992.
[37] Wu, S. and Manber, U., Fast text searching allowing errors. *Comm. ACM,* 35, 83–91, 1992.
[38] Yao, A.C., The complexity of pattern matching for a random string, *SIAM J. Comput.,* 8, 368–387, 1979.

Further Information

Problems and algorithms presented in the chapter are just a sample of questions related to pattern matching. They share the formal methods used to design efficient algorithms. A wider panorama of algorithms on texts may be found in a few books such as [15] and [33].

Research papers in pattern matching are disseminated in a few journals, among which are *Communications of the ACM, Journal of the ACM, Theoretical Computer Science, Journal of Algorithms, SIAM Journal on Computing, Algorithmica.*

Two main annual conferences present the latest advances of this field of research:

- *Combinatorial Pattern Matching,* which started in 1990 in Paris (France), and was held since in London (England), Tucson (Arizona), Padova (Italy), Asilomar (California), Helsinki (Finland), Laguna Beach (California).
- *Workshop on String Processing,* which started in 1993 in Belo Horizonte (Brazil), and was held since in Valparaiso (Chile), and Recife (Brazil).

But general conferences in computer science often have sessions devoted to pattern matching.

Information retrieval questions are treated in [17], where the reader can find references on the problem.

Several books on the design and analysis of general algorithms contain a chapter devoted to algorithms on texts. Here is a sample of these books [3, 13, 21, 31]. During the production of this book, a new book has appeared titled *Algorithms on Strings, Trees and Sequences,* by D. Gusfield, Cambridge University Press (1997).

12

Text Data Compression Algorithms

Maxime Crochemore
Université de Marne-la-Vallée

Thierry Lecroq
Université de Rouen

12.1 Text Compression

The chapter describes a few algorithms that compress texts. Compression serves both to save storage space and to save transmission time. We shall assume that the text is stored in a file. The aim of compression algorithms is to produce a new file, as short as possible, containing the compressed version of the same text. Methods presented here reduce the representation of text without any loss of information, so that decoding the compressed text restores exactly the original data.

The term "text" should be understood in a wide sense. It is clear that texts can be written in natural languages or can be texts usually generated by translators (like various types of compilers). But texts can also be images or other kinds of structures as well provided the data are stored in linear files.

The interest in data compression techniques remains important even if mass storage systems improve regularly because the amount of data grows accordingly. Moreover, a consequence of the extension of computer networks is that the quantity of data they exchange grows exponentially, so it is often necessary to reduce the size of files to reduce proportionally their transmission times. Other advantages in compressing files regard two connected issues: integrity of data and security. While the first is easily accomplished through redundancy checks during the decompression phase, the second often requires data compression before applying cryptography.

This chapter contains three classical text compression algorithms. Variants of these algorithms are implemented in practical compression software, in which they are often combined together or with other elementary methods. Moreover, we present all-purpose methods, that is, methods in which no sophisticated modeling of the statistics of texts is done. An adequate modeling of a well-defined family of texts may increase significantly the compression when coupled with the coding algorithms of this chapter.

Compression ratios of the methods depend on the input data. However, most often, the size of compressed text vary from 30% to 50% of the size of the input. At the end of this chapter, we present ratios obtained by the methods on several example texts. Results of this type can be used to compare the efficiency of methods in compressing data. But, the efficiency of algorithms is also evaluated by their running times, and sometimes by the amount of memory space they require at run time. These elements are important criteria of choice when a compression algorithm is to be implemented in a telecommunication software.

Two strategies are applied to design the algorithms. The first strategy is a statistical method that takes into account the frequencies of symbols to build a uniquely decipherable code optimal with respect to the compression (Sections 12.2 and 12.3). Section 12.4 presents a refinement of the coding algorithm of Huffman based on the binary representation of numbers. Huffman codes contain new codewords for the symbols occurring in the text. In this method fixed-length blocks of bits are encoded by different codewords. *A contrario* the second strategy encodes variable-length segments of the text (Section 12.5). To put it simply, the algorithm, while scanning the text, replaces some already read segments by just a pointer to their first occurrences. This second strategy often provides better compression ratios.

12.2 Static Huffman Coding

The Huffman method is an optimal statistical coding. It transforms the original code used for characters of the text (ASCII code on 8 bits, for instance). Coding the text is just replacing each symbol (more exactly each occurrence of it) by its new **codeword**. The method works for any length of blocks (not only 8 bits), but the running time grows exponentially with the length. In the following, we assume that symbols are originally encoded on 8 bits to simplify the description.

The Huffman algorithm uses the notion of **prefix code**. A prefix code is a set of words containing no word that is a **prefix** of another word of the set. The advantage of such a code is that decoding is immediate. Moreover, it can be proved that this type of code does not weaken the compression.

A prefix code on the binary alphabet $\{0, 1\}$ corresponds to a binary tree in which the links from a node to its left and right children are labeled by 0 and 1, respectively. Such a tree is called a (digital) **trie**. Leaves of the trie are labeled by the original characters, and labels of branches are the words of the code (codewords of characters). Working with prefix code implies that codewords are identified with leaves only. Moreover, in the present method codes are complete: they correspond to complete tries, i.e., tree in which internal nodes have all exactly two children.

In the model where characters of the text are given new codewords, the Huffman algorithm builds a code that is optimal in the sense that the compression is the best possible (if the model of the source text is a zero-order Markov process, that is if the probability of symbol occurrence are independent). The length of the encoded text is minimum. The code depends on the input text, and more precisely on the frequencies of characters in the text. The most frequent characters are given shortest codewords, while the least frequent symbols correspond to the longest codewords.

Encoding

The complete compression algorithm is composed of three steps: count of character frequencies, construction of the prefix code, encoding of the text. The last two steps use information computed by their preceding step.

The first step consists in counting the number of occurrences of each character in the original text (see Fig. 12.1). We use a special end marker, denoted by `END`, which virtually appears only once at the end of the text. It is possible to skip this first step if fixed statistics on the alphabet are used. In this case, however, the method is optimal according to the statistics, but not necessarily for the specific text.

The second step of the algorithm builds the tree of a prefix code, called a Huffman tree, using the character frequency $freq(a)$ of each character a in the following way:

```
H-COUNT (fin)
1  for each character a ∈ Σ
2      do freq(a) ← 0
3  while not end of file fin and a is the next symbol
4      do freq(a) ← freq(a) + 1
5  freq(END) ← 1
```

FIGURE 12.1 Counts the character frequencies.

- Create a one-node tree t for each character a, setting $weight(t) = freq(a)$ and $label(t) = a$,
- Repeat
  - Extract the two least weighted trees t_1 and t_2,
  - Create a new tree t_3 having left subtree t_1, right subtree t_2, and weight $weight(t_3) = weight(t_1) + weight(t_2)$,
- Until only one tree remains.

The tree is constructed by the algorithm H-BUILD-TREE in Fig. 12.2. The implementation uses two linear lists. The first list, *lleaves*, contains the leaves of the future tree associated each with a symbol. The list is sorted in increasing order of weights of leaves (frequencies of symbols). The second list, *ltrees*, contains the newly created trees. The operation of extracting the two least weighted trees is done by checking the two first trees of the list *lleaves* and the two first trees of the list *ltrees*. Each new tree is inserted at the end of the list of the trees. The only tree remaining at the end of the procedure is the coding tree.

```
H-BUILD-TREE
1   for each a ∈ Σ ∪ {END}
2       do if freq(a) ≠ 0
3             then create a new node t
4                  weight(t) ← freq(a)
5                  label(t) ← a
6   lleaves ← list of all created nodes in increasing order of weight
7   ltrees ← empty list
8   while LENGTH (lleaves) + LENGTH (ltrees) > 1
9       do (l, r) ← extract two nodes of smallest weight (among the two nodes at the
                beginning of lleaves and the two nodes at the beginning of ltrees)
10        create a new node t
11        weight(t) ← weight(l) + weight(r)
12        left(t) ← l
13        right(t) ← r
14        insert t at the end of ltrees
16  return t
```

FIGURE 12.2 Builds the Huffman coding tree.

After the coding tree is built, it is possible to recover the codewords associated with characters by a simple depth-first-search of the tree (see Fig. 12.3); $codeword(a)$ denotes the binary codeword associated with the character a.

In the third step, the original text is encoded. Since the code depends on the original text, in order to be able to decode the compressed text, the coding tree and the original codewords of symbols must be stored together with the compressed text.

This information is placed in a header of the compressed file, to be read at decoding time just before the decompression starts. The header is written during a depth-first traversal of the tree. Each time an internal node is encountered a 0 is produced. When a leaf is encountered a 1 is produced followed by the

```
H-BUILD-CODE (t, length)
1  if t is not a leaf
2     then temp[length] ← 0
3          H-BUILD-CODE (left(t), length + 1)
4          temp[length] ← 1
5          H-BUILD-CODE (right(t), length + 1)
6     else codeword(label(t)) ← temp[0 . . . length − 1]
```

FIGURE 12.3 Builds character codewords from the coding tree.

original code of the corresponding character on 9 bits (so that the end marker can be equal to 256 if all the 8-bit characters appear in the original text). This part of the encoding algorithm is shown in Fig. 12.4.

```
H-ENCODE-TREE (fout, t)
1  if t is not a leaf
2     then write a 0 in the file fout
3          H-ENCODE-TREE (fout, left(t))
4          H-ENCODE-TREE (fout, right(t))
5     else write a 1 in the file fout
6          write the original code of label(t) in the file fout
```

FIGURE 12.4 Stores the coding tree in the compressed file.

After the header of the compressed file is made, the encoding of the original text is realized by the algorithm of Fig. 12.5.

A complete implementation of the Huffman algorithm, composed of the three steps described above, is given in Fig. 12.6.

```
H-ENCODE-TEXT (fin, fout)
1  while not end of file fin and a is the next symbol
2     do write codeword(a) in the file fout
3  write codeword(END) in the file fout
```

FIGURE 12.5 Encodes the characters in the compressed file.

```
H-ENCODING (fin, fout)
1  H-COUNT (fin)
2  t ← H-BUILD-TREE
3  H-BUILD-CODE (t, 0)
4  H-ENCODE-TREE (fout, t)
5  H-ENCODE-TEXT (fin, fout)
```

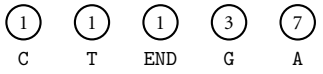
FIGURE 12.6 Complete function for Huffman encoding.

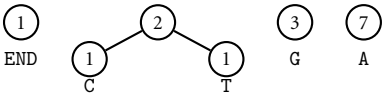
EXAMPLE 12.1:

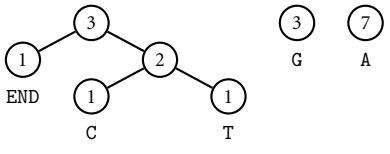
$y =$ ACAGAATAGAGA
Length of $y = 12 \times 8 = 104$ bits (assuming an 8-bit code)
Character frequencies:

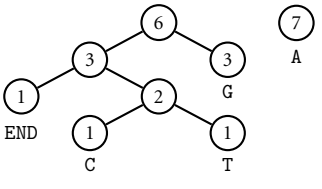
A	C	G	T	END
7	1	3	1	1

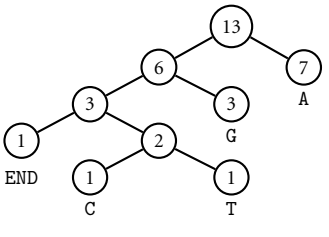
Different steps during the construction of the coding tree:











character codewords:

A	C	G	T	END
1	0010	01	0011	000

Encoded tree: 0001 *binary*(END,9)01 *binary*(C,9)1 *binary*(T,9)1 *binary*(G,9)1*binary*(A,9),
which produces a header of length 54 bits:
0001 100000000 01 001000011 1 001010100 1 001000111 1 001000001

Encoded text:

$$\underbrace{1}_{A}\ \underbrace{0010}_{C}\ \underbrace{1}_{A}\ \underbrace{01}_{G}\ \underbrace{1}_{A}\ \underbrace{1}_{A}\ \underbrace{0011}_{T}\ \underbrace{1}_{A}\ \underbrace{01}_{G}\ \underbrace{1}_{A}\ \underbrace{01}_{G}\ \underbrace{1}_{A}\ \underbrace{000}_{END}$$

of length 24 bits
Total length of the compressed file: 78 bits.

The construction of the tree takes $O(\sigma \log \sigma)$ time if the sorting of the list of the leaves is implemented efficiently. The rest of the encoding process runs in time linear in the sum of the sizes of the original and compressed texts.

Decoding

Decoding a file containing a text compressed by Huffman algorithm is a mere programming exercise. First, the coding tree is rebuilt by the algorithm of Fig. 12.7. Then, the original text is recovered by parsing the compressed text with the coding tree. The process begins at the root of the coding tree, and follows a left edge when a 0 is read or a right edge when a 1 is read. When a leaf is encountered, the corresponding character (in fact the original codeword of it) is produced and the parsing resumes at the root of the tree. The process ends when the codeword of the end marker is encountered. An implementation of the decoding of the text is presented in Fig. 12.8.

```
H-REBUILD-TREE (fin, t)
1   read bit b from fin
2   if b = 1
3      then make t a leaf
4           label(t) ← symbol corresponding to the 9 next bits read from fin
5      else create a new node l
6           left(t) ← l
7           H-REBUILD-TREE (fin, l)
8           create a new node r
9           right(t) ← r
10          H-REBUILD-TREE (fin, r)
```

FIGURE 12.7 Rebuilds the tree from the header of compressed file.

```
H-DECODE-TEXT (fin, fout, root)
1   t ← root
2   while label(t) ≠ END
3      do if t is a leaf
4         then write label(t) in the file fout
5              t ← root
6         else read bit b from fin
7              if b = 1
8                 then t ← right(t)
9                 else t ← left(t)
```

FIGURE 12.8 Recovers the original text.

The complete decoding program is given in Fig. 12.9. It calls the preceding functions. The running time of the decoding program is linear in the sum of the sizes of the texts it manipulates.

```
H-DECODING (fin, fout)
1   create a new node root
2   H-REBUILD-TREE (fin, root)
3   H-DECODE-TEXT (fin, fout, root)
```

FIGURE 12.9 Complete function for Huffman decoding.

12.3 Dynamic Huffman Coding

The two main drawbacks of the static Huffman method are: first, if the frequencies of characters the source text are not known *a priori*, the source text has to be read twice; second, the coding tree must be included in the compressed file. This is avoided by a dynamic method where the coding tree is updated each time a symbol is read from the source text. The current tree is a Huffman tree related to the part of the text that is already treated. The tree evolves exactly in the same way during the decoding process. The efficiency of the method is based on a characterization of Huffman trees, known as the *siblings property*.

Siblings property: Let T be a Huffman tree with n leaves (a complete binary weighted tree built by the procedure H-BUILD-TREE in which all leaves have positive weights). Then the nodes of T can be arranged in a sequence $(x_0, x_1, \ldots, x_{2n-2})$ such that:

1. The sequence of weights $(weight(x_0), weight(x_1), \ldots, weight(x_{2n-2}))$ is in decreasing order;
2. For any i $(0 \leq i \leq n-2)$, the consecutive nodes x_{2i+1} and x_{2i+2} are siblings (they have the same parent).

The compression and decompression processes initialize the dynamic Huffman tree by a one-node tree that correspond to an artificial character, denoted by `ART`. The weight of this single node is 1.

Encoding

Each time a symbol a is read from the source text, its codeword in the tree is sent. However this happens only if a appeared previously. Otherwise the code of `ART` is sent followed by the original codeword of a. Afterwards, the tree is modified in the following way: first, if a never occurred before, a new internal node is created and its two children are a new leaf labeled by a and the leaf `ART`; then, the tree is updated (see below) to get a Huffman tree for the new prefix of text.

Implementation

Each node is identified with an integer n, the root is the integer 0. The invariant of compression and decompression algorithms is that, if the tree has m nodes, the sequence of nodes $(m-1, \ldots, 1, 0)$ satisfies the siblings property. The tree is stored in a table, and we use the next notations, for a node n:

- $parent(n)$ is the parent of n ($parent(root) =$ `UNDEFINED`),
- $child(n)$ is the left child of n (if n is an internal node, otherwise $child(n) =$ `UNDEFINED`), and $child(n)+1$ is its right child (this is useful only at decoding time),
- $label(n)$ is the symbol associated with n when n is a leaf,
- $weight(n)$ is the weight of n (it is the frequency of $label(n)$ if n is a leaf).

Indeed, the child link is useful only at decoding time so that the implementation may differ between the two phases of the algorithm. But, to simplify the description and give a uniform treatment of the data structure, we assume the same implementation during the encoding and the decoding steps. Weights of nodes are handled by the procedure DH-UPDATE.

The correspondence between symbols and leaves of the tree is done by a table called *leaf*: for each symbol $a \in \Sigma \cup \{$`END`$\}$ $leaf[a]$ is the corresponding leaf of the tree, if any.

The coding tree initially contains only one node labeled by symbol `ART`. The initialization is given in Fig. 12.10.

Encoding the source text is a succession of three steps: read a symbol a from the source text, encode the symbol a according to the current tree, update the tree. It is described in Fig. 12.11.

Encoding a symbol a already encountered consists in accessing its associated leaf $leaf[a]$, then computing its codeword by a bottom-up walk in the tree (following the *parent* links up to the root). Each time a node n ($\neq$ root) is encountered if n is odd a 1 is sent (n is the right child of its parent), and if n is even a 0 is sent (n is then the left child of its parent). As the codeword of a is read in reverse direction a stack S

```
DH-INIT
1   root ← 0
2   child(root) ← UNDEFINED
3   weight(root) ← 1
4   parent(root) ← UNDEFINED
5   for each letter a ∈ Σ ∪ {END}
6       do    leaf[a] ← UNDEFINED
7   leaf[ART] ← root
```

FIGURE 12.10 Initializes the dynamic Huffman tree.

```
DH-ENCODING (fin, fout)
1   DH-INIT
2   while not end of file fin and a is the next symbol
3       do    DH-ENCODE-SYMBOL (a, fout)
4             DH-UPDATE (a)
5   DH-ENCODE-SYMBOL (END, fout)
```

FIGURE 12.11 Complete function for dynamic Huffman encoding.

is used to temporarily store the bits and send them properly. The procedure SEND (S, *fout*) send the bits of the stack S in the correct order to the file *fout*.
If a has not been encountered yet, then the code of ART is sent followed by the original codeword of a on 9 bits, and a new leaf is created for a (see Figs. 12.12 and 12.13).

```
DH-ENCODE-SYMBOL (a, fout)
1   S ← empty stack
2   n ← leaf[a]
3   if n = UNDEFINED
4       then n ← leaf[ART]
5   while n ≠ root
6       do    if n is odd
7                 then PUSH (S, 1)
8                 else PUSH (S, 0)
9             n ← parent(n)
10  SEND (S, fout)
11  if leaf[a] = UNDEFINED
12      then write in fout the original codeword of a on 9 bits
13           DH-ADD-NODE (a)
```

FIGURE 12.12 Encodes one symbol.

Decoding

At decoding time the compressed text is parsed with the coding tree. The current node is initialized with the root like in the encoding algorithm, and then the tree evolves symmetrically. Each time a 0 is read from the compressed file the walk down the tree follows the left link, and it follows the right link if a 1 is read. When the current node is a leaf, its associated symbol is written in the output file and the tree is updated exactly as it is done during the encoding phase.

Implementation

As for the encoding process the current Huffman tree is stored in a table, and it is initialized with the artificial symbol ART. The same elements of the data structure are used during the decompression. Note that the next node when parsing a bit b from node n is just $child(n) + b$ with the convention on left-

```
DH-ADD-NODE (a)
1   transform leaf[ART] into
2       an internal node of weight 1 with
3           a left child of weight 0 for leaf[a],
4           a right child of weight 1 for leaf[ART].
```

FIGURE 12.13 Adds a new symbol in the tree.

right links and 0-1 bits. The tree is updated by the procedure DH-UPDATE used previously and that is described in the next section. Figures 12.14 and 12.15 display the decoding mechanism.

```
DH-DECODING (fin, fout)
1   DH-INIT
2   a ← DH-DECODE-SYMBOL (fin)
3   while a ≠ END
4       do   write a in fout
5            DH-UPDATE (a)
6            a ← DH-DECODE-SYMBOL (fin)
```

FIGURE 12.14 Complete function for dynamic Huffman decoding.

```
DH-DECODE-SYMBOL (fin)
1   n ← root
2   while child(n) ≠ UNDEFINED
3       do read bit b from fin
4              n ← child(n) + b
5   a ← label(n)
6   if a = ART
7       then a ← symbol corresponding to the next 9 bits read from fin
8            DH-ADD-NODE (a)
9   return (a)
```

FIGURE 12.15 Decodes one symbol from the input file.

Updating

During encoding and decoding phases the current tree has to be updated to take into account the correct frequency of symbols. When a new symbol is considered the weight of its associated leaf is incremented by 1, and the weights of ancestors have to be modified correspondingly. The procedure that realizes the operation is shown in Fig. 12.16. Its proof of correctness is based on the siblings property.

We explain how the procedure DH-UPDATE works.

First, the weight of the leaf n corresponding to a is incremented by 1. Then, if point 1 of the siblings property is no longer satisfied, node n is exchanged with the closest node m ($m < n$) in the list such that $weight(m) < weight(n)$. Doing so, the nodes remain in decreasing order of their weights. It is important here that leaves have positive weights, because this guarantees that m is not a parent nor an ancestor of node n. Afterwards, the same operation is repeated on the parent of n until the root of the tree is reached.

The aim of procedure DH-SWAP-NODES used in Fig. 12.16 is to exchange the subtrees rooted at its input nodes m and n. In concrete terms, this remains to exchange the records stored at the two nodes in the table. It is meant that nothing is to be done if $m = n$.

```
DH-UPDATE (a)
1  n ← leaf[a]
2  while n ≠ root
3     do   weight(n) ← weight(n) + 1
4          m ← n
5          while weight(m − 1) < weight(n)
6             do   m ← m − 1
7          DH-SWAP-NODES (m, n)
8          n ← parent(m)
9  weight(root) ← weight(root) + 1
```

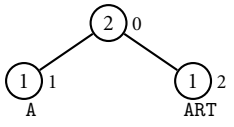
FIGURE 12.16 Updates the current Huffman tree.

EXAMPLE 12.2:

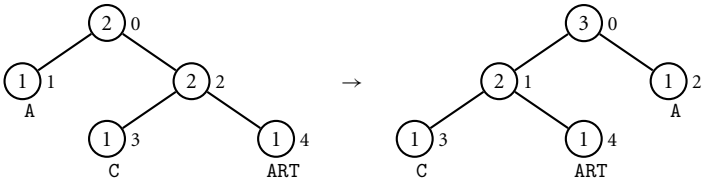
$y =$ ACAGAATAGAGA

Initial tree:

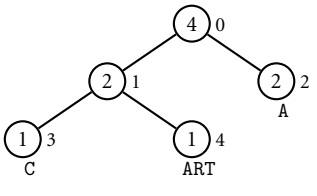
Next symbol is A:
The ASCII code of A is sent on 9 bits;
Bits sent: 001000001



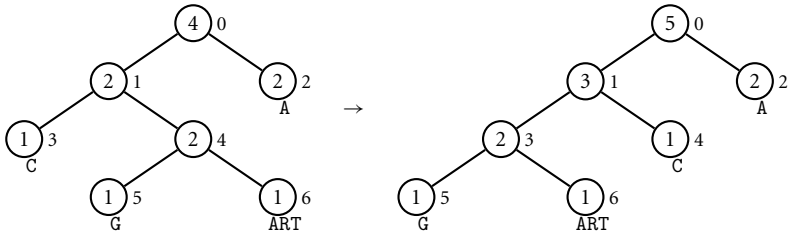
Next symbol is C:
The code of ART is sent followed by the ASCII code of C on 9 bits;
Bits sent: 1 001000011
Nodes 1 and 2 are swapped.



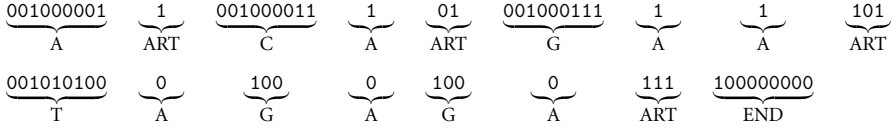
Next symbol is A:
The code of A is sent;
Bit sent: 1



Next symbol is G:
The code of ART is sent followed by the ASCII code of G on 9 bits;
Bits sent: 01 001000111
Nodes 3 and 4 are swapped.



Finally, the entire sequence of bits sent is the following:



The total length of the compressed text is 66.

12.4 Arithmetic Coding

Encoding

The basic idea of arithmetic coding is to consider symbol as digits of a numeration system, and texts as decimal parts of numbers between 0 and 1. The length of the interval attributed to a digit (it is 0.1 for digits in the usual base 10 system) is made proportional to the frequency of the digit in the text. The encoding is thus assimilated to a change in the base of a numeration system. To cope with precision problems, the number corresponding to a text is handled via a lower bound and an upper bound, which remains to associate with a text a subinterval of $[0, 1[$. The compression comes from the fact that large intervals require less precision to separate their bounds.

More formally, the interval associated with each symbol $a_i \in \Sigma\ (1 \leq i \leq \sigma)$ is denoted $I(a_i) = [l_i, h_i[$. The intervals satisfy the conditions: $l_1 = 0$, $h_\sigma = 1$, and $l_i = h_{i-1}$ for $1 < i \leq \sigma$. Note that $I(a_i) \cap I(a_j) = \emptyset$ if $a_i \neq a_j$.

The encoding phase consists in computing the interval corresponding to the input text. The basic step that deals with a symbol a_i of the source text transforms the current interval $[l, h[$ into $[l', h'[$ where $l' = l + (h - l) * l_i$ and $h' = l + (h - l) * h_i$, starting with the initial interval $[0, 1[$ (see Fig. 12.17). Indeed, in a theoretical approach, l only is needed to encode the input text.

```
AR-ENCODE (fin)
1  l ← 0
2  h ← 1
3  while not end of file fin and a_i is the next symbol
4     do   l ← l + (h − l) * l_i
5          h ← l + (h − l) * h_i
6  return(l)
```

FIGURE 12.17 Basic arithmetic encoding.

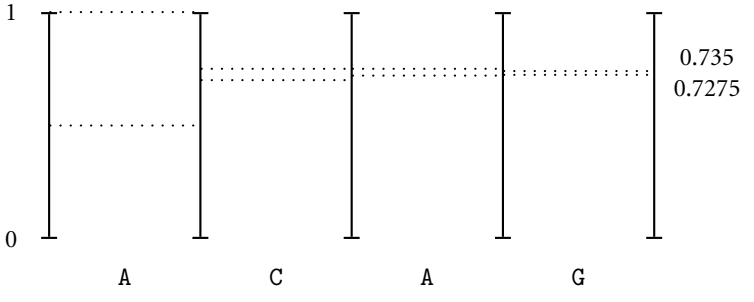
EXAMPLE 12.3:

$\Sigma = \{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}\}, \sigma = 4$

$$I(\mathtt{A}) = [0.5, 1[, \quad I(\mathtt{C}) = [0.4, 0.5[, \quad I(\mathtt{G}) = [0.1, 0.4[, \quad I(\mathtt{T}) = [0, 0.1[$$

Encoding ACAG gives:

symbol	l	h
	0	1
A	0.5	1
C	0.7	0.75
A	0.725	0.75
G	0.7275	0.735



Decoding

The reverse operation of decoding a text compressed by the previous algorithm theoretically requires only the lower bound l of the interval. Decoding the number l is done as follows: first find the symbol a_i such that $l \in I(a_i)$, produced the symbol a_i, and then replace l by l' defined by:

$$l' \leftarrow \frac{l - l_i}{h_i - l_i}.$$

The same process is repeated until $l = 0$ (see Fig. 12.18). Indeed, the implementation of the decoding phase simulates what is done on the current interval considered at encoding time.

```
AR-DECODE (l, fout)
1   while l ≠ 0
2       do   find a_i such that l ∈ I(a_i)
3            write a_i in file fout
4            l ← (l − l_i)/(h_i − l_i)
```

FIGURE 12.18 Basic arithmetic decoding.

EXAMPLE 12.4:

$\Sigma = \{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}\}, \sigma = 4$

$$I(\mathtt{A}) = [0.5, 1[, \quad I(\mathtt{C}) = [0.4, 0.5[, \quad I(\mathtt{G}) = [0.1, 0.4[, \quad I(\mathtt{T}) = [0, 0.1[$$

Decoding $l = 0.7275$:

l	a_i
0.7275	A
0.455	C
0.55	A
0.1	G
0	

Implementation

The main problem when implementing the arithmetic coding compression algorithm is to cope with precision on real numbers operations. The $[0, 1[$ interval of real numbers is substituted by the interval of integers $[0, 2^N - 1[$, where N is a fixed integer.

```
AR-ENCODING (fin, fout)
1   Initialize tables freq and cum-freq
2   l ← 0
3   h ← 2^N − 1
4   waiting-counter ← 0
5   while not end of file fin and a_i is the next symbol
6      do   AR-ENCODE-SYMBOL (a_i)
7           Update tables freq and cum-freq
8           Maintain symbols in decreasing order of frequencies
9   AR-ENCODE-SYMBOL (END)
10  waiting-counter ← waiting-counter + 1
11  AR-SEND-BIT (leftmost bit of h)
```

FIGURE 12.19 Complete arithmetic encoding function.

```
AR-ENCODE-SYMBOL (a_i)
1   l ← l + ((h − l + 1) * cum-freq[i])/cum-freq[0]
2   h ← l + ((h − l + 1) * cum-freq[i − 1])/cum-freq[0] − 1
3   repeat
4      if leftmost bit of l = leftmost bit of h
5         then AR-SEND-BIT (leftmost bit of h)
6              l ← 2 * l
7              h ← 2 * h + 1
8         else if h − l < cum-freq[0]
9                 then l ← 2 * (l − 2^(N−2))
10                     h ← 2 * (h − 2^(N−2)) + 1
11                     waiting-counter ← waiting-counter + 1
12  until leftmost bit of l ≠ leftmost bit of h and h − l ≥ cum-freq[0]
```

FIGURE 12.20 Encodes one symbol.

So, the algorithms work with integral values of size N. During the process, each time the binary representation of bounds l and h have a common prefix this prefix is sent and l is shifted to the left and filled with 0's while h is shifted to the left and filled with 1's.

The intervals associated with symbols of the alphabet are computed with the help of symbol frequencies, in a dynamic way: each character frequency is initialized with 1 and is incremented each time the symbol is encountered.

```
AR-SEND-BIT (bit, fout)
1   write bit in fout
2   while waiting-counter > 0
3      do   if bit = 0
4              then write 1 in fout
5                   write 0 in fout
6           waiting-counter ← waiting-counter − 1
```

FIGURE 12.21 Sends one bit followed by the waiting bits, if any.

We denote by $freq[i]$ the frequency of symbol a_i of the alphabet. We also consider the cumulative frequency of symbols, and set $cum\text{-}freq[i] = \Sigma_{j=i+1}^{\sigma} freq[j]$. Then $cum\text{-}freq[0]$ is the cumulative frequency of all symbols. Note that $cum\text{-}freq[0] - \sigma - 1$ is the length of the prefix of the input scanned so far. The symbols are maintained in decreasing order of their frequencies. This obviously save on the expected number of operations to update the table *cum-freq*.

Then when a symbol a_i is read from the source text, the current interval $[l, h[$ is updated in the following way:

$$l \leftarrow l + \frac{(h-l+1)*cum\text{-}freq[i]}{cum\text{-}freq[0]}$$

$$h \leftarrow l + \frac{(h-l+1)*cum\text{-}freq[i-1]}{cum\text{-}freq[0]} - 1$$

The common prefix (if any) of l and h is sent and l and h are shifted and filled (respectively, with 0's and 1's). At this point, if the interval is too short (if $h - l < cum\text{-}freq[0]$) it is extended to $[2 * (l - 2^{N-2}), 2 * (h - 2^{N-2}) + 1[$ and a waiting counter is incremented by 1. And the same operation is repeated as long as the interval is too short. After that, when a bit is sent, the reverse bit is sent the number of times indicated by the waiting counter.

EXAMPLE 12.5:

$\Sigma = \{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}\}$, $N = 8$

		A	C	G	T	END
i	0	1	2	3	4	5
cum-freq	5	4	3	2	1	0
freq	0	1	1	1	1	1

$[0, 255[\xrightarrow{\mathtt{A}} [0 + \frac{(255-0+1)*4}{5}, 0 + \frac{(255-0+1)*5}{5} - 1[= [204, 255[= [11001100_2, 11111111_2[$
11 is sent and $[00110000_2, 11111111_2[= [48, 255[$ is the next interval;

		A	C	G	T	END
i	0	1	2	3	4	5
cum-freq	6	4	3	2	1	0
freq	0	2	1	1	1	1

$[48, 255[\xrightarrow{\mathtt{C}} [96, 231[= [10011000_2, 10111001_2[$
10 is sent and $[01100000_2, 11100111_2[= [96, 231[$ is the next interval;

		A	C	G	T	END
i	0	1	2	3	4	5
cum-freq	7	5	3	2	1	0
freq	0	2	2	1	1	1

$[96, 231[\xrightarrow{\texttt{A}} [193, 231[= [11000001_2, 11100111_2[$
11 is sent and $[00000100_2, 10011111_2[= [4, 159[$ is the next interval;

Next symbol is G: 001 is sent;
Next symbol is A: 1 is sent;
Next symbol is A: 1 is sent;
Next symbol is T: 0111 is sent;
Next symbol is A: 10 is sent;
Next symbol is G: nothing is sent and the current interval becomes [111, 139[;

$[111, 139[\xrightarrow{\texttt{A}} [127, 139[= [01111111_2, 10001011_2[$, nothing is sent
The interval is too short and replaced by $[2 * (127 - 2^{N-2}), 2 * (139 - 2^{N-2}) + 1[= [126, 151[$ and one bit is waiting, $[01111110, 10010111[= [126, 151[$ is the next interval;

$[126, 151[\xrightarrow{\texttt{G}} [134, 138[= [10000110_2, 10001010_2[$
1+0+000 are sent and $[01100000_2, 10101111_2[= [96, 175[$ is the next interval;

$[96, 175[\xrightarrow{\texttt{A}} [141, 175[= [10001101_2, 10101111_2[$
10 is sent and $[00110100_2, 10111111_2[= [52, 191[$ is the next interval;

$[52, 191[\xrightarrow{\texttt{END}} [52, 59[= [00110100_2, 00111011_2[$
0011 is sent;

1+0 are sent in order to finish the encoding process.

```
AR-DECODING (fin, fout)
1   Init the tables freq and cum-freq
2   value ← 0
3   for i ← 1 to N
4       do    read bit b from fin
5             value ← 2 * value + b
6   l ← 0
7   h ← 2^N − 1
8   repeat
9       i ← AR-DECODE-SYMBOL (fin)
10      if a_i ≠ END
11          then write a_i in fout
12               Update the tables freq and cum-freq
13               Maintain symbols in decreasing order of frequencies
14  until a_i = END
```

FIGURE 12.22 Complete arithmetic decoding function.

The decoding process is exactly the reverse of the coding process. It uses a window of size N on the compressed file. First, the window is filled with the first N bits of the compressed file and *value* is the corresponding base 2 number. The current interval is initialized with $l = 0$ and $h = 2^N - 1$.

```
AR-DECODE-SYMBOL (fin)
1   cum ← ((value − l + 1) ∗ cum-freq[0] − 1)/(h − l + 1)
2   i ← 1
3   while cum-freq[i] > cum
4      do   i ← i + 1
5   l ← l + ((h − l + 1) ∗ cum-freq[i])/cum-freq[0]
6   h ← l + ((h − l + 1) ∗ cum-freq[i − 1])/cum-freq[0] − 1
7   repeat
8      if leftmost bit of l = leftmost bit of h
9         then l ← 2 ∗ l
10              h ← 2 ∗ h + 1
11              read bit b from fin
12              value ← 2 ∗ value + b
13        else  if h − l < cum-freq[0]
14                 then l ← 2 ∗ (l − 2^(N−2))
15                      h ← 2 ∗ (h − 2^(N−2)) + 1
16                      read bit b from fin
17                      value ← 2 ∗ (value − 2^(N−2)) + b
18  until leftmost bit of l ≠ leftmost bit of h and h − l ≥ cum-freq[0]
19  return i
```

FIGURE 12.23 Decodes one symbol.

Then, the symbol a_i to be produced is the first character such that

$$cum\text{-}freq[i] > \frac{(value - l + 1) * cum\text{-}freq[0] - 1)}{h - l + 1},$$

and l and h are then updated exactly in the same way than during the coding process. If the binary representations of l and h have a common prefix of length p they are both shifted p binary position to the left and l is filled by 0's, h is filled with 1's. The window on the compressed file is shifted p symbols to the right and the variable *value* is updated correspondingly. The tables *freq* and *cum-freq* are updated and the symbols are maintained in decreasing order of the frequencies as in the coding process.

This is repeated until the symbol END is produced.

EXAMPLE 12.6:

Decoding the text 11101100111011111010000010001110
$\Sigma = \{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}\}$

		A	C	G	T	END
i	0	1	2	3	4	5
cum-freq	5	4	3	2	1	0
freq	0	1	1	1	1	1

$value = 236 = 11101100_2$
$cum = 4$
$[0, 255[\xrightarrow{\mathtt{A}} [204, 255[= [11001100_2, 11111111_2[$: shift by 2
The next interval is $[00110000_2, 11111111_2[$ and $value = 10110011_2 = 179$

		A	C	G	T	END
i	0	1	2	3	4	5
cum-freq	6	4	3	2	1	0
freq	0	2	1	1	1	1

$value = 179 = 10110011_2$
$cum = 3$
$[48, 255[\xrightarrow{C} [152, 185[= [10011000_2, 10111001_2[$: shift by 2
The next interval is $[01100000_2, 11100111_2[$ and $value = 11001110_2 = 206$

		A	C	G	T	END
i	0	1	2	3	4	5
cum-freq	7	5	3	2	1	0
freq	0	2	2	1	1	1

$value = 206 = 11001110_2$
$cum = 5$
$[96, 231[\xrightarrow{A} [193, 231[= [11000001_2, 11100111_2[$: shift by 2
The next interval is $[00000100_2, 10011111_2[$ and $value = 00111011_2 = 59$

Next symbols are GAATAG and the current interval becomes $[111, 139[$ and $value = 132 = 10000100_2$

$cum = 10$
$[111, 139[\xrightarrow{A} [127, 139[= [01111111_2, 10001011_2[$: no shift
The interval is too short and is replaced by $[01111110_2, 10010111_2[$ and $value = 10001000_2 = 136$

$value = 136 = 10001000_2$
$cum = 6$
$[126, 151[\xrightarrow{G} [134, 138[= [10000110_2, 10001010_2[$: shift by 4
The next interval $[01100000_2, 10101111_2[$ and $value = 10001110_2 = 142$

$value = 142 = 10001110_2$
$cum = 9$
$[96, 175[\xrightarrow{A} [141, 175[= [10001101_2, 10101111_2[$: shift by 2
The next interval $[00110100_2, 10111111_2[$ and $value = 00111000_2 = 56$

$value = 56 = 00111000_2$
$cum = 0$
The symbol is END, the decoding process is over.

Maintaining the symbols in decreasing order of frequencies can be done in $O(\log \sigma)$ using a suitable data structure (see [6]).

12.5 LZW Coding

Ziv and Lempel designed a compression method using encoding **segments**. These segments of the original text are stored in a dictionary that is built during the compression process. When a segment of the dictionary is encountered later while scanning the text it is substituted by its index in the dictionary. In the model where portions of the text are replaced by pointers on previous occurrences, the Ziv–Lempel compression

scheme can be proved to be asymptotically optimal (on large enough texts satisfying good conditions on the probability distribution of symbols).

The dictionary is the central point of the algorithm. It has the property of being prefix-closed (every prefix of a word of the dictionary is in the dictionary), so that it can be implemented efficiently as a trie. Furthermore, a hashing technique makes its implementation efficient. The version described in this section is called the Lempel–Ziv–Welsh method after several improvements introduced by Welsh. The algorithm is implemented by the `compress` command existing under the UNIX operating system.

Encoding

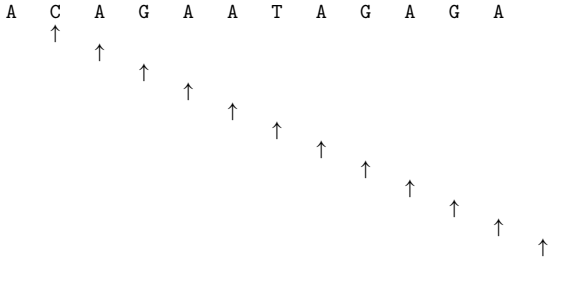
We describe the scheme of the coding method. The dictionary is initialized with all strings of length 1, the characters of the alphabet. The current situation is when we have just read a segment w of the text. Let a be the next symbol (just following the given occurrence w). Then we proceed as follows:

- If wa is not in the dictionary, we write the index of w in the output file, and add wa to the dictionary. We then reset w to a and process the next symbol (following a).
- If wa is in the dictionary we process the next symbol, with segment wa instead of w.

Initially, the segment w is set to the first symbol of the source text, so that it is clear that "w belongs to the dictionary" is an invariant of the operations described above.

EXAMPLE 12.7:

The alphabet is the 8-bit ASCII alphabet, $y =$ `ACAGAATAGAGA`
The dictionary initially contains the ASCII symbols, their indices are their ASCII codewords.



w	written	added
A	65	AC, 257
C	67	CA, 258
A	65	AG, 259
G	71	GA, 260
A	65	AA, 261
A	65	AT, 262
T	84	TA, 263
A		
AG	259	AGA, 264
A		
AG		
AGA	264	
	256	

Decoding

The decoding method is symmetric to the coding algorithm. The dictionary is recovered while the decompression process runs. It is basically done in this way:

- Read a code c in the compressed file,
- Write in the output file the segment w having index c in the dictionary,
- Add the word wa to the dictionary where a is the first letter of the next segment.

In this scheme, the dictionary is updated after the next segment is decoded because we need its first symbol a to concatenate at the end of the current segment w. So, a problem occurs if the next index computed at encoding time is precisely the index of the segment wa. Indeed, this happens only in a very special case, in which the symbol a is also the first symbol of w itself. This arises if the rest of the text to encode starts with a segment $azazax$ (a a symbol, z a string) for which az belongs to the dictionary but aza does not. During the compression process the index of az is output, and aza is added to the dictionary. Next, aza

is read and its index is output. During the decompression process the index of aza is read while the first occurrence of az has not been completed yet, the segment aza is not already in the dictionary. However, since this is the unique case where the situation occurs, the segment aza is recovered by taking the last segment az added to the dictionary concatenated with its first letter a.

EXAMPLE 12.8:

Decoding the sequence 65, 67, 65, 71, 65, 65, 84, 259, 264, 256
The dictionary initially contains the ASCII symbols, their indices are their ASCII codewords.

read	written	added
65	A	
67	C	AC, 257
65	A	CA, 258
71	G	AG, 259
65	A	GA, 260
65	A	AA, 261
84	T	AT, 262
259	AG	TA, 263
264	AGA	AGA, 264
256		

The critical situation occurs when reading the index 264 because, at that moment, no word of the dictionary has this index.

Implementation

We describe how the dictionary, which is the main data structure of the method, can be implemented. It is natural to consider two implementations adapted for the two phases respectively, because the dictionary is not manipulated in the same manner during these phases. They have in common a dictionary implemented as a trie stored in a table D. A node p of the trie is just an index on the table D. It has the three following components:

- $parent(p)$, a link to the parent node of p,
- $label(p)$, a character,
- $code(p)$, the codeword (index in the dictionary) associated with p.

In the compression algorithm shown in Fig. 12.24, for a node p we need to compute its child according to some letter a. This is done with by hashing, with a hashing function defined on pairs in the form (p, a). This provides a fast access to the children of a node.

The function HASH-SEARCH, with input $(D, (p, a))$, returns the node q such that $parent(q) = p$ and $label(q) = a$, if such a node exists and NIL otherwise. The procedure HASH-INSERT, with input $(D, (p, a, c))$, inserts a new node q in the dictionary D with $parent(q) = p$, $label(q) = a$ and $code(q) = c$.

For the decompression algorithm, no hashing technique is necessary on the table representation of the trie that implements the dictionary. Having the index of the next segment, a bottom-up walk in the trie produces the mirror image of the expected segment. A stack is then used to reverse it. We assume that the function $string(c)$ performs this specific work for a code c. The bottom-up walk follows the parent links of the data structure. The function $first(w)$ gives the first character of the word w. These features are part of the decompression algorithm displayed in Fig. 12.25.

The Ziv–Lempel compression and decompression algorithms run both in time linear in the sizes of the files provided the hashing technique is implemented efficiently. Indeed, it is very fast in practice, except when the table becomes full and should be reset. In this situation, usual implementations also reset the whole dictionary to its initial value.

The main advantage of Ziv–Lempel compression method, compared to Huffman coding, is that it captures long repeated segments in the source file and thus often yields better compression ratios.

```
LZW-CODING (fin, fout)
1   count ← −1
2   for each character a ∈ Σ
3       do count ← count + 1
4          HASH-INSERT (D, (−1, a, count))
5   count ← count + 1
6   HASH-INSERT (D, (−1, END, count))
7   p ← −1
8   while not end of file fin and a is the next symbol
9         q ← HASH-SEARCH (D, (p, a))
10        if q = NIL
11           then write code(p) on 1 + log(count) bits in fout
12                count ← count + 1
13                HASH-INSERT (D, (p, a, count))
14                p ← HASH-SEARCH (D, (−1, a))
15           else p ← q
16  write p on 1 + log(count) bits in fout
17  write code(HASH-SEARCH (D, (−1, END)) in 1 + log(count) bits in fout
```

FIGURE 12.24 LZW encoding algorithm.

12.6 Experimental Results

The table of Fig. 12.26 contains a sample of experimental results showing the behavior of compression algorithms on different types of texts. The table is extracted from [16].

The source files are French text, C sources, Alphabet, and Random. Alphabet is a file containing a repetition of the line `abc...zABC...Z`. Random is a file where the symbols have been generated randomly, all with the same probability and independently of each others.

The compression algorithms reported in the table are the Huffman algorithm of Section 12.2, the Ziv–Lempel algorithm of Section 12.5, and a third algorithm called Factor. This latter algorithm encodes segments of the source text as Ziv–Lempel algorithm does. But the segments are taken among all segments already encountered in the text before the current position. The method gives usually better compression ratio but is more difficult to implement. Compression based on arithmetic as presented in this chapter gives compression ratios slightly better than Huffman coding.

The table of Fig. 12.26 gives in percentage the sizes of compressed files. Results obtained by Ziv–Lempel and Factor algorithms are similar. Huffman coding gives the best result for the Random file. Finally, experience shows that exact compression methods often reduce the size of data to 30–50% of their original size.

12.7 Research Issues and Summary

The statistical compression algorithm is from Huffman [9]. The UNIX command `pack` implements the algorithm.

The dynamic version was discovered independently by Faller [5] and Gallager [7]. Practical versions were given by Cormack and Horspool [3] and Knuth [10]. A precise analysis leading to an improvement was presented in [13]. The command `compact` of UNIX implements the dynamic Huffman coding.

It is unclear to whom precisely should be attributed the idea of data compression using arithmetic coding. It is sometimes attributed to Elias [4], and has become popular after the publication of the article of Witten, Neal, and Cleary [15]. An efficient data structure for the tables of frequencies is due to Fenwick [6]. The main interest in arithmetic coding for text compression is that the two different processes "modeling" the statistics of texts and "coding" can be made independent modules.

```
LZW-DECODING (fin, fout)
1   count ← −1
2   for each character a ∈ Σ
3      do count ← count + 1
4         HASH-INSERT (D, (−1, a, count))
5   count ← count + 1
6   HASH-INSERT (D, (−1, END, count))
7   c ← first code on 1 + log(count) bits from fin
8   write string(c) in fout
9   a ← first(string(c))
10  repeat
11     d ← next code on 1 + log(count) from fin
12     if d > count
13        then count ← count + 1
14             parent(count) ← c
15             label(count) ← a
16             write string(c)a in fout
17             c ← d
18        else a ← first(string(d))
19             if a ≠ END
20                then count ← count + 1
21                     parent(count) ← c
22                     label(count) ← a
23                     write string(d) in fout
17                     c ← d
24                else exit
25  forever
```

FIGURE 12.25 LZW decoding algorithm.

Source texts	French	C sources	Alphabet	Random
Sizes in bytes	62816	684497	530000	70000
Huffman	53.27%	62.10%	72.65%	**55.58%**
Ziv-Lempel	**41.46%**	34.16%	2.13%	63.60%
Factor	47.43%	**31.86%**	**0.09%**	73.74%

FIGURE 12.26 Sizes of texts compressed with three algorithms.

Several variants of the Ziv–Lempel algorithm exist. The reader can refer to the books of Bell, Cleary, and Witten [1] or Storer [12] for a discussion on them.

The books of Held [8] and Nelson [11] present practical implementations of various compression algorithms, while the book of Crochemore and Rytter [2] overflows the strict topic of text compression and described more algorithms and data structures related to texts manipulations.

Defining Terms

Codeword: Sequence of bits of a code corresponding to a symbol.

Prefix: A word $u \in \Sigma^*$ is a prefix of a word $w \in \Sigma^*$ if $w = uz$ for some $z \in \Sigma^*$.

Prefix code: Set of words such that no word of the set is a prefix of another word contained in the set. A prefix code is represented by a coding tree.

Segment: A word $u \in \Sigma^*$ is a segment of a word $w \in \Sigma^*$ if u occurs in w, i.e., $w = vuz$ for two words $v, z \in \Sigma^*$. (u is also referred to as a factor or a subword of w.)

Trie: Tree in which edges are labeled by letters or words.

References

[1] Bell, T.C., Cleary, J.G., and Witten, I.H., *Text Compression,* Prentice Hall, Englewood Cliffs, NJ, 1990.

[2] Chochemore, M. and Rytter, W., *Text Algorithms,* Oxford University Press, 1994.

[3] Cormack, G.V. and Horspool, R.N.S., Algorithms for adaptive Huffman Codes. *Inf. Process. Lett.,* 18(3), 159–165, 1984.

[4] Elias, 1963.

[5] Faller, N., An adaptive system for data compression. In *Record of the 7th Asilomar Conference on Circuits, Systems, and Computers,* 593–597, 1973.

[6] Fenwick, P.M., A new data structure for cumulative frequency tables. *Software—Practice and Experience,* 24(7), 327–336, 1994.

[7] Gallager, R.G., Variations on a theme by Huffman. *IEEE Trans. Inf. Theory,* 24(6), 668–674, 1978.

[8] Held, G., *Data Compression,* John Wiley & Sons, New York, 1991.

[9] Huffman, D.A., A method for the construction of minimum redundancy codes. *Proceedings of the I.R.E.,* 40, 1098–1101, 1951.

[10] Knuth, D.E., Dynamic Huffman coding. *J. Algorithms,* 6(2), 163–180, 1985.

[11] Nelson, M., *The Data Compression Book,* M&T Books, 1992.

[12] Storer, J.A., *Data Compression: Methods and Theory,* Computer Science Press, 1988.

[13] Vitter, J.S., Design and analysis of dynamic Huffman codes. *J. ACM,* 34(4), 825–845, 1987.

[14] Welch, T.A., A technique for high-performance data compression. *IEEE Computer,* 17(6), 8–19, 1984.

[15] Witten, I.H., Neal, R., and Cleary, J.G., Arithmetic coding for data compression. *Comm. ACM,* 30(6), 520–540, 1987.

[16] Zipstein, M., Data compression with factor automata, *Theoret. Comput. Sci.,* 92, 213–221, 1992.

[17] Ziv, J. and Lempel, A., A universal algorithm for sequential data compression. *IEEE Trans. Inf. Theory,* 23(3), 337–343, 1977.

Further Information

The set of algorithms presented in this chapter provides the basic methods for data compression. In commercial software they are often combined with other more elementary techniques that are described in textbooks. A wider panorama of data compression algorithms on texts may be found in several books such as:

- Bell, T.C., Cleary, J.G., Witten, I.H. 1990. *Text Compression*. Prentice Hall, Englewood Cliffs, NJ.
- Crochemore, M., Rytter, W. 1994. *Text Algorithms.* Oxford University Press.
- Held, G. 1991. *Data Compression*. John Wiley & Sons, New York.
- Nelson, M. 1992. *The Data Compression Book.* M&T Books.
- Storer, J.A. 1988. *Data Compression: Methods and Theory.* Computer Science Press.

Research papers in text data compression are disseminated in a few journals, including *Communications of the ACM, Journal of the ACM, Theoretical Computer Science, Algorithmica, Journal of Algorithms, SIAM Journal on Computing, IEEE Trans. Information Theory.*

An annual conference presents the latest advances of this field of research:

- *Data Compression Conference,* which is regularly held at Snowbird (Utah) in spring.

Two other conferences on pattern matching also present research issues in this domain:

- *Combinatorial Pattern Matching*, which started in 1990 and was held in Paris (France), London (England), Tucson (Arizona), Padova (Italy), Asilomar (California), Helsinki (Finland), Laguna Beach (California), Aarhus (Denmark).
- *Workshop on String Processing,* which started in 1993 and was held in Belo Horizonte (Brasil), Valparaiso (Chile), and Recife (Brasil).

And general conferences in computer science often have sessions devoted to data compression algorithms.

13

General Pattern Matching

Alberto Apostolico
Purdue University and
Università di Padova

13.1 Introduction

This chapter reviews combinatorial and algorithmic problems related to searching and matching of strings and slightly more complicated structures such as arrays and trees. These problems arise in a vast variety of applications and in connection with various facets of storage, transmission, and retrieval of information. A list would include the design of structures for the efficient management of large repositories of strings, arrays and special types of graphs, fundamental primitives such as the various variants of exact and approximate searching, specific applications such as the identification of periodicities and other regularities, efficient implementations of ancillary functions such as compression and encoding of elementary discrete objects, etc. The main objective of studies in pattern matching is to abstract and identify some primitive manipulations, develop new techniques and efficient algorithms to perform them, both by serial and parallel or distributed computation, and implement the new algorithms.

Some initial pattern matching problems and techniques arose in the early 1970s in connection with emerging technologies and problems of the time, e.g., compiler design. Since then, the range of applications of the tools and methods developed in pattern matching has expanded to include text, image and signal processing, speech analysis and recognition, data compression, computational biology, computational chemistry, computer vision, information retrieval, symbolic computation, computational learning, computer security, graph theory and VLSI, etc. In little more than two decades, an initially sparse set of

more or less unrelated results has grown into a considerable body of knowledge. A complete bibliography of string algorithms would contain more than 500 articles. A.V. Aho [2] references over 140 papers in his recent survey of string-searching algorithms alone; advanced workshops and schools, books and special issues in major journals have already been dedicated to the subject and more are planned for the future. The interested reader will find a few reference books and conference proceedings in the bibliography of this chapter.

While each application domain presents peculiarities of its own, a number of pattern matching primitives are shared, in nearly identical forms, within wide spectra of distant and diverse areas. For instance, searching for identical or similar substrings in strings is of paramount interest to software development and maintenance, philology or plagiarism detection in the humanities, inference of common ancestries in molecular genetics, comparison of geological evolutions, stereo matching for robot vision, etc. Checking the equivalence (i.e., identity up to a rotation) of circular strings finds use in determining the homology of organisms with circular genomes, comparing closed curves in computer vision, establishing the equivalence of polygons in computer graphics, etc. Finding repeated patterns, symmetries, and cadences in strings is of interest to data compression, detection of recurrent events in symbolic dynamics, genome studies, intrusion detection in distributed computer systems, etc. Similar considerations hold for higher structures. In general, an intermediate objective of studies in pattern matching is to understand and characterize combinatorial structures and properties that are susceptible of exploitation in computational matching and searching on discrete elementary structures.

Most pattern matching issues are still subject to extensive investigation both within serial and parallel models of computation. This survey concentrates on sequential algorithms, but the reader is encouraged to explore for himself the rich repertoire of parallel algorithms developed in recent years. Most of these algorithms bear very little resemblance to their serial counterparts. Similar considerations apply to some algorithms formulated in a probabilistic setting.

Pattern matching problems may be classified according to a number of paradigms. One way is based on the type of structure (strings, arrays, trees, etc.) in terms of which they are posed. Another, is according to the model of computation used, e.g., RAM, PRAM, Turing machine. Yet another one is according to whether the manipulations that one seeks to optimize need be performed on-line, off-line, in real time, etc. One could distinguish further between matching and searching and, within the latter, between exact and approximate searches, or vice versa. The classification used here is thus somewhat arbitrary. We assume some familiarity of the reader with exact string searching, both on- and off-line, which is covered in a separate chapter of this volume. We start by reviewing some basic variants of string searching where occurrences of the pattern need not be exact. Next, we review algorithms for string comparisons. Then, we consider pattern matching on two-dimensional arrays and finally on rooted trees.

13.2 String Searching with Don't-Care Symbols

As already mentioned, we assume familiarity of the reader with the problem of **exact string searching,** in which we are interested in finding all the occurrences of a pattern string y into a textstring x. One of the natural departures from this formulation consists of assuming that a symbol can (perhaps only at some definite positions) match a small group of other symbols. At one extreme we may have, in addition to the symbols in the input alphabet Σ, a **don't care** symbol ϕ with the property that ϕ matches any other character in Σ. This gives raise to variants of string searching where, in principle, ϕ appears (i) only in the pattern, (ii) only in the text, or (iii) both in pattern and text. There seems to be no peculiar result on variant (ii), whence we shall consider this as just a special case of (iii). The situation is different with variant (i), which warrants the separate treatment which is given next.

Don't-Cares in Pattern Only

Fischer and Paterson [20] and Pinter [45] discuss the problem faced if one tried to extend the KMP string searching algorithm [33] in order to accommodate don't cares in the pattern: the obvious transitivity on character equality, that subtends those and other exact string searching, is lost with don't cares. Pinter noted that a partial recovery is possible if the number and positions of don't cares is fixed. In fact, in this case one may resort to ideas used by Aho and Corasick [3] in connection with exact multiple string searching and solve the problem within the same time complexity $O(n + m + r) \log |\Sigma|$ time, where $n = |x|$ is the length of the textstring, $m = |y|$ is the length of the pattern, and r is the total number of occurrences of the fragments of the pattern that would be obtained by cleaving the latter at don't cares.

We outline Pinter's approach. Since the don't cares appear in fixed known positions, we may consider the pattern decomposed into **segments** of Σ^+, say, $\hat{y_1}, \hat{y_2}, \ldots, \hat{y_p}$ and ϕ-**blocks** consisting of runs of occurrences of ϕ, respectively. Each $\hat{y_i}$ can be treated as an individual pattern in a multiple pattern matching machine. Through the search, one computes for each $\hat{y_i}$ a list of its occurrences in x. Let d_i be the known distance between the starting positions of $\hat{y_i}$ and $\hat{y}_{i+1}$. We may now merge the occurrence list while keeping track of these distances, using the natural observation that if a match occurred starting at position j, then the $\hat{y_i}$'s will appear in the same order and intersegment distance as they appear in the pattern. Here the merge process takes place after the search. To make his algorithm work in real time applications, Pinter used an array of counters, a data structure originally proposed by R.L. Rivest. Instead of merging lists, counters count from 0 to p while collecting evidence of a pattern occurrence. Specifically, the counting mechanism is as follows. Let the **offset** of a segment be the distance from the beginning of the pattern to the end of that segment. Whenever a segment match is detected ending at position j, then its offset f_j is subtracted from j, thus yielding the starting position $j - f_i$ of a corresponding candidate occurrence of the pattern. Next, the counter assigned to position $1 + (j - f_i) \bmod m$ is incremented by 1. Therefore, a counter initialized to zero reaches p iff the last m characters examined on the text matched the pattern. A check whether a counter has reached p is performed each time that counter is reused.

Manber and Baeza-Yates [39] consider the case where the pattern embeds a string of at most k don't cares, i.e., has the form $y = u\phi^i v$, where $i \leq k$, $u, v \in \Sigma^*$, and $|u| \leq m$ for some given k, m. Their algorithm is off-line in the sense that the text x is preprocessed to build the suffix array [40] associated with it. This operation costs $O(n \log |\Sigma|)$ time in the worst case. Once this is done, the problem reduces to one of efficient implementation of 2-dimensional orthogonal range queries (for these latter see, e.g., [16, 60]).

One more variant of string searching with don't care in pattern only is discussed in [55]. Also Takeda's algorithm is based on the algorithm in [4]. The problem is stated as follows. Consider a set $A = \{A_1, A_2, \ldots, A_p\}$, where each $A_i \subseteq \Sigma$ is a nonempty set called a **picture** and pictures are mutually disjoint. While a don't care symbol matches all symbols, a picture matches a subset of the alphabet. For any pattern y, we have now that $y \in (\Sigma \cup A)^+$. Then, given a set of patterns $Y = \{y^{(1)}, \ldots, y^{(k)}\}$, the problem is to find all occurrences of $y^{(i)}$ for $i = 1, ..., k$. Thus, when $A = \Sigma$, the problem reduces to plain string searching with don't cares. A pattern matching machine for such a family can be quite time consuming to build. Takeda managed to improve on time efficiency by saving on the number of explicit "goto" edges created in that machine.

Takeda's variant finds natural predecessors in an even more general class considered by K. Abrahamson [1]. This latter paradigm applies to an unbounded alphabet Σ, as long as individual symbols have finite encodings. Let $P = \{P_1, P_2, \ldots, P_k\}$ be a set of **pattern elements,** where each pattern element is a subset of Σ. There are **positive** and **negative** pattern elements. A positive element, is denoted by $< \sigma_1, \ldots, \sigma_f >$ and has the property of matching each one of the characters $\sigma_1, \sigma_2, \ldots, \sigma_f$. A negative element is denoted by $[\sigma_1, \ldots, \sigma_f]$ and will match any character of Σ except characters $\sigma_1, \sigma_2, \ldots, \sigma_f$. A pattern $y \in P^+$ identifies now a family of strings, namely, all strings in the form $y_1 y_2 \ldots y_m$ such that $y_i \in \Sigma$ is compatible with the element of P used to identify the ith element of y. Using a time–space tradeoff proof technique

due to Borodin, Abrahamson proved that the time-space lower bound on a subproblem with $n = 2m$ is $\Omega(m^2/\log m)$.

By combining **divide and conquer** with an idea of Fischer and Paterson [20] which will be discussed more thoroughly later, Abrahamson designed an algorithm taking time $O(N + M + n\hat{M}^{1/2} \log m \log^{1/2} m)$, where N and M denote the lengths of the encodings (e.g., in bits) of x and y, respectively, and $\hat{M}$ represents the number of distinct elements of Σ which are present in the pattern.

Don't-Cares in Pattern and Text

In an elegant, landmark paper, Fischer and Paterson [20] exposed the similarity of string searching to multiplication, thereby obtaining a number of interesting algorithms for exact string searching and some of its variants. It is not difficult to see that string matching problems can be rendered as special cases of a general linear product. Given two vectors X and Y, their **linear product** with respect to two suitable operations $\otimes$ and $\oplus$, is denoted by $X \begin{smallmatrix}\otimes\\ \oplus\end{smallmatrix} Y$, and is a vector $Z = Z_0 Z_1 \ldots Z_{m+n}$ where $Z_k = \bigoplus_{i+j=k} X_i \bigotimes Y_j$ for $k = 0, \ldots, m+n$. If we interpret $\oplus$ as the Boolean $\wedge$ and $\otimes$ as the symbol equivalence $\equiv$, then a match of the reverse Y^R of Y, occurs ending at position k in X, where $m \leq k \leq n$, if $[X_{k-m} \ldots X_k] \equiv [Y_m \ldots Y_0]$, that is, with obvious meaning, if $(X \begin{smallmatrix}\equiv\\ \wedge\end{smallmatrix} Y)_k$ =TRUE. This observation brings string searching into the family of Boolean, polynomial, and integer multiplications, and leads quickly to an $O(n \log m \log\log m)$ time solution even in the presence of don't cares, provided that the size of Σ is fixed.

To see this, we show first that string searching can be regarded as a Boolean linear product, i.e., one where $\oplus$ is $\vee$ and $\otimes$ is $\wedge$. Let the textstring be specified as $x = x_0 x_1 x_2 \ldots x_n$ and similarly let $y = y_0 y_1 y_2 \ldots y_m$ be the pattern. Recall that we assume a finite alphabet Σ, and that both x and y may contain some don't cares. For each $\rho \in \Sigma$, define $H_\rho(X_i) = 1$ if $x_i = \rho$, and $H_\rho(X_i) = 0$ if $x_i \neq \rho$ or $x_i = \phi$. Assume now that the vector X corresponding to string x contains only symbol σ and ϕ while Y, corresponding to string y, contains only symbol $\tau \neq \sigma$ and ϕ, with both σ and $\tau \in \Sigma$. Then $\bigwedge_{i+j=k} X_i \equiv Y_j$ means that $\bigwedge_{i+j=k} \neg H_\sigma(X_i) \vee \neg H_\tau(Y_j) \Longleftrightarrow \neg \bigvee_{i+j=k} H_\sigma(X_i) \wedge H_\tau(Y_j)$. The last term is a boolean product, whence such a product is not harder than string searching. On the other hand, $Z = X \begin{smallmatrix}\equiv\\ \wedge\end{smallmatrix} Y = \neg \bigvee_{\sigma \neq \tau; \sigma, \tau \in \Sigma} H_\sigma(X) \begin{smallmatrix}\wedge\\ \vee\end{smallmatrix} H_\tau(Y)$.

As is well known, the Boolean product can be obtained by performing the polynomial product, in which $\oplus$ is + and $\otimes$ is $\times$. For this, just encode TRUE and FALSE as 1 and 0, respectively. One way to compute the polynomial product is to embed the product in a single large integer multiplication. There are well known fast solutions for the latter problem. For the $\{0, 1\}$ string vectors X and Y, the maximum coefficient is $m + 1$, so if we choose r such that $2^r > m + 1$, compute the integers $X(2^r) = \Sigma_{i=0}^{n} X_i \cdot 2^{ri}$ and $Y(2^r) = \Sigma_{j=0}^{m} Y_j \cdot 2^{rj}$ and then multiply $X(2^r)$ and $Y(2^r)$, the result will be the product evaluated at 2^r. The consecutive blocks of length r in the binary representation of $Z(2^r)$ will give the coefficients of Z, which can be transferred back to the Boolean product, and from there back to the string matching product. The Schönhage–Strassen [53] algorithm multiplies an N-digit number by an M-digit number in time $O(N \cdot \log M \cdot \log\log M)$, for $N > M$, using a multi-tape Turing machine. For the string matching product, $N = nr = O(n \log m)$, $M = mr = O(m \log m)$, so that the problem is solved on that machine in time $O(n \log^2 m \log\log m)$. The algorithm as presented assumes that the alphabet finite. For any alphabet Σ of size polynomial in n, however, we can always encode the two input strings in binary at a cost of a multiplicative factor $O(\log |\Sigma|)$, and then execute just two Boolean products. This results in an extra $O(\log m)$ factor in the time complexity.

Note the adaptation of fast multiplication to string searching provides a basis for counting the mismatches generated by a pattern y at every position of a text x. This results from treating all symbols of Σ separately, and thus in overall time $O(n(|\Sigma|) \log^2 m \log\log m)$. This latter complexity is comparable to the above only for finite Σ. However, we shall see later that better bounds are achievable under this approach.

13.3 String Editing and Longest Common Subsequences

We now introduce three **edit operations** on strings. Namely, given any string w we consider the **deletion** of a symbol from w, the **insertion** of a new symbol in w, and the **substitution** of one of the symbols of w with another symbol from Σ. We assume that each edit operation has an associated nonnegative real number representing the **cost** of that operation. More precisely, the cost of deleting from w an occurrence of symbol a is denoted by $D(a)$, the cost of inserting some symbol a between any two consecutive positions of w is denoted by $I(a)$, and the cost of substituting some occurrence of a in w with an occurrence of b is denoted by $S(a, b)$. An **edit script** on w is any sequence Γ of viable edit operations on w, and the cost of Γ is the sum of all costs of the edit operations in Γ.

Now, let x and y be two strings of respective lengths $|x| = n$ and $|y| = m \leq n$. The **string editing problem** for input strings x and y consists of finding an edit script Γ' of minimum cost that transforms y into x. The cost of Γ' is the **edit distance from** y *to* x. Edit distances where individual operations are assigned integer or unit costs occupy a special place. Such distances are often called Levenshtein distances, since they were introduced by Levenshtein [38] in connection with error correcting codes. String editing finds applications in a broad variety of contexts, ranging from speech processing to geology, from text processing to molecular biology.

It is not difficult to see that the general (i.e., with unbounded alphabet and unrestricted costs) problem of edit distance computation is solved by a serial algorithm in $\Theta(mn)$ time and space, through dynamic programming. Due to widespread application of the problem, however, such a solution and a few basic variants were discovered and published in literature catering to diverse disciplines (see, e.g., [44, 47, 49, 59]). In computer science, the problem was dubbed "the string-to-string correction problem." The CS literature was possibly the last one to address the problem, but the interest in the CS community increased steadily in subsequent years. By the early 1980s, the problem had proved so pervasive, especially in biology, that a book by Sankoff and Kruskal [1983] was devoted almost entirely to it. Special issues of the *Bulletin of Mathematical Biology* and various other books and journals routinely devote significant portions to it.

An $\Omega(mn)$ lower bound was established for the problem by Wong and Chandra [61] for the case where the queries on symbols of the string are restricted to tests of equality. For unrestricted tests, a lower bound $\Omega(n \log n)$ was given by Hirschberg [27]. Algorithms slightly faster than $\Theta(mn)$ were devised by Masek and Paterson [41], through resort to the so-called "Four Russians Trick." The "Four Russians" are Arlazarov, Dinic, Kronrod, and Faradzev [8]. Along these lines, the total execution time becomes $\Theta(n^2/\log n)$ for bounded alphabets and $O(n^2(\log\log n)/\log n)$ for unbounded alphabets. The method applies only to the classical Levenshtein distance metric, and does not extend to general cost matrices. To this date, the problem of finding either tighter lower bounds or faster algorithms is still open.

The criterion that subtends the computation of edit distances by dynamic programming is readily stated. For this, let $C(i, j)$, $(0 \leq i \leq |y|,\ 0 \leq j \leq |x|)$ be the minimum cost of transforming the prefix of y of length i into the prefix of x of length j. Let w_k denote the kth symbol of string w. Then $C(0, 0) = 0$, $C(i, 0) = C(i-1, 0) + D(y_i)$ $(i = 1, 2, \ldots, |y|)$, $C(0, j) = C(0, j-1) + I(x_j)$ $(j = 1, 2, \ldots, |x|)$, and

$$C(i, j) = \min\left\{C(i-1, j-1) + S\left(y_i, x_j\right),\ C(i-1, j) + D\left(y_i\right),\ C(i, j-1) + I\left(x_j\right)\right\}$$

for all i, j, $(1 \leq i \leq |y|, 1 \leq j \leq |x|)$. Observe that, of all entries of the C-matrix, only the three entries $C(i-1, j-1)$, $C(i-1, j)$, and $C(i, j-1)$ are involved in the computation of the final value of $C(i, j)$. Hence, $C(i, j)$ can be evaluated row-by-row or column-by-column in $\Theta(|y||x|) = \Theta(mn)$ time. An optimal edit script can be retrieved at the end by backtracking through the local decisions that were made by the algorithm.

A few important problems are special cases of string editing, including the **longest common subsequence** problem, **local alignment,** i.e., the detection of local similarities of the kind sought typically in the analysis of molecular sequences such as DNA and proteins, and some important variants of **string searching with**

errors, or searching for approximate occurrences of a pattern string in a text string. As highlighted in the following brief discussion, a solution to the general string editing problem implies typically similar bounds for all these special cases.

Longest Common Subsequences

Perhaps the single most widely studied special case of string editing is the so-called longest common subsequence (LCS) problem. The problem is defined as follows. Given a string z over an alphabet $\Sigma = (i_1, i_2, \ldots i_s)$, a **subsequence** of z is any string w that can be obtained from z by deleting zero or more (not necessarily consecutive) symbols. The **longest common subsequence problem** for input strings $x = x_1 x_2 \ldots x_n$ and $y = y_1 y_2 \ldots y_m$ $(m \leq n)$ consists of finding a third string $w = w_1 w_2 \ldots w_l$ such that w is a subsequence of x and also a subsequence of y, and w is of maximum possible length. In general, string w is not unique.

Like the string editing problem itself, the LCS problem arises in a number of applications spanning from text editing to molecular sequence comparisons, and has been studied extensively over the past. Its relation to string editing can be understood as follows.

Observe that the effect of a given substitution can be always achieved, alternatively, through an appropriate sequence consisting of one deletion and one insertion. When the cost of a nonvacuous substitution (i.e., a substitution of a symbol with a different one) is higher than the global cost of one deletion followed by one insertion, then an optimum edit script will always avoid substitutions and produce instead y from x solely by insertions and deletions of overall minimum cost. Specifically, assume that insertions and deletions have unit costs, and that a cost higher than 2 is assigned to substitutions. Then, the pairs of matching symbols preserved in an optimal edit script constitute a longest common subsequence of x and y. It is not difficult to see that the cost e of such an optimal edit script, the length l of an LCS and the lengths of the input strings obey the simple relationship: $e = n + m - 2l$. Similar considerations can be developed for the variant where matching pairs are assigned weights and a **heaviest** common subsequence is sought (see, e.g., Jacobson and Vo [31]).

Lower bounds for the LCS problem are time $\Omega(n \log n)$ or linear time, according to whether the size s of Σ is unbounded or bounded [27]. Aho, Hirschberg and Ullman [4] showed that, for unbounded alphabets, any algorithm using only "equal–unequal" comparisons must take $\Omega(mn)$ time in the worst case. The asymptotically fastest general solution rests on the corresponding solution by Masek and Paterson [41] to the string editing, hence takes time $O(n^2 \log\log n / \log n)$. Time $\Theta(mn)$ is achieved by the following dynamic programming algorithm from Hirschberg [26].

Let $L[0 \ldots m, 0 \ldots n]$ be an integer matrix initially filled with zeroes.

The following code transforms L in such a way that $L[i, j]$ $(1 \leq i \leq m, 1 \leq j \leq n)$ contains the length of an LCS between $y_1 y_2 \ldots y_i$ and $x_1 x_2 \ldots x_j$.

for $i = 1$ **to** m **do**
  for $j = 1$ **to** n **do if** $y_i = x_j$ **then** $L[i, j] = L[i-1, j-1] + 1$
    else $L[i, j] = \text{Max}\{L[i, j-1], L[i-1, j]\}$

The correctness of this strategy follows from the obvious relations:

$$\begin{array}{lcccl} L[i-1, j] & \leq & L[i, j] & \leq & L[i-1, j] + 1\,; \\ L[i, j-1] & \leq & L[i, j] & \leq & L[i, j-1] + 1\,; \\ L[i-1, j-1] & \leq & L[i, j] & \leq & L[i-1, j-1] + 1\,. \end{array}$$

If only the length l of an LCS is desired, then this code can be adapted to use only linear space. If an LCS is required at the outset, then it is necessary to keep track of the decision made at every step by the algorithm, so that an LCS w can be retrieved at the end by backtracking. The early $\Theta(mn)$ time algorithm by Hirschberg [27] achieved both a linear space bound and the production of an LCS at the outset, through

a combination of dynamic programming and divide-and-conquer. Subsequent approaches to the LCS problem achieve time complexities better than $\Theta(mn)$ in favorable cases, though a quadratic performance is always touched and sometimes even exceeded in the worst cases. These approaches exploit in various ways the **sparsity** inherent to the LCS problem. Sparsity allows us to relate algorithmic performances to parameters other than the lengths of the input. Some such parameters are introduced next.

The ordered pair of positions i and j of L, denoted $[i, j]$, is a **match** iff $y_i = x_j$, and we use r to denote the total number of matches between x and y. If $[i, j]$ is a match, and an LCS $w_{i,j}$ of $y_1 y_2 \ldots y_i$ and $x_1 x_2 \ldots x_j$ has length k, then k is the **rank** of $[i, j]$. The match $[i, j]$ is **k-dominant** if it has rank k and for any other pair $[i', j']$ of rank k either $i' > i$ and $j' \leq j$ or $i' \leq i$ and $j' > j$. A little reflection establishes that computing the k-dominant matches ($k = 1, 2, \ldots, l$) is all is needed to solve the LCS problem (see, e.g., [7, 26]). Clearly, the LCS of x and y has length l iff the maximum rank attained by a dominant match is l. It is also useful to define, on the set of matches in L, the following partial order relation $\mathcal{R}$: match $[i, j]$ precedes match $[i', j']$ in $\mathcal{R}$ if $i < i'$ and $j < j$. A set of matches such that in any pair one of the matches always precedes the other in $\mathcal{R}$ constitutes a **chain** relative to the partial order relation $\mathcal{R}$. A set of matches such that in any pair neither match precedes the other in $\mathcal{R}$ is an **antichain**. Then, the LCS problem translates into the problem of finding a longest chain in the **poset** (partially ordered set) of matches induced by $\mathcal{R}$ (cf. [48]). A decomposition of a poset into antichains is **minimal** if it partitions the poset into the minimum possible number of antichains (refer, e.g., to [12]).

THEOREM 13.1 *[17] A maximal chain in a poset P meets all antichains in a minimal antichain decomposition of P.*

In other words, the number of antichains in a minimal decomposition represents also the length of a longest chain. Even though it is never explicitly stated, most known approaches to the LCS problem in fact compute a minimal antichain decomposition for the poset of matches induced by $\mathcal{R}$. The kth antichain in this decomposition is represented by the set of all matches having rank k. For general posets, a minimal antichain decomposition is computed by flow techniques [12], although not in time linear in the number of elements of the poset. Most LCS algorithms that exploit sparsity have their natural predecessors in either [30] or [26]. In terms of antichain decompositions, the approach of Hirschberg [26] consists of computing the antichains in succession, while that of Hunt and Szymanski [30] consists of extending partial antichains relative to all ranks already discovered, one new symbol of y at a time. The respective time complexities are $O(nl + n \log s)$ and $O(r \log n)$. Thus, the algorithm of Hunt and Szymanski is favorable in very sparse cases, but worse than quadratic when r tends to mn. An important specialization of this algorithm is that to the problem of finding a longest ascending subsequence in a permutation of the integers from 1 to n. Here, the total number of matches is n, which results in a total complexity $O(n \log n)$. Resort to the "fat-tree" structures introduced by Van Emde Boas [19] leads to $O(n \log \log n)$ for this problem, a bound which had been shown to be optimal by Fredman [21].

Figure 13.1 illustrates the concepts introduced thus far, displaying the final L-matrix for the strings $x =$ cbadbb and $y =$ abcabccbc. We use circles to represent matches, with bold circles denoting dominant matches. Dotted lines thread antichains relative to $\mathcal{R}$ and also separate regions.

Hirschberg's Paradigm: Finding Antichains One at a Time

We outline a $\Theta(mn)$ time LCS algorithm in which antichains of matches relative to the various ranks are discovered one after the other. Consider the *dummy* pair $[0, 0]$ as a "0-dominant match," and assume that all $(k-1)$-dominant matches for some k, $0 \leq k \leq l-1$ have been discovered at the expense of scanning the part of the L-matrix that would lie above or to the left of the antichain $(k-1)$, inclusive. To find the kth antichain, scan the unexplored area of the L-matrix from right to left and top-down, until a stream of matches is found occurring in some row i. The leftmost such match is the k-dominant match $[i, j]$ with

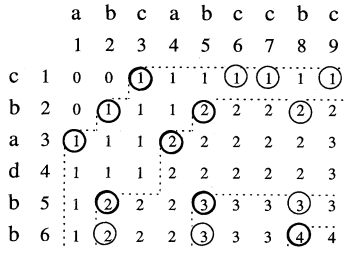


FIGURE 13.1 Illustrating antichain decompositions.

smallest i-value. The scan continues at next row and to the left of this match, and the process is repeated at successive rows until all of the kth antichain has been identified. The process may be illustrated like in Fig. 13.2. The large circles denote "pebbles" used to intercept the matches in an antichain. Initially, the pebbles are positioned on the matches created in the last column by the *ad-hoc* wildcard symbol \$. Next, pebbles are considered in succession from the top, and each pebble is moved to the leftmost match it can reach without crossing a previously discovered antichain. Once all pebbles have been considered, those contributing to the new antichain are identified easily.

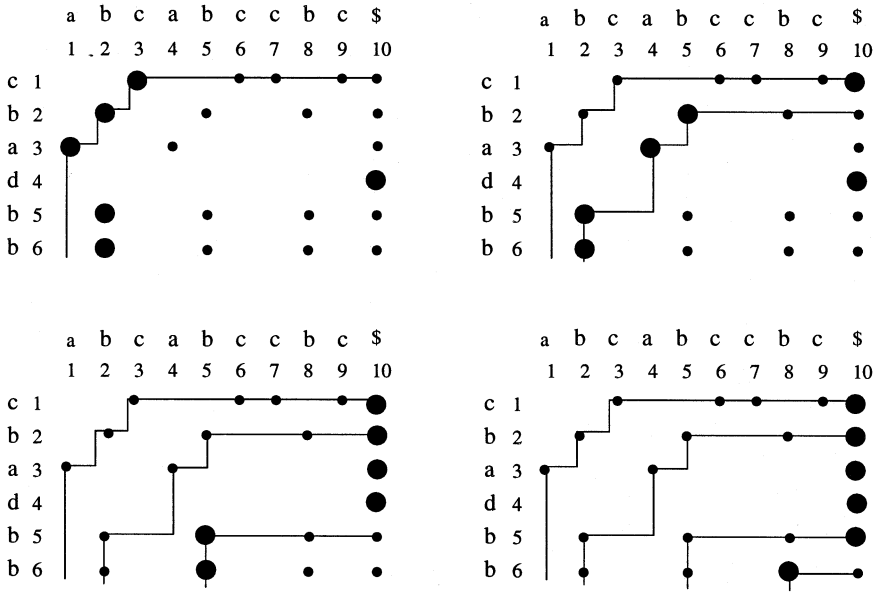


FIGURE 13.2 Hirschberg's paradigm: discovering one antichain at a time. The positions occupied by the pebbles at the end of consecutive antichain constructions are displayed clockwise from left-top.

Note that for each k the list of $(k-1)$-dominant matches is enough to describe the shape of the antichain and also to guide the searches involved at the subsequent stage. Thus, also in this case linear space is sufficient if one wishes to compute only the length of w.

An efficient implementation of this scheme leads to the algorithm by Hirschberg [26], which takes time $O(nl + n \log s)$ and space $O(d + n)$, where d is the number of dominant matches.

Incremental Antichain Decompositions and the Hunt–Szymanski Paradigm

When the number r of matches is small compared to m^2 (or to the expected value of lm), an algorithm with running time bounded in terms of r may be advantageous. Along these lines, Hunt and Szymanski [30] set up an algorithm (HS) with a time bound of $O((n + r) \log n)$. This algorithm works by computing, row after row, the ranks of all matches in each row. The treatment of a new row corresponds thus to extending the antichain decomposition relative to all preceding rows. A same match is never considered more than once. On the other hand, the time required by HS degenerates as r gets close to mn. In these cases this algorithm is outperformed by the algorithm of Hirschberg [26], which exhibits a bound of $O(ln)$ in all situations.

Algorithm HS is reproduced below. Essentially, it scans the list of matching positions $MATCHLIST$ associated with the ith row of L and considers those matches in succession, from right to left. For each match, HS decides whether it is a k-dominant match for some k through a binary search in the array $THRESH$ which maintains the leftmost previously discovered k-dominant match for each k. If the current match forces an update for rank k, then the contents of $THRESH[k]$ is modified accordingly. Observe that considering the matches in reverse order is crucial to the correct operation of HS. The details are found in the code below.

Algorithm "HS": element array $y[1 : m], x[1 : n]$;
integer array $THRESH[0 : m]$; list array $MATCHLIST[1 : m]$;
pointer array $LINK[1 : m]$; pointer PTR;
begin (*PHASE* 1: initializations
  for $i = 1$ **to** m **do**
  begin
    $MATCHLIST[i] = \{ j_1, j_2, ..., j_p\}$
    such that $j_1 > j_2 > \cdot > j_p$
    and $y_i = x_{j_q}$ *for* $1 \le q \le p$
    $THRESH[i] = n + 1$ *for* $1 \le i \le m$;
  end
  $THRESH[0] = 0$; $LINK[0] = null$;
PHASE 2 : find k-dominant matches)
  for $i = 1$ to m do
    for j *on* $MATCHLIST[i]$ do
      begin *find k such that*
        $THRESH[k-1] < j \le THRESH[k]$;
      if $j < THRESH[k]$ **then**
        begin
        $THRESH[k] = j$;
          $LINK[k] = newnode(i, j, LINK[k-1])$
        end
      end
(*PHASE* 3 : recover LCS w in reverse order)
  $\hat{k}$ = *largest k such that* $THRESH[k] \neq n + 1$;
  $PTR = LINK[\hat{k}]$;
  while $PTR \neq null$ **do begin**
  print the match $[i, j]$ *pointed to by* PTR;
  advance PTR **end**
end.

The total time spent by HS is bounded by $O((r + m) \log n + n \log s)$, where the $n \log s$ term is

charged by the preprocessing needed to create the lists of matches. The space is bounded by $O(d+n)$. As mentioned, this is good in sparse cases but becomes worse than quadratic for dense r.

13.4 String Searching with Errors

In this section, we assume unit cost for all edit operations. Given a pattern y and a text x, the most general variant of the problem consists of computing, for every position of the text, the best edit distance achievable between y and any substring w of x ending at that position. It is not difficult to express a solution in terms of a suitable adaptation of the recurrence previously introduced in connection with string editing. The first obvious change consists of setting all costs to 1 except that $S(y_i, x_j) = 0$ for $y_i = x_j$. Thus, we have now, for all i, j, $(1 \leq i \leq |y|, 1 \leq j \leq |x|)$,

$$C(i, j) = \min\{C(i-1, j-1) + 1,\ C(i-1, j) + 1,\ C(i, j-1) + 1\}\ .$$

A second change consists of setting the initial conditions so that $C(0,0) = 0$, $C(i,0) = i$ $(i = 1, 2, \ldots, m)$, $C(0, j) = 0$ $(j = 1, 2, \ldots, n)$. This has the effect of setting to zero the cost of prefixing y by any prefix of x. In other words, any prefix of the text can be skipped free of charge in an optimum edit script.

Clearly, the computation of the final value of $C(i, j)$ may proceed as in the general case, and it will still take $\Theta(|y||x|) = \Theta(mn)$ time. Note, however, that we are interested now in the entire last row of matrix C at the outset. Although we assumed unit costs, the validity of the method extends clearly to the case of general positive costs.

In practice, it is often more interesting to locate only those segments of x that present a high similarity with y under the adopted measure. Formally, given a pattern y, a text x and an integer k, this restricted version of the problem consists of locating all terminal positions of substrings w of x such that the edit distance between w and y is at most k. The recurrence given above will clearly produce this information. However, there are more efficient methods to deal with this restricted case. In fact, a time complexity $O(kn)$ and even sublinear expected time are achievable. We refer to Landau and Vishkin [36, 37], Sellers [49], Ukkonen [57], Galil and Giancarlo [22], Chang and Lawler [14], for detailed discussions. In the following, we review some basic principles subtending an $O(kn)$ algorithm for string searching with k differences. Note that when k is a constant the corresponding time complexity is linear.

The crux of the method is to limit computation to $O(k)$ elements in each diagonal of the matrix C. These entries will be called **extremal** and may be defined as follows: a diagonal entry is d-extremal if it is the deepest entry on that diagonal to be given value d $(d = 1, 2, \ldots, k)$. Note that a diagonal might not feature any, say, 1-extremal entry, in which case it would correspond to a perfect match of the pattern. The identification of d-extremal entries proceeds from extension of entries already known to be $(d-1)$-extremal. Specifically, assume we knew that entry $C(i, j)$ is $(d-1)$-extremal. Then, any entry reachable from $C(i, j)$ through a unit vertical, horizontal, or diagonal-mismatch step possibly followed by a maximal diagonal stream of matches is d-extremal at worst. In fact, the cost of a diagonal stream of matches is 0, whence the cost of an entry of the type considered cannot exceed d. On the other hand, that cost cannot be smaller than $d-1$, otherwise this would contradict the assumption $C(i, j) = d-1$. Let entries reachable from a $(d-1)$-extremal entry $C(i, j)$ through a unit vertical, horizontal, or diagonal-mismatch step be called d-**adjacent**. Then the following program encapsulates the basic computations.

Algorithm "KERR" :
element array $x[1:n]$, $y[1:m]$, $C[0:m; 0:n]$; integer k
begin
(*PHASE* 1 : initializations)
set first row of C to 0;
find the boundary set S_0 of 0-extremal entries by exact string searching;

```
(PHASE 2 :  identify k-extremal entries )
    for d = 1 to k do
        begin
        walk one step horizontally, vertically and (on mismatch) diagonally
        from each (d − 1)-extremal entry in set S_(d−1) to find d-adjacent entries;
        from each d-adjacent entry, compute the farthest d-valued
        entry reachable diagonally from it;
        end
    for i = 1 to n − m + 1 do
        begin
        select lowest d-entry on diagonal i
        and put it in the set S_d of d-extremal entries
        end
end.
```

It is easy to check that the algorithm performs k iterations in each one of which it does essentially a constant number of manipulations on each of the n diagonals. In turn, each one of these manipulations takes constant time except at the point where we ask to reach the farthest d-valued entry from some other entry on a same diagonal. We would know how to answer quickly that question if we knew how to handle the following query: given two arbitrary positions i and j in the two strings y and x, respectively, find the longest common prefix between the suffix of y that starts at position i and the suffix of x that starts at position j. In particular, our bound would follow if we knew how to process each query in constant time. It is not known how that could be done without preprocessing becoming somewhat heavy. On the other hand, it is possible to have it such that *all* queries have a cumulative amortized cost of $O(kn)$. This possibility rests on efficient algorithms for performing **lowest common ancestor** queries in trees. Space limitations do not allow us to belabor this point any further.

Note that the special case where insertions and deletions are forbidden is also solved by an algorithm very similar to the above and within the same time bound. This variant of the problem is often called **string searching with mismatches**. A probabilistic approach to this problem is implicit in [14], one more is described in [9]. When k cannot be considered a constant, an interesting alternative results from Abrahamson's approach to multiple-value string searching.

Specifically, this algorithm of Abrahamson's combines divide and conquer with the idea of Fischer and Paterson [20] which was discussed earlier. In divide-and-conquer, the problem is first partitioned into subproblems; these are then solved by ad-hoc techniques, and finally the partial solutions are combined. One possible way to "divide" is to take projections of the pattern into two complementary subsets, another is to split and handle separately the positive and negative portions of the pattern. We have already seen that the adaptation of fast multiplication to string searching leads to a time bound $O(n(|\Sigma|)\log^2 m \log\log m)$.

This performance is good for bounded Σ but quite poor when Σ is unbounded. In this latter case, however, some of the symbols must be very unfrequent. Using this observation, Abrahamson designed a projection into $\Sigma' = \{\sigma \in \Sigma : \sigma$ occurs at most z times in $y\}$ and the corresponding complement set Σ''. The rare symbols can be handled efficiently by some direct match-counting, since they cannot produce more than zn matches in total. The frequent ones are limited in number to m/z and we can apply multiplication to each one of them separately. The overall result is time $O(nm/z \log^2 m \log\log m)$, which becomes $O(nm^{1/2} \log m \log\log^{1/2} m)$ if we pick $z = m^{1/2} \log m \log\log^{1/2} m$.

13.5 Two-Dimensional Matching

The problem of matching and searching of two-dimensional objects arises in as many applications as there are ways to involve pictures and other planar representations and objects. Just like the full-fledged problem of recognizing the digitized signal of a spoken word in a speech finds a first rough approximation

in string searching, the problem of recognizing a particular subject in a scene finds a first, simplistic model in the computational task that we consider in this section: locating occurrences of a small array into a larger one. Even at this level of simplification, this task is enough complicated already that we shall ignore such variants as those allowing for different shapes and rotations, variants that do not appear in the one dimensional searches.

Two-dimensional matching may be exact and approximate just like with strings, but edit operations of insertion and deletion denature the structure of an array and thus may be meaningless in most settings. The literature on two-dimensional searching concentrates on exact matching, and so does the treatment of this section.

Searching with Automata

In exact two-dimensional searching, the input consists of a "text" array $X[n \times n]$ and a "pattern" array $Y[m \times m]$. The output consists of all locations (i, j) in X where there is an occurrence of Y, where the word "occurrence" is to be interpreted in the obvious sense that $X_{i+k,j+l} = Y_{k+1,l+1}, 0 \leq k, l \leq m-1$.

The naive attack leads to an $O(n^2m^2)$ solution for the problem. It is not difficult to reduce this down to $O(n^2m)$ by resorting to established string searching tools. This may be seen as follows. Imagine to build a linear pattern y where each character consists of one of the consecutive rows of Y. Now, build similarly the family of textstrings $x_1^{(i)}x_2^{(i)}\ldots x_{n-m+1}^{(i)}$ $(1 \leq i \leq n)$ such that $x_j^{(i)}$ is the character for $X_{i,j}X_{i,j+1}\cdots X_{i,j+m-1}$. Clearly, Y occurs at $X_{i,j}$ iff y occurs at $x_j^{(i)}$. If one could assume a constant cost for comparing a character of y with one of $x^{(i)}$, it would take $O(n)$ time by any of the known fast string searching to find the occurrences on y in each $x^{(i)}$. Hence, it would take optimal time $O(n^2)$ for the n strings in the global problem. Since comparing two strings of m characters each charges in fact m comparisons, then the overall bound becomes $O(n^2m)$, as stated.

Automata-based techniques were developed along these lines by Bird [11] and Baker [10]. Later efforts exposed also a germane problem which came to be called "dictionary matching" and acquired some independent interest. Some details of such an automata-based two-dimensional searching are given next.

The main idea is to build on the distinct rows of the pattern Y the Aho–Corasick [3] automaton for multiple string searching. Once this connection is made, it becomes possible to solve the problem at a cost of preprocessing time $O(m^2 \log|\Sigma|)$ (to build the automaton for at most m patterns with m characters each), and time $O(n^2 \log|\Sigma| + tocc)$ to scan the text. Here $tocc$, stands for total number of occurrences, i.e., is the size of the output. In multiple string matching, the parameter $tocc$ may play havoc with time linearity, since more than one pattern might end and thus have to be outputted at any given position. Here, however, the rows of Y are all of the same size, whence only one such row may occur at any given position.

Periods and Witnesses in Two Dimensions

Automata-based approaches such as those just discussed result in time complexities that carry a dependence to alphabet size. This is caused by the branching of forward transitions that leave the states of the machine in multiple string searching. Single string searching is not affected by this problem. In fact, single string searching found quickly linear solutions without alphabet dependency. In contrast, several years elapse before alphabet dependency was eliminated from two-dimensional searching.

Alphabet dependency was eliminated in steps, first from the search phase only, and finally also from preprocessing. A key factor in the first step of progress was offered by a two-dimensional extension of the notion of a **witness,** a concept first introduced and used by Vishkin [58] in connection with parallel exact string searching. It is certainly rare, and therefore quite remarkable, that a tool devised specifically to speed-up a *parallel* algorithm would find use in designing a better serial algorithm.

It is convenient to illustrate the idea of a witness on strings. Assume then to be given two copies of a pattern y, reciprocally aligned in such a way that the top copy is displaced, say, d positions ahead of the bottom one. A witness for d, if it exists, is any pair of mismatching characters that would prevent the two superimposed copies of y to coexist. Thus, if we were to be given two d-spaced, overlapping candidate occurrences of y on a text x, and a witness were defined for d, then at least one of the candidate occurrences of y in x will necessarily fail. One alternative way to regard a witness at d is as a counterexample to the claim that d is a period for y. The latter is a necessary, though not sufficient condition for having y occur twice, d positions apart.

The use of witnesses during the search phase presupposes preparation of appropriate tables. These tables essentially provide, for each d where this is true, a mismatch proving the incompatibility of two overlapping matches at a distance of d. The notion of a witness generalizes naturally to higher dimensions. In two dimensions two witnesses tables were introduced by Amir, Benson and Farach [5] as follows. Witness $Wit_{i,j}$ is any position (p, q) such that $X_{i+p,j+q}$ does not match $X_{i,j}$ or else it is 0. Note that, given an array W, there are essentially only two ways of superpositions one of thet two copies onto the other. These consist, respectively, of shifting one of the copies toward the right and bottom or toward the right and top of the other. These two families correspond to two witness tables that depend on whether $i < 0$ or $i \geq 0$. Amir, Benson and Farach [5] showed how to build the witness table in time $O(m^2 \log |\Sigma|)$.

Once the table is available, the search phase is performed in two stages that are called respectively **candidate consistency testing** and **candidate verification**. The candidates are the positions of X, interpreted as top-left corners of potential occurrences of the pattern. At the beginning each position is a viable candidate. A pair of candidates is consistent if the pattern could be placed at both places without conflicting with the witness tables. The task of the first phase is to use the witness tables to remove one in each pair of inconsistent candidates. Clearly, one character comparison with the position of the text array that corresponds to the witness suffices to carry out this "duel" between the candidates. Note that a duel might rule out both candidates, however, eliminating one will do.

At the end of the consistency check we can verify the surviving candidates. A same text symbol could belong to several candidates, but all of these candidates must agree on that symbol. Thus, each position in the text can be labeled true or false according to whether or not it complies with what all partecipating candidates surrounding it prescribe for that position. Conversely, whenever a candidate covers a position of the text that is labeled as false, then that candidate can no longer survive. A procedure set up along these lines leads to an $O(n^2)$ search phase, within a model of computation in which character comparisons take constant time and only result in assessing whether the characters are equal or unequal.

The preprocessing in this approach is still dependent on the size of the alphabet. Alphabet independent preprocessing and overall linear time algorithm was achieved by Galil and Park [24]. Like with strings, one may build an index structure based on preprocessing of the text and then run faster queries off-line with varying patterns. Details can be found in, e.g., [25].

13.6 Tree Matching

The discrete structures considered in this section are labeled, rooted trees, with the possible additional constraint that children of each node be ordered. Recall that a **tree** is any undirected, connected and acyclic graph. Choosing one of the vertices as the **root** makes the tree **rooted,** and fixing an order among the children of each node makes the tree **ordered**. Like with other classes of discrete objects, there is exact and approximate searching and matching of trees. We examine both of these issues next.

Exact Tree Searching

In exact tree searching, we are given two ordered trees, namely, a "pattern" tree P with m nodes and a "text" tree T with n nodes, and we are asked to find all occurrences of P in T. An occurrence of P in T

is an ordered subtree P' rooted at some node v of T such that P could be rigidly superimposed onto P' without any label mismatch or edge skip. The second condition means that the kth child of a node of P matches precisely the kth child of a node of T.

An $O(nm^{0.75}\text{polylog}(m))$ improvement over the trivial $O(mn)$ time algorithm was designed by Kosaraju [34]. A faster, $O(n\sqrt{m})polylog(m))$ algorithm, is due to Dubiner, Galil and Magen [18]. Their approach is ultimately reminiscent of Abrahamson's pidgeon-hole approach to generalizations of string searching such as those examined earlier in our discussion. It is based on a combination of periodicity properties in strings and some techniques of tree partitioning that achieve succint representations of long paths in the pattern.

Some notable variants of exact tree pattern matching arise in applications such as code generation and unification for logic programming and term-rewriting systems. In this context, a label can be a constant or a variable, where a variable at a leaf may match an entire subtree. In the most general setting, the input consists of a set S of patterns, rather than a single pattern, and of course of a text T. Early analyses and algorithms for the general problem are due to Hoffman and O'Donnel [29]. Two basic families of treatment descend, respectively, from matching the text tree from the root or from the leaves. The bottom-up approach is the more convenient of the two in the context of term rewriting systems. This approach is heavy on pattern preprocessing, where it may require exponential time and space, although essentially linear in the processing phase. Improvements and special cases are treated by Chase [15], Cai, Paige and Tarjan [13], and Thorup [56].

Tree Editing

The editing problem for unordered trees is NP-complete. However, much faster algorithms can be obtained for ordered trees. Early definitions and algorithms may be traced back to Selkow [50] and Tai [54]. In more recent years, the problem and some of its basic variants have been studied extensively by Shasha and Zhang and their co-authors. The outline given below concentrates on some of their work.

Let T be a tree of $|T| = n$ nodes, each node labeled with a symbol from some alphabet Σ. We consider three edit operations on T, consisting, respectively, of the deletion of a node v from T (followed by the reassignment of all children of v to the node of which v was formerly a child), the insertion of a new node along some consecutive arcs departing from a same node of T, and the substitution of the label of one of the nodes of T with another label from Σ. Like with strings, we assume that each edit operation has an associated nonnegative real number representing the cost of that operation. We similarly extend the notion of edit script on T to be any consistent sequence Γ of edit operations on T, and define the cost of Γ as the sum of all costs of the edit operations in Γ. These notions generalize easily to any ordered **forest** of trees.

Now, let F and F' be two forests of respective sizes $|F| = n$ and $|F'| = m$. The **forest editing problem** for input F and F' consists of finding an edit script Γ' of minimum cost that transforms F into F'. The cost of Γ' is the edit distance from F to F'. When F and F' consist each of exactly one tree, then we speak of the **tree editing problem**.

A convenient way to visualize the editing of trees or forests is by means of a mapping of nodes from one of two structures to the other. The map is represented by a set of links between node pairs (v, v') such that either these two nodes have precisely the same label—and thus node v is exactly conserved as v'—or else the label of v gets substituted with that of v'. Each node takes part in at most one link. The unaffected nodes of F (respectively, F') represent deletions (respectively, insertions). A mapping defined along these lines has the property of preserving both ancestor-descendant and sibling orders. In other words, a link from a descendant of v may only reach a descendant of v', and, similarly, links from two siblings to two others must not cross each other.

Early dynamic programming solutions for tree editing consume $\Theta(|F|^3|F'|^3)$ time. Much faster algorithms have been set up subsequently. Some other interesting problems are special cases of forest editing, including "tree alignment," the "largest common subtree" problem, and the problem of "approximate tree

matching" between a pattern tree and text tree. While any solution to the general tree editing problem implies similar bounds for all these special cases, some of the latter admit a faster treatment.

We review the criterion that subtends the computation of tree edit distances by dynamic programming after Zhang and Shaha [62]. This leads to an algorithm with time bounded by the product of the squares of the sizes of the trees. A convenient preliminary step is to resort to a linear representation for the trees involved. The discussion on mappings suggests that such a representation consist of assigning to each node its ordinal number in the postorder visit of the tree. Let x and y be the strings representing the postorder visits of two trees T and T'. Then a prefix of, say, x will identify in general some forest of subtrees each rooted at some descendant of the root of T. Note that the leftmost leaf in the leftmost tree is denoted precisely x_1. Let i_1 be the corresponding root, and let *forestdist*(i, j) represent the cost of transforming the subforest of T corresponding to $x_1 \ldots x_i$ into the subforest of T' corresponding to $y_1 \ldots y_j$. Let *treedist*(i, j) be the cost of transforming the tree rooted at x_i into the tree rooted at y_j. Then, in the most general case, these costs are dictated by the following recurrence:

$$\textit{forestdist}\ (i, j) = \min \begin{cases} \textit{forestdist}\ (i-1, j) + D\,(x_i) \\ \textit{forestdist}\ (i, j-1) + I\,(y_j) \\ \textit{forestdist}\ (l(i)-1, l(j)-1) + \textit{treedist}\ (i, j) \end{cases}$$

Here $l(i)$ (respectively, $l(j)$) is the index in x (respectively, y) of the leftmost leaf in the subtree rooted at the node x_i (x_j). We leave the initialization conditions for an exercise. Note that *treedist* is little more than a notational convention, since it is a special case of *forestdist*, and thus is computed essentially through the same recursion. In fact, a recursion in the above form can be applied to any pair of substrings of x and y, with obvious meaning. In the special case where both forests consist of a single tree, i.e., x_i and y_j have x_1 and y_1 as their respective leftmost leaves, then *treedist*(i, j) becomes the substitution cost $S(x_i, y_j)$.

Building the algorithm around the above recurrence, and the subtended postorder visits, brings about an important advantage: each time that *treedist* is invoked, the main ingredients for its computation (namely, the pairwaise distances of subtrees thereof) are already in place and thus need not be recomputed from scratch. We illustrate this point using $C(i, j)$, $(0 \le i \le |x|,\ 0 \le j \le |y|)$ as shorthand for *forestdist*. Observe that the recurrence above indicates that the value of $C(i, j)$ depends, in addition to the two neighboring values $C(i-1, j)$ and $C(i, j-1)$, on one generally more distant value $C(i', j')$. The pair (i', j') is called the *conjugate* of pair (i, j). The following facts are easy to check.

FACT 13.1 *Every pair (i, j) has at most one conjugate.*

FACT 13.2 *If (i, j) has conjugate (i', j'), then, for any pair (k, l) with $i' \le k \le i$ and $j' \le l \le j$ we also have $i' \le k' \le i$ and $j' \le l' \le j$.*

Figuratively, Fact 13.1 states that each pair (i, j) of C is associated with exactly one (possibly empty) submatrix of C, with upper-left corner at the conjugate (i', j') of (i, j) (inclusive) and lower right corner at $(i-1, j-1)$ (inclusive). Fact 13.2 states that the submatrices defined by two pairs and their corresponding conjugates are either nested or disjoint.

Like in the case of string editing, the "close" interdependencies among the entries of the C-matrix induce an $(|x|+1) \times (|y|+1)$ "grid directed acyclic graph" (GDAG for short). String editing can be viewed as a shortest-path problem on a GDAG. To take care also of the interdependencies by conjugacy that appear in tree editing, however, the GDAG must be augented by adding to the grid outerplanar edges connecting pairs of conjugate points.

Formally, an $l_1 \times l_2$ augmented GDAG (or AGDAG) is a directed acyclic graph whose vertices are the $l_1 l_2$ points of an $l_1 \times l_2$ grid, and such that the only edges from point (i, j) are to grid points $(i, j+1)$, $(i+1, j)$, $(i+1, j+1)$ and $(i'-1, j'-1)$, where (i, j) is the conjugate of (i', j'). We refer to Fig. 13.3

for an example. We make the convention of drawing the points such that point (i, j) is at the ith row from the top and jth column from the left. The top-left point is $(0, 0)$ and has no edge entering it (i.e., is a "source"), and the bottom-right point is (m, n) and has no edge leaving it (i.e., is a "sink").

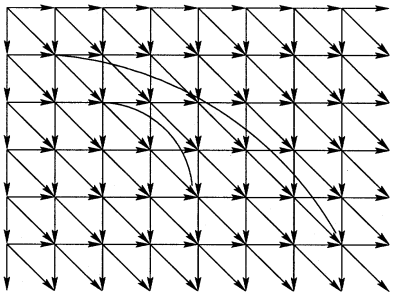


FIGURE 13.3 The upper-left corner of an AGDAG highlights the basic structure of such graphs: a grid with occasional outer-planar edges.

We associate an $(|x| + 1) \times (|y| + 1)$ AGDAG G with the tree editing problem in the natural way: the $(|x| + 1)(|y| + 1)$ vertices of G are in one-to-one correspondence with the $(|x| + 1)(|y| + 1)$ entries of the C-matrix. We draw edges connecting a point to its neighbors in the planar grid of the AGDAG, while the edge that is incident on point $(i - 1, j - 1)$ from the unique conjugate of (i, j), if the latter exists, are drawn outerplanar. Clearly, the cost of a grid edge from vertex (k, l) to vertex (i, j) is equal to $I(y_j)$ if $k = i$ and $l = j - 1$, to $D(x_i)$ if $k = i - 1$ and $l = j$, to $S(x_i, y_j)$ if $k = i - 1$ and $l = j - 1$. The cost of an outerplanar edge is the cost of the optimal solution to the submatrix associated with that edge. Thus, edit scripts that transform x into y or vice versa are in one-to-one correspondence to certain weighted paths of G that originate at the source (which corresponds to $C(0, 0)$) and end on the sink (which corresponds to $C(|x|, |y|)$). Specifically, in any such path horizontal or vertical edges can be traversed unconditionally, but the traversal of a diagonal edge from $(i - 1, j - 1)$ to (i, j) is allowed only if it follows the traversal of the outerplanar edge that is incident upon $(i - 1, j - 1)$ (if it exists). The details are left for an exercise.

13.7 Research Issues and Summary

The focus of this chapter is represented by combinatorial and algorithmic issues of searching and matching with strings and other simple structures like arrays and trees. We have reviewed the basic variants of these problems, with the notable exception of exact string searching. The latter is definitely the primeval problem in the set, and has been devoted so much study to warrant a separate chapter in the present Handbook.

We started by reviewing, in Section 13.2, string searching in the presence of don't care symbols. In Section 13.3, we considered the problem of comparing two strings for similarity, under some basic sets of edit operations. This latter problem subtends the important variants of string searching where the occurrences of the pattern need not be exact; rather, they might be corrupted by a number of mismatches, and possibly by insertions and deletions of symbols as well. We abandoned the realm of one-dimensional pattern matching in Section 13.5, in which we highlighted the comparatively less battered topics of exact searching on two-dimensional arrays. Finally, in Section 13.6, we reviewed exact and approximate searching on rooted trees.

As said at the beginning, most pattern matching issues are still subject to extensive investigation. Meanwhile, new problems and variants continue to arise in application areas that feature, in prominent position, the very information infrastructure under development. In most cases, the goal of current studies is to design better serial algorithms than those previously available. Parallel or distributed versions of the prob-

lems are also investigated. Typically, the solutions of such versions may be expected not to resemble in any significant way their serial predecessors. In fact (as exemplified by the previously encountered notion of a witness) they are more likely to expose novel combinatorial properties, some of which of intrinsic interest. Whether a problem be regarded within a serial, parallel, or distributed computational context, algorithms are also sought that display a good expected, rather than worst-case, performance. Relatively little work has been performed from this perspective, which requires often a thorough reexamination of the problem and may result in a totally new line of attack, as experienced in such classical instances as the Boyer–Moore string searching algorithm and Quicksort.

An exhausitve list of specific open problems of pattern matching would be impossible. Here we limit mention to a few important ones.

For problems of searching with don't care, string editing, longest common subsequence, and variations thereof, there are still wide and little understood gaps between the known, often trivial lower bounds and the efficiency of available algorithms. Likewise, relatively little is known in terms of nontrivial lower bounds for two-dimensional searches with mismatches, and also for both exact and approximate tree matching. Some general problems of fundamental nature remain unexplored across the entire board of pattern structures, problem variations, and computational models. Notable among these is the problem of preprocessing the "text" structure so that "patterns" presented on-line can be searched quickly thereafter. Such an approach has long been known to be elegantly and efficiently viable for exact searching on strings, but remains largely unexplored for approximate searches of all kind of patterns. The latter represent possibly the most recurrent queries in applications of molecular biology, information retrieval and other fields, so that progress in this direction would be valued enormously.

13.8 Defining Terms

Antichain: A subset of mutually incomparable elements in a partially ordered set.

Block: A sequence of don't care symbols.

Candidate consistency testing: The stage of two-dimensional matching where it is checked whether a candidate occurrence of the pattern is checked against the "witness" table.

Candidate verification: The stage of two-dimensional searching where candidate occurrences of the pattern, not ruled out previously as mutually incompatible, are actually tested.

Chain: A linearly ordered subset of a partially ordered set.

D-adjacent: An entry reachable from a $(d-1)$-extremal entry through a unit vertical, horizontal, or diagonal-mismatch step.

Divide and conquer: One of the basic paradigms of problem solving, in which the problem is decomposed (recursively) into smaller parts; solutions are then sought for the subproblems and finally combined in a solution for the whole.

Don't care: A "wildcard" symbol matching any other symbol of a given alphabet.

Edit operation: On a string, the operation of deletion, or insertion, or substitution, performed on a single symbol. On a tree T, the deletion of a node v from T followed by the reassignment of all children of v to the node of which v was formerly a child, or the insertion of a new node along some consecutive arcs departing from a same node of T, or the substitution of the label of one of the nodes of T with another label from Σ. Each edit operation has an associated nonnegative real number representing its cost.

Edit distance: For two given strings, the cost of a cheapest edit script transforming one of the strings into the other.

Edit script: A sequence of viable edit operations on a string.

Exact string searching: The algorithmic problem of finding all occurrences of a given string usually called "the pattern" in another, larger "text" string.

Extremal: Some of the entries of the auxiliary array used to perform string searching. An entry is d-extremal if it is the deepest entry on its diagonal to be given value d.

Forest: A collection of trees.

Forest editing problem: The problem of transforming one of two given forests into the other by an edit script of minimum cost.

Linear product: For two vectors X and Y, and with respect to two suitable operations $\otimes$ and $\oplus$, is a vector $Z = Z_0 Z_1 \ldots Z_{m+n}$ where $Z_k = \bigoplus_{i+j=k} X_i \bigotimes Y_j$ $(k = 0, \ldots, m+n)$.

Local alignment: The detection of local similarities among two or more strings.

Longest (or heaviest) common subsequence problem: The problem of finding a maximum-length (or maximum weight) subsequence for two or more input strings.

Lowest common ancestor: The deepest node in a tree that is an ancestor of two given leaves.

K-dominant match: A match $[i, j]$ having rank k and such that for any other pair $[i', j']$ of rank k either $i' > i$ and $j' \leq j$ or $i' \leq i$ and $j' > j$.

Match: The result of comparing two instances of a same symbol.

Minimal antichain decomposition: A decomposition of a poset into the minimum possible number of antichains.

Offset: The distance from the beginning of a string to the end of a segment in that string.

Pattern element: A positive (negative) pattern element is a "partial wildcard" presented as a subset of the alphabet Σ, with the symbols in the subset specifying which symbols of Σ are matched (mismatched) by the pattern element.

Picture: A collection of mutually disjoint subsets of an alphabet.

Poset: A set the elements of which are subject to a partial order.

Rank: For a given match, this is the number of matches in a longest chain terminating with that match, inclusive.

Segment: The substring of a pattern delimited by two don't cares or one don't care and one pattern boundary.

Sparsity: Used here to refer to LCS problem instances in which the number of matches is small compared to the product of the lengths of the input strings.

String editing problem: For input strings x and y, is the problem of finding an edit script of minimum cost that transforms y into x.

String searching with errors: Searching for approximate (e.g., up to a predefined number of symbol mismatches, insertions, and deletions) occurrences of a pattern string in a text string.

String searching with mismatches: The special case of string matching with errors where mismatches are the only type of error allowed.

Subsequence: Of a string, is any string that can be obtained by deleting zero or more symbols from that string.

Tree: A graph undirected, connected, and acyclic. In a rooted tree, a special node is selected and called the root: the nodes reachable from a node by crossing arcs in the direction away from the root are the children of that node. In unordered rooted trees, there is no pre-set order among the children of a node. Assuming such an order makes the tree ordered.

Tree editing problem: The problem of transforming one of two given trees into the other by an edit script of minimum cost.

Witness: A mismatch of two symbols of string y at a distance of d is a "witness" to the fact that in no subject y could occur twice at a distance of exactly d positions (equivalently, that d cannot be a period of y).

Acknowledgments

This work was supported in part by NSF Grants CCR-9201078 and CCR-9700276, by NATO Grant CRG 900293, by the National Research Council of Italy, by British Engineering and Physical Sciences Research Council Grant GR/L19362. Xuyan Xu contributed to Section 13.2 through bibliographic searching and drafting. The referees carried out a very careful scrutiny of the manuscript and made many helpful comments.

References

[1] Abrahamson, K., Generalized string matching, *SIAM. J. Comput.,* 16(6), 1039–1051, 1987.
[2] Aho, A.V., Algorithms for finding patterns in strings, *Handbook of Theoretical Computer Science,* J. van Leeuwen, Ed., Elsevier, Amsterdam, 255–300, 1990.
[3] Aho, A.V. and Corasick, M.J., Efficient string matching: An aid to bibliographic search, *CACM,* 18(6), 333–340, 1975.
[4] Aho, A.V., Hirschberg, D.S., and Ullman, J.D., Bounds on the complexity of the longest common subsequence problem, *J. Assoc. Comput. Mach.,* 23(1), 1–12, 1976.
[5] Amir, A., Benson, G., and Farach, M., An alphabet independent approach to two dimensional matching, *SIAM J. Comp.,* 23(2), 313–323, 1994.
[6] Apostolico, A., Browne, S. and Guerra, C., Fast linear space computations of longest common subsequences, *Theoretical Computer Science,* 92(1), 3–17, 1992.
[7] Apostolico, A. and Guerra, C., The longest common subsequence problem revisited, *Algorithmica,* 2, 315–336, 1987.
[8] Arlazarov, V.L., Dinic, E.A., Kronrod, M.A., and Faradzev, I.A., On economical construction of the transitive closure of a directed graph, *Dokl. Akad. Nauk SSSR,* 194, 487–488 (in Russian). English translation in *Soviet Math. Dokl.,* 11(5), 1209–1210, 1970.
[9] Atallah, M.J., Jacquet, P., and Szpankowski, W., A probabilistic approach to pattern matching with mismatches *Random Structures and Algorithms,* 4, 191–213, 1993.
[10] Baker, T.P., A technique for extending rapid exact-match string matching to arrays of more than one dimension, *SIAM J. Comp.,* 7(4), 533–541, 1978.
[11] Bird, R.S., Two dimensional pattern matching, *Information Processing Letters,* 6(5), 168–170, 1977.
[12] Bogart, K.P. *Introductory Combinatorics,* Pitman, NY, 1983.
[13] Cai, J., Paige, R., and Tarjan, R., More efficient bottom-up multi-pattern matching in trees, *Theoretical Computer Science,* 106(1), 21–60, 1992.
[14] Chang, W.I. and Lawler, E.L., Approximate string matching in sublinear expected time, in *Proc. 31st Annual IEEE Symp. on Foundations of Vomputer Science,* St. Louis, MO, 116–124, 1990.
[15] Chase, D., An improvement to bottom-up tree pattern matching, *Proceedings of the 14th Annual ACM Symp. on POPL,* 168–177, 1987.
[16] Chazelle, B., A functional approach to data structures and its use in multidimensional searching, *SIAM. J. Comput.,* 17(3), 427–462, 1988.
[17] Dilworth, R.P., A decomposition theorem for partially ordered sets, *Ann. Math.,* 51, 161–165, 1950.
[18] Dubiner, M., Galil, Z., and Magen, E., Faster tree pattern matching, *JACM,* 14(2), 205–213, 1994.

[19] van Emde Boas, P., Preserving order in a forest in less than logarithmic time, *Proc. 16th FOCS,* 75–84, 1975.
[20] Fischer, M.J. and Paterson, M., String matching and other products, *Complexity of Computation, SIAM-AMS Proceedings 7,* Karp, R., Ed., 113–125, 1973.
[21] Fredman, M.L., On computing the length of longest increasing subsequences, *Discrete Mathematics,* 11, 29–35, 1975.
[22] Galil Z. and Giancarlo, R., Data structures and algorithms for approximate string matching, *Jour. Complexity,* 4, 33–72, 1988.
[23] Galil, Z. and Park, K., An improved algorithm for approximate string matching, *SIAM Jour. Computing,* 19(6), 989–999, 1990.
[24] Galil, Z. and Park, K., Truly alphabet-independent two-dimensional pattern matching, *Proc. 33rd Symposium on the Foundations of Computer Science (FOCS 92),* 247–256, 1992.
[25] Giancarlo, R. and Grossi, R., On the construction of classes of suffix trees for square matrices: Algorithms and applications, *22nd Int. Colloquium on Automata, Languages, and Programming,* Z. Fulop and F. Gecseg, Eds., LNCS, 944, 111–122, 1995.
[26] Hirschberg, D.S., Algorithms for the longest common subsequence problem, *JACM,* 24(4), 664–675, 1977.
[27] Hirschberg, D.S., An information theoretic lower bound for the longest common subsequence problem, *Inform. Process. Lett.,* 7(1), 40–41, 1978.
[28] Hirschberg, D.S. and Ullman. 1976.
[29] Hoffman, C. and O'Donnel, J., Pattern matching in trees, *JACM,* 29(1), 68–95, 1982.
[30] Hunt, J.W. and Szymanski, T.G., A fast algorithm for computing longest common subsequences, *CACM,* 20(5), 350–353, 1977.
[31] Jacobson, G. and Vo, K.P., Heaviest increasing/common subsequence problems, *Combinatorial Pattern Matching, Proceedings of the Third Annual Symposium,* A. Apostolico, M. Crochemore, Z. Galil and U. Manber, Eds., Tucson, Arizona, 1992. Springer Verlag Lecture Notes in Computer Science 644, 52–66, 1992.
[32] Jiang, T., Wang, L., and Zhang, K., Alignment of trees—an alternative to tree edit, *Proceedings of the Fifth Symposium on Combinatorial Pattern Matching,* 75–86, 1994.
[33] Knuth, D.E., Morris, J.H., and Pratt, V.R., Fast pattern matching in strings, *SIAM. J. Comput.,* 6(2): 323-350, 1977.
[34] Kosaraju, S.R., Efficient tree pattern matching, *Proceedings of the 30th annual IEEE Symposium on Foundations of Computer Science,* 178–183, 1992.
[35] Kumar, S.K. and Rangan, C.P., A linear space algorithm for the LCS problem, *Acta Informatica,* 24, 353–362, 1987.
[36] Landau, G.M. and Vishkin, U., Introducing efficient parallelism into approximate string matching and a new serial algorithm, in *Proc. 18th Annual ACM STOC,* New York, 1986, 220–230, 1986.
[37] Landau, G.M. and Vishkin, U., Fast string matching with k differences, *Jour. Comp. and System Sci.,* 37, 63–78, 1988.
[38] Levenshtein, V.I., Binary codes capable of correcting deletions, insertions and reversals, *Soviet Phys. Dokl.,* 10, 707–710, 1966.
[39] Manber, U. and Baeza-Yates, R. An algorithm for string matching with a sequence of don't cares, *Inform. Process. Lett.,* 37(3), 133–136, 1991.
[40] Manber, U. and Myers, E.W., Suffix Array: A new method for on-line string searches, in: *Proc. 1st Annual ACM-SIAM Symposium on Discrete Algorithms,* San Francisco, CA, 319–327, 1990.
[41] Masek, W.J. and Paterson, M.S., A faster algorithm computing string edit distances, *J. Comput. System Sci.,* 20(1), 18–31, 1980.
[42] Muthukrishnan, S. and Hariharan, R., On the equivalence between the string matching with don't cares and the convolution, *Information and Computation,* 122(1), 140–148, 1995.

[43] Myers, E.W., An $O(ND)$ difference algorithm and its variations, *Algorithmica,* 1, 251–266, 1986.
[44] Needleman, R.B. and Wunsch, C.D., A general method applicable to the search for similarities in the amino-acid sequence of two proteins, *J. Molecular Bio.,* 48, 443–453, 1983.
[45] Pinter, R., Efficient string matching with don't-care patterns, *Combinatorial Algorithms on Words,* Apostolico, A. and Galil, Z., Eds., Springer Verlag, NATO ASI Series 12, 11–29, 1985.
[46] Sankoff, D. and Kurskal, 1983.
[47] Sankoff, D., Matching sequences under deletion-insertion constraints, *Proc. Nat. Acad. Sci. USA,* 69, 4–6, 1972.
[48] Sankoff, D. and Sellers, P.H., Shortcuts, diversions and maximal chains in partially ordered sets, *Discrete Mathematics,* 4, 287–293, 1973.
[49] Sellers, P.H., The theory and computation of evolutionary distance, *SIAM J. Appl. Math.,* 26, 787–793, 1974.
[50] Selkow, S.M., The tree-to-tree editing problem, *Information Processing Letters,* 6, 184–186, 1977.
[51] Shasha, D., Wang, J.T.L., and Zhang, K., Exact and approximate algorithms for unordered tree matching, *IEEE Trans. Systems, Man, and Cybernetics,* 24(4), 668–678, 1994.
[52] Shasha, D. and Zhang, K., Fast algorithms for the unit cost editing distance between trees, *J. Algorithms,* 11, 581–621, 1990.
[53] Schonhage, A. and Strassen, V., Schnelle Multiplikation grosser Zahlen, *Computing (Arch. Elektron. Rechnen),* 7, 281–292. MR 45 No. 1431, 1971.
[54] Tai, K.C., The tree-to-tree correction problem, *J. ACM,* 26, 422–433, 1979.
[55] Takeda, M., A fast matching algorithm for patterns with pictures, *Bull. Info. Cyber.,* 25(3-4), 137–153, 1993.
[56] Thorup, M., Efficient preprocessing of simple binary pattern forests, *Proceedings of the 4th Scandinavian Workshop on Algorithm Theory, Lecture Notes in Computer Science,* 824, 350–358, 1994.
[57] Ukkonen, E., Finding approximate patterns in strings, *Journal of Algorithms,* 6, 132–137, 1985.
[58] Vishkin, U. Optimal parallel pattern matching in strings, *Information and Control,* 67(1-3), 91–113, 1985.
[59] Wagner, R.A. and Fischer, M.J., The string to string correction problem, *J. Assoc. Comput. Mach.,* 21, 168–173, 1974.
[60] Willard, D.E., On the application of sheared retrieval to orthogonal range queries, in: *Proc. 2nd Annual ACM Symposium on Computational Geometry,* Yorktown Heights, NY, 80–89, 1986.
[61] Wong, C.K. and Chandra, A.K., Bounds for the string editing problem, *J. Assoc. Comput. Mach.,* 23(1), 13–16, 1976.
[62] Zhang, K. and Shasha, D., Simple fast algorithms for the editing distance between trees and related problems, *SIAM J. Computing,* 18(6), 1245–1262, 1989.

Further Information

Most books on design and analysis of algorithms devote one or more chapters to pattern matching. Here, we limit mention to specialized sources.

The collection of essays *Combinatorics on Words,* published in 1982 by Addison Wesley under a fictitious editorship (M. Lothaire) contains most of the basic properties used in string searching, and more. An early attempt at unified coverage of string algorithmics is found in *Combinatorial Algorithms on Words,* edited by A. Apostolico and Z. Galil in 1985 for Springer-Verlag. *Time Warps, String Edits and Macromolecules: The Theory and Practice of Sequence Comparison,* edited by D. Sankoff and J. B. Kruskal (Addison-Wesley, 1983), represents still a valuable source for sequence analysis and comparison tools in computational biology and other areas. A few more volumes of recent years are, in order of appearance: *Text Algorithms*

by M. Crochemore and W. Rytter (Oxford University Press, 1994), *String Searching Algorithms* by G.A. Stephen (World Scientific, 1994), *Pattern Matching Algorithms,* edited by A. Apostolico and Z. Galil (Oxford University Press, 1997), and *Algorithms on Strings, Trees and Sequences* by D. Gusfield (Cambridge University Press, 1997). This last volume puts particular emphasis on issues arising in computational biology. A broader treatment of this field can be found in *Introduction to Computational Biology* by M.S. Waterman (Chapman & Hall, 1995). *Data Compression, Methods and Theory* by J.A. Storer (Computer Science Press, 1988) describes applications of pattern matching to the important family of compression methods by "textual substitution."

A rich bibliography on "words, automata and algorithms" is maintained by I. Simon of the University of Saõ Paulo (Brazil). One on "sequence analysis and comparison" is maintained by William H. E. Day in Port Maitland, Canada. A collection of "pattern matching pointers" is currently maintained by S. Lonardi at http://www.cs.purdue.edu/homes/stelo/pattern.html.

Papers on the subject of pattern matching appear primarily in archival journals of theoretical computer science, but important contributions are also found in journals of application areas such as computational biology (notably, *CABIOS* and *Journal of Computational Biology*) and various specialties of computer science (cf., e.g., *IEEE Transactions* on Information Theory, Pattern Recognition, Machine Intelligence, Software, etc.). Special issues have been dedicated to pattern matching by *Algorithmica* and *Theoretical Computer Science.* Papers on the subject are presented at most major conferences. The *International Symposia on Combinatorial Pattern Matching* have gathered yearly since 1990. Beginning in 1992, proceedings have been published in the Lecture Notes in Computer Science Series of Springer-Verlag (serial numbers of volumes already published: 644, 684, 807, 937, 1075, 1264). Specifically flavored contributions appear also at conferences such as *RECOMB* (International Conference on Computational Molecular Biology), the *IEEE Annual Data Compression Conference,* the *South American Workshop on String Processing,* and others.

14

Average Case Analysis of Algorithms[1]

Wojciech Szpankowski
Purdue University

14.1 Introduction

An *algorithm* is a finite set of instructions for a treatment of *data* to meet some desired objectives. The most obvious reason for *analyzing* algorithms and data structures is to discover their characteristics in order to evaluate their suitability for various applications, or to compare them with other algorithms for the same application. Needless to say, we are interested in *efficient* algorithms in order to use efficiently such scarce resources as computer space and time.

Most often algorithm designs aim to optimize the asymptotic *worst case* performance, as popularized by Aho et al. [2]. Insightful, elegant, and generally useful constructions have been set up in this endeavor.

[1]This research was partially supported by NSF Grants NCR-9206315, 9415491, and CCR-9804760, and NATO Collaborative Grant CRG.950060.

Along these lines, however, the design of an algorithm is usually targeted at coping efficiently sometimes with unrealistic, even pathological inputs and the possibility is neglected that a simpler algorithm that works fast "on average" might perform just as well, or even better in practice. This alternative solution, called also a probabilistic approach, became an important issue two decades ago when it became clear that the prospects for showing the existence of polynomial time algorithms for NP-hard problems, were very dim. This fact, and the apparently high success rate of heuristic approaches to solving certain difficult problems, led Richard Karp [34] to undertake a more serious investigation of probabilistic analysis of algorithms. (But, one must realize that there are problems which are also hard "on average" as shown by Levin [42].) In the last decade we have witnessed an increasing interest in the probabilistic (also called *average case*) analysis of algorithms, possibly due to the high success rate of randomized algorithms for computational geometry, scientific visualization, molecular biology, etc. (e.g., see [46, 60]). Finally, we should point out that probabilistic analysis often depends on the input distribution which is usually unknown up front, and this might lead to unrealistic assumptions.

The *average case* analysis of algorithms can be roughly divided into two categories, namely: *analytic* in which complex analysis plays a pivotal role, and *probabilistic* in which probabilistic and combinatorial techniques dominate. The former was popularized by Knuth's monumental three volumes *The Art of Computer Programming* [39, 40, 41], whose prime goal was to accurately predict the performance characteristics of an algorithm. Such an analysis often sheds light on properties of computer programs and provides useful insights of combinatorial behaviors of such programs. Probabilistic methods were introduced by Erdös and Rényi and popularized by Alon and Spencer in their book [3]. In general, nicely structured problems are amiable to an analytic approach that usually gives much more precise information about the algorithm under consideration. On the other hand, structurally complex problems are more likely to be first solved by a probabilistic tool that later could be further enhanced by a more precise analytical approach. The average case analysis of algorithms, as a discipline, uses a number of branches of mathematics: combinatorics, probability theory, graph theory, real and complex analysis, and occasionally algebra, geometry, number theory, operations research, and so forth.

In this chapter, we choose one facet of the theory of algorithms, namely that of algorithms and data structures on words (strings) and present a brief exposition on certain analytic and probabilistic methods that have become popular. Our choice of the area stems from the fact that there has been a resurgence of interest in *string algorithms* due to several novel applications, most notably in computational molecular biology and data compression. Our choice of methods covered here is aimed at closing a gap between analytic and probabilistic methods. There are excellent books on analytic methods (cf. Knuth's three volumes [39, 40, 41], Sedgewick and Flajolet [49]) and probabilistic methods (cf. Alon and Spencer [3], Coffman and Lueker [10], and Motwani and Raghavan [46]), however, remarkably very few books have been dedicated to both analytic and probabilistic analysis of algorithms (with possible exceptions of Hofri [28] and Mahmoud [44]). Finally, before we launch our journey through probabilistic and analytic methods, we should add that in recent years several useful surveys on analysis of algorithms have been published. We mentioned here: Frieze and McDiarmid [22], Karp [35], Vitter and Flajolet [59], and Flajolet [16].

This chapter is organized as follows: In the next section we describe some algorithms and data structures on words (e.g., digital trees, suffix trees, edit distance, Lempel–Ziv data compression algorithm, etc.) that we use throughout to illustrate our ideas and methods of analysis. Then, we present probabilistic models for algorithms and data structures on words together with a short review from probability and complex analysis. Section 14.4 is devoted to probabilistic methods and discusses the sieve method, first and second moment methods, subadditive ergodic theorem, techniques of information theory (e.g., entropy and its applications), and large deviations (i.e., Chernoff's bound) and Azuma's type inequality. Finally, in the last section we concentrate on analytic techniques in which complex analysis plays a pivotal role. We shall discuss analytic techniques for recurrences and asymptotics (i.e., Rice's formula, singularity analysis, etc.), Mellin transform and its applications, and poissonization and depoissonization. In the future we plan to expand this chapter to a book.

14.2 Data Structures and Algorithms on Words

As mentioned above, in this survey we choose one facet of the theory of algorithms, namely that of data structures and algorithms on words (strings) to illustrate several probabilistic and analytic techniques of the analysis of algorithms. In this section, we briefly recall certain data structures and algorithms on words that we use throughout this chapter.

Algorithms on words have experienced a new wave of interest due to a number of novel applications in computer science, telecommunications, and biology. Undoubtly, the most popular data structures in algorithms on words are digital trees [41, 44] (e.g., tries, PATRICIA, digital search trees), and in particular suffix trees [2, 12, 54]. We discuss them briefly below, together with general *edit distance* problem [4, 9, 12, 60], and the *shortest common superstring* [7, 23, 58] problem, which recently became quite popular due to possible application to the DNA sequencing problem.

Digital Trees

We start our discussion with a brief review of **digital trees**. The most basic digital tree known as a **trie** (the name comes from re*trie*val) is defined first, and then other digital trees are described in terms of the trie.

The primary purpose of a trie is to store a set S of strings (words, keys), say $S = \{X_1, \ldots, X_n\}$. Each word $X = x_1x_2x_3\ldots$ is a finite or infinite string of symbols taken from a finite alphabet $\Sigma = \{\omega_1, \ldots, \omega_V\}$ of size $V = |\Sigma|$. A string will be stored in a leaf of the trie. The trie over S is built recursively as follows: For $|S| = 0$, the trie is, of course, empty. For $|S| = 1$, $trie(S)$ is a single node. If $|S| > 1$, S is split into V subsets $S_1, S_2, \ldots, S_V$ so that a string is in S_j if its first symbol is ω_j. The tries $trie(S_1), trie(S_2), \ldots, trie(S_V)$ are constructed in the same way except that at the kth step, the splitting of sets is based on the kth symbol. They are then connected from their respective roots to a single node to create $trie(S)$. Figure 14.1 illustrates such a construction (cf. [2, 41, 44]).

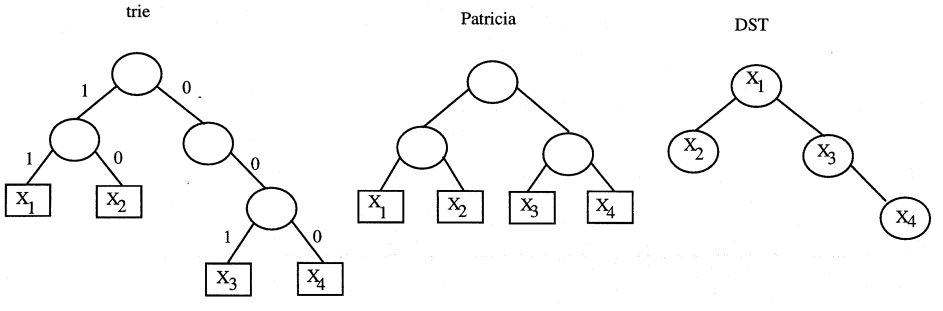


FIGURE 14.1 A trie, Patricia trie, and a digital search tree (DST) built from the following four strings $X_1 = 11100\ldots$, $X_2 = 10111\ldots$, $X_3 = 00110\ldots$, and $X_4 = 00001\ldots$.

There are many possible variations of the trie. The **PATRICIA trie** eliminates the waste of space caused by nodes having only one branch. This is done by collapsing one-way branches into a single node. In a **digital search tree** keys (strings) are directly stored in nodes, and hence, external nodes are eliminated. The branching policy is the same as in tries. Figure 14.1 illustrates these definitions (cf. [41, 44]).

The **suffix tree** and the *compact suffix tree* are similar to the trie and PATRICIA trie, but differ in the structure of the words that are being stored. In suffix trees and compact suffix trees, the words are suffixes of a given string X; that is, the word $X_j = x_jx_{j+1}x_{j+2}\ldots$ is the suffix of X which begins at the jth position of X (cf. [2]).

Certain characteristics of tries and suffix trees are of primary importance. Hereafter, we assume that a digital tree is built from n strings or a suffix tree is constructed from a string of length n. The *m-depth* $D_n(m)$ of the mth leaf in a trie is the number of internal nodes on the path from the root to the leaf. The (typical) depth of the trie D_n then, is the average depth over all its leaves, that is, $\Pr\{D_n \leq k\} = \frac{1}{n}\Sigma_{m=1}^{n}\Pr\{D_n(m) \leq k\}$. The *path length* L_n is the sum of all depths, that is, $L_n = \Sigma_{m=1}^{n} D_n(m)$. The *height* H_n of the trie is the maximum depth of a leaf in the trie and can also be defined as the length of the longest path from the root to a leaf, that is, $H_n = \max_{1\leq m\leq n}\{D_n(m)\}$. These characteristics are very useful in determining the expected size and shape of the data structures involved in algorithms on words. We study some of them in this chapter.

String Editing Problem

The string editing problem arises in many applications, notably in text editing, speech recognition, machine vision and, last but not least, molecular sequence comparison (cf. [60]). Algorithmic aspects of this problem have been studied rather extensively in the past. In fact, many important problems on words are special cases of string editing, including the *longest common subsequence* problem (cf. [9, 12]) and the problem of *approximate pattern matching* (cf. [12]). In the following, we review the string editing problem and its relationship to the longest path problem in a special grid graph.

Let Y be a string consisting of ℓ symbols on some alphabet Σ of size V. There are three operations that can be performed on a string, namely *deletion* of a symbol, *insertion* of a symbol, and *substitution* of one symbol for another symbol in Σ. With each operation is associated a *weight* function. We denote by $W_I(y_i)$, $W_D(y_i)$, and $W_Q(x_i, y_j)$ the weight of insertion and deletion of the symbol $y_i \in \Sigma$, and substitution of x_i by $y_j \in \Sigma$, respectively. An *edit script* on Y is any sequence of edit operations, and the total weight of it is the sum of weights of the edit operations.

The **string editing** problem deals with two strings, say Y of length ℓ (for ℓong) and X of length s (for $s$hort), and consists of finding an edit script of minimum (maximum) total weight that transforms X into Y. The maximum (minimum) weight is called the *edit distance from X to Y*, and its is also known as the Levenshtein distance. In molecular biology, the Levenshtein distance is used to measure similarity (homogeneity) of two molecular sequences, say DNA sequences (cf. [60]).

The string edit problem can be solved by the standard dynamic programming method. Let $C_{\max}(i, j)$ denote the maximum weight of transforming the prefix of Y of size i into the prefix of X of size j. Then

$$\begin{aligned} C_{\max}(i, j) \quad = \quad & \max\left\{C_{\max}(i-1, j-1) + W_Q\left(x_i, y_j\right), C_{\max}(i-1, j)\right. \\ & \left. + W_D\left(x_i\right), C_{\max}(i, j-1) + W_I\left(y_j\right)\right\} \end{aligned}$$

for all $1 \leq i \leq \ell$ and $1 \leq j \leq s$. We compute $C_{\max}(i, j)$ row by row to obtain finally the total cost $C_{\max} = C_{\max}(\ell, s)$ of the maximum edit script.

The key observation for us is to note that interdependency among the partial optimal weights $C_{\max}(i, j)$ induce an $\ell \times s$ grid-like directed acyclic graph, called further a *grid graph.* In such a graph vertices are points in the grid and edges go only from (i, j) point to neighboring points, namely $(i, j+1)$, $(i+1, j)$ and $(i+1, j+1)$. A horizontal edge from $(i-1, j)$ to (i, j) carries the weight $W_I(y_j)$; a vertical edge from $(i, j-1)$ to (i, j) has weight $W_D(x_i)$; and a diagonal edge from $(i-1, j-1)$ to (i, j) has weight $W_Q(x_i, y_j)$. Figure 14.2 shows an example of such an edit graph. The edit distance is the longest (shortest) path from the point $O = (0, 0)$ to $E = (\ell, s)$.

Shortest Common Superstring

Various versions of the **shortest common superstring** (in short: SCS) problem play important roles in data compression and DNA sequencing. In fact, in laboratories DNA sequencing (cf. [60]) is routinely done by sequencing large numbers of relatively short fragments, and then heuristically finding a short common

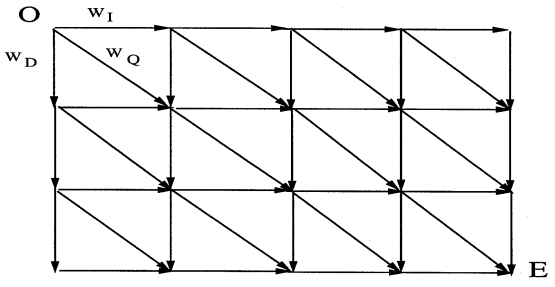


FIGURE 14.2 Example of a grid graph of size $\ell = 4$ and $s = 3$.

superstring. The problem can be formulated as follows: given a collection of strings, say $X_1, X_2, \ldots, X_n$ over an alphabet Σ, find the shortest string Z such that each of X_i appears as a substring (a consecutive block) of Z.

It is known that computing the shortest common superstring is NP-hard. Thus, constructing a good approximation to SCS is of prime interest. It has been shown recently, that a greedy algorithm can compute in $O(n \log n)$ time a superstring that in the worst case is only β times (where $2 \leq \beta \leq 4$) longer than the shortest common superstring [7, 58]. Often, one is interested in maximizing total overlap of SCS using a greedy heuristic and to show that such a heuristic produces an overlap O_n^{gr} that approximates well the optimal overlap O_n^{opt} where n is the number of strings.

More precisely, suppose $X = x_1 x_2 \ldots x_r$ and $Y = y_1 y_2 \ldots y_s$ are strings over the same finite alphabet Σ. We also write $|X|$ for the length of X. We define their *overlap* $o(X, Y)$ by

$$o(X, Y) = \max \left\{ j : \; y_i = x_{r-i+1}, \; 1 \leq i \leq j \right\} .$$

Let $\mathcal{S}$ be a set of all superstrings built over the strings $X_1, \ldots, X_n$. Then

$$O_n^{\text{opt}} = \sum_{i=1}^{n} |X_i| - \min_{Z \in \mathcal{S}} |Z|$$

represents the optimal overlap over $\mathcal{S}$.

14.3 Probabilistic Models

In this section, we first discuss a few probabilistic models of randomly generated strings. Then, we briefly review some basic facts from probability theory (e.g., types of stochastic convergence), and finally we provide some elements of complex analysis that we shall use in this chapter.

Probabilistic Models of Strings

As expected, random shape of data structures on words depends on the underlying probabilistic assumptions concerning the strings involved. Below, we discuss a few basic probabilistic models that one often encounters in the analysis of problems on words.

We start with the most elementary model, namely the **Bernoulli model** that is defined as follows:

(B) BERNOULLI MODEL
Symbols of the alphabet $\Sigma = \{\omega_1, \ldots, \omega_V\}$ occur independently of one another; and $\mathsf{Pr}\{x_j = \omega_i\} = p_i$ with $\Sigma_{i=1}^{V} p_i = 1$. If $p_1 = p_2 = \ldots = p_V = 1/V$, then the model is called *symmetric;* otherwise, it is *asymmetric.* Throughout the paper we only consider *binary alphabet* $\Sigma = \{0, 1\}$ with $p := p_1$ and $q := p_2 = 1 - p$.

In many cases, assumption (B) is not very realistic. For instance, if the strings are words from the English language, then there certainly is a dependence among the symbols of the alphabet. As an example, h is much more likely to follow an s than a b. When this is the case, assumption (B) can be replaced by

(M) MARKOVIAN MODEL
There is a Markovian dependency between consecutive symbols in a key; that is, the probability $p_{ij} = \Pr\{X_{k+1} = \omega_j | X_k = \omega_i\}$ describes the conditional probability of sampling symbol ω_j immediately after symbol ω_i.

There is another generalization of the Markovian model, namely the *mixing model,* which is very useful in practice, especially when dealing with problems of data compression or molecular biology when one expects long dependency among symbols of a string.

(MX) MIXING MODEL
Let $\mathcal{F}_m^n$ be a σ-field generated by $\{X_k\}_{k=m}^n$ for $m \leq n$. There exists a function $\alpha(\cdot)$ of g such that: (i) $\lim_{g\to\infty} \alpha(g) = 0$, (ii) $\alpha(1) < 1$, and (iii) for any m, and two events $A \in \mathcal{F}_{-\infty}^m$ and $B \in \mathcal{F}_{m+g}^{\infty}$ the following holds;

$$(1 - \alpha(g))\Pr\{A\}\Pr\{B\} \leq \Pr\{AB\} \leq (1 + \alpha(g))\Pr\{A\}\Pr\{B\} .$$

In words, model (MX) says that the dependency between $\{X_k\}_{k=1}^m$ and $\{X_k\}_{k=m+g}^{\infty}$ is getting weaker and weaker as g becomes larger (note that when the sequence $\{X_k\}$ is i.i.d., then $\Pr\{AB\} = \Pr\{A\}\Pr\{B\}$). The "quantity" of dependency is characterized by $\alpha(g)$ (cf. [8]).

Quick Review from Probability: Types of Stochastic Convergence

We begin with some elementary definitions from probability theory. The reader is referred to [14, 15] for more detailed discussions. Let the random variable X_n denote the value of a parameter of interest depending on n (e.g., depth in a suffix tree and/or trie built over n strings). The expected value $\mathbf{E}[X_n]$ or mean and the variance $\mathbf{Var}[X_n]$ can be computed as $\mathbf{E}[X_n] = \Sigma_{k=0}^{\infty} k\Pr\{X_n = k\}$ and $\mathbf{Var}[X_n] = \Sigma_{k=0}^{\infty}(k - \mathbf{E}[X_n])^2\Pr\{X_n = k\}$.

Convergence of Random Variables

It is important to note the different ways in which random variables are said to converge. To examine the different methods of convergence, let X_n be a sequence of random variables, and let their distribution functions be $F_n(x)$, respectively.

The first notion of convergence of a sequence of random variables is known as **convergence in probability**. The sequence X_n converges to a random variable X *in probability,* denoted $X_n \to X$ (pr.) or $X_n \xrightarrow{p} X$, if for any $\epsilon > 0$,

$$\lim_{n\to\infty} \Pr\{|X_n - X| < \epsilon\} = 1 .$$

Note that this does not say that the difference between X_n and X becomes very small. What converges here is the *probability* that the difference between X_n and X becomes very small. It is, therefore, possible, although unlikely, for X_n and X to differ by a significant amount and for such differences to occur infinitely often.

A stronger kind of convergence that does not allow such behavior is called **almost sure convergence** or **strong convergence**. A sequence of random variables X_n converges to a random variable X *almost surely,* denoted $X_n \to X$ (a.s.) or $X_n \xrightarrow{(a.s.)} X$, if for any $\epsilon > 0$,

$$\lim_{N\to\infty} \Pr\left\{\sup_{n\geq N} |X_n - X| < \epsilon\right\} = 1 .$$

From this formulation of almost sure convergence, it is clear that if $X_n \to X$ (a.s.), the probability of infinitely many large differences between X_n and X is zero. The sequence X_n in this case is said to satisfy the strong law of large numbers. As the term strong implies, almost sure convergence implies convergence in probability.

A simple criterion for almost sure convergence can be inferred from the *Borel–Cantelli lemma.* We give it in the following corollary.

LEMMA 14.1 **(Borel–Cantelli)** Let $\epsilon > 0$. If $\Sigma_{n=0}^{\infty} \mathsf{Pr}\{|X_n - X| > \epsilon\} < \infty$, then $X_n \to X$ (a.s.).

PROOF It follows directly from the following chain of inequalities (the reader is referred to "Sieve Methods and Its Variations" for more explanations on these inequalities):

$$\mathsf{Pr}\left\{\sup_{n\geq N} |X_n - X| \geq \epsilon\right\} = \mathsf{Pr}\left\{\bigcup_{n\geq N} (|X_n - X| \geq \epsilon)\right\} \leq \sum_{n\geq N} \mathsf{Pr}\{|X_n - X| \geq \epsilon\} \to 0 .$$

The last convergence is a consequence of our assumption that $\Sigma_{n=0}^{\infty} \mathsf{Pr}\{|X_n - X| > \epsilon\} < \infty$.

A third type of convergence is defined on the distribution functions $F_n(x)$. The sequence of random variables X_n **converges in distribution** or **converges in law** to the random variable X, denoted $X_n \xrightarrow{d} X$, if $\lim_{n\to\infty} F_n(x) = F(x)$ for each point of continuity of $F(x)$. Almost sure convergence implies convergence in distribution.

Finally, the convergence *in mean of order p* implies that $\mathbf{E}[|X_n - X|^p] \to 0$ as $n \to \infty$, and convergence *in moments* requires $\mathbf{E}[X_n^p] \to \mathbf{E}[X^p]$ for any p as $n \to \infty$. It is well known that almost sure convergence and convergence in mean imply the convergence in probability. On the other hand, the convergence in probability leads to the convergence in distribution. If the limiting random variable X is a constant, then the convergence in distribution also implies the convergence in probability (cf. [14]).

Generating Functions

The distribution of a random variable can also be described using generating functions. The *ordinary generating function* $G_n(u)$, and a bivariate *exponential generating function* $g(z, u)$ are defined as

$$G_n(u) = \mathbf{E}\left[u^{X_n}\right] = \sum_{k=0}^{\infty} \mathsf{Pr}\{X_n = k\}\, u^k$$

and $g(z, u) = \Sigma_{n=0}^{\infty} G_n(u) \frac{z^n}{n!}$, respectively. These functions are well-defined for any complex numbers z and u such that $|u| < 1$. Observe that

$$\begin{aligned} \mathbf{E}[X_n] &= G_n'(1) , \\ \mathbf{Var}[X_n] &= G_n''(1) + G_n'(1) - \left[G_n'(1)\right]^2 . \end{aligned}$$

Levy's Continuity Theorem

Our next step is to relate convergence in distribution to convergence of generating functions. The following results, known as *Levy's continuity theorem* is an archi-fact for most distributional analysis. For our purpose we formulate it in terms of the Laplace transform of X_n, namely $G_n(e^{-t}) = \mathbf{E}[e^{-tX_n}]$ for real t (cf. [14]).

THEOREM 14.1 **(Continuity Theorem)** *Let X_n and X be random variables with Laplace transforms*

$G_n(e^{-t})$ and $G(e^{-t})$, respectively. A necessary and sufficient condition for $X_n \xrightarrow{d} X$ is that $G_n(e^{-t}) \to G(e^{-t})$ for all $t \geq 0$.

The above theorem holds if we set $t = i\nu$ for $-\infty < \nu < \infty$ (i.e., we consider characteristic functions). Moreover, if the above holds for t complex number, then we automatically derive convergence in moments due to the fact that an analytical function possesses all its derivatives.

Finally, in order to establish central limit theorem (i.e., convergence to a normal distribution) a theorem by Goncharov (cf. [39], Chap 1.2.10, Ex. 13) is useful (it follows directly from the Continuity Theorem). This theorem states that a sequence of random variables X_n with mean $\mathbf{E}[X_n] = \mu_n$ and standard deviation $\sigma_n = \sqrt{\mathbf{Var}[X_n]}$ approaches a normal distribution if the following holds:

$$\lim_{n\to\infty} e^{-\tau\mu_n/\sigma_n} G_n\left(e^{\tau/\sigma_n}\right) = e^{\tau^2/2}$$

for all $\tau = i\nu$ and $-\infty < \nu < \infty$, and X_n converges in moments if τ is a complex number.

Review from Complex Analysis

Much of the necessary complex analysis involves the use of Cauchy's integral formula and Cauchy's residue theorem. We briefly recall a few facts from analytic functions, and then discuss the above two theorems. For precise definitions and formulations the reader is referred to [27]. We shall follow here Flajolet and Sedgewick [21].

A function $f(z)$ of complex variable z is analytic at point $z = a$ if it is differentiable in a neighborhood of $z = a$ or equivalently it has a convergent series representation around $z = a$. Let us concentrate our discussion only on *meromorphic functions* that are analytical with an exception of a finite number of points called poles. More formally, a meromorphic function $f(z)$ can be represented in a neighborhood of $z = a$ with $z \neq a$ by Laurent series as follows: $f(z) = \Sigma_{n\geq -M} f_n (z-a)^n$ for some integer M. If the above holds with $f_{-M} \neq 0$, then it is said that $f(z)$ has a *pole* of order M at $z = a$. **Cauchy's Integral Theorem** states that for any analytical function $f(z)$,

$$f_n := \left[z^n\right] f(z) = \frac{1}{2\pi i} \oint f(z) \frac{dz}{z^{n+1}} ,$$

and the circle is traversed counterclockwise, where throughout the chapter we write $[z^n]f(z)$ for the coefficient of $f(z)$ at z^n.

An important tool frequently used in the analytic analysis of algorithms is *residue theory.* The residue of $f(z)$ at a point a is the coefficient of $(z-a)^{-1}$ in the expansion of $f(z)$ around a, and it is denoted as $\mathrm{Res}[f(z);\ z = a] = f_{-1}$. There are many simple rules to evaluate residues and the reader can find them in any standard book on complex analysis (e.g., [27]). Actually, the easiest way to compute a residue of a function is to use the `series` commend in MAPLE that produces a series development of a function. The residue is simply the coefficient at $(z-a)^{-1}$. For example, the following session of MAPLE computes series of $f(z) = \Gamma(z)/(1-2^z)$ at $z = 0$ where $\Gamma(z)$ is the Euler gamma function [1]:

```
series(GAMMA(z)/(1-2^z), z=0, 4);
```

$$-\frac{1}{\ln(2)} z^{-2} - \frac{-\gamma - \frac{1}{2}\ln(2)}{\ln(2)} z^{-1} - \frac{-\frac{1}{6}\ln(2)^2 + \frac{1}{12}\pi^2 + \frac{1}{2}\gamma^2 + \frac{1}{4}(2\gamma + \ln(2))\ln(2)}{\ln(2)} + \mathrm{O}(z)$$

From the above we see that $\text{Res}[f(z);\ z=0] = \frac{\gamma}{\log 2} + \frac{1}{2}$.

Residues are very important in evaluating contour integrals. In fact, a well-known theorem in complex analysis, that is, **Cauchy's residue theorem** states that if $f(z)$ is analytic within and on the boundary of C except at a finite number of poles $a_1, a_2, \ldots, a_N$ inside of C having residues $\text{Res}[f(z);\ z = a_1], \ldots, \text{Res}[f(z);\ z = a_N]$, then

$$\oint f(z)dz = 2\pi i \sum_{j=1}^{N} \text{Res}\left[f(z);\ z = a_j\right] ,$$

where the circle is traversed counterclockwise.

14.4 Probabilistic Techniques

In this section we discuss several probabilistic techniques that have been successfully applied to the average case analysis of algorithms. We start with some elementary **inclusion–exclusion** principle known also as **sieve methods**. Then, we present very useful **first** and **second moment methods**. We continue with the **subadditive ergodic theorem** that is quite popular for deducing certain properties of problems on words. Next, we turn our attention to some probabilistic methods of information theory, and in particular we discuss **entropy** and **asymptotic equipartition property**. Finally, we look at some **large deviations** results and **Azuma's type inequality**. In this section, as well in the next one where analytic techniques are discussed, we adopt the following scheme of presentation: First, we describe the method and give a short intuitive derivation. Then, we illustrate it on some nontrivial examples taken from the problems on words discussed in Section 14.2.

Sieve Method and Its Variations

The *inclusion–exclusion principle* is one of the oldest tools in combinatorics, number theory (where this principle is known as *sieve method*), discrete mathematics, and probabilistic analysis. It provides a tool to estimate probability of a union of *not* disjoint events, say $\bigcup_{i=1}^{n} A_i$ where A_i are events for $i = 1, \ldots, n$. Before we plunge into our discussion, let us first show a few examples of problems on words for which an estimation of the probability of a union of events is required.

EXAMPLE 14.1: Depth and Height in a Trie

In "Digital Trees" we discussed tries built over n binary strings $X_1, \ldots, X_n$. We assume that those strings are generated according to the Bernoulli model with one symbol, say "0," occurring with probability p and the other, say "1," with probability $q = 1 - p$. Let C_{ij}, known as *alignment* between ith and jth strings, be defined as the length of the longest string that is a prefix of X_i and X_j. Then, it is easy to see that the mth depth $D_n(m)$ (i.e., length of a path in trie from the root to the external node containing X_m), and the height H_n (i.e., the length of the longest path in a trie) can be expressed as follows:

$$D_n(m) = \max_{1 \le i \ne m \le n} \left\{C_{i,m}\right\} + 1 , \tag{14.1}$$

$$H_n = \max_{1 \le i < j \le n} \left\{C_{ij}\right\} + 1 . \tag{14.2}$$

Certainly, the alignments C_{ij} are dependent random variables even for the Bernoulli model. The above equations expressed the depth and the height as an *order statistic* (i.e., maximum of the sequence $C_{i,j}$ for $i, j = 1, \ldots, n$). We can estimate some probabilities associated with the depth and the height as a union

of properly defined events. Indeed, let $A_{ij} = \{C_{ij} > k\}$ for some k. Then, one finds

$$\mathsf{Pr}\{D_n(m) > k\} = \mathsf{Pr}\left\{\bigcup_{i=1,\neq m}^{n} A_{i,m}\right\}, \tag{14.3}$$

$$\mathsf{Pr}\{H_n > k\} = \mathsf{Pr}\left\{\bigcup_{i\neq j=1}^{n} A_{i,j}\right\}. \tag{14.4}$$

In passing, we should point out that for the shortest common superstring problem (cf. "Shortest Common Superstring") we need to estimate a quantity $M_n(m)$ which is similar to $D_n(m)$ except that C_{im} is defined as the length of the longest string that is a prefix of X_i and suffix of X_m for fixed m. One easily observes that $M_n(m) \stackrel{d}{=} D_n(m)$, that is, these two quantities are equal *in distribution.*

We have just seen that often we need to estimate a probability of union of events. The following formula is known as *inclusion–exclusion formula* (cf. [6])

$$\mathsf{Pr}\left\{\bigcup_{i=1}^{n} A_i\right\} = \sum_{r=1}^{n}(-1)^{r+1}\sum_{|J|=r}\mathsf{Pr}\left\{\bigcap_{j\in J} A_j\right\}. \tag{14.5}$$

The next example illustrates it on the depth on a trie.

EXAMPLE 14.2: Generating Function of the Depth in a Trie

Let us compute the generating function of the depth $D_n := D_n(1)$ for the first string X_1. We start with (14.3), and after some easy algebraic manipulation, (14.5) leads to (cf. [31])

$$\mathsf{Pr}\{D_n > k\} = \mathsf{Pr}\left\{\bigcup_{i=2}^{n}\left[C_{i,1} \geq k\right]\right\} = \sum_{r=1}^{n-1}(-1)^{r+1}\binom{n-1}{r}\mathsf{Pr}\left\{C_{2,1} \geq k, \ldots, C_{r+1,1} \geq k\right\};$$

since the probability $\mathsf{Pr}\{C_{2,1} \geq k, \ldots, C_{r+1,1} \geq k\}$ does not depend on the choice of strings (i.e., it is the same for any r-tuple of strings selected). Moreover, it can be easily explicitly computed. Indeed, we obtain $\mathsf{Pr}\{C_{2,1} \geq k, \ldots, C_{r+1,1} \geq k\} = (p^{r+1} + q^{r+1})^k$, since r independent binary strings must agree on the first k symbols (we recall that p stands for the probability a symbol, say "0," occurrence and $q = 1 - p$). Thus, the generating function $D_n(u) = \mathbf{E}[u^{D_n}] = \Sigma_{k\geq 0}\mathsf{Pr}\{D_n = k\}u^k$ becomes

$$\mathbf{E}\left[u^{D_n}\right] = 1 + \sum_{r=1}^{n-1}(-1)^{r+1}\binom{n-1}{r}\frac{1-u}{1-u\left(p^{r+1}+q^{r+1}\right)}$$

The last formula is a simple consequence of the above, and the following well known fact from the theory of generating function $\mathbf{E}[u^X]$ for a random variable X:

$$\mathbf{E}\left[u^X\right] = \frac{1}{1-u}\sum_{k=0}^{\infty}\mathsf{Pr}\{X \leq k\}\,u^k$$

for $|u| < 1$.

In many real computations, however, one cannot explicitly compute the probability of the events union. Often, one must retreat to inequalities that actually are enough to reach one's goal. The most simple yet still very powerful is the following inequality

$$\mathsf{Pr}\left\{\bigcup_{i=1}^{n} A_i\right\} \leq \sum_{i=1}^{n}\mathsf{Pr}\{A_i\}. \tag{14.6}$$

The latter is an example of a series of inequalities due to Bonferroni which can be formulated as follows: For every even integer $m \geq 0$ we have

$$\begin{aligned}\sum_{j=1}^{m}(-1)^{j-1}\sum_{1\leq t_1<\cdots<t_j\leq n} & \quad \mathsf{Pr}\left\{A_{t_1}\cap\cdots\cap A_{t_j}\right\}\\ &\leq \mathsf{Pr}\left\{\bigcup_{i=1}^{n}A_i\right\}\\ &\leq \sum_{j=1}^{m+1}(-1)^{j-1}\sum_{1\leq t_1<\cdots<t_j\leq n}\mathsf{Pr}\left\{A_{t_1}\cap\cdots\cap A_{t_j}\right\}\ .\end{aligned}$$

In combinatorics (e.g., enumeration problems) and probability the so called *inclusion–exclusion principle* is very popular and had many successes. We formulate it in a form of a theorem whose proof can be found in Bollobás [6].

THEOREM 14.2 **(Inclusion–Exclusion Principle)** *Let $A_1, \ldots, A_n$ be events in a probability space, and let p_k be the probability of exactly k of them to occur. Then:*

$$p_k = \sum_{r=k}^{n}(-1)^{r+k}\binom{r}{k}\sum_{|J|=r}\mathsf{Pr}\left\{\bigcap_{j\in J}A_j\right\}\ .$$

EXAMPLE 14.3: Computing a Distribution Through Its Moments (cf.[6])

Let X be a random variable defined on $\{0, 1, \ldots, n\}$, and let $\mathbf{E}_r[X] = \mathbf{E}X(X-1)\cdots(X-r+1)$ be the rth factorial moment of X. Then

$$\mathsf{Pr}\{X=k\} = \frac{1}{k!}\sum_{r=k}^{n}(-1)^{r+k}\frac{\mathbf{E}_r[X]}{(r-k)!}\ .$$

Indeed, it suffices to set $A_i = \{X \geq i\}$ for all $i = 1, \ldots, n$, and observe that $\Sigma_{|J|=r}\mathsf{Pr}\{\bigcap_{j\in J}A_j\} = \mathbf{E}_r[X]/r!$. Since the event $\{X = k\}$ is equivalent to the event that exactly k of A_i occur, a simple application of Theorem 14.2 proves the announced result.

Inequalities: First and Second Moment Methods

In this subsection, we review some inequalities that play a considerable role in probabilistic analysis of algorithms. In particular, we discuss *first* and *second moment methods.*

We start with a few standard inequalities (cf. [14]):

Markov Inequality: For a nonnegative random variable X and $\varepsilon > 0$ the following holds:

$$\mathsf{Pr}\{X \geq \varepsilon\} \leq \frac{\mathbf{E}[X]}{\varepsilon}\ .$$

Indeed: let $I(A)$ be the indicator function of A (i.e., $I(A) = 1$ if A occurs, and zero otherwise). Then,

$$\mathbf{E}[X] \geq \mathbf{E}\left[XI\left(X \geq \varepsilon\right)\right] \geq \varepsilon\mathbf{E}\left[I\left(X \geq \varepsilon\right)\right] = \varepsilon\mathsf{Pr}\left\{X \geq \varepsilon\right\}\ .$$

Chebyshev's Inequality: If one replaces X by $|X - \mathbf{E}[X]|$ in the Markov inequality, then

$$\mathsf{Pr}\{|X - \mathbf{E}[X]| > \varepsilon\} \leq \frac{\mathbf{Var}[X]}{\varepsilon^2}\ .$$

Schwarz's Inequality (also called Cauchy–Schwarz): Let X and Y be such that $\mathbf{E}[X^2] < \infty$ and $\mathbf{E}[Y^2] < \infty$. Then

$$\mathbf{E}[|XY|]^2 \leq \mathbf{E}[X^2]\mathbf{E}[Y^2] ,$$

where $\mathbf{E}[X]^2 := (\mathbf{E}[X])^2$.

Jensen's Inequality: Let $f(\cdot)$ be a downward convex function, that is, for $\lambda \in (0, 1)$ we have $\lambda f(x) + (1 - \lambda) f(y) \geq f(\lambda x + (1 - \lambda)y)$. Then

$$f(\mathbf{E}[X]) \leq \mathbf{E}[f(X)] .$$

The remainder part of this subsection is devoted to the first and the second moment methods that we illustrate on several examples arising in the analysis of digital trees. The *first moment method* for a nonnegative random variable X states that

$$\mathsf{Pr}\{X > 0\} \leq \mathbf{E}[X] . \tag{14.7}$$

This follows directly from Markov's inequality after setting $\varepsilon = 1$. The above inequality implies also the basic Bonferroni inequality (14.6). Indeed, let A_i $(i = 1, \ldots, n)$ be events, and set $X = I(A_1) + \cdots + I(A_n)$. Inequality (14.6) follows.

In a typical application of (14.7), we expect to show that $\mathbf{E}[X] \to 0$, just $X = 0$ occurs almost always or *with high probability* (whp). We illustrate it in the next example.

EXAMPLE 14.4: Upper Bound on the Height in a Trie

In Example 14.1 we showed that the height H_n of a trie is given by (14.2) or (14.4). Thus, using the first moment method we have

$$\mathsf{Pr}\{H_n \geq k + 1\} \leq \mathsf{Pr}\left\{ \max_{1 \leq i < j \leq n} \{C_{ij}\} \geq k \right\} \leq n^2 \mathsf{Pr}\{C_{ij} \geq k\}$$

for any integer k. From Example 14.2 we know that $\mathsf{Pr}\{C_{ij} \geq k\} = (p^2 + q^2)^k$. Let $P = p^2 + q^2$, $Q = P^{-1}$, and set $k = 2(1 + \varepsilon) \log_Q n$ for any $\varepsilon > 0$. Then, the above implies

$$\mathsf{Pr}\left\{ H_n \geq 2(1 + \varepsilon) \log_Q n + 1 \right\} \leq \frac{n^2}{n^{2(1+\varepsilon)}} = \frac{1}{n^{2\varepsilon}} \to 0 ,$$

thus, $H_n/(2 \log_Q n) \leq 1$ (pr.). Below, in Example 14.5, we will actually prove that $H_n/(2 \log_Q n) = 1$ (pr.) by establishing a matching lower bound.

Let us look now at the *second moment method.* Setting in the Chebyshev inequality $\varepsilon = \mathbf{E}[X]$ we prove that

$$\mathsf{Pr}\{X = 0\} \leq \frac{\mathbf{Var}[X]}{\mathbf{E}[X]^2} .$$

But, one can do better (cf. [3, 10]). Using Schwarz's inequality for a random variable X we obtain the following chain of inequalities

$$\mathbf{E}[X]^2 = \mathbf{E}[I(X \neq 0)X]^2 \leq \mathbf{E}[I(X \neq 0)]\mathbf{E}\left[X^2\right] = \mathsf{Pr}\{I(X \neq 0)\}\mathbf{E}\left[X^2\right] ,$$

which finally implies the second moment inequality

$$\mathsf{Pr}\{X > 0\} \geq \frac{\mathbf{E}[X]^2}{\mathbf{E}[X^2]} . \tag{14.8}$$

Actually, another formulation of this inequality due to Chung and Erdös is quite popular. To derive it, set in (14.8) $X = I(A_1) + \cdots + I(A_n)$ for a sequence of events $A_1, \ldots, A_n$. Noting that $\{X > 0\} = \bigcup_{i=1}^{n} A_i$, we obtain from (14.8) after some algebra

$$\mathsf{Pr}\left\{\bigcup_{i=1}^{n} A_i\right\} \geq \frac{\left(\sum_{i=1}^{n} \mathsf{Pr}\{A_i\}\right)^2}{\sum_{i=1}^{n} \mathsf{Pr}\{A_i\} + \sum_{i \neq j} \mathsf{Pr}\left\{A_i \cap A_j\right\}} \,. \tag{14.9}$$

In a typical application, if we are able to prove that $\mathbf{Var}[X]/\mathbf{E}[X^2] \to 0$, then we can show that $\{X > 0\}$ almost always. The next example — which is a continuation of Example 14.4 — illustrates this point.

EXAMPLE 14.5: Lower Bound for the Height in a Trie

We now prove that $\mathsf{Pr}\{H_n \geq 2(1-\varepsilon)\log_Q n\} \to 1$ for any $\varepsilon > 0$, just completing the proof that $H_n/(2\log_Q n) \to 1$ (pr.). We use the Chung–Erdös formulation, and set $A_{ij} = \{C_{ij} \geq k\}$. Throughout this example, we assume $k = 2(1-\varepsilon)\log_Q n$. Observe that now in (14.9) we must replace the single summation index i by a double summation index (i, j). The following is obvious: $\Sigma_{1 \leq i < j \leq n} \mathsf{Pr}\{A_{ij}\} = \frac{1}{2} n(n-1) P^k$, where $P = p^2 + q^2$. The other sum in (14.9) is a little harder to deal with. We must sum over (i, j), (l, m), and we consider two cases: (i) all indices are different, (ii) $i = l$ (i.e., we have (i, j), (i, m)). In the second case we must consider the probability $\mathsf{Pr}\{C_{ij} \geq k, C_{i,m} \geq k\}$. But, as in Example 14.2, we obtain $\mathsf{Pr}\{C_{ij} \geq k, C_{i,m} \geq k\} = (p^3 + q^3)^k$ since once you choose a symbol in the string X_i you must have the same symbol at the same position in X_j, X_m. In summary,

$$\sum_{(ij),(l,m)} \mathsf{Pr}\left\{C_{ij} \geq k,\ C_{lm} \geq k\right\} \leq \frac{1}{4} n^4 P^{2k} + n^3 \left(p^3 + q^3\right)^k \,.$$

To complete the derivation, it suffices to observe that

$$\left(p^3 + q^3\right)^{\frac{1}{3}} \leq P^{\frac{1}{2}} \,.$$

which is easy to prove by elementary methods (cf. [60]). Then, (14.9) becomes

$$\begin{aligned} \mathsf{Pr}\{H_n \geq k+1\} = \mathsf{Pr}\left\{\bigcup_{i=1}^{n} A_i\right\} &\geq \frac{1}{n^{-2} P^{-k} + 1 + 4\left(p^3 + q^3\right)^k / \left(n P^{2k}\right)} \\ &\geq \frac{1}{1 + n^{-2\varepsilon} + 4/\left(n P^{k/2}\right)} \geq \frac{1}{1 + n^{-2\varepsilon} + 4n^{-\varepsilon}} \to 1 \,. \end{aligned}$$

Thus, we have shown that $H_n/(2\log_Q n) \geq 1$ (pr.), which completes our proof of

$$\lim_{n \to \infty} \mathsf{Pr}\left\{2(1-\varepsilon)\log_Q n \leq H_n \leq 2(1+\varepsilon)\log_Q n\right\} = 1$$

for any $\varepsilon > 0$.

Subadditive Ergodic Theorem

The celebrated *ergodic theorem* of Birkhoff [15] found many useful applications in computer science. It is used habitually during a computer simulation run or whenever one must perform experiments and collect data. However, for probabilistic analysis of algorithms a generalization of this result due to Kingman [36] is more important. We briefly review it here and illustrate on a few examples.

Let us start with the following well known fact attributed to Fekete (cf. [50]). Assume a (deterministic) sequence $\{x_n\}_{n=0}^{\infty}$ satisfies the so called *subadditivity property,* that is,

$$x_{m+n} \le x_n + x_m$$

for all integers $m, n \ge 0$. It is easy to see that then (cf. [14])

$$\lim_{n\to\infty} \frac{x_n}{n} = \inf_{m\ge 1} \frac{x_m}{m} = \alpha$$

for some $\alpha \in [-\infty, \infty)$. Indeed, fix $m \ge 0$, write $n = km + l$ for some $0 \le l \le m$, and observe that by the above subadditivity property

$$x_n \le k x_m + x_l \ .$$

Taking $n \to \infty$ with $n/k \to m$ we finally arrive at $\limsup_{n\to\infty} \frac{x_n}{n} \le \inf_{m\ge 1} \frac{x_m}{m} \le \alpha$ where the last inequality follows from arbitrariness of m. This completes the derivation since $\liminf_{n\to\infty} \frac{x_n}{n} \ge \alpha$ is automatic. One can also see that replacing "$\le$" in the subadditivity property by "$\ge$" (thus, *superadditivity property*) will not change our conclusion except that $\inf_{m\ge 1} \frac{x_m}{m}$ should be replaced by $\sup_{m\ge 1} \frac{x_m}{m}$.

In the early 1970s people started asking whether the above deterministic subadditivity result could be extended to a sequence of random variables. Such an extension would have an impact on many research problems of those days. For example, Chvatal and Sankoff [9] used ingenious tricks to establish the probabilistic behavior of the *Longest Common Subsequence* problem (cf. "String Editing Problem" and below) while we show below that it is a trivial consequence of a stochastic extension of the above subadditivity result. In 1976 Kingman [36] presented the first proof of what later will be called *Subadditivity Ergodic Theorem.* Below, we present an extension of Kingman's result (cf. [50]).

To formulate it properly we must consider a sequence of doubly indexed random variables $X_{m,n}$ with $m \le n$. One can think of it as $X_{m,n} = (X_m, X_{m+1}, \ldots, X_n)$, that is, as a substring of a single-indexed sequence X_n.

THEOREM 14.3 **(Subadditive Ergodic Theorem [36])** (i) *Let $X_{m,n}$ $(m < n)$ be a sequence of nonnegative random variables satisfying the following three properties*

(a) $X_{0,n} \le X_{0,m} + X_{m,n}$ *(subadditivity);*
(b) $X_{m,n}$ *is stationary (i.e., the joint distributions of $X_{m,n}$ are the same as $X_{m+1,n+1}$) and ergodic (cf. [14]);*
(c) $\mathbf{E}[X_{0,1}] < \infty$.

Then,

$$\lim_{n\to\infty} \frac{\mathbf{E}[X_{0,n}]}{n} = \gamma \quad \text{and} \quad \lim_{n\to\infty} \frac{X_{0,n}}{n} = \gamma \quad \text{(a.s.)} \tag{14.10}$$

for some constant γ.
(ii) (**Almost Subadditive Ergodic Theorem**) *If the subadditivity inequality is replaced by*

$$X_{0,n} \le X_{0,m} + X_{m,n} + A_n \tag{14.11}$$

such that $\lim_{n\to\infty} \mathbf{E}[A_n/n] = 0$, then (14.10) holds, too.

We must point out, however, that the above result proves only the existence of a constant γ such that (14.10) holds. It says *nothing* how to compute it, and in fact many ingenious methods have been devised in the past to bound this constant. We discuss it in a more detailed way in the examples below.

EXAMPLE 14.6: String Editing Problem

Let us consider the string editing problem of the subsection "String Editing Problem." To recall, one is interested in estimating the maximum cost $C_{\max}$ of transforming one sequence into another. This problem can be reduced to finding the longest (shortest) path in a special grid graph (cf. Fig. 14.2). Let us assume that the weights W_I, W_D, and W_Q are independently distributed, thus, we adopt the Bernoulli model (B) of "Probabilistic Models of Strings." Then, using the subadditive ergodic theorem it is immediate to see that

$$\lim_{n\to\infty} \frac{C_{\max}}{n} = \lim_{n\to\infty} \frac{EC_{\max}}{n} = \alpha \quad \text{(a.s.)} ,$$

for some constant $\alpha > 0$, provided ℓ/s has a limit as $n\to\infty$. Indeed, let us consider the $\ell \times s$ grid with starting point O and ending point E (cf. Fig. 14.2). Call it Grid(O,E). We also choose an arbitrary point, say A, inside the grid so that we can consider two grids, namely Grid(O,A) and Grid(A,E). Actually, point A splits the edit distance problem into two subproblems with objective functions $C_{\max}(O, A)$ and $C_{\max}(A, E)$. Clearly, $C_{\max}(O, E) \geq C_{\max}(O, A) + C_{\max}(A, E)$. Thus, under our assumption regarding weights, the objective function $C_{\max}$ is superadditive, and direct application of the *Superadditive Ergodic Theorem* proves the result.

Entropy and Its Applications

Entropy and *mutual information* was introduced by Shannon in 1948, and overnight a new field of *information theory* was born. Over the last 50 years information theory underwent many changes, and remarkable progress was achieved. These days entropy and the **Shannon–McMillan–Breiman Theorem** are standard tools of the average case analysis of algorithms. In this subsection, we review some elements of information theory and illustrate its usage to the analysis of algorithms.

Let us start with a simple observation: Consider a binary sequence of symbols of length n, say $(X_1, \ldots, X_n)$, with p denoting the probability of one symbol and $q = 1 - p$ the probability of the other symbol. When $p = q = 1/2$, then $\mathsf{Pr}\{X_1, \ldots, X_n\} = 2^{-n}$, and it does not matter what are the actual values of $X_1, \ldots, X_n$. In general, $\mathsf{Pr}\{X_1, \ldots, X_n\}$ is not the same for all possible values of $X_1, \ldots, X_n$, however, we shall show that a **typical** sequences $(X_1, \ldots, X_n)$ have "asymptotically" the same probability. Indeed, consider $p \neq q$ in the example above. Then, the probability of a typical sequence is approximately equal to (we use here the central limit theorem for i.i.d. sequences):

$$p^{np+O(\sqrt{n})} q^{nq+O(\sqrt{n})} = e^{-n(-p\log p - q\log q)+O(\sqrt{n})} \sim e^{-nh}$$

where $h = -p\log p - q\log q$ is the *entropy* of the underlying Bernoulli model. Thus, a typical sequence X_1^n has asymptotically the same probability equal to e^{-nh}.

To be more precise, let us consider a *stationary and ergodic sequence* $\{X_k\}_{k=1}^{\infty}$, and define $X_m^n = (X_m, X_{m+1}, \ldots, X_n)$ for $m \leq n$ as a substring of $\{X_k\}_{k=1}^{\infty}$. The entropy rate h of $\{X_k\}_{k=1}^{\infty}$ is defined as (cf. [11, 14])

$$h := -\lim_{n\to\infty} \frac{\mathbf{E}\left[\log \mathsf{Pr}\left\{X_1^n\right\}\right]}{n} , \tag{14.12}$$

where one can prove the limit above exists. We must point out that $\mathsf{Pr}\{X_1^n\}$ is a *random variable* since X_1^n is a random sequence!

We show now how to derive the Shannon–McMillan–Breiman Theorem in the case of the Bernoulli model and the mixing model, and later we formulate the theorem in its full generality. Consider first the Bernoulli model, and let $\{X_k\}$ be generated by a Bernoulli source. Thus,

$$-\frac{\log \mathsf{Pr}\left\{X_1^n\right\}}{n} = -\frac{1}{n}\sum_{i=1}^{n} \log \mathsf{Pr}\{X_i\} \to \mathbf{E}\left[-\log \mathsf{Pr}\{X_1\}\right] = h \qquad \text{(a.s.)} ,$$

where the last implication follows from the *Strong Law of Large Numbers* (cf. [14]) applied to the sequence $(-\log \Pr\{X_1\}, \ldots, -\log \Pr\{X_n\})$. One *should* notice a difference between the definition of the entropy (14.12) and the result above. In (14.12) we take the *average* of $\log \Pr\{X_1^n\}$ while in the above we proved that *almost surely* for all but finitely sequences the probability $\Pr\{X_1^n\}$ can be closely approximated by e^{-nh}. For the Bernoulli model, we have already seen it above, but we are aiming at showing that the above conclusion is true for much more general probabilistic models.

As the next step, let us consider the mixing model (MX) (that includes as a special case the Markovian model (M)). For the mixing model the following is true:

$$\Pr\left\{X_1^{n+m}\right\} \leq c \Pr\left\{X_1^n\right\} \Pr\left\{X_{n+1}^{n+m}\right\}$$

for some constant $c > 0$ and any integers $n, m \geq 0$. Taking logarithm we obtain

$$\log \Pr\left\{X_1^{n+m}\right\} \leq \log \Pr\left\{X_1^n\right\} + \log \Pr\left\{X_{n+1}^{n+m}\right\} + \log c$$

which satisfies the subadditivity property (14.11) of the *Subadditive Ergodic Theorem* discussed in "Subadditive Ergodic Theorem." Thus, by (14.10) we have

$$h = -\lim_{n\to\infty} \frac{\log \Pr\left\{X_1^n\right\}}{n} \quad \text{(a.s.)} .$$

Again, the reader should notice the difference between this result and the definition of the entropy.

We are finally ready to state the Shannon–McMillan–Breiman Theorem in its full generality (cf. [14]).

THEOREM 14.4 **(Shannon–McMillan–Breiman)** *For a stationary and ergodic sequence* $\{X_k\}_{k=-\infty}^{\infty}$ *the following holds:*

$$h = -\lim_{n\to\infty} \frac{\log \Pr\left\{X_1^n\right\}}{n} \quad \text{(a.s.)} .$$

where h *is the entropy rate of the process* $\{X_k\}$.

An important conclusion of this result is the so-called **asymptotic equipartition property** (AEP) which basically asserts that asymptotically typical sequences have the same probability approximately equal to e^{-nh}. More precisely, *For a stationary and ergodic sequence* X_1^n, *the state space* Σ^n *can be partitioned into two subsets* $\mathcal{B}_n^\varepsilon$ *("bad set") and* $\mathcal{G}_n^\varepsilon$ *("good set") such that for given* $\varepsilon > 0$ *there is* N_ε *so that for* $n \geq N_\varepsilon$ *we have* $\Pr\{\mathcal{B}_n^\varepsilon\} \leq \varepsilon$, *and* $e^{-nh(1+\varepsilon)} \leq \Pr\{x_1^n\} \leq e^{-nh(1-\varepsilon)}$ *for all* $x_1^n \in \mathcal{G}_n^\varepsilon$.

EXAMPLE 14.7: Shortest Common Superstring or Depth in a Trie/Suffix Tree

For concreteness let us consider the Shortest Common Superstring discussed in the subsection "Shortest Common Superstring," but the same arguments as below can be used to derive the depth in a trie (cf. [48]) or a suffix tree (cf. [54]). Define C_{ij} as the length of the longest suffix of X_i that is equal to the prefix of X_j. Let $M_n(i) = \max_{1\leq j\leq n, j\neq i}\{C_{ij}\}$. We write M_n for a generic random variable distributed as $M_n(i)$ (observe that $M_n \stackrel{d}{=} M_n(i)$ for all i, where $\stackrel{d}{=}$ means "equal in distribution"). We would like to prove that in the mixing model, for any $\varepsilon > 0$,

$$\lim_{n\to\infty} \Pr\left\{(1-\varepsilon)\frac{1}{h}\log n \leq M_n \leq (1+\varepsilon)\frac{1}{h}\log n\right\} = 1$$

provided $\alpha(g) \to 0$ as $g \to \infty$, that is, $M_n/\log n \to h$ (pr.). To prove an upper bound, we take *any fixed* typical sequence $w_k \in \mathcal{G}_k^\varepsilon$ as defined in AEP above, and observe that $\Pr\{M_n \geq k\} \leq n\Pr\{w_k\} + \Pr\{\mathcal{B}_k\}$.

The result follows immediately after substituting $k = (1+\varepsilon)h^{-1}\log n$ and noting that $\mathsf{Pr}\{w_k\} \le e^{nh(1-\varepsilon)}$. For a lower bound, let $w_k \in \mathcal{G}_k^{\varepsilon}$ be any fixed typical sequence with $k = \frac{1}{h}(1-\varepsilon)\log n$. Define Z_k as the number of strings $j \neq i$ such that a prefix of length k is equal to w_k *and* a suffix of length k of the ith string is equal to $w_k \in \mathcal{G}_k^{\varepsilon}$. Since w_k is fixed, the random variables C_{ij} are independent, and hence, by the *second moment method* (cf. "Inequalities: First and Second Moment Method")

$$\mathsf{Pr}\{M_n < k\} = \mathsf{Pr}\{Z_k = 0\} \le \frac{\mathbf{Var}\, Z_k}{(\mathbf{E}Z_k)^2} \le \frac{1}{n\mathsf{Pr}\{w_k\}} = O\left(n^{-\varepsilon^2}\right) ,$$

since $\mathbf{Var}\, Z_k \le n\, P(w_k)$, and this completes the derivation.

In many problems on words another kind of entropy is widely used (cf. [4, 5, 54]). It is called **Rényi entropy** and defined as follows: For $-\infty \le b \le \infty$, the bth *order Rényi entropy* is

$$h_b = \lim_{n\to\infty} \frac{-\log\left(\mathbf{E}\left[\mathsf{Pr}\left\{X_1^n\right\}^{b-1}\right]\right)}{bn} = \lim_{n\to\infty} \frac{-\log\left(\sum_{w\in\Sigma^n}\left(\mathsf{Pr}\{w\}\right)^b\right)^{-1/b}}{n} , \tag{14.13}$$

provided the above limit exists. In particular, by *inequalities on means* we obtain $h_0 = h$ and

$$\begin{aligned} h_{-\infty} &= \lim_{n\to\infty} \frac{\max\left\{-\log \mathsf{Pr}\left\{X_1^n\right\}\ , \mathsf{Pr}\left\{X_1^n\right\} > 0\right\}}{n} , \\ h_{\infty} &= \lim_{n\to\infty} \frac{\min\left\{-\log \mathsf{Pr}\left\{X_1^n\right\}\ , \mathsf{Pr}\left\{X_1^n\right\} > 0\right\}}{n} . \end{aligned}$$

For example, the entropy $h_{-\infty}$ appears in the formulation of the shortest path in digital trees (cf. [48, 54]), the entropy h_{∞} is responsible for the height in PATRICIA tries (cf. [48, 54]), while h_2 determines the height in a trie. Indeed, we claim that in the mixing model the height H_n in a trie behaves probabilistically as $H_n/\log n \to 2/h_2$. To prove it, one should follow the footsteps of our discussion in Examples 14.4 and 14.5 (details can be found in [48, 54]).

Central Limit and Large Deviations Results

Convergence of a sum of independent, identically distributed (i.i.d.) random variables is central to probability theory. In the analysis of algorithms, we mostly deal with *weakly dependent* random variables, but often results from the i.i.d. case can be extended to this new situation by some clever tricks. A more systematic treatment of such cases is usually done through generating functions and complex analysis techniques (cf. [30, 31, 32, 44]), which we briefly discuss in the next section. Hereafter, we concentrate on the i.i.d. case.

Let us consider a sequence $X_1, \ldots, X_n$ of i.i.d. random variables, and let $S_n = X_1 + \cdots + X_n$. Define $\mu := \mathbf{E}[X_1]$ and $\sigma^2 := \mathbf{Var}[X_1]$. We pay particular interest to another random variable, namely

$$s_n := \frac{S_n - n\mu}{\sigma\sqrt{n}}$$

which distribution function we denote as $F_n(x) = \mathsf{Pr}\{s_n \le x\}$. Let also $\Phi(x)$ be the distribution function of the standard normal distribution, that is,

$$\Phi(x) := \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{x} e^{-\frac{1}{2}t^2} dt .$$

The **Central Limit Theorem** asserts that $F_n(x) \to \Phi(x)$ for continuity points of $F_n(\cdot)$, provided $\sigma < \infty$ (cf. [14, 15]). A stronger version is due to Berry-Esséen who proved that

$$|F_n(x) - \Phi(x)| \le \frac{2\rho}{\sigma^2\sqrt{n}} \tag{14.14}$$

where $\rho = \mathbf{E}[|X-\mu|^3] < \infty$. Finally, Feller [15] has shown that if centralized moments $\mu_2, \ldots, \mu_r$ exist, then

$$F_n(x) = \Phi(x) - \frac{1}{\sqrt{2\pi}} e^{-\frac{1}{2}x^2} \sum_{k=3}^{r} n^{-\frac{1}{2}k+1} R_k(x) + O\left(n^{-\frac{1}{2}r+\frac{1}{2}}\right)$$

uniformly in x, where $R_k(x)$ is a polynomial depending only on $\mu_1, \ldots, \mu_r$ but not on n and r.

One should notice from the above, in particular from (14.14), the weakness of central limit results that are able only to assess the probability of *small deviations* from the mean. Indeed, the results above are true for $x = O(1)$ (i.e., for $S_n \in (\mu n - O(\sqrt{n}), \mu n + O(\sqrt{n}))$ due to only a polynomial rate of convergence as shown in (14.14). To see it more clearly, we quote a result from Greene and Knuth [25] who estimated

$$\Pr\{S_n = \mu n + r\} = \frac{1}{\sigma\sqrt{2\pi n}} \exp\left(\frac{-r^2}{2\sigma^2 n}\right)\left(1 - \frac{\kappa_3}{2\sigma^4}\left(\frac{r}{n}\right) + \frac{\kappa_3}{6\sigma^6}\left(\frac{r^3}{n^2}\right)\right) + O\left(n^{-\frac{3}{2}}\right) \qquad (14.15)$$

where κ_3 is the third cumulant of X_1. Observe now that when $r = O(\sqrt{n})$ (which is equivalent to $x = O(1)$ in our previous formulæ) the error term *dominates* the leading term of the above asymptotic, thus, the estimate is quite useless.

From the above discussion, one should conclude that the central limit theorem has limited range of application, and one should expect another law for *large deviations* from the mean, that is, when $x_n \to \infty$ in the above formulæ. The most interesting from the application point of view is the case when $x = O(\sqrt{n})$ (or $r = O(n)$), that is, for $\Pr\{S_n = n(\mu+\delta)\}$ for $\delta \neq 0$. We shall discuss this large deviations behavior next.

Let us first try to "guess" a large deviation behavior of $S_n = X_1 + \cdots X_n$ for i.i.d. random variables. We estimate $\Pr\{S_n \geq an\}$ for $a > 1$ as $n \to \infty$. Observe that (cf. [14])

$$\Pr\{S_{n+m} \geq (n+m)a\} \geq \Pr\{S_m \geq ma,\ S_{n+m} - S_m \geq na\} = \Pr\{S_n \geq na\}\Pr\{S_m \geq ma\}\ ,$$

since S_m and $S_{n+m} - S_m$ are independent. Taking logarithm of the above, and recognizing that $\log \Pr\{S_n \geq an\}$ is a superadditive sequence (cf. "Subadditive Ergodic Theorem"), we obtain

$$\lim_{n\to\infty} \frac{1}{n} \log \Pr\{S_n \geq na\} = -I(a)\ ,$$

where $I(a) \geq 0$. Thus, S_n decays *exponentially* when far away from its mean, not in a Gaussian way as the central limit theorem would predict! Unfortunately, we obtain the above result from the subadditive property which allowed us to conclude the existence of the above limit, but says nothing about $I(a)$.

In order to take the full advantage of the above derivation, we should say something about $I(a)$ and, more importantly, to show that $I(a) > 0$ under some mild conditions. For the latter, let us first assume that the *moment generating function* $M(\lambda) = \mathbf{E}[e^{\lambda X_1}] < \infty$ for some $\lambda > 0$. Let also $\kappa(\lambda) = \log M(\lambda)$ be the *cumulant function* of X_1. Then, by Markov's inequality (cf. "Inequalities: First and Second Moment Methods")

$$e^{\lambda na}\Pr\{S_n \geq na\} = e^{\lambda na}\Pr\left\{e^{\lambda S_n} \geq e^{\lambda na}\right\} \leq \mathbf{E}e^{\lambda S_n}\ .$$

Actually, due to arbitrariness of $\lambda > 0$, we finally arrive at the so-called **Chernoff bound**, that is,

$$\Pr\{S_n \geq na\} \leq \min_{\lambda>0}\left\{e^{-\lambda na}\mathbf{E}\left[e^{\lambda S_n}\right]\right\}\ . \qquad (14.16)$$

We should emphasize that the above bound is true for *dependent* random variables since we only used Markov's inequality applied to S_n.

Returning to the i.i.d. case, we can rewrite the above as

$$\Pr\{S_n \geq na\} \leq \min_{\lambda>0}\left\{\exp(-n(a\lambda - \kappa(\lambda))\right\}\ .$$

But, under mild conditions the above minimization problem is easy to solve. One finds that the minimum is attended at λ_a which satisfies $a = M'(\lambda_a)/M(\lambda_a)$. Thus, we proved that $I(a) \geq a\lambda_a - \log M(\lambda_a)$. A more careful evaluation of the above leads to the following classical large deviations result (cf. [14])

THEOREM 14.5 *Assume $X_1, \ldots, X_n$ are i.i.d. Let $M(\lambda) = \mathbf{E}[e^{\lambda X_1}] < \infty$ for some $\lambda > 0$, the distribution of X_i is not a point mass at μ, and there exists $\lambda_a > 0$ in the domain of the definition of $M(\lambda)$ such that*

$$a = \frac{M'(\lambda_a)}{M(\lambda_a)} .$$

Then

$$\lim_{n\to\infty} \frac{1}{n} \log \mathsf{Pr}\{S_n \geq na\} = -\big(a\lambda_a - \log M(\lambda_a)\big)$$

for $a > \mu$.

A major strengthening of this theorem is due to Gärtner and Ellis (cf. [13]) who extended it to weakly dependent random variables. Let us consider S_n as a sequence of random variables (e.g., $S_n = X_1 + \ldots + X_n$), and let $M_n(\lambda) = \mathbf{E}[e^{\lambda S_n}]$. The following is known (cf. [13, 29]):

THEOREM 14.6 **(Gärtner–Ellis)** *Let*

$$\lim_{n\to\infty} \frac{\log M_n(\lambda)}{n} = c(\lambda)$$

exist and is finite in a subinterval of the real axis. If there exists λ_a such that $c'(\lambda_a)$ is finite and $c'(\lambda_a) = a$, then

$$\lim_{n\to\infty} \frac{1}{n} \log \mathsf{Pr}\{S_n \geq na\} = -(a\lambda_a - c(\lambda_a)) .$$

Let us return again to the i.i.d. case and see if we can strengthen Theorem 14.5 which in its present form gives only a logarithmic limit. We explain our approach on a simple example, following Greene and Knuth [25]. Let us assume that $X_1, \ldots, X_n$ are discrete i.i.d. with common generating function $G(z) = \mathbf{E}[z^X]$. We recall that $[z^m]G(z)$ denote the coefficient at z^m of $G(z)$. In (14.15) we show how to compute such a coefficient at $m = \mu n + O(\sqrt{n})$ of $G^n(z) = Ez^{S_n}$. We observed also that (14.15) cannot be used for large deviations, since the error term was dominating the leading term in such a case. But, one may shift the mean of S_n to a new value such that (14.15) is valid again. Thus, let us define a new random variable $\widetilde{X}$ whose generating function is $\widetilde{G}(z) = \frac{G(z\alpha)}{G(\alpha)}$ where α is a constant that is to be determined. Observe that $\mathbf{E}[\widetilde{X}] = \widetilde{G}'(1) = \alpha G'(\alpha)/G(\alpha)$. Suppose we need large deviations result around $m = n(\mu + \delta)$ where $\delta > 0$. Clearly, (14.15) cannot be applied directly. Now, a proper choice of α can help. Let us select α such that the new $\widetilde{S}_n = \widetilde{X}_1 + \cdots + \widetilde{X}_n$ has mean $m = n(\mu + \delta)$. This results in setting α to be a solution of

$$\frac{\alpha G'(\alpha)}{G(\alpha)} = \frac{m}{n} = \mu + \delta .$$

In addition, we have the following obvious identity

$$\left[z^m\right] G^n(z) = \frac{G^n(\alpha)}{\alpha^m} \left[z^m\right] \left(\frac{G(\alpha z)}{G(\alpha)}\right)^n . \tag{14.17}$$

But, now we can use (14.15) to the right-hand side of the above, since the new random variable $\widetilde{S}_n$ has mean around m.

To illustrate the above technique that is called **shift of mean** we present an example.

EXAMPLE 14.8: Large Deviations by "Shift of Mean" (cf. [25]).

Let S_n be binomially distributed with parameter 1/2, that is, $G^n(z) = ((1+z)/2)^n$. We want to estimate the probability $\mathsf{Pr}\{S_n = n/3\}$, which is far away from its mean ($ES_n = n/2$) and central limit result (14.15) cannot be applied. We apply the shift of mean method, and compute α as

$$\frac{\alpha G'(\alpha)}{G(\alpha)} = \frac{\alpha}{1+\alpha} = \frac{1}{3} ,$$

thus, $\alpha = 1/2$. Using (14.15) we obtain

$$\left[z^{n/3}\right]\left(\frac{2}{3}+\frac{1}{3}z\right)^n = \frac{3}{2\sqrt{\pi n}}\left(1-\frac{7}{24n}\right)+O\left(n^{-5/2}\right) .$$

To derive the result we want (i.e., coefficient at $z^{n/3}$ of $(z/2+1/2)^n$), one must apply (14.17). This finally leads to

$$\left[z^{n/3}\right](z/2+1/2)^n = \left(\frac{3\cdot 2^{1/3}}{4}\right)^n \frac{3}{2\sqrt{\pi n}}\left(1-\frac{7}{24n}+O\left(n^{-2}\right)\right) ,$$

which is a large deviations result (the reader should observe the exponential decay of the above probability).

The last example showed that one may expect a stronger large deviation result than presented in Theorem 14.5. Indeed, under proper mild conditions it can be proved that Theorem 14.5 extends to (cf. [13, 29])

$$\mathsf{Pr}\{S_n \geq na\} \sim \frac{1}{\sqrt{2\pi n}\sigma_a\lambda_a}\exp(-nI(a))$$

where $\sigma_a = M''(\lambda_a)$ with λ_a and $I(a) = a\lambda_a - \log M(\lambda_a)$ is defined in Theorem 14.5.

Finally, we deal with an interesting extension of the above large deviations results initiated by Azuma, and recently significantly extended by Talagrand [56]. These results are known in the literature under the name **Azuma's type inequality** or **method of bounded differences** (cf. [45]). It can be formulated as follows:

THEOREM 14.7 **(Azuma's Type Inequality)** *Let X_i be i.i.d. random variables such that for some function $f(\cdot,\ldots,\cdot)$ the following is true*

$$\left|f(X_1,\ldots,X_i,\ldots,X_n) - f\left(X_1,\ldots,X_i',\ldots,X_n\right)\right| \leq c_i , \tag{14.18}$$

where $c_i < \infty$ are constants, and X_i' has the same distribution as X_i. Then,

$$\mathsf{Pr}\{|f(X_1,\ldots,X_n) - \mathbf{E}[f(X_1,\ldots,X_n)]| \geq t\} \leq 2\exp\left(-2t^2/\sum_{i=1}^n c_i^2\right) \tag{14.19}$$

for some $t > 0$.

We finish this discussion with an application of the Azuma inequality.

EXAMPLE 14.9: Concentration of Mean for the Editing Problem

Let us consider again the editing problem from the subsection "String Editing Problem." The following is true:

$$\mathsf{Pr}\{|C_{\max} - EC_{\max}| > \varepsilon\mathbf{E}[C_{\max}]\} \leq 2\exp\left(-\varepsilon^2\alpha n\right) ,$$

provided all weights are bounded random variables, say $\max\{W_I, W_D, W_Q\} \leq 1$. Indeed, under the Bernoulli model, the X_i are i.i.d. (where $X_i, 1 \leq i \leq n = l+s$, represents symbols of the two underlying sequences), and therefore (14.18) holds with $f(\cdot) = C_{\max}$. More precisely,

$$\left|C_{\max}(X_1, \ldots, X_i, \ldots, X_n) - C_{\max}\left(X_1, \ldots, X_i', \ldots, X_n\right)\right| \leq \max_{1 \leq i \leq n} \{W_{\max}(i)\} ,$$

where $W_{max}(i) = \max\{W_I(i), W_D(i), W_Q(i)\}$. Setting $c_i = 1$ and $t = \varepsilon E C_{\max} = O(n)$ in the Azuma inequality we obtain the desired concentration of mean inequality.

14.5 Analytic Techniques

Analytic analysis of algorithms was initiated by Knuth almost 30 years ago in his *magnum opus* [39, 40, 41], which treated many aspects of fundamental algorithms, seminumerical algorithms, and sorting and searching. A modern introduction to analytic methods can be found in a marvelous book by Sedgewick and Flajolet [49], while advanced analytic techniques are covered in a forthcoming book *Analytical Combinatorics* by Flajolet and Sedgewick [21]. In this section, we only briefly discuss functional equations arising in the analysis of digital trees, complex asymptotics techniques, Mellin transform, and analytic depoissonization.

Recurrences and Functional Equations

Recurrences and functional equations are widely used in computer science. For example, the divide-and-conquer recurrence equations appear in the analysis of searching and sorting algorithms (cf. [41]). Hereafter, we concentrate on recurrences and functional equations that arise in the analysis of digital trees and problems on words.

However, to introduce the reader into the main subject we first consider two well-known functional equations. Let us enumerate the number of *unlabeled binary trees* built over n vertices. Call this number b_n, and let $B(z) = \Sigma_{n=0}^{\infty} b_n z^n$ be its *ordinary generating function.* Since each such a tree is constructed in a recursive manner with left and right subtrees being unlabeled binary trees, we immediately arrive at the following recurrence $b_n = b_0 b_{n-1} + \cdots + b_{n-1} b_0$ for $n \geq 1$ with $b_0 = 1$ by definition. Multiplying by z^n and summing from $n = 1$ to infinity, we obtain $B(z) - 1 = zB^2(z)$ which is a simple functional equation that can be solved to find

$$B(z) = \frac{1 - \sqrt{1 - 4z}}{2z} .$$

To derive the above functional equation, we used a simple fact that the generating function $C(z)$ of the convolution c_n of two sequences, say a_n and b_n (i.e., $c_n = a_0 b_n + a_1 b_{n-1} + \cdots + a_n b_0$), is the product of $A(z)$ and $B(z)$, that is, $C(z) = A(z)B(z)$.

The above functional equation and its solution can be used to obtain an explicit formula on b_n. Indeed, we first recall that $[z^n]B(z)$ denotes the coefficient at z^n of $B(z)$ (i.e., b_n). A standard analysis leads to (cf. [39, 44])

$$b_n = \left[z^n\right] B(z) = \frac{1}{n+1}\binom{2n}{n} ,$$

which is the famous *Catalan number.*

Let us now consider a more challenging example, namely, enumeration of *rooted labeled trees.* Let t_n the number of rooted labeled trees, and $t(z) = \Sigma_{n=0}^{\infty} \frac{t_n}{n!} z^n$ its *exponential generating function.* It is known that $t(z)$ satisfies the following functional equation (cf. [28, 49]) $t(z) = ze^{t(z)}$. To find t_n, we apply *Lagrange's Inversion Formula.* Let $\Phi(u)$ be a formal power series with $[u^0]\Phi(u) \neq 0$, and let $X(z)$ be a solution of $X = z\Phi(X)$. The coefficients of $X(z)$ or in general $\Psi(X(z))$ where Ψ is an arbitrary series can be found

by

$$\begin{aligned} \left[z^n\right] X(z) &= \frac{1}{n}\left[u^{n-1}\right](\Phi(u))^n \ , \\ \left[z^n\right]\Psi(X(z)) &= \frac{1}{n}\left[u^{n-1}\right](\Phi(u))^n\,\Psi'(u) \ . \end{aligned}$$

In particular, an application of the above to $t(z)$ leads to $t_n = n^{n-1}$, and to an interesting formula (which we encounter again in Example 14.14)

$$t(z) = \sum_{n=1}^{\infty} \frac{n^{n-1}}{n!} z^n \ . \tag{14.20}$$

After these introductory remarks, we can now concentrate on certain recurrences that arise in problems on words; in particular in digital trees and shortest common superstring problems. Let x_n be a generic notation for a quantity of interest (e.g., depth, size or path length in a digital tree built over n strings). Given x_0 and x_1, the following three recurrences originate from problems on tries, PATRICIA tries, and digital search trees, respectively (cf. [16, 28, 37, 41, 44, 49, 51, 52, 53]):

$$x_n = a_n + \beta \sum_{k=0}^{n} \binom{n}{k} p^k q^{n-k} \left(x_k + x_{n-k}\right) \ , \quad n \geq 2 \tag{14.21}$$

$$x_n = a_n + \beta \sum_{k=1}^{n-1} \binom{n}{k} p^k q^{n-k} \left(x_k + x_{n-k}\right) - \alpha\left(p^n + q^n\right) x_n \ , \quad n \geq 2 \tag{14.22}$$

$$x_{n+1} = a_n + \beta \sum_{k=0}^{n} \binom{n}{k} p^k q^{n-k} \left(x_k + x_{n-k}\right) \quad n \geq 0 \tag{14.23}$$

where a_n is a known sequence (also called additive term), α and β are some constants, and $p + q = 1$.

To solve this recurrences and to obtain explicit or asymptotic expression for x_n we apply exponential generating function approach. We need to know the following two facts: Let a_n and b_n be sequences with $a(z) = \Sigma_{n=0}^{\infty} \frac{a_n}{n!} z^n$ and $b(z)$ as their exponential generating functions. (Hereafter, we consequently use lower-case letters for exponential generating functions, like $a(z)$, and upper-case letters for ordinary generating functions, like $A(z)$). Then,

- For any integer $h \geq 0$
$$\frac{d^h}{dz^h} a(z) = \sum_{n=0}^{\infty} \frac{a_{n+h}}{n!} z^n \ .$$
- If $c_n = \Sigma_{r=0}^{n} \binom{n}{r} a_r b_{n-r}$, then the exponential generating function $c(z)$ of c_n becomes $c(z) = a(z)b(z)$.

Now, we are ready to attack the above recurrences and show how they can be solved. Let us start with the simplest one, namely (14.21). Multiplying it by z^n, summing up, and taking into account the initial conditions, we obtain

$$x(z) = a(z) + \beta x(zp) e^{zq} + \beta x(zq) e^{zp} + d(z) \ , \tag{14.24}$$

where $d(z) = d_0 + d_1 z$ and d_0 and d_1 depend on the initial condition for $n = 0, 1$. The trick is to introduce the so called **Poisson transform** $\widetilde{X}(z) = x(z)e^{-z}$ which reduces the above functional equation to

$$\widetilde{X}(z) = \widetilde{A}(z) + \beta \widetilde{X}(zp) + \beta \widetilde{X}(z) + d(z)e^{-z} \ . \tag{14.25}$$

Observe that $\widetilde{x}_n$ and x_n are related by $x_n = \Sigma_{k=0}^{n}\binom{n}{k}\widetilde{x}_k$. Using this, and comparing coefficients of $\widetilde{X}(z)$ at z^n we finally obtain

$$x_n = x_0 + n\,(x_1 - x_0) + \sum_{k=2}^{n}(-1)^k\binom{n}{k}\frac{\hat{a}_k + kd_1 - d_0}{1-\beta\left(p^k+q^k\right)}\,, \tag{14.26}$$

where $n![z^n]\widetilde{A}(z) = \widetilde{a}_n := (-1)^n\hat{a}_n$. In fact, $\hat{a}_n$ and a_n form the so called *binomial inverse relations,* i.e.,

$$\hat{a}_n = \sum_{k=0}^{n}\binom{n}{k}(-1)^k a_k\,, \qquad a_n = \sum_{k=0}^{n}\binom{n}{k}(-1)^k\hat{a}_k\,,$$

and $\hat{\hat{a}}_n = a_n$ (cf. [41]).

EXAMPLE 14.10: Average Path Length in a Trie

Let us consider a trie in the Bernoulli model, and estimate the average length, ℓ_n, of the external path length, i.e., $\ell_n = \mathbf{E}[L_n]$ (cf. "Digital Trees"). Clearly, $\ell_0 = \ell_1 = 0$ and for $n \geq 2$

$$\ell_n = n + \sum_{k=0}^{n}\binom{n}{k}p^k q^{n-k}\left(\ell_k + \ell_{n-k}\right)\,.$$

Thus, by (14.26)

$$\ell_n = \sum_{k=2}^{n}(-1)^k\binom{n}{k}\frac{k}{1-p^k-q^k}\,.$$

Below, we shall discuss the asymptotics of ℓ_n (cf. Example 14.15).

Let us now consider recurrence (14.22), which is much more intricate. It has an exact solution only for some special cases (cf. [41, 51, 53]) that we discuss below. We first consider a simplified version of (14.22), namely

$$x_n\left(2^n-2\right) = 2^n a_n + \sum_{k=1}^{n-1}\binom{n}{k}x_k\,.$$

with $x_0 = x_1 = 0$ (for a more general recurrence of this type see [51]). After multiplying by z^n and summing up we arrive at

$$x(z) = \left(e^{z/2}+1\right)x\,(z/2) + a(z) - a_0 \tag{14.27}$$

where $x(z)$ and $a(z)$ are exponential generating functions of x_n and a_n. To solve this recurrence we must observe that after multiplying both sides by $z/(e^z-1)$ and defining

$$\check{X}(z) = x(z)\frac{z}{e^z-1} \tag{14.28}$$

we obtain a new functional equation that is easier to solve, namely

$$\check{X}(z) = \check{X}\,(z/2) + \check{A}(z)\,,$$

where in the above we assume for simplicity $a_0 = 0$. This functional equation is of a similar type as $\widetilde{X}(x)$ considered above. The coefficient $\check{x}_n$ at z^n of $\check{X}(z)$ can also be easily extracted. One must, however, translate coefficients $\check{x}_n$ into the original sequence x_n. In order to accomplish this, let us introduce the Bernoulli polynomials $B_n(x)$ and Bernoulli numbers $B_n = B_n(0)$, that is, $B_n(x)$ is defined as

$$\frac{ze^{tz}}{e^z-1} = \sum_{k=0}^{\infty}B_k(t)\frac{z^k}{k!}\,.$$

Furthermore, we introduce *Bernoulli inverse relations* for a sequence a_n as

$$\check{a}_n = \sum_{k=0}^{n} \binom{n}{k} B_k a_{n-k} \quad \Longleftrightarrow \quad a_n = \sum_{k=0}^{n} \binom{n}{k} \frac{\check{a}_k}{k+1} .$$

One should know that (cf. [41])

$$a_n = \binom{n}{r} q^n \quad \Longleftrightarrow \quad \check{a}_n = \binom{n}{r} q^r B_{n-r}(q)$$

for $0 < q < 1$. For example, for $a_n = \binom{n}{r} q^n$ the above recurrence has a particular simply solution, namely:

$$x_n = \sum_{k=1}^{n} (-1)^k \binom{n}{k} \frac{B_{k+1}(1-q)}{k+1} \frac{1}{2^{k+1}-1} .$$

A general solution to the above recurrence can be found in [51], and it involves $\check{a}_n$.

EXAMPLE 14.11: Unsuccessful Search in PATRICIA

Let us consider the number of trials u_n in an unsuccessful search of a string in a PATRICIA trie constructed over the symmetric Bernoulli model (i.e., $p = q = 1/2$). As in Knuth [41]

$$u_n \left(2^n - 2\right) = 2^n \left(1 - 2^{1-n}\right) + \sum_{k=1}^{n-1} \binom{n}{k} u_k$$

and $u_0 = u_1 = 0$. A simple application of the above derivation leads, after some algebra, to (cf. [53])

$$u_n = 2 - \frac{4}{n+1} + 2\delta_{n0} + \frac{2}{n+1} \sum_{k=2}^{n} \binom{n+1}{k} \frac{B_k}{2^{k-1}-1}$$

where $\delta_{n,k}$ is the Kronecker delta, that is, $\delta_{n,k} = 1$ for $n = k$ and zero otherwise.

We were able to solve the functional equations (14.24) and (14.27) exactly, since we reduce them to a simple functional equation of the form (14.25). In particular, Eq. (14.27) became (14.25), since luckily $e^z - 1 = (e^{z/2} - 1)(e^{z/2} + 1)$, as already pointed out by Knuth [41], but one cannot expect that much luck with other functional equations. Let us consider a general functional equation

$$F(z) = a(z) + b(z) F(\sigma(z)) , \tag{14.29}$$

where $a(z), b(z), \sigma(z)$ are known functions. Formally, iterating this equation we obtain

$$F(z) = \sum_{k=0}^{\infty} a\left(\sigma^{(k)}(z)\right) \prod_{j=0}^{k-1} b\left(\sigma^{(j)}(z)\right) ,$$

where $\sigma^{(k)}(z)$ is the kth iterate of $\sigma(\cdot)$. When applying the above to solve real problems, one must assure the existence of the infinite series involved. In some cases (cf. [19, 38]), we can provide asymptotic solutions to such complicated formulæ by appealing to the Mellin transform which we discuss below in "Mellin Transform and Asymptotics."

Finally, we deal with the recurrence (14.23). Multiplying both sides by $z^n/n!$ and using the above discussed properties of exponential generating functions we obtain for $x(z) = \Sigma_{n \geq 0} x_n \frac{z^n}{n!}$

$$x'(z) = a(z) + x(zp) e^{zq} + x(zq) e^{zp} ,$$

which becomes after the substitution $\widetilde{X}(z) = x(z)e^{-z}$

$$\widetilde{X}'(z) + \widetilde{X}(z) = \widetilde{A}(z) + \widetilde{X}(zp) + \widetilde{X}(zq)\,. \tag{14.30}$$

The above is a differential–functional equation that we did not discuss so far. It can be solved, since a direct translation of coefficients gives $\widetilde{x}_{n+1} + \widetilde{x}_n = \widetilde{a}_n + \widetilde{x}_n(p^n + q^n)$. Fortunately, this is a simple linear recurrence that has an explicit solution. Taking into account $x_n = \Sigma_{k=0}^n \binom{n}{k}\widetilde{x}_k$, we finally obtain

$$x_n = x_0 - \sum_{k=1}^{n}(-1)^k\binom{n}{k}\sum_{i=1}^{k-1}\hat{a}_i\prod_{j=i+1}^{k-1}\left(1 - p^j - q^j\right) = x_0 - \sum_{k=1}^{n}(-1)^k\binom{n}{k}\sum_{i=1}^{k-1}\hat{a}_i\frac{Q_k}{Q_i}\,, \tag{14.31}$$

where $Q_k = \prod_{j=2}^{k}(1 - p^j - q^j)$, and $\hat{a}_n$ is the binomial inverse of a_n as defined above. In passing, we should observe that solutions of the recurrences (14.21), (14.22), and (14.23) have a form of an alternating sum, that is, $x_n = \sum_{k=1}^n (-1)^k\binom{n}{k} f_n$ where f_n has an explicit formula. In "Mellin Transform and Asymptotics," we discuss how to obtain asymptotics of such an alternating sum.

EXAMPLE 14.12: Expected Path Length in a Digital Search Tree

Let ℓ_n be the expected path length in a digital search tree. Then (cf. [21, 41]) for all $n \geq 0$,

$$\ell_{n+1} = n + \sum_{k=1}^{n}\binom{n}{k}p^k q^{n-k}\left(\ell_k + \ell_{n-k}\right)$$

with $\ell_0 = 0$. By (14.31) it has the following solution:

$$\ell_n = \sum_{k=2}^{n}(-1)^k\binom{n}{k}Q_{k-1}$$

where Q_k is defined above.

We were quite lucky when solving the above differential–functional equation, since we could reduce it to a linear recurrence of *first* order. However, this is not any longer true when we consider the so-called b-digital search trees in which one assumes that a node can store up to b strings. Then, the general recurrence (14.23) becomes

$$x_{n+b} = a_n + \beta\sum_{k=0}^{n}\binom{n}{k}p^k q^{n-k}\left(x_k + x_{n-k}\right) \quad n \geq 0,$$

provided $x_0, \ldots, x_{b-1}$ are given. Our previous approach would lead to a linear recurrence of order b that does not possess a nice explicit solution. The "culprit" lies in the fact that the exponential generating function of a sequence $\{x_{n+b}\}_{n=0}^{\infty}$ is the bth derivative of the exponential generating function $x(z)$ of $\{x_n\}_{n=0}^{\infty}$. On the other hand, if one considers *ordinary* generating function $X(z) = \Sigma_{n\geq 0}x_n z^n$, then the sequence $\{x_{n+b}\}_{n=0}^{\infty}$ translates into $z^{-b}(X(z) - x_0 - \cdots - x_{b-1}z^{b-1})$. This observation led Flajolet and Richmond [18] to reconsider the standard approach to the above binomial recurrences, and to introduce ordinary generating function into the play. A careful reader observes, however, that then one must translate into ordinary generating functions sequences such as $s_n = \Sigma_{k=0}^n\binom{n}{k}a_k$ (which were easy under exponential generating functions since they become $a(z)e^z$). But, it is not difficult to see that

$$s_n = \sum_{k=0}^{n}\binom{n}{k}a_k \quad \Longrightarrow \quad S(z) = \frac{1}{1-z}A\left(\frac{z}{1-z}\right)\,.$$

Indeed,

$$\begin{aligned}\frac{1}{1-z}A\left(\frac{z}{1-z}\right) &= \sum_{m=0}^{\infty} a_m z^m \frac{1}{(1-z)^{m+1}} = \sum_{m=0}^{\infty} a_m z_m \sum_{j=0}^{\infty} \binom{m+j}{j} z^j \\ &= \sum_{n=0}^{\infty} z^n \sum_{k=0}^{\infty} \binom{n}{k} a_k \ .\end{aligned}$$

Thus, the above recurrence for $p = q = 1/2$ and any $b \geq 1$ can be translated into ordinary generating functions as

$$\begin{aligned}X(z) &= \frac{1}{1-z}G\left(\frac{z}{1-z}\right) \\ G(z)(1+z)^b &= 2z^b G\,(z/2) + P(z)\ ,\end{aligned}$$

where $P(z)$ is a function of a_n and initial conditions. But, the latter functional equation falls under (14.29) and its solution was already discussed above.

Complex Asymptotics

When analyzing an algorithm we often aim at predicting its rate of growth of time or space complexity for large inputs, n. Precise analysis of algorithms aims at obtaining precise asymptotics and/or full asymptotic expansions of such performance measures. For example, in the previous subsection we studied some parameters of tries (e.g., path length ℓ_n, unsuccessful search u_n, etc.) that depend on input of size n. We observed that these quantities are expressed by some complicated alternating sums (cf. Examples 14.10–14.12). One might be interested in precise rate of growth of these quantities. More precisely, if x_n represents a quantity of interest with input size n, we may look for a simple explicit function a_n (e.g., $a_n = \log n$ or $a_n = \sqrt{n}$) such that $x_n \sim a_n$ (i.e., $\lim_{n\to\infty} x_n/a_n = 1$), or we may be aiming at a very precise asymptotic expansion such as $x_n = a_n^1 + a_n^2 + \cdots + o(a_n^k)$ where for each $1 \leq i \leq k$ we have $a_n^{i+1} = o(a_n^i)$.

The reader is referred to an excellent recent survey by Odlyzko [47] on asymptotic methods. In this subsection, we briefly discuss some elementary facts of asymptotic evaluation, and describe a few useful methods.

Let $A(z) = \Sigma_{n=0}^{\infty} a_n z^n$ be the generating function of a sequence $\{a_n\}_{n=0}^{\infty}$. In the previous subsection, we look at $A(z)$ as a *formal power series.* Now, we ask whether $A(z)$ converges, and what is its region of convergence. It turns out that the radius of convergence for $A(z)$ is responsible for the asymptotic behavior of a_n for large n. Indeed, by *Hadamard's Theorem* [27, 57] we know that radius R of convergence of $A(z)$ (where z is a complex variable) is given by

$$R = \frac{1}{\limsup_{n\to\infty} |a_n|^{1/n}}\ .$$

In other words, for every $\varepsilon > 0$ there exists N such that for $n > N$ we have

$$|a_n| \leq \left(R^{-1} + \varepsilon\right)^n\ ;$$

for infinitely many n we have

$$|a_n| \geq \left(R^{-1} - \varepsilon\right)^n\ .$$

Informally saying, $\frac{1}{n}\log|a_n| \sim 1/R$, or even less formally the exponential growth of a_n is determined by $(1/R)^n$. In summary, singularities of $A(z)$ determine asymptotic behavior of its coefficients for large n. In fact, from *Cauchy's Integral Theorem* (cf. "Review From Complex Analysis") we know that

$$|a_n| \leq \frac{M(r)}{r^n}$$

where $M(r)$ is the maximum value of $|A(z)|$ over a circle of radius $r < R$.

Our goal now is to make a little more formal our discussion above, and deal with multiple singularities. We restrict ourselves to *meromorphic* functions $A(z)$, i.e., ones that are analytic with the exception of a finite number of *poles.* To make our discussion even more concrete we study the following function

$$A(z) = \sum_{j=1}^{r} \frac{a_{-j}}{(z-\rho)^j} + \sum_{j=0}^{\infty} a_j (z-\rho)^j \ .$$

More precisely, we assume that $A(z)$ has the above *Laurent expansion* around a pole ρ of multiplicity r. Let us further assume that the pole ρ is the closest to the origin, that is, $R = |\rho|$ (and there are no more poles on the circle of convergence). In other words, the sum of $A(z)$ which we denote for simplicity as $A_1(z)$, is analytic in the circle $|z| \le |\rho|$, and its possible radius of convergence $R' > |\rho|$. Thus, coefficients a'_n of $A_1(n)$ are bounded by $|a'_n| = O((1/R' + \varepsilon)^n)$ for any $\varepsilon > 0$. Let us now deal with the first part of $A(z)$. Using the fact that $[z^n](1-z)^{-r} = \binom{n+r-1}{r-1}$ for a positive integer r, we obtain

$$\begin{aligned}
\sum_{j=1}^{r} \frac{a_j}{(z-\rho)^j} &= \sum_{j=1}^{r} \frac{a_j (-1)^j}{\rho^j \, (1 - z/\rho)^j} \\
&= \sum_{j=1}^{r} (-1)^j a_j \rho^{-j} \sum_{n=0}^{\infty} \binom{n+j-1}{n} \left(\frac{z}{\rho}\right)^n \\
&= \sum_{n=1}^{\infty} z^n \sum_{j=1}^{r} (-1)^j a_j \binom{n}{j-1} \rho^{-(n+j)} \ .
\end{aligned}$$

In summary, we prove that

$$\left[z^n\right] A(z) = \sum_{j=1}^{r} (-1)^j a_j \binom{n}{j-1} \rho^{-(n+j)} + O\left(\left(1/R' + \varepsilon\right)^n\right)$$

for $R' > \rho$ and any $\varepsilon > 0$.

EXAMPLE 14.13: Frequency of a Given Pattern Occurrence

Let H be a *given* pattern of size m, and consider a random text of length n generated according to the Bernoulli model. An old and well-studied problem of pattern matching (cf. [15]) asks for an estimation of the number O_n of pattern H occurrences in the text. Let $T_r(z) = \Sigma_{n=0}^{\infty} \mathsf{Pr}\{O_n = r\} z^n$ denote the generating function of $\mathsf{Pr}\{O_n = r\}$ for $|z| \le 1$. It can be proved (cf. [24, 26]) that

$$T_r(z) = \frac{z^m P(H) (D(z) + z - 1)^{r-1}}{D^{r+1}(z)} \ ,$$

where $D(z) = P(H) z^m + (1-z) A_H(z)$ and $A(z)$ is the so-called *autocorrelation polynomial* (a polynomial of degree m). It is also easy to see that there exists smallest $\rho > 1$ such that $D(\rho) = 0$. Then, an easy application of the above analysis leads to

$$\mathsf{Pr}\left\{O_n(H) = r\right\} = \sum_{j=1}^{r+1} (-1)^j a_j \binom{n}{j-1} \rho^{-(n+j)} + O\left(\rho_1^{-n}\right)$$

where $\rho_1 > \rho$ and $a_{r+1} = \rho^m P(H) \, (\rho - 1)^{r-1} \left(D'(\rho)\right)^{-r-1}$.

The method just described can be called the method of *subtracted singularities,* and its general description follows: Imagine that we are interested in the asymptotic formula for coefficients a_n of a function $A(z)$, whose circle of convergence is R. Let us also assume that we can find a simpler function, say $\bar{A}(z)$ that has the same singularities as $A(z)$ (e.g., in the example above $\bar{A}(z) = \Sigma_{j=1}^{r} \frac{a_{-j}}{(z-\rho)^j}$). Then, $A_1(z) = A(z) - \bar{A}(z)$ is analytical in a larger disk, of radius $R' > R$, say, and its coefficients are not dominant in an asymptotic sense. To apply this method successfully, we need to develop asymptotics of some known functions (e.g., $(1-z)^\alpha$ for any real α) and establish the so called *transfer theorems* (cf. [17]). This leads us to the so-called *singularity analysis* of Flajolet and Odlyzko [17], which we discuss next.

We start with the observation that $[z^n]A(z) = \rho^n [z^n]A(z/\rho)$, that is, we need only to study singularities at, say, $z = 1$. The next observation deals with asymptotics of $(1-z)^{-\alpha}$. Above, we show how to obtain coefficients at z^n of this function when α is a natural number. Then, the function $(1-z)^{-\alpha}$ has a pole of order α at $z = 1$. However, when $\alpha \neq 1, 2 \ldots$, then the function has an *algebraic singularity* (in fact, it is then a multivalued function). Luckily enough, we can proceed formally as follows:

$$\begin{aligned} \left[z^n\right](1-z)^{-\alpha} &= \binom{n+\alpha-1}{n} = \frac{\Gamma(n+\alpha)}{\Gamma(\alpha)\Gamma(n+1)} \\ &= \frac{n^{\alpha-1}}{\Gamma(\alpha)}\left(1 + \frac{\alpha(\alpha-1)}{2n} + O\left(\frac{1}{n^2}\right)\right), \end{aligned}$$

provided $\alpha \notin \{0, -1, -2, \ldots\}$. In the above, $\Gamma(x) = \int_0^\infty e^{-t}x^{t-1}dx$ is the Euler gamma function (cf. [1, 27]), and the latter asymptotic expansion follows from the Stirling formula. Even more generally, let

$$A(z) = (1-z)^{-\alpha}\left(\frac{1}{z}\log\frac{1}{1-z}\right)^\beta .$$

Then, as shown by Flajolet and Odlyzko [17],

$$a_n = \left[z^n\right]A(z) = \frac{n^{\alpha-1}}{\Gamma(\alpha)}\left(1 + C_1\frac{\beta}{\log n} + C_2\frac{\beta(\beta-1)}{2\log^2 n} + O\left(\frac{1}{\log^3 n}\right)\right), \tag{14.32}$$

provided $\alpha \notin \{0, -1, -2, \ldots\}$, and C_1 and C_2 are constants that can be calculated explicitly.

The most important aspect of the singularity theory comes next: In many instances we do *not* have an explicit expression for the generating function $A(z)$ but only an expansion of $A(z)$ around a singularity. For example, let $A(z) = (1-z)^{-\alpha} + O(B(z))$. In order to pass to coefficients of a_n we need a "transfer theorem" that will allow us to pass to coefficients of $B(z)$ under the "Big Oh" notation. We shall discuss them below.

We need a definition of Δ-analyticity around the singularity $z = 1$:

$$\Delta = \left\{z : |z| < R\,,\, z \neq 1\,,\, \left|\arg(z-1)\right| > \phi\right\}$$

for some $R > 1$ and $0 < \phi < \pi/2$ (i.e., the domain Δ is an extended disk around $z = 1$ with a circular part rooted at $z = 1$ deleted). Then (cf. [17]):

THEOREM 14.8 *Let $A(z)$ be Δ-analytic that satisfies in a neighborhood of $z = 1$ either*

$$A(z) = O\left((1-z)^{-\alpha}\log^\beta(1-z)^{-1}\right)$$

or

$$A(z) = o\left((1-z)^{-\alpha}\log^\beta(1-z)^{-1}\right).$$

Then, either

$$\left[z^n\right] = O\left(n^{\alpha-1}\log^\beta n\right)$$

or

$$[z^n] = o\left(n^{\alpha-1}\log^{\beta} n\right) ,$$

respectively.

A classical example of singularity analysis is the Flajolet and Odlyzko analysis of the height of binary trees (cf. [17]), however, we finish this subsection with a simpler application that quite well illustrates the theory (cf. [55]).

EXAMPLE 14.14: Certain Sums from Coding Theory

In coding theory the following sum is of some interest:

$$S_n = \sum_{i=0}^{n} \binom{n}{i} \left(\frac{i}{n}\right)^i \left(1-\frac{i}{n}\right)^{n-i} .$$

Let $s_n = n^n S_n$. If $s(z)$ denotes the exponential generating function of s_n, then by a simple application of the convolution principle of exponential generating functions we obtain $s(z) = (b(z))^2$ where $b(z) = (1-t(z))^{-1}$ and $t(z)$ is the "tree function" defined in "Recurrences and Functional Equations" (cf. (14.20)). In fact, we already know that this function also satisfies the functional equation $t(z) = ze^{t(z)}$. One observes that $z = e^{-1}$ is the singularity point of $t(z)$, and (cf. [55])

$$\begin{aligned} t(z) - 1 &= -\sqrt{2(1-ez)} + \frac{2}{3}(1-ez) - \frac{11\sqrt{2}}{36}(1-ez)^{3/2} \\ &\quad + \frac{43}{135}(1-ez)^2 + O\left((1-ez)^{5/2}\right) , \\ s(z) &= \frac{1}{2h(z)\left(1+\frac{\sqrt{2}}{3}\sqrt{h(z)}+\frac{11}{36}h(z)+O\left(h^{3/2}(z)\right)\right)^2} \\ &= \frac{1}{2(1-ez)} + \frac{\sqrt{2}}{3\sqrt{(1-ez)}} + \frac{1}{36} + \frac{\sqrt{2}}{540}\sqrt{1-ez} + O(1-ez) . \end{aligned}$$

Thus, an application of the singularity analysis leads finally to the following asymptotic expansion

$$S_n = \sqrt{\frac{n\pi}{2}} + \frac{2}{3} + \frac{\sqrt{2\pi}}{24}\frac{1}{\sqrt{n}} - \frac{4}{135}\frac{1}{n} + O\left(1/n^{3/2}\right) .$$

For more sophisticated examples the reader is referred to [17, 21, 55].

Mellin Transform and Asymptotics

In previous sections, we study functional equations such as (14.25) or more generally (14.30). They can be summarized by the following general functional equation:

$$f^{(b)}(z) = a(z) + \alpha f(zp) + \beta f(zq) , \tag{14.33}$$

where $f^{(b)}(z)$ denotes the bth derivative of $f(z)$, α, β are constants, and $a(z)$ is a known function. An important point to observe is that in the applications described so far the unknown function $f(z)$ was usually a Poisson transform, that is, $\widetilde{f}(z) = \Sigma_{n\geq 0} f_n \frac{z^n}{n!} e^{-z}$. We briefly discuss consequences of this point at the end of this subsection where some elementary *depoissonization* results will be presented. An effective approach to solve asymptotically (either for $z \to 0$ or $z \to \infty$) the above function equation is by the so

called **Mellin transform** which we discuss next. Knuth [41], together with De Bruijn, is responsible for introducing the Mellin transform into the "orbit" of the average case analysis of algorithms; however, it was popularized by Flajolet and his school who applied Mellin transforms to "countably" many problems of analysis of algorithms and analytic combinatorics. We base this subsection mostly on a survey by Flajolet et al. [19].

For a function $f(x)$ on $x \in [0, \infty)$ we define the Mellin transform as

$$\mathcal{M}(f, s) = f^*(s) = \int_0^\infty f(x) x^{s-1} dx \ ,$$

where s is a complex number. For example, observe that from the definition of the Euler gamma function, we have $\Gamma(s) = \mathcal{M}(e^x, s)$. The Mellin transform is a special case of the Laplace transform (set $x = e^t$) or the Fourier transform (set $x = e^{i\omega}$). Therefore, using the inverse Fourier transform, one establishes the inverse Mellin transform (cf. [27]), namely,

$$f(x) = \frac{1}{2\pi i} \int_{c-i\infty}^{c+i\infty} f^*(s) x^{-s} ds \ ,$$

provided $f(x)$ is piecewise continuous. In the above, the integration is along a vertical line $\Im(s) = c$, and c must belong to the so-called fundamental strip where the Mellin transform exists (see properly (P1) below).

The usefulness of the Mellin transform to the analysis of algorithms is a consequence of a few properties that we discuss in the sequel.

(P1) Fundamental Strip
Let $f(x)$ be a piecewise continuous function on the interval [0,∞) such that

$$f(x) = \begin{cases} O\left(x^\alpha\right) & x \to 0 \\ O\left(x^\beta\right) & x \to \infty \ . \end{cases}$$

Then the Mellin transform of $f(x)$ exists for any complex number s in the **fundamental strip** $-\alpha < \Re(s) < -\beta$, which we will denote $\langle -\alpha; -\beta \rangle$.

(P2) Smallness of Mellin transforms
Let $s = \sigma + it$. By the Riemann–Lebesgue lemma

$$f^*(\sigma + it) = o\left(|t|^{-r}\right) \quad \text{as } t \to \pm\infty$$

provided $f \in \mathcal{C}^r$ where $\mathcal{C}^r$ is the set of functions having continuous r derivatives.

(P3) Basic Functional Properties
The following holds in appropriate fundamental strips:

$$\begin{aligned}
f(\mu x) &\quad \Leftrightarrow \quad \mu^{-s} f^*(s) \quad (\mu > 0) \\
f(x^\rho) &\quad \Leftrightarrow \quad \frac{1}{\rho} f^*(s/\rho) \quad (\rho > 0) \\
\frac{d}{dx} f(x) &\quad \Leftrightarrow \quad -(s-1) f^*(s-1) \\
\int_0^x f(t) dt &\quad \Leftrightarrow \quad -\frac{1}{s} f^*(s+1) \\
f(x) = \sum_{k \geq 0} \lambda_k g(\mu_k x) &\quad \Leftrightarrow \quad f^*(s) = g^*(s) \sum_{k \geq 0} \lambda_k \mu_k^{-s} \quad \text{(Harmonic Sum Rule)}
\end{aligned}$$

(P4) Asymptotics for $x \to 0$ and $x \to \infty$

Let the fundamental strip of $f^*(s)$ be the set of all s such that $-\alpha < \Re(s) < -\beta$ and assume that for $s = \sigma + i\tau$, $f^*(s) = O(|s|^r)$ with $r > 1$ as $|s| \to \infty$. If $f^*(s)$ can be analytically continued to a meromorphic function for $-\beta \le \Re(s) \le M$ with finitely many poles λ_k such that $\Re(\lambda_k) < M$, then as $x \to \infty$,

$$F(x) = -\sum_{\lambda_k \in \mathcal{H}} \operatorname{Res}\left\{F^*(s)x^{-s}, s = \lambda_k\right\} + O\left(x^{-M}\right) \quad x \to \infty$$

where M is as large as we want. (In a similar fashion one can continue the function $f^*(s)$ to the left to get an asymptotic formula for $x \to 0$.)

Sketch of a Proof. Consider the rectangle R with the corners at $c - iA$, $M - iA$, $M + iA$, and $c + iA$. Choose A so that the sides of R do not pass through any singularities of $F^*(s)x^{-s}$. When evaluating

$$\lim_{A\to\infty}\int_R = \lim_{A\to\infty}\left(\int_{c-iA}^{c+iA} + \int_{c+iA}^{M+iA} + \int_{M+iA}^{M-iA} + \int_{M-iA}^{c-iA}\right),$$

the second and fourth integrals contribute very little, since $F^*(s)$ is small for s with a large imaginary part by property (P2). The contribution of the fourth integral is computed as follows:

$$\left|\int_{M+i\infty}^{M-i\infty} F^*(s)x^{-s}ds\right| = \left|\int_{\infty}^{-\infty} F^*(M+it)x^{-M-it}dt\right| \le \left|x^{-M}\right| \int_{\infty}^{-\infty} \left|F^*(M+it)\right| \left|x^{-it}\right| dt\,.$$

But the last integrand decreases exponentially as $|t| \to \infty$, thus, giving a contribution of $O(x^{-M})$. Finally, using Cauchy's residue theorem and taking into account the negative direction of R, we have

$$-\sum_{\lambda_k \in \mathcal{H}} Res\left\{F^*(s)x^{-s}, s = \lambda_k\right\} = \frac{1}{2i\pi}\int_{c-i\infty}^{c+i\infty} F^*(s)x^{-s}ds + O\left(x^{-M}\right),$$

which proves the desired result.

Specifically, the above implies that if the above smallness condition on $f^*(s)$ is satisfied for $-\beta < \Re(s) \le M$, $(M > 0)$, then

$$f^*(s) = \sum_{k=0}^{K} \frac{d_k}{(s-b)^{k+1}}\,, \tag{14.34}$$

leads to

$$f(x) = -\sum_{k=0}^{K} \frac{d_k}{k!}x^{-b}(-\log x)^k + O\left(x^{-M}\right) \quad x \to \infty\,. \tag{14.35}$$

In a similar fashion, if for $-M < \Re(s) < -\alpha$ the smallness condition of $f^*(s)$ holds and

$$f^*(s) = \sum_{k=0}^{K} \frac{d_k}{(s-b)^{k+1}}\,, \tag{14.36}$$

then

$$f(x) = \sum_{k=0}^{K} \frac{d_k}{k!}x^{-b}(-\log x)^k + O\left(x^{M}\right) \quad x \to 0\,. \tag{14.37}$$

(P5) Mellin Transform in The Complex Plane (cf. [19, 33])

If $f(z)$ is analytic in a cone $\theta_1 \le \arg(z) \le \theta_2$ with $\theta_1 < 0 < \theta_2$, then the Mellin transform $f^*(s)$ can be defined by replacing the path of the integration $[0, \infty[$ by any curve starting at $z = 0$ and going to ∞

inside the cone, and it is identical with the real transform $f^*(s)$ of $f(z) = F(z)\Big|_{z\in\mathbb{R}}$. In particular, if $f^*(s)$ fulfills an asymptotic expansion such as (14.34) or (14.36), then (14.35) or (14.37) for $f(z)$ holds in $z\to\infty$ and $z \to 0$ in the cone, respectively.

Let us now apply Mellin transforms to some problems studies above. For example, consider a trie for which the functional equation (14.25) becomes

$$\widetilde{X}(z) = \widetilde{A}(z) + \widetilde{X}(zp) + \widetilde{X}(zq) ,$$

where $p + q = 1$ and $\widetilde{A}(z)$ is the Poisson transform of a known function. Thanks to property (P3) the Mellin transform translates the above functional equation to an algebraic one which can be immediately solved, resulting in

$$X^*(s) = \frac{A^*(s)}{1 - p^{-s} - q^{-s}} ,$$

provided there exists a fundamental strip for $X^*(s)$ where also $A^*(s)$ is well defined. Now, due to property (P4) we can easily compute asymptotics of $\widetilde{X}(z)$ as $z \to \infty$ in a cone. More formally, we obtain asymptotics for z real, say x, and then either analytically continue our results or apply property (P5) which basically says that there is a cone in which the asymptotic results for real x can be extended to a complex z. Examples of usage of this technique can be found in [21, 28, 30, 32, 33, 38, 41, 44].

This is a good plan to attack problems as the above; however, one must translate asymptotics of the Poisson transform $\widetilde{X}(z)$ into the original sequence, say x_n. One would like to have $x_n \sim \widetilde{X}(n)$, but this is not true in general (e.g., take $x_n = (-1)^n$). To assure the above asymptotic equivalence, we briefly discuss the so called **depoissonization** [30, 32, 33]. We cite below only one result that found many applications in the analysis of algorithms (cf. [32]).

THEOREM 14.9 *Let $\tilde{X}(z) = \Sigma_{n\geq 0} x_n \frac{z^n}{n!} e^{-z}$ be the Poisson transform of a sequence x_n that is assumed to be an entire function of z. We postulate that in a cone S_θ ($\theta < \pi/2$) the following two conditions simultaneously hold for some real numbers A, B, $R > 0$, β, and $\alpha < 1$:*

(I) *For $z \in S_\theta$*

$$|z| > R \quad \Rightarrow \quad \left|\widetilde{X}(z)\right| \leq B|z|^\beta \Psi(|z|) ,$$

where $\Psi(x)$ is a slowly varying function, that is, such that for fixed t $\lim_{x\to\infty} \frac{\Psi(tx)}{\Psi(x)} = 1$ (e.g., $\Psi(x) = \log^d x$ for some $d > 0$);

(O) *For $z \notin S_\theta$*

$$|z| > R \quad \Rightarrow \quad \left|\widetilde{X}(z)e^z\right| \leq A \exp(\alpha|z|) .$$

Then,

$$x_n = \widetilde{X}(n) + O\left(n^{\beta-1}\Psi(n)\right) \tag{14.38}$$

or more precisely:

$$x_n = \widetilde{X}(n) - \frac{1}{2}\widetilde{X}''(n) + O\left(n^{\beta-2}\Psi(n)\right) .$$

where $\widetilde{X}''(n)$ is the second derivative of $\widetilde{X}(z)$ at $z = n$.

The verification of conditions (I) and (O) is usually not too difficult, and can be accomplished directly on the functional equation at hand through the so called *increasing domains* method discussed in [32].

Finally, we should say that there is an easier approach to deal with a majority of functional equations of type (14.25). As we pointed out, such equations possess solutions that can be represented as alternating sums (cf. (14.26) and Examples 14.10–14.12). Let us consider a general alternating sum

$$S_n = \sum_{k=m}^{n} (-1)^k \binom{n}{k} f_k$$

where f_k is a known, but otherwise, general sequence. The following two equivalent approaches (cf. [41, 52]) use complex integration (the second one is actually a Mellin-like approach) to simplify the computations of asymptotics of S_n for $n \to \infty$ (cf. [39, 52]).

THEOREM 14.10 **(Rice's Formula)** (i) *Let $f(s)$ be an analytical continuation of $f(k) = f_k$ that contains the half line $[m, \infty)$. Then,*

$$S_n := \sum_{k=m}^{n} (-1)^k \binom{n}{k} f_k = \frac{(-1)^n}{2\pi i} \int_C f(s) \frac{n!}{s(s-1)\cdots(s-n)} ds$$

where C is a positively enclosed curve that encircles $[m, n]$ and does not include any of the integers $0, 1, \ldots, m-1$.
(ii) Let $f(s)$ be analytic left to the vertical line $(\frac{1}{2} - m - i\infty, \frac{1}{2} - m + i\infty)$ with subexponential growth at infinity, then

$$\begin{aligned} S_n &= \frac{1}{2\pi i} \int_{\frac{1}{2}-m-i\infty}^{\frac{1}{2}-m+i\infty} f(-z) B(N+1, z) dz \\ &= \frac{1}{2\pi i} \int_{\frac{1}{2}-m-i\infty}^{\frac{1}{2}-m+i\infty} f(-z) n^{-z} \Gamma(z) \\ &\quad \left(1 - \frac{z(z+1)}{2n} + \frac{z(1+z)}{24n^2}(3(1+z)^2 + z - 1) + O\left(n^{-3}\right)\right) dz \end{aligned}$$

where $B(x, y) = \Gamma(x)\Gamma(y)/\Gamma(x+y)$ is the Beta function.

The precise growth condition for $f(z)$ of part (ii) can be found in [52].

EXAMPLE 14.15: Asymptotics of Some Alternating Sums

In Examples 14.10–14.12 we deal with alternating sums of the following general type:

$$S_n(r) = \sum_{k=2}^{n} (-1)^k \binom{n}{k} \binom{k}{r} \frac{1}{p^{-k} - q^{-k}} ,$$

where $p + q = 1$. We now use Theorem 14.10 to obtain asymptotics of S_n as n becomes large and r is fixed. To simplify our computation we use part (ii) of the above theorem, which leads to

$$S_n(r) = \frac{1}{2\pi i} \frac{(-1)^n}{r!} \int_{\frac{1}{2}-[2-r]^+-i\infty}^{\frac{1}{2}-[2-r]^+ + i\infty} n^{r-z} \Gamma(z) \frac{1}{1 - p^{r-z} - q^{r-z}} dz + e_n .$$

$x^+ = \max\{0, x\}$, where e_n is an error term that we discuss later. The above integral should remind the reader of the integral appearing in the inverse Mellin transform. Thus, we can estimate it using a similar approach. First of all, we observe that the function under the integral has infinitely many poles satisfying

$$1 = p^{r-z} + q^{r-z} .$$

It can be proved (cf. [32]) that these poles, say z_k for $k = 0, \pm 1, \ldots$, lie on a line $\Re(z) = r - 1$ provided $\log p / \log q$ is rational, which we assume to hold. Thus, we can write $z_k = r - 1 + i y_k$ where $y_0 = 0$ and otherwise a real number for $k \neq 0$. Observe also that the line at $\Re(z) = r - 1$ lies *right* to the line of integration $(\frac{1}{2} - [2-r]^+ - i\infty, \frac{1}{2} - [2-r]^+ + i\infty)$. To take advantages of the Cauchy residue theorem, we consider a big rectangle with left side being the line of integration, the right size position at $\Re(z) = M$

(where M is a large number), and bottom and top side position at $\Im(z) = \pm A$, say. We further observe that the right side contributes only $O(n^{r-M})$ due to the factor n^{r-M} in the integral. Both bottom and top sides contribute negligible amounts too, since the gamma function decays exponentially fast with the increase of imaginary part (i.e., when $A \to \infty$). In summary, the integral is equal to a circular integral (around the rectangle) plus a negligible part $O(n^{r-M})$. But, then by Cuachy's residue theorem the latter integral is equal to minus the sum of all residues at z_k, that is,

$$S_n(r) = -\sum_{k-\infty}^{\infty} \operatorname{Res}\left(\frac{n^{r-z}\Gamma(z)}{1-p^{r-z}-q^{r-z}},\ z = z_k\right) + O\left(n^{r-M}\right) .$$

We can compute the residues using MAPLE (as shown in "Review From Complex Analysis"). Equivalently, for $k = 0$ (the main contribution to the asymptotics comes from $z_0 = r - 1$) we can use the following expansions around $w = z - z_0$:

$$\begin{aligned} n^{r-z} &= n\left(1 - w\ln n + O\left(w^2\right)\right) , \\ \left(1 - p^{r-z} - q^{r-z}\right)^{-1} &= -w^{-1}h^{-1} + \frac{1}{2}h_2h^{-2} + O(w) , \\ \Gamma(z) &= (-1)^{r+1}\left(w^{-1} - \gamma + \delta_{r,0}\right) + O(w) \quad r = 0, 1 \end{aligned}$$

where $h = -p\ln p - q\ln q$, $h_2 = p\ln^2 p + q\ln^2 q$, and $\gamma = 0.577215\ldots$ is the Euler constant and $\delta_{r,0}$ is the Kronecker symbol. Considering in addition the residues coming from z_k for $k \neq 0$ we finally arrive at

$$S_n(r) = \begin{cases} \frac{1}{h}n\left(\ln n + \gamma - \delta_{r,0} + \frac{1}{2}h_2\right) + (-1)^r n P_r(n) + e_n & r = 0, 1 \\ n\frac{(-1)^r}{r(r-1)h} + (-1)^r n P_r(n) + e_n & r \geq 2 \end{cases}$$

where the error term can be computed easily to be $e_n = O(1)$ (using the arguments as above and observing that the error term has a similar integral representation but with term n^{-1} in front of it). In the above $P_r(n)$ is a contribution from z_k for $k \neq 0$, and it is a fluctuating function with small amplitude. For example, when $p = q = 1/2$, then

$$P_r(n) = \frac{1}{\ln 2}\sum_{k\neq 0}\Gamma\left(r + 2\pi ik/\log 2\right)\exp\left(-2\pi ik\log_2 n\right)$$

is a periodic function of $\log x$ with period 1, mean 0 and amplitude $\leq 10^{-6}$ for $r = 0, 1$.

14.6 Research Issues and Summary

In this chapter we presented a brief overview of probabilistic and analytic methods of the average-case analysis of algorithms. Among probabilistic methods we discussed the inclusion–exclusion principle, first and second moments methods, subadditive ergodic theorem, entropy and asymptotic equipartition property, central limit theorems, large deviations, and Azuma's type inequality. Analytic tools discussed in this chapter are recurrences, functional equations, complex asymptotics, Mellin transform, and analytic depoissonization. These techniques were illustrated in examples that arose in the design and analysis of algorithms on words.

We can trace back probabilistic techniques to the 1960 famous paper of Erdös and Rènyi "On the Evolution of Random Graphs." The analytic approach "was born" in 1963 when Knuth analyzed successfully the average numbers of probes in the linear hashing. Since then we have witnessed an increasing interest in the average-case analysis and design of algorithms, possibly due to the high success rate of randomized

algorithms for computational geometry, scientific visualization, molecular biology, etc. We now see the emergence of combinatorial and asymptotic methods that permit us to classify data structures into broad categories that are susceptible to a unified treatment. Probabilistic methods that have been so successful in the study of random graphs and hard combinatorial optimization problems also play an important role in the field. These developments have two important consequences for the analysis of algorithms: it becomes possible to predict average-case behavior under more *general* probabilistic models, at the same time it becomes possible to analyze much more structurally *complex* algorithms.

14.7 Defining Terms

Asymptotic equipartition property: A set of all strings of a given length can be partitioned into a set of "bad states" of a low probability, and a set of "good states" such that every string in the latter set has approximately the same probability.

Digital trees: Trees that store digital information (e.g., strings, keys, etc.). There are several types of digital trees, namely: **tries**, **PATRICIA tries**, **digital search trees**, and **suffix trees**.

Edit distance: The smallest number of insertions/deletions/substitutions required to change one string onto another.

Inclusion–exclusion principle: A rule that allows to compute the probability of *exactly r* occurrences of events $A_1, A_2, \ldots, A_n$.

Large deviations: When away by more than the standard deviation from the mean, apply the large deviation principle!

Meromorphic function: A function that is analytic, except for a finite number of poles (at which the function ceases to be defined).

Poissonization and depoissonization: When a deterministic input is replaced by a Poisson process, then the new model is called the Poisson model. After finding a solution to the Poisson model, one must interpret it in terms of the original problem, i.e., depoissonize the Poisson model.

Probabilistic models: Underlying probabilistic models that determine the input distribution (e.g., of generated strings). We discussed the **Bernoulli model,** the **Markovian model,** and the **mixing model**.

Rice's method: A method of complex asymptotics that can handle certain alternating sums arising in the analysis of algorithms.

Shortest common superstring: A shortest possible string that contains as substrings a number of given strings.

Singularity analysis: A complex asymptotic technique for determining the asymptotics of certain algebraic functions.

Subadditive ergodic theorem: If a stationary and ergodic process satisfies the subadditive inequality, then it grows almost surely linearly in time.

Acknowledgment

The author thanks his colleagues P. Jacquet, J. Kieffer, G. Louchard, H. Prodinger, and K. Park for reading earlier versions of this chapter, and for comments that led to improvements in the presentation.

References

[1] Abramowitz, M. and Stegun, I., *Handbook of Mathematical Functions,* Dover, New York, 1964.

[2] Aho, A., Hopcroft, J., and Ullman, J., *The Design and Analysis of Computer Algorithms,* Addison-Wesley, Reading, MA, 1974.
[3] Alon, N. and Spencer, J., *The Probabilistic Method,* John Wiley & Sons, New York, 1992.
[4] Arratia, R. and Waterman, M., A Phase Transition for the Score in Matching Random Sequences Allowing Deletions, *Annals of Applied Probability,* 4, 200–225, 1994.
[5] Arratia, R., Gordon, L., and Waterman, M., The Erdös-Rényi Law in Distribution for Coin Tossing and Sequence Matching, *Annals of Statistics,* 18, 539–570, 1990.
[6] Bollobás, B., *Random Graphs,* Academic Press, London, 1985.
[7] Blum, A., Jiang, T., Li, M., Tromp, J., and Yannakakis, M., Linear Approximation of Shortest Superstring, *J. the ACM,* 41, 630–647, 1994.
[8] Bradely, R., Basic Properties of Strong Mixing Conditions, in *Dependence in Probability and Statistics* E. Eberlein and M. Taqqu, Eds., 165–192, 1986.
[9] Chvatal V. and Sankoff, D., Longest Common Subsequence of Two Random Sequences, *J. Appl. Prob.,* 12, 306–315, 1975.
[10] Coffman, E. and Lueker, G., *Probabilistic Analysis of Packing and Partitioning Algorithms,* John Wiley & Sons, New York, 1991.
[11] Cover, T.M. and Thomas, J.A., *Elements of Information Theory,* John Wiley & Sons, New York, 1991.
[12] Crochemore, M. and Rytter, W., *Text Algorithms,* Oxford University Press, New York, 1995.
[13] Dembo, A. and Zeitouni, O., *Large Deviations Techniques,* Jones and Bartlett, Boston, 1993.
[14] Durrett, R., *Probability: Theory and Examples,* Wadsworth, Belmont, CA, 1991.
[15] Feller, W., *An Introduction to Probability Theory and its Applications,* Vol. II, John Wiley & Sons, 1971.
[16] Flajolet, P., Analytic Analysis of Algorithms, *Lectures Notes in Computer Science,* Vol. 623, W. Kuich, Ed., 186–210, Springer-Verlag, 1992.
[17] Flajolet, P. and Odlyzko, A., Singularity Analysis of Generating Functions, *SIAM J. Disc. Methods,* 3, 216–240, 1990.
[18] Flajolet, P. and Richmond, B., Generalized Digital Trees and Their Difference-Differential Equations, *Random Structures and Algorithms,* 3, 305–320, 1992.
[19] Flajolet, P., Gourdon, X., and Dumas, P., Mellin Transforms and Asymptotics: Harmonic Sums, *Theoretical Computer Science,* 144, 3–58, 1995.
[20] Flajolet, P. and Sedgewick, R., Mellin Transforms and Asymptotics: Finite Differences and Rice's Integrals. *Theoretical Computer Science,* 144, 101–124, 1995.
[21] Flajolet, P. and Sedgewick, R., *Analytical Combinatorics,* in preparation (available also at `http://pauillac.inria.fr/algo/flajolet/Publications/publist.html`).
[22] Frieze, A. and McDiarmid, C., Algorithmic Theory of Random Graphs, *Random Structures & Algorithms,* 10, 5–42, 1997.
[23] Frieze, A. and Szpankowski, W., Greedy Algorithms for the Shortest Common Superstring That Are Asymptotically Optimal, *Algorithmica,* 21, 21–36, 1998.
[24] Fudos, I., Pitoura, E., and Szpankowski, W., On Pattern Occurrences in a Random Text, *Information Processing Letters,* 57, 307–312, 1996.
[25] Greene, D.H. and Knuth, D.E., *Mathematics for the Analysis of Algorithms,* Birkhauser, 1981.
[26] Guibas, L. and Odlyzko, A.M., String Overlaps, Pattern Matching, and Nontransitive Games, *J. Combin.Theory Ser. A,* 30, 183–208, 1981.
[27] Henrici, P., *Applied and Computational Complex Analysis,* Vols. 1–3, John Wiley & Sons, 1977.
[28] Hofri, M., *Analysis of Algorithms. Computational Methods and Mathematical Tools,* Oxford University Press, New York, 1995.
[29] Hwang, H-K., Large Deviations for Combinatorial Distributions I: Central Limit Theorems, *Ann. Appl. Probab.,* 6, 297–319, 1996.

[30] Jacquet, P. and Régnier, M., Normal Limiting Distribution of the Size of Tries, *Proc. Performance'87,* 209–223, North Holland, Amsterdam, 1987.
[31] Jacquet, P. and Szpankowski, W., Autocorrelation on Words and Its Applications. Analysis of Suffix Trees by String-Ruler Approach, *J. Combin.Theory Ser. A,* 66, 237–269, 1994.
[32] Jacquet, P. and Szpankowski, W., Asymptotic Behavior of the Lempel-Ziv Parsing Scheme and Digital Search Trees, *Theoretical Computer Science,* 144, 161–197, 1995.
[33] Jacquet, P. and Szpankowski, W., Analytical Depoissonization and Its Applications, *Theoretical Computer Science,* 201, 1–62, 1998.
[34] Karp, R., The Probabilistic Analysis of Some Combinatorial Search Algorithms. In *Algorithms and Complexity,* J.F. Traub, Ed., Academic Press, New York, 1976.
[35] Karp, R., An Introduction to Randomized Algorithms, *Discrete Applied Mathematics,* 34, 165–201, 1991.
[36] Kingman, J.F.C., *Subadditive Processes,* in Ecole d'Eté de Probabilités de Saint-Flour V-1975, Lecture Notes in Mathematics, 539, Springer-Verlag, Berlin, 1976.
[37] Kirschenhofer, P., Prodinger, H., and Szpankowski, W., On the Variance of the External Path in a Symmetric Digital Trie, *Discrete Applied Mathematics,* 25, 129–143, 1989.
[38] Kirschenhofer, P., Prodinger, H., and Szpankowski, W., Analysis of a Splitting Process Arising in Probabilistic Counting and Other Related Algorithms, *Random Structures & Algorithms,* 9, 379–401, 1996.
[39] Knuth, D.E., *The Art of Computer Programming. Fundamental Algorithms,* Vol. 1. Addison-Wesley, Reading, MA, 1973.
[40] Knuth, D.E., *The Art of Computer Programming. Seminumerical Algorithms.* Vol. II. Addison Wesley, Reading, MA, 1981.
[41] Knuth, D.E., *The Art of Computer Programming. Sorting and Searching,* Vol. 3., Addison-Wesley, Reading, MA, 1973.
[42] Levin, L., Average Case Complete Problems, *SIAM J. Computing,* 15, 285–286, 1986.
[43] Louchard, G., Random Walks, Gaussian Processes and List Structures, *Theor. Comp. Sci.,* 53, 99–124, 1987.
[44] Mahmoud, H., *Evolution of Random Search Trees,* John Wiley & Sons, New York 1992.
[45] McDiarmid, C., On the Method of Bounded Differences, in *Surveys in Combinatorics,* J. Siemons, Ed., Vol. 141, 148–188, London Mathematical Society Lecture Notes Series, Cambridge University Press, 1989.
[46] Motwani, R. and Raghavan, P., *Randomized Algorithms,* Cambridge University Press, Cambridge, 1995.
[47] Odlyzko, A., Asymptotic Enumeration, in *Handbook of Combinatorics,* Vol. II, R. Graham, M. Götschel and L. Lovász, Eds., Elsevier Science, 1063-1229, 1995.
[48] Pittel, B., Asymptotic Growth of a Class of Random Trees, *Ann. Probability,* 18, 414–427, 1985.
[49] Sedgewick, R. and Flajolet, P., *An Introduction to the Analysis of Algorithms,* Addison-Wesley, Reading, MA, 1995.
[50] Steele, J.M., *Probability Theory and Combinatorial Optimization,* SIAM, Philadelphia, PA, 1997.
[51] Szpankowski, W., Solution of a Linear Recurrence Equation Arising in the Analysis of Some Algorithms, *SIAM J. Alg. Disc. Methods,* 8, 233–250, 1987.
[52] Szpankowski, W., The Evaluation of an Alternating Sum with Applications to the Analysis of Some Data Structures, *Information Processing Letters,* 28, 13–19, 1988.
[53] Szpankowski, W., Patricia Tries Again Revisited, *J. ACM,* 37, 691–711, 1990.
[54] Szpankowski, W., A Generalized Suffix Tree and Its (Un)Expected Asymptotic Behaviors, *SIAM J. Computing,* 22, 1176–1198, 1993.
[55] Szpankowski, W., On Asymptotics of Certain Sums Arising in Coding Theory, *IEEE Trans. Information Theory,* 41, 2087–2090, 1995.
[56] Talagrand, M., A New look at Independence, *Ann. Appl. Probab.,* 6, 1–34, 1996.

[57] Titchmarsh, E.C., *The Theory of Functions,* Oxford University Press, Oxford, 1944.
[58] Ukkonen, E., A Linear-Time Algorithm for Finding Approximate Shortest Common Superstrings, *Algorithmica,* 5, 313–323, 1990.
[59] Vitter, J. and Flajolet, P., Average-Case Analysis of Algorithms and Data Structures, In *Handbook of Theoretical Computer Science,* J. van Leewen, Ed., 433-524, Elsevier Science Publishers, 1990.
[60] Waterman, M., *Introduction to Computational Biology,* Chapman & Hall, London, 1995.

Further Information

In this chapter we illustrated probabilistic techniques on examples from "stringology," that is, problems on words. Probabilistic methods found applications in many other facets of computer science, namely, random graphs (cf. [3, 22]), computational geometry (cf. [46]), combinatorial algorithms (cf. [10, 34, 50]), molecular biology (cf. [60]), and so forth. Probabilistic methods are useful in the design of randomized algorithms that make random choices during their executions. The reader interested in these algorithms is referred to [35, 46]. Analytic techniques are discussed in Knuth [39, 40, 41], recent book of Sedgewick and Flajolet [49], and in a forthcoming new book by the same authors [21].

Finally, a homepage of **Analysis of Algorithms** was recently created. The interested reader is invited to visit `http://pauillac.inria.fr/algo/AofA/index.html`.

15

Randomized Algorithms

Rajeev Motwani[1]
Stanford University

Prabhakar Raghavan
IBM Almaden Research Center

15.1 Introduction

A **randomized algorithm** is one that makes random choices during its execution. The behavior of such an algorithm may thus, be random even on a fixed input. The design and analysis of a randomized algorithm focuses on establishing that it is likely to behave "well" on *every* input; the likelihood in such a statement depends only on the probabilistic choices made by the algorithm during execution and not on any assumptions about the input. It is especially important to distinguish a randomized algorithm from the *average-case analysis* of algorithms, where one analyzes an algorithm assuming that its input is drawn from a fixed probability distribution. With a randomized algorithm, in contrast, no assumption is made about the input.

Two benefits of randomized algorithms have made them popular: simplicity and efficiency. For many applications, a randomized algorithm is the simplest algorithm available, or the fastest, or both. Below we make these notions concrete through a number of illustrative examples. We assume that the reader has

[1]Supported by an Alfred P. Sloan Research Fellowship, an IBM Faculty Partnership Award, an ARO MURI Grant DAAH04-96-1-0007, and NSF Young Investigator Award CCR-9357849, with matching funds from IBM, Schlumberger Foundation, Shell Foundation, and Xerox Corporation.

had undergraduate courses in algorithms and complexity, and in probability theory. A comprehensive source for randomized algorithms is the book by the authors [24]. The articles by Karp [17], Maffioli, Speranza, and Vercellis [21] and Welsh [44] are good surveys of randomized algorithms. The book by Mulmuley [25] focuses on randomized geometric algorithms.

Throughout this chapter we assume the RAM model of computation, in which we have a machine that can perform the following operations involving registers and main memory: input–output operations, memory–register transfers, indirect addressing, branching and arithmetic operations. Each register or memory location may hold an integer which can be accessed as a unit, but an algorithm has no access to the representation of the number. The arithmetic instructions permitted are $+, -, \times, /$. In addition, an algorithm can compare two numbers, and evaluate the square root of a positive number. In this article $\mathbf{E}[X]$ will denote the expectation of a random variable X, and $\mathbf{Pr}[A]$ will denote the probability of an event A.

15.2 Sorting and Selection by Random Sampling

Some of the earliest randomized algorithms included algorithms for sorting a set S of numbers, and the related problem of finding the kth smallest element in S. The main idea behind these algorithms is the use of *random sampling:* a randomly chosen member of S is unlikely to be one of its largest or smallest elements; rather, it is likely to be "near the middle." Extending this intuition suggests that a random sample of elements from S is likely to be spread "roughly uniformly" in S. We now describe randomized algorithms for sorting and selection based on these ideas.

Algorithm RQS:

Input: A set of numbers S.

Output: The elements of S sorted in increasing order.

1. Choose an element y uniformly at random from S: every element in S has equal probability of being chosen.

2. By comparing each element of S with y, determine the set S_1 of elements smaller than y and the set S_2 of elements larger than y.

3. Recursively sort S_1 and S_2. Output the sorted version of S_1, followed by y, and then the sorted version of S_2.

Algorithm **RQS** is an example of a *randomized algorithm* — an algorithm that makes random choices during execution. It is inspired by the **Quicksort** algorithm due to Hoare [12], and described in [24]. We assume that the random choice in Step 1 can be made in unit time. What can we prove about the running time of **RQS**?

We now analyze the *expected* number of comparisons in an execution of **RQS.** Comparisons are performed in Step 2, in which we compare a randomly chosen element to the remaining elements. For $1 \leq i \leq n$, let $S_{(i)}$ denote the element of *rank* i (the ith smallest element) in the set S. Define the random variable X_{ij} to assume the value 1 if $S_{(i)}$ and $S_{(j)}$ are compared in an execution, and the value 0 otherwise. Thus, the total number of comparisons is $\Sigma_{i=1}^{n} \Sigma_{j>i} X_{ij}$. By linearity of expectation the expected number of comparisons is

$$\mathbf{E}\left[\sum_{i=1}^{n}\sum_{j>i} X_{ij}\right] = \sum_{i=1}^{n}\sum_{j>i} \mathbf{E}\left[X_{ij}\right] . \tag{15.1}$$

Let p_{ij} denote the probability that $S_{(i)}$ and $S_{(j)}$ are compared during an execution. Then

$$\mathbf{E}\left[X_{ij}\right] = p_{ij} \times 1 + \left(1 - p_{ij}\right) \times 0 = p_{ij} \; . \tag{15.2}$$

To compute p_{ij} we view the execution of **RQS** as a binary tree T each node of which is labeled with a distinct element of S. The root of the tree is labeled with the element y chosen in Step 1, the left subtree of y contains the elements in S_1, and the right subtree of y contains the elements in S_2. The structures of the two subtrees are determined recursively by the executions of **RQS** on S_1 and S_2. The root y is compared to the elements in the two subtrees, but no comparison is performed between an element of the left subtree and an element of the right subtree. Thus, there is a comparison between $S_{(i)}$ and $S_{(j)}$ if and only if one of these elements is an ancestor of the other.

Consider the permutation π obtained by visiting the nodes of T in increasing order of the level numbers, and in a left-to-right order within each level; recall that the ith level of the tree is the set of all nodes at distance exactly i from the root. The following two observation lead to the determination of p_{ij}.

1. There is a comparison between $S_{(i)}$ and $S_{(j)}$ if and only if $S_{(i)}$ or $S_{(j)}$ occurs earlier in the permutation π than any element $S_{(\ell)}$ such that $i < \ell < j$. To see this, let $S_{(k)}$ be the earliest in π from among all elements of rank between i and j. If $k \notin \{i, j\}$, then $S_{(i)}$ will belong to the left subtree of $S_{(k)}$ while $S_{(j)}$ will belong to the right subtree of $S_{(k)}$, implying that there is no comparison between $S_{(i)}$ and $S_{(j)}$. Conversely, when $k \in \{i, j\}$, there is an ancestor-descendant relationship between $S_{(i)}$ and $S_{(j)}$, implying that the two elements are compared by **RQS.**
2. Any of the elements $S_{(i)}, S_{(i+1)}, \ldots, S_{(j)}$ is equally likely to be the first of these elements to be chosen as a partitioning element and hence, to appear first in π. Thus, the probability that this first element is either $S_{(i)}$ or $S_{(j)}$ is exactly $2/(j - i + 1)$.

It follows that $p_{ij} = 2/(j - i + 1)$. By (15.1) and (15.2), the expected number of comparisons is given by

$$\begin{aligned} \sum_{i=1}^{n} \sum_{j>i} p_{ij} &= \sum_{i=1}^{n} \sum_{j>i} \frac{2}{j - i + 1} \\ &\leq \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{k+1} \\ &\leq 2 \sum_{i=1}^{n} \sum_{k=1}^{n} \frac{1}{k} \; . \end{aligned}$$

It follows that the expected number of comparisons is bounded above by $2nH_n$, where H_n is the *nth harmonic number,* defined by $H_n = \Sigma_{k=1}^{n} 1/k$.

THEOREM 15.1 *The expected number of comparisons in an execution of* **RQS** *is at most* $2nH_n$.

Now $H_n = \ln n + \Theta(1)$, so that the expected running time of **RQS** is $\mathrm{O}\left(n \log n\right)$. Note that this expected running time *holds for every input.* It is an expectation that depends only on the random choices made by the algorithm, and *not* on any assumptions about the distribution of the input.

Randomized Selection

We now consider the use of random sampling for the problem of selecting the kth smallest element in a set S of n elements drawn from a totally ordered universe. We assume that the elements of S are all distinct,

although it is not very hard to modify the following analysis to allow for multisets. Let $r_S(t)$ denote the rank of an element t (the kth smallest element has rank k) and recall that $S_{(i)}$ denotes the ith smallest element of S. Thus, we seek to identify $S_{(k)}$. We extend the use of this notation to subsets of S as well. The following algorithm is adapted from one due to Floyd and Rivest [9].

Algorithm LazySelect:

Input: A set S of n elements from a totally ordered universe, and an integer k in $[1, n]$.

Output: The kth smallest element of S, $S_{(k)}$.

1. Pick $n^{3/4}$ elements from S, chosen independently and uniformly at random with replacement; call this multiset of elements R.

2. Sort R in $O(n^{3/4} \log n)$ steps using any optimal sorting algorithm.

3. Let $x = kn^{-1/4}$. For $\ell = \max\{\lfloor x - \sqrt{n} \rfloor, 1\}$ and $h = \min\{\lceil x + \sqrt{n} \rceil, n^{3/4}\}$, let $a = R_{(\ell)}$ and $b = R_{(h)}$. By comparing a and b to every element of S, determine $r_S(a)$ and $r_S(b)$.

4. **if** $k < n^{1/4}$, let $P = \{y \in S \mid y \leq b\}$ and $r = k$;
else if $k > n - n^{1/4}$, let $P = \{y \in S \mid y \geq a\}$ and $r = k - r_S(a) + 1$;
else if $k \in [n^{1/4}, n - n^{1/4}]$, let $P = \{y \in S \mid a \leq y \leq b\}$ and $r = k - r_S(a) + 1$;
Check whether $S_{(k)} \in P$ *and* $|P| \leq 4n^{3/4} + 2$. If not, repeat Steps 1–3 until such a set P is found.

5. By sorting P in $O(|P| \log |P|)$ steps, identify P_r, which is $S_{(k)}$.

Figure 15.1 illustrates Step 3, where small elements are at the left end of the picture and large ones to the right. Determining (in Step 4) whether $S_{(k)} \in P$ is easy, since we know the ranks $r_S(a)$ and $r_S(b)$ and we compare either or both of these to k, depending on which of the three **if** statements in Step 4 we execute. The sorting in Step 5 can be performed in $O(n^{3/4} \log n)$ steps.

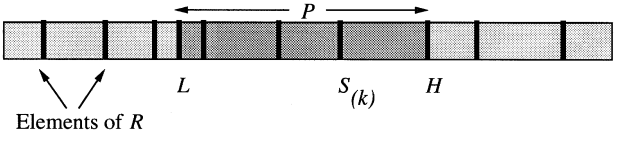


FIGURE 15.1 The **LazySelect** algorithm.

Thus, the idea of the algorithm is to identify two elements a and b in S such that both of the following statements hold with high probability:

1. The element $S_{(k)}$ that we seek is in P, the set of elements between a and b;
2. The set P of elements is not very large, so that we can sort P inexpensively in Step 5.

As in the analysis of **RQS** we measure the running time of **LazySelect** in terms of the number of comparisons performed by it. The following theorem is established using the *Chebyshev bound* from elementary probability theory; a full proof may be found in [24].

THEOREM 15.2 *With probability* $1 - O(n^{-1/4})$, **LazySelect** *finds* $S_{(k)}$ *on the first pass through Steps 1–5, and thus, performs only* $2n + o(n)$ *comparisons.*

This adds to the significance of **LazySelect:** the best known deterministic selection algorithms use $3n$ comparisons in the worst case, and are quite complicated to implement.

15.3 A Simple Min-Cut Algorithm

Two events $\mathcal{E}_1$ and $\mathcal{E}_2$ are said to be *independent* if the probability that they both occur is given by

$$\mathbf{Pr}\left[\mathcal{E}_1 \cap \mathcal{E}_2\right] = \mathbf{Pr}\left[\mathcal{E}_1\right] \times \mathbf{Pr}\left[\mathcal{E}_2\right] \ . \tag{15.3}$$

More generally when $\mathcal{E}_1$ and $\mathcal{E}_2$ are not necessarily independent,

$$\mathbf{Pr}\left[\mathcal{E}_1 \cap \mathcal{E}_2\right] = \mathbf{Pr}\left[\mathcal{E}_1 \mid \mathcal{E}_2\right] \times \mathbf{Pr}\left[\mathcal{E}_2\right] = \mathbf{Pr}\left[\mathcal{E}_2 \mid \mathcal{E}_1\right] \times \mathbf{Pr}\left[\mathcal{E}_1\right] \ , \tag{15.4}$$

where $\mathbf{Pr}\left[\mathcal{E}_1 \mid \mathcal{E}_2\right]$ denotes the *conditional probability* of $\mathcal{E}_1$ given $\mathcal{E}_2$. When a collection of events is not independent, the probability of their intersection is given by the following generalization of (15.4):

$$\mathbf{Pr}\left[\cap_{i=1}^{k}\mathcal{E}_i\right] = \mathbf{Pr}\left[\mathcal{E}_1\right] \times \mathbf{Pr}\left[\mathcal{E}_2 \mid \mathcal{E}_1\right] \times \mathbf{Pr}\left[\mathcal{E}_3 \mid \mathcal{E}_1 \cap \mathcal{E}_2\right] \cdots \mathbf{Pr}\left[\mathcal{E}_k \mid \cap_{i=1}^{k-1}\mathcal{E}_i\right] \ . \tag{15.5}$$

Let G be a connected, undirected multigraph with n vertices. A *multigraph* may contain multiple edges between any pair of vertices. A *cut* in G is a set of edges whose removal results in G being broken into two or more components. A *min-cut* is a cut of minimum cardinality. We now study a simple algorithm due to Karger [14] for finding a min-cut of a graph.

We repeat the following step: pick an edge uniformly at random and merge the two vertices at its end points. If as a result there are several edges between some pairs of (newly formed) vertices, retain them all. Remove edges between vertices that are merged, so that there are never any self-loops. This process of merging the two end-points of an edge into a single vertex is called the *contraction* of that edge. With each contraction, the number of vertices of G decreases by one. Note that as long as at least two vertices remain, an edge contraction does not reduce the min-cut size in G. The algorithm continues the contraction process until only two vertices remain; at this point, the set of edges between these two vertices is a cut in G and is output as a candidate min-cut. What is the probability that this algorithm finds a min-cut?

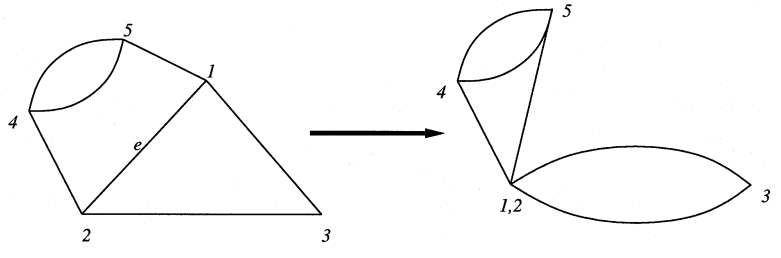


FIGURE 15.2 A step in the min-cut algorithm; the effect of contracting edge $e = (1, 2)$ is shown.

DEFINITION 15.1 For any vertex v in a multigraph G, the *neighborhood* of G, denoted $\Gamma(v)$, is the set of vertices of G that are adjacent to v. The *degree* of v, denoted $d(v)$, is the number of edges incident on v. For a set S of vertices of G, the neighborhood of S, denoted $\Gamma(S)$, is the union of the neighborhoods of the constituent vertices.

Note that $d(v)$ is the same as the cardinality of $\Gamma(v)$ when there are no self-loops or multiple edges between v and any of its neighbors.

Let k be the min-cut size and let C be a particular min-cut with k edges. Clearly G has at least $kn/2$ edges (otherwise there would be a vertex of degree less than k, and its incident edges would be a min-cut of size less than k). We bound from below the probability that no edge of C is ever contracted during an execution of the algorithm, so that the edges surviving till the end are exactly the edges in C.

For $1 \leq i \leq n-2$, let $\mathcal{E}_i$ denote the event of *not* picking an edge of C at the ith step. The probability that the edge randomly chosen in the first step is in C is at most $k/(nk/2) = 2/n$, so that $\mathbf{Pr}[\mathcal{E}_1] \geq 1 - 2/n$. Conditioned on the occurrence of $\mathcal{E}_1$, there are at least $k(n-1)/2$ edges during the second step so that $\mathbf{Pr}[\mathcal{E}_2 \mid \mathcal{E}_1] \geq 1 - 2/(n-1)$. Extending this calculation, $\mathbf{Pr}\left[\mathcal{E}_i \mid \cap_{j=1}^{i-1}\mathcal{E}_j\right] \geq 1 - 2/(n-i+1)$. We now invoke (15.5) to obtain

$$\mathbf{Pr}\left[\cap_{i=1}^{n-2}\mathcal{E}_i\right] \geq \prod_{i=1}^{n-2}\left(1 - \frac{2}{n-i+1}\right) = \frac{2}{n(n-1)}\,.$$

Our algorithm may err in declaring the cut it outputs to be a min-cut. But the probability of discovering a particular min-cut (which may in fact be the unique min-cut in G) is larger than $2/n^2$, so the probability of error is at most $1 - 2/n^2$. Repeating the above algorithm $n^2/2$ times making independent random choices each time, the probability that a min-cut is not found in any of the $n^2/2$ attempts is [by (15.3)] at most

$$\left(1 - \frac{2}{n^2}\right)^{n^2/2} < 1/e\,.$$

By this process of repetition, we have managed to reduce the probability of failure from $1 - 2/n^2$ to less than $1/e$. Further executions of the algorithm will make the failure probability arbitrarily small (the only consideration being that repetitions increase the running time). Note the extreme simplicity of this randomized min-cut algorithm. In contrast, most **deterministic algorithms** for this problem are based on network flow and are considerably more complicated.

Classification of Randomized Algorithms

The randomized sorting algorithm and the min-cut algorithm exemplify two different types of randomized algorithms. The sorting algorithm *always* gives the correct solution. The only variation from one run to another is its running time, whose distribution we study. Such an algorithm is called a **Las Vegas algorithm.**

In contrast, the min-cut algorithm may sometimes produce a solution that is incorrect. However, we prove that the probability of such an error is bounded. Such an algorithm is called a **Monte Carlo algorithm.** In Section 15.3 we observed a useful property of a Monte Carlo algorithm: if the algorithm is run repeatedly with independent random choices each time, the failure probability can be made arbitrarily small, at the expense of running time. In some randomized algorithms both the running time and the quality of the solution are random variables; sometimes these are also referred to as Monte Carlo algorithms. The reader is referred to [24] for a detailed discussion of these issues.

15.4 Foiling an Adversary

A common paradigm in the design of randomized algorithms is that of *foiling an adversary.* Whereas an adversary might succeed in defeating a deterministic algorithm with a carefully constructed "bad" input, it is difficult for an adversary to defeat a deterministic algorithm in this fashion. Due to the random choices made by the randomized algorithm the adversary cannot, while constructing the input, predict the precise

behavior of the algorithm. An alternative view of this process is to think of the randomized algorithm as first picking a series of random numbers which it then uses in the course of execution as needed. In this view, we may think of the random numbers chosen at the start as "selecting" one of a family of deterministic algorithms. In other words a randomized algorithm can be thought of as a probability distribution on deterministic algorithms. We illustrate these ideas in the setting of AND-OR *tree evaluation*; the following algorithm is due to Snir [38].

For our purposes an AND-OR tree is a rooted complete binary tree in which internal nodes at even distance from the root are labeled AND and internal nodes at odd distance are labeled OR. Associated with each leaf is a Boolean *value.* The *evaluation* of the game tree is the following process. Each leaf *returns* the value associated with it. Each OR node returns the Boolean OR of the values returned by its children, and each AND node returns the Boolean AND of the values returned by its children. At each step an evaluation algorithm chooses a leaf and reads its value. We do not charge the algorithm for any other computation. We study the number of such steps taken by an algorithm for evaluating an AND-OR tree, the worst case being taken over all assignments of Boolean values to the leaves.

Let T_k denote an AND-OR tree in which every leaf is at distance $2k$ from the root. Thus, any root-to-leaf path passes through k AND nodes (including the root itself) and k OR nodes, and there are 2^{2k} leaves. An algorithm begins by specifying a leaf whose value is to be read at the first step. Thereafter, it specifies such a leaf at each step, based on the values it has read on previous steps. In a deterministic algorithm, the choice of the next leaf to be read is a deterministic function of the values at the leaves read so far. For a randomized algorithm, this choice may be randomized. It is not hard to show that for any deterministic evaluation algorithm, there is an instance of T_k that forces the algorithm to read the values on all 2^{2k} leaves.

We now give a simple randomized algorithm and study the expected number of leaves it reads on any instance of T_k. The algorithm is motivated by the following simple observation. Consider a single AND node with two leaves. If the node were to return 0, at least one of the leaves must contain 0. A deterministic algorithm inspects the leaves in a fixed order, and an adversary can therefore always "hide" the 0 at the second of the two leaves inspected by the algorithm. Reading the leaves in a random order foils this strategy. With probability $1/2$, the algorithm chooses the hidden 0 on the first step, so its expected number of steps is $3/2$, which is better than the worst case for any deterministic algorithm. Similarly, in the case of an OR node, if it were to return a 1 then a randomized order of examining the leaves will reduce the expected number of steps to $3/2$. We now extend this intuition and specify the complete algorithm.

To evaluate an AND node v, the algorithm chooses one of its children (a subtree rooted at an OR node) at random and evaluates it by recursively invoking the algorithm. If 1 is returned by the subtree, the algorithm proceeds to evaluate the other child (again by recursive application). If 0 is returned, the algorithm returns 0 for v. To evaluate an OR node, the procedure is the same with the roles of 0 and 1 interchanged. We establish by induction on k that the expected cost of evaluating any instance of T_k is at most 3^k.

The basis ($k = 0$) is trivial. Assume now that the expected cost of evaluating any instance of T_{k-1} is at most 3^{k-1}. Consider first a tree T whose root is an OR node, each of whose children is the root of a copy of T_{k-1}. If the root of T were to evaluate to 1, at least one of its children returns 1. With probability $1/2$ this child is chosen first, incurring (by the inductive hypothesis) an expected cost of at most 3^{k-1} in evaluating T. With probability $1/2$ both subtrees are evaluated, incurring a net cost of at most $2 \times 3^{k-1}$. Thus, the expected cost of determining the value of T is

$$\leq \frac{1}{2} \times 3^{k-1} + \frac{1}{2} \times 2 \times 3^{k-1} = \frac{3}{2} \times 3^{k-1} . \tag{15.6}$$

If on the other hand the OR were to evaluate to 0 both children must be evaluated, incurring a cost of at most $2 \times 3^{k-1}$.

Consider next the root of the tree T_k, an AND node. If it evaluates to 1, then both its subtrees rooted at OR nodes return 1. By the discussion in the previous paragraph and by linearity of expectation, the expected cost of evaluating T_k to 1 is at most $2 \times (3/2) \times 3^{k-1} = 3^k$. On the other hand, if the instance

of T_k evaluates to 0, at least one of its subtrees rooted at OR nodes returns 0. With probability 1/2 it is chosen first, and so the expected cost of evaluating T_k is at most

$$2 \times 3^{k-1} + \frac{1}{2} \times \frac{3}{2} \times 3^{k-1} \leq 3^k \ .$$

THEOREM 15.3 *Given any instance of T_k, the expected number of steps for the above randomized algorithm is at most 3^k.*

Since $n = 4^k$ the expected running time of our randomized algorithm is $n^{\log_4 3}$, which we bound by $n^{0.793}$. Thus, the expected number of steps is smaller than the worst case for any deterministic algorithm. Note that this is a Las Vegas algorithm and always produces the correct answer.

15.5 The Minimax Principle and Lower Bounds

The randomized algorithm of the preceding section has an expected running time of $n^{0.793}$ on any uniform binary AND-OR tree with n leaves. Can we establish that *no randomized algorithm* can have a lower expected running time? We first introduce a standard technique due to Yao [45] for proving such lower bounds. This technique applies only to algorithms that terminate in finite time on all inputs and sequences of random choices.

The crux of the technique is to relate the running times of randomized algorithms for a problem to the running times of *deterministic* algorithms for the problem *when faced with randomly chosen inputs.* Consider a problem where the number of distinct inputs of a fixed size is finite, as is the number of distinct (deterministic, terminating, and always correct) algorithms for solving that problem. Let us define the **distributional complexity** of the problem at hand as the expected running time of the best deterministic algorithm for the worst distribution on the inputs. Thus, we envision an adversary choosing a probability distribution on the set of possible inputs, and study the best deterministic algorithm for this distribution. Let $\boldsymbol{p}$ denote a probability distribution on the set $\mathcal{I}$ of inputs. Let the random variable $C(I_{\boldsymbol{p}}, A)$ denote the running time of deterministic algorithm $A \in \mathcal{A}$ on an input chosen according to $\boldsymbol{p}$. Viewing a randomized algorithm as a probability distribution $\boldsymbol{q}$ on the set $\mathcal{A}$ of deterministic algorithms, we let the random variable $C(I, A_{\boldsymbol{q}})$ denote the running time of this randomized algorithm on the worst-case input.

PROPOSITION 15.1 **(Yao's Minimax Principle):** For all distributions $\boldsymbol{p}$ over $\mathcal{I}$ and $\boldsymbol{q}$ over $\mathcal{A}$,

$$\min_{A \in \mathcal{A}} \mathbf{E}\left[C(I_{\boldsymbol{p}}, A)\right] \leq \max_{I \in \mathcal{I}} \mathbf{E}\left[C(I, A_{\boldsymbol{q}})\right] \ .$$

In other words, the expected running time of the optimal deterministic algorithm for an arbitrarily chosen input distribution $\boldsymbol{p}$ is a lower bound on the expected running time of the optimal (Las Vegas) randomized algorithm for Π. Thus, to prove a lower bound on the **randomized complexity** it suffices to choose any distribution $\boldsymbol{p}$ on the input and prove a lower bound on the expected running time of deterministic algorithms for that distribution. The power of this technique lies in the flexibility in the choice of $\boldsymbol{p}$ and, more importantly, the reduction to a lower bound on deterministic algorithms. It is important to remember that the deterministic algorithm "knows" the chosen distribution $\boldsymbol{p}$.

The above discussion dealt only with lower bounds on the performance of Las Vegas algorithms. We briefly discuss Monte Carlo algorithms with error probability $\epsilon \in [0, 1/2]$. Let us define the distributional complexity with error ϵ, denoted $\min_{A \in \mathcal{A}} \mathbf{E}[C_\epsilon(I_{\boldsymbol{p}}, A)]$, to be the minimum expected running time of any deterministic algorithm that errs with probability at most ϵ under the input distribution $\boldsymbol{p}$. Similarly, we denote by $\max_{I \in \mathcal{I}} \mathbf{E}[C_\epsilon(I, A_{\boldsymbol{q}})]$ the expected running time (under the worst input) of any randomized

algorithm that errs with probability at most ϵ (again, the randomized algorithm is viewed as a probability distribution $\boldsymbol{q}$ on deterministic algorithms). Analogous to Proposition 15.1, we then have the following:

PROPOSITION 15.2 For all distributions $\boldsymbol{p}$ over $\mathcal{I}$ and $\boldsymbol{q}$ over $\mathcal{A}$ and any $\epsilon \in [0, 1/2]$,

$$\frac{1}{2}\left(\min_{A\in\mathcal{A}} \mathbf{E}\left[C_{2\epsilon}\left(I_{\boldsymbol{p}}, A\right)\right]\right) \leq \max_{I\in\mathcal{I}} \mathbf{E}\left[C_{\epsilon}\left(I, A_{\boldsymbol{q}}\right)\right] .$$

Lower Bound for Game Tree Evaluation

We now apply Yao's Minimax Principle to the AND-OR tree evaluation problem. A randomized algorithm for AND-OR tree evaluation can be viewed as a probability distribution over deterministic algorithms, because the length of the computation as well as the number of choices at each step are both finite. We may imagine that all of these coins are tossed before the beginning of the execution.

The tree T_k is equivalent to a balanced binary tree all of whose leaves are at distance $2k$ from the root, and all of whose internal nodes compute the NOR function: a node returns the value 1 if both inputs are 0, and 0 otherwise. We proceed with the analysis of this tree of NORs of depth $2k$.

Let $p = (3 - \sqrt{5})/2$; each leaf of the tree is independently set to 1 with probability p. If each input to a NOR node is independently 1 with probability p, its output is 1 with probability

$$\left(\frac{\sqrt{5}-1}{2}\right)^2 = \frac{3-\sqrt{5}}{2} = p .$$

Thus, the value of every node of the NOR tree is 1 with probability p, and the value of a node is independent of the values of all the other nodes on the same level. Consider a deterministic algorithm that is evaluating a tree furnished with such random inputs; let v be a node of the tree whose value the algorithm is trying to determine. Intuitively, the algorithm should determine the value of one child of v before inspecting any leaf of the other subtree. An alternative view of this process is that the deterministic algorithm should inspect leaves visited in a depth-first search of the tree, except of course that it ceases to visit subtrees of a node v when the value of v has been determined. Let us call such an algorithm a *depth-first pruning* algorithm, referring to the order of traversal and the fact that subtrees that supply no additional information are "pruned" away without being inspected. The following result is due to Tarsi [40].

PROPOSITION 15.3 Let T be a NOR tree each of whose leaves is independently set to 1 with probability q for a fixed value $q \in [0, 1]$. Let $W(T)$ denote the minimum, over all deterministic algorithms, of the expected number of steps to evaluate T. Then, there is a depth-first pruning algorithm whose expected number of steps to evaluate T is $W(T)$.

Proposition 15.3 tells us that for the purposes of our lower bound, we may restrict our attention to depth-first pruning algorithms. Let $W(h)$ be the expected number of leaves inspected by a depth-first pruning algorithm in determining the value of a node at distance h from the leaves, when each leaf is independently set to 1 with probability $(3 - \sqrt{5})/2)$. Clearly

$$W(h) = W(h-1) + (1-p) \times W(h-1) ,$$

where the first term represents the work done in evaluating one of the subtrees of the node, and the second term represents the work done in evaluating the other subtree (which will be necessary if the first subtree returns the value 0, an event occurring with probability $1 - p$). Letting h be $\log_2 n$, and solving, we get $W(h) \geq n^{0.694}$.

THEOREM 15.4 *The expected running time of any randomized algorithm that always evaluates an instance of T_k correctly is at least $n^{0.694}$, where $n = 2^{2k}$ is the number of leaves.*

Why is our lower bound of $n^{0.694}$ less than the upper bound of $n^{0.793}$ that follows from Theorem 15.3? The reason is that we have not chosen the best possible probability distribution for the values of the leaves. Indeed, in the NOR tree if both inputs to a node are 1, no reasonable algorithm will read leaves of both subtrees of that node. Thus, to prove the best lower bound we have to choose a distribution on the inputs that precludes the event that both inputs to a node will be 1; in other words, the values of the inputs are chosen at random but not independently. This stronger (and considerably harder) analysis can in fact be used to show that the algorithm of Section 15.4 is optimal; the reader is referred to the paper of Saks and Wigderson [33] for details.

15.6 Randomized Data Structures

Recent research into data structures has strongly emphasized the use of randomized techniques to achieve increased efficiency without sacrificing simplicity of implementation. An illustrative example is the randomized data structure for dynamic dictionaries called *skip list* that is due to Pugh [27].

The dynamic dictionary problem is that of maintaining a set of keys X drawn from a totally ordered universe so as to provide efficient support of the following operations: $\mathsf{find}(q, X)$ — decide whether the query key q belongs to X and return the information associated with this key if it does indeed belong to X; $\mathsf{insert}(q, X)$ — insert the key q into the set X, unless it is already present in X; $\mathsf{delete}(q, X)$ — delete the key q from X, unless it is absent from X. The standard approach for solving this problem involves the use of a binary search tree and gives worst-case time per operation that is $O(\log n)$, where n is the size of X at the time the operation is performed. Unfortunately, achieving this time bound requires the use of complex rebalancing strategies to ensure that the search tree remains "balanced," i.e., has depth $O(\log n)$. Not only does rebalancing require more effort in terms of implementation, it also leads to significant overheads in the running time (at least in terms of the constant factors subsumed by the big-oh notation). The skip list data structure is a rather pleasant alternative that overcomes both these shortcomings.

Before getting into the details of randomized skip lists, we will develop some of the key ideas without the use of randomization. Suppose we have a totally ordered data set $X = \{x_1 < x_2 < \cdots < x_n\}$. A *gradation* of X is a sequence of nested subsets (called *levels*)

$$X_r \subseteq X_{r-1} \subseteq \cdots \subseteq X_2 \subseteq X_1$$

such that $X_r = \emptyset$ and $X_1 = X$. Given an ordered set X and a gradation for it, the level of any element $x \in X$ is defined as

$$L(x) = \max \{i \mid x \in X_i\} \ ,$$

that is, $L(x)$ is the largest index i such that x belongs to the ith level of the gradation. In what follows, we will assume that two special elements $-\infty$ and $+\infty$ belong to each of the levels, where $-\infty$ is smaller than all elements in X and $+\infty$ is larger than all elements in X.

We now define an ordered list data structure with respect to a gradation of the set X. The first level, X_1, is represented as an ordered linked list, and each node x in this list has a stack of $L(x) - 1$ additional nodes directly above it. Finally, we obtain the skip list with respect to the gradation of X by introducing horizontal and vertical pointers between these nodes as illustrated in Fig. 15.3. The skip list in Fig. 15.3 corresponds to a gradation of the data set $X = \{1, 3, 4, 7, 9\}$ consisting of the following 6 levels:

$$\begin{aligned} X_6 &= \emptyset \\ X_5 &= \{3\} \\ X_4 &= \{3, 4\} \end{aligned}$$

$$
\begin{aligned}
X_3 &= \{3, 4, 9\} \\
X_2 &= \{3, 4, 7, 9\} \\
X_1 &= \{1, 3, 4, 7, 9\}
\end{aligned}
$$

Observe that starting at the ith node from the bottom in the left-most column of nodes, and traversing the horizontal pointers in order yields a set of nodes corresponding to the elements of the ith level X_i.

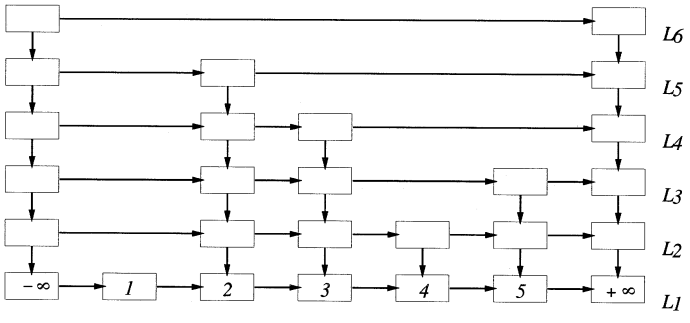


FIGURE 15.3 A skip list.

Additionally, we will view each level i as defining a set of *intervals* each of which is defined as the set of elements of X spanned by a horizontal pointer at level i. The sequence of levels X_i can be viewed as successively coarser partitions of X. In Fig. 15.3, the levels determine the following partitions of X into a intervals.

$$
\begin{aligned}
X_6 &= [-\infty, +\infty] \\
X_5 &= [-\infty, 3] \cup [3, +\infty] \\
X_4 &= [-\infty, 3] \cup [3, 4] \cup [4, +\infty] \\
X_3 &= [-\infty, 3] \cup [3, 4] \cup [4, 9] \cup [9, +\infty] \\
X_2 &= [-\infty, 3] \cup [3, 4] \cup [4, 7] \cup [7, 9] \cup [9, +\infty] \\
X_1 &= [-\infty, 1] \cup [1, 3] \cup [3, 4] \cup [4, 7] \cup [7, 9] \cup [9, +\infty]
\end{aligned}
$$

An alternate view of the skip list is in terms of a tree defined by the interval partition structure, as illustrated in Fig. 15.4 for the example above. In this tree, each node corresponds to an interval, and the intervals at a given level are represented by nodes at the corresponding level of the tree. When an interval J at level $i + 1$ is a superset of an interval I at level i, then the corresponding node J has the node I as a child in this tree. Let $C(I)$ denote the number of children in the tree of a node corresponding to the interval I, i.e., it is the number of intervals from the previous level that are subintervals of I. Note that the tree is not necessarily binary since the value of $C(I)$ is arbitrary. We can view the skip list as a threaded version of this tree, where each thread is a sequence of (horizontal) pointers linking together the nodes at a level into an ordered list. In Fig. 15.4, the broken lines indicate the threads, and the full lines are the actual tree pointers.

Finally, we need some notation concerning the membership of an element x in the intervals defined above, where x is not necessarily a member of X. For each possible x, let $I_j(x)$ be the interval at level j containing x. In the degenerate case where x lies on the boundary between two intervals, we assign it to the leftmost such interval. Observe that the nested sequence of intervals containing y,

$$
I_r(y) \subseteq I_{r-1}(y) \subseteq \cdots \subseteq I_1(y) \,,
$$

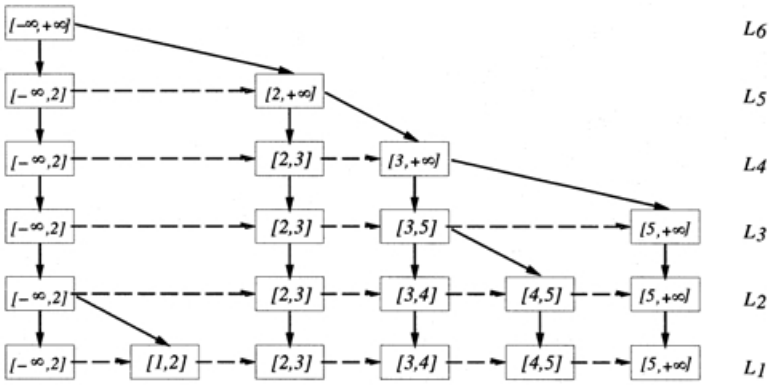


FIGURE 15.4 Tree representation of a skip list.

corresponds to a root-leaf path in the tree corresponding to the skip list.

It remains to specify the choice of the gradation that determines the structure of a skip list. This is precisely where we introduce randomization into the structure of a skip list. The idea is to define a random gradation. Our analysis will show that with high probability, the search tree corresponding to a random skip list is "balanced," and then the dictionary operations can be efficiently implemented.

We define the *random gradation* for X as follows: given level X_i, the next level X_{i+1} is determined by independently choosing to retain each element $x \in X_i$ with probability $1/2$. The random selection process begins with $X_1 = X$ and terminates when for the first time the resulting level is empty. Alternatively, we may view the choice of the gradation as follows: for each $x \in X$, choose the level $L(x)$ independently from the geometric distribution with parameter $p = 1/2$ and place x in the levels $X_1, \ldots, X_{L(x)}$. We define r to be one more than the maximum of these level numbers. Such a random level is chosen for every element of X upon its insertion and remains fixed until its deletion.

We omit the proof of the following theorem bounding the space complexity of a randomized skip list. The proof is a simple exercise, and it is recommended that the reader verify this to gain some insight into the behavior of this data structure.

THEOREM 15.5 *A random skip list for a set X of size n has expected space requirement $O(n)$.*

We will go into more details about the time complexity of this data structure. The following lemma underlies the running time analysis.

LEMMA 15.1 The number of levels r in a random gradation of a set X of size n has expected value $\mathbf{E}[r] = O(\log n)$. Further, $r = O(\log n)$ with high probability.

PROOF We will prove the high probability result; the bound on the expected value follows immediately from this. Recall that the level numbers $L(x)$ for $x \in X$ are i.i.d. (independent and identically distributed) random variables distributed geometrically with parameter $p = 1/2$; notationally, we will denote these random variables by $Z_1, \ldots, Z_n$. Now, the total number of levels in the skip list can be determined as

$$r = 1 + \max_{x \in X} L(x) = 1 + \max_{1 \le i \le n} Z_i \ ,$$

that is, as one more than the maximum of n i.i.d. geometric random variables.

For such geometric random variables with parameter p, it is easy to verify that for any positive real t, $\mathbf{Pr}[Z_i > t] \leq (1-p)^t$. It follows that

$$\mathbf{Pr}\left[\max_i Z_i > t\right] \leq n(1-p)^t = \frac{n}{2^t} ,$$

since $p = 1/2$ in this case. For any $\alpha > 1$, setting $t = \alpha \log n$, we obtain that

$$\mathbf{Pr}\left[r > \alpha \log n\right] \leq \frac{1}{n^{\alpha-1}} .$$

We can now infer that the tree representing the skip list has height $\mathsf{O}(\log n)$ with high probability. To show that the overall search time in a skip list is similarly bounded, we must first specify an efficient implementation of the find operation. We present the implementation of the dictionary operations in terms of the tree representation; it is fairly easy to translate this back into the skip list representation.

To implement $\mathsf{find}(y, X)$, we must walk down the path

$$I_r(y) \subseteq I_{r-1}(y) \subseteq \cdots \subseteq I_1(y) .$$

For this, at level j, starting at the node $I_j(y)$, we use the vertical pointer to descend to the leftmost child of the current interval; then, via the horizontal pointers, we move rightwards till the node $I_j(y)$ is reached. Note that it is easily determined whether y belongs to a given interval, or to an interval to its right. Further, in the skip list, the vertical pointers allow access only to the leftmost child of an interval, and therefore we must use the horizontal pointers to scan its children.

To determine the expected cost of a $\mathsf{find}(y, X)$ operation, we must take into account both the number of levels and the number of intervals/nodes scanned at each level. Clearly, at level j, the number of nodes visited is no more than the number of children of $I_{j+1}(y)$. It follows that the cost of find can be bounded by

$$\mathsf{O}\left(\sum_{j=1}^{r} \left(1 + C\left(I_j(y)\right)\right)\right) .$$

The following lemma shows that this quantity has expectation bounded by $\mathsf{O}(\log n)$.

LEMMA 15.2 For any y, let $I_r(y), \ldots, I_1(y)$ be the search path followed by $\mathsf{find}(y, X)$ in a random skip list for a set X of size n. Then,

$$\mathbf{E}\left[\sum_{j=1}^{r} \left(1 + C(I_j(y))\right)\right] = \mathsf{O}(\log n) .$$

PROOF We begin by showing that for any interval I in a random skip list, $\mathbf{E}[C(I)] = \mathsf{O}(1)$. By Lemma 15.1, we are guaranteed that $r = \mathsf{O}(\log n)$ with high probability, and so we will obtain the desired bound. It is important to note that we really do need the high probability bound on Lemma 15.1, since it is incorrect to multiply the expectation of r with that of $1 + C(I)$ (the two random variables need not be independent). However, in the approach we will use, since $r > \alpha \log n$ with probability at most $1/n^{\alpha-1}$ and $\Sigma_j(1 + C(I_j(y))) = \mathsf{O}(n)$, it can be argued that the case $r > \alpha \log n$ does not contribute significantly to the expectation of $\Sigma_j C(I_j(y))$.

To show that the expected number of children of an interval J at level i is bounded by a constant, we will show that the expected number of siblings of J (children of its parent) is bounded by a constant; in

fact, we will only bound the number of right siblings since the argument for the number of left siblings is identical. Let the intervals to the right of J be the following:

$$J_1 = [x_1, x_2]\,;\, J_2 = [x_2, x_3]\,;\, \ldots;\, J_k = [x_k, +\infty]\ .$$

Since these intervals exist at level i, each of the elements $x_1, \ldots, x_k$ belong to X_i. If J has s right siblings, then it must be the case that $x_1, \ldots, x_s \notin X_{i+1}$, and $x_{s+1} \in X_{i+1}$. The latter event occurs with probability $1/2^{s+1}$ since each element of X_i is independently chosen to be in X_{i+1} with probability $1/2$. Clearly, the number of right siblings of J can be viewed as a random variable that is geometrically distributed with parameter $1/2$. It follows that the expected number of right siblings of J is at most 2.

Consider now the implementation of the insert and delete operations. In implementing the operation insert(y, X), we assume that a random level $L(y)$ is chosen for y as described earlier. If $L(y) > r$, then we start by creating new levels from $r + 1$ to $L(y)$ and then redefine r to be $L(y)$. This requires $O(1)$ time per level, since the new levels are all empty prior to the insertion of y. Next we perform find(y, X) and determine the search path $I_r(y)$, ..., $I_1(y)$, where r is updated to its new value if necessary. Given this search path, the insertion can be accomplished in time $\mathsf{O}(L(y))$ by splitting around y the intervals $I_1(y), \ldots, I_{L(y)}(y)$ and updating the pointers as appropriate. The delete operation is the converse of the insert operation; it involves performing find(y, X) followed by collapsing the intervals that have y as an end-point. Both operations incur cost that is the cost of a find operation and additional cost proportional to $L(y)$. By Lemmas 15.1 and 15.2, we obtain the following theorem.

THEOREM 15.6 *In a random skip list for a set X of size n, the operations* find, insert, *and* delete *can be performed in expected time* $\mathsf{O}(\log n)$.

15.7 Random Reordering and Linear Programming

The *linear programming problem* is a particularly notable example of the two main benefits of randomization — simplicity and speed. We now describe a simple algorithm for linear programming based on a paradigm for randomized algorithms known as *random reordering.* For many problems it is possible to design natural algorithms based on the following idea. Suppose that the input consists of n elements. Given any subset of these n elements, there is a solution to the partial problem defined by these elements. If we start with the empty set and add the n elements of the input one at a time, maintaining a partial solution after each addition, we will obtain a solution to the entire problem when all the elements have been added. The usual difficulty with this approach is that the running time of the algorithm depends heavily on the order in which the input elements are added; for any fixed ordering, it is generally possible to force this algorithm to behave badly. The key idea behind random reordering is to *add the elements in a random order.* This simple device often avoids the pathological behavior that results from using a fixed order.

The linear programming problem is to find the extremum of a linear objective function of d real variables subject to a set H of n constraints that are linear functions of these variables. The intersection of the n half-spaces defined by the constraints is a polyhedron in d-dimensional space (which may be empty, or possibly unbounded). We refer to this polyhedron as the *feasible region.* Without loss of generality [34] we assume that the feasible region is nonempty and bounded. (Note that we are not assuming that we can *test* an arbitrary polyhedron for nonemptiness or boundedness; this is known to be equivalent to solving a linear program.) For a set of constraints S, let $\mathcal{B}(S)$ denote the optimum of the linear program defined by S; we seek $\mathcal{B}(S)$.

Consider the following algorithm due to Seidel [36]: add the n constraints in random order, one at a time. After adding each constraint, determine the optimum subject to the constraints added so far. This algorithm may also be viewed in the following "backwards" manner, which will prove useful in the sequel.

Algorithm SLP:

Input: A set of constraints H, and the dimension d.

Output: The optimum $\mathcal{B}(H)$.

0. If there are only d constraints, output $\mathcal{B}(H) = H$.

1. Pick a random constraint $h \in H$;
Recursively find $\mathcal{B}(H \backslash \{h\})$.

2.1. if $\mathcal{B}(H \backslash \{h\})$ does not violate h, output $\mathcal{B}(H \backslash \{h\})$ to be the optimum $\mathcal{B}(H)$.

2.2. else project all the constraints of $H \backslash \{h\}$ onto h and recursively solve this new linear programming problem of one lower dimension.

The idea of the algorithm is simple. Either h (the constraint chosen randomly in Step 1) is redundant (in which case we execute Step 2.1), or it is not. In the latter case, we know that the vertex formed by $\mathcal{B}(H)$ must lie on the hyperplane bounding h. In this case, we project all the constraints of $H \backslash \{h\}$ onto h and solve this new linear programming problem (which has dimension $d - 1$).

The optimum $\mathcal{B}(H)$ is defined by d constraints. At the top level of recursion, the probability that a random constraint h violates $\mathcal{B}(H \backslash \{h\})$ is at most d/n. Let $T(n, d)$ denote an upper bound on the expected running time of the algorithm for any problem with n constraints in d dimensions. Then, we may write

$$T(n, d) \leq T(n-1, d) + O(d) + \frac{d}{n} \left[O(dn) + T(n-1, d-1) \right] . \tag{15.7}$$

In (15.7), the first term on the right denotes the cost of recursively solving the linear program defined by the constraints in $H \backslash \{h\}$. The second accounts for the cost of checking whether h violates $\mathcal{B}(H \backslash \{h\})$. With probability d/n it does, and this is captured by the bracketed expression, whose first term counts the cost of projecting all the constraints onto h. The second counts the cost of (recursively) solving the projected problem, which has one fewer constraint and dimension. The following theorem may be verified by substitution, and proved by induction.

THEOREM 15.7 *There is a constant b such that the recurrence (15.7) satisfies the solution* $T(n, d) \leq bnd!$.

In contrast if the choice in Step 1 of **SLP** were not random, the recurrence (15.7) would be

$$T(n, d) \leq T(n-1, d) + O(d) + O(dn) + T(n-1, d-1) , \tag{15.8}$$

whose solution contains a term that grows quadratically in n.

15.8 Algebraic Methods and Randomized Fingerprints

Some of the most notable randomized results in theoretical computer science, particularly in complexity theory, have involved a nontrivial combination of randomization and algebraic methods. In this section we describe a fundamental randomization technique based on algebraic ideas. This is the randomized fingerprinting technique, originally due to Freivalds [10], for the verification of identities involving matrices, polynomials, and integers. We also describe how this generalizes to the so-called Schwartz–Zippel

technique for identities involving multivariate polynomials (independently due to Schwartz [35] and Zippel [46]; see also DeMillo and Lipton [6]). Finally, following Lovász [20], we apply the technique to the problem of detecting the existence of perfect matchings in graphs.

The *fingerprinting* technique has the following general form. Suppose we wish to decide the equality of two elements x and y drawn from some "large" universe U. Assuming any reasonable model of computation, this problem has a deterministic complexity $\Omega(\log |U|)$. Allowing randomization, an alternative approach is to choose a random function from U into a smaller space V such that with high probability x and y are identical if and only if their images in V are identical. These images of x and y are said to be their *fingerprints*, and the equality of fingerprints can be verified in time $O(\log |V|)$. Of course, for any fingerprint function the average number of elements of U mapped to an element of V is $|U|/|V|$; so, it would appear impossible to find good fingerprint functions that work for arbitrary or worst-case choices of x and y. However, as we will show below, when the identity-checking is only required to be correct for x and y chosen from a small subspace S of U, particularly a subspace with some algebraic structure, it is possible to choose good fingerprint functions without any *a priori* knowledge of the subspace, provided the size of V is chosen to be comparable to the size of S.

Throughout this section we will be working over some unspecified field $\mathcal{F}$. Since the randomization will involve uniform sampling from a finite subset of the field, we do not even need to specify whether the field is finite or not. The reader may find it helpful in the infinite case to assume that $\mathcal{F}$ is the field $\mathcal{Q}$ of rational numbers, and in the finite case to assume that $\mathcal{F}$ is $\mathcal{Z}_p$, the field of integers modulo some prime number p.

Freivalds' Technique and Matrix Product Verification

We begin by describing a fingerprinting technique for verifying matrix product identities. Currently, the fastest algorithm for matrix multiplication (due to Coppersmith and Winograd [5]) has running time $O(n^{2.376})$, improving significantly on the obvious $O(n^3)$ time algorithm; however, the fast matrix multiplication algorithm has the disadvantage of being extremely complicated. Suppose we have an implementation of the fast matrix multiplication algorithm and, given its complex nature, are unsure of its correctness. Since program verification appears to be an intractable problem, we consider the more reasonable goal of verifying the correctness of the output produced by executing the algorithm on specific inputs. (This notion of verifying programs on specific inputs is the basic tenet of the theory of *program checking* recently formulated by Blum and Kannan [4].) More concretely, suppose we are given three $n \times n$ matrices $\boldsymbol{X}$, $\boldsymbol{Y}$, and $\boldsymbol{Z}$ over a field $\mathcal{F}$, and would like to verify that $\boldsymbol{XY} = \boldsymbol{Z}$. Clearly, it does not make sense to use simpler but slower matrix multiplication algorithm for the verification, as that would defeat the whole purpose of using the fast algorithm in the first place. Observe that, in fact, there is no need to recompute $\boldsymbol{Z}$; rather, we are merely required to verify that the product of $\boldsymbol{X}$ and $\boldsymbol{Y}$ is indeed equal to $\boldsymbol{Z}$. Freivalds' technique gives an elegant solution that leads to an $O(n^2)$ time randomized algorithm with bounded error probability.

The idea is to first pick a random vector $\boldsymbol{r} \in \{0, 1\}^n$, i.e., each component of $\boldsymbol{r}$ is chosen independently and uniformly at random from the set $\{0, 1\}$ consisting of the additive and multiplicative identities of the field $\mathcal{F}$. Then, in $O(n^2)$ time, we can compute $\boldsymbol{y} = \boldsymbol{Yr}$, $\boldsymbol{x} = \boldsymbol{Xy} = \boldsymbol{XYr}$, and $\boldsymbol{z} = \boldsymbol{Zr}$. We would like to claim that the identity $\boldsymbol{XY} = \boldsymbol{Z}$ can be verified by merely checking that $\boldsymbol{x} = \boldsymbol{z}$. Quite clearly, if $\boldsymbol{XY} = \boldsymbol{Z}$ then $\boldsymbol{x} = \boldsymbol{z}$; unfortunately, the converse is not true in general. However, given the random choice of $\boldsymbol{r}$, we can show that for $\boldsymbol{XY} \neq \boldsymbol{Z}$, the probability that $\boldsymbol{x} \neq \boldsymbol{z}$ is at least $1/2$. Observe that the fingerprinting algorithm errs only if $\boldsymbol{XY} \neq \boldsymbol{Z}$ but $\boldsymbol{x}$ and $\boldsymbol{z}$ turn out to be equal, and this has a bounded probability.

THEOREM 15.8 *Let $\boldsymbol{X}$, $\boldsymbol{Y}$, and $\boldsymbol{Z}$ be $n \times n$ matrices over some field $\mathcal{F}$ such that $\boldsymbol{XY} \neq \boldsymbol{Z}$; further, let $\boldsymbol{r}$ be chosen uniformly at random from $\{0, 1\}^n$ and define $\boldsymbol{x} = \boldsymbol{XYr}$ and $\boldsymbol{z} = \boldsymbol{Zr}$. Then,*

$$\mathbf{Pr}[\boldsymbol{x} = \boldsymbol{z}] \leq 1/2 .$$

PROOF Define $\boldsymbol{W} = \boldsymbol{XY} - \boldsymbol{Z}$ and observe that $\boldsymbol{W}$ is not the all-zeroes matrix. Since $\boldsymbol{Wr} = \boldsymbol{XYr} - \boldsymbol{Zr} = \boldsymbol{x} - \boldsymbol{z}$, the event $\boldsymbol{x} = \boldsymbol{z}$ is equivalent to the event that $\boldsymbol{Wr} = 0$. Assume, without loss of generality, that the first row of $\boldsymbol{W}$ has a nonzero entry and that the nonzero entries in that row precede all the zero entries. Define the vector $\boldsymbol{w}$ as the first row of $\boldsymbol{W}$, and assume that the first $k > 0$ entries in $\boldsymbol{w}$ are nonzero. Since the first component of $\boldsymbol{Wr}$ is $\boldsymbol{w}^T\boldsymbol{r}$, giving an upper bound on the probability that the inner product of $\boldsymbol{w}$ and $\boldsymbol{r}$ is zero will give an upper bound on the probability that $\boldsymbol{x} = \boldsymbol{z}$.

Observe that $\boldsymbol{w}^T\boldsymbol{r} = 0$ if and only if

$$r_1 = \frac{-\sum_{i=2}^{k} w_i r_i}{w_1} . \tag{15.9}$$

Suppose that while choosing the random vector $\boldsymbol{r}$, we choose $r_2, \ldots, r_n$ before choosing r_1. After the values for $r_2, \ldots, r_n$ have been chosen, the right-hand side of (15.9) is fixed at some value $v \in \mathcal{F}$. If $v \notin \{0, 1\}$, then r_1 will never equal v; conversely, if $v \in \{0, 1\}$, then the probability that $r_1 = v$ is $1/2$. Thus, the probability that $\boldsymbol{w}^T\boldsymbol{r} = 0$ is at most $1/2$, implying the desired result.

We have reduced the matrix multiplication verification problem to that of verifying the equality of two vectors. The reduction itself can be performed in $O(n^2)$ time and the vector equality can be checked in $O(n)$ time, giving an overall running time of $O(n^2)$ for this Monte Carlo procedure. The error probability can be reduced to $1/2^k$ via k independent iterations of the Monte Carlo algorithm. Note that there was nothing magical about choosing the components of the random vector $\boldsymbol{r}$ from $\{0, 1\}$, since any two distinct elements of $\mathcal{F}$ would have done equally well. This suggests an alternative approach toward reducing the error probability, as follows: each component of $\boldsymbol{r}$ is chosen independently and uniformly at random from some subset $\mathcal{S}$ of the field $\mathcal{F}$; then, it is easily verified that the error probability is no more than $1/|\mathcal{S}|$.

Finally, note that Freivalds' technique can be applied to the verification of any matrix identity $\boldsymbol{A} = \boldsymbol{B}$. Of course, given $\boldsymbol{A}$ and $\boldsymbol{B}$, just comparing their entries takes only $O(n^2)$ time. But there are many situations where, just as in the case of matrix product verification, computing $\boldsymbol{A}$ explicitly is either too expensive or possibly even impossible, whereas computing $\boldsymbol{Ar}$ is easy. The random fingerprint technique is an elegant solution in such settings.

Extension to Identities of Polynomials

The fingerprinting technique due to Freivalds is fairly general and can be applied to many different versions of the identity verification problem. We now show that it can be easily extended to identity verification for symbolic polynomials, where two polynomials $P_1(x)$ and $P_2(x)$ are deemed identical if they have identical coefficients for corresponding powers of x. Verifying integer or string equality is a special case since we can represent any string of length n as a polynomial of degree n by using the kth element in the string to determine the coefficient of the kth power of a symbolic variable.

Consider first the polynomial product verification problem: given three polynomials $P_1(x)$, $P_2(x)$, $P_3(x) \in \mathcal{F}[x]$, we are required to verify that $P_1(x) \times P_2(x) = P_3(x)$. We will assume that $P_1(x)$ and $P_2(x)$ are of degree at most n, implying that $P_3(x)$ has degree at most $2n$. Note that degree n polynomials can be multiplied in $O(n \log n)$ time via fast Fourier transforms, and that the evaluation of a polynomial can be done in $O(n)$ time.

The randomized algorithm we present for polynomial product verification is similar to the algorithm for matrix product verification. It first fixes a set $\mathcal{S} \subseteq \mathcal{F}$ of size at least $2n + 1$ and chooses $r \in \mathcal{S}$ uniformly at random. Then, after evaluating $P_1(r)$, $P_2(r)$ and $P_3(r)$ in $O(n)$ time, the algorithm declares the identity $P_1(x)P_2(x) = P_3(x)$ to be correct if and only if $P_1(r)P_2(r) = P_3(r)$. The algorithm makes an error only in the case where the polynomial identity is false but the value of the three polynomials at r indicates otherwise. We will show that the error event has a bounded probability.

Consider the degree $2n$ polynomial $Q(x) = P_1(x)P_2(x) - P_3(x)$. The polynomial $Q(x)$ is said to be *identically zero,* denoted by $Q(x) \equiv 0$, if each of its coefficients equals zero. Clearly, the polynomial

identity $P_1(x)P_2(x) = P_3(x)$ holds if and only if $Q(x) \equiv 0$. We need to establish that if $Q(x) \not\equiv 0$, then with high probability $Q(r) = P_1(r)P_2(r) - P_3(r) \neq 0$. By elementary algebra we know that $Q(x)$ has at most $2n$ distinct roots. It follows that unless $Q(x) \equiv 0$, not more that $2n$ different choices of $r \in \mathcal{S}$ will cause $Q(r)$ to evaluate to 0. Therefore, the error probability is at most $2n/|\mathcal{S}|$. The probability of error can be reduced either by using independent iterations of this algorithm, or by choosing a larger set $\mathcal{S}$. Of course, when $\mathcal{F}$ is an infinite field (e.g., the reals), the error probability can be made 0 by choosing r uniformly from the entire field $\mathcal{F}$; however, that requires an infinite number of random bits!

Note that we could also use a deterministic version of this algorithm where each choice of $r \in \mathcal{S}$ is tried once. But this involves $2n + 1$ different evaluations of each polynomial, and the best known algorithm for multiple evaluations needs $\Theta(n\log^2 n)$ time, which is more than the $\mathsf{O}(n \log n)$ time requirement for actually performing a multiplication of the polynomials $P_1(x)$ and $P_2(x)$.

This verification technique is easily extended to a generic procedure for testing any polynomial identity of the form $P_1(x) = P_2(x)$ by converting it into the identity $Q(x) = P_1(x) - P_2(x) \equiv 0$. Of course, when P_1 and P_2 are explicitly provided, the identity can be deterministically verified in $\mathsf{O}(n)$ time by comparing corresponding coefficients. Our randomized technique will take just as long to merely evaluate $P_1(x)$ and $P_2(x)$ at a random value. However, as in the case of verifying matrix identities, the randomized algorithm is quite useful in situations where the polynomials are implicitly specified, e.g., when we only have a "black box" for computing the polynomials with no information about their coefficients, or when they are provided in a form where computing the actual coefficients is expensive. An example of the latter situation is provided by the following problem concerning the determinant of a symbolic matrix. In fact, the determinant problem will require a technique for the verification of polynomial identities of *multivariate* polynomials that we will discuss shortly.

Consider an $n \times n$ matrix $\boldsymbol{M}$. Recall that the determinant of the matrix $\boldsymbol{M}$ is defined as follows:

$$\det(\boldsymbol{M}) = \sum_{\pi \in \mathcal{S}_n} \operatorname{sgn}(\pi) \prod_{i=1}^{n} M_{i,\pi(i)} \,, \tag{15.10}$$

where $\mathcal{S}_n$ is the symmetric group of permutations of order n, and $\operatorname{sgn}(\pi)$ is the sign of a permutation π. (The sign function is defined to be $\operatorname{sgn}(\pi) = (-1)^t$, where t is the number of pairwise exchanges required to convert the identity permutation into π.) Although the determinant is defined as a summation with $n!$ terms, it is easily evaluated in polynomial time provided that the matrix entries M_{ij} are explicitly specified. Consider the Vandermonde matrix $\boldsymbol{M}(x_1, \ldots, x_n)$ which is defined in terms of the indeterminates $x_1, \ldots, x_n$ such that $M_{ij} = x_i^{j-1}$, i.e.,

$$\boldsymbol{M} = \begin{pmatrix} 1 & x_1 & x_1^2 & \ldots & x_1^{n-1} \\ 1 & x_2 & x_2^2 & \ldots & x_2^{n-1} \\ & & & \cdot & \\ & & & \cdot & \\ & & & \cdot & \\ 1 & x_n & x_n^2 & \ldots & x_n^{n-1} \end{pmatrix} .$$

It is known that for the Vandermonde matrix, $\det(\boldsymbol{M}) = \prod_{i<j}(x_i - x_j)$. Consider the problem of verifying this identity without actually devising a formal proof. Computing the determinant of a symbolic matrix is infeasible as it requires dealing with a summation over $n!$ terms. However, we can formulate the identity verification problem as the problem of verifying that the polynomial $Q(x_1, \ldots, x_n) = \det(\boldsymbol{M}) - \prod_{i<j}(x_i - x_j)$ is identically zero. Based on our discussion of Freivalds' technique, it is natural to consider the substitution of random values for each x_i. Since the determinant can be computed in polynomial time for any specific assignment of values to the symbolic variables $x_1, \ldots, x_n$, it is easy to evaluate the polynomial Q for random values of the variables. The only issue is that of bounding the error probability for this randomized test.

We now extend the analysis of Freivalds' technique for univariate polynomials to the multivariate case. But first, note that in a multivariate polynomial $Q(x_1, \ldots, x_n)$, the degree of a term is the sum of the exponents of the variable powers that define it, and the total degree of Q is the maximum over all terms of the degrees of the terms.

THEOREM 15.9 *Let $Q(x_1, \ldots, x_n) \in \mathcal{F}[x_1, \ldots, x_n]$ be a multivariate polynomial of total degree m. Let $\mathcal{S}$ be a finite subset of the field $\mathcal{F}$, and let $r_1, \ldots, r_n$ be chosen uniformly and independently from $\mathcal{S}$. Then,*

$$\mathbf{Pr}\left[Q(r_1, \ldots, r_n) = 0 \mid Q(x_1, \ldots, x_n) \not\equiv 0\right] \leq \frac{m}{|\mathcal{S}|} .$$

PROOF We will proceed by induction on the number of variables n. The basis of the induction is the case $n = 1$, which reduces to verifying the theorem for a univariate polynomial $Q(x_1)$ of degree m. But we have already seen for $Q(x_1) \not\equiv 0$, the probability that $Q(r_1) = 0$ is at most $m/|\mathcal{S}|$, taking care of the basis.

We now assume that the induction hypothesis holds for multivariate polynomials with at most $n - 1$ variables, where $n > 1$. In the polynomial $Q(x_1, \ldots, x_n)$ we can factor out the variable x_1 and thereby express Q as

$$Q(x_1, \ldots, x_n) = \sum_{i=0}^{k} x_1^i P_i(x_2, \ldots, x_n) ,$$

where $k \leq m$ is the largest exponent of x_1 in Q. Given our choice of k, the coefficient $P_k(x_2, \ldots, x_n)$ of x_1^k cannot be identically zero. Note that the total degree of P_k is at most $m - k$. Thus, by the induction hypothesis, we conclude that the probability that $P_k(r_2, \ldots, r_n) = 0$ is at most $(m - k)/|\mathcal{S}|$.

Consider now the case where $P_k(r_2, \ldots, r_n)$ is indeed not equal to 0. We define the following univariate polynomial over x_1 by substituting the random values for the other variables in Q:

$$q(x_1) = Q(x_1, r_2, r_3, \ldots, r_n) = \sum_{i=0}^{k} x_1^i P_i(r_2, \ldots, r_n) .$$

Quite clearly, the resulting polynomial $q(x_1)$ has degree k and is not identically zero (since the coefficient of x_1^k is assumed to be nonzero). As in the basis case, we conclude that the probability that $q(r_1) = Q(r_1, r_2, \ldots, r_n)$ evaluates to 0 is bounded by $k/|\mathcal{S}|$.

By the preceding arguments, we have established the following two inequalities:

$$\begin{aligned} \mathbf{Pr}\left[P_k(r_2, \ldots, r_n) = 0\right] &\leq \frac{m-k}{|\mathcal{S}|} ; \\ \mathbf{Pr}\left[Q(r_1, r_2, \ldots, r_n) = 0 \mid P_k(r_2, \ldots, r_n) \neq 0\right] &\leq \frac{k}{|\mathcal{S}|} . \end{aligned}$$

Using the elementary observation that for any two events $\mathcal{E}_1$ and $\mathcal{E}_2$, $\mathbf{Pr}[\mathcal{E}_1] \leq \mathbf{Pr}[\mathcal{E}_1 \mid \overline{\mathcal{E}}_2] + \mathbf{Pr}[\mathcal{E}_2]$, we obtain that the probability that $Q(r_1, r_2, \ldots, r_n) = 0$ is no more than the sum of the two probabilities on the right-hand side of the two obtained inequalities, which is $m/|\mathcal{S}|$. This implies the desired result.

This randomized verification procedure has one serious drawback: when working over large (or possibly infinite) fields, the evaluation of the polynomials could involve large intermediate values, leading to inefficient implementation. One approach to dealing with this problem in the case of integers is to perform all computations modulo some small random prime number; it can be shown that this does not have any adverse effect on the error probability.

Detecting Perfect Matchings in Graphs

We close by giving a surprising application of the techniques from the preceding section. Let $G(U, V, E)$ be a bipartite graph with two independent sets of vertices $U = \{u_1, \ldots, u_n\}$ and $V = \{v_1, \ldots, v_n\}$, and edges E that have one end-point in each of U and V. We define a matching n G as a collection of edges $M \subseteq E$ such that each vertex is an end-point of at most one edge in M; further, a perfect matching is defined to be a matching of size n, i.e., where each vertex occurs as an end-point of exactly one edge in M. Any perfect matching M may be put into a 1-to-1 correspondence with the permutations in $\mathcal{S}_n$, where the matching corresponding to a permutation $\pi \in \mathcal{S}_n$ is given by the collection of edges $\{(u_i, v_{\pi(i)}) \mid 1 \leq i \leq n\}$. We now relate the matchings of the graph to the determinant of a matrix obtained from the graph.

THEOREM 15.10 *For any bipartite graph $G(U, V, E)$, define a corresponding $n \times n$ matrix $\boldsymbol{A}$ as follows:*

$$A_{ij} = \begin{cases} x_{ij} & (u_i, v_j) \in E \\ 0 & (u_i, v_j) \notin E \end{cases} .$$

Let the multivariate polynomial $Q(x_{11}, x_{12}, \ldots, x_{nn})$ denote the determinant $\det(\boldsymbol{A})$. *Then, G has a perfect matching if and only if $Q \not\equiv 0$.*

PROOF We may express the determinant of $\boldsymbol{A}$ as follows:

$$\det(\boldsymbol{A}) = \sum_{\pi \in \mathcal{S}_n} \operatorname{sgn}(\pi) A_{1,\pi(1)} A_{2,\pi(2)} \cdots A_{n,\pi(n)} .$$

Note that there cannot be any cancellation of the terms in the summation since each indeterminate x_{ij} occurs at most once in $\boldsymbol{A}$. Thus, the determinant is not identically zero if and only if there exists some permutation π for which the corresponding term in the summation is nonzero. Clearly, the term corresponding to a permutation π is non zero if and only if $A_{i,\pi(i)} \neq 0$ for each i, $1 \leq i \leq n$; this is equivalent to the presence in G of the perfect matching corresponding to π.

The matrix of indeterminates is sometimes referred to as the *Edmonds matrix* of a bipartite graph. The above result can be extended to the case of non-bipartite graphs, and the corresponding matrix of indeterminates is called the Tutte matrix. Tutte [41] first pointed out the close connection between matchings in graphs and matrix determinants; the simpler relation between bipartite matchings and matrix determinants was given by Edmonds [7].

We can turn the above result into a simple randomized procedure for testing the existence of perfect matchings in a bipartite graph (due to Lovász [20]): using the algorithm from Section "Extension to Identities of Polynomials," determine whether the determinant is identically zero or not. The running time of this procedure is dominated by the cost of computing a determinant, which is essentially the same as the time required to multiply two matrices. Of course, there are algorithms for *constructing* a maximum matching in a graph with m edges and n vertices in time $O(m\sqrt{n})$ (see Hopcroft and Karp [13], Micali and Vazirani [22, 43], and Feder and Motwani [8]). Unfortunately, the time required to compute the determinant exceeds $m\sqrt{n}$ for small m, and so the benefit in using this randomized *decision* procedure appears marginal at best. However, this technique was extended by Rabin and Vazirani [30, 31] to obtain simple algorithms for the actual *construction* of maximum matchings; although their randomized algorithms for matchings are simple and elegant, they are still slower than the deterministic $O(m\sqrt{n})$ time algorithms known earlier. Perhaps more significantly, this randomized decision procedure proved to be an essential ingredient in devising fast *parallel* algorithms for computing maximum matchings [18, 26].

15.9 Research Issues and Summary

Perhaps the most important research issue in the area of randomized algorithms is to prove or disprove that are problems solvable in polynomial time by either Las Vegas or Monte Carlo algorithms, but cannot be solved in polynomial time by any deterministic algorithm. Another important direction for future work is to devise high quality pseudo-random number generators, which take a small seed of truly random bits and stretch it into a much longer string that can be used as the random string to fuel randomized algorithms.

15.10 Defining Terms

Deterministic algorithm: An algorithm whose execution is completely determined by its input.

Distributional complexity: The expected running time of the best possible deterministic algorithm over the worst possible probability distribution on the inputs.

Las Vegas algorithm: A randomized algorithm that always produces correct results, with the only variation from one run to another being in its running time.

Monte Carlo algorithm: A randomized algorithm that may produce incorrect results, but with bounded error probability.

Randomized algorithm: An algorithm that makes random choices during the course of its execution.

Randomized complexity: The expected running time of the best possible randomized algorithm over the worst input.

References

[1] Aleliunas, R., Karp, R.M., Lipton, R.J. Lovász, L., and Rackoff, C., Random walks, universal traversal sequences, and the complexity of maze problems. In *Proceedings of the 20th Annual Symposium on Foundations of Computer Science,* 218–223, San Juan, Puerto Rico, Oct. 1979.

[2] Aragon, C.R. and Seidel, R.G., Randomized search trees. In *Proceedings of the 30th Annual IEEE Symposium on Foundations of Computer Science,* 540–545, 1989.

[3] Ben-David, S., Borodin, A., Karp, R.M., Tardos, G., and Wigderson, A., On the power of randomization in on-line algorithms. *Algorithmica,* 11(1), 2–14, 1994.

[4] Blum, M. and Kannan, S., Designing programs that check their work. In *Proceedings of the 21st Annual ACM Symposium on Theory of Computing,* 86–97, ACM, 1989.

[5] Coppersmith, D. and Winograd, S., Matrix multiplication via arithmetic progressions. *Journal of Symbolic Computation,* 9, 251–280, 1990.

[6] DeMillo, R.A. and Lipton, R.J., A probabilistic remark on algebraic program testing. *Information Processing Letters,* 7, 193–195, 1978.

[7] Edmonds, J., Systems of distinct representatives and linear algebra. *Journal of Research of the National Bureau of Standards, 71B,* 4, 241–245, 1967.

[8] Feder, T. and Motwani, R., Clique partitions, graph compression and speeding-up algorithms. In *Proceedings of the 25th Annual ACM Symposium on Theory of Computing,* 123–133, 1991.

[9] Floyd, R.W. and Rivest, R.L., Expected time bounds for selection. *Communications of the ACM,* 18, 165–172, 1975.

[10] Freivalds, R., Probabilistic machines can use less running time. In *Information Processing 77, Proceedings of IFIP Congress 77,* 839–842, Gilchrist, B., Ed., Amsterdam, Aug. 1977. North-Holland Publishing Company.

[11] Goemans, M.X. and Williamson, D.P., 0.878-approximation algorithms for MAX-CUT and MAX-2SAT. In *Proceedings of the 26th Annual ACM Symposium on Theory of Computing,* 422–431, 1994.
[12] Hoare, C.A.R., Quicksort. *Computer Journal,* 5, 10–15, 1962.
[13] Hopcroft, J.E. and Karp, R.M., An $n^{5/2}$ algorithm for maximum matching in bipartite graphs. *SIAM Journal on Computing,* 2, 225–231, 1973.
[14] Karger, D.R., Global min-cuts in $\mathcal{RNC}$, and other ramifications of a simple min-cut algorithm. In *Proceedings of the 4th Annual ACM-SIAM Symposium on Discrete Algorithms,* 1993.
[15] Karger, D.R., Klein, P.N., and Tarjan, R.E., A randomized linear-time algorithm for finding minimum spanning trees. *Journal of the ACM,* 42, 321–328, 1995.
[16] Karger, D., Motwani, R., and Sudan, M., Approximate graph coloring by semidefinite programming. In *Proceedings of the 35th Annual IEEE Symposium on Foundations of Computer Science,* 2–13, 1994.
[17] Karp, R.M., An introduction to randomized algorithms. *Discrete Applied Mathematics,* 34, 165–201, 1991.
[18] Karp, R.M., Upfal, E., and Wigderson, A., Constructing a perfect matching is in random $\mathcal{NC}$. *Combinatorica,* 6, 35–48, 1986.
[19] Karp, R.M., Upfal, E., and Wigderson, A., The complexity of parallel search. *Journal of Computer and System Sciences,* 36, 225–253, 1988.
[20] Lovász, L., On determinants, matchings and random algorithms. In *Fundamentals of Computing Theory,* Budach, L., Ed., Akademia-Verlag, Berlin, 1979.
[21] Maffioli, F., Speranza, M.G., and Vercellis, C., Randomized algorithms. In *Combinatorial Optimization: Annotated Bibliographies,* 89–105. O'hEigertaigh, M., Lenstra, J.K., and Rinooy Kan, A.H.G., Eds., John Wiley & Sons, New York, 1985.
[22] Micali, S. and Vazirani, V.V., An $O(\sqrt{|V|}|e|)$ algorithm for finding maximum matching in general graphs. In *Proceedings of the 21st Annual IEEE Symposium on Foundations of Computer Science,* 17–27, 1980.
[23] Motwani, R., Naor, J., and Raghavan, P., Randomization in approximation algorithms. In *Approximation Algorithms,* Hochbaum, D., Ed., PWS, 1996.
[24] Motwani, R. and Raghavan, P., *Randomized Algorithms.* Cambridge University Press, New York, 1995.
[25] Mulmuley, K., *Computational Geometry: An Introduction Through Randomized Algorithms.* Prentice Hall, New York, 1993.
[26] Mulmuley, K., Vazirani, U.V., and Vazirani, V.V., Matching is as easy as matrix inversion. *Combinatorica,* 7, 105–113, 1987.
[27] Pugh, W., Skip lists: A probabilistic alternative to balanced trees. *Communications of the ACM,* 33(6), 668–676, 1990.
[28] Rabin, M.O., Probabilistic algorithm for testing primality. *Journal of Number Theory,* 12, 128–138, 1980.
[29] Rabin, M.O., Randomized Byzantine generals. In *Proceedings of the 24th Annual Symposium on Foundations of Computer Science,* 403–409, 1983.
[30] Rabin, M.O. and Vazirani, V.V., Maximum matchings in general graphs through randomization. Technical Report TR-15-84, Aiken Computation Laboratory, Harvard University, 1984.
[31] Rabin, M.O. and Vazirani, V.V., Maximum matchings in general graphs through randomization. *Journal of Algorithms,* 10, 557–567, 1989.
[32] Raghavan, P. and Snir, M., Memory versus randomization in on-line algorithms. *IBM Journal of Research and Development,* 38, 683–707, 1994.
[33] Saks, M. and Wigderson, A., Probabilistic Boolean decision trees and the complexity of evaluating game trees. In *Proceedings of the 27th Annual IEEE Symposium on Foundations of Computer Science,* 29–38, Toronto, Ontario, 1986.

[34] Schrijver, A., *Theory of Linear and Integer Programming.* John Wiley & Sons, New York, 1986.
[35] Schwartz, J.T., Fast probabilistic algorithms for verification of polynomial identities. *Journal of the ACM,* 27(4), 701–717, Oct. 1980.
[36] Seidel, R.G., Small-dimensional linear programming and convex hulls made easy. *Discrete and Computational Geometry,* 6, 423–434, 1991.
[37] Sinclair, A., *Algorithms for Random Generation and Counting: A Markov Chain Approach.* Progress in Theoretical Computer Science. Birkhäuser, Boston, 1992.
[38] Snir, M., Lower bounds on probabilistic linear decision trees. *Theoretical Computer Science,* 38, 69–82, 1985.
[39] Solovay, R. and Strassen, V., A fast Monte-Carlo test for primality. *SIAM Journal on Computing,* 6(1), 84–85, Mar. 1977. See also *SIAM Journal on Computing 7,* 118, 1 Feb. 1978.
[40] Tarsi, M., Optimal search on some game trees. *Journal of the ACM,* 30, 389–396, 1983.
[41] Tutte, W.T., The factorization of linear graphs. *Journal of the London Math. Soc.,* 22, 107–111, 1947.
[42] Valiant, L.G., A scheme for fast parallel communication. *SIAM Journal on Computing,* 11, 350–361, 1982.
[43] Vazirani, V.V., A theory of alternating paths and blossoms for proving correctness of $O(\sqrt{V}E)$ graph maximum matching algorithms. *Combinatorica,* 14(1), 71–109, 1994.
[44] Welsh, D.J.A., Randomised algorithms. *Discrete Applied Mathematics,* 5, 133–145, 1983.
[45] Yao, A.C-C., Probabilistic computations: Towards a unified measure of complexity. In *Proceedings of the 17th Annual Symposium on Foundations of Computer Science,* 222–227, 1977.
[46] Zippel, R.E., Probabilistic algorithms for sparse polynomials. In *Proceedings of EUROSAM 79,* volume 72 of *Lecture Notes in Computer Science,* 216–226, Marseille, 1979.

Further Information

In this section we give pointers to a plethora of randomized algorithms not covered here. The reader should also note that the examples above are but a (random!) sample of the many randomized algorithms for each of the problems considered. These algorithms have been chosen to illustrate the main ideas behind randomized algorithms, rather than to represent the state of the art for these problems. The reader interested in other algorithms for these problems is referred to the book by Motwani and Raghavan [24].

Randomized algorithms also find application in a number of other areas: in load-balancing [42], approximation algorithms and combinatorial optimization [11, 16, 23], graph algorithms [1, 15], data structures [2], counting and enumeration [37], parallel algorithms [18, 19], distributed algorithms [29], geometric algorithms [25], online algorithms [3, 32], and number-theoretic algorithms [28, 39]. The reader interested in these applications may consult these articles or the book by Motwani and Raghavan [24].

16

Algebraic Algorithms[1]

Angel Díaz
IBM T.J. Watson Research Center

Ioannis Z. Emiris
INRIA Sophia-Antipolis

Erich Kaltofen
North Carolina State University

Victor Y. Pan
City University of New York

16.1 Introduction

The title's subject is the algorithmic approach to algebra: arithmetic with numbers, polynomials, matrices, differential polynomials, such as $y'' + (1/2 + x^4/4)y$, truncated series, and algebraic sets, i.e., quantified expressions such as $\exists x \in \mathbb{R}: x^4 + p \cdot x + q = 0$, which describes a subset of the two-dimensional space with

[1] *Angel Díaz*—Supported by NSF grant No. CCR-9319776 and by GTE under a Graduate Computer Science Fellowship.
Ioannis Z. Emiris—Supported by the European Union under ESPRIT FRISCO project LTR 21.024.
Erich Kaltofen—Supported by NSF grant No. CCR-9319776 and CCR-9712267.
Victor Y. Pan—Supported by NSF grants Nos. CCR-9020690 and CCR-9625344 and by PSC CUNY Awards 666327 and 667340.
Angel Díaz and Erich Kaltofen—Part of this work was done while the first and third authors were at the Department of Computer Science at Rensselaer Polytechnic Institute in Troy, New York.

coordinates p and q for which the given quartic equation has a real root. Algorithms that manipulate such objects are the backbone of modern symbolic mathematics software such as the Maple and Mathematica systems, to name but two among many useful systems. This chapter restricts itself to algorithms in four areas: linear algebra, root finding for univariate polynomials, solution of systems of nonlinear algebraic equations, and polynomial factorization (see Section 16.5 on some pointers to the vast further material on algebraic algorithms and "Some Computations Related to Matrix Multiplication" and [118] on further applications to the graph and combinatorial computations).

16.2 Matrix Computations and Approximation of Polynomial Zeros

This section covers several major algebraic and numerical problems of scientific and engineering computing that are usually solved numerically, with rounding-off or chopping the input and computed values to a fixed number of bits that fit the computer precision (Sections 16.3 and 16.4 are devoted to some fundamental infinite precision symbolic computations, and in "Rational Matrix Computations, Computations in Finite Fields, and Semirings" we comment on the infinite precision techniques for some matrix computations). We also study approximation of polynomial zeros, which is an important, fundamental, and very popular subject, too. In our presentation, we will very briefly list the major subtopics of our huge subject and will give some pointers to the bibliography. We will include brief coverage of the topics of the algorithm design and analysis, regarding the complexity of matrix computation and of approximating polynomial zeros. The reader may find further material on these subjects and further references in the survey articles [112, 115, 116, 124] and in the books [12, 13, 36, 62, 64, 68, 140, 145].

Products of Vectors and Matrices, Convolution of Vectors

An $m \times n$ matrix $A = [\ a_{i,j}\ ,\ i = 0, 1, \ldots, m-1;\ j = 0, 1, \ldots, n-1\ ]$ is a 2-dimensional array, whose (i, j)-entry is $[A]_{i,j} = a_{i,j}$. A is a column vector of dimension m if $n = 1$ and is a row vector of dimension n if $m = 1$. Transposition, hereafter, indicated by the superscript T, transforms a row vector $\vec{v}^{\,T} = [\ v_0, \ldots, v_{n-1}\ ]$ into a column vector $\vec{v} = [\ v_0, \ldots, v_{n-1}\ ]^T$.

For two vectors, $\vec{u} = [\ u_0, \ldots, u_{m-1}]^T$ and $\vec{v} = [\ v_0, \ldots, v_{n-1}\ ]^T$, their *outer product* is an $m \times n$ matrix,

$$W = \vec{u}\vec{v}^{\,T} = \left[w_{i,j}\ ,\ i = 0, \ldots, m-1; j = 0, \ldots, n-1\right]\ ,$$

where $w_{i,j} = u_i\ v_j$, for all i and j, and their *convolution* vector is said to equal

$$\vec{w} = \vec{u} \circ \vec{v} = [w_0, \ldots, w_{m+n-2}]^T\ ,\quad w_k = \sum_{i=0}^{k} u_i v_{k-i}\ ,$$

where $u_i = v_j = 0$, for $i \geq m,\ j \geq n$; in fact, $\vec{w}$ is the coefficient vector of the product of 2 polynomials,

$$u(x) = \sum_{i=0}^{m-1} u_i x^i \quad \text{and} \quad v(x) = \sum_{i=0}^{n-1} v_i x^i\ ,$$

having coefficient vectors $\vec{u}$ and $\vec{v}$, respectively.

If $m = n$, then the scalar value

$$\vec{v}^{\,T}\ \vec{u} = \vec{u}^{\,T}\ \vec{v} = u_0v_0 + u_1v_1 + \cdots + u_{n-1}v_{n-1} = \sum_{i=0}^{n-1} u_i v_i$$

is called the *inner* (*dot,* or *scalar*) *product* of $\vec{u}$ and $\vec{v}$.

The straightforward algorithms compute the inner and outer products of $\vec{u}$ and $\vec{v}$ and their convolution vector by using $2n-1$, mn, and $mn+(m-1)(n-1) = 2mn-m-n+1$ arithmetic operations (hereafter, referred to as **ops**), respectively. By counting the ops involved, we estimate the arithmetic cost of the computations. This is a realistic measure for the computational complexity if the precision of computing is within the range of the computer precision. In practical numerical computations with rounding-off, the latter requirement is usually stated in terms of *conditioning of the computational problem* and *numerical stability of algorithms* [62, 68, 145], and [12, ch. 3]. A more universal and computer independent measure is given by the number of Boolean (bit) operations involved into the computations, which grows with the growth of both ops number and precision of computing.

The above upper bounds on the numbers of ops for computing the inner and outer products are sharp, that is, cannot be decreased, for the general pair of the input vectors $\vec{u}$ and $\vec{v}$, whereas (see, e.g., [12]) one may apply the *fast Fourier transform* (FFT) (see Chapter 17) in order to compute the convolution vector $\vec{u} \circ \vec{v}$ much faster, for larger m and n; namely, it suffices to use $4.5K \log K + 2K$ ops, for $K = 2^k$, $k = \lceil \log(m+n-1) \rceil$. (Here and hereafter, all logarithms are binary unless specified otherwise.)

If $A = [a_{i,j}]$ and $B = [b_{j,k}]$ are $m \times n$ and $n \times p$ matrices, respectively, and $\vec{v} = [v_k]$ is a p-dimensional vector, then the straightforward algorithms compute the vector

$$\vec{w} = B\vec{v} = [\, w_0, \ldots, w_{n-1} \,]^T, \qquad w_i = \sum_{j=0}^{p-1} b_{i,j} v_j, \quad i = 0, \ldots, n-1,$$

by using $(2p-1)n$ ops (sharp bound), and compute the *matrix product*

$$AB = \left[w_{i,k},\ i = 0, \ldots, m-1;\ k = 0, \ldots, p-1 \right]$$

by using $2mnp - mp$ ops, which is $2n^3 - n^2$ if $m = n = p$. The latter upper bound is not sharp: the subroutines for $n \times n$ matrix multiplication on some modern computers, such as CRAY and Connection Machines, rely on algorithms using $O(n^{2.81})$ ops [62]. Such algorithms rely on representing an $n \times n$ matrix A, for $n = 2^s$, as a 2×2 block matrix,

$$A = \begin{bmatrix} A_{0,0} & A_{0,1} \\ A_{1,0} & A_{1,1} \end{bmatrix},$$

where the blocks $A_{i,j}$, $i, j = 0, 1$, are $(n/2) \times (n/2)$ matrices. Multiplication of a pair of such 2×2 block matrices is reduced to 7 multiplications of $(n/2) \times (n/2)$ matrices represented as linear combinations of the input blocks and to 15 or 18 matrix additions/subtractions. For block multiplications, one applies the same algorithm recursively, until arriving at matrices of a small size. More involved technically advanced but nonpractical algorithms decrease the exponent from 2.81 below 2.376 [12, 13, 32].

If all the input entries and components are bounded integers having short binary representation, each of the above operations with vectors and matrices can be reduced to a single multiplication of 2 longer integers, by means of the techniques of *binary segmentation* (cf. [113, Sect. 40]; [115, 117], or [12, Examples 3.9.1–3.9.3]). The Boolean cost of the computations does not decrease much or even at all, but the techniques may be practically useful if the 2 longer integers still fit the computer precision.

For an $n \times n$ matrix B and an n-dimensional vector $\vec{v}$, one frequently needs to compute the vectors $B^i \vec{v}$, $i = 1, 2, \ldots, k-1$, which define *Krylov sequence* or *Krylov matrix*

$$\left[B^i \vec{v},\ i = 0, 1, \ldots, k-1 \right],$$

[62, 64]. The straightforward sequential algorithm takes on $(2n-1)n(k-1)$ ops, which is order n^3 if k is of order n. An alternative algorithm (most effective for parallel computation) first computes the matrix powers

$$B^2, B^4, B^8, \ldots, B^{2^s}, \qquad s = \lceil \log k \rceil - 1,$$

and then, the products of $n \times n$ matrices B^{2^i} by $n \times 2^i$ matrices, for $i = 0, 1, \ldots, s$:

$$\begin{aligned} & B \quad v\,, \\ & B^2 \quad [\, v,\ Bv\,] = \left[\, B^2v,\ B^3v\,\right], \\ & B^4 \quad \left[\, v,\ Bv,\ B^2v,\ B^3v\,\right] = \left[\, B^4v,\ B^5v,\ B^6v,\ B^7v\,\right], \\ & \vdots \end{aligned}$$

The last step completes the evaluation of the Krylov sequence, which amounts to $2s + 1$ matrix multiplications, and, therefore, can be performed (in theory) in $O(n^{2.376} \log k)$ ops, for $k = n$ [32].

Some Computations Related to Matrix Multiplication

Several fundamental matrix computations can be ultimately reduced to relatively few (that is, to a constant number, or, say, to $O(\log n)$) $n \times n$ matrix multiplications. The list of such computations includes the evaluation of the **determinant,** $det\ A$, of an $n \times n$ matrix A; its *inverse* A^{-1} (where A is nonsingular, that is, where $det\ A \neq 0$); the coefficients of its **characteristic polynomial,** $c_A(x) = det(xI - A)$, x denoting a scalar variable and I being the $n \times n$ identity matrix, which has ones on its diagonal and zeros elsewhere; its *minimal polynomial,* $m_A(x)$; its *rank,* rank A; the solution vector $\vec{x} = A^{-1}\,\vec{v}$ to a nonsingular *linear system of equations,* $A\,\vec{x} = \vec{v}$; various *orthogonal* and *triangular factorizations* of A, and a submatrix of A having the maximal rank, as well as some fundamental *computations with singular matrices.* Furthermore, similar reductions to matrix multiplication have been obtained for some apparently quite distant computational problems, including some major problems of combinatorial and graph computations such as Boolean matrix multiplication and computing the transitive closure of a graph [1], computing all pair shortest distances in graphs [12, p. 222], and pattern recognition. Consequently, all these operations can be performed by using (theoretically) $O(n^{2.376})$ ops (cf. [12, chap. 2]). One of the basic ideas is to represent the input matrix A as a block matrix and, operating with its blocks (rather than with its entries), to apply fast matrix multiplication algorithms. In particular, suppose that all the square northwestern blocks of A (called leading principal submatrices) are nonsingular. (This assumption holds for a large and practically important class of matrices, it also can be achieved by means of symmetrization of A or by randomization). Then one may compute det A and A^{-1} by factoring A as a 2×2 block matrix,

$$A = \begin{bmatrix} I & O \\ A_{1,0}A_{0,0}^{-1} & I \end{bmatrix} \begin{bmatrix} A_{0,0} & O \\ O & S \end{bmatrix} \begin{bmatrix} I & A_{0,0}^{-1}A_{0,1} \\ O & I \end{bmatrix},$$

$S = A_{1,1} - A_{1,0}A_{0,0}^{-1}A_{0,1}$, I and O denoting the identity and null matrices, respectively, and then recursively factorizing $A_{0,0}$ and S (see, e.g.,[12, sect. 2.2]). This is very close to recursive 2×2 block version of Gaussian elimination (cf. "Gaussian Elimination Algorithm"). Many of such block matrix algorithms are practically important for parallel computations (see "Parallel Matrix Computations"). On the other hand, however, due to various other considerations (accounting, in particular, for the overhead constants hidden in the $''O''$ notation, for the memory space requirements, and particularly, for numerical stability problems), these computations are, in practice, based either on the straightforward algorithm for matrix multiplication or on other methods allowing order n^3 arithmetic operations (cf. [62]).

In the next 3 sections, we will more closely consider the solution of a linear system of equations, $A\,\vec{v} = \vec{b}$, which is the most frequent operation in practice of scientific and engineering computing and is highly important theoretically. We will partition the known solution methods depending on whether the coefficient matrix A is *dense and unstructured, sparse,* or *dense and structured.* We will omit the study of singular (in particular, over- and underdetermined) linear systems, their least-squares solution, and various computations with singular matrices, referring the reader to [12, 62, 117]. Also, we refer the reader

to [42] (and references therein) on the major subject of solving sparse linear systems of equations in finite fields.

Gaussian Elimination Algorithm

The solution of a nonsingular (lower or upper) triangular linear system $A\ \vec{x} = \vec{v}$ only involves about n^2 ops. For example [117], let $n = 3$,

$$\begin{aligned} x_1 + 2x_2 - x_3 &= 3\,, \\ -2x_2 - 2x_3 &= -10\,, \\ -6x_3 &= -18\,. \end{aligned}$$

Compute $x_3 = 3$ from the last equation, substitute into the previous ones, and arrive at a triangular system of $n - 1 = 2$ equations. In $n - 1$ (in our case, in 2) such recursive substitution steps, we compute the solution.

The triangular case is itself important; furthermore, every nonsingular linear system is reduced to 2 triangular ones by means of *forward elimination* of the variables, which essentially amounts to computing the PLU-factorization of the input matrix A, that is, to computing 2 lower triangular matrices L and U^T (where L has unit values on its diagonal) and a permutation matrix P such that $A = PLU$. (A permutation matrix P is filled with zeros and ones and has exactly one nonzero entry in each row and in each column; in particular, this implies that $P^T = P^{-1}$. $P\ \vec{u}$ has the same components as $\vec{u}$ but written in a distinct (fixed) order, for any vector $\vec{u}$.) As soon as the latter factorization is available, we may compute $\vec{x} = A^{-1}\ \vec{v}$ by solving 2 triangular systems, that is, at first, $L\ \vec{y} = P^T\ \vec{v}$, in $\vec{y}$, and then $U\ \vec{x} = \vec{y}$, in $\vec{x}$. Computing the factorization (elimination stage) is more costly than the subsequent *back substitution stage,* the latter involving about $2n^2$ ops. Gaussian elimination algorithm involves about $2n^3/3$ ops and some comparisons, required in order to ensure appropriate *pivoting*, also called *elimination ordering.* Pivoting enables us to avoid divisions by small values. Otherwise, we would have needed a higher precision of computing, thus, facing numerical stability problems. Theoretically, one may employ fast matrix multiplication and compute the matrices P, L, and U in $O(n^{2.376})$ ops [1] (and then compute the vectors $\vec{y}$ and $\vec{x}$ in $O(n^2)$ ops). Pivoting can be dropped for some important classes of linear systems, notably, for *positive definite* and for *diagonally dominant* systems ([62, 115, 117], or [12]).

We refer the reader to [62, p. 82–83] or [117, p. 794] on sensitivity of the solution to the input and round-off errors in numerical computing. The output errors grow with the **condition number** of A, represented by $||A||\ ||A^{-1}||$ for a fixed matrix norm or by the ratio of maximum and minimum singular values of A. Except for ill-conditioned linear systems $A\ \vec{x} = \vec{v}$, having very large condition numbers, a rough initial approximation to the solution can be rapidly refined (cf. [62]) via the *iterative improvement algorithm,* as soon as we know P and rough approximations to the matrices L and U of the PLU factorization of A. Then, b correct bits of each output value can be computed in $O(b + n)n^2$ ops as $b \to \infty$.

Sparse Linear Systems. Direct and Iterative Solution Algorithms

A matrix is sparse if it is filled mostly with zeros, say, if its all nonzero entries lie on 3 or 5 of its diagonals; it can be stored economically if the disposition of its nonzero entries has a certain structure. Such matrices arise in many important applications, in particular, to solving partial and ordinary differential equations (PDEs and ODEs). Then, memory space and computation time can be dramatically decreased (say, from order n^2 to order $n \log n$ words of memory and from n^3 to $n^{3/2}$ or $n \log n$ ops) by using some special data structures and special solution methods. The methods are either direct, that is, are modifications of Gaussian elimination with some special policies of elimination ordering that preserve sparsity during the computation (notably, *Markowitz rule* and *nested dissection* [58, 61, 98, 118]), or various iterative algorithms. The latter ones usually rely either on computing Krylov sequences [64] or on multilevel or

multigrid techniques [53, 104, 126], specialized for solving linear systems that arise from discretization of PDEs. *Banded linear systems* have $n \times n$ coefficient matrices $A = [a_{i,j}]$ where $a_{i,j} = 0$ if $i - j > g$ or $j - i > h$, for $g + h$ being much less than n. For such systems, the nested dissection is known under the name of *block cyclic reduction* and is highly effective, but [128] gives some alternative algorithms too. Some special techniques for parallel computation of Krylov sequences for sparse and other special matrices A can be found in [119]; according to these techniques, Krylov sequence is recovered from the solution of the associated linear system $(I - A)\,\vec{x} = \vec{v}$, which is solved fast in the case of a special matrix A.

Dense and Structured Matrices and Linear Systems

Many dense $n \times n$ matrices are defined by $O(n)$, say, by less than $2n$, parameters and can be multiplied by a vector by using $O(n \log n)$ or $O(n \log^2 n)$ ops. Such matrices arise in numerous applications (to signal and image processing, coding, algebraic computation, PDEs, integral equations, particle simulation, Markov chains, and many others). An important example is given by $n \times n$ *Toeplitz matrices* $T = [t_{i,j}]$, $t_{i,j} = t_{i+1,j+1}$ for $i,\ j = 0, 1, \ldots, n-2$. Such a matrix can be represented by $2n - 1$ entries of its first row and first column or by $2n - 1$ entries of its first and last columns. The product $T\,\vec{v}$ is defined by vector convolution, and its computation uses $O(n \log n)$ ops. Other major examples are given by *Hankel matrices* (obtained by reflecting the row or column sets of Toeplitz matrices), *circulant* (which are a subclass of Toeplitz matrices), *Bézout, Sylvester, Vandermonde,* and *Cauchy* matrices. The known solution algorithms for linear systems with such dense structured coefficient matrices use from order $n \log n$ to order $n \log^2 n$ ops. These properties and algorithms are extended via associating some linear operators of displacement and scaling to some more general classes of matrices and linear systems. (See Chapter 17 in this volume for details and further bibliography.)

Parallel Matrix Computations

Algorithms for matrix multiplication are particularly suitable for parallel implementation; one may exploit natural association of processors to rows and/or columns of matrices or to their blocks, particularly, in the implementation of matrix multiplication on loosely coupled multiprocessors (cf. [62, 130]). In particular, the straightforward algorithm for $n \times n$ matrix multiplication uses $2n^2$ fetches of input data and n^2 writings into the memory, for $2n^3 - n^2$ ops, that is, in block matrix algorithms, the slow data manipulation operations are relatively few (versus the more numerous but faster ops). This motivates special attention to and rapid progress in devising effective practical parallel algorithms for block matrix computations (see "Further Information"). The theoretical complexity of parallel computations is usually measured by the computational and communication time and the number of processors involved; decreasing all these parameters, we face a trade-off; the product of time and processor bounds (called potential work of parallel algorithms) cannot usually be made substantially smaller than the sequential time bound for the solution. This follows because, according to a variant of *Brent's scheduling principle,* a single processor can simulate the work of s processors in time $O(s)$. The usual goal of designing a parallel algorithm is in decreasing its parallel time bound (ideally, to a constant, logarithmic or polylogarithmic level, relative to n) and keeping its work bound at the level of the record sequential time bound for the same computational problem (within constant, logarithmic, or at worst polylog factors). This goal has been easily achieved for matrix and vector multiplications, but turned out to be nontrivial for linear system solving, inversion, and some other related computational problems. The recent solution for general matrices [79, 80] relies on computation of a Krylov sequence and the coefficients of the minimum polynomial of a matrix, by using randomization and auxiliary computations with **structured matrices** (see the details in [12, 122]).

Rational Matrix Computations, Computations in Finite Fields, and Semirings

Rational algebraic computations with matrices are performed for a rational input given with no errors, and the computations are also performed with no errors. The precision of computing can be bounded by reducing the computations modulo one or several fixed primes or prime powers. At the end, the exact output values $z = p/q$ are recovered from $z \; mod \; M$ (if M is sufficiently large relative to p and q) by using the continued fraction approximation algorithm, which is the Euclidean algorithm applied to integers (cf. [115, 116], and [12, Sect. 3 of ch. 3]). If the output z is known to be an integer lying between $-m$ and m and if $M > 2m$, then z is recovered from $z \bmod M$ as follows:

$$z = \begin{cases} z \bmod M & \text{if} \quad z \bmod M < m \\ -M + z \bmod M & \text{otherwise} . \end{cases}$$

The reduction modulo a prime p may turn a nonsingular matrix A and a nonsingular linear system $A\vec{x} = \vec{v}$ into singular ones, but this is proved to occur only with a low probability for a random choice of the prime p in a fixed sufficiently large interval (see [12, Sect. 9 of ch. 4]). To compute the output values z modulo M for a large M, one may first compute them modulo several relatively prime integers $m_1, m_2, \ldots, m_k$ having no common divisors and such that $m_1 m_2 \ldots m_k = M$ and then easily recover $z \bmod M$ by means of the Chinese remainder algorithm. For matrix and polynomial computations, there is an effective alternative technique of *p-adic* (*Newton–Hensel*) *lifting* (cf. [12, Sect. 3 of ch. 3]), which is particularly powerful for computations with dense structured matrices, since it preserves the structure of a matrix. We refer the reader to [6] and [56] on some special techniques that enable one to control the growth of all intermediate values computed in the process of performing rational Gaussian elimination, with no round-off and no reduction modulo an integer. The highly important topic of randomized solution of linear systems of equations in finite fields is covered in [42], which also contains further bibliography. [63] and [118] describe some applications of matrix computations on semirings (with no divisions and subtractions allowed) to graph and combinatorial computations.

Matrix Eigenvalues and Singular Values Problems

The matrix eigenvalue problem is one of the major problems of matrix computation: given an $n \times n$ matrix A, one seeks the maximum k and a $k \times k$ diagonal matrix Λ and an $n \times k$ matrix V of full rank k such that

$$A \; V = V \Lambda \; . \tag{16.1}$$

The diagonal entries of Λ are called the *eigenvalues* of A; the entry (i, i) of Λ is associated with the ith column of V, called an *eigenvector* of A. The eigenvalues of an $n \times n$ matrix A coincide with the zeros of the characteristic polynomial

$$c_A(x) = \det(\, xI - A \,) \; .$$

If this polynomial has n distinct zeros, then $k = n$, and V of (16.1) is a nonsingular $n \times n$ matrix. The Toeplitz matrix $A = I + Z$, $Z = (z_{i,j})$, $z_{i,j} = 0$ unless $j = i + 1$, $z_{i,i+1} = 1$, is an example of a matrix for which $k = 1$, so that the matrix V degenerates to a vector.

In principle, one may compute the coefficients of $c_A(x)$, the characteristic polynomial of A, and then approximate its zeros (see the next section) in order to approximate the eigenvalues of A. Given the eigenvalues, the corresponding eigenvectors can be recovered by means of the inverse power iteration [62, 145]. Practically, the computation of the eigenvalues via the computation of the coefficients of $c_A(x)$ is not recommended, due to arising numerical stability problems [145], and most frequently, the eigenvalues and eigenvectors of a general (unsymmetric) matrix are approximated by means of the *QR algorithm* [62, 145]. Before application of this algorithm, the matrix A is simplified by transforming it into the more special (*lower Hessenberg*) *form*, $H = [h_{i,j}]$, $h_{i,j} = 0$ if $i - j > 1$, by a *similarity transformation,*

$$H = UAU^H \; , \tag{16.2}$$

where $U = [u_{i,j}]$ is a unitary matrix, $U^H U = I$, $U^H = [\overline{u}_{j,i}]$ is the Hermitian transpose of U, $\overline{z}$ denoting the complex conjugate of z; $U^H = U^T$ if U is a real matrix [62]. Similarity transformation into Hessenberg form is a *rational transformation* of a matrix into a special *canonical form, Smith* and *Hermite forms* are two other most important representatives of canonical forms [56, 60, 76].

The eigenvalue problem is *symmetric,* if the matrix A is real symmetric,

$$A^T = \left[a_{j,i}\right] = A = \left[a_{i,j}\right] ,$$

or complex Hermitian,

$$A^H = \left[\overline{a}_{j,i}\right] = A = \left[a_{i,j}\right] .$$

The symmetric eigenvalue problem is simpler: the matrix V of (16.1) is a nonsingular $n \times n$ matrix, and all the eigenvalues are real and little sensitive to small input perturbations of A [62, 129]. Furthermore, the Hessenberg matrix H of (16.2) becomes symmetric tridiagonal (cf. [62] or [12, Sect. 2.3]). For such a matrix H, application of the QR algorithm is dramatically simplified; moreover, two competitive algorithms are also widely used, that is, the *bisection* [129] (a slightly slower but very robust algorithm) and the *divide-and-conquer* method [34, 62]. The latter method has a modification [11] that only uses $O(n \log^2 n(\log n + \log^2 b))$ arithmetic operations in order to compute all the eigenvalues of an $n \times n$ symmetric tridiagonal matrix A within the output error bound $2^{-b}||A||$, where $||A|| \leq n \max |a_{i,j}|$.

A natural generalization of the eigenproblem (16.1) is to the *generalized eigenproblem.* Given a pair A, B of matrices, the generalized eigenvalue Λ and the generalized eigenvector V satisfy

$$A\ V = B\ V\Lambda\ .$$

The solution algorithm should not compute matrix $B^{-1}A$ explicitly, so as to avoid problems of numerical stability.

Another important extension of the symmetric eigenvalue problem is *singular value decomposition* (SVD) of a (generally unsymmetric and, possibly, rectangular) matrix A: $A = U\Sigma\ V^T$, where U and V are unitary matrices, $U^H U = V^H V = I$, and Σ is a diagonal (generally rectangular) matrix, filled with zeros, except for its diagonal, filled with (positive) singular values of A and, possibly, with zeros. The SVD is widely used in the study of numerical stability of matrix computations and in numerical treatment of singular and ill-conditioned (close to singular) matrices. It is a basic tool, for instance, in the study of approximate polynomial GCD [33, 49].

Approximating Polynomial Zeros

Solution of an nth degree polynomial equation,

$$p(x) = \sum_{i=0}^{n} p_i\ x^i = 0\ , \qquad p_n \neq 0$$

(where one may assume that $p_{n-1} = 0$; this can be achieved via shifting the variable x), is a classical problem that has greatly influenced the development of mathematics throughout the centuries [124]. The problem remains highly important for the theory and practice of present day computing, and dozens of new algorithms for its approximate solution appear every year. Among the existent implementations of such algorithms, the practical heuristic champions in efficiency (in terms of computer time and memory space used, according to the results of many experiments) are various modifications of *Newton's iteration,* $z(i+1) = z(i) - a(i)p(z(i))/p'(z(i))$, $a(i)$ being the step-size parameter [101], *Laguerre's method* [54, 66], and the randomized *Jenkins–Traub algorithm* [69] (all three for approximating a single zero z of $p(x)$), which can be extended to approximating other zeros by means of deflation of the input polynomial via its

numerical division by $x - z$. For simultaneous approximation of all the zeros of $p(x)$, one may apply the Durand–Kerner algorithm, which is defined by the following recurrence:

$$z_j(l+1) = z_j(l) - \frac{p\left(z_j(l)\right)}{p_n \prod_{i \neq j}\left(z_j(l) - z_i(l)\right)}, \quad j = 1, \ldots, n, \quad l = 1, 2, \ldots . \tag{16.3}$$

Here, the most customary choice (see [13] for some effective alternatives) for the n initial approximations $z_j(0)$ to the n zeros of

$$p(x) = p_n \prod_{j=1}^{n}\left(x - z_j\right)$$

is given by $z_j(0) = Z\, exp(2\pi\sqrt{-1}/n)$, $j = 1, \ldots, n$, Z exceeding (by some fixed factor $t > 1$) $\max_j |z_j|$; for instance, one may set

$$Z = 2\,t \max_{i<n} |p_i/p_n| . \tag{16.4}$$

For a fixed l and for all j, the computation according to (16.3) is simple, only involving order n^2 ops, (or even $O(n \log^2 n)$ ops with deteriorated numerical stability [13]). Furthermore, according to the results of many experiments, the iteration (16.3) rapidly converges to the solution, though no theory confirms or explains these results. Similar is the situation with various modifications of this algorithm, which are now even more popular than the original algorithms. The reader is referred to [13, 105, 116], and [124] on many of these algorithms, some implementation issues, and further extensive bibliography.

Neither of the algorithms (cited above) supports any reasonable good estimates for the computational complexity of approximating polynomial zeros for the worst case inputs, but such estimates have been achieved based on algorithms of two other groups. One such group is given by the modern modifications and improvements (due to [114, 120, 132]) of *Weyl's quadtree construction* of 1924. In this approach, an initial square S, containing all the zeros of $p(x)$ is recursively partitioned into 4 congruent subsquares; say, $S = \{ x , | Im\, x | < Z, | Re\, x | < Z \}$ for Z of (16.4). In the center of each of them, a proximity test is applied that estimates the distance from this center to the closest zero of $p(x)$. If such a distance exceeds one half of the diagonal length, then the subsquare contains no zeros of $p(x)$ and is discarded. When this process ensures a strong isolation from each other for the components formed by the remaining squares, then certain extensions of Newton's iteration [120, 132] or some iterative techniques based on numerical integration [114] are applied and very rapidly converge to the desired approximations to the zeros of $p(x)$, within the error bound $2^{-b}Z$ for Z of (16.4). As a result, the algorithms of [114, 120] solve the entire problem of approximating (within $2^{-b}Z$) all the zeros of $p(x)$ at the overall cost of performing $O(\,(n^2 \log n)\ \log(bn)\,)$ ops (cf. [13]).

The second group is given by the *divide-and-conquer algorithms.* They first compute a sufficiently wide annulus A, which is free of the zeros of $p(x)$ and contains comparable numbers of such zeros (that is, the same numbers up to a fixed constant factor) in its exterior and its interior. Then the 2 factors of $p(x)$ are numerically computed, that is, $F(x)$, having all its zeros in the interior of the annulus, and $G(x) = p(x)/F(x)$, having no zeros there. The same process is recursively repeated for $F(x)$ and $G(x)$ until factorization of $p(x)$ into the product of linear factors is computed numerically. From this factorization, approximations to all the zeros of $p(x)$ are obtained. The algorithms of [121] based on this approach only require $O(n \log(bn)\ (\log n)^2)$ ops in order to approximate all the n zeros of $p(x)$ within $2^{-b}Z$ for Z of (16.4). (Note that this is a quite sharp bound: at least n ops are necessary in order to output n distinct values.)

The computations for the polynomial zero problem are ill-conditioned, that is, they generally require a high precision for the worst case input polynomials in order to ensure a required output precision, no matter which algorithm is applied for the solution. Consider, for instance, the polynomial $(x - \frac{6}{7})^n$ and perturb its x-free coefficient by 2^{-bn}. Observe the resulting jumps of the zero $x = 6/7$ by 2^{-b}, and observe similar jumps if the coefficients p_i are perturbed by $2^{(i-n)b}$ for $i = 1, 2, \ldots, n-1$. Therefore, to ensure

the output precision of b bits, we need an input precision of at least $(n-i)b$ bits for each coefficient p_i, $i = 0, 1, \ldots, n-1$. Consequently, for the worst case input polynomial $p(x)$, any solution algorithm needs at least about by factor n increase of the precision of the input and of computing, versus the output precision.

Numerically unstable algorithms may require even a higher input and computation precision, but inspection shows that this is not the case for the algorithms of [13, 114, 120, 121, 132].

16.3 Systems of Nonlinear Equations

Given a system $P = \{p_1(x_1, \ldots, x_n),\ p_2(x_1, \ldots, x_n), \ldots, p_r(x_1, \ldots, x_n)\}$ of nonlinear polynomials with rational coefficients (each $p_i(x_1, \ldots, x_n)$ is said to be an element of $\mathbb{Q}[x_1, \ldots, x_n]$, the ring of polynomials in $x_1, \ldots, x_n$ over the field of rational numbers), the n-tuple of complex numbers $(a_1, \ldots, a_n)$ is a solution of the system if $f_i(a_1, \ldots, a_n) = 0$ for each i with $1 \leq i \leq r$. In this section, we explore the problem of exactly solving a system of nonlinear equations over the field $\mathbb{Q}$. We also indicate how an initial phase of exact algebraic computation leads to certain numerical methods that approximate the solutions; the interaction of symbolic and numeric computation is currently an active domain of research [47]. We provide an overview and cite references to different symbolic techniques used for solving systems of algebraic (polynomial) equations. In particular, we describe methods involving *resultant* and *Gröbner basis* computations.

The *Sylvester resultant method* is the technique most frequently utilized for determining a common zero of two polynomial equations in one variable [88]. However, using the Sylvester method successively to solve a system of multivariate polynomials proves to be inefficient. Successive resultant techniques, in general, lack efficiency as a result of their sensitivity to the ordering of the variables [85]. It is more efficient to eliminate all variables together from a set of polynomials, thus, leading to the notion of the *multivariate resultant.* The three most commonly used multivariate resultant formulations are the *Dixon* [40, 86], *Macaulay* [21, 23, 99], and *sparse resultant formulations* [22, 139].

The theory of Gröbner bases provides powerful tools for performing computations in multivariate polynomial rings. Formulating the problem of solving systems of polynomial equations in terms of polynomial ideals, we will see that a Gröbner basis can be computed from the input polynomial set, thus, allowing for a form of back substitution in order to compute the common roots.

Although not discussed, it should be noted that the *characteristic set algorithm* can be utilized for polynomial system solving. Ritt [133] introduced the concept of a characteristic set as a tool for studying solutions of algebraic differential equations. In 1984, Wu [147] in search of an effective method for automatic theorem proving, converted Ritt's method to ordinary polynomial rings. Given the before mentioned system P, the characteristic set algorithm transforms P into a triangular form, such that the set of common zeros of P is equivalent to the set of roots of the triangular system [85].

Throughout this exposition we will also see that these techniques used to solve nonlinear equations can be applied to other problems as well, such as computer-aided design and automatic geometric theorem proving.

The Sylvester Resultant

The question of whether two polynomials $f(x),\ g(x) \in \mathbb{Q}[x]$,

$$\begin{aligned} f(x) &= f_n x^n + f_{n-1} x^{n-1} + \ldots + f_1 x + f_0\,, \\ g(x) &= g_m x^m + g_{m-1} x^{m-1} + \ldots + g_1 x + g_0\,, \end{aligned}$$

have a common root leads to a condition that has to be satisfied by the coefficients of both f and g. Using a derivation of this condition due to Euler, the *Sylvester matrix* of f and g (which is of order $m+n$) can

be formulated. The vanishing of the determinant of the Sylvester matrix, known as the *Sylvester resultant,* is a necessary and sufficient condition for f and g to have common roots [88].

As a running example let us consider the following system in two variables provided by Lazard [94]:

$$\begin{aligned} f &= x^2 + xy + 2x \quad + y - 1 = 0\,, \\ g &= x^2 \quad + 3x - y^2 + 2y - 1 = 0\,. \end{aligned}$$

The Sylvester resultant can be used as a tool for eliminating several variables from a set of equations [85]. Without loss of generality, the roots of the Sylvester resultant of f and g treated as polynomials in y, whose coefficients are polynomials in x, are the x-coordinates of the common zeros of f and g. More specifically, the Sylvester resultant of the Lazard system with respect to y is given by the following determinant:

$$\det\left(\begin{bmatrix} x+1 & x^2+2x-1 & 0 \\ 0 & x+1 & x^2+2x-1 \\ -1 & 2 & x^2+3x-1 \end{bmatrix}\right) = -x^3 - 2x^2 + 3x\,.$$

An alternative matrix formulation named after Bézout yields the same determinant. This formulation is discussed below in the context of multivariate polynomials, in "Resultants of Multivariate Systems."

The roots of the Sylvester resultant of f and g are $\{-3, 0, 1\}$. For each x value, one can substitute the x value back into the original polynomials yielding the solutions $(-3, 1)$, $(0, 1)$, $(1, -1)$.

The method just outlined can be extended recursively, using *polynomial GCD computations,* to a larger set of multivariate polynomials in $\mathbb{Q}[x_1, \ldots, x_n]$. This technique, however, is impractical for eliminating many variables, due to an explosive growth of the degrees of the polynomials generated in each elimination step.

The Sylvester formulations has led to a *subresultant theory,* developed simultaneously by G.E. Collins and W.S. Brown and J. Traub. The subresultant theory produced an efficient algorithm for computing polynomial GCDs and their resultants, while controlling intermediate expression swell [15, 31, 88].

It should be noted that by adopting an implicit representation for symbolic objects, the intermediate expression swell introduced in many symbolic computations can be palliated. Recently, polynomial GCD algorithms have been developed that use implicit representations and thus, avoid the computationally costly content and primitive part computations needed in those GCD algorithms for polynomials in explicit representation [38, 71, 83].

Resultants of Multivariate Systems

The solvability of a set of nonlinear multivariate polynomials can be determined by the vanishing of a generalization of the Sylvester resultant of two polynomials in a single variable. We examine two generalizations, namely, the classical and the sparse resultants. The *classical resultant* of a system of n homogeneous polynomials in n variables vanishes exactly when there exists a common solution in *projective* space [85, 141]. The *sparse resultant* characterizes solvability over a smaller space which coincides with affine space under certain genericity conditions [57, 139]. The main algorithmic question, then, is to construct a matrix whose determinant is the resultant or a nontrivial multiple of it.

Due to the special structure of the Sylvester matrix, Bézout developed a method for computing the resultant as a determinant of order $\mathrm{Max}(m, n)$ during the eighteenth century. Cayley [29] reformulated Bézout's method leading to Dixon's [40] extension to the bivariate case. Dixon's method can be generalized to a set

$$\{p_1\,(x_1, \ldots, x_n)\,,\, p_2\,(x_1, \ldots, x_n)\,, \ldots,\, p_{n+1}\,(x_1, \ldots, x_n)\}$$

of $n + 1$ generic n-degree polynomials in n variables [87]. The vanishing of the determinant of the Dixon matrix is a necessary and sufficient condition for the polynomials to have a nontrivial projective common

zero, and also a necessary condition for the existence of an affine common zero. The Dixon formulation gives the resultant up to a multiple, and hence, in the affine case it may happen that the vanishing of the Dixon determinant does not necessarily indicate that the equations in question have a common root. A nontrivial multiple, known as the *projection operator,* can be extracted via a method based on so-called *rank subdeterminant computation* (RSC) [87]. It should be noted that the RSC method can also be applied to the Macaulay and sparse resultant formulations as is detailed below. A more general and simpler method for extracting a projection operator from the Dixon matrix is discussed in [28, thm. 3.3.4]. This article, along with [43], explain the correlation between residue theory and the Dixon matrix, which yields an alternative method for studying and approximating all common solutions.

In 1916, Macaulay [99] constructed a matrix whose determinant is a multiple of the classical resultant for n homogeneous polynomials in n variables. The Macaulay matrix simultaneously generalizes the Sylvester matrix and the coefficient matrix of a system of linear equations [85]. As the Dixon formulation, the Macaulay determinant is a multiple of the resultant. Macaulay, however, proved that a certain minor of his matrix divides the matrix determinant so as to yield the exact resultant in the case of generic homogeneous polynomials. Canny [21] has invented a general method that perturbs any polynomial system and extracts a nontrivial projection operator.

Using recent results pertaining to sparse polynomial systems [57, 139], a matrix formula for computing the sparse resultant of $n+1$ polynomials in n variables was given by Canny and Emiris [22] and consequently improved in [25, 48]. The determinant of the sparse resultant matrix, like the Macaulay and Dixon matrices, only yields a projection operation, not the exact resultant.

Here, sparsity means that only certain monomials in each of the $n + 1$ polynomials have nonzero coefficients. Sparsity is measured in geometric terms, namely, by the *Newton polytope* of the polynomial, which is the convex hull of the exponent vectors corresponding to nonzero coefficients. The *mixed volume* of the Newton polytopes of n polynomials in n variables is defined as a certain integer-valued function that bounds the number of affine common roots of these polynomials, according to a theorem of [10]. This remarkable theorem is the cornerstone of sparse elimination. The mixed volume bound is significantly smaller than the classical Bézout bound for polynomials with small Newton polytopes. Since these bounds also determine the degree of the sparse and classical resultants, respectively, the latter has larger degree for sparse polynomials.

Polynomial System Solving by Using Resultants

Suppose we are asked to find the common zeros of a set of n polynomials in n variables $\{p_1(x_1, \ldots, x_n), p_2(x_1, \ldots, x_n), \ldots, p_n(x_1, \ldots, x_n)\}$. By augmenting the polynomial set by a generic linear polynomial [21, 24, 85], one can construct the *u-resultant* of a given system of polynomials. The u-resultant is named after the vector of indeterminates u, traditionally used to represent the generic coefficients of the additional linear polynomial. The u-resultant factors into linear factors over the complex numbers, providing the common zeros of the given polynomials equations. The u-resultant method takes advantage of the properties of the multivariate resultant, and hence, can be constructed using either Dixon's, Macaulay's, or sparse formulations. An alternative approach, where we *hide* a variable in the coefficient field, instead of adding a polynomial, is discussed in [46, 102].

Consider the previous example augmented by a generic linear form:

$$\begin{aligned} f_1 &= x^2 + xy + 2x \quad + y - 1 = 0\,, \\ f_2 &= x^2 \quad + 3x - y^2 + 2y - 1 = 0\,, \\ f_l &= \qquad ux \qquad + vy + w = 0\,. \end{aligned}$$

As described in Canny, Kaltofen and Lakshman [23], the following matrix M corresponds to the

Macaulay u-resultant of the above system of polynomials, with z being the homogenizing variable:

$$M = \begin{bmatrix} 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 & 0 & u & 0 & 0 & 0 \\ 2 & 0 & 1 & 3 & 0 & 1 & 0 & u & 0 & 0 \\ 0 & 1 & 0 & -1 & 0 & 0 & v & 0 & 0 & 0 \\ 1 & 2 & 1 & 2 & 3 & 0 & w & v & u & 0 \\ -1 & 0 & 2 & -1 & 0 & 3 & 0 & w & 0 & u \\ 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 2 & -1 & 0 & 0 & v & 0 \\ 0 & -1 & 1 & 0 & -1 & 2 & 0 & 0 & w & v \\ 0 & 0 & -1 & 0 & 0 & -1 & 0 & 0 & 0 & w \end{bmatrix}.$$

It should be noted that

$$\det(M) = (u - v + w)(-3u + v + w)(v + w)(u - v)$$

corresponds to the affine solutions $(1, -1)$, $(-3, 1)$, $(0, 1)$, and one solution at infinity.

Resultants may also be applied to reduce polynomial system solving to a regular or generalized eigenproblem (cf. "Matrix Eigenvalues and Singular Values Problems"), thus, transforming the nonlinear question to a problem in linear algebra. This is a classical technique that enables us to approximate all solutions (cf. [3, 24, 28, 46]). For demonstration, consider the previous system and its resultant matrix M. The matrix rows are indexed by the following row vector of monomials in the eliminated variables:

$$\vec{v} = \left[x^3, x^2y, x^2, xy^2, xy, x, y^3, y^2, y, 1\right].$$

Vector $\vec{v}M$ expresses the polynomials indexing the columns of M, which are multiples of the three input polynomials by various monomials. Let us specialize variables u and v to random values. Then the matrix M contains a single variable w and is denoted $M(w)$. Solving the linear system $\vec{v}M(w) = \vec{0}$ in vector $\vec{v}$ and in scalar w is a generalized eigenproblem, since $M(w)$ can be represented as $M_0 + wM_1$, where M_0 and M_1 have numeric entries. If, moreover, M_1 is invertible, we arrive at the following eigenproblem:

$$\vec{v}\,(M_0 + wM_1) = \vec{0} \Longleftrightarrow \vec{v}\left(-M_1^{-1}M_0 - wI\right) = \vec{0} \Longleftrightarrow \vec{v}\left(-M_1^{-1}M_0\right) = w\vec{v}\,.$$

For every solution (a, b) of the original system, there is a vector $\vec{v}$ among the computed eigenvectors, which we evaluate at $x = a,\ y = b$ and from which the solution can be recovered by means of division (cf. [46]). As for the eigenvalues, they correspond to the values of w at the solutions.

Recently, the structure of various resultant matrices has been studied, continuing work in [23]. By exploiting the quasi-Toeplitz or quasi-Hankel structure of such matrices, it is possible to accelerate their construction, the computation of the resultant itself, and the approximation of the system's solutions [44, 108].

An empirical comparison of the detailed resultant formulations can be found in [86, 102]. The multivariate resultant formulations have been used for diverse applications such as *algebraic and geometric reasoning* [28, 87, 102], *computer-aided design* [89, 138], *robot kinematics* [45, 102], computing *molecular conformations* [45, 103] and for *implicitization and finding base points* [30, 102].

Gröbner Bases

Solving systems of nonlinear equations can be formulated in terms of polynomial ideals [7, 56, 146]. Let us first establish some terminology.

The ideal generated by a system of polynomial equations $p_1, \ldots, p_r$ over $\mathbb{Q}[x_1, \ldots, x_n]$ is the set of all linear combinations

$$(p_1, \ldots, p_r) = \{h_1 p_1 + \ldots + h_r p_r \mid h_1, \ldots, h_r \in \mathbb{Q}[x_1, \ldots, x_n]\} .$$

The algebraic variety of $p_1, \ldots, p_r \in \mathbb{Q}[x_1, \ldots, x_n]$ is the set of their common zeros,

$$V(p_1, \ldots, p_r) = \left\{(a_1, \ldots, a_n) \in \mathbb{C}^n \mid f_1(a_1, \ldots, a_n) = \ldots = f_r(a_1, \ldots, a_n) = 0\right\} .$$

A version of the *Hilbert Nullstellensatz* states that

$$V(p_1, \ldots, p_r) = \text{the empty set } \emptyset \Longleftrightarrow 1 \in (p_1, \ldots, p_r) \text{ over } \mathbb{Q}[x_1, \ldots, x_n] ,$$

which relates the solvability of polynomial systems to the ideal membership problem.

A term $t = x_1^{e_1} x_2^{e_2} \ldots x_n^{e_n}$ of a polynomial is a product of powers with $\deg(t) = e_1 + e_2 + \cdots + e_n$. In order to add needed structure to the polynomial ring we will require that the terms in a polynomial be ordered in an admissible fashion [56, 85]. Two of the most common admissible orderings are the **lexicographic order** ($\prec_l$), where terms are ordered as in a dictionary, and the **degree order** ($\prec_d$), where terms are first compared by their degrees with equal degree terms compared lexicographically. A variation to the lexicographic order is the *reverse lexicographic order,* where the lexicographic order is reversed [35, p. 96].

It is this above mentioned structure that permits a type of simplification known as polynomial reduction. Much like a polynomial remainder process, the process of polynomial reduction involves subtracting a multiple of one polynomial from another to obtain a smaller degree result [7, 56, 85, 146].

A polynomial g is said to be reducible with respect to a set $P = \{p_1, \ldots, p_r\}$ of polynomials if it can be reduced by one or more polynomials in P. When g is no longer reducible by the polynomials in P, we say that g is *reduced* or is *a normal form* with respect to P.

For an arbitrary set of basis polynomials, it is possible that different reduction sequences applied to a given polynomial g could reduce to different normal forms. A basis $G \subseteq \mathbb{Q}[x_1, \ldots, x_n]$ is a *Gröbner basis* if and only if every polynomial in $\mathbb{Q}[x_1, \ldots, x_n]$ has a unique normal form with respect to G. Bruno Buchberger [16, 17, 18, 19] showed that every basis for an ideal $(p_1, \ldots, p_r)$ in $\mathbb{Q}[x_1, \ldots, x_n]$ can be converted into a Gröbner basis $\{p_1^*, \ldots, p_s^*\} = GB(p_1, \ldots, p_r)$, concomitantly designing an algorithm that transforms an arbitrary ideal basis into a Gröbner basis. Another characteristic of Gröbner bases is that by using the above mentioned reduction process we have

$$g \in (p_1 \ldots, p_r) \Longleftrightarrow \left(g \bmod p_1^*, \ldots, p_s^*\right) = 0 .$$

Further, by using the Nullstellensatz it can be shown that $p_1 \ldots, p_r$ viewed as a system of algebraic equations is solvable if and only if $1 \notin GB(p_1, \ldots, p_r)$.

Depending on which admissible term ordering is used in the Gröbner bases construction, an ideal can have different Gröbner bases. However, an ideal cannot have different (reduced) Gröbner bases for the same term ordering.

Any system of polynomial equations can be solved using a lexicographic Gröbner basis for the ideal generated by the given polynomials. It has been observed, however, that Gröbner bases, more specifically lexicographic Gröbner bases, are hard to compute [7, 56, 93, 146]. In the case of zero-dimensional ideals, those whose varieties have only isolated points, Faugère et al. [50] outlined a change of basis algorithm which can be utilized for solving zero-dimensional systems of equations. In the zero-dimensional case, one computes a Gröbner basis for the ideal generated by a system of polynomials under a degree ordering. The so-called *change of basis algorithm* can then be applied to the degree ordered Gröbner basis to obtain a Gröbner basis under a lexicographic ordering.

Turning to Lazard's example in form of a polynomial basis,

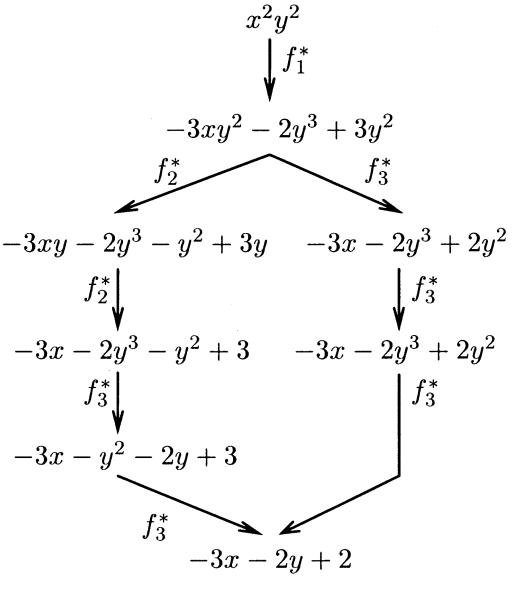
$$\begin{array}{lllllll} f_1 & = & x^2 & +xy & +2x & +y & -1 , \\ f_2 & = & x^2 & & +3x & -y^2 & +2y & -1 , \end{array}$$

one obtains (under lexicographical ordering with $x \prec_l y$) a Gröbner basis in which the variables are triangularized such that the finitely many solutions can be computed via back substitution:

$$\begin{array}{lcccccc} f_1^* & = & x^2 & & +3x & +2y & -2\,, \\ f_2^* & = & & xy & -x & -y & +1\,, \\ f_3^* & = & & & y^2 & & -1\,. \end{array}$$

It should be noted that the final univariate polynomial is of minimal degree and the polynomials used in the back substitution will have degree no larger than the number of roots.

As an example of the process of polynomial reduction with respect to a Gröbner basis, the following demonstrates two possible reduction sequences to the same normal form. The polynomial x^2y^2 is reduced with respect to the previously computed Gröbner basis $\{f_1^*, f_2^*, f_3^*\} = GB(f_1, f_2)$ along the following two distinct reduction paths, both yielding $-3x - 2y + 2$ as the normal form.



There is a strong connection between lexicographic Gröbner bases and the previously mentioned resultant techniques. For some types of input polynomials, the computation of a reduced system via resultants might be much faster than the computation of a lexicographic Gröbner basis. A good comparison between the Gröbner computations and the different resultant formulations can be found in [86, 102].

In a survey article, Buchberger [18] detailed how Gröbner bases can be used as a tool for many polynomial ideal theoretic operations. Other applications of Gröbner basis computations include automatic geometric theorem proving [84, 147], multivariate polynomial factorization and GCD computations [59], and polynomial interpolation [91, 92].

16.4 Polynomial Factorization

The problem of factoring polynomials is a fundamental task in symbolic algebra. An example in one's early mathematical education is the factorization $x^2 - y^2 = (x + y) \cdot (x - y)$, which in algebraic terms is a factorization of a polynomial in two variables with integer coefficients. Technology has advanced to a state where most polynomial factorization problems are doable on a computer, in particular, with any of the

popular mathematical software, such as the Mathematica or Maple systems. For instance, the factorization of the determinant of a 6×6 symmetric Toeplitz matrix over the integers is computed in Maple as

```
>readlib(showtime):
>showtime():
O1 := T := linalg[toeplitz]([a,b,c,d,e,f]);
```

$$T := \begin{bmatrix} a & b & c & d & e & f \\ b & a & b & c & d & e \\ c & b & a & b & c & d \\ d & c & b & a & b & c \\ e & d & c & b & a & b \\ f & e & d & c & b & a \end{bmatrix}$$

```
time 0.03 words 7701
O2 := factor(linalg[det](T));
```

$$\begin{aligned} -(&2dca - 2bce + 2c^2a - a^3 - da^2 + 2d^2c + d^2a + b^3 + 2abc - 2c^2b \\ &+ d^3 + 2ab^2 - 2dcb - 2cb^2 - 2ec^2 + 2eb^2 + 2fcb + 2bae \\ &+ b^2f + c^2f + be^2 - ba^2 - fdb - fda - fa^2 - fba + e^2a - 2db^2 \\ &+ dc^2 - 2deb - 2dec - dba)(2dca - 2bce - 2c^2a + a^3 \\ &- da^2 - 2d^2c - d^2a + b^3 + 2abc - 2c^2b + d^3 - 2ab^2 + 2dcb \\ &+ 2cb^2 + 2ec^2 - 2eb^2 - 2fcb + 2bae + b^2f + c^2f + be^2 - ba^2 \\ &- fdb + fda - fa^2 + fba - e^2a - 2db^2 + dc^2 + 2deb - 2dec \\ &+ dba) \end{aligned}$$

```
time 27.30 words 857700
```

Clearly, the Toeplitz determinant factorization requires more than tricks from high school algebra. Indeed, the development of modern algorithms for the polynomial factorization problem is one of the great successes of the discipline of symbolic mathematical computation. Kaltofen has surveyed the algorithms until 1992 in [70, 73, 74], mostly from a computer science perspective. In this article we shall focus on the applications of the known fast methods to problems in science and engineering. For a more extensive set of references, please refer to Kaltofen's survey articles.

Polynomials in a Single Variable Over a Finite Field

At the first glance, the problem of factoring an integer polynomial modulo a prime number appears to be very similar to the problem of factoring an integer represented in a prime radix. That is simply not so. The factorization of the polynomial $x^{511} - 1$ can be done modulo 2 on a computer in a matter of milliseconds, while the factorization of the integer $2^{511} - 1$ into its integer factors is a computational challenge. For those interested, the largest prime factors of $2^{511} - 1$ have 57 and 67 decimals digits, respectively, which makes a tough but not undoable 123 digit product for the number field sieve factorizer [97]. Irreducible factors of polynomials modulo 2 are needed to construct finite fields. For example, the factor $x^9 + x^4 + 1$ of $x^{511} - 1$ leads to a model of the finite field with 2^9 elements, GF(2^9), by simply computing with the polynomial remainders modulo $x^9 + x^4 + 1$ as the elements. Such irreducible polynomials are used for setting up error-correcting codes, for instance, the BCH codes [100]. Berlekamp's [8, 9] pioneering work on factoring polynomials over a finite field by linear algebra is done with this motivation. The linear

algebra tools that Berlekamp used seem to have been introduced to the subject as early as in 1937 by Petr (cf. [134]).

Today, factoring algorithms for univariate polynomials over finite fields form the innermost subalgorithm to lifting-based algorithms for factoring polynomials in one [148] and many [109] variables over the integers. When Maple computed the factorization of the above Toeplitz determinant, it began with factoring a univariate polynomial modulo a prime integer. The case when the prime integer is very large has lead to a significant development in computer science itself. As it turns out, by selecting random residues the expected performance of the algorithms can be speeded up exponentially [9, 131]. Randomization is now an important tool for designing efficient algorithms and has proliferated to many fields of computer science. Paradoxically, the random elements are produced by a congruential random number generator, and the actual computer implementations are quite deterministic, which leads some computer scientists to believe that random bits can be eliminated in general at no exponential slow-down. Nonetheless, for the polynomial factoring problem modulo a large prime, no fast methods are known to-date that would work without this "probabilistic" approach.

One can measure the computing time of selected algorithms in terms of n, the degree of the input polynomial, and p, the cardinality of the field. When counting arithmetic operations modulo p (including reciprocals), the best known algorithms are quite recent. Berlekamp's 1970 method performs $O(n^{\omega} + n^{1+o(1)} \log p)$ residue operations. Here and below, ω denotes the exponent implied by the used linear system solver, i.e., $\omega = 3$ when classical methods are used, and $\omega = 2.376$ when asymptotically fast (though impractical) matrix multiplication is assumed. The correction term $o(1)$ accounts for the $\log n$ factors derived from the FFT-based fast polynomial multiplication and remaindering algorithms. An approach in the spirit of Berlekamp's but possibly more practical for $p = 2$ has recently been discovered by Niederreiter [110]. A very different technique by Cantor and Zassenhaus [27] first separates factors of different degrees and then splits the resulting polynomials of equal degree factors. It has $O(n^{2+o(1)} \log p)$ complexity and is the basis for the following two methods. Algorithms by von zur Gathen and Shoup [142] have running time $O(n^{2+o(1)} + n^{1+o(1)} \log p)$ and those by Kaltofen and Shoup [81] have running time $O(n^{1.815} \log p)$, the latter with fast matrix multiplication. The techniques of von zur Gathen and Shoup have recently been applied to speed the case $p = 2^k$ for large k in terms of fixed precision (bit) operations [82].

For n and p simultaneously large, a variant of the method by Kaltofen and Shoup [81] that uses classical linear algebra and runs in $O(n^{2.5} + n^{1+o(1)} \log p)$ residue operations is the current champion among the practical algorithms. With it Shoup, using his own fast polynomial arithmetic package [136], has factored a random-like polynomial of degree 2048 modulo a 2048-bit prime number in about 12 days on a Sparc-10 computer using 68 Mbyte of main memory. For even larger n, but smaller p, parallelization helps, and Kaltofen and Lobo [77] could factor a polynomial of degree $n = 15001$ modulo $p = 127$ in about 6 days on 8 computers that are rated at 86.1 MIPS. To date, the largest polynomial factored modulo 2 is a random polynomial of degree 262143; this was accomplished by von zur Gathen and Gerhard [143]. They employ Cantor's fast polynomial multiplication algorithm based on additive transforms [26].

Polynomials in a Single Variable over Fields of Characteristic Zero

As mentioned before, generally usable methods for factoring univariate polynomials over the rational numbers begin with the Hensel lifting techniques introduced by Zassenhaus [148]. The input polynomial is first factored modulo a suitable prime integer p, and then the factorization is lifted to one modulo p^k for an exponent k of sufficient size to accommodate all possible integer coefficients that any factors of the polynomial might have. The lifting approach is fast in practice, but there are hard-to-factor polynomials on which it runs an exponential time in the degree of the input. This slowdown is due to so-called parasitic modular factors. The polynomial $x^4 + 1$, for example, factors modulo all prime integers but is irreducible over the integers: it is the cyclotomic equation for 8th roots of unity. The products of all subsets of modular

factors are candidates for integer factors, and irreducible integer polynomials with exponentially many such subsets exist [78].

The elimination of the exponential bottleneck by giving a polynomial-time solution to the integer polynomial factoring problem, due to Lenstra et al. [95], is considered a major result in computer science algorithm design. The key ingredient to their solution is the construction of integer relations to real or complex numbers. For the simple demonstration of this idea, consider the polynomial

$$x^4 + 2x^3 - 6x^2 - 4x + 8 \,.$$

A root of this polynomial is $\alpha \approx 1.236067977$, and $\alpha^2 \approx 1.527864045$. We note that $2\alpha + \alpha^2 \approx 4.000000000$, hence, $x^2 + 2x - 4$ is a factor. The main difficulty is to efficiently compute the integer linear relation with relatively small coefficients for the high precision big-float approximations of the powers of a root. Lenstra et al. solve this diophantine optimization problem by means of their now famous lattice reduction procedure, which is somewhat reminiscent of the ellipsoid method for linear programming.

The determination of linear integer relations among a set of real or complex numbers is a useful task in science in general. Very recently, some stunning identities could be produced by this method, including the following formula for π [5]:

$$\pi = \sum_{n=0}^{\infty} \frac{1}{16^n} \left(\frac{4}{8n+1} - \frac{2}{8n+4} - \frac{1}{8n+5} - \frac{1}{8n+6} \right) .$$

Even more surprising, the lattice reduction algorithm can prove that no linear integer relation with integers smaller than a chosen parameter exists among the real or complex numbers. There is an efficient alternative to the lattice reduction algorithm, originally due to Ferguson and Forcade [52] and recently improved by Ferguson and Bailey [51].

The complexity of factoring an integer polynomial of degree n with coefficients of no more than l bits is thus, a polynomial in n and l. From a theoretical point of view, an algorithm with a low estimate is by Miller [106] and has a running time of $O(n^{5+o(1)}l^{1+o(1)} + n^{4+o(1)}l^{2+o(1)})$ bit-operations. It is expected that the relation-finding methods will become usable in practice on hard-to-factor polynomials in the near future. If the hard-to-factor input polynomial is irreducible, an alternate approach can be used to prove its irreducibility. One finds an integer evaluation point at which the integral value of the polynomial has a large prime factor, and the irreducibility follows by mathematical theorems. Monagan [107] has proven large hard-to-factor polynomials irreducible in this way, which would be hopeless by the lifting algorithm.

Coefficient fields other than finite fields and the rational numbers are of interest. Computing the factorizations of univariate polynomials over the complex numbers is the root finding problem described earlier in "Approximating Polynomial Zeros." When the coefficient field has an extra variable, such as the field of fractions of polynomials ("rational functions") the problem reduces, by an old theorem of C.F. Gauss, to factoring multivariate polynomials, which we discuss below. When the coefficient field is the field of Laurent series in t with a finite segment of negative powers,

$$\frac{c_{-k}}{t^k} + \frac{c_{-k+1}}{t^{k-1}} + \cdots + \frac{c_{-1}}{t} + c_0 + c_1 t + c_2 t^2 + \cdots, \quad \text{where } k \geq 0 \,,$$

fast methods appeal to the theory of Puiseux series, which constitute the domain of algebraic functions [144].

Polynomials in Two Variables

Factoring bivariate polynomials by reduction to univariate factorization via homomorphic projection and subsequent lifting can be done similarly to the univariate algorithm [109]. The second variable y takes the role of the prime integer p and $f(x, y) \bmod y = f(x, 0)$. Lifting is possible only if $f(x, 0)$ had

no multiple root. Provided that $f(x, y)$ has no multiple factor, which can be insured by a simple GCD computation, the squarefreeness of $f(x, 0)$ can be obtained by variable translation $\hat{y} = y + a$, where a is an easy-to-find constant in the coefficient field. For certain domains, such as the rational numbers, any irreducible multivariate polynomial $h(x, y)$ can be mapped to an irreducible univariate polynomial $h(x, b)$ for some constant b. This is the important *Hilbert irreducibility theorem,* whose consequence is that the combinatorial explosion observed in the univariate lifting algorithm is, in practice, unlikely. However, the magnitude and probabilistic distribution of good points b is not completely analyzed.

For so-called non-Hilbertian coefficient fields good reduction is not possible. An important such field are the complex numbers. Clearly, all $f(x, b)$ completely split into linear factors, while $f(x, y)$ may be irreducible over the complex numbers. An example for an irreducible polynomial is $f(x, y) = x^2 - y^3$. Polynomials that remain irreducible over the complex numbers are called absolutely irreducible. An additional problem is the determination of the algebraic extension of the ground field in which the absolutely irreducible factors can be expressed. In the example

$$x^6 - 2x^3y^2 + y^4 - 2x^3 = \left(x^3 - \sqrt{2}x - y^2\right) \cdot \left(x^3 + \sqrt{2}x - y^2\right) ,$$

the needed extension field is $\mathbb{Q}(\sqrt{2})$. The relation-finding approach proves successful for this problem. The root is computed as a Taylor series in y, and the integrality of the linear relation for the powers of the series means that the multipliers are polynomials in y of bounded degree. Several algorithms of polynomial-time complexity and pointers to the literature are found in [75].

Bivariate polynomials constitute implicit representations of algebraic curves. It is an important operation in geometric modeling to convert from implicit to parametric representation. For example, the circle

$$x^2 + y^2 - 1 = 0$$

has the rational parameterization

$$x = \frac{2t}{1 + t^2}, \quad y = \frac{1 - t^2}{1 + t^2}, \quad \text{where } -\infty \le t \le \infty .$$

Algorithms are known that can find such rational parameterizations provided that they exist [135]. It is crucial that the inputs to these algorithms are absolutely irreducible polynomials.

Polynomials in Many Variables

Polynomials in many variables, such as the symmetric Toeplitz determinant exhibited above, are rarely given explicitly, due to the fact that the number of possible terms grows exponentially in the number of variables: there can be as many as $\binom{n+v}{n} \ge 2^{\min\{n,v\}}$ terms in a polynomial of degree n with v variables. Even the factors may be dense in canonical representation, but could be sparse in another basis: for instance, the polynomial

$$(x_1 - 1)\,(x_2 - 2) \cdots (x_v - v) + 1$$

has only 2 terms in the "shifted basis," while it has 2^v terms in the power basis, i.e., in expanded format.

Randomized algorithms are available that can efficiently compute a factor of an implicitly given polynomial, say, a matrix determinant, and even can find a shifted basis with respect to which a factor would be sparse, provided, of course, that such a shift exists. The approach is by manipulating polynomials in so-called black box representations [83]: a black box is an object that takes as input a value for each variable, and then produces the value of the polynomial it represents at the specified point. In the Toeplitz example the representation of the determinant could be the Gaussian elimination program which computes it. We note that the size of the polynomial in this case would be nearly constant, only the variable names and the dimension need to be stored. The factorization algorithm then outputs procedures which

will evaluate all irreducible factors at an arbitrary point (supplied as the input). These procedures make calls to the black box given as input to the factorization algorithm in order to evaluate them at certain points, which are derived from the point at which the procedures computing the values of the factors are probed. It is, of course, assumed that subsequent calls evaluate one and the same factor and not associates that are scalar multiples of one another. The algorithm by Kaltofen and Trager [83] finds procedures that with a controllably high probability evaluate the factors correctly. Randomization is needed to avoid parasitic factorizations of homomorphic images that provide some static data for the factor boxes and cannot be avoided without mathematical conjecture. The procedures that evaluate the individual factors are deterministic.

Factors constructed as black box programs are much more space efficient than those represented in other formats, for example, the straight-line program format [72]. More importantly, once the black box representation for the factors is found, sparse representations can be rapidly computed by any of the new sparse interpolation algorithms. See [65] for the latest method allowing shifted bases and pointers to the literature of other methods, including ones for the standard power bases.

The black box representation of polynomials is normally not supported by commercial computer algebra systems such as Axiom, Maple, or Mathematica. Díaz and Kaltofen have developed the FOXBOX system in C++ that makes black box methodology available to users of such systems [37, 39]. They have computed a factor over the rationals of the determinant of a generic 13 by 13 symmetric Toepitz matrix, which has 4982 nonzero terms, in 15 hours and 25 minutes on a single processor of a Sun Ultra 2 computer.

16.5 Research Issues and Summary

Section 16.3 of this chapter has briefly reviewed polynomial system solving based on resultant matrices as well as Gröbner bases. Both approaches are currently active, especially in applications dealing with small and medium-size systems. Handling the nongeneric cases, including multiple roots and nonisolated solutions, is probably the most crucial issue today in relation to resultants. An interesting practical question is to delimit those cases where each of the two methods and their different variants is preferable.

16.6 Defining Terms

Characteristic polynomial: A polynomial associated with a square matrix, the determinant of the matrix when a single variable is subtracted to its diagonal entries. The roots of the characteristic polynomial are the eigenvalues of the matrix.

Condition number: A scalar derived from a matrix that measures its relative nearness to a singular matrix. Very close to singular means a large condition number, in which case numeric inversion becomes an ill-conditioned problem.

Degree order: An order on the terms in a multivariate polynomial; for two variables x and y with $x \prec y$ the ascending chain of terms is $1 \prec x \prec y \prec x^2 \prec xy \prec y^2 \cdots$.

Determinant: A polynomial in the entries of a square matrix with the property that its value is nonzero if and only if the matrix is invertible.

Lexicographic order: An order on the terms in a multivariate polynomial; for two variables x and y with $x \prec y$ the ascending chain of terms is $1 \prec x \prec x^2 \prec \cdots \prec y \prec xy \prec x^2y \cdots \prec y^2 \prec xy^2 \cdots$.

Matrix eigenvector: A column vector v such that, given square matrix A, $Av = \lambda v$, where λ is the matrix eigenvalue corresponding to v. A generalized eigenvector v is such that, given square matrices A, B, it satisfies $Av = \lambda Bv$. Both definitions extend to a row vector which premultiplies the corresponding matrix.

Ops: Arithmetic operations, i.e., additions, subtractions, multiplications, or divisions; as in **flops,** i.e., floating point operations.

Singularity: A square matrix is singular if there is a nonzero second matrix such that the product of the two is the zero matrix. Singular matrices do not have inverses.

Sparse matrix: A matrix where many of the entries are zero.

Structured matrix: A matrix where each entry can be derived by a formula depending on few parameters. For instance, the Hilbert matrix has $1/(i + j - 1)$ as the entry in row i and column j.

References

Note: many of Erich Kaltofen's publications are accessible through links in the online BibTeX bibliography database at www.math.ncsu.edu/~kaltofen/bibliography/.

[1] Aho, A., Hopcroft, J., and Ullman, J., *The Design and Analysis of Algorithms.* Addison-Wesley, Reading, MA, 1974.

[2] Anderson, E. et al., *LAPACK Users' Guide.* SIAM Publications, Philadelphia, PA, 1992.

[3] Auzinger, W. and Stetter, H.J., An elimination algorithm for the computation of all zeros of a system of multivariate polynomial equations. In *Proc. Intern. Conf. on Numerical Math., Intern. Series of Numerical Math., 86,* 12–30. Birkhäuser, Basel, 1988.

[4] Bach, E. and Shallit, J., *Algorithmic Number Theory, Volume 1: Efficient Algorithms.* The MIT Press, Cambridge, MA, 1996.

[5] Bailey, D., Borwein, P., and Plouffe, S., On the rapid computation of various polylogarithmic constants. *Math. Comp.,* 66, 903–913, 1997. `http://mosaic.cecm.sfu.ca/preprints/1995pp.html`, 1995.

[6] Bareiss, E.H., Sylvester's identity and multistep integers preserving Gaussian elimination. *Math. Comp.,* 22, 565–578, 1968.

[7] Becker, T. and Weispfenning, V., *Gröbner bases: A Computational Approach to Commutative Algebra.* Springer-Verlag, New York, 1993.

[8] Berlekamp, E.R., Factoring polynomials over finite fields. *Bell Systems Tech. J.,* 46, 1853–1859, 1967. Republished in revised form in: E. R. Berlekamp, *Algebraic Coding Theory,* Chapter 6, McGraw-Hill, New York, 1968.

[9] Berlekamp, E.R., Factoring polynomials over large finite fields. *Math. Comp.,* 24, 713–735, 1970.

[10] Bernstein, D.N., The number of roots of a system of equations. *Funct. Anal. and Appl.,* 9(2), 183–185, 1975.

[11] Bini, D. and Pan, V.Y., Parallel complexity of tridiagonal symmetric eigenvalue problem. In *Proc. 2nd Ann. ACM-SIAM Symp. on Discrete Algorithms,* 384–393. ACM Press, New York, and SIAM Publications, Philadelphia, PA, 1991.

[12] Bini, D. and Pan, V.Y., *Polynomial and Matrix Computations, Volume 1, Fundamental Algorithms.* Birkhäuser, Boston, 1994.

[13] Bini, D. and Pan, V.Y., *Polynomial and Matrix Computations, Volume 2.* Birkhäuser, Boston, 1998.

[14] Borodin, A. and Munro, I., *Computational Complexity of Algebraic and Numeric Problems.* American Elsevier, New York, 1975.

[15] Brown, W.S. and Traub, J.F., On Euclid's algorithm and the theory of subresultants. *J. ACM,* 18, 505–514, 1971.

[16] Buchberger, B., A theoretical basis for the reduction of polynomials to canonical form. *ACM SIGSAM Bulletin,* 10(3), 19–29, 1976.

[17] Buchberger, B., A note on the complexity of constructing Gröbner-bases. In *Proc. EUROCAL '83,* van Hulzen, J.A., Ed., Springer Lec. Notes Comp. Sci., 137–145, 1983.

[18] Buchberger, B., Gröbner bases: An algorithmic method in polynomial ideal theory. In *Recent Trends in Multidimensional Systems Theory,* Bose, N.K., Ed., 184–232. D. Reidel, Dordrecht (Holland), 1985.

[19] Buchberger, B., *Ein Algorithmus zum Auffinden der Basiselemente des Restklassenringes nach einem nulldimensionalen Polynomideal.* Dissertation, University Innsbruck, Austria, Fall 1965.

[20] Bürgisser, P., Clausen, M., and Shokrollahi, M.A., *Algebraic Complexity Theory.* Springer, Berlin, 1997.

[21] Canny, J., Generalized characteristic polynomials. *J. Symbolic Comput.,* 9(3), 241–250, 1990.

[22] Canny, J. and Emiris, I., An efficient algorithm for the sparse mixed resultant. In *Proc. AAECC-10,* Cohen, G., Mora, T., and Moreno, O., Eds., volume 673 of *Springer Lect. Notes Comput. Sci.,* 89–104, 1993.

[23] Canny, J., Kaltofen, E., and Lakshman, Y., Solving systems of non-linear polynomial equations faster. In *Proc. ACM-SIGSAM 1989 Internat. Symp. Symbolic Algebraic Comput. ISSAC '89,* 121–128. ACM, 1989.

[24] Canny, J. and Manocha, D., Efficient techniques for multipolynomial resultant algorithms. In *Proc. Internat. Symp. Symbolic Algebraic Comput. ISSAC '91,* Watt, S.M., Ed., 85–95, ACM Press, New York, 1991.

[25] Canny, J. and Pedersen, P., An algorithm for the Newton resultant. Technical Report 1394, Computer Science Department, Cornell University, 1993.

[26] Cantor, D.G., On arithmetical algorithms over finite fields. *J. Combinatorial Theory, Series A,* 50, 285–300, 1989.

[27] Cantor, D.G. and Zassenhaus, H., A new algorithm for factoring polynomials over finite fields. *Math. Comp.,* 36, 587–592, 1981.

[28] Cardinal, J.-P. and Mourrain, B., Algebraic approach of residues and applications. In *The Mathematics of Numerical Analysis,* Renegar, J., Shub, M., and Smale, S., Eds., volume 32 of *Lectures in Applied Math.,* 189–210. AMS, Providence, RI, 1996.

[29] Cayley, A., On the theory of eliminaton. *Cambridge and Dublin Mathematical Journal,* 3, 210–270, 1865.

[30] Chionh, E., *Base Points, Resultants and Implicit Representation of Rational Surfaces.* Ph.D. Thesis, Department Computer Science, University Waterloo, 1990.

[31] Collins, G.E., Subresultants and reduced polynomial remainder sequences. *J. ACM,* 14, 128–142, 1967.

[32] Coppersmith, D. and Winograd, S., Matrix multiplication via arithmetic progressions. *J. Symbolic Comput.,* 9(3), 251–280, 1990.

[33] Corless, R.M., Gianni, P.M., Trager, B.M., and Watt, S.M., The singular value decomposition for polynomial systems. In Levelt [96], 96–103.

[34] Cuppen, J.J.M., A divide and conquer method for the symmetric tridiagonal eigenproblem. *Numer. Math.,* 36, 177–195, 1981.

[35] Davenport, J.H., Tournier, E., and Siret, Y., *Computer Algebra Systems and Algorithms for Algebraic Computation.* Academic Press, London, 1988.

[36] Demmel, J.J.W., *Applied Numerical Linear Algebra.* SIAM Publications, Philadelphia, PA, 1997.

[37] Díaz, A., FoxBox *a System for Manipulating Symbolic Objects in Black Box Representation.* Ph.D. Thesis, Rensselaer Polytechnic Instit., Troy, New York, 1997.

[38] Díaz, A. and Kaltofen, E., On computing greatest common divisors with polynomials given by black boxes for their evaluation. In *Proc. 1995 Internat. Symp. Symbolic Algebraic Comput. ISSAC '95,* Levelt, A.H.M., Ed., 232–239, ACM Press, New York, l995.

[39] Díaz, A. and Kaltofen, E., FOXBOX a system for manipulating symbolic objects in black box representation. In *Proc. 1998 Internet Symp. Symbolic Algebraic Comput. ISSAC '98,* Gloor, O., Ed., ACM Press, New York, 1998. To appear.
[40] Dixon. A.L., The elimination of three quantics in two independent variables. In *Proc. London Mathematical Society,* 6, 468–478, 1908.
[41] Dongarra, J. et al., *LINPACK Users' Guide.* SIAM Publications, Philadelphia, PA, 1978.
[42] Eberly, W. and Kaltofen, E., On randomized Lanczos algorithms. In Küchlin [90], 176–183.
[43] Elkadi, M. and Mourrain, B., Approche effective des résidus algébriques. Technical Report 2884, INRIA, Sophia-Antipolis, France, 1996.
[44] Emiris, I.Z. and Pan, V.Y., The structure of sparse resultant matrices. In *Proc. ACM Intern. Symp. on Symbolic and Algebraic Computation,* 189–196, Maui, HI, 1997.
[45] Emiris, I.Z., *Sparse Elimination and Applications in Kinematics.* Ph.D. Thesis, Computer Science Division, University of California at Berkeley, 1994.
[46] Emiris, I.Z., On the complexity of sparse elimination. *J. Complexity,* 12, 134–166, 1996.
[47] Emiris, I.Z., Symbolic-numeric algebra for polynomials. In *Encyclopedia of Computer Science and Technology.* Marcel Dekker, New York, 1998. To appear.
[48] Emiris, I.Z. and Canny, J.F., Efficient incremental algorithms for the sparse resultant and the mixed volume. *J. Symbolic Computation,* 20(2), 117–149, 1995.
[49] Emiris, I.Z., Galligo, A., and Lombardi, H., Certified approximate univariate GCDs. *J. Pure Applied Algebra, Special Issue on Algorithms for Algebra,* 117 & 118, 229–251, 1997.
[50] Faugère, J., Gianni, P., Lazard, D., and Mora, T., Efficient computation of zero-dimensional Gröbner bases by change of ordering. *J. Symbolic Comput.,* 16(4), 329–344, 1993.
[51] Ferguson, H.R.P. and Bailey, D.H., Analysis of PSLQ, an integer relation finding algorithm. Technical Report NAS-96-005, NASA Ames Research Center, 1996.
[52] Ferguson, H.R.P. and Forcade, R.W., Multidimensional Euclidean algorithms. *J. Reine Angew. Math.,* 334, 171–181, 1982.
[53] Fiorentino, G. and Serra, S., Multigrid methods for symmetric positive definite block Toeplitz matrices with nonnegative generating functions. *SIAM J. Sci. Comput.,* 17, 1068–1081, 1996.
[54] Foster, L.V., Generalizations of Laguerre's method: higher order methods. *SIAM J. Numer. Anal.,* 18, 1004–1018, 1981.
[55] Garbow, B.S. et al., *Matrix Eigensystem Routines: EISPACK Guide Extension.* Springer, New York, 1972.
[56] Geddes, K.O., Czapor, S.R., and Labahn, G., *Algorithms for Computer Algebra.* Kluwer Academic, 1992.
[57] Gelfand, I.M., Kapranov, M.M., and Zelevinsky, A.V., *Discriminants, Resultants and Multidimensional Determinants.* Birkhäuser Verlag, Boston, 1994.
[58] George, A. and Liu, J.W.-H., *Computer Solution of Large Sparse Positive Definite Linear Systems.* Prentice Hall, Englewood Cliffs, NJ, 1981.
[59] Gianni, P. and Trager, B., GCD's and factoring polynomials using Gröbner bases. *Proc. EUROCAL '85, Vol. 2, Springer Lec. Notes Comp. Sci.,* 204, 409–410, 1985.
[60] Giesbrecht, M., Nearly optimal algorithms for canonical matrix forms. *SIAM J. Comput.,* 24(5), 948–969, 1995.
[61] Gilbert, J.R. and Tarjan, R.E., The analysis of a nested dissection algorithm. *Numer. Math.,* 50, 377–404, 1987.
[62] Golub, G.H. and Van Loan, C.F., *Matrix Computations,* 3rd ed., Johns Hopkins University Press, Baltimore, MD, 1996.
[63] Gondran, M. and Minoux, M., *Graphs and Algorithms.* Wiley–Interscience, New York, 1984.
[64] Greenbaum, A., *Iterative Methods for Solving Linear Systems.* SIAM Publications, Philadelphia, PA, 1997.

[65] Grigoriev. D.Yu. and Lakshman, Y.N., Algorithms for computing sparse shifts for multivariate polynomials. In *Proc. 1995 Internat. Symp. Symbolic Algebraic Comput. ISSAC '95,* Levelt, A.H.M., Ed., 96–103, ACM Press, New York, 1995.

[66] Hansen, E., Patrick, M., and Rusnak, J., Some modifications of Laguerre's method. *BIT*, 17, 409–417, 1977.

[67] Heath, M.T., Ng, E., and Peyton, B.W., Parallel algorithms for sparse linear systems. *SIAM Review*, 33, 420–460, 1991.

[68] Higham, N.J., *Accuracy and Stability of Numerical Algorithms.* SIAM, Philadelphia, PA, 1996.

[69] Jenkins, M.A. and Traub, J.F., A three-stage variable-shift iteration for polynomial zeros and its relation to generalized Rayleigh iteration. *Numer. Math.,* 14, 252–263, 1970.

[70] Kaltofen, E., Polynomial factorization. In *Computer Algebra,* Buchberger, B., Collins, G., and Loos, R., Eds., 2nd ed., 95–113. Springer-Verlag, Vienna, 1982.

[71] Kaltofen, E., Greatest common divisors of polynomials given by straight-line programs. *J. ACM,* 35(1), 231–264, 1988.

[72] Kaltofen, E., Factorization of polynomials given by straight-line programs. In *Randomness and Computation,* Micali, S., Ed., volume 5 of *Advances in Computing Research,* 375–412. JAI Press, Greenwich, CT, 1989.

[73] Kaltofen, E., Polynomial factorization 1982-1986. In *Computers in Mathematics,* Chudnovsky, D.V. and Jenks, R.D., Eds., volume 125 of *Lecture Notes in Pure and Applied Mathematics,* 285–309. Marcel Dekker, New York, 1990.

[74] Kaltofen, E., Polynomial factorization 1987-1991. In *Proc. LATIN '92,* Simon, I., Ed., volume 583 of *Springer Lect. Notes Comput. Sci.,* 294–313, 1992.

[75] Kaltofen, E., Effective Noether irreducibility forms and applications. *J. Comput. System Sci.,* 50(2), 274–295, 1995.

[76] Kaltofen, E., Krishnamoorthy, M.S., and Saunders, B.D., Parallel algorithms for matrix normal forms. *Linear Algebra and Applications,* 136, 189–208, 1990.

[77] Kaltofen, E. and Lobo, A., Factoring high-degree polynomials by the black box Berlekamp algorithm. In *Proc. Internat. Symp. Symbolic Algebraic Comput. ISSAC '94,* von zur Gathen, J. and Giesbrecht, M., Eds., 90–98, ACM Press, New York, 1994.

[78] Kaltofen, E., Musser, D.R., and Saunders, B.D., A generalized class of polynomials that are hard to factor. *SIAM J. Comp.,* 12(3), 473–485, 1983.

[79] Kaltofen, E. and Pan, V., Processor efficient parallel solution of linear systems over an abstract field. In *Proc. 3rd Ann. ACM Symp. Parallel Algor. Architecture,* 180–191, ACM Press, New York, 1991.

[80] Kaltofen, E. and Pan, V., Processor-efficient parallel solution of linear systems II: the positive characteristic and singular cases. In *Proc. 33rd Annual Symp. Foundations of Comp. Sci.,* 714–723, Los Alamitos, CA, 1992. IEEE Computer Society Press.

[81] Kaltofen, E. and Shoup, V., Subquadratic-time factoring of polynomials over finite fields. In *Proc. 27th Annual ACM Symp. Theory Comp.,* 398–406, ACM Press, New York, 1995. *Math. Comput.,* in press.

[82] Kaltofen, E. and Shoup, V., Fast polynomial factorization over high algebraic extensions of finite fields. In Küchlin [90], 184–188.

[83] Kaltofen, E. and Trager, B., Computing with polynomials given by black boxes for their evaluations: Greatest common divisors, factorization, separation of numerators and denominators. *J. Symbolic Computation,* 9(3), 301–320, 1990.

[84] Kapur, D., Geometry theorem proving using Hilbert's Nullstellensatz. *J. Symbolic Comp.,* 2, 399–408, 1986.

[85] Kapur, D. and Lakshman, Y.N., Elimination methods an introduction. In *Symbolic and Numerical Computation for Artificial Intelligence.* Donald, B., Kapur, D., and Mundy, J., Eds., Academic Press, 1992.

[86] Kapur, D. and Saxena, T., Comparison of various multivariate resultant formulations. In *Proc. Internat. Symp. Symbolic Algebraic Comput. ISSAC '95,* Levelt, A.H.M., Ed., 187–195, ACM Press, New York, 1995.

[87] Kapur, D., Saxena, T., and Yang, L., Algebraic and geometric reasoning using Dixon resultants. In *Proc. Internat. Symp. Symbolic Algebraic Comput. ISSAC '94,* von zur Gathen, J. and Giesbrecht, M., Eds., 99–107, ACM Press, New York, 1994.

[88] Knuth, D.E., *The Art of Computer Programming, Vol. 2, Seminumerical Algorithms,* 2nd ed., Addison-Wesley, Reading, MA, 1981. 3rd ed., 1997.

[89] Krishnan, S. and Manocha, D., Numeric-symbolic algorithms for evaluating one-dimensional algebraic sets. In *Proc. ACM Intern. Symp. on Symbolic and Algebraic Computation,* 59–67, 1995.

[90] Küchlin, W., Ed., *ISSAC 97 Proc. 1997 Internat. Symp. Symbolic Algebraic Comput.*, ACM Press, New York, 1997.

[91] Lakshman, Y.N. and Saunders, B.D., On computing sparse shifts for univariate polynomials. In *Proc. Internat. Symp. Symbolic Algebraic Comput. ISSAC '94,* von zur Gathen, J. and Giesbrecht, M., Eds., 108–113, ACM Press, New York, 1994.

[92] Lakshman, Y.N. and Saunders, B.D., Sparse polynomial interpolation in non-standard bases. *SIAM J. Comput.,* 24(2), 387–397, 1995.

[93] Lakshman, Y.N., *On the complexity of computing Gröbner bases for zero-dimensional polynomial ideals.* Ph.D. Thesis, Computer Science Department, Rensselaer Polytechnic Institute, Troy, New York, 1990.

[94] Lazard, D., Resolution des systemes d'equation algebriques. *Theoretical Comput. Sci.,* 15, 77–110, 1981. In French.

[95] Lenstra, A.K., Lenstra, H.W., and Lovász, L., Factoring polynomials with rational coefficients. *Math. Ann.,* 261, 515–534, 1982.

[96] Levelt, A.H.M., Ed., *Proc. 1995 Internat. Symp. Symbolic Algebraic Comput. ISSAC'95,* ACM Press, New York, 1995.

[97] Leyland, P., Cunningham project data. Internet document, Oxford University, `ftp://sable.ox.ac.uk/pub/math/cunningham/`, Nov. 1995.

[98] Lipton, R.J., Rose, D., and Tarjan, R.E., Generalized nested dissection. *SIAM J. on Numer. Analysis,* 16(2), 346–358, 1979.

[99] Macaulay, F.S., Algebraic theory of modular systems. Cambridge Tracts 19, Cambridge, 1916.

[100] MacWilliams, F.J. and Sloan, N.J.A., *The Theory of Error-Correcting Codes,* North-Holland, New York, 1977.

[101] Madsen, K., A root-finding algorithm based on Newton's method. *BIT*, 13, 71–75, 1973.

[102] Manocha, D., *Algebraic and Numeric Techniques for Modeling and Robotics*. Ph.D. Thesis, Comp. Science Div., Dept. of Electrical Engineering and Computer Science, University of California, Berkeley, 1992.

[103] Manocha, D., Zhu, Y., and Wright, W., Conformational analysis of molecular chains using nano-kinematics. *Computer Applications of Biological Sciences,* 11(1), 71–86, 1995.

[104] McCormick, S., Ed., *Multigrid Methods.* SIAM Publications, Philadelphia, 1987.

[105] McNamee, J.M., A bibliography on roots of polynomials. *J. Comput. Applied Math.,* 47(3), 391–394, 1993.

[106] Miller, V., Factoring polynomials via relation-finding. In *Proc. ISTCS '92,* Dolev, D., Galil, Z., and Rodeh, M., Eds., volume 601 of *Springer Lect. Notes Comput. Sci.,* 115–121, 1992.

[107] Monagan, M.B., A heuristic irreducibility test for univariate polynomials. *J. Symbolic Comput.,* 13(1), 47–57, 1992.

[108] Mourrain, B. and Pan, V.Y., Solving special polynomial systems by using structured matrices and algebraic residues. In *Proc. Workshop on Foundations of Computational Mathematics,* Cucker, F. and Shub, M., Eds., 287–304, Springer, Berlin, 1997.

[109] Musser, D.R., Multivariate polynomial factorization. *J. ACM,* 22, 291–308, 1975.

[110] Niederreiter, H., New deterministic factorization algorithms for polynomials over finite fields. In *Finite Fields: Theory, Applications and Algorithms,* Mullen, L. and Shiue, P.J.-S., Eds., volume 168 of *Contemporary Mathematics,* 251–268. American Math. Society, Providence, RI, 1994.
[111] Ortega, J.M. and Voight, R.G., Solution of partial differential equations on vector and parallel computers. *SIAM Review,* 27(2), 149–240, 1985.
[112] Pan, V.Y., How can we speed up matrix multiplication? *SIAM Rev.,* 26(3), 393–415, 1984.
[113] Pan, V.Y., *How to Multiply Matrices Faster*, volume 179 of *Lecture Notes in Computer Science.* Springer Verlag, Berlin, 1984.
[114] Pan, V.Y., Sequential and parallel complexity of approximate evaluation of polynomial zeros. *Computers in Mathematics (with Applications),* 14(8), 591–622, 1987.
[115] Pan, V.Y., Complexity of algorithms for linear systems of equations. In *Computer Algorithms for Solving Linear Algebraic Equations (State of the Art),* Spedicato, E., Ed., volume 77 of *NATO ASI Series,* Series F: Computer and Systems Sciences, 27–56, Springer, Berlin, 1991.
[116] Pan, V.Y., Complexity of computations with matrices and polynomials. *SIAM Review*, 34(2), 225–262, 1992.
[117] Pan, V.Y., Linear systems of algebraic equations. In *Encyclopedia of Physical Sciences and Technology,* Yelles, M., Ed., volume 8, 2nd ed., 779–804, 1992. Volume 7, 1st ed., 304–329, 1987.
[118] Pan, V.Y., Parallel solution of sparse linear and path systems. In *Synthesis of Parallel Algorithms,* Reif, J.H., Ed., chapter 14, 621–678. Morgan Kaufmann, San Mateo, CA, 1993.
[119] Pan, V.Y., Parallel computation of a Krylov matrix for a sparse and structured input. *Mathematical and Computer Modelling,* 21(11), 97–99, 1995.
[120] Pan, V.Y., On approximating complex polynomial zeros: Modified quadtree (Weyl's) construction and improved Newton's iteration. Technical Report 2894, INRIA, Sophia-Antipolis, France, 1996. Revised 1997.
[121] Pan, V.Y., Optimal and nearly optimal algorithms for approximating polynomial zeros. *Computers in Mathematics (with Applications),* 31(12), 97–138, 1996. Proceedings version: 27th Ann. ACM STOC, 741–750, ACM Press, New York, 1995.
[122] Pan, V.Y., Parallel computation of polynomial GCD and some related parallel computations over abstract fields. *Theor. Comp. Science,* 162(2), 173–223, 1996.
[123] Pan, V.Y., Faster solution of the key equation for decoding the BCH error-correcting codes. In *Proc. ACM Symp. Theory of Comp.,* 168–175. ACM Press, New York, 1997.
[124] Pan, V.Y., Solving a polynomial equation: Some history and recent progress. *SIAM Review,* 39(2), 187–220, 1997.
[125] Pan, V.Y. and Preparata, F.P., Work-preserving speed-up of parallel matrix computations. *SIAM J. Comput.,* 24(4), 811–821, 1995.
[126] Pan, V.Y. and Reif, J.H., Compact multigrid. *SIAM J. on Scientific and Statistical Computing,* 13(1), 119–127, 1992.
[127] Pan, V.Y. and Reif, J.H., Fast and efficient parallel solution of sparse linear systems. *SIAM J. Comp.,* 22(6), 1227–1250, 1993.
[128] Pan, V.Y., Sobze, I., and Atinkpahoun, A., On parallel computations with band matrices. *Information and Computation*, 120(2), 227–250, 1995.
[129] Parlett, B., *Symmetric Eigenvalue Problem.* Prentice Hall, Englewood Cliffs, NJ, 1980.
[130] Quinn, M.J., *Parallel Computing: Theory and Practice.* McGraw-Hill, New York, 1994.
[131] Rabin, M.O., Probabilistic algorithms in finite fields. *SIAM J. Comp.,* 9, 273–280, 1980.
[132] Renegar, J., On the worst case arithmetic complexity of approximating zeros of polynomials. *J. Complexity,* 3(2), 90–113, 1987.
[133] Ritt, J.F., *Differential Algebra.* AMS, New York, 1950.
[134] Schwarz, Št., On the reducibility of polynomials over a finite field. *Quart. J. Math. Oxford Ser. (2),* 7, 110–124, 1956.

[135] Sendra, J.R. and Winkler, F., Symbolic parameterization of curves. *J. Symbolic Comput.,* 12(6), 607–631, 1991.
[136] Shoup, V., A new polynomial factorization algorithm and its implementation. *J. Symbolic Comput.,* 20(4), 363–397, 1995.
[137] Smith, B.T. et al., *Matrix Eigensystem Routines: EISPACK Guide,* 2nd ed. Springer, New York, 1970.
[138] Stederberg, T. and Goldman, R., Algebraic geometry for computer-aided design. *IEEE Computer Graphics and Applications,* 6(6), 52–59, 1986.
[139] Sturmfels, B., Sparse elimination theory. In *Proc. Computat. Algebraic Geom. and Commut. Algebra,* Eisenbud, D. and Robbiano, L., Eds., Cortona, Italy, 1991.
[140] Trefethen, L.N. and Bau III, D., *Numerical Linear Algebra.* SIAM Publications, Philadelphia, 1997.
[141] van der Waerden, B.L., *Modern Algebra,* 3rd ed., F. Ungar, New York, 1950.
[142] von zur Gathen, J. and Shoup, V., Computing Frobenius maps and factoring polynomials. *Comput. Complexity,* 2, 187–224, 1992.
[143] von zur Gathen, J. and Gerhard, J., Arithmetic and factorization over $\mathbb{F}_2$. In *ISSAC 96 Proc. 1996 Internat. Symp. Symbolic Algebraic Comput.,* Lakshman, Y.N., Ed., 1–9, ACM Press, New York, 1996.
[144] Walsh, P.G., The computation of Puiseux expansions and a quantitative version of Runge's theorem on diophantine equations. Ph.D. Thesis, University Waterloo, Waterloo, Canada, 1993.
[145] Wilkinson, J.H., *The Algebraic Eigenvalue Problem.* Clarendon Press, Oxford, England, 1965.
[146] Winkler, F., *Polynomial Algorithms in Computer Algebra.* Springer, Wien, 1996.
[147] Wu, W., Basis principles of mechanical theorem proving in elementary geometries. *J. Syst. Sci. and Math Sci.,* 4(3), 207–235, 1984.
[148] Zassenhaus, H., On Hensel factorization I. *J. Number Theory,* 1, 291–311, 1969.
[149] Zippel, R., *Effective Polynomial Computations,* 384. Kluwer Academic, Boston, MA, 1993.

Further Information

The books [1, 4, 12, 13, 14, 20, 35, 56, 88, 146, 149] provide a much broader introduction to the general subject and further bibliography. There are well-known libraries and packages of subroutines for the most popular numerical matrix computations, in particular, [41] for solving linear systems of equations, [137] and [55] for approximating matrix eigenvalues, and [2] for both of the two latter computational problems. There is a comprehensive treatment of numerical matrix computations [62], with extensive bibliography, and there are several more specialized books on them [58, 64, 68, 129, 140, 145] as well as many survey articles [67, 111, 117] and thousands of research articles. Two journals are currently dedicated to the subject, the *Journal of Symbolic Computation* by Academic Press, London and *Applicable Algebra in Engineering, Communication and Computing* by Springer-Verlag. The annual *International Symposium on Symbolic and Algebraic Computation* is what we consider the flagship conference of the research community.

Special (more efficient) parallel algorithms have been devised for special classes of matrices, such as sparse [118, 127], banded [128], and dense structured [12, 122]. We also refer to [125] on simple but surprisingly effective extension of Brent's principle for improving the processor and work efficiency of parallel matrix algorithms (with applications to path computations in graphs) and to [62, 67, 111] on practical parallel algorithms for matrix computations.

17

Applications of FFT[1]

Ioannis Z. Emiris
INRIA Sophia-Antipolis

Victor Y. Pan
City University of New York

17.1 Introduction

The subject of this chapter lies in the area of theoretical computer science, though it borrows certain results from computational mathematics, and is fundamental to the theory and practice of signal and image processing and scientific and engineering computing.

A central theme is to bridge the gap between polynomial arithmetic on the one hand, and integer arithmetic and matrix computations on the other. This is the premise of applying the **fast Fourier transform (FFT)** on a wide range of problems, yielding the fastest known algorithms for performing the basic arithmetic operations on integers, polynomials, and dense structured matrices. For instance, this chapter shows different ways of multiplying two univariate polynomials fast. This means that, instead of the classical method that requires a quadratic number of multiplications and additions between constants, we consider algorithms with significantly lower complexity in terms of the polynomial degrees. In particular, we detail an FFT-based approach, which requires a quasi-linear number of operations. Another example

[1]This material is based on work supported in part by the European Union under ESPRIT FRISCO project LTR 21.024 (first author), by the National Science Foundation, under Grant nos. CCR-9020690 and CCR-9625344 (second author), and by PSC CUNY Award nos. 666327 and 667340 (second author).

is a popular and very efficient method for performing arithmetic on modern computers over rationals of arbitrary size.

We state the complexity bounds under the random access machine (RAM) model of computation [1]. Typically, a unit cost is assigned to addition, subtraction, multiplication, and division between real numbers, as well as to reading or writing into a memory location; this is the *arithmetic* model. In estimating the complexity of integer operations, we shall assign different costs to these basic operations, depending on the bit size of the involved parameters; this is the *Boolean* or *bit* model. The distinction between the two models shall be explicit or obvious from the context.

Section 17.2 examines the fundamental transforms between vectors, in particular, the discrete Fourier transform, its inverse, vector convolution, its extensions to wrapped convolutions, and the sine and cosine transforms. The basic algorithm for computing these transforms is the fast Fourier transform (FFT), which is discussed in some detail and then applied to these problems.

Section 17.3 applies these results to some fundamental operations on univariate polynomials and shows the correlations between the main vector transforms and polynomial arithmetic. These results are carried over to the integers, then extended to the polynomials in several variables.

Section 17.4 examines structured matrices defined by significantly fewer elements than the full matrix size, typically, a linear function of the matrix dimension, instead of quadratic. Furthermore, we perform all the fundamental computations with such structured matrices in time quasi linear in the matrix dimension, which is a dramatic improvement over the quadratic or higher complexity estimates for the complexity of the same computations with arbitrary matrices. Such an improvement relies on exploiting correlations among structured matrices, fast polynomial arithmetic, and FFT. Lastly, we examine some transformations among different classes of structured matrices and between computations with such matrices and some other major algebraic computational problems. Although this is a mere glimpse at the numerous applications of structured matrices, we hope to illustrate the richness of the subject.

We practically omit the *lower bound* topics, referring the reader to [14, 19, 62] on some nontrivial results. The reader always may apply the obvious information lower bounds: since each arithmetic and each Boolean operand has two operations and one output, there must always be at least $\max\{O, I/2\}$ operations, where O and I are the sizes of the output and input, respectively. We also omit certain generalizations of Fourier transforms that are of some interest in theoretical computer science; see, for instance, [19]. There are several important applications of FFT in engineering, such as signal and image processing and solving PDEs (see the end of "The Discrete Fourier Transform and Its Inverse" and the introduction of Section 17.4), which could not be covered in a chapter of the present size.

We try to cite books, surveys, or comprehensive articles, rather than the publications that first contained a certain result. There is a tacit understanding that the interested reader will look into the bibliography of the cited references, in which the historical development is outlined.

Hereafter, log stands for $\log_2$ unless specified otherwise.

17.2 Some Fundamental Transforms

The transforms discussed here can be thought of as mappings that transform a given vector to another vector. They are fundamental because a variety of interesting and general problems can be solved by means of these transforms. More importantly, polynomial and integer arithmetic can be reduced to application of such transforms, and this reduction yields algorithms supporting the record estimates for the asymptotic computational complexity (see Section 17.3).

"The Discrete Fourier Transform and Its Inverse" defines the **discrete Fourier transform (DFT)** and some closely related transforms, and outlines the main algorithm for solving DFT, namely, the fast Fourier transform (FFT). The record complexity bounds for polynomial arithmetic, including multiplication, division, transformation under a shift of the variable, evaluation, **interpolation,** and approximating polynomial zeros, are based on FFT. The FFT and fast polynomial algorithms are the basis for many

other fast polynomial computations, performed both numerically and symbolically. "Vector Convolution" studies vector convolution and shows its equivalence to DFT and also to generalized DFT. "Sine and Cosine Transforms" recalls the sine, cosine, and some other transforms.

Abundant further material and bibliography on transforms and convolution can be found in [10, 12, 13, 16, 27, 34, 35, 81, 89].

The Discrete Fourier Transform and Its Inverse

The **discrete Fourier transform (DFT)** of the coefficient vector $\mathbf{p} = [p_0, \ldots, p_n]^T$ of a polynomial

$$p(x) = p_0 + p_1 x + \cdots + p_n x^n \tag{17.1}$$

is the vector $[p(1), \ldots, p(\omega^{K-1})]^T$, where ω is a primitive Kth root of 1, so that $\{1, \omega, \omega^2, \ldots, \omega^{K-1}\}$ is the set of all the Kth roots of 1 and $K = 2^k = n+1$, for a natural k. For simplicity, the reader may assume the study of DFT in the complex field but it can be extended to other fields, rings, and even groups, where the Kth roots of 1 are defined [21, 27, 29, 38]. The problem of computing the DFT is solved by applying the **fast Fourier transform (FFT)** algorithm, which is an important example of *recursive* algorithms based on the **divide-and-conquer** method. The FFT algorithm can be traced back at least to [83] and, to some extent, even to a work of K.F. Gauss of 1805, though its introduction in modern times has been credited to [31] (see [1, 12, 14, 62, 92] for details). To derive FFT, write

$$p(x) = q\left(x^2\right) + xs\left(x^2\right) = q(y) + xs(y)\,,$$

where $y = x^2$, $q(y) = p_0 + p_2 y + \cdots + p_{n-1} y^{(n-1)/2}$, $s(y) = p_1 + p_3 y + \cdots + p_n y^{(n-1)/2}$, and the polynomials $q(y)$ and $s(y)$ have degree at most $(n-1)/2$. To evaluate $p(x)$ at $x = \omega^h$ for $h = 0, 1, \ldots, n$, we first compute $q(y)$ and $s(y)$ at $y = (\omega^2)^h$ and then $q(\omega^{2h}) + xs(\omega^{2h})$ at $x = \omega^h$. There exist only $K/2$ distinct values among all the integer powers of ω^2, since ω^2 is a $(K/2)$-nd root of 1. Half of the multiplications of ω^h by $s(\omega^{2h})$ can be saved because $\omega^i = -\omega^{i+K/2}$ for all i. By applying this method recursively, we arrive at the overall complexity bound of $1.5\ K \log K$ arithmetic operations, hereafter referred to as **ops.** Note that the old classical algorithm uses $2n^2$ ops to compute DFT, not including the cost of computing the roots of 1.

For demonstration, we describe FFT over the field of complex numbers for polynomial

$$p(x) = 3x^3 + 2x^2 - x + 5\,.$$

Here, $K = 4$, and the 4th roots of unity are $1, \omega = \sqrt{-1}, \omega^2 = -1$, and $\omega^3 = -\omega$. We write

$$p(x) = \left(5 + 2x^2\right) + x\left(-1 + 3x^2\right) = q(y) + xs(y), \qquad y = x^2\,.$$

At the cost of performing 2 multiplications (by 1 and ω) and 4 additions/subtractions, this reduces DFT for $p(x)$ to the evaluation of $q(y)$ and $s(y)$ at the points 1 and $\omega^2 = -1$. We compute $q(1), q(\omega^2), s(1)$, and $s(\omega^2)$, by using 2 multiplications (of the values 2 and 3 by 1) and 4 additions/subtractions: $5+2,\ 5-2,\ -1+3,\ -1-3$. Overall, 4 multiplications and 8 additions were required, which is indeed bounded by $1.5\ K\ \log K = 12$.

In actual computations, we may use complex approximations to $\omega^i = \exp(2\pi i\sqrt{-1}/K),\ i = 0, \ldots, K-1$ [62]. In computations where all the p_i are integers, we may perform the computations over the ring Z_m of integers modulo an appropriate natural m or use other special techniques (cf. [91] and [12, Sect. 1.7]). For appropriate choices of m, the desired Kth roots of 1 are readily available in Z_m (see [1, p. 265–269], [14, p. 86–87], or [12, Sect. 3.3]). Performing FFT over finite fields or rings is also required in several applications, for instance, integer multiplication (see [12, ch. 1]).

The **inverse discrete Fourier transform (IDFT)** of a vector $\mathbf{r}$ of polynomial values $r_h = p(\omega^h),\ h = 0, 1, \cdots, K-1$, on a set of all the Kth roots of 1 is the coefficient vector $\mathbf{p} = [p_0, \cdots, p_n]^T$ of $p(x)$. Computing DFT and IDFT is a special case of the more general problems of multipoint polynomial **evaluation** and **interpolation** ("Evaluation, Interpolation, and Chinese Remainder Computations"), whose equivalence to **structured matrix** computations is analyzed in "Vandermonde and Cauchy (generalized Hilbert) Matrices."

The IDFT can be computed in at most $K + 1.5K \log K$ ops by means of applying the *inverse FFT* algorithm, which is again a divide-and-conquer procedure, reminiscent of FFT (see [12, p. 12]). Alternatively, we may reduce DFT and IDFT to each other and to matrix computations, based on the following vector equation: $\Omega\mathbf{p} = \mathbf{r}/\sqrt{K}$. Here, $\Omega = [\omega^{ij}/\sqrt{K}]$ is the *Fourier matrix*, $\mathbf{p} = [p_j]$ is the input vector of DFT, and $\mathbf{r} = [r_i]$, $i, j = 0, 1, \ldots, n$, where $\mathbf{r}/\sqrt{K}$ is the output vector $[p(\omega^h)]$ of DFT. It follows that $\Omega^{-1} = [\omega^{-ij}/\sqrt{K}]$ and $\mathbf{p} = \Omega^{-1}\mathbf{r}/\sqrt{K} = [\omega^{-ij}]\mathbf{r}/K$. Since ω^{-1} is a Kth root of 1, the latter matrix-by-vector product can be computed by means of an FFT; it will remain to divide the resulting vector by K to complete the computation of the IDFT.

The problem of computing the *generalized DFT* at any number K of points is defined as follows: Given a value w, having the reciprocal w^{-1} but not necessarily being any root of 1, and a vector $\mathbf{p} = (p_0, p_1, \ldots, p_{K-1})$, compute the vector $\mathbf{r}$ of the generalized DFT, $\mathbf{r} = (r_h)$, $r_h = \Sigma_{j=0}^{K-1} p_j w^{hj}$ for $h = 0, 1, \ldots, K-1$.

In "Vector Convolution," we will obtain the complexity bound of $O(K \log K)$ ops for computing generalized DFT for any K, by reducing the problem to computing vector convolution, defined in "Vector Convolution" and shown to be equivalent to the generalized DFT.

By using a variable shift ("Multiplication, Division, and Variable Shift") and scaling of the variable, we may extend the solution to the generalized DFT so as to evaluate $p(x)$ on the set $\{ah^{2i} + fh^i + g, i = 0, 1, \ldots, n-1\}$ for any fixed 4-tuple of constants (a, f, g, h), by using $O(n \log n)$ ops [2].

Then again, the bound $O(n \log n)$ is a dramatic improvement of the classical algorithms for IDFT and generalized DFT, which require order of n^2 ops. In practice of computations, DFT and IDFT are frequently computed on $K \geq 10,000$ or even $K \geq 100,000$ points, so the practical impact of FFT has been immense. There are several issues related to efficient implementation of FFT, as well as further techniques for reducing the complexity of computing the DFT and IDFT for specific smaller K, even though these algorithms do not decrease the asymptotic complexity estimates [8, 89, 91]. There exist public domain codes implementing FFT freely accessible via netlib [3, 4, 43, 87], and certain libraries of arbitrary-precision integer arithmetic [4, 7, 45] use FFT.

Some comments are in order on conditioning and numerical stability. It is fortunate that the DFT is a well-conditioned problem and that FFT is a numerically stable algorithm if we consider both input and output as vectors and measure the errors in terms of vector norms. More formally, let $\mathbf{x}$ and $\mathbf{y}$ be a pair of K-dimensional complex vectors, for $K = 2^k$, such that $\mathbf{y} = DFT(\mathbf{x})$ is the DFT of $\mathbf{x}$, let $FFT(\mathbf{x})$ denote the vector computed by applying the matrix version of the FFT algorithm described above and by using floating point arithmetic with d bits, $d \geq 10$, and let $e_K(x)$ express the error vector of dimension K. Then, according to [12, prop. 3.4.1], we have

$$\begin{aligned} FFT(x) &= DFT(x + e_K(x))\,; \\ ||e_K(x)|| &\leq \left(\left(1 + \rho\, 2^{-d}\right)^k - 1\right) ||x||,\ 0 < \rho < 4.83\,, \end{aligned}$$

where $||\cdot|| = ||\cdot||_2$ denotes the Euclidean norm. Moreover,

$$||e_K(x)|| \leq (5k)\, 2^{-d} ||x|| \quad \text{for any} \quad K \leq 2^{2^{d-6}}\,.$$

It immediately follows ([12, cor. 3.4.1]) that

$$||FFT(x) - DFT(x)|| \le 5\sqrt{K}(\log K)2^{-d}||x|| ,$$

if $K < 2^{2^{d-6}}$.

Similar results hold for IDFT and certain other computational problems reducible to computing DFT. In particular, we may apply FFT with a relatively low precision when we compute the product of two polynomials with integer coefficients (see "Vector Convolution"), and then rounding off still gives us the output with no error. On the other hand, polynomial division and computing the greatest common divisor (GCD) and the zeros of polynomials (see "Multiplication, Division, and Variable Shift" and "GCD, LCM, and Padé Approximation" and Chapter 16) are generally ill-conditioned problems [12, 52, 62].

The application of FFT to the computation of the continuous Fourier transform is central to several engineering and numerical applications [44]. More specifically, there are important applications to digital filters, image restoration, and numerical solution of PDEs, which are not developed here; see, for instance, [24].

Vector Convolution

Another fundamental problem equivalent to computing the DFT, as well as the generalized DFT, is the computation of the **convolution,** as well as the positive and/or negative *wrapped convolution,* of 2 vectors $u = (u_i)$ and $v = (v_i)$. Given the values $u_0, v_0, u_1, v_1, \ldots, u_n, v_n$, we seek the coefficients w_i, w_i^+ and/or w_i^- of the polynomials

$$\begin{aligned} w(x) &= \sum_{i=0}^{2n} w_i x^i, \quad w^+(x) = \sum_{i=0}^{n} w_i^+ x^i, \quad w_i^-(x) = \sum_{i=0}^{n} w_i^- x^i , \\ w_i^+ &= w_i + \hat{w}_i, \quad w_i^- = w_i - \hat{w}_i , \\ w_i &= \sum_{j=0}^{i} u_j v_{i-j}, \quad \hat{w}_i = w_{n+1+i} = \sum_{j=i+1}^{n} u_j v_{n+1+i-j}, \quad i = 0, 1, \ldots, n, \quad \hat{w}_n = 0 . \end{aligned}$$

In fact, $w(x) = u(x)v(x)$, *so that the coefficient vector of the product of 2 polynomials is the convolution* of their coefficient vectors, $w^+(x) = w(x) \bmod (x^{n+1}-1)$, $w^-(x) = w(x) \bmod (x^{n+1}+1)$, $u(x) = \Sigma_{j=0}^{n} u_j x^j$, $v(x) = \Sigma_{j=0}^{n} v_j x^j$.

We will reduce this problem essentially to the solution of a few DFT problems. Let $n+1 = K = 2^k$ for a natural k, for otherwise, we may pad $u(x)$ and $v(x)$ with l zero leading coefficients each, for $l < n$, so as to bring the degree values to the form $K - 1 = 2^k - 1$, $k = \lceil \log_2 n \rceil$. By applying Toom's evaluation–interpolation techniques [88], we may evaluate at first $u(x)$ and $v(x)$ at the $(2K)$-th roots of 1 (2 FFTs), then multiply the $2K$ computed values pairwise, which gives us the values of $w(x)$ at the $(2K)$-th roots of 1, and then obtain the coefficients of $w(x)$ (via IDFT) at the overall cost of $9K \log K + 2K$ ops. By reducing $w(x)$ modulo $(x^{n+1} \pm 1)$, we may also obtain $w^+(x)$ and $w^-(x)$.

Let us compute w^+ by performing two DFTs and an IDFT at Kth roots of 1. Consider the values of $w^+(x)$ at ω^h, where ω is a primitive $(n+1)$-st root of unity and $h = 0, \ldots, n$. Applying $\hat{w}_n = 0$, we have

$$w^+\left(\omega^h\right) = \sum_{i=0}^{n} \left(w_i + \hat{w}_i\right) \omega^{ih} = \sum_{i=0}^{n} \omega^{ih} \sum_{j=0}^{i} u_j v_{i-j} + \sum_{i=0}^{n-1} \omega^{ih} \sum_{j=i+1}^{n} u_j v_{n+1+i-j} .$$

Since $\omega^{n+1} = 1$, the second summand can be written as follows:

$$\left(\omega^{n+1}\right)^h \sum_{i=0}^{n-1} \omega^{ih} \sum_{j=i+1}^{n} u_j v_{n+1+i-j} = \sum_{i=n+1}^{2n} \omega^{ih} \sum_{j=0}^{i} u_j v_{i-j} ,$$

where the second summand is simplified by the fact that $u_j = v_j = 0$ for $j < 0$. Let $w(x)$ be the product polynomial $u(x)v(x)$. Then $w^+(\omega^h) = w(\omega^h), \quad h = 0, \ldots, n$, and we may recover $\omega^+(x)$, by means of Toom's technique, at the cost of 3 FFTs, each on the set of $(n+1)$-st roots of 1. Therefore, computing the positive wrapped convolution has complexity of $4.5K \log K + K$ ops.

A similar argument applies to the negative wrapped convolution. Let ψ be a $(2n+2)$-nd primitive root of unity and let ω and h be as above. Then $w^-(x) = u(x)v(x)$, for $x = \psi\omega^h$, since $x^{n+1} = -1$, and it suffices to use 3 FFTs, each on $n+1$ points $\psi\omega^h$, at the overall cost $4.5K \log K + K$.

Conversely, let us reduce the generalized DFT problem to convolution. We seek

$$r_h = \sum_{j=0}^{K-1} p_j \omega^{hj} = \omega^{-h^2/2} \sum_{j=0}^{K-1} p_j \omega^{(j+h)^2/2} \omega^{-j^2/2}, \quad \text{for } h = 0, \ldots, K-1 .$$

Let $w_i = r_h \omega^{h^2/2}$, $v_{i-j} = \omega^{(j+h)^2/2}$, and $u_j = p_j \omega^{-j^2/2}$, for $i = K-1, \ldots, 2K-2$ and $j = 0, \ldots, K-1$. Essentially, we change indices by setting $i = 2K-2-h$. The undefined values of u_j are zero; in particular, $u_j = 0$ for $j \geq K$. Then,

$$w_i = \sum_{j=0}^{K-1} u_j v_{i-j} + \sum_{j=K}^{i} u_j v_{i-j}, \qquad i = K-1, \ldots, 2K-2 ,$$

which is a part of the convolution problem $w_i = \Sigma_{j=0}^{i} u_j v_{i-j}$, where $i = 0, \ldots, 2K-2$ and $u_s = v_s = 0$ for $s \geq K$. Hence, the generalized DFT can be reduced to two wrapped convolutions, followed by $2K$ multiplications for computing the u_j and recovering r_h from w_i. Therefore, the *asymptotic complexity of generalized DFT* is $O(K \log K)$, *the same as for DFT and (wrapped) convolutions.*

More generally, $4.5K \log K + 2K$ ops suffice for computing the **convolution of two vectors** of lengths $m+1$ and $n+1$, respectively, which is the coefficient vector of the product of two polynomials of degrees at most m and n with given coefficient vectors, where $K = 2^k$ and $k = \lceil \log(m+n+1) \rceil$. The problem and the solution are immediately extended to multiplication of several polynomials [12].

For smaller m and n, the convolution and the associated polynomial product can be alternatively computed by means of the straightforward (classical) algorithm that uses $(m+1)(n+1)$ multiplications and $(m+1)(n+1) - m - n - 1$ additions.

For moderate m and n, $m \leq n$, one may prefer the alternative algorithm of [61], where we assume for simplicity that n is a power of 2 and rely on the recursive application of the following equations:

$$\begin{aligned} u(x)v(x) &= \left(u_0(x) + x^{n/2} u_1(x)\right)\left(v_0(x) + x^{n/2} v_1(x)\right) \\ &= u_0(x)v_0(x)\left(1 - x^{n/2}\right) + (u_1(x) + u_0(x))\,(v_1(x) + v_0(x))\,x^{n/2} \\ &\quad + u_1(x)v_1(x)\left(x^n - x^{n/2}\right) . \end{aligned} \tag{17.2}$$

The algorithm uses $O(n^{\log 3})$ ops, where $\log 3 = 1.5849\ldots$, and has a small overhead constant.

Sine and Cosine Transforms

Besides FFT, other related transforms widely used in signal processing include *sine, cosine, Hartley,* and *wavelet* transforms. Given a vector $\mathbf{y} = [y_1, \ldots, y_n]$, its sine transform can be defined as the vector $\mathbf{x} = [x_1, \ldots, x_n]$, where

$$x_i = \sum_{j=1}^{n} y_j \sin \frac{\pi i j}{n+1}, \quad i = 1, \ldots, n ,$$

or, equivalently, $\mathbf{x} = \sqrt{\frac{n+1}{2}}Sy$, where $S = [\sqrt{\frac{2}{n+1}} \sin \frac{\pi ij}{n+1}]$, $i, j = 1, \ldots, n$, $S^T = S^{-1} = S$. The cosine transform of vector $[y_j]$ can be defined analogously by substituting sine by cosine. For further variants of sine and cosine transforms see [56, 81].

The sine and cosine transforms can be performed in $O(n \log n)$ ops by means of FFT [56].

An important application of the sine transform is in computations with the matrix algebra τ, introduced in [9]; further applications can be found in [53]. This algebra consists of all the $n \times n$ matrices $A = [a_{ij}]$, such that

$$\begin{aligned} a_{i,j-1} + a_{i,j+1} &= a_{i-1,j} + a_{i+1,j}, \quad i, j = 0, 1, \ldots, n-1\,; \\ a_{i,j} &= 0 \text{ if } i \in \{-1, n\}, \text{ or } j \in \{-1, n\}\,. \end{aligned}$$

Then A is also denoted $\tau(a)$, where a is vector $[a_{0,0}, a_{1,0}, \ldots, a_{n-1,0}]$, i.e., the first column of A. Several properties of this matrix algebra, including its connection to Chebyshev-like polynomials, are discussed in [12, ch. 2].

For any matrices $A \in \tau$ and S defined above, $SAS = D$ is a diagonal matrix with nonzero entries $d_1, \ldots, d_n$, given by $d_i = (Sa)_i / (\sqrt{2/(n+1)} \sin \frac{\pi i}{n+1})$, where a denotes the first column of matrix A. Furthermore, given two vectors u, v, each having n components, the following vectors can be computed in $O(n \log n)$ ops, by means of a constant number of sine transforms: $\tau(u)v$, the first column of $\tau(u)\tau(v)$, and $\tau(u)^{-1}v$, if the matrix $\tau(u)$ is nonsingular. The matrix algebra τ is important in the analysis of the spectral properties of band Toeplitz matrices and the computation of their tensor rank, as well as in solving Toeplitz and block Toeplitz linear systems (see "Circulant, Toeplitz, and Hankel Matrices").

The reader is referred to [10, 86] on wavelet and Hartley transforms.

17.3 Fast Polynomial and Integer Arithmetic

Computations with integers and polynomials, in one or more variables, is of fundamental importance in computational mathematics and computer science. Such operations lie at the core of every computer algebra system. In this section, we focus on univariate polynomials and show some extensions to integer arithmetic and multivariate polynomials. The connection between vector manipulation, on the one hand, and univariate polynomial and integer arithmetic, on the other hand, relies on representing a polynomial or an integer by a vector of coefficients or digits, respectively. A large number of problems in science and engineering can be solely expressed in terms of polynomials. In addition, polynomials serve as the basis for the study of more complex structures, such as rational functions, algebraic functions, power series and transcendental functions.

The first theme of this section is to apply the transforms seen so far in order to yield the record asymptotic upper bounds on the complexity of several fundamental operations with univariate polynomials. In exploiting the power of FFT, we are implicitly assuming that the polynomials are dense; in other words, most of their coefficients are nonzero and, therefore, it makes sense to represent them by the vector of their coefficients. Different representations of sparse polynomials (for instance, by the straight-line programs for their evaluation) and different complexity measures (for instance, in terms of the output size) can be found in [1, 17, 57, 92].

In particular, we cover polynomial multiplication, division, evaluation and interpolation, as well as further extensions of these basic operations. Our study shows that all of these problems have the same asymptotic complexity within a logarithmic factor, which is due to their correlation to DFT, IDFT, and vector convolution.

Some algorithms for manipulating univariate polynomials can be adapted to performing the analogous operation over the integers and vice versa (see "Integer Arithmetic").

Lastly, "Multivariate Polynomials" examines the case of polynomials in more than one indeterminates.

Multiplication, Division, and Variable Shift

Multiplication of univariate polynomials is an important and fundamental operation, to which all other important operations can be ultimately reduced. This section also studies division of polynomials with remainder and computing the reciprocal of a polynomial. We show that both of these operations, as well as the transformation of a polynomial under a shift of the variable, have the same asymptotic complexity as multiplication. The "linear" operations of addition, subtraction, and multiplication by a constant for degree n polynomials can be performed in $n + 1$ ops.

Polynomial *multiplication* is the problem of computing the coefficients $w_i = \Sigma_{j=0}^{i} u_j v_{i-j}$, $i = 0, 1, \ldots, m + n$, of the polynomial $w(x) = u(x)v(x)$ provided that we are given the coefficients of a pair of polynomials $u(x)$ and $v(x)$,

$$u(x) = \sum_{i=0}^{m-1} u_i x^i, \quad v(x) = \sum_{i=0}^{n-1} v_i x^i ,$$

and that $u_j = 0$, for $j > m$, $v_k = 0$, for $k > n$. We have already solved this problem, in $O(K \log K)$ ops, for $K = m + n$, in "Vector Convolution" as the problem of computing the convolution of 2 vectors.

Given polynomials $u(x)$ and $v(x)$ of degrees m and n, respectively, polynomial *division* with a *remainder* is the computation of the coefficients of the unique pair of polynomials $q(x) = \Sigma_{g=0}^{m-n} q_g x^g$ (quotient) and $r(x) = \Sigma_{h=0}^{n-1} r_h x^h$ (remainder) such that

$$u(x) = q(x)v(x) + r(x), \quad \deg r(x) < n .$$

$r(x)$ is also called the **residue** of $u(x)$ modulo $v(x)$ and is denoted $u(x) \bmod v(x)$.

A related problem is the computation of the *reciprocal* of a polynomial. Given a natural n and a polynomial $u(x)$, $u(0) \neq 0$, compute the first n coefficients of the formal power series $w(x)$ such that $w(x)u(x) = 1$ or, equivalently, compute $w(x) \bmod x^n$. Both problems of polynomial division and computation of the reciprocal can be reduced to polynomial multiplication [1, 12, 14] and solved in $O(n \log n)$ ops.

Another reduction of polynomial division to FFT is direct, by means of Toom's techniques of evaluation-interpolation with no reduction to polynomial multiplication. This is easily done where it is known that $r(x) \equiv 0$, but in [76] such a reduction to FFT has been elaborated for parallel approximate division of any pair of polynomials.

So far, we have assumed computations in the fields that support $O(n \log n)$ reduction to FFT. For an arbitrary ring of constants with unity, only the reduction to FFT in $O(n \log n \log \log n)$ is supported, so that the cited complexity estimates for these operations with polynomials are multiplied by an $O(\log \log n)$ factor, thus, yielding an $O(n \log n \log \log n)$ overall bound [21]. Over any ring of constants, however, multiplication, division, and the reciprocal computation have complexities related by constant factors, since each of them can be solved by means of a *constant number* of applications of any one of the others [12]. Hereafter, we will write $M(n)$ to denote the asymptotic arithmetic complexity of solving these problems, where n is the degree of the input polynomials.

Shifting the variable, also known as a **Taylor shift,** is the problem of computing the coefficients $q_0(\Delta)$, $q_1(\Delta), \ldots, q_n(\Delta)$ of the polynomial

$$q(y) = p(y + \Delta) = \sum_{h=0}^{n} p_h (y + \Delta)^h = \sum_{g=0}^{n} q_g(\Delta) y^g ,$$

given a scalar Δ and the coefficients $p_0, p_1, \ldots, p_n$ of a polynomial $p(x)$. This problem is solved in $O(M(n))$ ops [2, 12].

The practical impact of using FFT and the fast convolution algorithm in computer algebra, however, is more limited than one may suspect, for several reasons. In many cases, the application of the fast

convolution algorithm requires special care in order to contract the resulting growth of the precision needed in order to represent the auxiliary parameters [39]. In addition to the classical algorithms, supporting the bound $M(n) = O(n^2)$, and one of [61], supporting the bound $M(n) = O(n^{\log 3})$, strong competitors to FFT and fast convolution are the algorithms based on the techniques of binary segmentation (see Chapter 16), which is effective for computations in finite fields and in other cases where the input parameters are represented by short binary integers.

Evaluation, Interpolation, and Chinese Remainder Computations

Historically, the first topic in the algebraic computational complexity theory is the evaluation of a polynomial at a single point. Given the coefficients of a polynomial

$$p(x) = p_0 + p_1 x + \cdots + p_n x^n ,$$

compute its value at a point x_0. Newton's optimal solution, known as Horner's rule, is based on the decomposition

$$p(x_0) = (\cdots((p_n x_0 + p_{n-1})x_0 + p_{n-2})x_0 + \cdots + p_1)x_0 + p_0$$

and uses n multiplications and n additions [1, 14, 62], though about 50% of multiplications can be saved if we allow to precondition the coefficients and if we count only operations depending on x [62].

More generally, given the coefficients p_i of a polynomial $p(x)$, one may need to evaluate $p(x)$ on a fixed set of points $\{x_0, \ldots, x_{K-1}\}$. The problem can be reduced to K applications of Horner's rule and solved in $2Kn$ ops. Alternative reduction to polynomial multiplications and divisions yields an $O(M(m)\log m)$ algorithm, where $m = \max\{n, K\}$. The cost turns into $O(m\log^2 m)$ provided that $M(m) = O(m\log m)$, where polynomial arithmetic is implemented based on FFT. We will show the two stages of this approach referring to [1, 12, 14] for further details.

Assume that $K \le 2^k \le n+1$. Note that $p(x_i) = p(x) \bmod (x - x_i)$, $i = 0, \ldots, K-1$. The first stage is based on the *fan-in method.* Successively compute the polynomials $m_i^{(j)} = m_{2i}^{(j-1)} m_{2i+1}^{(j-1)}$, for $i = 0, \ldots, \frac{K}{2^j} - 1$ and $j = 1, \ldots, k-1$, starting with the given moduli $m_i^{(0)} = x - x_i$ for all i. The degree of $m_i^{(j)}$ in x is 2^j, so that for a fixed j and given the $m_i^{(j-1)}$, we compute all $m_i^{(j)}$ in $2^{k-j}M(2^j)$ ops.

The second stage is called the *fan-out method* and is another instance of the divide-and-conquer approach. Since $m_0^{(k-1)} = \prod_{i=0}^{K/2-1} m_i$ and $m_1^{(k-1)} = \prod_{i=K/2}^{K-1} m_i$, we have

$$p(x) \bmod (x - x_i) = \left(p(x) \bmod m_0^{(k-1)}\right) \bmod (x - x_i), \ i = 0, \ldots, K/2 - 1, \quad \text{and}$$
$$p(x) \bmod (x - x_i) = \left(p(x) \bmod m_1^{(k-1)}\right) \bmod (x - x_i), \ i = K/2, \ldots, K - 1.$$

Thus, we have reduced the evaluation of $p(x)$ to the evaluation of two polynomials of roughly half degree, by means of two polynomial divisions. This stage continues recursively and has the same asymptotic complexity as the fan-in stage, since polynomial division and multiplication have the same asymptotic complexity.

The overall complexity is $O(M(n)\log K)$. The algorithm can be adapted to the case $K > n+1$ with complexity $O((K/n)M(n)\log K)$. Hence, the bound $O(M(m)\log m)$, stated earlier, for $m = \max\{n, K\}$, which grows to $O(m^{\log 3}\log m)$, $\log 3 = 1.5849\ldots$, if, instead of FFT, we apply the method of [61] for polynomial multiplication.

In short, *evaluation on a set of points reduces to multiplications,* which can be implemented based on DFT. The computation of the latter is, obviously, a special case of the evaluation problem. Thus, all basic problems seen so far can be *reduced to any one of them,* at most with a polylogarithmic asymptotic overhead.

This is also the case with *interpolation* to a function by a polynomial, which is the inverse of the polynomial evaluation problem and is stated as follows: Given two sets of values,

$$\left\{x_i \ : \quad i = 0, \ldots, n; \quad x_i \neq x_j \text{ for } i \neq j\right\} \ , \quad \{r_i \ : i = 0, \ldots, n\} \ ,$$

evaluate the coefficients $p_0, p_1, \ldots, p_n$ of the polynomial $p(x)$ of Eq. (17.1) satisfying $p(x_i) = r_i, i = 0, 1, \ldots, n$. The problem always has a unique solution, which the classical interpolation algorithms compute in $O(n^2)$ ops [30, 48]. However, by using FFT, $O(n \log^2 n)$ ops suffice. This bound is optimal up to the factor $O(\log n)$ [14, 19, 62].

The fast algorithm [12, 40] uses the Lagrange interpolation formula:

$$p(x) = L(x) \sum_{i=0}^{n} \frac{r_i}{L'(x_i)\,(x - x_i)}, \qquad \text{where} \qquad L(x) = \prod_{i=0}^{n} (x - x_i) \ , \tag{17.3}$$

$L'(x_i) = \prod_{k=1,\ k\neq i}^{n} (x_i - x_k)$ for $i = 0, 1, \ldots, n$, and $L'(x)$ is the formal derivative of $L(x)$. Interpolation then reduces to application of, at first, the fan-in method to computing the polynomials $L(x)$ and $L'(x)$, secondly, the evaluation algorithm to finding the values $L'(x_i)$ for all i, and, thirdly, polynomial multiplications to obtain $p(x)$ from (17.3). The overall cost is $O(M(n) \log n)$, where $M(n) = O(n \log n)$ is used.

A generalization of interpolation is the *Chinese remainder* problem. Here, we shall examine its univariate polynomial version, though its name comes from its application to the integers (cf. "Integer Arithmetic"), by Chinese mathematicians in the 2nd century A.D. or even earlier [62]. Let us be given the coefficients of $2h$ polynomials $m_i(x), r_i(x), \ i = 1, \ldots, h$, where $\deg m_i(x) > \deg r_i(x)$ and where the polynomials $m_i(x)$ are pairwise relatively prime, that is, pairwise have only constant common divisors or, equivalently, $\gcd(m_i, m_j) = 1$ when $i \neq j$. Then, we are asked to compute the unique polynomial $p(x) = p(x) \bmod \prod_{i=1}^{h} m_i(x)$, such that $r_i(x) = p(x) \bmod m_i(x), i = 1, \ldots, h$. When every $m_i(x)$ is of the form $x - x_i$, we come back to the interpolation problem.

The Chinese remainder theorem states that there always exists a unique solution to this problem. The importance of this theorem cannot be overestimated, since it allows us to reduce computation with a high-degree polynomial to similar computations with several smaller-degree polynomials, namely, to computations modulo each $m_i(x)$. Then the **Chinese remainder algorithm** combines the results modulo each $m_i(x)$ so as to yield $p(x)$.

There are two approaches to recovering $p(x) = p(x) \bmod \prod_{i=1}^{h} m_i(x)$ from given $r_i(x)$ and $m_i(x)$, $i = 1, \cdots, h$. The first uses essentially Lagrange's interpolation formula, simply by generalizing the polynomial values in formula (17.3) to polynomial remainders. The second approach is named after Newton and is incremental, in the sense that successive steps compute $p_{k-1}(x) = p(x) \bmod \prod_{i=1}^{k} m_i(x)$, for $k = 1, 2, \ldots, h$, where the last step gives the final result:

$$p_k(x) = p_{k-1}(x) + \Big(\big(r_k(x) - p_{k-1}(x)\big)\, s_k(x) \bmod m_k(x)\Big) \prod_{i=1}^{k-1} m_i(x) \ ,$$

$$\text{where } \ s_k(x) \prod_{i=1}^{k-1} m_i(x) \bmod m_k(x) = 1, \quad k = 1, 2, \ldots, h, \quad p_h(x) = p(x) \ .$$

Let $N = \sum_{i=1}^{h} d_i, \ d_i = \deg m_i(x)$. Then the overall complexity of computing $p_h(x) = p(x)$ by this algorithm is $O(M(N)h + \Sigma_{i=1}^{h} M(d_i) \log d_i)$, which, based on FFT, turns into $O(Nh \log N)$.

Further details can be found in [1, 12, 14, 17, 62, 92].

GCD, LCM, and Padé Approximation

In this section, we will study the computation of the **greatest common divisor (GCD)** and the **least common multiple (LCM) of two polynomials**, and we will list some of their numerous applications to polynomial and rational computations.

The classical solution method is the Euclidean algorithm, which is a major general tool for many algebraic and numerical computations. Despite its ancient origins, the problem of computing greatest common divisors, with its numerous facets, is still an active area of research.

Given the coefficients of two polynomials

$$u(x) = \sum_{i=0}^{m} u_i x^i, \ v(x) = \sum_{j=0}^{n} v_j x^j, \ m \geq n \geq 0, \ u_m v_n \neq 0 \, ,$$

their greatest common divisor, denoted $\gcd(u(x), v(x))$, is a common divisor of $u(x)$ and $v(x)$ having the highest degree in x. The GCD of $u(x)$ and $v(x)$ is unique up to within constant factors, or it can be assumed monic, and then it is unique. For example,

$$\gcd\left(x^5 + x^4 + x^3 + x^2 + x + 1, x^4 - 2x^3 + 3x^2 - x - 7\right) = x + 1 \, ,$$

because $x + 1$ divides both polynomials and they have no common divisor of degree greater than or equal to two.

Algorithm 17.1 (Euclid's). *Set* $u_0(x) = u(x), \ v_0(x) = v(x)$. *Compute*

$$\begin{aligned} u_{i+1}(x) &= v_i(x) \, , \\ v_{i+1}(x) &= u_i(x) \bmod v_i(x) = u_i(x) - q_{i+1}(x) v_i(x), \ i = 0, 1, \ldots, \ell - 1 \, , \end{aligned} \tag{17.4}$$

where $q_{i+1}(x)$ is the quotient polynomial, and ℓ is such that $v_\ell(x) = 0$. At the end of this process, $u_\ell(x) = \gcd(u(x), v(x))$.

The correctness of the algorithm can be deduced from the following equation:

$$\gcd\left(u_i(x), v_i(x)\right) = \gcd\left(u_{i+1}(x), v_{i+1}(x)\right) \, ,$$

which holds for all i. Euclid's algorithm only involves arithmetic operations, so that the output coefficients of the GCD are rational functions in the input coefficients. Assuming that $m = n$, the algorithm involves $O(n^2)$ ops, but using a matrix representation of the recurrence and the fan-in method yields an $O(M(n) \log n)$ bound [12].

The sequence $v_1(x), v_2(x), \ldots, v_l(x)$ is called a *polynomial remainder sequence.* It can be generalized to the sequence of remainder polynomials obtained if the division step (17.4) is substituted by

$$a_{i+1} v_{i+1}(x) = b_i u_i(x) \bmod v_i(x) = b_i u_i(x) - q_{i+1}(x) v_i(x), \ i = 0, 1, \ldots, \ell - 1 \, ,$$

where the a_{i+1} and b_i, for $i = 0, \ldots, \ell - 1$, are scalars. When all scalars are units we recover the original equation. The polynomial *pseudo-remainder sequence* is obtained by setting $a_{i+1} = 1$ and $b_i = c_i^{d_i}$, for $i = 0, \ldots, \ell - 1$, where c_i is the leading coefficient of $v_i(x)$ and $d_i = \deg u_i(x) - \deg v_i(x) + 1$. There exists the third remainder sequence, called the *subresultant* sequence, which is closely related to the Sylvester resultant (see Chapter 16). These variations aim at palliating the swell of the intermediate coefficients. For details, consult [1, 12, 17, 33, 57, 92].

A closely related problem is the least common multiple (LCM) computation. Given the coefficients of the two polynomials $u(x)$ and $v(x)$, compute the coefficients of $\operatorname{lcm}(u(x), v(x))$, that is, of a common

multiple of $u(x)$ and $v(x)$ having the minimum degree. In the previous example, the LCM is $x^8 - 2x^7 + 4x^6 - 3x^5 - 3x^4 - 3x^3 - 4x^2 - x - 7$. Given $u(x)$, $v(x)$, and their GCD, we may immediately compute

$$\operatorname{lcm}(u(x), v(x)) = \frac{u(x)v(x)}{\gcd(u(x), v(x))} .$$

The cited references for the GCD problem also present alternative algorithms for the LCM; the best asymptotic complexity bound is $O(M(n)\log n)$, if $n \geq m$.

In the (m, n) *Padé approximation* problem, we are given two natural numbers m and n and the first $N + 1 = m + n + 1$ Taylor coefficients of an analytic function $V(x)$ decomposed at $x = 0$, and we are seeking two polynomials $R(x)$ and $T(x)$ satisfying the relations

$$R(x) - T(x)V(x) = 0 \bmod x^{N+1} , \ N = m + n, \ \deg T(x) \leq n, \ \deg R(x) \leq m .$$

This is actually a special case of Hermite's interpolation problem (cf. [12] on both subjects). Its complexity is $O(M(N)\log N)$ ops. An alternative solution of [15] relying on Toeplitz computations (discussed in "Circulant, Toeplitz, and Hankel Matrices") requires $O(N\log^3 N)$ ops but leads to a faster parallel algorithm [12].

A well-known application of Padé approximation is to computing the *minimum span for a linear recurrence,* also known as the *Berlekamp–Massey* problem and having important applications to algebraic coding theory, sparse polynomial interpolation, and parallel matrix computations [6, 12, 49, 59, 92]. Given a natural s and $2s$ numbers $v_0, \ldots, v_{2s-1}$, compute the minimum natural $n \leq s$ and n numbers $t_0, \ldots, t_{n-1}$ such that $v_i = t_{n-1}v_{i-1} + \cdots + t_0 v_{i-n}$, for $i = n, n + 1, \ldots, 2s - 1$. Its solution can be obtained by extending the solution of Padé approximation problem and requires $O(M(s)\log s)$ ops [6, 15, 75].

Integer Arithmetic

This section carries over the results seen so far for univariate polynomials to the domain of integers. The basic premise of this correlation, which goes in both directions, is that any binary integer can be thought of as a polynomial with coefficients in {0, 1}. For instance, the binary representation of 24 is 11000, which corresponds to the univariate polynomial $u(x) = x^4 + x^3$, which has degree 4 and the coefficient vector [1, 1, 0, 0, 0]. One difference is that operating with integers we must take care of the carry bits.

As a result, the algorithms for some basic operations on univariate polynomials can be applied to the integers and vice versa, and in many cases the complexity estimates do not change significantly when we shift from arithmetic operations with polynomials to binary operations with integers [1, 12]. This is illustrated below for the multiplication and GCD algorithms. This section also emphasizes the role of Chinese remaindering, which, as we mentioned already, has been historically introduced in the context of integers but also applies to polynomials. Likewise, the algorithms of "Multiplication, Division, and Variable Shift" for polynomial multiplication can be applied to integer multiplication modulo $2^N + 1$, whose complexity we denote $\mu(N)$. Assume that for a given pair of N-bit integers u and v,

$$u = \sum_{i=0}^{N-1} u_i 2^i , \quad v = \sum_{i=0}^{N-1} v_i 2^i , \quad u_i, v_i \in \{0, 1\} ,$$

we seek the integer uv. The classical algorithm has complexity $O(N^2)$. [61] recursively apply the equation

$$uv = U_0V_0\left(1 - 2^{N-1}\right) + (U_1 + U_0)(V_1 + V_0)\,2^{N-1} + U_1V_1\left(2^{2N-2} - 2^{N-1}\right) ,$$

where $u = U_0 + 2^{N-1}U_1$ and $v = V_0 + 2^{N-1}V_1$; see Eq. (17.2). This algorithm uses $O(N^{\log 3})$ Boolean operations, where $\log 3 = 1.5849\ldots$, and has a small overhead constant. The evaluation–interpolation

idea demonstrated for polynomial multiplication enable us to reduce the complexity to $O(N^{1+\epsilon})$ for any positive ϵ. Finally, by exploiting fast convolution by means of FFT over finite rings of constants, Schönhage and Strassen reduced the asymptotic complexity bound to $O(N \log N \log\log N)$; see [84, 85], [1, p. 270–274], or [12, p. 78–79]. Only the (obvious) information lower estimate of order N is known for $\mu(N)$.

In practice, the Schönhage–Strassen algorithm is used only for very large N, for instance, in applications to polynomial root-finding, because the overhead constant is considerable. At present, the algorithms used in practice otherwise are either the classical method or the one of [61]. Recently, however, in addition to the two latter algorithms, the algorithm of [85] has been implemented on modern computers [7, 45].

Integer division is the problem where, given two positive integers u, v with the bit sizes n, m, respectively, we seek the unique pair of integers q, r for which $u = qv + r$, where $0 \leq r < v$. The classical algorithm has Boolean complexity $O(mn)$, whereas the fast multiplication algorithm yields the $O(\mu(m))$ bound [1, 14, 62]. The asymptotic complexity of integer multiplication and division is actually the same [1, 12].

Given two **integers** u and v, their **greatest common divisor (GCD)** is the largest integer that divides both u and v, and their **least common multiple (LCM)** is the smallest integer that is divisible by both u and v. For instance, $\gcd(16, 24) = 8$, whereas $\mathrm{lcm}(16, 24) = 48$. The Euclidean algorithm was the historically first algorithm for integer GCD. The GCD of a pair of positive integers less than 2^n can be computed in $O(\mu(n) \log n) = O(n \log^2 n \log\log n)$ Boolean ops, where $\mu(n)$ denotes the Boolean complexity of multiplying two integers modulo $2^n + 1$ [1, 12, 50, 62].

Likewise, historically, the Chinese remainder algorithm has been first devised for integers. Given the **integer residues** r_i with respect to fixed integer moduli m_i, for $i = 0, 1, \ldots, k$, where $\gcd(m_i, m_j) = 1$ for $i \neq j$, we seek an integer $p = p \bmod \prod_{i=0}^{k} m_i$ such that $r_i = p \bmod m_i$ for all i. The Chinese remainder theorem states that such an integer p exists and is unique. The Boolean complexity of computing such an integer p is $O(\mu(N) \log N)$ ops, where $N = \Sigma_i \lceil \log m_i \rceil$.

An important application of the Chinese remainder theorem is in reducing the bulk of an arbitrary-precision integer computation to computations with fixed-precision integers. We map the input integers into their residues moduli m_i, then perform the computation in the finite field or ring of integers modulo m_i for each i, and, finally, use the Chinese remainder algorithm in order to compute the exact answer (see [36] for extensions).

The above technique of modular arithmetic is one of the most efficient methods for conducting integer (as well as rational) arithmetic on modern computers. Typically, the moduli used are primes that fit in a computer word. The implementation of the Chinese remainder algorithm relies either on an extension of Lagrange's formula (17.3) or on Newton's incremental approach (cf. "Evaluation, Interpolation, and Chinese Remainder Computations"). For a comprehensive discussion as well as other alternatives, see [1, 12, 14, 17, 62, 92].

Multivariate Polynomials

Polynomials in several variables generalize the univariate case, which we have examined so far. Multivariate polynomials appear in a wide variety of scientific and engineering applications (see Chapter 16 of this handbook). This section outlines some basic results regarding multiplication, evaluation and interpolation of multivariate polynomials over arbitrary rings of constants.

Just as we did in the univariate case, we do not consider in depth the question of representation, but assume that polynomials are represented by a coefficient vector indexed by the corresponding monomials in some order. Other operations, such as addition, subtraction, taking integer powers and division, as well as the various representations are treated extensively in [17, 92].

A polynomial in n variables is of the form

$$p(x_1, \ldots, x_n) = \sum_{i_1, \ldots, i_n} p_{i_1, i_2, \ldots, i_n} x_1^{i_1} x_2^{i_2} \ldots x_n^{i_n} ,$$

where the coefficients correspond to distinct monomials. Recall that the degree of p in x_j is the maximum i_j for which $p_{i_1, i_2, \ldots, i_n} \neq 0$, the total degree of a monomial $x_1^{i_1} x_2^{i_2} \cdots x_n^{i_n}$ is $i_1 + \cdots + i_n$, and the total degree of the polynomial p is the maximum total degree of any monomial. If d is the maximum degree in any variable, then the total number of terms is $O(d^n)$. A polynomial most of whose coefficients are nonzero is called *dense*.

To compute the product of two such polynomials, we may reduce the problem to the univariate case by means of Kronecker's substitution:

$$x_1 = y , \quad x_{k+1} = y^{D_1 \ldots D_k} , \quad k = 1, \ldots, n-1 .$$

Here, $D_j = 2d_j + 1$ exceeds the degree in x_j of the product polynomial provided that both input polynomials have degrees at most d_j in x_j, $j = 1, 2, \ldots, n$. Once the univariate product is computed, we may recover the product polynomial in the n variables x_i by inverting the Kronecker map. This method yields the complexity bound $O(N \log N \log\log N)$, where $N = \prod_{i=1}^{n} D_i$.

A certain improvement for polynomials with a large number of variables in [73] relies on the evaluation–interpolation method and (over any field of constants) yields complexity bound

$$O\left(N \log N \log\log D\right) = O\left(n D^n \log D \log\log D\right) , \tag{17.5}$$

where $D = \max\{D_1, \ldots, D_n\}$, $N = D^n$, and the number of terms is of order D^n.

Another algorithm for dense polynomials is due to [20], also relies on the evaluation–interpolation scheme, but only applies over fields of characteristic 0. The algorithm supports, for a product polynomial with at most T terms, each of a total degree at most T, the complexity bound $O(M(T) \log T) = O(T \log^2 T \log\log T)$, which is inferior to the estimate (17.5) under the bound D on the degree of each variable but superior under the bound T on the total number of terms.

Alternatively, we may extend the algorithm of [61], having complexity $O(D^{n \log 3})$.

Evaluation and interpolation of dense multivariate polynomials may use points on a grid (or lattice) in which each variable is assigned the values in a fixed set. Let $E(d)$ and $I(d)$ denote, respectively, the complexity of evaluating and interpolating a univariate polynomial of degree bounded by $(d-1)/2$ on d points. Then, for grids of d_j values for each variable x_j, the complexity of evaluation and interpolation of a multivariate polynomial is $d^{n-1} E(d)$ and $n d^{n-1} I(d)$, respectively, where $d = \max\{d_1, d_2, \ldots, d_n\}$. By applying FFT, we yield the bounds $O(d^n \log^2 d)$ and $O(n d^n \log^2 d)$, for evaluation and interpolation, respectively. For dense polynomials, these bounds are satisfactory. The approach of [20] adapts the algorithm of [5] and also relies on solving a transposed Vandermonde system fast. It yields the bounds

$$O\left(T \log T \log\log T \log t\right) \qquad \text{and} \qquad O\left(T \log^2 T \log\log T\right)$$

for evaluation and interpolation, respectively, over fields of characteristic zero, where t is the actual number of nonzero terms and T is an upper bound on the number of input terms. Alternatively, we can use the total degree of the product polynomial to estimate T, which is typically the case when this technique is applied to multiplication.

With multivariate polynomials, sparsity considerations become more important. Typically, the critical computation is interpolation. On the other hand, evaluation depends strongly on the form in which the polynomial is expressed and which is often a **determinant** formula [64]. In addition to multiplication, computing the GCD can also be reduced to interpolation [92]. So, in the rest of this section, we concentrate on sparse multivariate interpolation.

There are two main approaches with different advantages and drawbacks. One approach is Zippel's randomized algorithm. Its merit is that it does not require a bound on the number of nonzero terms as input but requires a bound d on the degree in each variable. The computation involves Vandermonde matrices (cf. "Vandermonde and Cauchy (generalized Hilbert) Matrices"), and its complexity is $O^*(nd^2t)$, where t denotes the number of nonzero terms of the input polynomial, and $O^*(\cdot)$ indicates that some polylogarithmic factors may have been omitted. There exists a deterministic version of this algorithm with higher, but still polynomial, complexity. For further information, see [49, 58, 92].

Another, historically the first, approach is due to [5]. The algorithm of [5] does not need any degree bounds, but uses a bound on the actual number of terms t, on which both the algorithm and its estimates complexity, $O^*(ndt)$, depend. The algorithm applies to fields of characteristic equal to zero or to a very large positive integer. It proceeds by finding the exponents of the nonzero monomials which is reduced to the solution of the Berlekamp–Massey problem (cf. "GCD, LCM, and Padé Approximation"). Then, at the cost $O^*(ndt)$, the algorithm computes the corresponding coefficients, by exploiting the structure of Toeplitz and Vandermonde matrices [5, 58].

17.4 Structured Matrices

Matrices with special structure are encountered in several applications to problems in sciences and engineering. These matrices have several repeated entries or entries that satisfy certain relations. More formally, an $n \times n$ **structured matrix** is typically defined by $O(n)$ entries, say by less than $2n$ entries, instead of the n^2 entries required in order to specify a general matrix. Moreover, structured matrices can typically be multiplied by a vector in $O(n \log n)$ or $O(n \log^2 n)$ ops, instead of $2n^2 - n$ ops, required for a general matrix. Structured matrices arise in numerous applications such as control, signal and image processing, coding, a variety of algebraic computations, solution of PDEs, integral equations, singular integrals, conformal mappings, particle simulation, and Markov chains [24, 66]. In addition, several fundamental parallel computations with general matrices can be effectively reduced to computations with structured matrices [12].

This section focuses on computations with dense structured matrices, including Vandermonde, generalized Hilbert or Cauchy, circulant, Toeplitz, Hankel, and Bézout matrices. Computations with matrices of these classes are strongly related to computations with polynomials, thus, enabling us to employ FFT in order to arrive at a dramatic acceleration of the algorithms versus the case of general matrices. For example, solving a nonsingular linear system of n equations with a structured matrix of coefficients takes $O(n \log^2 n)$ or $O(n \log n)$ ops. Furthermore, a substantial improvement of parallel computations with general matrices can be obtained based on their reduction to computations with dense structured matrices and polynomials.

We refer the reader to [12] for information about other important classes of structured matrices, for instance, Frobenius (companion) matrices.

Hereafter, $(A)_{i,j}$ denotes the (i, j)-th entry of a matrix A.

Vandermonde and Cauchy (generalized Hilbert) Matrices

This section defines two important classes of structured matrices and demonstrates the improvement (versus general matrices) in the running time of basic matrix operations when this structure is exploited, based on correlation to computations with polynomials and FFT.

An $m \times n$ *Vandermonde* matrix V has its entries $(V)_{i,j} = v_i^j$ for $i = 0, 1, \ldots, m$, $j = 0, 1, \ldots, n$. If $m = n$, then V has the determinant

$$\det V = \prod_{0 \le i < k \le n} (v_k - v_i) \ . \tag{17.6}$$

Clearly, a square Vandermonde matrix is nonsingular if and only if $v_i \neq v_k$ for $i \neq k$.

For example, for $m = 1$ and $n = 3$ we have

$$V = \begin{pmatrix} 1 & v_0 & v_0^2 & v_0^3 \\ 1 & v_1 & v_1^2 & v_1^3 \end{pmatrix}.$$

Some authors call V^T, rather than V, a Vandermonde matrix. A *generalized Vandermonde* matrix G (cf. [92]) is defined by $(G)_{i,j} = v_i^{e_j}$, $i = 0, \ldots, m$, $j = 0, \ldots, n$, for some sequence $0 \leq e_0 < e_1 < \cdots < e_{n-1}$ of integers.

An important example of a Vandermonde matrix is given by the matrix $\sqrt{K}\Omega$, which is the scaled *Fourier matrix* $\Omega = (\omega^{ij}/\sqrt{K})$, $K = n + 1$, associated to the DFT and the inverse DFT, where ω is a primitive nth root of 1. Different FFT algorithms correspond to different factorizations of the matrix ω; see [89] and [12, Sect. 3.4].

Multiplication of a Vandermonde matrix $V = (v_i^j)$ by a vector $\mathbf{p}$ is equivalent to the evaluation at the points $v_0, \cdots, v_m$ of the polynomial with the coefficient vector $\mathbf{p}$. The solution of a linear system $V\mathbf{x} = \mathbf{v}$, where $\mathbf{x}$ and $\mathbf{v}$ are vectors and V is a nonsingular square Vandermonde matrix, is equivalent to interpolating from the vector $\mathbf{v}$ of the polynomial values at the points $v_0, \cdots, v_m$ to the coefficient vector $\mathbf{x}$. Due to the algorithms of "Evaluation, Interpolation, and Chinese Remainder Computations," both of these operations with Vandermonde matrices can be performed in $O((m + n)\log^2(m + n))$ ops, which is a dramatic decrease versus the case of a general $m \times n$ matrix. On the other hand, the estimated parallel complexity of these operations with general matrices can be decreased based on their reduction to operations with other structured matrices (see "Correlations among Structured Matrices"), and the Vandermonde–polynomial correlations have also been exploited in the reverse direction, in order to improve the known methods for polynomial evaluation and interpolation [77, 79].

The same operations with V^T can be performed in $O(n \log^2 n)$ ops, where V is an $n \times n$ Vandermonde matrix (see Algorithm 17.2 in "Circulant, Toeplitz, and Hankel Matrices"). The same bound also holds for computing the absolute value $|\det V|$. The speed of performing all these computations relies on their reduction to basic polynomial operations and, ultimately, to FFT. Straightforward computation of the determinant of an $n \times n$ Vandermonde matrix V requires $O(n^2)$ ops, based on Eq. (17.6), but for a real matrix V [47] also determines the sign of $\det V$, by means of ordering $v_0, \cdots, v_n$ and then using $O(n \log n)$ comparisons.

The next class of structured matrices is named after Cauchy. An $m \times n$ *Cauchy matrix* C is defined by two vectors $\mathbf{s} = [s_i]$ and $\mathbf{t} = [t_j]$, such that $s_i \neq t_j$ for $i = 0, \ldots, m - 1$, $j = 0, 1, \ldots, n - 1$, and $(C)_{ij} = 1/(t_i - s_j)$. For instance, if $m = 2$, $n = 3$, then

$$C = \begin{pmatrix} (t_0 - s_0)^{-1} & (t_0 - s_1)^{-1} & (t_0 - s_2)^{-1} \\ (t_1 - s_0)^{-1} & (t_1 - s_1)^{-1} & (t_1 - s_2)^{-1} \end{pmatrix}.$$

Cauchy matrices generalize *Hilbert matrices* $(C)_{i,j} = 1/(i + j + 1)$ and are sometimes called *generalized Hilbert matrices.* On the other hand, Cauchy matrices form a special subclass of *Loewner matrices* [41, 42], that is, matrices B such that $(B)_{i,j} = (u_i - v_j)/(s_i - t_j)$.

Both (post)multiplication of a Cauchy matrix by a vector and solution (for vector $\mathbf{x}$) of the equation $C\mathbf{x} = \mathbf{v}$, where C is a square and nonsingular Cauchy matrix and $\mathbf{v}$ is a vector, reduce to polynomial evaluation and interpolation and take $O((m + n)\log^2(m + n))$ ops, due to application of FFT. The algorithm of [82] approximates the product Cv at the cost $c_\epsilon(m+n)\log(m+n)$ ops where c_ϵ depends on the approximation error ϵ and the minimum distance $\min_{i,j} |t_i - s_j|$. Some typical applications of Cauchy matrices include the study of integral equations, conformal mappings, and singular integrals [67, 82] (see also "Correlation among Structured Matrices").

Circulant, Toeplitz, and Hankel Matrices

Toeplitz matrices and matrices closely related to them are among the most used structured matrices.

T is a *Toeplitz matrix* if $(T)_{i,j} = (T)_{i+k,j+k}$, for all positive k, that is, if all the entries of T are invariant in their shifts in the diagonal direction. T is completely defined by its first row and its first column. For example, below we display a 4×3 Toeplitz matrix specified by the coefficients of polynomial $v(x) = v_2x^2 + v_1x + v_0$ in such a way as to express the multiplication of polynomials $u(x) = u_2x^2 + u_1x + u_0$ and $v(x)$ as (post)multiplication of the matrix by the column vector of the coefficients of $u(x)$.

$$\begin{pmatrix} v_2 & 0 & 0 \\ v_1 & v_2 & 0 \\ v_0 & v_1 & v_2 \\ 0 & v_0 & v_1 \\ 0 & 0 & v_0 \end{pmatrix} \begin{pmatrix} u_2 \\ u_1 \\ u_0 \end{pmatrix} = \begin{pmatrix} v_2u_2 \\ v_1u_2 + v_2u_1 \\ v_0u_2 + v_1u_1 + v_2u_0 \\ v_0u_1 + v_1u_0 \\ v_0u_0 \end{pmatrix}. \tag{17.7}$$

In general, in this way, Toeplitz matrices of such a form express polynomial multiplication and, hence, vector convolution. Furthermore, a general Toeplitz matrix T can be embedded into a matrix of a such form as its middle block of rows. Then the product of T by a vector can be immediately extracted from the associated polynomial product (convolution).

H is a *Hankel matrix* if $(H)_{i,j} = (H)_{i-k,j+k}$, that is, if all the entries of H are invariant in their shifts in the antidiagonal direction. H is completely defined by its first row and its last column. For example, a 4×3 Hankel matrix is

$$H = \begin{pmatrix} v_0 & v_1 & v_2 & v_3 \\ v_1 & v_2 & v_3 & v_4 \\ v_2 & v_3 & v_4 & v_5 \end{pmatrix}.$$

The correlation of Hankel and Toeplitz matrices is formalized by introducing the *reversion matrix.*

$$J = J^{-1} = \begin{pmatrix} 0 & 0 & \cdots & 1 \\ \vdots & \vdots & \cdots & \vdots \\ 0 & 1 & \cdots & 0 \\ 1 & 0 & \cdots & 0 \end{pmatrix}. \tag{17.8}$$

Then TJ and JT are Hankel matrices for any Toeplitz matrix T, whereas HJ and JH are Toeplitz matrices for any Hankel matrix H. Consequently, computations with Hankel matrices can be reduced to computations with Toeplitz matrices, and vice versa. Hereafter, we will focus on the Toeplitz class.

$C_f = C_f(v)$, for a vector $v = [v_0, \ldots, v_{m-1}]^T$ and for a scalar f, is an *f-circulant* $m \times n$ matrix if $(C_f)_{i,j} = v_{i-j \bmod m}$ for $i \geq j$; $(C_f)_{i,j} = f v_{i-j \bmod m}$ for $i < j$. For example, for $m = n = 4$ we have

$$C_f(v) = \begin{pmatrix} v_0 & fv_3 & fv_2 & fv_1 \\ v_1 & v_0 & fv_3 & fv_2 \\ v_2 & v_1 & v_0 & fv_3 \\ v_3 & v_2 & v_1 & v_0 \end{pmatrix}.$$

$1-$circulant and $(-1)-$circulant matrices are called *circulant* and *anticirculant,* respectively. For any f, an f-circulant matrix is a special Toeplitz matrix, completely defined by its first column v and by the scalar f. It is possible to embed an $m \times n$ Toeplitz matrix into an $m \times (m + n - 1)$ circulant matrix. Therefore, certain operations on the former matrix, such as multiplication by a vector, can be reduced to multiplication of a circulant matrix by a vector.

Circulant matrix manipulation is fast due to FFT and the following well-known result:

THEOREM 17.1 [28, 32] *Let C_f be an $n \times n$ f-circulant matrix, with complex $f \neq 0$, and let c_f^T denote its first row. Let Ω be the $n \times n$ Fourier matrix, $(\Omega)_{i,j} = \omega^{ij}/\sqrt{n}$, $i, j = 0, 1, \ldots, n-1$, where ω is a*

primitive n-th root of unity, $(\Omega^H)_{i,j} = \omega^{-ij}/\sqrt{n}$, *and* $\Omega^H\Omega = I$. *Let* $D_f = diag(1, g, g^2, \ldots, g^{n-1})$, $g^n = f$, *and let D be another diagonal matrix with entries given by the vector* $\sqrt{n}\,\Omega D_f c_f$. *Then*

$$\Omega D_f C_f D_f^{-1} \Omega^H = D, \quad \textit{or, equivalently,} \quad C_f = D_f^{-1}\Omega^H D \Omega D_f \ .$$

This immediately implies that multiplication of an $n \times n$ f-circulant matrix by a vector reduces to vector convolution, which can be implemented by three DFTs on n elements and thus, has complexity $O(n \log n)$. Consequently, multiplication of an $m \times n$ Toeplitz matrix by a vector takes $O((m+n)\log(m+n))$ ops, and similarly for a Hankel matrix. Such an alternative reduction of Tv to convolution is slightly more effective because the total size of FFTs involved is smaller. Conversely, computing the DFT of an n-vector, for a prime n, reduces to multiplying a vector by an $(n-1) \times (n-1)$ circulant matrix [91].

For a nonsingular $n \times n$ Toeplitz matrix, the solution of a linear system $Tx = b$ has been extensively studied (see [12] and the references therein). Different algorithms have different advantages with respect to running time and numerical stability, the record time-complexity bound being $O(n \log^2 n)$ and based on reductions to FFT. If the field of constants does not support FFT, then the alternative construction of [21] gives us all stated time bounds multiplied by $O(\log\log(m+n))$.

Given a square Toeplitz matrix T, the problem of computing its determinant and the coefficients of its characteristic polynomial both reduce to a sequence of multiplications of this matrix T by vectors. This technique is effective for any matrix that can be multiplied by a vector at a low computational cost; it has been introduced in [90] and is based on the computation of the associated *Krylov sequence* (see Chapter 16 of this handbook). If the field of constants supports FFT, then the complexity of both operations is $O(n^2 \log n)$. To compute only the absolute value of the determinant, $O(n \log^2 n)$ ops suffice; moreover, if an $n \times n$ Toeplitz matrix is strongly nonsingular, its determinant is computed in $O(n \log^2 n)$ ops [12].

The inverse of a Toeplitz or Hankel matrix generally contains roughly $n^2/2$ distinct entries. There exist asymptotically optimal algorithms that compute the inverse in $O(n^2)$ ops, as well as slightly slower algorithms having better numerical stability (see [12] and the references therein). A major result in this area is the following theorem extending one of Gohberg and Semencul (see [12, 15, 51] for this fundamental result and its variants).

THEOREM 17.2 *Let T be a nonsingular* $n \times n$ *Toeplitz matrix,* $(T)_{i,j} = t_{i-j}$, $i, j = 0, 1, \ldots, n-1$; $t = [s, t_{1-n}, t_{2-n}, \ldots, t_{-1}]^T$, *for any fixed scalar s;* $x = [x_0, \ldots, x_{n-1}]^T = T^{-1}t$; $v = [-1, x_{n-1}, \ldots, x_1]^T$; $y = [y_0, \ldots, y_{n-1}]^T = T^{-1}[1, 0, \ldots, 0]^T$; $u = [0, y_{n-1}, \ldots, y_1]^T$. *For a given* $n \times 1$ *vector a, matrix* $L(a)$ *is an* $n \times n$ *lower triangular Toeplitz matrix whose first column is a. Then* $T^{-1} = L(x)L^T(u) - L(y)L^T(v)$.

In the special cases of a lower (upper) triangular Toeplitz or f-circulant matrix, the inverse is again a lower (upper) triangular Toeplitz or f-circulant matrix, respectively, and can be computed in $O(n \log n)$ ops.

Generally, the product of two Toeplitz matrices is not a Toeplitz matrix but is in the generalized class of Toeplitz-like matrices, to be introduced in "Correlations among Structured Matrices."

The classes of *block Toeplitz* and *block Hankel matrices* generalize Toeplitz and Hankel matrices, respectively. These are Toeplitz or Hankel matrices whose entries are matrices themselves. A $p \times q$ block Toeplitz (Hankel) matrix with blocks of size $r \times s$ can be turned into an $r \times s$ block matrix with $p \times q$ Toeplitz (Hankel) blocks and vice versa, by a sequence of row and column permutations.

Another class of structured matrices are *banded* matrices, where all nonzero entries are concentrated on a relatively small number of diagonals. Banded Toeplitz matrices are important in several applications [12, Sect. 2.11]. The sine transform enables us to simplify computations with such matrices. For instance, it is possible to embed a symmetric $(2k+1)$-diagonal $n \times n$ Toeplitz matrix T into an $(n+2\lfloor k/2 \rfloor) \times (n+2\lfloor k/2 \rfloor)$ matrix of the τ algebra, defined in "Sine and Cosine Transforms." Here is an example with $k = 2$ and $n = 4$:

Embedding the matrix

$$T = \begin{pmatrix} a_0 & a_1 & a_2 & 0 \\ a_1 & a_0 & a_1 & a_2 \\ a_2 & a_1 & a_0 & a_1 \\ 0 & a_2 & a_1 & a_0 \end{pmatrix} \quad \text{yields} \quad \begin{pmatrix} a_0 - a_2 & a_1 & a_2 & 0 & 0 & 0 \\ a_1 & a_0 & a_1 & a_2 & 0 & 0 \\ a_2 & a_1 & a_0 & a_1 & a_2 & 0 \\ 0 & a_2 & a_1 & a_0 & a_1 & a_2 \\ 0 & 0 & a_2 & a_1 & a_0 & a_1 \\ 0 & 0 & 0 & a_2 & a_1 & a_0 - a_2 \end{pmatrix} \in \tau \ . \tag{17.9}$$

This property allows us to reduce the solution of a band Toeplitz linear system to performing the sine transform and to solving a $k \times k$ linear system [9].

Block Toeplitz and block Hankel matrices, block matrices with Toeplitz and Hankel blocks, banded Toeplitz matrices, and several other classes of Toeplitz-like and Hankel-like matrices, including proper Toeplitz and f-circulant matrices, naturally arise in numerous applications, in particular, to control, signal processing, systems theory, solution of PDEs, and algebraic computations. A simple example is a Sylvester matrix S, which is a 2×1 block matrix with Toeplitz blocks. It can be multiplied with a vector in quasi-linear time. Hence, the Krylov sequence associated with S can be computed in quasi-quadratic time, which yields fast algorithms for determinant computation, inversion, etc. Similarly, the structure of various multivariate resultant matrices leads to an acceleration of their construction, of computing the resultant itself, and of approximating the solutions of a polynomial system of equations (see Chapter 16 of this handbook). On these and other computations with structured matrices, see [12, 23, 25, 37, 51, 65, 82] and references therein.

To conclude this section, we recall an algorithm of [20] for a linear system $V^T \mathbf{x} = \mathbf{w}$. An alternative and simpler approach would be to reduce this question to one on V by applying Tellegen's theorem; see [19, thm. 13.20]. Still, we recall below some interesting techniques from [20]. Matrix V^T is an $n \times n$ transposed Vandermonde matrix, such that $(V)_{i,j} = v_i^j$, for $i, j = 0, \ldots, n-1$, and $\mathbf{w}$ is a given column vector. The problem is reduced to computing the vector $\mathbf{w}^T V^{-1}$, $\mathbf{w}^T \left(V^T V\right)^{-1} V^T$. Observe that

$$\left(V^T V\right)_{i,j} = \sum_{k=0}^{n-1} v_k^{i+j}, \qquad i, j = 0, \ldots, n-1 \ ,$$

so that $V^T V$ is a Hankel matrix. Consider the values v_i as the roots of the (monic) polynomial $p(x) = \Sigma_{i=0}^{n-1} p_i x^i = \prod_{i=0}^{n-1} (x - v_k)$, $p_{n-1} = 1$, and compute all $2n-1$ distinct entries of $V^T V$, that is, $s_j = \Sigma_{i=0}^{n-1} v_i^j$, $j = 0, \ldots, 2n-2$, as the first $2n-1$ power sums of the roots $v_0, \ldots, v_{n-1}$.

Formally, we solve a linear system $V^T \mathbf{x} = \mathbf{w}$ as follows:

Algorithm 17.2

1. Compute the coefficients $p_0, \ldots, p_{n-2}$ of the above monic polynomial $p(x)$ by the fan-in method used in polynomial evaluation (cf. "Evaluation, Interpolation, and Chinese Remainder Computations"). This requires $O(n \log^2 n)$ ops. Write $p_{n-1} = 1$.

2. Compute the $s_0, \ldots, s_{2n-2}$ by solving in $O(n \log n)$ ops the lower triangular Toeplitz system of Newton's identities,

$$\begin{pmatrix} 1 & & & & & \\ p_{n-1} & \ddots & & & O & \\ \vdots & \ddots & \ddots & & & \\ p_0 & & \ddots & \ddots & & \\ & \ddots & & \ddots & \ddots & \\ O & & p_0 & \ldots & p_{n-1} & 1 \end{pmatrix} \begin{pmatrix} s_0 \\ \\ \vdots \\ \\ s_{2n-2} \end{pmatrix} = \begin{pmatrix} -p_{n-1} \\ -2p_{n-2} \\ \vdots \\ -np_0 \\ 0 \\ \vdots \\ 0 \end{pmatrix} ,$$

3. Compute the vector $\mathbf{y} = (V^T V)^{-1}\mathbf{w}$ *by solving the Hankel linear system* $(V^T V)\mathbf{y} = \mathbf{w}$. *This requires* $O(n \log^2 n)$ *ops.*

4. Compute and output the desired vector $\mathbf{x} = V\mathbf{y} = (V^T)^{-1}\mathbf{w}$. *Due to the results of "Vandermonde and Cauchy (generalized Hilbert) Matrices," this step is equivalent to polynomial evaluation and uses* $O(n \log^2 n)$ *ops.*

In summary, the multiplication of V^T by a vector takes $O(n \log^2 n)$ ops, and the same complexity bound holds for solving a linear system defined by the transpose of a Vandermonde matrix.

Algorithm 17.2 has been applied in [20] to multivariate polynomial multiplication (cf. "Multivariate Polynomials").

Bézout Matrices

Bézout matrices are structured matrices arising in several fundamental algebraic operations. In particular, they play a central role in studying the solutions of a system of two univariate polynomials and the GCD of two univariate polynomials, especially, when Boolean complexity and numerical stability issues become critical. Moreover, this study can be generalized to the solution of a system of several multivariate polynomial equations. This section introduces Bézout's matrices, demonstrates their structure and correlations to structured matrices studied above, and discusses some applications and computational complexity issues.

Consider two polynomials, $u(x) = \Sigma_{i=0}^{n} u_i x^i$ and $v(x) = \Sigma_{i=0}^{m} v_i x^i$, of degrees n and m, respectively, where $m \leq n$. The expression

$$\frac{u(z)v(w) - v(z)u(w)}{z - w} = \sum_{i,j=0}^{n-1} b_{i,j} z^i w^j \tag{17.10}$$

is easily verified to be a polynomial in w and z. This polynomial is called the generating function of the $n \times n$ *Bézout matrix* B or, simply, *Bezoutian* of $u(x)$ and $v(x)$, with $(B)_{i,j} = b_{i,j}$. The polynomial of (17.10) is sometimes also called the Bezoutian of $u(x)$ and $v(x)$.

By definition, $B(u, v) = -B(v, u)$. Observe, as particular cases, that

$$B(u, 1) = \begin{pmatrix} u_1 & \dots & u_n \\ \vdots & \ddots & \\ u_n & & O \end{pmatrix}, \qquad B\left(u, x^n\right) = -\begin{pmatrix} O & & u_0 \\ & \ddots & \vdots \\ u_0 & \dots & u_{n-1} \end{pmatrix}$$

are triangular Hankel matrices. It has been proven that $B(u, v)$, for general $u(x)$ and $v(x)$, can be written in terms of matrices of this form (see, for instance, [63]):

$$B(u, v) = B(v, 1)JB\left(u, x^n\right) - B(u, 1)JB\left(v, x^n\right), \tag{17.11}$$

where J is the reversion matrix of (17.8).

There is a deeper connection of Bezoutians to Hankel matrices [12, Sect. 2.9]. Given univariate polynomials $u(x)$, $v(x)$, of respective degrees n, m, where $n > m$, consider the infinite sequence of values $h_0, h_1, \dots$, in the coefficient field of $u(x)$, $v(x)$, known as the *Markov parameters* and defined as follows:

$$\frac{v\left(x^{-1}\right)}{xu\left(x^{-1}\right)} = \sum_{i=0}^{\infty} h_i x^i \quad \text{or, equivalently,} \quad \frac{v(x)}{u(x)} = \sum_{i=0}^{\infty} h_i x^{-i-1}.$$

The Markov parameters generate an $n \times n$ Hankel matrix $H(u, v)$ such that $(H)_{i,j} = h_{i+j}$ for $i, j = 0, 1, \dots, n-1$. Given $u(x)$, $v(x)$, the entries of $H(u, v)$ are computed by solving a lower triangular Toeplitz system of $2n-1$ equations in $O(n \log n)$ ops. These equations are obtained by truncating the first formal power series above and expressing polynomial multiplication by a Toeplitz matrix as in Eq. (17.7).

If $u(x)$ and $v(x)$ are relatively prime, then $H(u, v)$ is nonsingular. Moreover, for any nonsingular $n \times n$ Hankel matrix H, there exists a pair of relatively prime polynomials $u(x)$, $v(x)$, of degrees n and $m < n$, respectively, such that $u(x)$ is monic and $H = H(u, v)$. Matrix $H(u, v)$, for polynomials $u(x)$ and $v(x)$ whose degrees m and n satisfy $n > m$, is related to the Bezoutian $B(u, v)$ of the same polynomials by

$$B(u, v) = B(u, 1)H(u, v)B(u, 1) \quad \text{and} \quad B(u, w)H(u, v) = I_n ,$$

where $w(x)$ is a univariate polynomial of degree less than n and such that $w(x)v(x) = 1 \bmod u(x)$ and where I_n is the $n \times n$ identity matrix. The second expression states, in words, that the inverse of any nonsingular Hankel matrix is a Bezoutian. This expression, together with Eq. (17.11), extends the Gohberg–Semencul formula and Theorem 17.2 to Hankel matrices.

An important application of these results is to compute the polynomial GCD [12]. Assume that $u(x)$, $v(x)$ are monic and their degrees satisfy $m < n$. Let $k \geq 1$ be such that the degree of $\gcd(u(x), v(x))$ in x is $n - k$. Then $\operatorname{rank}(H(u, v)) = k$, $\det H_k \neq 0$, where H_k is the $k \times k$ leading principal submatrix of $H(u, v)$. Moreover, if $H_{k+1}\mathbf{w} = 0$, $\mathbf{w} = [w_0, \ldots, w_k]$, $w_k = 1$, then

$$u(x) = \gcd(u(x), v(x)) \sum_{i=0}^{k} w_i x^i .$$

This result yields an algorithm for the GCD using $O(n^2 \log n)$ ops and supporting the record parallel complexity. In addition, this algorithm is quite stable numerically (since the Bezoutian entries have bounded moduli) and supports the record Boolean complexity estimates for computing the GCD of a pair of polynomials defined over the integers or rationals.

Bezoutians have been studied in classical elimination theory (starting with univariate polynomials and then extended to the multivariate case), in stability theory, as well as in computing the reduction of a general matrix to the tridiagonal form. The reader may consult [12, ch. 2], for a comprehensive introduction, and [22, 41, 52, 63] for further information.

Correlations among Structured Matrices

In this section we extend effective algorithms for Toeplitz matrix computations to computations with more general classes of dense structured matrices. In this way, we will unify efficient computations with various dense structured matrices. This approach has further extensions to computations with polynomials, rational functions, and general matrices. In particular, we apply certain shifts and scaling operators to the matrices in order to represent in a simpler and more convenient form.

The main tool are linear operators F that map an $m \times n$ structured matrix A into the product of two matrices $F(A) = GH^T$, where G is $m \times d$ and H is $n \times d$ matrices for a smaller d. Then $r = rank(F(A))$, the rank of the matrix $F(A)$, is called the *F-rank* of A, and the pair of the matrices G and H is called an *F-generator* of A of *length* d.

For the structured matrices seen so far, there exist linear operators F such that the F-generators of A have constant length. Moreover, A can be efficiently expressed via the generators. Thus, an $m \times n$ matrix $F(A)$ with its generator (G, H) of length r can be represented by using only $(m + n)r$, rather than mn, words of storage space. Furthermore, we shall see that to compute the vector $A\mathbf{v}$, for a given pair of matrices G, H, satisfying $F(A) = GH^T$, and for a vector $\mathbf{v}$, we only need $O((m + n)r \log^h(m + n))$ ops, for $h = 1$ or $h = 2$, rather than $(2n - 1)m$ ops. Now, for a given dense structured matrix A, we look for an operator F that transforms A into a low rank matrix, from which we could easily recover the original matrix. Then we may take all the advantages of operating with low rank matrices, even though the structured input matrix may have the full rank. Below we provide some further specifications; more details are found in [12, 47, 55], including a discussion on the powerful theorems that relate the F-rank of a nonsingular matrix and that of its inverse.

Let $Z = Z_k$ be the following $k \times k$ square *down-shift matrix,*

$$Z = \begin{pmatrix} 0 & \dots & & \dots & 0 \\ 1 & 0 & & & \vdots \\ 0 & \ddots & \ddots & & \\ \vdots & & \ddots & \ddots & \vdots \\ 0 & \dots & \dots & 1 & 0 \end{pmatrix}.$$

We generalize the class of Toeplitz matrices to the class of *Toeplitz-like* $m \times n$ matrices A, whose F-rank is bounded from above by a fixed constant independent of m and n, where F is one of the two following operators F_+ or F_-:

$$F_+(A) = F_{Z,Z^T}(A) = A - Z_m A Z_n^T, \quad F_-(A) = F_{Z^T,Z}(A) = A - Z_m^T A Z_n .$$

$F(A) = A - PAQ$ and $F(A) = PA - QA$, for a pair of fixed matrices P, Q, are typical forms of linear operators defining various classes of structured matrices.

Due to the shifting properties of multiplications by Z and Z^T, the operators F_+, F_- turn into zero all the entries of a Toeplitz matrix A, except for the first row and column under F_+ and for the last row and column under F_-, which are invariant in the transition from A to $F(A)$. The above F-generators, for $F = F_+$ and $F = F_-$, have lengths at most 2 for Toeplitz matrices. The length only increases to $p + q$ for $p \times q$ block matrices with Toeplitz blocks. For these operators, it is possible to recover a matrix A from $F(A)$. Therefore, in the Toeplitz case, we may shift to operating with $F(A)$, and thus, save computational resources of time and space. For instance, we may compute the product Av for arbitrary vector v in time $O(nr \log n)$, where r is the rank of $F(A)$, which is typically much smaller than n.

For a concrete illustration, take matrix $A = T$ of Example (17.9). Then we have

$$F_+(A) = \begin{pmatrix} a_0 & a_1 & a_2 & 0 \\ a_1 & 0 & 0 & 0 \\ a_2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} = \begin{pmatrix} a_0 & 1 \\ a_1 & 0 \\ a_2 & 0 \\ 0 & 0 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & a_1 & a_2 & 0 \end{pmatrix},$$

where the rank and length of $F(A)$ is 2. In the above notation $F_+(A) = GH$, and let g_i, h_i be the respective column vectors of G, H, for $i = 1, 2$. Then, $F_+(A) = \Sigma_{i=1}^2 g_i h_i^T$. The important property is that A can be expressed as a sum of d products of triangular Toeplitz matrices. Here, $A = \Sigma_{i=1}^2 L(g_i) L^T(h_i)$, where for any $1 \times n$ vector v, $L(v)$ is a lower triangular $n \times n$ Toeplitz matrix, with v being its first column. This kind of representation using $F_+(A)$, $F_-(A)$ is possible for any matrix A. A small number of products $L(g_i)L^T(h_i)$ in the sum characterizes matrices with Toeplitz-like structure, and then we yield efficient ways for several basic operations, including vector multiplication and the recovery of A from its F-image, for the various F such as $F = F_+$ and $F = F_-$.

Analogously, A is said to be *Hankel-like* if the F-rank does not exceed a fixed constant under one of the following two operators:

$$F_{Z,Z}(A) = A - Z_m A Z_n, \quad F_{Z^T,Z^T}(A) = A - Z_m^T A Z_n^T ,$$

and the stated properties of Toeplitz-like matrices can be extended.

A further generalization is to Toeplitz-like plus Hankel-like matrices, expressed as the sum of a Toeplitz-like matrix and a Hankel-like matrix. This class includes both Toeplitz-like and Hankel-like matrices. Suppose that such a pair of $n \times n$ matrices is given by their F-generators with a constant F-length. The F-generator of their product can be computed in $O(n \log n)$ ops by using FFT. At the computational cost $O(n^2 \log n)$, we can compute the inverse and the characteristic polynomial and solve a linear system $Tx = b$ of an $n \times n$ Toeplitz-like plus Hankel-like matrix T. This complexity bound is record and

nearly optimal (up to a logarithmic factor) for the inverse and characteristic polynomial, even in the class of Toeplitz matrices, and the approach yields the record parallel complexity bounds also for solving a Toeplitz-like plus Hankel-like linear system of equations [11, 12, 69]. Sequential algorithms for Toeplitz-like plus Hankel-like linear systems cost $O(n \log^2 n)$ ops (see [12]). There are also several effective iterative algorithms for Toeplitz and other structured linear systems (see [70, 71, 72, 78, 80]), and the reader is also referred to [11, 12, 69, 72] on some other basic properties of Toeplitz-like and Toeplitz-like plus Hankel-like matrices.

Let $D(\mathbf{v}) = D_n(\mathbf{v}) = diag(v_0, v_1, \ldots, v_{n-1})$ where $\mathbf{v}$ is the vector $[v_0, v_1, \ldots, v_{n-1}]^T$ and let us extend the class of Cauchy (or generalized Hilbert) matrices as follows: an $m \times n$ matrix C is said to be *Cauchy (Hilbert)-like* if the F-rank of C is bounded by a constant independent of m and n, F is the operator

$$F_{\mathbf{s},\mathbf{t}}(C) = D_m(\mathbf{s})C - CD_n(\mathbf{t}) ,$$

and $\mathbf{s}$ and $\mathbf{t}$ denote a pair of m- and n-dimensional vectors, respectively.

By definition, a Loewner matrix is Cauchy (Hilbert)-like.

An operator of the Cauchy type is represented by a nonsingular matrix if and only if every entry of $\mathbf{s}$ is distinct from all entries of $\mathbf{t}$. For the Cauchy matrix C with $(C)_{i,j} = \frac{1}{s_i - t_j}$, this operator gives us $F_{\mathbf{s},\mathbf{t}}(C) = \mathbf{e}\mathbf{e}^T$, where $\mathbf{e} = [1, \ldots, 1]^T$. We refer the reader to [12] on further study of the matrices of this class.

The class of $m \times n$ *Vandermonde-like* matrices is formed by the matrices having a constant F-rank, where F is defined by any of the following matrix equations:

$$\begin{aligned} F_{\mathbf{v},Z}(V) &= D_m(\mathbf{v})V - VZ_n, \quad F_{\mathbf{v},Z^T}(V) = D_m(\mathbf{v})V - VZ_n^T , \\ F_{Z,\mathbf{v}}(V) &= VD_n(\mathbf{v}) - Z_mV, \quad F_{Z^T,\mathbf{v}}(V) = VD_n(\mathbf{v}) - Z_m^TV . \end{aligned}$$

If $m = n$ and V is defined by an n-dimensional vector $\mathbf{v}$, then

$$F_{\mathbf{v},Z}(V) = \begin{pmatrix} 0 & \cdots & 0 & v_0^n \\ \vdots & & \vdots & \vdots \\ 0 & \cdots & 0 & v_{n-1}^n \end{pmatrix} = D^n(\mathbf{v})\,\mathbf{e}\,[0, \ldots, 0, 1] .$$

Both Cauchy-like and Vandermonde-like matrices can be recovered from their respective images under the operators specified here.

We conclude this section by following [68] in order to demonstrate how associating the operators unifies the treatment of dense structured matrices, revealing some important but hidden correlations among various classes of such matrices (such correlations have been already demonstrated when we solved a transposed Vandermonde linear system in "Circulant, Toeplitz, and Hankel Matrices"). The first basic idea is that several operations can be applied to the generators without explicitly constructing the matrices. Addition and subtraction, for the operators examined, satisfy

$$F(A \pm B) = F(A) \pm F(B) ,$$

and the F-generator of the sum or difference is the union of the other two generators.

For multiplication, we may rely on a result of [68], generalizing a result of [26], which allows us to compute a generator of the product in terms of the generators of the two given matrices. Let K, L, M, and N be four fixed matrices, $\Delta = LM - I$, I be the identity matrix, and let $F_{(P,Q)}$ be an operator defined by the equation $F_{(P,Q)}(A) = A - PAQ$, for any matrices P, Q, and A. Then

$$F_{(K,N)}(AB) = F_{(K,L)}(A)B + KALF_{(M,N)}(B) + KA(LM - I)BN .$$

Let r=rank $F_{(K,N)}(AB)$, r_1 = rank $F_{(K,L)}(A)$, and r_2 = rank $F_{(M,N)}(B)$. It follows that $r \leq$ rank $(LM - I) + r_1 + r_2$.

Based on this result, [68] extends the algorithms for structured matrices seen so far to matrices with similar structures. For instance, specialize L and M as follows:

$$L = Z,\ M = Z^T, \quad \text{or} \quad L = Z^T,\ M = Z\,,$$

where Z is the down-shift matrix defined above. If we require all three operators be of Hankel and/or Toeplitz type, then K and N are implicitly defined. If all matrices are of size $n \times n$ and the length of F-generator of A, B is d_1, d_2, respectively, then an F-generator for AB of a length at most $d_1 + d_2 + 1$ can be computed in $O(n(d_1 + d_2)^2 \log(n(d_1 + d_2)))$ ops. Moreover, the rank of $LM - I$ for both specializations is 1, so the F-rank of the product is bounded by $1 + r_1 + r_2$. This implies that the product of two Toeplitz- or Hankel-like matrices is again Toeplitz- or Hankel-like, with F-rank bounded as in the second to last line of the following table from [68] and [12, Sect. 2.12].

Different specializations of matrices K, L, M, and N imply the rest of the entries in the table. We write HT, C, and V, for the classes of Hankel-like plus Toeplitz-like, Cauchy-like, and Vandermonde-like matrices, respectively.

A	B	AB	F-rank of AB
HT	V	V	$r \le r_1 + r_2 + 1$
V	V	C	$r \le r_1 + r_2 + 1$
V	V	HT	$r \le r_1 + r_2$
C	V	V	$r \le r_1 + r_2$
HT	HT	HT	$r \le r_1 + r_2 + 1$
C	C	C	$r \le r_1 + r_2 + 1$

By using these transitions, we may reduce the original computational problem to other problems for which we have effective algorithms. For instance, suppose that, given an $F_{\mathbf{s},\mathbf{t}}$-generator for a Cauchy-like matrix A, with $\mathbf{s},\mathbf{t}$ vectors, we wish to compute its determinant and its inverse, when the latter exists. We may choose B to be the Vandermonde matrix defined by the inverse entries of $\mathbf{t}$ and compute an F-generator for AB. Then the matrix AB is Vandermonde-like, with rank bounded as in the fourth line of the table above. Now, $\det A = \det(AB)/\det B$ and $A^{-1} = B(AB)^{-1}$, so we have reduced the original problems to a sequence of operations on Vandermonde-like matrices.

After the transformations displayed in the above table have been established in [68], some of them have been further simplified, due to the following result of [46]:

THEOREM 17.3 *$C = \Omega T D_{2n} \Omega^{-1}$ is a Cauchy-like matrix, having a r-generator $G^* = \Omega G$, $H^* = \Omega D(w) H$, for $F = F_{D_n, D_{-n}} = F(D_n, D_{-n})$, that is,*

$$F_{D_n, D_{-n}} C = D_n C - C D_{-n} = G^* \left(H^*\right)^T\,,$$

provided that T is a Toeplitz-like matrix with an F-generator G, H for $F = F_{Z_1, Z_{-1}} = F(Z_1, Z_{-1})$,

$$F_{Z_1, Z_{-1}} T = Z_1 T - T Z_{-1} = G H^T\,,$$

$D_k = diag(1, w_k, w_k^2, \ldots, w_k^{n-1})$, $w_k = exp(2\pi\sqrt{-1}/k)$, and $\Omega = 1/\sqrt{n}(w_n^{ij})$ is a Fourier matrix, $\Omega^{-1} = 1/\sqrt{n}(w_n^{-ij})$, $i, j = 0, 1, \ldots, n-1$.

Due to this result, the transformation from any Toeplitz-like matrix into a Cauchy-like matrix is immediately reduced to FFT, which leads to effective practical algorithms for Toeplitz-like linear systems [46]. Another example of applications of this kind is the fast algorithms for polynomial evaluation and interpolation based on manipulation with structured matrices of the classes HT, V, and C [77, 79].

Additional bibliography on the material of this section can be found in [12].

17.5 Research Issues and Summary

We have reviewed the known highly effective algorithms for the discrete Fourier transform (DFT) and its inverse, as well as for some related transforms, all of them based on the celebrated fast Fourier transform (FFT) algorithm. We have shown immediate application of FFT to computing convolution of vectors (polynomial products), which is a fundamental operation of computer algebra and signal processing. We have also demonstrated further applications to other basic operations with integers, univariate and multivariate polynomials and power series, as well as to the fundamental computations with circulant, Toeplitz, Hankel, Vandermonde, Cauchy (generalized Hilbert), and Bézout matrices, as well as to other structured matrices related to the above classes of matrices via the associated linear operators. We exemplified some major techniques establishing such relations and other major basic techniques for extending the power of FFT to numerous other computational problems, and we supplied pointers to further bibliography. Some of these techniques are quite recent, and further research is very promising, in particular, via the study of structured matrices and polynomial systems of equations [20, 37, 65]. New practical and theoretical research directions have emerged recently related to the numerical implementation of FFT, leading to interesting algebraic techniques [18] and certain finite group applications [29, 38].

17.6 Defining Terms

Chinese remainder algorithm: The algorithm recovering a unique integer (or polynomial) p modulo M from the h residues of p modulo pairwise relatively prime integers (or, respectively, polynomials) $m_1, \cdots, m_h$, where M is the product $m_1 \cdots m_h$. (The **residue** of p modulo m is the remainder of the division of p by m.)

Convolution of two vectors: A vector that contains the coefficients of the product of two polynomials whose coefficients make up the given vectors. The positive and negative wrapped convolutions are the coefficient vectors of the two polynomials obtained via reduction of the polynomial product by $x^n + 1$ and $x^n - 1$, respectively.

Determinant (of an $n \times n$ matrix): A polynomial of degree n in the entries of the matrix with the property of being invariant in the elementary transformations of a matrix used in Gaussian elimination; the determinant of the product of matrices is the product of their determinants; the determinant of a triangular matrix is the product of its diagonal entries; the determinant of any matrix is nonzero if and only if the matrix is invertible (nonsingular).

Discrete Fourier transform (DFT): The vector of the values of a given polynomial at the set of all the Kth roots of unity. **The inverse discrete Fourier transform (IDFT)** of a vector **v:** The vector of the coefficients of a polynomial whose values at the Kth roots of 1 form a given vector **v.**

Divide-and-conquer: A general algorithmic method of dividing a given problem into two (or more) subproblems of smaller sizes that are easier to solve, then synthesizing the overall solution from the solutions to the subproblems.

Fast Fourier transform (FFT): An algorithm that uses $1.5K \log K$ ops in order to compute the DFT at the Kth roots of 1, where $K = 2^k$, for a natural k. It also computes the IDFT in $K + 1.5K \log K$ ops.

Greatest common divisor (GCD) of 2 or several integers (or polynomials): The largest positive integer (or a polynomial of the largest degree) that divides both or all of the input integers (or polynomials).

Interpolation: The computation of the coefficients of a polynomial in one or more variables, given its values at a set of points. The inverse problem is the **evaluation** of a given polynomial **on a set of points.**

Least common multiple (LCM) of 2 or several integers (or polynomials): The smallest positive integer (or a polynomial of the smallest degree) divisible by both or by all of the input integers (or polynomials).

Ops: Arithmetic operations, i.e., additions, subtractions, multiplications, or divisions.

Structured matrix: A matrix whose each entry can be derived by a formula depending on a few parameters. For instance, the Hilbert matrix has $\frac{1}{i+j-1}$ as the entry in row i and column j.

Taylor shift of the variable: Recovery of the coefficient vector of a polynomial after a linear substitution of its variable, $y = x + \Delta$ for x, where Δ is a fixed shift value.

References

[1] Aho, A.V., Hopcroft, J.E., and Ullman, J.D., *The Design and Analysis of Computer Algorithms.* Addison-Wesley, Reading, MA, 1974.

[2] Aho, A., Steiglitz, K., and Ullman, J., Evaluating polynomials at fixed set of points. *SIAM J. Comput.,* 4, 533–539, 1975.

[3] Bailey, D.H., A high-performance FFT algorithm for vector supercomputers. *ACM Trans. on Math. Soft.,* 19(3), 288–319, 1993.

[4] Bailey, D.H., Multiprecision translation and execution of FORTRAN programs. *ACM Trans. on Math. Soft.,* 19(3), 288–319, 1993b.

[5] Ben-Or, M. and Tiwari, P., A deterministic algorithm for sparse multivariate polynomial interpolation. In *Proc. ACM Symp. Theory of Computing,* 301–309, ACM Press, New York, 1988.

[6] Berlekamp, E.R., *Algebraic Coding Theory.* McGraw-Hill, New York, 1968.

[7] Biehl, I., Buchmann, J., and Papanikolaou, T., LiDIA: A library for computational number theory. Technical Report SFB 124–C1, Universität des Saarlandes, Saarbrücken 66041 Germany, 1995. `http://www-jb.cs.uni-sb.de/linktop/LiDIA`.

[8] Bini, D. and Bozzo, E., Fast discrete transform by means of eigenpolynomials, *Comp. & Math. (with Appl.),* 26(9), 35–52, 1993.

[9] Bini, D. and Capovani, M., Spectral and computational properties of band symmetric Toeplitz matrices. *Linear Algebra Appl.,* 52/53, 99–126, 1983.

[10] Bini, D. and Favati, P., On a matrix algebra related to the Hartley transform. *SIAM J. Matrix Anal. Appl.,* 14(2), 500–507, 1993.

[11] Bini, D. and Pan, V.Y., Improved parallel computation with Toeplitz-like and Hankel-like matrices. *Linear Algebra Appl.,* 188/189, 3–29, 1993.

[12] Bini, D. and Pan, V.Y., *Polynomial and Matrix Computations,* volume 1: Fundamental Algorithms. Birkhäuser, Boston, 1994.

[13] Blahut, R.E., *Fast Algorithms for Digital Signal Processing,* Addison-Wesley, New York, 1984.

[14] Borodin, A. and Munro, I., *The Computational Complexity of Algebraic and Numeric Problems.* American Elsevier, New York, 1975.

[15] Brent, R.P., Gustavson, F.G., and Yun, D.Y.Y., Fast solution of Toeplitz systems of equations and computation of Padé approximations. *J. Algorithms,* 1, 259–295, 1980.

[16] Brigham, E.O., *The Fast Fourier Transform,* Prentice-Hall, Englewood Cliffs, NJ, 1974.

[17] Buchberger, B., Collins, G.E., and Loos, R., Eds., *Computer Algebra: Symbolic and Algebraic Computation,* volume 4 of *Computing Supplementum,* 2nd ed., Springer-Verlag, Wien, 1982.

[18] Buhler, J., Shokrollahi, M.A., and Stemann, V., Fast and precise computations of discrete Fourier transforms using cyclotomic integers. In *Proc. ACM Symp. Theory of Comp.,* 40–47. ACM Press, 1997.

[19] Bürgisser, P., Clausen, M., and Shokrollahi, M.A., *Algebraic Complexity Theory.* Springer-Verlag, Berlin, 1997.

[20] Canny, J., Kaltofen, E., and Lakshman, Y., Solving systems of non-linear polynomial equations faster. In *Proc. ACM Intern. Symp. Symbolic Algebraic Comput. (ISSAC '89),* 121–128. ACM Press, New York, 1989.
[21] Cantor, D.G. and Kaltofen, E., On Fast Multiplication of Polynomials over Arbitrary Rings, *Acta Informatica,* 28(7), 697–701, 1991.
[22] Cardinal, J.-P. and Mourrain, B., Algebraic approach of residues and applications. In *The Mathematics of Numerical Analysis,* Renegar, J., Shub, M., and Smale, S., Eds., volume 32 of *Lectures in Applied Math.,* 189–210. AMS, 1996.
[23] Chan, T.F., An optimal circulant preconditioner for Toeplitz systems. *SIAM J. Sci. Stat. Comput.,* 9, 766–771, 1988.
[24] Chan, R., Scientific applications of iterative Toeplitz solvers. *Calcolo,* 33, 249–267, 1996.
[25] Chan, R.H. and Strang, G., Toeplitz equations by conjugate gradients with circulant preconditioner. *SIAM J. Sci. Stat. Comput.,* 10, 104–119, 1989.
[26] Chun, J., Kailath, T., and Lev-Ari, H., Fast parallel algorithm for QR-factorization of structured matrices. *SIAM J. Sci. Stat. Comput.,* 8(6), 899–913, 1987.
[27] Clausen, M., Fast generalized FFT. *Theor. Computer Science,* 56, 55–63, 1989.
[28] Cline, R.E., Plemmons, R.J., and Worm, G., Generalized inverses of certain Toeplitz matrices. *Linear Algebra Appl.,* 8, 25–33, 1974.
[29] Cole, R. and Hariharan, R., An $O(n \log n)$ algorithm for the maximum agreement subtree problem for binary trees. In *Proc. 7th ACM-SIAM Symp. Discrete Algorithms,* 323–332, 1996.
[30] Conte, C.D. and de Boor, C., *Elementary Numerical Analysis: an Algorithmic Approach.* McGraw-Hill, New York, 1980.
[31] Cooley, J.W. and Tukey, J.W., An algorithm for the machine calculation of complex Fourier series. *Math. of Comp.,* 19(90), 297–301, 1965.
[32] Davis, P., *Circulant Matrices.* John Wiley & Sons, New York, 1974.
[33] Díaz, A. and Kaltofen, E., On computing greatest common divisors with polynomials given by black boxes for their evaluation. In *Proc. ACM Intern. Symp. Symbolic Algebraic Comput. (ISSAC '95),* Levelt, A., Ed., 232–239, ACM Press, New York, 1995.
[34] Duhamel, P. and Vetterli, M., Fast Fourier transforms: a tutorial review. *Signal Processing,* 19, 259–299, 1990.
[35] Elliott, D.F. and Rao, K.R., *Fast Transform Algorithms, Analyses, and Applications.* Academic Press, New York, 1982.
[36] Emiris, I.Z., A complete implementation for computing general dimensional convex hulls. *Intern. J. Computational Geometry & Applications, Special Issue on Geometric Software,* 8(2), 1998. A preliminary version as Tech. Report 2551, INRIA Sophia-Antipolis, France, 1995, 1997.
[37] Emiris, I.Z. and Pan, V.Y., The structure of sparse resultant matrices. In *Proc. ACM Int. Symp. on Symb. Alg. Comp. (ISSAC '97),* 189–196. ACM Press, New York, 1997.
[38] Farach, M., Przytycka, T.M., and Thorup, M., Computing the agreement of trees with bounded degrees. In *Proc. 3rd Annual European Symp. on Algorithms,* Spirakis, P., Ed., volume 979 of *Lect. Notes in Comp. Science,* 381–393. Springer-Verlag, New York, 1995.
[39] Fateman, R.J., Polynomial multiplication, powers and asymptotic analysis: Some comments. *SIAM J. Comp.,* 3(3), 196–213, 1974.
[40] Fiduccia, C.M., A rational view of the fast Fourier transform. In *Proc. 25th Allerton Conf. Commun., Control and Computing,* 1987.
[41] Fiedler, M., Hankel and Loewner matrices. *Linear Algebra Appl.,* 58, 75–95, 1984.
[42] Fiedler, M. and Ptak, V., Loewner and Bezout matrices. *Linear Algebra Appl.,* 101, 187–220, 1988.
[43] Frigo, M. and Johnson, S.J., The fastest Fourier transform in the west. Available at `http://theory.lcs.mit.edu/~fftw`.

[44] Geddes, K.O., Czapor, S.R., and Labahn, G., *Algorithms for Computer Algebra.* Kluwer Academic Publishers, Norwell, MA, 1992.
[45] Free Software Foundation., GNU multiple precision library, 1996. `ftp://prep.ai.mit.edu/pub/gnu/gmp-M.N.tar.gz`.
[46] Gohberg, I., Kailath, T., and Olshevsky, V., Fast Gaussian elimination with partial pivoting for matrices with displacement structure. *Math. of Comp.,* 64(212), 1557–1576, 1995.
[47] Gohberg, I. and Olshevsky, V., Complexity of multiplication with vectors for structured matrices. *Linear Algebra Appl.,* 202, 163–192, 1994.
[48] Golub, G.H. and Van Loan, C.F., *Matrix Computations,* 3rd ed. The Johns Hopkins University Press, Baltimore, MD, 1996.
[49] Grigoriev, D.Y., Karpinski, M., and Singer, M.F., Fast parallel algorithms for sparse multivariate polynomial interpolation over finite fields. *SIAM J. Computing,* 19(6), 1059–1063, 1990.
[50] Hardy, G.H. and Wright, E.M., *An Introduction to the Theory of Numbers,* 5th ed. Clarendon Press, Oxford, 1979.
[51] Heinig, G. and Rost, K., *Algebraic Methods for Toeplitz-like Matrices and Operators,* volume 13 of *Operator Theory.* Birkhäuser, 1984.
[52] Householder, A.S., *The Numerical Treatment of a Single Nonlinear Equation.* McGraw-Hill, Boston, 1970.
[53] Huckle, T., Iterative methods for ill-conditioned Toeplitz matrices. *Calcolo,* 33, 1996.
[54] Ja Ja, J., *An Introduction to Parallel Algorithms.* Addison-Wesley, MA, 1992.
[55] Kailath, T., Kung, S.-Y., and Morf, M., Displacement ranks of matrices and linear equations. *J. Math. Anal. Appl.,* 68(2), 395–407, 1979.
[56] Kailath, T. and Olshevsky, V., Displacement structure approach to discrete-trigonometric transform based preconditioners of G. Strang type and of T. Chan type. *Calcolo,* 33, 1996.
[57] Kaltofen, E., Factorization of polynomials given by straight-line programs. In *Randomness and Computation,* Micali, S., Ed., volume 5 of *Advances in Computing Research,* 375–412. JAI Press, Greenwich, CT, 1989.
[58] Kaltofen, E. and Lakshman, Y., Improved sparse multivariate polynomial interpolation algorithms. In *Proc. ACM Intern. Symp. Symbolic Algebraic Comput. (ISSAC '88),* volume 358 of *Lect. Notes in Comp. Science,* 467–474. Springer-Verlag, 1988.
[59] Kaltofen, E., Lakshman, Y.N., and Wiley, J.M., Modular rational sparse multivariate polynomial interpolation. In *Proc. ACM Intern. Symp. Symb. Algebr. Comput. (ISSAC '90),* 135–139. ACM Press, New York, 1990.
[60] Kaltofen, E. and Pan, V.Y., Processor efficient parallel solution of linear systems over an abstract field. In *Proc. 3rd Ann. ACM Symp. on Parallel Algorithms and Architectures,* 180–191. ACM Press, New York, 1991.
[61] Karatsuba, A. and Ofman, Y., Multiplication of multidigit numbers on automata. *Soviet Physics Dokl.,* 7, 595–596, 1963.
[62] Knuth, D.E., *The Art of Computer Programming: Seminumerical Algorithms,* volume 2. Addison-Wesley, Reading, MA, 1997.
[63] Lancaster, P. and Tismenetsky, M., *The Theory of Matrices.* Academic Press, 1985.
[64] Manocha, D. and Canny, J., Multipolynomial resultant algorithms. *J. Symbolic Computation,* 15(2), 99–122, 1993.
[65] Mourrain, B. and Pan, V.Y., Solving special polynomial systems by using structured matrices and algebraic residues. In *Proc. Workshop on Foundations of Computational Mathematics,* Cucker, F. and Shub, M., Eds., 287–304. Springer-Verlag, Berlin, 1997.
[66] Neuts, M.F., *Structured Stochastic Matrices of M/G/1 Type and Their Applications.* Marcel Dekker, New York, 1989.
[67] O'Donnell, S.T. and Rokhlin, V., A fast algorithm for numerical evaluation of conformal mappings. *SIAM J. Sci. Stat. Comput.,* 10(3), 475–487, 1989.

[68] Pan, V.Y., Computations with dense structured matrices. *Math. of Comp.,* 55(191), 179–190, 1990.
[69] Pan, V.Y., Parametrization of Newton's iteration for computations with structured matrices and applications. *Comp. & Math. (with Appl.),* 24(3), 61–75, 1992.
[70] Pan, V.Y., Complexity of computations with matrices and polynomials, *SIAM Review,* 34(2), 225–262, 1992.
[71] Pan, V.Y., Parallel solution of Toeplitz-like linear systems. *J. of Complexity,* 8, 1–21, 1992b.
[72] Pan, V.Y., Concurrent iterative algorithm for Toeplitz-like linear systems, *IEEE Trans. on Parallel and Distributed Systems,* 4(5), 592–600, 1993.
[73] Pan, V.Y., Simple multivariate polynomial multiplication. *J. Symb. Comp.,* 18, 183–186, 1994.
[74] Pan, V.Y., Approximate polynomial GCDs, Padé approximation, polynomial zeros and bipartite graphs, *Proc. 9th Ann. ACM–SIAM Symp. on Discrete Algorithms (SODA '98),* 68-77, 1998.
[75] Pan, V.Y., Faster solution of the key equation for decoding the BCH error-correcting codes. In *Proc. ACM Symp. Theory of Comp.,* 168–175. ACM Press, 1997.
[76] Pan, V.Y., Landowne, E., and Sadikou, A., Univariate polynomial division with a remainder by means of evaluation and interpolation. *Information Process. Letters,* 44, 149–153, 1992.
[77] Pan, V.Y., Sadikou, A., Landowne, E., and Tiga, O., A new approach to fast polynomial interpolation and multipoint evaluation. *Comp. & Math. (with Appl.),* 25(9), 25–30, 1993.
[78] Pan, V.Y., Zheng, A.L., Dias, O., and Huang, X.H., A fast, preconditioned conjugate gradient Toeplitz and Toeplitz-like solver, *Comp. & Math. (with Appl.),* 30(8), 57–63, 1995.
[79] Pan, V.Y., Zheng, A.L., Huang, X.H., and Yu, Y.Q., Fast multipoint polynomial evaluation and interpolation via computations with structured matrices. *Annals of Numerical Mathematics,* 4, 483–510, 1997.
[80] Pan, V.Y., Zheng, A.L., Huang, X.H., and Dias, O., Newton's iteration for inversion of cauchy-like and other structured matrices. *J. of Complexity,* 13, 108–124, 1997a.
[81] Press, W., Flannery, B., Teukolsky, S., and Vetterling, W., *Numerical Recipes in C: The Art of Scientific Computing.* Cambridge University Press, Cambridge, 1988, and 2nd ed. 1992.
[82] Rokhlin, F., Rapid solution of integral equations of classical potential theory. *J. Comput. Physics,* 60, 187–207, 1985.
[83] Runge, C. and König, H., *Die Grundlehren der mathematischen Wissenshaften,* 11. Springer-Verlag, Berlin, 1924.
[84] Schönhage, A., Grotefeld, A.F.W., and Vetter, E., *Fast Algorithms: A Multitape Turing Machine Implementation.* Wissenschaftsverlag, Mannheim, Germany, 1994.
[85] Schönhage, A. and Strassen, V., Schnelle multiplikation großer zahlen. *Computing,* 7, 281–292, 1971. In German.
[86] Strang, G., Wavelet transforms versus Fourier transforms. *Bulletin (New Series) of the American Mathematical Society,* 28(2), 288–305, 1993.
[87] Swarztrauber, P., FFT algorithms for vector computers. *Parallel Computing,* 1, 45–63, 1984. Implementation at
http://www.psc.edu/general/software/packages/fftpack/fftpack.html
or ftp://netlib.att.com/netlib.
[88] Toom, A.L., The complexity of a scheme of functional elements realizing the multiplication of integers. *Soviet Math. Doklady,* 3, 714–716, 1963.
[89] Van Loan, C.F., *Computational Frameworks for the Fast Fourier Transform.* SIAM Publications, Philadelphia, PA, 1992.
[90] Wiedemann, D.H., Solving sparse linear equations over finite fields. *IEEE Trans. Inf. Theory,* 32(1), 54–62, 1986.
[91] Winograd, S., *Arithmetic Complexity of Computations.* SIAM, Philadelphia, PA, 1980.
[92] Zippel, R., *Effective Polynomial Computation.* Kluwer Academic, Boston, 1993.

Further Information

The main research journals in this area are *Computers and Mathematics (with Applications), Journal of the ACM, Journal of Symbolic Computation, Linear Algebra and Its Applications, Mathematics of Computation, SIAM Journal of Computing, SIAM Journal of Matrix Analysis and Applications,* and *Theoretical Computer Science.* New implementations are reported in the *ACM Transactions on Mathematical Software.*

The main annual conferences in this area are the "ACM-SIGSAM International Symposium on Symbolic and Algebraic Computation," the "ACM Symposium on Theory of Computing," the "ACM-SIAM Symposium on Discrete Algorithms," the "European Symposium on Algorithms," the "IEEE Symposium on Foundations of Computer Science," and the "Symposium on Parallel Algorithms and Architectures."

The following books contain general information on the topics of this chapter [1, 12, 14, 17, 62, 92], where the last three present the state of the art, and [12] also points to an extensive list of applications. Some references on transforms and convolution have been listed at the beginning of Section 17.2.

Parallel computation is an important subject, which we have barely touched. We refer the interested reader to [12, 54, 70].

18

Multidimensional Data Structures[1]

Hanan Samet
University of Maryland

18.1 Introduction

The representation of **multidimensional data** is an important issue in applications in diverse fields that include database management systems, computer graphics, computer vision, computational geometry, image processing, geographic information systems (GIS), pattern recognition, VLSI design, and others. The most common definition of multidimensional data is a collection of points in a higher dimensional space. These points can represent locations and objects in space as well as more general records. As an example of a record, consider an employee record that has attributes corresponding to the employee's name, address, sex, age, height, weight, and social security number. Such records arise in database management systems and can be treated as points in, for this example, a seven-dimensional space (i.e., there is one dimension for each attribute), albeit the different dimensions have different type units (i.e., name and address are strings of characters, sex is binary; while age, height, weight, and social security number are numbers).

When multidimensional data corresponds to locational data, we have the additional property that all of the attributes have the same unit, which is distance in space. In this case, we can combine the attributes and pose queries that involve proximity. For example, we may wish to find the closest city to Chicago within the two-dimensional space from which the locations of the cities are drawn. Another query seeks to find all cities within 50 miles of Chicago. In contrast, such queries are not very meaningful when the attributes do not have the same type. For example, it is not customary to seek the person with age–weight combination closest to John Jones, as we do not have a commonly accepted unit of year-pounds (year-kilograms) or

Supported by the National Science Foundation under grant IRI-9712715.

definition thereof. It should be clear that we are not speaking of queries involving Boolean combinations of the different attributes (e.g., range queries), which are quite common.

When multidimensional data spans a continuous physical space (i.e., an infinite collection of locations), the issues become more interesting. In particular, we are no longer just interested in the locations of objects but, we are also interested in the space that they occupy (i.e., their extent). Some example objects include lines (e.g., roads, rivers), regions (e.g., lakes, counties, buildings, crop maps, polygons, polyhedra), rectangles, and surfaces. The objects may be disjoint or could even overlap. One way to deal with such data is to store it explicitly by parametrizing it and thereby reduce it to a point in a higher dimensional space. For example, a line in two-dimensional space can be represented by the coordinate values of its endpoints (i.e., a pair of x and a pair of y coordinate values) and then stored as a point in a four-dimensional space. Thus, in effect, we have constructed a transformation (i.e., mapping) from a two-dimensional space (i.e., the space from which the lines are drawn) to a four-dimensional space (i.e., the space containing the representative point corresponding to the line).

The transformation approach is fine if we are just interested in retrieving the data. It is appropriate for queries about the objects (e.g., determining all lines that pass through a given point or that share an endpoint, etc.) and the immediate space that they occupy. However, the drawback of the transformation approach is that it ignores the geometry inherent in the data (e.g., the fact that a line passes through a particular region) and its relationship to the space in which it is embedded.

For example, suppose that we want to detect if two lines are near each other, or, alternatively, to find the nearest line to a given line. This is difficult to do in the four-dimensional space, regardless of how the data in it is organized, since proximity in the two-dimensional space from which the lines are drawn is not necessarily preserved in the four-dimensional space. In other words, although the two lines may be very close to each other, the Euclidean distance between their representative points may be quite large.

Of course, we could overcome these problems by projecting the lines back to the original space from which they were drawn, but in such a case, we may ask what was the point of using the transformation in the first place? In other words, at the least, the representation that we choose for the data should allow us to perform operations on the data. Thus, we need special representations for spatial multidimensional data other than point representations. One solution is to use data structures that are based on spatial occupancy.

Spatial occupancy methods decompose the space from which the spatial data is drawn (e.g., the two-dimensional space containing the lines) into regions called *buckets.* They are also commonly known as **bucketing methods.** Traditional bucketing methods such as the grid file [45], BANG file [22], LSD trees [27], buddy trees [55], etc. have been designed for multidimensional point data that need not be locational. In the case of spatial data, these methods have usually been applied to the transformed data (i.e., the representative points). In contrast, we discuss their application to the actual objects in the space from which the objects are drawn (i.e., two dimensions in the case of a collection of line segments).

In this chapter, we explore a number of different representations of multidimensional data bearing the above issues in mind. In the case of point data, we examine representations of both locational and nonlocational data, as well as combinations of the two. While we cannot give exhaustive details of all of the data structures, we try to explain the intuition behind their development as well as to give literature pointers to where more information can be found. Many of these representations are described in greater detail in [50, 51], including an extensive bibliography. Our approach is primarily a descriptive one. Most of our examples are of two-dimensional spatial data, although we do touch briefly on three-dimensional data.

At times, we discuss bounds on execution time and space requirements. Nevertheless, this information is presented in an inconsistent manner. The problem is that such analyses are very difficult to perform for many of the data structures that we present. This is especially true for the data structures that are based on spatial occupancy (e.g., **quadtree** and **R-tree** variants). In particular, such methods have good observable average-case behavior but may have very bad worst cases which may only arise rarely in practice. Their analysis is beyond the scope of this chapter and usually we do not say anything about it. Nevertheless,

these representations find frequent use in applications where their behavior is deemed acceptable, and is often found to be better than that of solutions whose theoretical behavior would appear to be superior. The problem is primarily attributed to the presence of large constant factors which are usually ignored in the *big O* and Ω analyses [38].

The rest of this chapter is organized as follows. Section 18.2 reviews a number of representations of point data of arbitrary dimensionality. Section 18.3 describes bucketing methods that organize collections of spatial objects (as well as multidimensional point data) by aggregating their bounding rectangles. Sections 18.2 and 18.3 are applicable to both spatial and nonspatial data, although all the examples that we present are of spatial data. Section 18.4 focuses on representations of region data, while Section 18.5 discusses a subcase of region data, which consists of collections of rectangles. Section 18.6 deals with curvilinear data, which also includes polygonal subdivisions and collections of line segments. Section 18.7 contains a summary and a brief indication of some research issues. Section 18.8 reviews some of the definitions of the terms used in this chapter. Note that although our examples are primarily from a two-dimensional space, the representations are applicable to higher dimensional spaces as well.

18.2 Point Data

Our discussion assumes that there is one record per data point, and that each record contains several attributes or keys (also frequently called fields, dimensions, coordinates, and axes). In order to facilitate retrieval of a record based on some of its attribute values, we also assume the existence of an ordering for the range of values of each of these attributes. In the case of locational attributes, such an ordering is quite obvious as the values of these attributes are numbers. In the case of alphanumeric attributes, the ordering is usually based on the alphabetic sequence of the characters making up the attribute value. Other data such as color could be ordered by the characters making up the name of the color or possibly the color's wavelength. It should be clear that finding an ordering for the range of values of an attribute is generally not an issue; the real issue is what ordering to use!

The representation that is ultimately chosen for the data depends, in part, on answers to the following questions:

1. What operations are to be performed on the data?
2. Should we organize the data or the embedding space from which the data is drawn?
3. Is the database static or dynamic (i.e., can the number of data points grow and shrink at will)?
4. Can we assume that the volume of data is sufficiently small so that it can all fit in core, or should we make provisions for accessing disk-resident data?

Disk-resident data implies grouping the data (either the underlying space based on the volume — that is, the amount — of the data it contains or the points, hopefully, by the proximity of their values) into sets (termed *buckets*) corresponding to physical storage units (i.e., pages). This leads to questions about their size, and how they are to be accessed.

1. Do we require a constant time to retrieve a record from a file or is a logarithmic function of the number of records in the file adequate? This is equivalent to asking if the access is via a directory in the form of an array (i.e., direct access) or a tree?
2. How large can the directories be allowed to grow before it is better to rebuild them?
3. How should the buckets be laid out on the disk?

Clearly, these questions are complex and we cannot address them all here. Some are answered in other sections. In this section, we focus primarily on dynamic data with an emphasis on two dimensions (i.e., attributes) and concentrate on the following queries:

1. Point queries — that is, if a particular point is present.

2. Range queries.
3. Boolean combinations of 1 and 2.

Most of the representations that we describe can be extended easily to higher dimensions, although some like the priority search tree are basically for two-dimensional data. Our discussion and examples are based on the fact that all of the attributes are locational or numeric and that they have the same range, although all of the representations can also be used to handle nonlocational and nonnumeric attributes. When discussing behavior in the general case, we assume a data set of N points and d attributes.

The simplest way to store point data is in a sequential list. Accesses to the list can be sped up by forming sorted lists for the various attributes which are known as *inverted lists* (e.g., [37]). There is one list for each attribute. This enables pruning the search with respect to the value of one of the attributes. In order to facilitate random access, the lists can be implemented using range trees [10].

It should be clear that the inverted list is not particularly useful for range searches. The problem is that it can only speed up the search for one of the attributes (termed the *primary* attribute). A number of solutions have been proposed. These solutions can be decomposed into two classes. One class of solutions enhances the range tree corresponding to the inverted list to include information about the remaining attributes in its internal nodes. This is the basis of the multidimensional range tree and variants of the priority search tree [15, 41] that are discussed at the end of this section.

The second class of solutions is more widely used and is exemplified by the *fixed-grid* method [9, 37]. It partitions the space from which the data is drawn into rectangular cells by overlaying it with a grid. Each grid cell c contains a pointer to another structure (e.g., a list) which contains the set of points that lie in c. Associated with the grid is an access structure to enable the determination of the grid cell associated with a particular point p. This access structure acts like a directory and is usually in the form of a d-dimensional array with one entry per grid cell or a tree with one leaf node per grid cell.

There are two ways to build a fixed grid. We can either subdivide the space into equal-sized intervals along each of the attributes (resulting in congruent grid cells) or place the subdivision lines at arbitrary positions that are dependent on the underlying data. In essence, the distinction is between organizing the data to be stored and organizing the embedding space from which the data is drawn [45]. In particular, when the grid cells are congruent (i.e., equal-sized when all of the attributes are locational with the same range and termed a *uniform grid*), use of an array access structure is quite simple and has the desirable property that the grid cell associated with point p can be determined in constant time. Moreover, in this case, if the width of each grid cell is twice the search radius for a rectangular range query, then the average search time is $O(F \cdot 2^d)$ where F is the number of points that have been found [11]. Figure 18.1 is an example of a uniform-grid representation for a search radius equal to 10 (i.e., a square of size 20×20[2]).

Use of an array access structure when the grid cells are not congruent requires us to have a way of keeping track of their size so that we can determine the entry of the array access structure corresponding to the grid cell associated with point p. One way to do this is to make use of what are termed *linear scales,* which indicate the positions of the grid lines (or partitioning hyperplanes in $d > 2$ dimensions). Given a point p, we determine the grid cell in which p lies by finding the "coordinate values" of the appropriate grid cell. The linear scales are usually implemented as one-dimensional trees containing ranges of values.

The use of an array access structure is fine as long as the data is static. When the data is dynamic, it is likely that some of the grid cells become too full while other grid cells are empty. This means that we need to rebuild the grid (i.e., further partition the grid or reposition the grid partition lines or hyperplanes) so that the various grid cells are not too full. However, this creates many more empty grid cells as a result

[2]Note that although the data consists of three attributes, one of which is nonlocational (i.e., name) and two of which are locational (i.e., the coordinate values), retrieval is only on the basis of the locational attribute values. Thus, there is no ordering on the name, and, therefore, we treat this example as two-dimensional locational data.

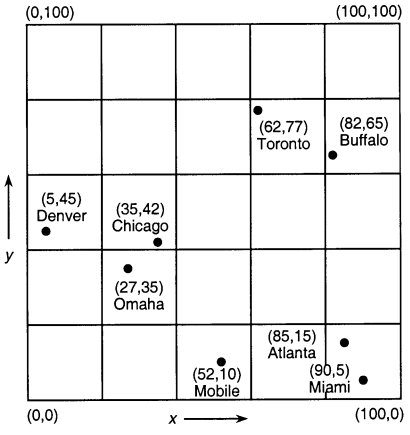


FIGURE 18.1 Uniform-grid representation corresponding to a set of points with a search radius of 20.

of repartitioning the grid (i.e., empty grid cells are split into more empty grid cells). In this case, we have two alternatives. The first is to assign an ordering to all the grid cells and to impose a tree access structure on the elements of the ordering that correspond to nonempty grid cells. The effect of this alternative is analogous to using a mapping from d dimensions to one dimension and then applying one of the one-dimensional access structures such as a B-tree, balanced binary tree, etc., to the result of the mapping. There are a number of possible mappings including row, Morton (i.e., bit interleaving or bit interlacing), and Peano–Hilbert (e.g.,[51])[3]. This alternative is applicable regardless of whether or not the grid cells are congruent. Of course, if the grid cells are not congruent, then we must also record their size in the element of the access structure.

The second alternative is to merge spatially adjacent empty grid cells into larger empty grid cells, while splitting grid cells that are too full, thereby making the grid adaptive. Again, the result is that we can no longer make use of an array access structure to retrieve the grid cell that contains query point p. Instead, we make use of a tree access structure in the form of a k-ary tree where k is usually 2^d. Thus, what we have done is marry a k-ary tree with the fixed-grid method. This is the basis of the point quadtree [17] and the PR quadtree [46, 51] which are multidimensional generalizations of binary trees.

The difference between the point quadtree and the PR quadtree is the same as the difference between *trees* and *tries* [20], respectively. The binary search tree [37] is an example of the former since the boundaries of different regions in the search space are determined by the data being stored. Address computation methods such as radix searching [37] (also known as digital searching) are examples of the latter, since region boundaries are chosen from among locations that are fixed regardless of the content of the data set. The process is usually a recursive halving process in one dimension, recursive quartering in two dimensions, etc., and is known as **regular decomposition.**

In two dimensions, a point quadtree is just a two-dimensional binary search tree. The first point that is inserted serves as the root, while the second point is inserted into the relevant quadrant of the tree rooted at the first point. Clearly, the shape of the tree depends on the order in which the points were inserted. For example, Fig. 18.2 is the point quadtree corresponding to the data of Fig. 18.1 inserted in the order `Chicago`, `Mobile`, `Toronto`, `Buffalo`, `Denver`, `Omaha`, `Atlanta`, and `Miami`.

In two dimensions, the PR quadtree is based on a recursive decomposition of the underlying space into four congruent (usually square in the case of locational attributes) cells until each cell contains no more

[3]These mappings have been investigated primarily for purely locational multidimensional point data. They cannot be applied directly to the key values for nonlocational point data.

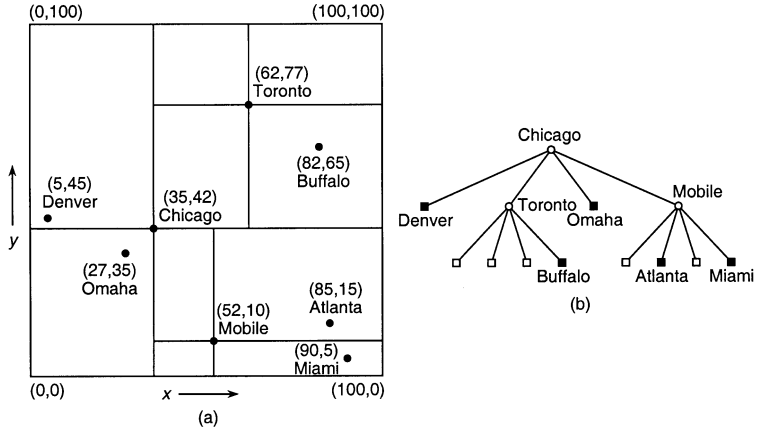


FIGURE 18.2 A point quadtree and the records it represents corresponding to Fig.18.1: (a) the resulting partition of space, and (b) the tree representation.

than one point. For example, Fig. 18.3 is the PR quadtree corresponding to the data of Fig. 18.1. The shape of the PR quadtree is independent of the order in which data points are inserted into it. The disadvantage of the PR quadtree is that the maximum level of decomposition depends on the minimum separation between two points. In particular, if two points are very close, then the decomposition can be very deep. This can be overcome by viewing the blocks or nodes as buckets with capacity c and only decomposing a block when it contains more than c points.

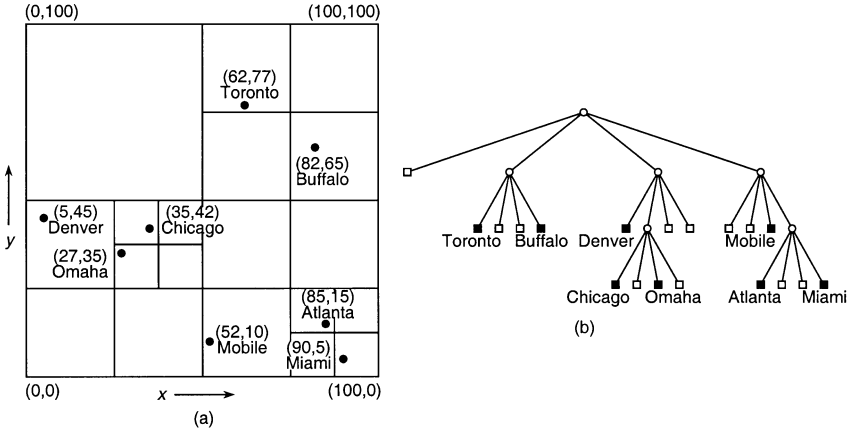


FIGURE 18.3 A PR quadtree and the records it represents corresponding to Fig.18.1: (a) the resulting partition of space, and (b) the tree representation.

As the dimensionality of the space increases, each level of decomposition of the quadtree results in many new cells as the fanout value of the tree is high (i.e., 2^d). This is alleviated by making use of a **k-d tree** [7]. The k-d tree is a binary tree where at each level of the tree, we subdivide along a different attribute so that, assuming d locational attributes, if the first split is along the x axis, then after d levels, we cycle back and again split along the x axis. It is applicable to both the point quadtree and the PR quadtree (in which case we have a *PR k-d tree,* or a **bintree** in the case of region data).

At times, in the dynamic situation, the data volume becomes so large that a tree access structure is inefficient. In particular, the grid cells can become so numerous that they cannot all fit into memory, thereby causing them to be grouped into sets (termed *buckets*) corresponding to physical storage units (i.e., pages) in secondary storage. The problem is that, depending on the implementation of the tree access structure, each time we must follow a pointer, we may need to make a disk access. This has led to a return to the use of an array access structure. The difference from the array used with the static fixed-grid method described earlier is that the array access structure (termed *grid directory*) may be so large (e.g., when d gets large) that it resides on disk as well, and the fact that the structure of the grid directory can be changed as the data volume grows or contracts. Each grid cell (i.e., an element of the grid directory) contains the address of a bucket (i.e., page) that contains the points associated with the grid cell. Notice that a bucket can correspond to more than one grid cell. Thus, any page can be accessed by two disk operations: one to access the grid cell and one more to access the actual bucket.

This results in *EXCELL* [59] when the grid cells are congruent (i.e., equal-sized for locational data), and *grid file* [45] when the grid cells need not be congruent. The difference between these methods is most evident when a grid partition is necessary (i.e., when a bucket becomes too full and the bucket is not shared among several grid cells). In particular, a grid partition in the grid file only splits one interval in two, thereby resulting in the insertion of a $(d-1)$-dimensional cross section. On the other hand, a grid partition in EXCELL means that all intervals must be split in two, thereby doubling the size of the grid directory.

Fixed-grids, quadtrees, k-d trees, grid file, EXCELL, as well as other hierarchical representations are good for range searching as they make it easy to implement the query. A typical query is one that seeks all cities within 80 miles of St. Louis, or, more generally, within 80 miles of the latitude position of St. Louis and within 80 miles of the longitude position of St. Louis.[4] In particular, these structures act as pruning devices on the amount of search that will be performed as many points will not be examined since their containing cells lie outside the query range. These representations are generally very easy to implement and have good expected execution times, although they are quite difficult to analyze from a mathematical standpoint. However, their worst cases, despite being rare, can be quite bad. These worst cases can be avoided by making use of variants of range trees [10] and priority search trees [41]. They are applicable to both locational and nonlocational attributes, although our presentation assumes that all the attributes are locational.

A one-dimensional range tree is a balanced binary search tree where the data points are stored in the leaf nodes and the leaf nodes are linked in sorted order by use of a doubly-linked list. A range search for $[L : R]$ is performed by searching the tree for the node with the smallest value that is $\geq L$, and then following the links until reaching a leaf node with a value greater than R. For N points, this process takes $O(\log_2 N + F)$ time and uses $O(N)$ storage. F is the number of points found.

A two-dimensional range tree is a binary tree of binary trees. It is formed in the following manner. First, sort all of the points along one of the attributes, say x, and store them in the leaf nodes of a balanced binary search tree, say T. With each non-leaf node of T, say I, associate a one-dimensional range tree, say T_I, of the points in the subtree rooted at I where now these points are sorted along the other attribute, say y. The range tree also can be adapted easily to handle d-dimensional data. In such a case, for N points, a d-dimensional range search takes $O(\log_2^d N + F)$ time, where F is the number of points found. The d-dimensional range tree uses $O(N \cdot \log_2^{d-1} N)$ storage.

The *priority search tree* is a related data structure that is designed for solving queries involving semi-infinite ranges in two-dimensional space. A typical query has a range of the form $([L_x : R_x], [L_y : \infty])$.

[4]The difference between these two formulations of the query is that the former admits a circular search region, while the latter admits a rectangular search region. In particular, the latter formulation is applicable to both locational and non-locational attributes, while the former is only applicable to locational attributes.

For example, Fig. 18.4 is the priority search tree for the data of Fig. 18.1. It is built as follows. Assume that no two data points have the same x coordinate value. Sort all the points along the x coordinate value and store them in the leaf nodes of a balanced binary search tree (a range tree in our formulation), say T. We proceed from the root node toward the leaf nodes. With each node I of T, associate the point in the subtree rooted at I with the maximum value for its y coordinate that has not already been stored at a shallower depth in the tree. If such a point does not exist, then leave the node empty. For N points, this structure uses $O(N)$ storage.

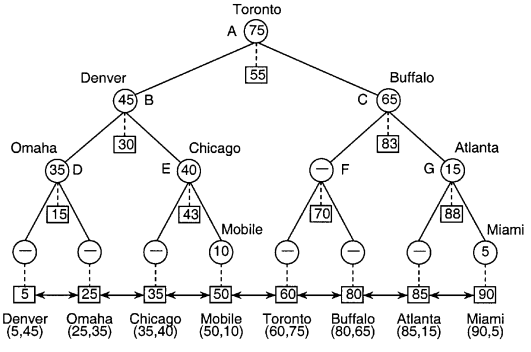


FIGURE 18.4 Priority search tree for the data of Fig.18.1. Each leaf node contains the value of its x coordinate in a square box. Each nonleaf node contains the appropriate x coordinate midrange value in a box using a link drawn with a broken line. Circular boxes indicate the value of the y coordinate of the point in the corresponding subtree with the maximum value for its y coordinate that has not already been associated with a node at a shallower depth in the tree.

It is not easy to perform a two-dimensional range query of the form $([L_x : R_x], [L_y : R_y])$ with a priority search tree. The problem is that only the values of the x coordinates are sorted. In other words, given a leaf node C that stores point (x_C, y_C), we know that the values of the x coordinates of all nodes to the left of C are smaller than x_C and the values of all those to the right of C are greater than x_C. On the other hand, with respect to the values of the y coordinates, we only know that all nodes below non-leaf node D with value y_D have values less than or equal to y_D; the y coordinate values associated with the remaining nodes in the tree that are not ancestors of D may be larger or smaller than y_D. This is not surprising, because a priority search tree is really a variant of a range tree in x and a heap (i.e., priority queue) [37] in y.

A heap enables finding the maximum (minimum) value in $O(1)$ time. Thus, it is easy to perform a semi-infinite range query of the form $([L_x : R_x], [L_y : \infty])$, as all we need do is descend the priority search tree and stop as soon as we encounter a y coordinate value that is less than L_y. For N points, performing a semi-infinite range query in this way takes $O(\log_2 N + F)$ time, where F is the number of points found.

The priority search tree is used as the basis of the *range priority tree* [15] to reduce the order of execution time of a two-dimensional range query to $O(\log_2 N + F)$ time (but still using $O(N \cdot \log_2 N)$ storage). Define an *inverse priority search tree* to be a priority search tree S such that with each node of S, say I, we associate the point in the subtree rooted at I with the minimum (instead of the maximum!) value for its y coordinate that has not already been stored at a shallower depth in the tree. The range priority tree is a balanced binary search tree (i.e., a range tree), say T, where all the data points are stored in the leaf nodes and are sorted by their y coordinate values. With each non-leaf node of T, say I, which is a left son of its father, we store a priority search tree of the points in the subtree rooted at I. With each non-leaf node of T, say I, which is a right son of its father we store an inverse priority search tree of the points in the subtree rooted at I. For N points, the range priority tree uses $O(N \cdot \log_2 N)$ storage.

Performing a range query for $([L_x : R_x], [L_y : R_y])$ using a range priority tree is done in the following manner. We descend the tree looking for the nearest common ancestor of L_y and R_y, say Q. The values of the y coordinates of all points in the left son of Q are less than R_y. We want to retrieve just the ones that are greater than or equal to L_y. We can obtain them with the semi-infinite range query $([L_x : R_x], [L_y : \infty])$. This can be done by using the priority tree associated with the left son of Q. Similarly, the values of the y coordinates of all points in the right son of Q are greater than L_y. We want to retrieve just the ones that are less than or equal to R_y. We can obtain them with the semi-infinite range query $([L_x : R_x], [-\infty : R_y])$. This can be done by using the inverse priority search tree associated with the right son of Q. Thus, for N points the range query takes $O(\log_2 N + F)$ time, where F is the number of points found.

18.3 Bucketing Methods

There are four principal approaches to decomposing the space from which the records are drawn. They are applicable regardless of whether the attributes are locational or nonlocational, although our discussion assumes that they are locational and that the records correspond to spatial objects. One approach makes use of an object hierarchy. It propagates the space occupied by the objects up the hierarchy with the identity of the propagated objects being implicit to the hierarchy. In particular, associated with each object is an object description (e.g., for region data, it is the set of locations in space corresponding to the cells that make up the object). Actually, since this information may be rather voluminous, it is often the case that an approximation of the space occupied by the object is propagated up the hierarchy rather than the collection of individual cells that are spanned by the object. For spatial data, the approximation is usually the minimum bounding rectangle for the object, while for nonspatial data it is simply the hyperrectangle whose sides have lengths equal to the ranges of the values of the attributes. Therefore, associated with each element in the hierarchy is a bounding rectangle corresponding to the union of the bounding rectangles associated with the elements immediately below it.

The R-tree (e.g., [6, 26]) is an example of an object hierarchy that finds use especially in database applications. The number of objects or bounding rectangles that are aggregated in each node is permitted to range between $m \leq \lceil M/2 \rceil$ and M. The root node in an R-tree has at least two entries unless it is a leaf node, in which case it has just one entry corresponding to the bounding rectangle of an object. The R-tree is usually built as the objects are encountered rather than waiting until all objects have been input. The hierarchy is implemented as a tree structure with grouping being based, in part, on proximity of the objects or bounding rectangles.

For example, consider the collection of line segment objects given in Fig. 18.5 shown embedded in a 4×4 grid. Figure 18.6(a) is an example R-tree for this collection with $m = 2$ and $M = 3$. Figure 18.6(b) shows the spatial extent of the bounding rectangles of the nodes in Fig. 18.6(a), with heavy lines denoting the bounding rectangles corresponding to the leaf nodes, and broken lines denoting the bounding rectangles corresponding to the subtrees rooted at the nonleaf nodes. Note that the R-tree is not unique. Its structure depends heavily on the order in which the individual objects were inserted into (and possibly deleted from) the tree.

Given that each R-tree node can contain a varying number of objects or bounding rectangles, it is not surprising that the R-tree was inspired by the B-tree [12]. Therefore, nodes are viewed as analogous to disk pages. Thus, the parameters defining the tree (i.e., m and M) are chosen so that a small number of nodes is visited during a spatial query (i.e., point and range queries), which means that m and M are usually quite large. The actual implementation of the R-tree is really a B^+-tree [12] as the objects are restricted to the leaf nodes.

The efficiency of the R-tree for search operations depends on its ability to distinguish between occupied space and unoccupied space (i.e., coverage), and to prevent a node from being examined needlessly due to a false overlap with other nodes. In other words, we want to minimize coverage and overlap. These goals

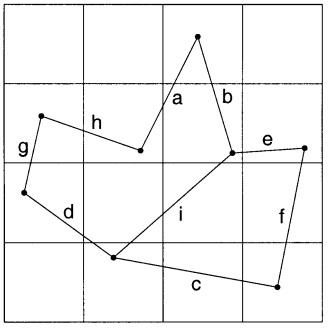


FIGURE 18.5 Example collection of line segments embedded in a 4×4 grid.

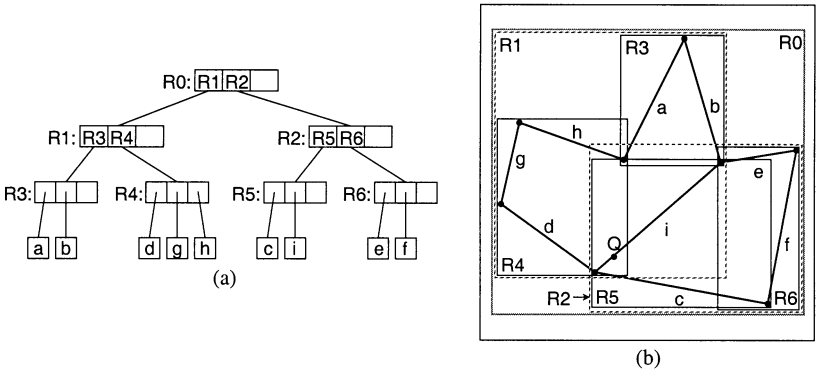


FIGURE 18.6 (a) R-tree for the collection of line segments with $m = 2$ and $M = 3$, in Fig.18.5, and (b) the spatial extents of the bounding rectangles. Notice that the leaf nodes in the index also store bounding rectangles, although this is only shown for the nonleaf nodes.

guide the initial R-tree creation process as well, subject to the previously mentioned constraint that the R-tree is usually built as the objects are encountered rather than waiting until all objects have been input.

The drawback of the R-tree (and any representation based on an object hierarchy) is that it does not result in a disjoint decomposition of space. The problem is that an object is only associated with one bounding rectangle (e.g., line segment `i` in Fig. 18.6 is associated with bounding rectangle `R5`, yet it passes through `R1`, `R2`, `R4`, and `R5`, as well as through `R0` as do all the line segments). In the worst case, this means that when we wish to determine which object (e.g., an intersecting line in a collection of line segment objects, or a containing rectangle in a collection of rectangle objects) is associated with a particular point in the two-dimensional space from which the objects are drawn, we may have to search the entire collection. For example, in Fig. 18.6, when searching for the line segment that passes through point Q, we need to examine bounding rectangles `R0`, `R1`, `R4`, `R2`, and `R5`, rather than just `R0`, `R2`, and `R5`.

This drawback can be overcome by using one of three other approaches that are based on a decomposition of space into disjoint cells. Their common property is that the objects are decomposed into disjoint subobjects such that each of the subobjects is associated with a different cell. They differ in the degree of regularity imposed by their underlying decomposition rules, and by the way in which the cells are aggregated into buckets.

The price paid for the disjointness is that in order to determine the area covered by a particular object, we have to retrieve all the cells that it occupies. This price is also paid when we want to delete an object. Fortunately, deletion is not so common in such applications. A related costly consequence of disjointness is that when we wish to determine all the objects that occur in a particular region, we often need to retrieve some of the objects more than once. This is particularly troublesome when the result of the operation serves as input to another operation via composition of functions. For example, suppose we wish to

compute the perimeter of all the objects in a given region. Clearly, each object's perimeter should only be computed once. Eliminating the duplicates is a serious issue (see [1] for a discussion of how to deal with this problem for a collection of line segment objects, and [2] for a collection of rectangle objects).

The first method based on disjointness partitions the embedding space into disjoint subspaces, and hence, the individual objects into subobjects, so that each subspace consists of disjoint subobjects. The subspaces are then aggregated and grouped in another structure, such as a B-tree, so that all subsequent groupings are disjoint at each level of the structure. The result is termed a k-d-B-tree [49]. The R^+-tree [56, 58] is a modification of the k-d-B-tree where at each level we replace the subspace by the minimum bounding rectangle of the subobjects or subtrees that it contains. The cell tree [25] is based on the same principle as the R^+-tree except that the collections of objects are bounded by minimum convex polyhedra instead of minimum bounding rectangles.

The R^+-tree (as well as the other related representations) is motivated by a desire to avoid overlap among the bounding rectangles. Each object is associated with all the bounding rectangles that it intersects. All bounding rectangles in the tree (with the exception of the bounding rectangles for the objects at the leaf nodes) are nonoverlapping.[5] The result is that there may be several paths starting at the root to the same object. This may lead to an increase in the height of the tree. However, retrieval time is sped up.

Figure 18.7 is an example of one possible R^+-tree for the collection of line segments in Fig. 18.5. This particular tree is of order (2,3) although in general it is not possible to guarantee that all nodes will always have a minimum of 2 entries. In particular, the expected B-tree performance guarantees are not valid (i.e., pages are not guaranteed to be m/M full) unless we are willing to perform very complicated record insertion and deletion procedures. Notice that line segment objects `c`, `h`, and `i` appear in two different nodes. Of course, other variants are possible since the R^+-tree is not unique.

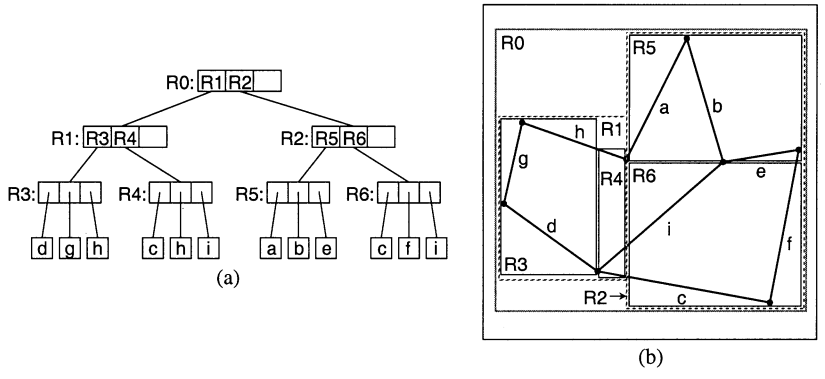


FIGURE 18.7 (a) R^+-tree for the collection of line segments in Fig.18.5 with $m = 2$ and $M = 3$, and (b) the spatial extents of the bounding rectangles. Notice that the leaf nodes in the index also store bounding rectangles, although this is only shown for the non-leaf nodes.

Methods such as the R^+-tree (as well as the R-tree) have the drawback that the decomposition is data-dependent. This means that it is difficult to perform tasks that require composition of different operations and data sets (e.g., set-theoretic operations such as overlay). The problem is that although these methods

[5] From a theoretical viewpoint, the bounding rectangles for the objects at the leaf nodes should also be disjoint However, this may be impossible (e.g., when the objects are line segments and if many of the line segments intersect at a point).

are good are distinguishing between occupied and unoccupied space in a particular image, they are unable to correlate occupied space in two distinct images, and likewise for unoccupied space in the two images.

In contrast, the remaining two approaches to the decomposition of space into disjoint cells have a greater degree of data-independence. They are based on a regular decomposition. The space can be decomposed either into blocks of uniform size (e.g., the uniform grid [19]) or adapt the decomposition to the distribution of the data (e.g., a quadtree-based approach such as [54]). In the former case, all the blocks are congruent (e.g., the 4×4 grid in Fig. 18.5). In the latter case, the widths of the blocks are restricted to be powers of two[6] and their positions are also restricted. Since the positions of the subdivision lines are restricted, and essentially the same for all images of the same size, it is easy to correlate occupied and unoccupied space in different images.

The uniform grid is ideal for uniformly distributed data, while quadtree-based approaches are suited for arbitrarily distributed data. In the case of uniformly distributed data, quadtree-based approaches degenerate to a uniform grid, albeit they have a higher overhead. Both the uniform grid and the quadtree-based approaches lend themselves to set-theoretic operations, and thus they are ideal for tasks that require the composition of different operations and data sets. In general, since spatial data are not usually uniformly distributed, the quadtree-based regular decomposition approach is more flexible. The drawback of quadtree-like methods is their sensitivity to positioning in the sense that the placement of the objects relative to the decomposition lines of the space in which they are embedded effects their storage costs and the amount of decomposition that takes place. This is overcome to a large extent by using a bucketing adaptation that decomposes a block only if it contains more than b objects.

18.4 Region Data

There are many ways of representing region data. We can represent a region either by its boundary (termed a **boundary-based representation**) or by its interior (termed an **interior-based representation**). In this section, we focus on representations of collections of regions by their interior. In some applications, regions are really objects that are composed of smaller primitive objects by use of geometric transformations and Boolean set operations. *Constructive solid geometry* (CSG) [48] is a term usually used to describe such representations. They are beyond the scope of this chapter. Instead, unless noted otherwise, our discussion is restricted to regions consisting of congruent cells of unit area (volume) with sides (faces) of unit size that are orthogonal to the coordinate axes.

Regions with arbitrary boundaries are usually represented by either using approximating bounding rectangles or more general boundary-based representations that are applicable to collections of line segments that do not necessarily form regions. In that case, we do not restrict the line segments to be perpendicular to the coordinate axes. Such representations are discussed in Section 18.6. It should be clear that although our presentation and examples in this section deal primarily with two-dimensional data, they are valid for regions of any dimensionality.

The region data is assumed to be uniform in the sense that all the cells that comprise each region are of the same type. In other words, each region is homogeneous. Of course, an image may consist of several distinct regions. Perhaps the best definition of a region is as a set of four-connected cells (i.e., in two dimensions, the cells are adjacent along an edge rather than a vertex) each of which is of the same type. For example, we may have a crop map where the regions correspond to the four-connected cells on which the same crop is grown. Each region is represented by the collection of cells that comprise it. The set of

[6] More precisely, for arbitrary attributes that can be locational and nonlocational, there exist $j \geq 0$ such that the product of w_i, the width of the block along attribute i, and 2^j is equal to the length of the range of values of attribute i.

collections of cells that make up all of the regions is often termed an *image array,* because of the nature in which they are accessed when performing operations on them. In particular, the array serves as an access structure in determining the region associated with a location of a cell as well as all remaining cells that comprise the region.

When the region is represented by its interior, then often we can reduce the storage requirements by aggregating identically valued cells into blocks. In the rest of this section we discuss different methods of aggregating the cells that comprise each region into blocks as well as the methods used to represent the collections of blocks that comprise each region in the image.

The collection of blocks is usually a result of a space decomposition process with a set of rules that guide it. There are many possible decompositions. When the decomposition is recursive, we have the situation that the decomposition occurs in stages and often, although not always, the results of the stages form a containment hierarchy. This means that a block b obtained in stage i is decomposed into a set of blocks b_j that span the same space. Blocks b_j are, in turn, decomposed in stage $i + 1$ using the same decomposition rule. Some decomposition rules restrict the possible sizes and shapes of the blocks as well as their placement in space. Some examples include

- Congruent blocks at each stage
- Similar blocks at all stages
- All sides of a block are of equal size
- All sides of each block are powers of two.

Other decomposition rules dispense with the requirement that the blocks be rectangular (i.e., there exist decompositions using other shapes such as triangles, etc.), while still others do not require that they be orthogonal, although, as stated before, we do make these assumptions here. In addition, the blocks may be disjoint or be allowed to overlap. Clearly, the choice is large. In the following, we briefly explore some of these decomposition processes. We restrict ourselves to disjoint decompositions, although this need not be the case (e.g., the field tree [18]).

The most general decomposition permits aggregation along all dimensions. In other words, the decomposition is arbitrary. The blocks need not be uniform or similar. The only requirement is that the blocks span the space of the environment. The drawback of arbitrary decompositions is that there is little structure associated with them. This means that it is difficult to answer queries such as determining the region associated with a given point, besides exhaustive search through the blocks. Thus, we need an additional data structure known as an index or an access structure. A very simple decomposition rule that lends itself to such an index in the form of an array is one that partitions a d-dimensional space having coordinate axes x_i into d-dimensional blocks by use of h_i hyperplanes that are parallel to the hyperplane formed by $x_i = 0$ $(1 \leq i \leq d)$. The result is a collection of $\prod_{i=1}^{d}(h_i + 1)$ blocks. These blocks form a grid of irregular-sized blocks rather than congruent blocks. There is no recursion involved in the decomposition process. We term the resulting decomposition an *irregular grid,* as the partition lines are at arbitrary positions in contrast to a *uniform grid* [19] where the partition lines are positioned so that all of the resulting grid cells are congruent.

Although the blocks in the irregular grid are not congruent, we can still impose an array access structure by adding d access structures termed *linear scales.* The linear scales indicate the position of the partitioning hyperplanes that are parallel to the hyperplane formed by $x_i = 0$ $(1 \leq i \leq d)$. Thus, given a location l in space, say (a,b) in two-dimensional space, the linear scales for the x and y coordinate values indicate the column and row, respectively, of the array access structure entry which corresponds to the block that contains l. The linear scales are usually represented as one-dimensional arrays although they can be implemented using tree access structures such as binary search trees, range trees, segment trees, etc.

Perhaps the most widely known decompositions into blocks are those referred to by the general terms *quadtree* and **octree** [50, 51]. They are usually used to describe a class of representations for two- and three-dimensional data (and higher as well), respectively, that are the result of a recursive decomposition

of the environment (i.e., space) containing the regions into blocks (not necessarily rectangular) until the data in each block satisfies some condition (e.g., with respect to its size, the nature of the regions that comprise it, the number of regions in it, etc.). The positions and/or sizes of the blocks may be restricted or arbitrary. It is interesting to note that quadtrees and octrees may be used with both interior-based and boundary-based representations, although only the former are discussed in this section.

There are many variants of quadtrees and octrees (see also Sections 18.2, 18.5, and 18.6), and they are used in numerous application areas including high energy physics, VLSI, finite element analysis, and many others. Below, we focus on *region quadtrees* [35] and to a lesser extent on *region octrees* [32, 43] They are specific examples of interior-based representations for two- and three-dimensional region data (variants for data of higher dimension also exist), respectively, that permit further aggregation of identically-valued cells.

Region quadtrees and region octrees are instances of a restricted-decomposition rule where the environment containing the regions is recursively decomposed into four or eight, respectively, rectangular congruent blocks until each block is either completely occupied by a region or is empty (such a decomposition process is termed *regular*). For example, Fig. 18.8(a) is the block decomposition for the region quadtree corresponding to three regions A, B, and C. Notice that in this case, all the blocks are square, have sides whose size is a power of 2, and are located at specific positions. In particular, assuming an origin at the upper-left corner of the image containing the regions, then the coordinate values of the upper-left corner of each block (e.g., (a, b) in two dimensions) of size $2^i \times 2^i$ satisfy the property that $a \bmod 2^i = 0$ and $b \bmod 2^i = 0$.

The traditional, and most natural, access structure for a region quadtree corresponding to a d-dimensional image is a tree with a fanout of 2^d [e.g., Fig. 18.8(b)]. Each leaf node in the tree corresponds to a different block b and contains the identity of the region associated with b. Each non-leaf node f corresponds to a block whose volume is the union of the blocks corresponding to the 2^d sons of f. In this case, the tree is a containment hierarchy and closely parallels the decomposition in the sense that they are both recursive processes and the blocks corresponding to nodes at different depths of the tree are similar in shape.

Determining the region associated with a given point p is achieved by a process that starts at the root of the tree and traverses the links to the sons whose corresponding blocks contain p. This process has an $O(m)$ cost where the image has a maximum of m levels of subdivision (e.g., an image all of whose sides are of length 2^m).

Observe that using a tree with fanout 2^d as an access structure for a regular decomposition means that there is no need to record the size and location of the blocks as this information can be inferred

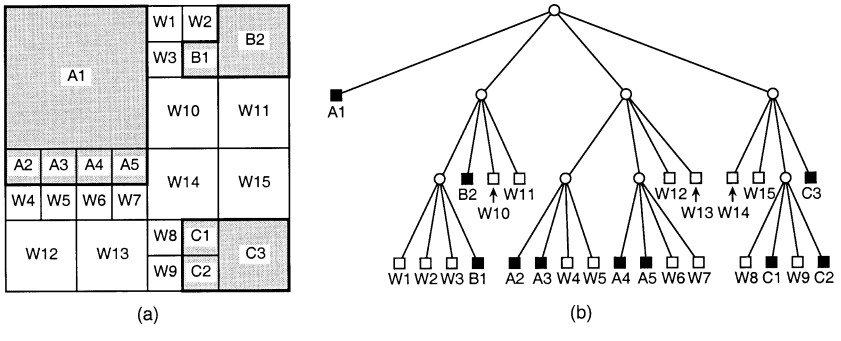


FIGURE 18.8 (a) Block decomposition and (b) its tree representation for the region quadtree corresponding to a collection of three regions A, B, and C.

from knowledge of the size of the underlying space. This is because the 2^d blocks that result from each subdivision step are congruent. For example, in two dimensions, each level of the tree corresponds to a quartering process that yields four congruent blocks. Thus, as long as we start from the root, we know the location and size of every block.

One of the motivations for the development of data structures such as the region quadtree is a desire to save space. The formulation of the region quadtree that we have just described makes use of an access structure in the form of a tree. This requires additional overhead to encode the internal nodes of the tree as well as the pointers to the subtrees. In order to further reduce the space requirements, a number of alternative access structures to the tree with fanout 2^d have been proposed. They are all based on finding a mapping from the domain of the blocks to a subset of the integers (i.e., to one dimension) and then using the result of the mapping as the index in one of the familiar tree-like access structures (e.g., a binary search tree, range tree, B^+-tree, etc.). The effect of these mappings is to provide an ordering on the underlying space. There are many possible orderings (e.g., Chapter 2 in [50]), with the most popular shown in Fig. 18.9. The domain of these mappings is the location of the cells in the underlying space, and thus we need to use some easily identifiable cell in each block such as the one in the block's upper-left corner. Of course, we also need to know the size of each block. This information can be recorded in the actual index as each block is uniquely identified by the location of the cell in its upper-left corner.

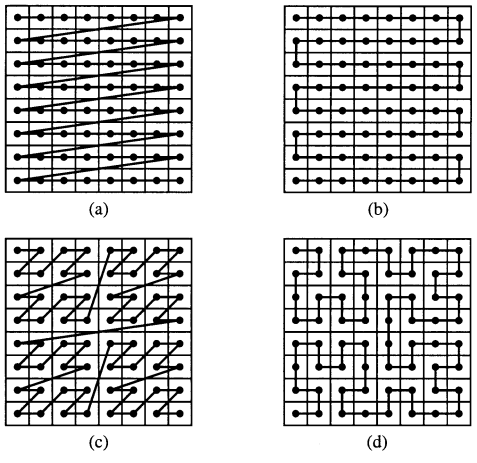


FIGURE 18.9 The result of applying four common different space-ordering methods to an 8×8 collection of cells whose first element is in the upper-left corner: (a) row order, (b) row-prime order, (c) Morton order, (d) Peano–Hilbert.

Since the size of each block b in the region quadtree can be specified with a single number indicating the depth in the tree at which b is found, we can simplify the representation by incorporating the size into the mapping. One mapping simply concatenates the result of interleaving the binary representations of the coordinate values of the upper-left corner (e.g., (a, b) in two dimensions) and i of each block of size 2^i so that i is at the right. The resulting number is termed a *locational code* and is a variant of the Morton order [Fig. 18.9(c)]. Assuming such a mapping and sorting the locational codes in increasing order yields an ordering equivalent to that which would be obtained by traversing the leaf nodes (i.e., blocks) of the tree representation [e.g., Fig. 18.8(b)] in the order `NW`, `NE`, `SW`, `SE`. The Morton ordering [as well as the Peano–Hilbert ordering shown in Fig. 18.9(d)] is particularly attractive for quadtree-like block decompositions, because all cells within a quadtree block appear in consecutive positions in the

ordering. Alternatively, these two orders exhaust a quadtree block before exiting it. Therefore, once again, determining the region associated with point p consists of simply finding the block containing p.

As the dimensionality of the space (i.e., d) increases, each level of decomposition in the region quadtree results in many new blocks as the fanout value 2^d is high. In particular, it is too large for a practical implementation of the tree access structure. In this case, an access structure termed a *bintree* [36, 53, 60] with a fanout value of 2 is used. The bintree is defined in a manner analogous to the region quadtree except that at each subdivision stage, the space is decomposed into two equal-sized parts. In two dimensions, at odd stages we partition along the y axis and at even stages we partition along the x axis. In general, in the case of d dimensions, we cycle through the different axes every d levels in the bintree.

The region quadtree, as well as the bintree, is a regular decomposition. This means that the blocks are congruent—that is, at each level of decomposition, all of the resulting blocks are of the same shape and size. We can also use decompositions where the sizes of the blocks are not restricted in the sense that the only restriction is that they be rectangular and be a result of a recursive decomposition process. In this case, the representations that we described must be modified so that the sizes of the individual blocks can be obtained. An example of such a structure is an adaptation of the point quadtree [17] to regions. Although the point quadtree was designed to represent points in a higher dimensional space, the blocks resulting from its use to decompose space do correspond to regions. The difference from the region quadtree is that in the point quadtree, the positions of the partitions are arbitrary, whereas they are a result of a partitioning process into 2^d congruent blocks (e.g., quartering in two dimensions) in the case of the region quadtree.

As in the case of the region quadtree, as the dimensionality d of the space increases, each level of decomposition in the point quadtree results in many new blocks since the fanout value 2^d is high. In particular, it is too large for a practical implementation of the tree access structure. In this case, we can adapt the k-d tree [7], which has a fanout value of 2, to regions. As in the point quadtree, although the k-d tree was designed to represent points in a higher dimensional space, the blocks resulting from its use to decompose space do correspond to regions. Thus, the relationship of the k-d tree to the point quadtree is the same as the relationship of the bintree to the region quadtree. In fact, the k-d tree is the precursor of the bintree and its adaptation to regions is defined in a similar manner in the sense that for d-dimensional data we cycle through the d axes every d levels in the k-d tree. The difference is that in the k-d tree, the positions of the partitions are arbitrary, whereas they are a result of a halving process in the case of the bintree.

The k-d tree can be further generalized so that the partitions take place on the various axes at an arbitrary order, and, in fact, the partitions need not be made on every coordinate axis. The k-d tree is a special case of the *BSP tree* (denoting *binary space partitioning*) [23] where the partitioning hyperplanes are restricted to be parallel to the axes, whereas in the BSP tree they have an arbitrary orientation. The BSP tree is a binary tree. In order to be able to assign regions to the left and right subtrees, we need to associate a direction with each subdivision line. In particular, the subdivision lines are treated as separators between two halfspaces.[7] Let the subdivision line have the equation $a \cdot x + b \cdot y + c = 0$. We say that the right subtree is the "positive" side and contains all subdivision lines formed by separators that satisfy $a \cdot x + b \cdot y + c \geq 0$. Similarly, we say that the left subtree is "negative" and contains all subdivision lines formed by separators that satisfy $a \cdot x + b \cdot y + c < 0$. As an example, consider Fig. 18.10(a), which is an arbitrary space decomposition whose BSP tree is given in Fig. 18.10(b). Notice the use of arrows to indicate the direction of the positive halfspaces. The BSP tree is used in computer graphics to facilitate viewing.

[7] A (linear) *halfspace* in d-dimensional space is defined by the inequality $\Sigma_{i=0}^{d} a_i \cdot x_i \geq 0$ on the $d+1$ homogeneous coordinates ($x_0 = 1$). The halfspace is represented by a column vector a. In vector notation, the inequality is written as $a \cdot x \geq 0$. In the case of equality, it defines a hyperplane with a as its normal. It is important to note that halfspaces are volume elements; they are not boundary elements.

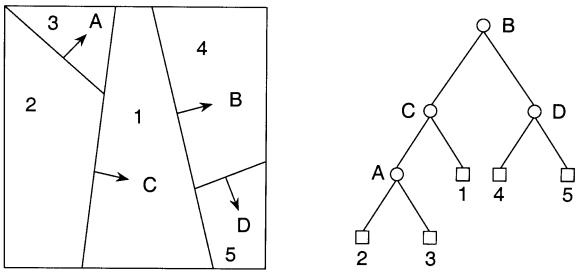


FIGURE 18.10 (a) An arbitrary space decomposition and (b) its BSP tree. The arrows indicate the direction of the positive halfspaces.

As mentioned before, the various hierarchical data structures that we described can also be used to represent regions in three dimensions and higher. As an example, we briefly describe the region octree which is the three-dimensional analog of the region quadtree. It is constructed in the following manner. We start with an image in the form of a cubical volume and recursively subdivide it into eight congruent disjoint cubes (called octants) until blocks are obtained of a uniform color or a predetermined level of decomposition is reached. Figure 18.11(a) is an example of a simple three-dimensional object whose region octree block decomposition is given in Fig. 18.11(b) and whose tree representation is given in Fig. 18.11(c).

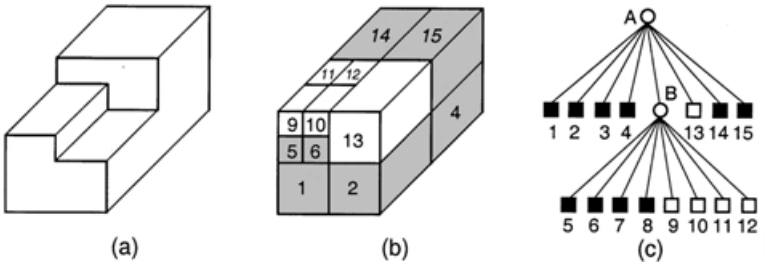


FIGURE 18.11 (a) Example three-dimensional object; (b) its region octree block decomposition; and (c) its tree representation.

The aggregation of cells into blocks in region quadtrees and region octrees is motivated, in part, by a desire to save space. Some of the decompositions have quite a bit of structure, thereby leading to inflexibility in choosing partition lines, etc. In fact, at times, maintaining the original image with an array access structure may be more effective from the standpoint of storage requirements. In the following, we point out some important implications of the use of these aggregations. In particular, we focus on the region quadtree and region octree. Similar results could also be obtained for the remaining block decompositions.

The aggregation of similarly valued cells into blocks has an important effect on the execution time of the algorithms that make use of the region quadtree. In particular, most algorithms that operate on images represented by a region quadtree are implemented by a preorder traversal of the quadtree, and thus their execution time is generally a linear function of the number of nodes in the quadtree. A key to the analysis of the execution time of quadtree algorithms is the **Quadtree Complexity Theorem** [32], which states that the number of nodes in a region quadtree representation for a simple polygon (i.e., with nonintersecting edges and without holes) is $O(p+q)$ for a $2^q \times 2^q$ image with perimeter p measured in terms of the

width of unit-sized cells (i.e., pixels). In all but the most pathological cases (e.g., a small square of unit width centered in a large image), the q factor is negligible, and thus, the number of nodes is $O(p)$.

The Quadtree Complexity Theorem also holds for three-dimensional data [42] (i.e., represented by a region octree) where perimeter is replaced by surface area, as well as for objects of higher dimensions d for which it is proportional to the size of the $(d-1)$-dimensional interfaces between these objects. The most important consequence of the Quadtree complexity theorem is that it means that most algorithms that execute on a region quadtree representation of an image, instead of one that simply imposes an array access structure on the original collection of cells, usually have an execution time that is proportional to the number of blocks in the image rather than the number of unit-sized cells. In its most general case, this means that the use of the region quadtree, with an appropriate access structure, in solving a problem in d-dimensional space will lead to a solution whose execution time is proportional to the $(d-1)$-dimensional space of the surface of the original d-dimensional image. On the other hand, use of the array access structure on the original collection of cells results in a solution whose execution time is proportional to the number of cells that comprise the image. Therefore, region quadtrees and region octrees act like dimension-reducing devices.

18.5 Rectangle Data

The rectangle data type lies somewhere between the point and region data types. It can also be viewed as a special case of the region data type in the sense that it is a region with only four sides. Rectangles are often used to approximate other objects in an image for which they serve as the minimum rectilinear enclosing object. For example, bounding rectangles are used in cartographic applications to approximate objects such as lakes, forests, hills, etc. In such a case, the approximation gives an indication of the existence of an object. Of course, the exact boundaries of the object are also stored; but they are only accessed if greater precision is needed. For such applications, the number of elements in the collection is usually small, and most often the sizes of the rectangles are of the same order of magnitude as the space from which they are drawn.

Rectangles are also used in VLSI design rule checking as a model of chip components for the analysis of their proper placement. Again, the rectangles serve as minimum enclosing objects. In this application, the size of the collection is quite large (e.g., millions of components) and the sizes of the rectangles are several orders of magnitude smaller than the space from which they are drawn.

It should be clear that the actual representation that is used depends heavily on the problem environment. At times, the rectangle is treated as the Cartesian product of two one-dimensional intervals with the horizontal intervals being treated in a different manner than the vertical intervals. In fact, the representation issue is often reduced to one of representing intervals. For example, this is the case in the use of the plane-sweep paradigm [47] in the solution of rectangle problems such as determining all pairs of intersecting rectangles. In this case, each interval is represented by its left and right endpoints. The solution makes use of two passes.

The first pass sorts the rectangles in ascending order on the basis of their left and right sides (i.e., x coordinate values) and forms a list. The second pass sweeps a vertical scan line through the sorted list from left to right halting at each one of these points, say p. At any instant, all rectangles that intersect the scan line are considered *active* and are the only ones whose intersection needs to be checked with the rectangle associated with p. This means that each time the sweep line halts, a rectangle either becomes active (causing it to be inserted in the set of active rectangles) or ceases to be active (causing it to be deleted from the set of active rectangles). Thus, the key to the algorithm is its ability to keep track of the active rectangles (actually just their vertical sides) as well as to perform the actual one-dimensional intersection test.

Data structures such as the segment tree [8], interval tree [14], and the priority search tree [41] can be used to organize the vertical sides of the active rectangles so that, for N rectangles and F intersecting

pairs of rectangles, the problem can be solved in $O(N \cdot \log_2 N + F)$ time. All three data structures enable intersection detection, insertion, and deletion to be executed in $O(\log_2 N)$ time. The difference between them is that the segment tree requires $O(N \cdot \log_2 N)$ space while the interval tree and the priority search tree only need $O(N)$ space.

The key to the use of the priority search tree to solve the rectangle intersection problem is that it treats each vertical side (y_B, y_T) as a point (x, y) in a two-dimensional space (i.e., it transforms the corresponding interval into a point as discussed in Section 18.1). The advantage of the priority search tree is that the storage requirements for the second pass only depend on the maximum number M of vertical sides that can be active at any one time. This is achieved by implementing the priority search tree as a red-black balanced binary tree [24], thereby guaranteeing updates in $O(\log_2 M)$ time. This also has an effect on the execution time of the second pass which is $O(N \cdot \log_2 M + F)$ instead of $O(N \cdot \log_2 N + F)$. Of course, the first pass which must sort the endpoints of the horizontal sides still takes $O(N \cdot \log_2 N)$ time for all three representations.

Most importantly, the priority search tree enables a more dynamic solution than either the segment or interval trees as only the endpoints of the horizontal sides need to be known in advance. On the other hand, for the segment and interval trees, the endpoints of both the horizontal and vertical sides must be known in advance. Of course, in all cases, all solutions based on the plane-sweep paradigm are inherently not dynamic as the paradigm requires that we examine all of the data one by one. Thus, the addition of even one new rectangle to the database forces the re-execution of the algorithm on the entire database.

In this chapter, we are primarily interested in dynamic problems. The data structures that are chosen for the collection of the rectangles are differentiated by the way in which each rectangle is represented. One representation discussed in Section 18.1 reduces each rectangle to a point in a higher dimensional space, and then treats the problem as if we have a collection of points [28]. Again, each rectangle is a Cartesian product of two one-dimensional intervals where the difference from its use with the plane-sweep paradigm is that each interval is represented by its centroid and extent. Each set of intervals in a particular dimension is, in turn, represented by a grid file [45], which is described in Section 18.2.

The second representation is region-based in the sense that the subdivision of the space from which the rectangles are drawn depends on the physical extent of the rectangle — not just one point. Representing the collection of rectangles, in turn, with a tree-like data structure has the advantage that there is a relation between the depth of node in the tree and the size of the rectangle(s) that is (are) associated with it. Interestingly, some of the region-based solutions make use of the same data structures that are used in the solutions based on the plane-sweep paradigm.

There are three types of region-based solutions currently in use. The first two solutions use the R-tree and the R^+-tree (discussed in Section 18.3) to store rectangle data (in this case the objects are rectangles instead of arbitrary objects). The third is a quadtree-based approach and uses the MX-CIF quadtree [34].

In the *MX-CIF quadtree,* each rectangle is associated with the quadtree node corresponding to the smallest block which contains it in its entirety. Subdivision ceases whenever a node's block contains no rectangles. Alternatively, subdivision can also cease once a quadtree block is smaller than a predetermined threshold size. This threshold is often chosen to be equal to the expected size of the rectangle [34]. For example, Fig. 18.12 is the MX-CIF quadtree for a collection of rectangles. Rectangles can be associated with both terminal and nonterminal nodes.

It should be clear that more than one rectangle can be associated with a given enclosing block, and thus, often we find it useful to be able to differentiate between them. This is done in the following manner [34]. Let P be a quadtree node with centroid (CX,CY), and let S be the set of rectangles that are associated with P. Members of S are organized into two sets according to their intersection (or collinearity of their sides) with the lines passing through the centroid of P's block—that is, all members of S that intersect the line $x = CX$ form one set and all members of S that intersect the line $y = CY$ form the other set.

If a rectangle intersects both lines (i.e., it contains the centroid of P's block), then we adopt the convention that it is stored with the set associated with the line through $x = CX$. These subsets are implemented as binary trees (really tries), which in actuality are one-dimensional analogs of the MX-CIF quadtree. For

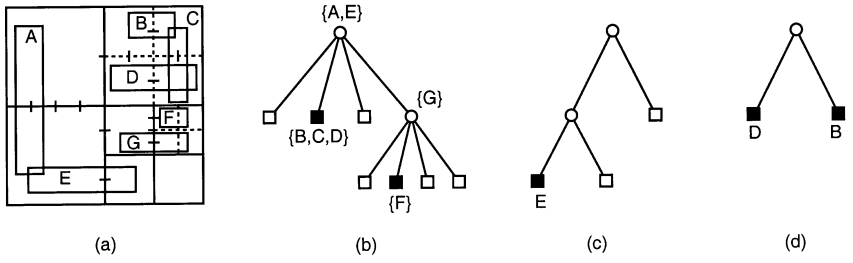


FIGURE 18.12 (a) Collection of rectangles and the block decomposition induced by the MX-CIF quadtree; (b) the tree representation of (a); (c) the binary trees for the y axes passing through the root of the tree in (b), and (d) the NE son of the root of the tree in (b).

example, Fig. 18.12(b) and Fig. 18.12(d) illustrate the binary trees associated with the y axes passing through the root and the NE son of the root, respectively, of the MX-CIF quadtree of Fig. 18.12(c). Interestingly, the MX-CIF quadtree is a two-dimensional analog of the interval tree described above. More precisely, the MX-CIF is a two-dimensional analog of the tile tree [40] which is a regular decomposition version of the interval tree. In fact, the tile tree and the one-dimensional MX-CIF quadtree are identical when rectangles are not allowed to overlap.

18.6 Line Data and Boundaries of Regions

Section 18.4 was devoted to variations on hierarchical decompositions of regions into blocks, an approach to region representation that is based on a description of the region's interior. In this section, we focus on representations that enable the specification of the boundaries of regions, as well as curvilinear data and collections of line segments. The representations are usually based on a series of approximations which provide successively closer fits to the data, often with the aid of bounding rectangles. When the boundaries or line segments have a constant slope (i.e., linear and termed *line segments* in the rest of this discussion), then an exact representation is possible.

There are several ways of approximating a curvilinear line segment. The first is by digitizing it and then marking the unit-sized cells (i.e., pixels) through which it passes. The second is to approximate it by a set of straight line segments termed a *polyline*. Assuming a boundary consisting of straight lines (or polylines after the first stage of approximation), the simplest representation of the boundary of a region is the polygon. It consists of vectors which are usually specified in the form of lists of pairs of x and y coordinate values corresponding to their start and end points. The vectors are usually ordered according to their connectivity. One of the most common representations is the chain code [21], which is an approximation of a polygon's boundary by use of a sequence of unit vectors in the four (and sometimes eight) principal directions.

Chain codes, and other polygon representations, break down for data in three dimensions and higher. This is primarily due to the difficulty in ordering their boundaries by connectivity. The problem is that in two dimensions connectivity is determined by ordering the boundary elements $e_{i,j}$ of boundary b_i of object o so that the end vertex of the vector v_j corresponding to $e_{i,j}$ is the start vertex of the vector v_{j+1} corresponding to $e_{i,j+1}$. Unfortunately, such an implicit ordering does not exist in higher dimensions as the relationship between the boundary elements associated with a particular object are more complex.

Instead, we must make use of data structures that capture the topology of the object in terms of its faces, edges, and vertices. The winged-edge data structure is one such representation that serves as the basis of the boundary model (also known as *BRep* [5]). Such representations are not discussed further here.

Polygon representations are very local. In particular, if we are at one position on the boundary, we don't know anything about the rest of the boundary without traversing it element by element. Thus, using such

representations, given a random point in space, it is very difficult to find the nearest line to it as the lines are not sorted. This is in contrast to hierarchical representations which are global in nature. They are primarily based on rectangular approximations to the data as well as on a regular decomposition in two dimensions. In the rest of this section, we discuss a number of such representations.

In Section 18.3 we already examined two hierarchical representations (i.e., the R-tree and the R^+-tree) that propagate object approximations in the form of bounding rectangles. In this case, the sides of the bounding rectangles had to be parallel to the coordinate axes of the space from which the objects are drawn. In contrast, the *strip tree* [4] is a hierarchical representation of a single curve that successively approximates segments of it with bounding rectangles that does not require that the sides be parallel to the coordinate axes. The only requirement is that the curve be continuous; it need not be differentiable.

The strip tree data structure consists of a binary tree whose root represents the bounding rectangle of the entire curve. The rectangle associated with the root corresponds to a rectangular strip, that encloses the curve, whose sides are parallel to the line joining the endpoints of the curve. The curve is then partitioned in two at one of the locations where it touches the bounding rectangle (these are not tangent points as the curve only needs to be continuous; it need not be differentiable). Each subcurve is then surrounded by a bounding rectangle and the partitioning process is applied recursively. This process stops when the width of each strip is less than a predetermined value.

In order to be able to cope with more complex curves such as those that arise in the case of object boundaries, the notion of a strip tree must be extended. In particular, closed curves and curves that extend past their endpoints require some special treatment. The general idea is that these curves are enclosed by rectangles which are split into two rectangular strips, and from now on the strip tree is used as before.

The strip tree is similar to the point quadtree in the sense that the points at which the curve is decomposed depend on the data. In contrast, a representation based on the region quadtree has fixed decomposition points. Similarly, strip tree methods approximate curvilinear data with rectangles of arbitrary orientation, while methods based on the region quadtree achieve analogous results by use of a collection of disjoint squares having sides of length power of two. In the following we discuss a number of adaptations of the region quadtree for representing curvilinear data.

The simplest adaptation of the region quadtree is the MX quadtree [32, 33]. It is built by digitizing the line segments and labeling each unit-sized cell (i.e., pixel) through which it passes as of type `boundary`. The remaining pixels are marked `WHITE` and are merged, if possible, into larger and larger quadtree blocks. Figure 18.13(a) is the MX quadtree for the collection of line segment objects in Fig. 18.5. A drawback of the MX quadtree is that it associates a thickness with a line. Also, it is difficult to detect the presence of a vertex whenever five or more line segments meet.

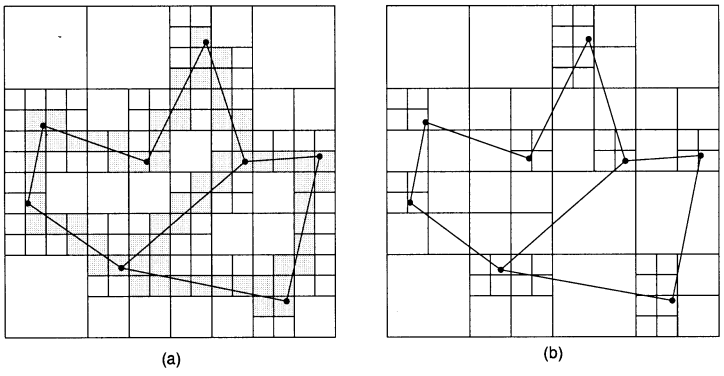


FIGURE 18.13 (a) MX quadtree and (b) edge quadtree for the collection of line segments of Fig.18.5.

The edge quadtree [57, 61] is a refinement of the MX quadtree based on the observation that the number of squares in the decomposition can be reduced by terminating the subdivision whenever the square contains a single curve that can be approximated by a single straight line. For example, Fig. 18.13(b) is the edge quadtree for the collection of line segment objects in Fig. 18.5. Applying this process leads to quadtrees in which long edges are represented by large blocks or a sequence of large blocks. However, small blocks are required in the vicinity of corners or intersecting edges. Of course, many blocks will contain no edge information at all.

The PM quadtree family [44, 54] (see also edge-EXCELL [59]) represents an attempt to overcome some of the problems associated with the edge quadtree in the representation of collections of polygons (termed *polygonal maps*). In particular, the edge quadtree is an approximation because vertices are represented by pixels. There are a number of variants of the PM quadtree. These variants are either vertex-based or edge-based. They are all built by applying the principle of repeatedly breaking up the collection of vertices and edges (forming the polygonal map) until obtaining a subset that is sufficiently simple so that it can be organized by some other data structure.

The PM_1 quadtree [54] is an example of a vertex-based PM quadtree. Its decomposition rule stipulates that partitioning occurs as long as a block contains more than one line segment unless the line segments are all incident at the same vertex which is also in the same block [e.g., Fig. 18.14(a)]. Given a polygonal map whose vertices are drawn from a grid (say $2^m \times 2^m$), and where edges are not permitted to intersect at points other than the grid points (i.e., vertices), it can be shown that the maximum depth of any leaf node in the PM_1 quadtree is bounded from above by $4m + 1$ [52]. This enables a determination of the maximum amount of storage that will be necessary for each node.

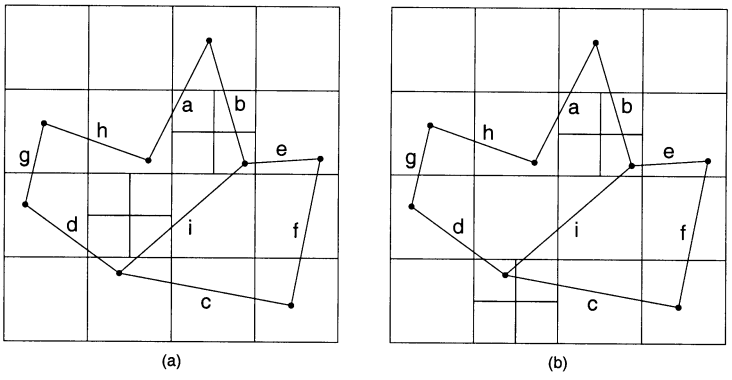


FIGURE 18.14 (a) PM_1 quadtree and (b) PMR quadtree for the collection of line segments of Fig. 18.5.

A similar representation has been devised for three-dimensional images (e.g., [3] and the references cited in [51]). The decomposition criteria are such that no node contains more than one face, edge, or vertex unless the faces all meet at the same vertex or are adjacent to the same edge. This representation is quite useful, since its space requirements for polyhedral objects are significantly smaller than those of a region octree.

The PMR quadtree [44] is an edge-based variant of the PM quadtree. It makes use of a probabilistic splitting rule. A node is permitted to contain a variable number of line segments. A line segment is stored in a PMR quadtree by inserting it into the nodes corresponding to all the blocks that it intersects. During this process, the occupancy of each node that is intersected by the line segment is checked to see if the insertion causes it to exceed a predetermined *splitting threshold.* If the splitting threshold is exceeded, then the node's block is split *once,* and only once, into four equal quadrants.

For example, Fig. 18.14(b) is the PMR quadtree for the collection of line segment objects in Fig. 18.5 with a splitting threshold value of 2. The line segments are inserted in alphabetic order (i.e., a–i). It should be clear that the shape of the PMR quadtree depends on the order in which the line segments are inserted. Note the difference from the PM_1 quadtree in Fig. 18.14(a)—that is, the NE block of the SW quadrant is decomposed in the PM_1 quadtree while the SE block of the SW quadrant is not decomposed in the PM_1 quadtree.

On the other hand, a line segment is deleted from a PMR quadtree by removing it from the nodes corresponding to all the blocks that it intersects. During this process, the occupancy of the node and its siblings is checked to see if the deletion causes the total number of line segments in them to be less than the predetermined splitting threshold. If the splitting threshold exceeds the occupancy of the node and its siblings, then they are merged and the merging process is reapplied to the resulting node and its siblings. Notice the asymmetry between the splitting and merging rules.

The PMR quadtree is very good for answering queries such as finding the nearest line to a given point [29, 30] (see [31] for an empirical comparison with hierarchical object representations such as the R-tree and R^+-tree). It is preferred over the PM_1 quadtree (as well as the MX and edge quadtrees) as it results in far fewer subdivisions. In particular, in the PMR quadtree there is no need to subdivide in order to separate line segments that are very "close" or whose vertices are very "close," which is the case for the PM_1 quadtree. This is important, since four blocks are created at each subdivision step. Thus, when many subdivision steps that occur in the PM_1 quadtree result in creating many empty blocks, the storage requirements of the PM_1 quadtree will be considerably higher than those of the PMR quadtree. Generally, as the splitting threshold is increased, the storage requirements of the PMR quadtree decrease while the time necessary to perform operations on it will increase.

Using a random image model and geometric probability, it has been shown [39], theoretically and empirically using both random and real map data, that for sufficiently high values of the splitting threshold (i.e., ≥ 4), the number of nodes in a PMR quadtree is asymptotically proportional to the number of line segments and is independent of the maximum depth of the tree. In contrast, using the same model, the number of nodes in the PM_1 quadtree is a product of the number of lines and the maximal depth of the tree (i.e., n for a $2^n \times 2^n$ image). The same experiments and analysis for the MX quadtree confirmed the results predicted by the Quadtree complexity theorem (see Section 18.4), which is that the number of nodes is proportional to the total length of the line segments.

Observe that although a bucket in the PMR quadtree can contain more line segments than the splitting threshold, this is not a problem. In fact, it can be shown [51] that the maximum number of line segments in a bucket is bounded by the sum of the splitting threshold and the depth of the block (i.e., the number of times the original space has been decomposed to yield this block).

18.7 Research Issues and Summary

A review has been presented of a number of representations of multidimensional data. Our focus has been on multidimensional spatial data with extent rather than just multidimensional point data. Moreover, the multidimensional data was not restricted to locational attributes in that the handling of nonlocational attributes for point data was also described. There has been a particular emphasis on hierarchical representations. Such representations are based on the "divide-and-conquer" problem-solving paradigm. They are of interest because they enable focusing computational resources on the interesting subsets of data. Thus, there is no need to expend work where the payoff is small. Although many of the operations for which they are used can often be performed equally as efficiently, or more so, with other data structures, hierarchical data structures are attractive because of their conceptual clarity and ease of implementation.

When the hierarchical data structures are based on the principle of regular decomposition, we have the added benefit that different data sets (often of differing types) are in registration. This means that they are partitioned in known positions that are often the same or subsets of one another for the different data

sets. This is true for all the features including regions, points, rectangles, lines, volumes, etc. This means that a query such as "finding all cities with more than 20,000 inhabitants in wheat growing regions within 30 miles of the Mississippi River" can be executed by simply overlaying the region (crops), point (i.e., cities), and river maps even though they represent data of different types. Alternatively, we may extract regions such as those within 30 miles of the Mississippi River. Such operations find use in applications involving spatial data such as geographic information systems.

Current research in multidimensional representations is highly application-dependent in the sense that the work is driven by the application. Many of the recent developments have been motivated by the interaction with databases. The choice of a proper representation plays a key role in the speed with which responses are provided to queries. Knowledge of the underlying data distribution is also a factor and research is ongoing to make use of this information in the process of making a choice. Most of the initial applications in which the representation of multidimensional data has been important have involved spatial data of the kind described in this chapter. Such data is intrinsically of low dimensionality (i.e., two and three). Future applications involve higher dimensional data for applications such as image databases where the data are often points in feature space. The incorporation of the time dimension is also an important issue that confronts many database researchers.

18.8 Defining Terms

Bintree: A regular decomposition k-d tree for region data.

Boundary-based representation: A representation of a region that is based on its boundary.

Bucketing methods: Data organization methods that decompose the space from which spatial data is drawn into regions called buckets. Some conditions for the choice of region boundaries include the number of objects that they contain or on their spatial layout (e.g., minimizing overlap or coverage).

Fixed-grid method: Space decomposition into rectangular cells by overlaying a grid on it. If the cells are congruent (i.e., of the same width, height, etc.), then the grid is said to be uniform.

Interior-based representation: A representation of a region that is based on its interior (i.e., the cells that comprise it).

K-d tree: General term used to describe space decomposition methods that proceed by recursive decomposition across a single dimension at a time of the space containing the data until some condition is met such as that the resulting blocks contain no more than b objects (e.g., points, lines, etc.) or that the blocks are homogeneous. The k-d tree is usually a data structure for points which cycles through the dimensions as it decomposes the underlying space.

Multidimensional data: Data that has several attributes. It includes records in a database management system, locations in space, and also spatial entities that have extent such as lines, regions, volumes, etc.

Octree: A quadtree-like decomposition for three dimensional data.

Quadtree: General term used to describe space decomposition methods that proceed by recursive decomposition across all the dimensions (technically two dimensions) of the space containing the data until some condition is met such as that the resulting blocks contain no more than b objects (e.g., points, lines, etc.) or that the blocks are homogeneous (e.g., region data). The underlying space is not restricted to two-dimensions although this is the technical definition of the term. The result is usually a disjoint decomposition of the underlying space.

Quadtree complexity theorem: The number of nodes in a quadtree region representation for a simple polygon (i.e., with nonintersecting edges and without holes) is $O(p+q)$ for a $2^q \times 2^q$ image with perimeter p measured in pixel widths. In most cases, q is negligible, and thus, the number of nodes is proportional to the perimeter. It also holds for three-dimensional data

where the perimeter is replaced by surface area, and in general for d-dimensions where instead of perimeter we have the size of the $(d-1)$-dimensional interfaces between the d-dimensional objects.

R-tree: An object hierarchy where associated with each element of the hierarchy is the minimum bounding rectangle of the union of the minimum bounding rectangles of the elements immediately below it. The elements at the deepest level of the hierarchy are groups of spatial objects. The result is usually a nondisjoint decomposition of the underlying space. The objects are aggregated on the basis of proximity and with the goal of minimizing coverage and overlap.

Regular decomposition: A space decomposition method that partitions the underlying space by recursively halving it across the various dimensions instead of permitting the partitioning lines to vary.

Acknowledgments

The assistance of Gisli Hjaltason in the preparation of the figures is greatly appreciated.

References

[1] Aref, W.G. and Samet. H., Uniquely reporting spatial objects: yet another operation for comparing spatial data structures. In *Proceedings of the Fifth International Symposium on Spatial Data Handling,* 178–189, Charleston, SC, Aug. 1992.

[2] Aref, W.G. and Samet, H., Hashing by proximity to process duplicates in spatial databases. In *Proceedings of the Third International Conference on Information and Knowledge Management,* 347–354, Gaithersburg, MD, ACM Press. Dec. 1994.

[3] Ayala, D., Brunet, P., Juan, R., and Navazo, I., Object representation by means of nonminimal division quadtrees and octrees. *ACM Transactions on Graphics,* 4(1), 41–59, Jan. 1985.

[4] Ballard, D.H., Strip trees: a hierarchical representation for curves. *Communications of the ACM,* 24(5), 310–321, May 1981. (Also corrigendum, *Communications of the ACM,* 25, 3, Mar. 1982, 213.)

[5] Baumgart, B.G., A polyhedron representation for computer vision. In *Proceedings of the National Computer Conference 44,* 589–596, Anaheim, CA, May 1975.

[6] Beckmann, N., Kriegel, H.P., Schneider, R., and Seeger, B., The R^*-tree: an efficient and robust access method for points and rectangles. In *Proceedings of the ACM SIGMOD Conference,* 322–331, Atlantic City, NJ, Jun. 1990.

[7] Bentley, J.L., Multidimensional binary search trees used for associative searching. *Communications of the ACM,* 18(9), 509–517, Sep. 1975.

[8] Bentley, J.L., Algorithms for Klee's rectangle problems. (unpublished), 1977.

[9] Bentley, J.L. and Friedman, J.H., Data structures for range searching. *ACM Computing Surveys,* 11(4), 397–409, Dec. 1979.

[10] Bentley, J.L. and Mauer, H.A., Efficient worst-case data structures for range searching. *Acta Informatica,* 13, 155–168, 1980.

[11] Bentley, J.L., Stanat, D.F., and Williams, Jr., E.H., The complexity of finding fixed-radius near neighbors. *Information Processing Letters,* 6(6), 209–212, Dec. 1977.

[12] Comer, D., The ubiquitous B-tree. *ACM Computing Surveys,* 11(2), 121–137, Jun. 1979.

[13] de Berg, M., van Kreweld, M., Overmars, M., and Schwarzkopf, O., *Computational geometry: algorithms and applications.* Springer-Verlag, Berlin, Germany, 1997.

[14] Edelsbrunner, W., Dynamic rectangle intersection searching. Institute for Information Processing 47, Technical University of Graz, Graz, Austria, Feb. 1980.

[15] Edelsbrunner, H., A note on dynamic range searching. *Bulletin of the EATCS,* (15), 34–40, Oct. 1981.
[16] Edelsbrunner, H., *Algorithms in Combinatorial Geometry.* Springer-Verlag, Berlin, 1987.
[17] Finkel, R.A. and Bentley, J.L., Quad trees: a data structure for retrieval on composite keys. *Acta Informatica,* 4(1), 1–9, 1974.
[18] Frank, A.U. and Barrera, R., The fieldtree: a data structure for geographic information systems. In *Design and Implementation of Large Spatial Databases — First Symposium, SSD'89,* Buchmann, A., Günther, O., Smith, T.R., and Wang, Y.F., Eds., 29–44, Santa Barbara, Jul. 1989. (Also Springer-Verlag Lecture Notes in Computer Science 409.)
[19] Franklin, W.R., Adaptive grids for geometric operations. *Cartographica,* 21(2&3), 160–167, Summer & Autumn, 1984.
[20] Fredkin, E., Trie memory. *Communications of the ACM,* 3(9), 490–499, Sep. 1960.
[21] Freeman, H., Computer processing of line-drawing images. *ACM Computing Surveys,* 6(1), 57–97, Mar. 1974.
[22] Freeston, M., The BANG file: a new kind of grid file. In *Proceedings of the ACM SIGMOD Conference,* 260–269, San Francisco, CA, May 1987.
[23] Fuchs, H., Kedem, Z.M., and Naylor, B.F., On visible surface generation by a priori tree structures. *Computer Graphics,* 14(3), 124–133, Jul. 1980. (Also *Proceedings of the SIGGRAPH'80 Conference,* Seattle, WA, Jul. 1980).
[24] Guibas, L.J. and Sedgewick, R., A dichromatic framework for balanced trees. In *Proceedings of the Nineteenth Annual IEEE Symposium on the Foundations of Computer Science,* 8–21, Ann Arbor, MI, Oct. 1978.
[25] Günther, O., *Efficient structures for geometric data management.* Ph.D. Thesis, University of California at Berkeley, Berkeley, CA, 1987. (Also Lecture Notes in Computer Science 337, Springer-Verlag, Berlin, 1988).
[26] Guttman, A., R-trees: a dynamic index structure for spatial searching. In *Proceedings of the ACM SIGMOD Conference,* 47–57, Boston, MA, Jun. 1984.
[27] Henrich, A., Six, H.W., and Widmayer, P., The LSD tree: spatial access to multidimensional point and non-point data. In *Proceedings of the Fifteenth International Conference on Very Large Data Bases,* Apers, P.M.G. and Wiederhold, G., Eds., 45–53, Amsterdam, The Netherlands, Aug. 1989.
[28] Hinrichs, K. and Nievergelt, J., The grid file: a data structure designed to support proximity queries on spatial objects. In *Proceedings of the WG'83 (International Workshop on Graphtheoretic Concepts in Computer Science),* Nagl, M. and Perl, J., Eds., 100–113, Linz, Austria, 1983. Trauner Verlag.
[29] Hjaltason, G.R. and Samet, H., Ranking in spatial databases. In *Advances in Spatial Databases — Fourth International Symposium, SSD'95,* Egenhofer, M.J. and Herring, J.R., Eds., 83–95, Portland, ME, Aug. 1995. (Also Springer-Verlag Lecture Notes in Computer Science 951).
[30] Hoel, E.G. and Samet, H., Efficient processing of spatial queries in line segment databases. In *Advances in Spatial Databases — Second Symposium, SSD'91,* Günther, O. and Schek, H.J., Eds., 237–256, Zurich, Switzerland, Aug. 1991. (Also Springer-Verlag Lecture Notes in Computer Science 525).
[31] Hoel, E.G. and Samet, H., A qualitative comparison study of data structures for large line segment databases. In *Proceedings of the ACM SIGMOD Conference,* 205–214, San Diego, CA, Jun. 1992.
[32] Hunter, G.M., *Efficient computation and data structures for graphics.* Ph.D. Thesis, Princeton University, Princeton, NJ, 1978.
[33] Hunter, G.M. and Steiglitz, K., Operations on images using quad trees. *IEEE Transactions on Pattern Analysis and Machine Intelligence,* 1(2), 145–153, Apr. 1979.

[34] Kedem, G., The quad-CIF tree: a data structure for hierarchical on-line algorithms. In *Proceedings of the Nineteenth Design Automation Conference,* 352–357, Las Vegas, Jun. 1982.
[35] Klinger, A., Patterns and search statistics. In *Optimizing Methods in Statistics,* Rustagi, J.S., Ed., 303–337. Academic Press, New York, 1971.
[36] Knowlton, K., Progressive transmission of grey-scale and binary pictures by simple efficient, and lossless encoding schemes. *Proceedings of the IEEE,* 68(7), 885–896, Jul. 1980.
[37] Knuth, D.E., *The Art of Computer Programming vol. 3, Sorting and Searching.* Addison-Wesley, Reading, MA, 1973.
[38] Knuth, D.E., Big omicron and big omega and big theta. *SIGACT News,* 8(2), 18–24, Apr.-Jun. 1976.
[39] Lindenbaum, M. and Samet, H., A probabilistic analysis of trie-based sorting of large collections of line segments. Computer Science Department TR-3455, University of Maryland, College Park, MD, Apr. 1995.
[40] McCreight, E.M., Efficient algorithms for enumerating intersecting intervals and rectangles. Technical Report CSL-80-09, Xerox Palo Alto Research Center, Palo Alto, CA, Jun. 1980.
[41] McCreight, E.M., Priority search trees. *SIAM Journal on Computing,* 14(2), 257–276, May 1985.
[42] Meagher, D., Octree encoding: a new technique for the representation, the manipulation, and display of arbitrary 3-d objects by computer. Electrical and Systems Engineering IPL-TR-80-111, Rensselaer Polytechnic Institute, Troy, NY, Oct. 1980.
[43] Meagher, D., Geometric modeling using octree encoding. *Computer Graphics and Image Processing,* 19(2), 129–147, Jun. 1982.
[44] Nelson, R.C. and Samet, H., A consistent hierarchical representation for vector data. *Computer Graphics,* 20(4), 197–206, Aug. 1986. (Also *Proceedings of the SIGGRAPH'86 Conference,* Dallas, Aug. 1986).
[45] Nievergelt, J., Hinterberger, H., and Sevcik, K.C., The grid file: an adaptable, symmetric multikey file structure. *ACM Transactions on Database Systems,* 9(1), 38–71, Mar. 1984.
[46] Orenstein, J.A., Multidimensional tries used for associative searching. *Information Processing Letters,* 14(4), 150–157, Jun, 1982.
[47] Preparata, F.P. and Shamos, M.I., *Computational Geometry: An Introduction.* Springer-Verlag, New York, 1985.
[48] Requicha, A.A.G., Representations of rigid solids: theory, methods, and systems. *ACM Computing Surveys,* 12(4), 437–464, Dec. 1980.
[49] Robinson, J.T., The *k–d–b*–tree: a search structure for large multidimensional dynamic indexes. In *Proceedings of the ACM SIGMOD Conference,* 10–18, Ann Arbor, MI, Apr. 1981.
[50] Samet, H., *Applications of Spatial Data Structures: Computer Graphics, Image Processing, and GIS.* Addison-Wesley, Reading, MA, 1990.
[51] Samet, H., *The Design and Analysis of Spatial Data Structures.* Addison-Wesley, Reading, MA, 1990.
[52] Samet, H., Shaffer, C.A., and Webber, R.E., Digitizing the plane with cells of non–uniform size. *Information Processing Letters,* 24(6), 369–375, Apr. 1987.
[53] Samet, H. and Tamminen, M., Efficient component labeling of images of arbitrary dimension represented by linear bintrees. *IEEE Transactions on Pattern Analysis and Machine Intelligence,* 10(4), 579–586, Jul. 1988.
[54] Samet, H. and Webber, R.E., Storing a collection of polygons using quadtrees. *ACM Transactions on Graphics,* 4(3), 182–222, Jul. 1985. (Also *Proceedings of Computer Vision and Pattern Recognition 83,* Washington, DC, Jun. 1983, 127–132, and University of Maryland Computer Science TR–1372).

[55] Seeger, B. and Kriegel, H.P., The buddy-tree: an efficient and robust access method for spatial data base systems. In *Proceedings of the 16th International Conference on Very Large Databases (VLDB),* McLeod, D., Sacks-Davis, R., and Schek, H., Eds., 590–601, Brisbane, Australia, Aug. 1990.

[56] Sellis, T., Roussopoulos, N., and Faloutsos, C., The R^+–tree: a dynamic index for multi–dimensional objects. In *Proceedings of the 13th International Conference on Very Large Databases (VLDB),* Stocker, P.M. and Kent, W., Eds., 71–79, Brighton, England, Sep. 1987. (Also University of Maryland Computer Science TR–1795).

[57] Shneier, M., Two hierarchical linear feature representations: edge pyramids and edge quadtrees. *Computer Graphics and Image Processing,* 17(3), 211–224, Nov. 1981.

[58] Stonebraker, M., Sellis, T., and Hanson, E., An analysis of rule indexing implementations in data base systems. In *Proceedings of the First International Conference on Expert Database Systems,* 353–364, Charleston, SC, Apr. 1986.

[59] Tamminen, M., The EXCELL method for efficient geometric access to data. *Acta Polytechnica Scandinavica,* 1981. (Mathematics and Computer Science Series No. 34).

[60] Tamminen, M., Comment on quad– and octtrees. *Communications of the ACM,* 27(3), 248–249, Mar. 1984.

[61] Warnock, J.E., A hidden surface algorithm for computer generated half tone pictures. Computer Science Department TR 4–15, University of Utah, Salt Lake City, Jun. 1969.

Further Information

It is impossible to give a complete enumeration of where research on multidimensional data structures is published, since it is often mixed with the application. Hands-on experience with some of the representations described in this chapter can be obtained by looking at the JAVA applets on `http://www.cs.umd.edu/~hjs/quadtree/index.html`. Multidimensional spatial data is covered in the texts by Samet [50, 51]. Their perspective is one from computer graphics, image processing, geographic information systems (GIS), databases, solid modeling, as well as VLSI design and computational geometry. A more direct computational geometry perspective can be found in the books by Edelsbrunner [16], Preparata and Shamos [47], and Overmars et al. [13].

New developments in the field of multidimensional data structures are reported in many different conferences, again since it is so application-driven. Some good starting pointers from the GIS perspective are the Symposium on Spatial Databases and the International Workshop on Spatial Data Handling, which are held in alternating years. From the standpoint of computational geometry, the annual ACM Symposium on Computational Geometry and the annual ACM-SIAM Symposium on Discrete Algorithms are good sources. From the perspective of databases, the annual ACM Conference on the Management of Data (SIGMOD) and the Very Large Database Conference (VLDB) usually contain a few papers dealing with the application of such representation. Other useful sources are the proceedings of the annual SIGGRAPH Conference.

Journals where such research appears are as varied as the applications. Theoretical results can be found in *SIAM Journal of Computing* while those from the GIS perspective may be found in a new journal called *GeoInformatica.* Many related articles are also found in the computer graphics and computer vision journals such as *ACM Transactions on Graphics,* the old *Computer Vision, Graphics and Image Processing,* which has been renamed *Graphical Models and Image Processing* and *Image Understanding,* and *IEEE Transactions on Pattern Analysis and Machine Intelligence.*

19

Computational Geometry I

D.T. Lee

Northwestern University

19.1 Introduction

Computational geometry, since its inception [42] in 1975, has received a great deal of attention from researchers in the area of design and analysis of algorithms. It has evolved into a discipline of its own. It is concerned with the computational complexity of geometric problems that arise in various disciplines such as pattern recognition, computer graphics, geographical information system, computer vision, **CAD/CAM,** robotics, VLSI layout, operations research, statistics, etc. In contrast with the classical approach to proving mathematical theorems about geometry-related problems, this discipline emphasizes the computational aspect of these problems and attempts to exploit the underlying geometric properties possible, e.g., the metric space, to derive efficient algorithmic solutions.

An objective of this discipline in the theoretical context is to study the computational complexity (giving lower bounds) of geometric problems, and to devise efficient algorithms (giving upper bounds) whose complexity preferably *matches* the lower bounds. That is, not only are we interested in the *intrinsic* difficulty of geometric computational problems under a certain computation model, but we are also concerned with the algorithmic solutions that are efficient or provably optimal in the worst or average case. In this regard, the **asymptotic time** (or **space**) **complexity** of an algorithm, i.e., the behavior of an algorithm, as the input size approaches infinity, is of interest. Due to its applications to various science and engineering related disciplines, researchers in this field have begun to address the *efficacy* of the algorithms, the issues concerning *robustness* and *numerical stability* [25, 50], and the actual running times of their implementations.

In this and the following chapter we concentrate mostly on the theoretical development of this field in the context of sequential computation, and discuss a number of typical topics and the algorithmic

approaches. We will adopt the *real* RAM (random access machine) model of computation in which all arithmetic operations, comparisons, kth-root, exponential or logarithmic functions take unit time.

19.2 Convex Hull

The convex hull of a set of points in $\Re^k$ is the most fundamental problem in computational geometry. Given is a set of points in $\Re^k$, and we are interested in computing its convex hull, which is defined to be the smallest convex set containing these points. There are two ways to represent a convex hull. An *implicit* representation is to list all the **extreme points,** whereas an *explicit* representation is to list all the extreme d-faces of dimensions $d = 0, 1, \ldots, k-1$. Thus, the complexity of any convex hull algorithm would have two parts, computation part and the output part. An algorithm is said to be *output-sensitive* if its complexity depends on the size of output.

Convex Hulls in Two and Three Dimensions

For an arbitrary set of n points in two and three dimensions, we can compute the convex hull using the *Graham scan, gift-wrapping* method, or *divide-and-conquer* paradigm, which are briefly described below.

Note that the convex hull of an arbitrary set of points in two dimensions is a convex polygon. We'll describe algorithms that compute the *upper hull* of S, since the convex hull is just the union of the *upper* and *lower hulls.* Let v_0 denote the point with minimum x-coordinate; if there are more than one, pick the one with the maximum y coordinate. Let v_{n-1} be similarly defined except that it denotes the point with the maximum x-coordinate. In two dimensions, the upper hull consists of two vertical lines passing through v_0 and v_{n-1} respectively and a sequence of edges, known as a *polygonal chain,* $\mathcal{C} = \{\overline{v_{j_{i-1}}, v_{j_i}} \mid i = 1, 2, \ldots, k\}$, where $v_{j_0} = v_0$ and $v_{j_k} = v_{n-1}$, such that the entire set S of points lies on one side of the lines $\mathcal{L}_i$ containing each edge $\overline{v_{j_{i-1}}, v_{j_i}}$. See Fig. 19.1(a) for an illustration of the upper hull. The lower hull is similarly defined.

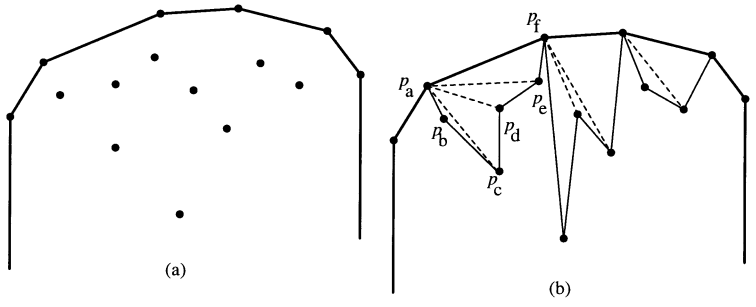


FIGURE 19.1 The upper hull of a set of points.

The *Graham scan* computes the convex hull by (i) sorting the input set of points in ascending order of their x-coordinates (in case of ties, in ascending order of their y-coordinates), (ii) connecting these points into a polygonal chain P stored as a doubly linked list L, and (iii) performing a linear scan to compute the upper hull of the polygon [42].

The triple (v_i, v_j, v_k) of points is said to form a *right* turn if and only if the determinant

$$\begin{vmatrix} x_i & y_i & 1 \\ x_j & y_j & 1 \\ x_k & y_k & 1 \end{vmatrix} < 0 ,$$

where (x_i, y_i) are the x- and y-coordinates of v_i. If the determinant is positive, then the triple (v_i, v_j, v_k)

of points is said to form a *left* turn. The points v_i, v_j, and v_k are *collinear* if the determinant is zero. This is also known as the **side test,** determining on which side of the line defined by points v_i and v_j the point v_k lies.

It is obvious that when we scan points in L in ascending order of x-coordinate, the middle point of a triple (v_i, v_j, v_k) that does not form a right turn is not on the upper hull and can be deleted. The following is the algorithm.

ALGORITHM GRAHAM_SCAN

Input: A set S of points sorted in lexicographically ascending order of their (x, y)-coordinate values.

Output: A sorted list L of points in ascending x-coordinates.

```
begin
    if (|S| == 2) return (v0, vn−1);
    i = 0; vn−1 =next(vn−1); /* set sentinel */
    pa = v0; pb =next(pa), pc =next(pb);
    while (pb ≠ vn−1) do
        if (pa, pb, pc) forms a right turn
        then begin /* advance */
                pa = pb; pb = pc;
                pc = next(pb);
             end
        else begin /* backtrack */
                delete pb;
                if (pa ≠ v0)
                then pa = prev(pa);
                pb = next(pa); pc = next(pb);
             end
    pt =next(v0);
    L = {v0, pt};
    while (pt ≠ vn−1) do
        begin
            pu =next(pt);
            L = L ∪ {pt, pu};
            pt = pu;
        end;
    return (L);
end.
```

Step (i) being the dominating step, ALGORITHM GRAHAM_SCAN, takes $O(n \log n)$ time. Figure 19.1(b) shows the initial list L and vertices not on the upper-hull are removed from L. For example, p_b is removed since (p_a, p_b, p_c) forms a left turn; p_c is removed since (p_a, p_c, p_d) forms a left turn; p_d, and p_e are removed for the same reason.

One can also use the *gift-wrapping* technique to compute the upper hull. Starting with a vertex that is known to be on the upper hull, say the point $v_0 = v_{i_0}$. We sweep clockwise the half-line emanating from v_0 in the direction of the positive y-axis. The first point v_{i_1} this half-line hits will be the next point on the upper hull. We then march to v_{i_1}, repeat the same process by sweeping clockwise the half-line emanating from v_{i_1} in the direction from v_{i_0} to v_{i_1}, and find the next vertex v_{i_2}. This process terminates when we reach v_{n-1}. This is similar to wrapping an object with a *rope.* Finding the next vertex takes time proportional to the number of points not yet known to be on the upper hull. Thus, the total time spent

is $O(n\mathcal{H})$, where $\mathcal{H}$ denotes the number of points on the upper hull. The gift-wrapping algorithm is output-sensitive, and is more efficient than the ALGORITHM GRAHAM_SCAN if the number of points on the upper hull is small, i.e., $o(\log n)$.

One can also compute the upper hull recursively by divide-and-conquer. This method is more amenable to parallelization. The divide-and-conquer paradigm consists of the following steps.

ALGORITHM UPPER_HULL_D&C (*2d-Point S*)

Input: A set S of points.

Output: A sorted list L of points in ascending x-coordinates.

1. If $|S| \leq 3$, compute the upper hull UH(S) explicitly and **return** (UH(S)).
2. Divide S by a vertical line $\mathcal{L}$ into two approximately equal subsets S_l and S_r such that S_l and S_r lie, respectively to the left and to the right of $\mathcal{L}$.
3. UH(S_l) = Upper_Hull_D&C(S_l).
4. UH(S_r) = Upper_Hull_D&C(S_r).
5. UH(S) =**Merge**($UH(S_l), UH(S_r)$).
6. **return** (UH(S)).

The key step is the **Merge** of two upper hulls, each of which is the solution to a subproblem derived from the recursive step. These two upper hulls are separated by a vertical line $\mathcal{L}$. The **Merge** step basically calls for computation of a common tangent, called *bridge* over line $\mathcal{L}$, of these two upper hulls (Fig. 19.2).

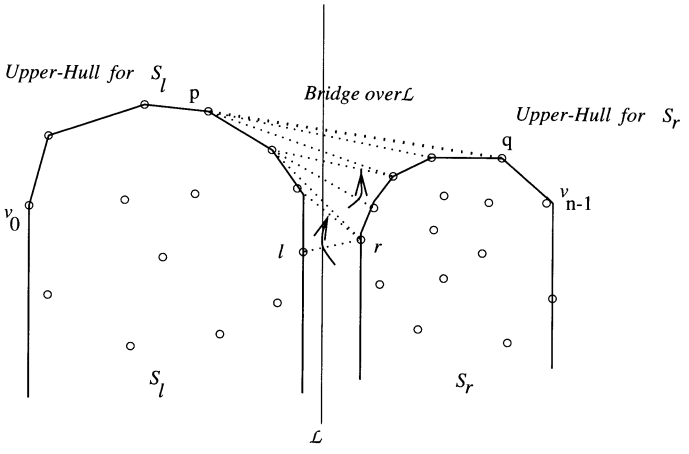


FIGURE 19.2 The bridge $\overline{p, q}$ over the vertical line $\mathcal{L}$.

The computation of the *bridge* begins with a segment connecting the rightmost point l of the left upper hull to the leftmost point r of the right upper hull, resulting in a sorted list L. Using the *Graham scan* one can obtain in *linear* time the two endpoints of the bridge, ($\overline{p, q}$ shown in Fig. 19.2), such that the entire set of points lies on one side of the line, called *supporting line,* containing the bridge. The running time of the divide-and-conquer algorithm is easily shown to be $O(n \log n)$ since the merge step can be done in $O(n)$ time.

A more sophisticated output-sensitive and *optimal* algorithm which runs in $O(n \log \mathcal{H})$ time has been developed by Kirkpatrick and Seidel [35]. It is based on a variation of the divide-and-conquer paradigm, called **divide-and-marriage-before-conquest** method. It has been shown to be asymptotically optimal; a lower bound proof of $\Omega(n \log \mathcal{H})$ can be found in [35]. The main idea in achieving the optimal result is that

of eliminating *redundant* computations. Observe that in the divide-and-conquer approach after the bridge is obtained, some vertices belonging to the left and right upper hulls that are *below* the bridge are deleted. Had we known that these vertices are not on the final hull, we could have saved time without computing them. Kirkpatrick and Seidel capitalized on this concept and introduced the **marriage-before-conquest** principle putting **Merge** step before the two recursive calls.

The divide-and-conquer scheme can be easily generalized to three dimensions. The merge step in this case calls for computing common supporting faces that *wrap* two recursively computed convex polyhedra. It is observed by Preparata and Shamos [42] that the common supporting faces are computed from connecting two *cyclic* sequences of edges, one on each polyhedron (Fig. 19.3). See [2] for a characterization of the two cycles of *seam* edges. The computation of these supporting faces can be accomplished in linear time, giving rise to an $O(n \log n)$ time algorithm. By applying the marriage-before-conquest principle Edelsbrunner and Shi [23] obtained an $O(n \log^2 \mathcal{H})$ algorithm.

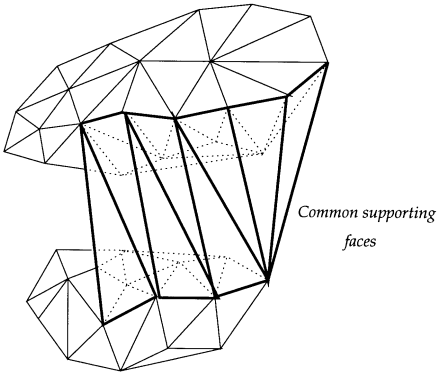


FIGURE 19.3 Common supporting faces of two disjoint convex polyhedra.

The gift-wrapping approach for computing the convex hull in three dimensions would mimic the process of wrapping a gift with a piece of paper. One starts with a plane supporting S, i.e., a plane determined by three points of S such that the entire set of points lie on one side. In general, the supporting face is a triangle $\Delta(a, b, c)$. Pivoting at an edge, say (a, b) of this triangle, one rotates the plane in space until it hits a third point v, thereby determining another supporting face $\Delta(a, b, v)$. This process repeats until the entire set of points are *wrapped* by a collection of supporting faces. These supporting faces are called 2-faces, the edges common to two supporting faces, 1-faces, and the vertices (or extreme points) common to 2-faces and 1-faces are called 0-faces. The gift-wrapping method has a running time of $O(n\mathcal{H})$, where $\mathcal{H}$ is the total number of i-faces, $i = 0, 1, 2$.

The following optimal output-sensitive algorithm that runs in $O(n \log \mathcal{H})$ time in two dimensions is due to Chan [10]. A similar algorithm for three dimensions can be obtained. It is a modification of the *gift-wrapping* method, (also known as the Jarvis' March method,) and uses a *grouping* technique.

ALGORITHM 2DHULL(S)

1. For $i = 1, 2, \ldots$ do
2. $\quad P \leftarrow$ HULL2D($S, \mathcal{H}_0, \mathcal{H}_0$), where $\mathcal{H}_0 = \min\{2^{2^i}, n\}$
3. $\quad$ If $P \neq nil$ then return P.

FUNCTION HULL2D($S, m, \mathcal{H}_0$)

1. Partition S into subsets $S_1, S_2, \ldots, S_{\lceil \frac{n}{m} \rceil}$, each of size at most m
2. For $i = 1, 2, \ldots, \lceil \frac{n}{m} \rceil$ do

3. Compute $CH(S_i)$ and preprocess it in a suitable data structure
4. $p_0 \leftarrow (0, -\infty)$, $p_1 \leftarrow$ the rightmost point of S
5. For $j = 1, 2, \ldots, \mathcal{H}_0$ do
6. For $i = 1, 2, \ldots, \lceil \frac{n}{m} \rceil$ do
7. Compute a point $q_i \in S_i$ that maximizes $\angle p_{j-1} p_j q_i$
8. $p_{j+1} \leftarrow$ a point q from $\{q_1, \ldots, q_{\lceil \frac{n}{m} \rceil}\}$ maximizing $\angle p_{j-1} p_j q$
9. If $p_{j+1} = p_1$ then return list $(p_1, \ldots, p_j)$
10. return nil

Let us analyze the complexity of the algorithm. In Step 2, we use an $O(m \log m)$ time algorithm for computing the convex hull for each subset of m points, e.g., Graham's scan for S in two dimensions, and Preparata–Hong algorithm for S in three dimensions. Thus, it takes $O((\frac{n}{m})m \log m) = O(n \log m)$ time. In Step 5 we build a suitable data structure that supports the computation of the supporting vertex or supporting face in logarithmic time. In two dimensions we can use an array that stores the vertices on the convex hull in say, clockwise, order. In three dimensions we use Dobkin–Kirkpatrick hierarchical representation of the faces of the convex hull [20]. Thus, Step 5 takes $\mathcal{H}_0(\frac{n}{m})O(\log m)$ time. Setting $m = \mathcal{H}_0$ gives an $O(n \log \mathcal{H}_0)$ time. Note that setting $m = 1$ we have the Jarvis' March, and setting $m = n$ the two dimensional convex hull algorithm degenerates to the Graham's Scan. Since we do not know $\mathcal{H}$ in advance, we use in Step 2 of ALGORITHM 2DHULL(S) a sequence $\mathcal{H}_i = 2^{2^i}$ such that $\mathcal{H}_1 + \ldots \mathcal{H}_{k-1} < \mathcal{H} \leq \mathcal{H}_1 + \ldots \mathcal{H}_k$ to guess it. The total running time is

$$O\left(\sum_{i=1}^{k} n \log \mathcal{H}_i\right) = O\left(\sum_{i=1}^{\lceil \log \log \mathcal{H} \rceil} n 2^i\right) = O\left(n \log \mathcal{H}\right)$$

Convex Hulls in k-Dimensions, $k > 3$

For convex hulls of higher dimensions, a recent result by Chazelle [13] showed that the convex hull can be computed in time $O(n \log n + n^{\lfloor k/2 \rfloor})$, which is optimal in all dimensions $k \geq 2$ in the worst case. But this result is insensitive to the output size. The gift-wrapping approach generalizes to higher dimensions and yields an output-sensitive solution with running time $O(n\mathcal{H})$, where $\mathcal{H}$ is the total number of i-faces, $i = 0, 1, \ldots, k-1$ and $\mathcal{H} = O(n^{\lfloor k/2 \rfloor})$ [22]. One can also use *beneath-beyond* method [42] of adding points one at a time in ascending order along one of the coordinate axis.[1] We compute the convex hull $CH(S_{i-1})$ for points $S_{i-1} = \{p_1, p_2, \ldots, p_{i-1}\}$. For each added point p_i we update $CH(S_{i-1})$ to get $CH(S_i)$ for $i = 2, 3, \ldots, n$ by deleting those t-faces, $t = 0, 1, \ldots, k-1$, that are internal to $CH(S_{i-1} \cup \{p_i\})$. It is shown by Seidel [22] that $O(n^2 + \mathcal{H} \log h)$ time is sufficient, where h is the number of extreme points. Most recently Chan [10] obtained an algorithm based on gift-wrapping method that runs in $O(n \log \mathcal{H} + (n\mathcal{H})^{1-1/(\lfloor k/2 \rfloor + 1)} \log^{O(1)} n)$ time. Note that the algorithm is optimal when $k = 2, 3$. In particular, it is optimal when $\mathcal{H} = o(n^{1-\epsilon})$ for some $0 < \epsilon < 1$.

We conclude this section with the following theorem [10].

THEOREM 19.1 *The convex hull of a set S of n points in $\Re^k$ can be computed in $O(n \log \mathcal{H})$ time for $k = 2$ or $k = 3$, and in $O(n \log \mathcal{H} + (n\mathcal{H})^{1-1/(\lfloor k/2 \rfloor + 1)} \log^{O(1)} n)$ time for $k > 3$, where $\mathcal{H}$ is the number of i-faces, $i = 0, 1, \ldots, k-1$.*

[1] If the points of S are not given *a priori*, the algorithm can be made **on-line** by adding an extra step of checking if the newly added point is internal or external to the current convex hull. If internal, just discard it.

Convex Layers of a Planar Set

The convex layers $\mathcal{C}(S)$ of a set S of n points in the Euclidean plane is obtained by a process, known as *onion peeling,* i.e., compute the convex hull of S and remove its vertices from S, until S becomes empty. Figure 19.4 shows the convex layer of a point set. This onion peeling process of a point set is central in the study of robust estimators in statistics, in which the outliers, points lying on the outermost convex layers, should be removed. In this section we describe an efficient algorithm due to Chazelle [11] that runs in optimal $O(n \log n)$ time.

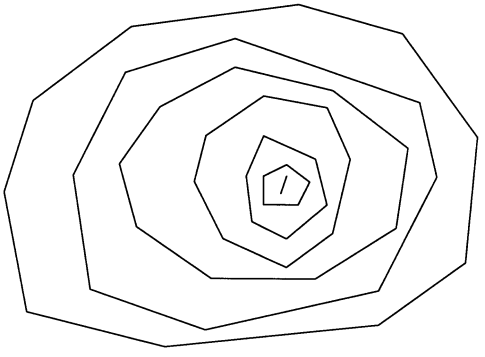


FIGURE 19.4 Convex layers of a point set.

As described in "Convex Hulls in Two and Three Dimensions," each convex layer of $\mathcal{C}(S)$ can be decomposed into two convex polygonal chains, called upper and lower hulls (Fig. 19.5).

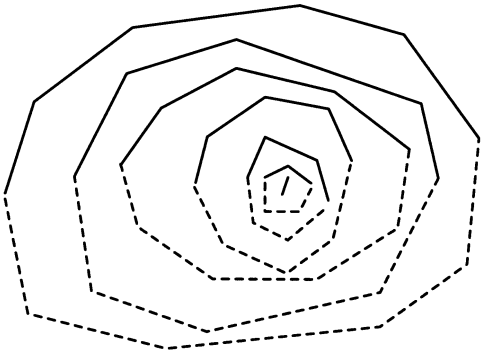


FIGURE 19.5 Decomposition of each convex layer into upper and lower hulls.

Let l and r denote the points with the minimum and maximum x-coordinate, respectively, in a convex layer. The upper (respectively, lower) hull of this layer runs clockwise (respectively, counterclockwise) from l to r. The upper and lower hulls are the same if the convex layer has one or two points. Assume that the set S of points $p_0, p_1, \ldots, p_{n-1}$ are ordered in nondescending order of their x-coordinates. We shall concentrate on the computation of upper hulls of $\mathcal{C}(S)$; the other case is symmetric. Consider the complete binary tree $\mathcal{T}(S)$ with leaves $p_0, p_1, \ldots, p_{n-1}$ from left to right. Let $S(v)$ denote the set of points stored at the leaves of the subtree rooted at node v of $\mathcal{T}$ and let $U(v)$ denote its upper hull of the convex hull of $S(v)$. Thus, $U(\rho)$, where ρ denotes the root of $\mathcal{T}$ is the upper hull of the convex hull of S– in the outermost layer. The union of all the upper hulls $U(v)$ for all nodes v is a tree, called *hull graph* [11].

(A similar graph is also computed for the lower hull of the convex hull.) To minimize the amount of space, at each internal node v we store the bridge (common tangent) connecting a point in $U(v_l)$ and a point in $U(v_r)$, where v_l, v_r are the left and right children of node v. Figures 19.6(a) and (b) illustrate the binary tree $\mathcal{T}$ and the corresponding hull graph, respectively.

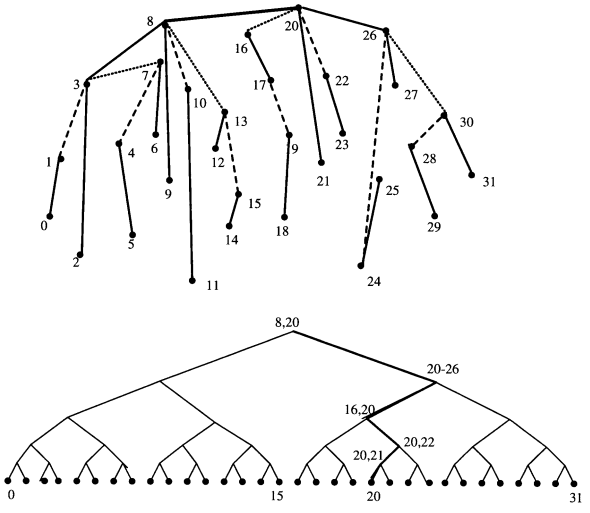


FIGURE 19.6 A complete binary tree and hull graph of upper hulls.

Computation of the hull graph proceeds from bottom up. Computing the bridge at each node takes time linear in the number of vertices on the respective upper hulls in the left and right subtrees. Thus, the total time needed to compute the hull graph is $O(n \log n)$. The bridges computed at each node v which are incident upon a vertex p_k are naturally separated into two subsets divided by the vertical line $\mathcal{L}(p_k)$ passing through p_k. Those on the left are arranged in a list $L(p_k)$ in counterclockwise order from the positive y direction of $\mathcal{L}(p_k)$, and those on the right are arranged in a list $R(p_k)$ in clockwise order. This adjacency list at each vertex in the hull graph can be maintained fairly easily. Suppose the bridge at node v connects vertex p_j in the left subtree and vertex p_k in the right subtree. The edge $\overline{p_j, p_k}$ will be inserted at the *first* position in the current lists $R(p_j)$ and $L(p_k)$. That is, edge $\overline{p_j, p_k}$ is the *top* edge in both lists $R(p_j)$ and $L(p_k)$. It is easy to retrieve the vertices on the upper hull of the outermost layer from the hull graph beginning at the root node of $\mathcal{T}$.

To compute the upper hull of the next convex layer, one needs to remove those vertices on the first layer (including those vertices in the lower hull). Thus, update of the hull graph includes deletion of vertices on both upper hull and lower hull. Deletions of vertices on the upper hull can be performed in an arbitrary order. But if deletions of vertices on the lower hull from the hull-graph are done in say clockwise order, then the update of the adjacency list of each vertex p_k can be made easy, e.g., $R(p_k) = \emptyset$. The deletion of a vertex p_k on the upper hull entails removal of edges incident on p_k in the hull graph. Let $v_1, v_2, \ldots, v_l$ be the list of internal nodes on the leaf-to-root path from p_k. The edges in $L(p_k)$ and $R(p_k)$ are deleted from bottom up in $O(1)$ time each, i.e., the top edge in each list gets deleted last. Figure 19.6(b) shows the leaf-to-root path when vertex p_{20} is deleted. Figures 19.7(a)–(f) show the updates of bridges when p_{20} is deleted and Fig. 19.7(g) is the final upper-hull after the update is finished. It can be shown that the overall time for deletions can be done in $O(n \log n)$ time [11].

THEOREM 19.2 *The convex layers of a set of n points in the plane can be computed in $O(n \log n)$ time.*

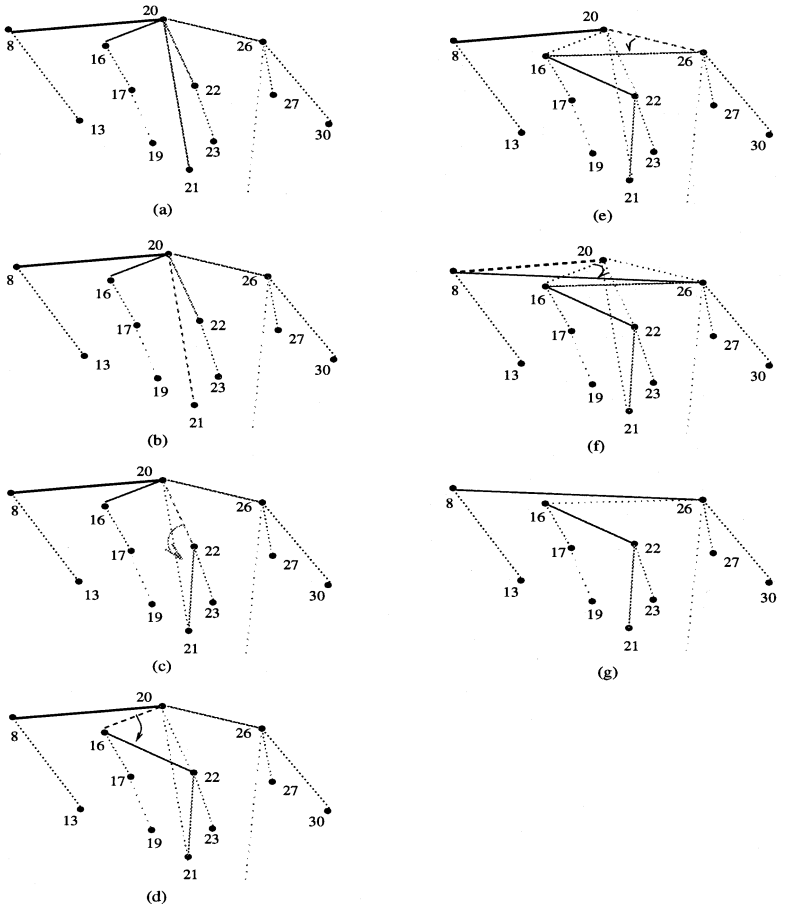


FIGURE 19.7 Update of hull graph.

Applications of Convex Hulls

Convex hulls have applications in clustering, linear regression, and Voronoi diagrams (see Chapter 20.) The following problems have solutions derived from the convex hull.

Problem C1 (Set Diameter) Given a set S of n points, find the two points that are the farthest apart, i.e., find $p_i, p_j \in S$ such that $d(p_i, p_j) = \max\{d(p_k, p_l)\}\ \forall p_k, p_l \in S$, where $d(p, q)$ denotes the Euclidean distance between p and q.

In two dimensions $O(n \log n)$ time is both sufficient and necessary in the worst case [42]. It is easy to see that the farthest pair must be extreme points of the convex hull of S. Once the convex hull is computed, the pair in two dimensions can be found in linear time by observing that it admits a pair of parallel supporting lines. Various attempts, including geometric sampling and parametric search method, have been made to solve this problem in three dimensions. Most recently, Ramos [43] obtained an $O(n \log^2 n)$ time deterministic algorithm, which is an $O(\log n)$ factor off from the best randomized algorithm due to Clarkson and Shor [17].

Problem C2 (Smallest Enclosing Rectangle) Given a set S of n points, find the smallest rectangle that encloses the set.

Problem C3 (Regression Line) Given a set S of n points, find a line such that the maximum distance from S to the line is minimized.

These two problems can be solved in optimal time $O(n \log n)$ using the convex hull of S [38] in two dimensions. In k-dimensions Houle et al. [31] gave an $O(n^{\lfloor k/2+1 \rfloor})$ time and $O(n^{\lfloor (k+1)/2 \rfloor})$ space algorithm. The time complexity is essentially that of computing the convex hull of the point set.

19.3 Maxima Finding

In this section we discuss a problem concerned with *extremes* of a point set which is somewhat related to that of convex hull problems. Consider a set S of n points in $\Re^k$ in the Cartesian coordinate system. Let $(x_1(p), x_2(p), \ldots, x_k(p))$ denote the coordinates of point $p \in \Re^k$. Point p is said to *dominate* point q, denoted $p \succeq q$, (or q is dominated by p, denoted $q \preceq p$) if $x_i(p) \geq x_i(q)$ for all $1 \leq i \leq k$. A point p is said to be *maximal* (or a *maximum*) in S if no point in S dominates p. The maxima-finding problem is that of finding the set $\mathcal{M}(S)$ of maximal elements for a set S of points in $\Re^k$.

Maxima in Two and Three Dimensions

In two dimensions the problem can be done fairly easily by a plane-sweep technique. (For a more detailed description of plane-sweep technique, see, e.g., [36] or "Trapezoidalization".) Assume that the set S of points $p_1, p_2, \ldots, p_n$ are ordered in nondescending order of their x-coordinates, i.e., $x(p_1) \leq x(p_2) \leq \cdots \leq x(p_n)$.

We shall scan the points from right to left. The point p_n is necessarily a maximal element. As we scan the points, we maintain the maximum y-coordinate among those that have been scanned so far. Initially, $\max_y = y(p_n)$. The next point p_i is a maximal element if and only if $y(p_i) > \max_y$. If $y(p_i) > \max_y$, then $p_i \in \mathcal{M}(S)$, and $\max_y$ is set to $y(p_i)$, and we continue. Otherwise $p_i \preceq p_j$ for some $j > i$. Thus, after the initial sorting, the set of maxima can be computed in linear time. Note that the set of maximal elements satisfies the property that their x- and y-coordinates are totally ordered: if they are ordered in strictly ascending x-coordinate, their y-coordinates are ordered in strictly descending order.

In three dimensions we can use the same strategy. We will scan the set in descending order of the x-coordinate by a plane $\mathcal{P}$ orthogonal to the x-axis. Point p_n as before is a maximal element. Suppose we have computed $\mathcal{M}(S_{i+1})$, where $S_{i+1} = \{p_{i+1}, \ldots, p_n\}$, and we are scanning point p_i. Consider the orthogonal projection S^x_{i+1} of the points in S_{i+1} to $\mathcal{P}$ with $x = x(p_i)$. We now have an instance of an *on-line* two dimensional maximal problem, i.e., for point p_i, if $p^x_i \preceq p^x_j$ for some $p^x_j \in S^x_{i+1}$, then it is not a maximal element, otherwise it is (p^x_i denotes the projection of p_i onto $\mathcal{P}$). If we maintain the points in $\mathcal{M}(S^x_{i+1})$ as a **height-balanced binary search tree** in either y- or z-coordinate, then testing whether p_i is maximal or not can be done in logarithmic time. If it is dominated by some point in $\mathcal{M}(S^x_{i+1})$, then it is ignored. Otherwise, it is in $\mathcal{M}(S^x_{i+1})$ (and also in $\mathcal{M}(S_{i+1})$); $\mathcal{M}(S^x_{i+1})$ will then be updated to be $\mathcal{M}(S^x_i)$ accordingly. The update may involve deleting points in $\mathcal{M}(S^x_{i+1})$ that are no longer maximal because they are dominated by p^x_i. Figure 19.8 shows the effect of adding a maximal element p^x_i to the set $\mathcal{M}(S^x_{i+1})$ of maximal elements. Points in the shaded area will be deleted. Thus, after the initial sorting, the set of maxima in three dimensions can be computed in $O(n \log \mathcal{H})$ time, as the *on-line* two dimensional maximal problem takes $O(\log \mathcal{H})$ time to maintain $\mathcal{M}(S^x_i)$ for each point p_i, where $\mathcal{H}$ denotes the size of $\mathcal{M}(S)$.

Since the total number of points deleted is at most n, we conclude the following:

LEMMA 19.1 Given a set of n points in two and three dimensions, the set of maxima can be computed in $O(n \log n)$ time.

For two and three dimensions one can solve the problem in optimal time $O(n \log \mathcal{H})$, where $\mathcal{H}$ denotes the size of $\mathcal{M}(S)$. The key observation is that we need not *sort* S in its entirety. For instance, in two dimensions one can solve the problem by divide-and-marriage-before-conquest paradigm. We first use a

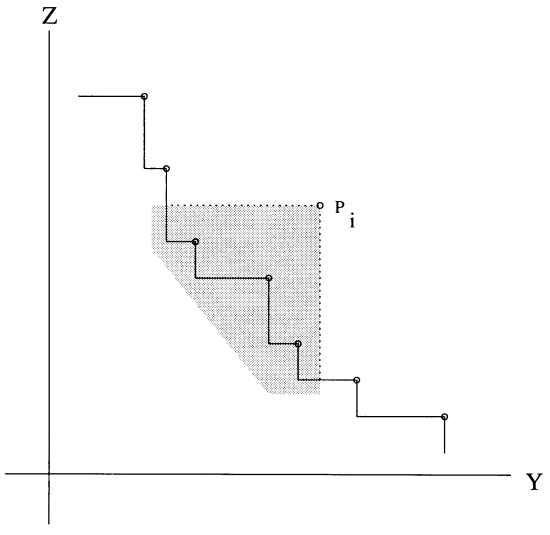


FIGURE 19.8 Update of maximal elements.

linear time median finding algorithm to divide the set into two halves L and R with points in R having larger x-coordinate values than those of points in L. We then recursively compute $\mathcal{M}(R)$. Before we recursively compute $\mathcal{M}(L)$ we note that points in L that are dominated by points in $\mathcal{M}(R)$ can be eliminated from consideration. We trim L before we invoke the algorithm recursively. That is, we compute $\mathcal{M}(L')$ recursively, where $L' \subseteq L$ consists of points $q \not\preceq p$ for all $p \in \mathcal{M}(R)$. A careful analysis of the running time shows that the complexity of this algorithm is $O(n \log \mathcal{H})$. For three dimensions we note that other than the initial sorting step, the subsequent plane-sweep step takes $O(n \log \mathcal{H})$ time. It turns out that one can replace the full-fledged $O(n \log n)$ sorting step with a so-called *lazy sorting* of S using a technique similar to those described in "Convex Hulls in Two and Three Dimensions" to derive an output-sensitive algorithm.

THEOREM 19.3 *Given a set S of n points in two and three dimensions, the set $\mathcal{M}(S)$ of maxima can be computed in $O(n \log \mathcal{H})$ time, where $\mathcal{H}$ is the size of $\mathcal{M}(S)$.*

Maxima in Higher Dimensions

The set of maximal elements in $\Re^k$, $k \geq 4$, can be solved by a generalization of plane-sweep method to higher dimensions. We just need to maintain a data structure for $\mathcal{M}(S_{i+1})$, where $S_{i+1} = \{p_{i+1}, \ldots, p_n\}$, and test for each point p_i if it is a maximal element in S_{i+1}, reducing the problem to one dimension lower, assuming that the points in S are sorted and scanned in descending lexicographical order. Thus, in a straightforward manner we can compute $\mathcal{M}(S)$ in $O(n^{k-2} \log n)$ time. However, we shall show that one can compute the set of maxima in $O(n \log^{k-2} n)$ time, for $k > 3$ by divide-and-conquer[2].

Let us first consider a *bichromatic maxima-finding* problem. Consider a set of n *red* and a set of m *blue* points, denoted R and B, respectively. The bichromatic maxima-finding problem is to find a subset of points in R that are not dominated by any points in B and vice versa. That is, find $\mathcal{M}(R, B) = \{r | r \not\preceq b, b \in B\}$ and $\mathcal{M}(B, R) = \{b | b \not\preceq r, r \in R\}$.

[2]This was improved to $O(n \log^{k-3} n \log \log n)$ time by Gabow et al. [27].

In three dimensions, this problem can be solved by plane-sweep in a manner similar to the maxima-finding problem as follows. As before, the sets R and B are sorted in nondescending order of x-coordinates and we maintain two subsets of points $\mathcal{M}(R^x_{i+1})$ and $\mathcal{M}(B^x_{j+1})$, which are the maxima of the projections of R_{i+1} and B_{j+1} onto the yz-plane for $R_{i+1} = \{r_{i+1}, \ldots, r_n\} \subseteq R$ and $B_{j+1} = \{b_{j+1}, \ldots, b_m\} \subseteq B$, respectively. When the next point $r_i \in R$ is scanned, we test if r^x_i is dominated by any points in $\mathcal{M}(B^x_{j+1})$. The point $r_i \in \mathcal{M}(R, B)$, if r^x_i is not dominated by any points in $\mathcal{M}(B^x_{j+1})$. We then update the set of maxima for $R^x_i = R^x_{i+1} \cup \{r^x_i\}$. That is, if $r^x_i \preceq q$ for $q \in \mathcal{M}(R^x_{i+1})$, then $\mathcal{M}(R^x_i) = \mathcal{M}(R^x_{i+1})$. Otherwise, the subset of $\mathcal{M}(R^x_{i+1})$ dominated by r^x_i is removed, and r^x_i is included in $\mathcal{M}(R^x_i)$. If the next point scanned is $b_j \in B$, we perform similar operations. Thus, for each point scanned we spend $O(\log n + \log m)$ time.

LEMMA 19.2 The bichromatic maxima-finding problem for a set of n red and m blue points in three dimensions can be solved in $O(N \log N)$ time, where $N = m + n$.

Using Lemma 19.2 as basis, one can solve the bichromatic maxima-finding problem in $\Re^k$ in $O(N \log^{k-2} N)$ time for $k \geq 3$ using multidimensional divide-and-conquer.

LEMMA 19.3 The bichromatic maxima-finding problem for a set of n red and m blue points in $\Re^k$ can be solved in $O(N \log^{k-2} N)$ time, where $N = m + n$, and $k \geq 3$.

Let us now turn to the maxima-finding problem in $\Re^k$. We shall use an ordinary divide-and-conquer method to solve the maxima-finding problem. Assume that the points in $S \subseteq \Re^k$ have been sorted in all dimensions. Let L_x denote the median of all the x-coordinate values. We first divide S into two subsets S_1 and S_2, each of size approximately $|S|/2$ such that the points in S_1 have x-coordinates larger than L_x and those of points in S_2 are less than L_x. We then recursively compute $\mathcal{M}(S_1)$ and $\mathcal{M}(S_2)$. It is clear that $\mathcal{M}(S_1) \subseteq \mathcal{M}(S)$. However, some points in $\mathcal{M}(S_2)$ may be dominated by points in $\mathcal{M}(S_1)$, and hence, are not in $\mathcal{M}(S)$. We then project points in S onto the hyperplane $\mathcal{P}$: $x = L_x$. The problem now reduces to the bichromatic maxima-finding problem in $\Re^{k-1}$, i.e., finding among $\mathcal{M}(S_2)$ those that are maxima with respect to $\mathcal{M}(S_1)$. By Lemma 19.3 we know that this bichromatic maxima-finding problem can be solved in $O(n \log^{k-3} n)$ time. Since the merge step takes $O(n \log^{k-3} n)$ time, we conclude the following:

THEOREM 19.4 *The maxima-finding problem for a set of n points in $\Re^k$ can be solved in $O(n \log^{k-2} n)$ time, for $k \geq 3$.*

We note here also that if we apply the *trimming* operation of S_2 with $\mathcal{M}(S_1)$, i.e., removing points in S_2 that are dominated by points in $\mathcal{M}(S_1)$, before recursion, one can compute $\mathcal{M}(S)$ more efficiently as stated in the following theorem.

THEOREM 19.5 *The maxima-finding problem for a set S of n points in $\Re^k$, $k \geq 4$, can be solved in $O(n \log^{k-2} \mathcal{H})$ time, where $\mathcal{H}$ is the number of maxima in S.*

Maximal Layers of a Planar Set

The maximal layers of a set of points in the plane can be obtained by a process similar to that of convex layers discussed in "Convex Layers of a Planar Set." A brute-force method would yield an $O(\delta \cdot n \log \mathcal{H})$ time, where δ is the number of layers and $\mathcal{H}$ is the maximum number of maximal elements in any layer. In this section we shall present an algorithm due to Atallah and Kosaraju [6] for computing not only the maximal layers, but also some other functions associated with dominance relation.

Consider a set S of n points. As in the previous section, let $\mathcal{D}_S(p)$ denote the set of points in S dominated by p, i.e., $\mathcal{D}_S(p) = \{q \in S | q \preceq p\}$. Since p is always dominated by itself, we shall assume $\mathcal{D}_S(p)$ does not include p, when $p \in S$. The first subproblem we consider is the *maxdominance problem,* which is defined as follows: for each $p \in S$, find $\mathcal{M}(\mathcal{D}_S(p))$. That is, for each $p \in S$ we are interested in computing the set of maximal elements among those points that are dominated by p. Another related problem is to compute the labels of each point p from the labels of those points in $\mathcal{M}(\mathcal{D}_S(p))$. More specifically, suppose each point is associated with a weight $w(p)$. The label $l_S(p)$ is defined to be $w(p)$ if $\mathcal{D}_S(p) = \emptyset$ and is $w(p) + \max\{l_S(q), q \in \mathcal{M}(\mathcal{D}_S(p))\}$. The max function can be replaced with min or any other associative functional operation. In other words, $l_S(p)$ is equal to the maximum among the labels of all the points dominated by p. Suppose we let $w(p) = 1$ for all $p \in S$. Then those points with labels equal to 1 are points that do not dominate any points. These points can be thought of as *minimal* points in S. That a point p_i has label λ implies there exists a sequence of λ points $p_{j_1}, p_{j_2}, \ldots, p_{j_\lambda} = p_i$, such that $p_{j_1} \preceq p_{j_2} \preceq \cdots \preceq p_{j_\lambda} = p_i$. In general, points with label λ are on the λ^{th} minimal layer and the maximum label gives the number of minimal layers. If we modify the definition of domination to be p dominates q if and only if $x(p) \leq x(q)$ and $y(p) \leq y(q)$, then the minimal layers obtained using the method to be described below correspond to the maximal layers.

Let us now discuss the labeling problem defined earlier. We recall a few terms as used in [6].[3]

Let L and R denote two subsets of points of S separated by a vertical line, such that $x(l) \leq x(r)$ for all $l \in L$ and $r \in R$. $leader_R(p)$, $p \in R$ is the point $\mathcal{H}_p$ in $\mathcal{D}_R(p)$ with the largest y-coordinate. $Strip_L(p, R)$, $p \in R$ is the subset of points of $\mathcal{D}_L(p)$ dominated by p but with y-coordinates greater than $leader_R(p)$, i.e., $Strip_L(p, R) = \{q \in \mathcal{D}_L(p) | y(q) > y(\mathcal{H}_p)\}$ for $p \in R$. $Left_L(p, R)$, $p \in R$, is defined to be the largest $l_S(q)$ over all $q \in Strip_L(p, R)$ if $Strip_L(p, R)$ is nonempty, and $-\infty$ otherwise.

Observe that for each $p \in R$ $\mathcal{M}(\mathcal{D}_S(p))$ is the concatenation of $\mathcal{M}(\mathcal{D}_R(p))$ and $Strip_L(p, R)$. Assume that the points in $S = \{p_1, p_2, \ldots, p_n\}$ have been sorted as $x(p_1) < x(p_2) < \cdots < x(p_n)$. We shall present a divide-and-conquer algorithm that can be called with $R = S$ and $Left_\emptyset(p, S) = -\infty$ for all $p \in S$ to compute $l_S(p)$ for all $p \in S$. The correctness of the algorithm hinges on the following lemma.

LEMMA 19.4 For any point $p \in R$, if $\mathcal{D}_S(p) \neq \emptyset$, then $l_S(p) = w(p) + \max\{Left_L(p, R), \max\{l_S(q), q \in \mathcal{M}(\mathcal{D}_R(p))\}\}$.

ALGORITHM MAXDOM_LABEL(R)

Input: A consecutive sequence of m points of S, i.e., $R = \{p_r, p_{r+1}, \ldots, p_{r+m-1}\}$ and for each $p \in R$, $Left_L(p, R)$, where $L = \{p_1, p_2, \ldots, p_{r-1}\}$. Assume a list Q_R of points of R sorted by increasing y-coordinate.

Output: The labels $l_S(q)$, $q \in R$.

1. If $m = 1$ then we set $l_S(p_r)$ to $w(p_r) + Left_L(p_r, R)$, if $Left_L(p_r, R) \neq -\infty$ and to $w(p_r)$ if $Left_L(p_r, R) = -\infty$, and **return.**
2. Partition R by a vertical line $\mathcal{V}$ into subsets R_1 and R_2 such that $|R_1| = |R_2| = m/2$ and R_1 is to the left of R_2. Extract from Q_R the lists Q_{R_1} and Q_{R_2}.
3. Call MAXDOM_LABEL(R_1). Since $Left_L(p, R_1)$ equals $Left_L(p, R)$, this call will return the labels for all $q \in R_1$ which are the final labels for $q \in R$.
4. Compute $Left_{R_1}(p, R_2)$.

[3] Some of the notation is slightly modified. In [6] min is used in the *label* function, instead of max. See [6] for details.

5. Compute $Left_{L\cup R_1}(p, R_2)$, given $Left_{R_1}(p, R_2)$ and $Left_L(p, R)$. That is, for each $p \in R_2$, set $Left_{L\cup R_1}(p, R_2)$ to be $\max\{Left_{R_1}(p, R_2), Left_L(p, R)\}$.
6. Call MAXDOM_LABEL(R_2). This will return the labels for all $q \in R_2$ which are the final labels for $q \in R$.

All steps other than Step 4 are self-explanatory. Steps 4 and 5 are needed in order to set up the correct invariant condition for the second recursive call. The computation of $Left_{R_1}(p, R_2)$ and its complexity is the key to the correctness and time complexity of the algorithm MAXDOM_LABEL(R). We briefly discuss this problem and show that this step can be done in $O(m)$ time. Since all other steps take linear time, the overall time complexity is $O(m \log m)$.

Consider in general two subsets L and R of points separated by a vertical line $\mathcal{V}$, with L lying to the left of R and points in $L \cup R$ are sorted in ascending y-coordinate (Fig. 19.9). Suppose we have computed the labels $l_L(p)$, $p \in L$. We compute $Left_L(p, R)$ by using a plane sweep technique scanning points in $L \cup R$ is ascending y-coordinate. We will maintain for each point $r \in R$ $Strip_L(r, R)$ along with the highest and rightmost points in the subset, denoted $1st_L(r, R)$ and $last_L(r, R)$, respectively, and $leader_R(r)$. For each point $p \in L \cap Strip_L(r, R)$ for some $r \in R$ we maintain a label $max\_l(p)$, which is equal to $\max\{l_L(q) | q \in Strip_L(r, R)$ and $y(q) < y(p)\}$.

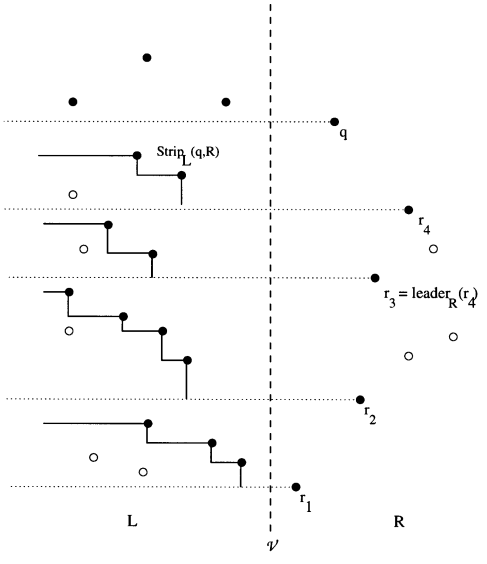


FIGURE 19.9 Computation of $Left_L(p, R)$.

A stack ST_R will be used to store $leader_R(r_i)$ of $r_i \in R$ such that any element r_t in ST_R is $leader_R(r_{t+1})$ for point r_{t+1} above r_t, and the top element r_i of ST_R is the last scanned point in R. For instance in Fig. 19.9 ST_R contains r_4, r_3, r_2, and r_1 when r_4 is scanned. Another stack ST_L is used to store $Strip_L(r, R)$ for a yet-to-be-scanned point $r \in R$. (The staircase above r_4 in Fig. 19.9 is stored in ST_L. The solid staircases indicate $Strip_L(r_i, R)$ for $r_i, i = 2, 3, 4$.)

Let the next point scanned be denoted q. If $q \in L$, we pop off the stack ST_L all points that are dominated by q until q'. And we compute $max\_l(q)$ to be the larger of $l_L(q)$ and $max\_l(q')$. We then push q onto ST_L. That is, we update ST_L to make sure that all the points in ST_L are maximal.

Suppose $q \in R$. Then $Strip_L(q, R)$ is initialized to be the entire contents of ST_L and let $1st_L(q, R)$ be the top element of ST_L and $last_L(q, R)$ be the bottom element of ST_L.

If the top element of ST_R is equal to $leader_R(q)$, we set $Left_L(q, R)$ to $max\_l(q')$, where q' is $1st_L(q, R)$,

initialize ST_L to be empty and continue to scan the next point. Otherwise we need to pop off the stack ST_R all points that are *not* dominated by q, until q', which is $leader_R(q)$. As shown in Fig. 19.9, $r_i, i = 4, 3, 2$ will be popped off ST_R when q is scanned. As point r_i is popped off ST_R, $Strip_L(r_i, R)$ is *concatenated* with $Strip_L(q, R)$ to maintain its maximality. That is, the points in $Strip_L(r_i, R)$ are scanned from $1st_L(r_i, R)$ to $last_L(r_i, R)$ until a point, if any, α_i is encountered such that $x(\alpha_i) > x(last_L(q, R))$. $max\_1(q')$, $q' = 1st_L(q, R)$, is set to be the larger of $max\_1(q')$ and $max\_1(\alpha)$ and $last_L(q, R)$ is temporarily set to be $last_L(r_i, R)$. If no such α_i exists, then the entire $Strip_L(r_i, R)$ is ignored. This process repeats until $leader_R(q)$ of q is on top of ST_R. At that point, we would have computed $Strip_L(q, R)$ and $Left_L(q, R)$ is $max\_1(q')$, where $q' = 1st_L(q, R)$. We initialize ST_L to be empty and continue to scan the next point.

It has been shown in [6] that this scanning operation takes linear time (with path compression), so the overall algorithm takes $O(m \log m)$ time.

THEOREM 19.6 *Given a set S of n points with weights $w(p_i)$, $p_i \in S, i = 1, 2, \ldots, n$,* ALGORITHM MAXDOM_LABEL(S) *returns $l_S(p)$ for each point $p \in S$ in $O(n \log n)$ time.*

Now let us briefly describe the algorithm for the *maxdominance problem.* That is to find for each $p \in S$, $\mathcal{M}(\mathcal{D}_S(p))$.

ALGORITHM MAXDOM_LIST(S)

Input: A sorted sequence of n points of S, i.e., $S = \{p_1, p_2, \ldots, p_n\}$, where $x(p_1) < x(p_2) < \cdots < x(p_n)$.

Output: $\mathcal{M}(\mathcal{D}_S(p))$ for each $p \in S$ and the list Q_S containing the points of S in ascending y-coordinates.

1. If $n = 1$ then we set $\mathcal{M}(\mathcal{D}_S(p_1)) = \emptyset$ and **return.**
2. Call ALGORITHM MAXDOM_LIST(L), where $L = \{p_1, p_2, \ldots, p_{n/2}\}$. This call returns $\mathcal{M}(\mathcal{D}_S(p))$ for each $p \in L$ and the list Q_L.
3. Call ALGORITHM MAXDOM_LIST(R), where $R = \{p_{n/2+1}, \ldots, p_n\}$. This call returns $\mathcal{M}(\mathcal{D}_R(p))$ for each $p \in R$ and the list Q_R.
4. Compute for each $r \in R$ $Strip_L(r, R)$ using the algorithm described in Step 4 of ALGORITHM MAXDOM_LABEL(R).
5. For every $r \in R$ compute $\mathcal{M}(\mathcal{D}_S(p))$ by concatenating $Strip_L(r, R)$ and $\mathcal{M}(\mathcal{D}_R(p))$.
6. Merge Q_L and Q_R into Q_S and return.

Since Steps 4, 5, and 6, excluding the output time, can be done in linear time, we have the following.

THEOREM 19.7 *The maxdominance problem of a set S of n points can be solved in $O(n \log n + \mathcal{F})$ time, where $\mathcal{F} = \Sigma_{p \in S} |\mathcal{M}(\mathcal{D}_S(p))|$.*

19.4 Row Maxima Searching in Monotone Matrices

The *row maxima-searching problem* in a matrix is that given an $n \times m$ matrix M of real entries, find the leftmost maximum entry in each row.

A matrix is said to be *monotone,* if $i_1 > i_2$ implies that $j(i_1) \geq j(i_2)$, where $j(i)$ is the index of the leftmost column containing the maximum in row i. It is *totally monotone* if all of its submatrices are monotone.

In fact if every 2×2 submatrix $M[i, j; k, l]$ with $i < j$ and $k < l$ is monotone, then the matrix is totally monotone. Or equivalently if $M(i, k) < M(i, l)$ implies $M(j, k) < M(j, l)$ for any $i < j$ and $k < l$, then M is totally monotone.

The algorithm for solving the row maxima-searching problem is due to Aggarwal et al. [1], and is commonly referred to as the SMAWK algorithm. Specifically the following results were obtained: $O(m \log n)$ time for an $n \times m$ monotone matrix, and $\theta(m)$ time, $m \geq n$, and $\theta(m(1 + \log(n/m)))$ time, $m < n$, if the matrix is totally monotone.

We use as an example the distance matrix between pairs of vertices of a convex n-gon P, represented as a sequence of vertices $p_1, p_2, \ldots, p_n$ in counterclockwise order. For an integer j, let $*j$ denote $((j-1) \bmod n) + 1$. Let M be an $n \times (2n-1)$ matrix defined as follows. If $i < j \leq i + n - 1$ then $M[i, j] = d(p_i, p_{*j})$, where $d(p_i, p_j)$ denotes the Euclidean distance between two vertices p_i and p_j. If $j \leq i$ then $M[i, j] = j - i$, and if $j \geq i + n$ then $M[i, j] = -1$. The problem of computing for each vertex its farthest neighbor is now the same as the row maxima-searching problem.

Consider submatrix $M[i, j; k, l]$, with $i < j$ and $k < l$, that has only positive entries, i.e., $i < j < k < l < i + n$. In this case vertices p_i, p_j, p_{*k}, and p_{*l} are in counterclockwise order around the polygon. From the triangle inequality we have $d(p_i, p_{*k}) + d(p_j, p_{*l}) \geq d(p_i, p_{*l}) + d(p_j, p_{*k})$. Thus, $M[i, j; k, l]$ is monotone. The nonpositive entries ensure that all other 2×2 submatrices are monotone. We'll show below that the all farthest neighbor problem for each vertex of a convex n-gon can be solved in $O(n)$ time.

A straightforward divide-and-conquer algorithm for the row maxima-searching problem in monotone matrices is as follows.

ALGORITHM MAXIMUM_D&C

1. Find the maximum entry $j = j(i)$, in the ith row, where $i = \lceil \frac{n}{2} \rceil$.
2. Recursively solve the row maxima-searching problem for the submatrices $M[1, \ldots, i-1; 1, \ldots, j]$ when $i, j > 1$ and $M[i+1, \ldots, n; j, \ldots, m]$ when $i < n$ and $j < m$.

The time complexity required by the algorithm is given by the recurrence

$$f(n, m) \leq m + \max_{1 \leq j \leq m} \left(f(\lceil n/2 \rceil - 1, j) + f(\lfloor n/2 \rfloor, m - j + 1) \right).$$

with $f(0, m) = f(n, 1) =$ constant. We have $f(n, m) = O(m \log n)$.

Now let us consider the case when the matrix is totally monotone. We distinguish two cases (a) $m \geq n$ and (b) $m < n$.

Case (a): Wide matrix $m \geq n$.

An entry $M[i, j]$ is *bad* if $j \neq j(i)$, i.e., column j is not a solution to row i. Column j, $M[*, j]$ is *bad* if all $M[i, j]$, $1 \leq i \leq n$ are bad.

LEMMA 19.5 For $j_1 < j_2$ if $M[r, j_1] \geq M[r, j_2]$, then $M[i, j_2]$, $1 \leq i \leq r$, are bad; otherwise $M[i, j_1]$, $r \leq i \leq n$, are bad.

Consider an $n \times n$ matrix C, the *index* of C is defined to be the largest k such that $C[i, j]$, $1 \leq i < j$, $1 \leq j \leq k$ are bad.

The following algorithm REDUCE reduces in $O(m)$ time a totally monotone $m \times n$ matrix M to an $n \times n$ matrix C, a submatrix of M, such that for $1 \leq i \leq n$ it contains column $M^{j(i)}$. That is, bad columns of M (which are known not to be contain solutions) are eliminated.

ALGORITHM REDUCE(M)

1. $C \leftarrow M$; $k \leftarrow 1$;

2. **while** C has more than n columns **do**

   case $C(k,k) \geq C(k,k+1)$ and $k < n$: $k \leftarrow k+1$;
   $C(k,k) \geq C(k,k+1)$ and $k = n$: Delete column C^{k+1};
   $C(k,k) < C(k,k+1)$: Delete column C^k; **if** $k > 1$ **then** $k \leftarrow k-1$
   end case

3. **return**(C)

The following algorithm solves the maxima-searching problem in an $n \times m$ totally monotone matrix, where $m \geq n$.

ALGORITHM MAX_COMPUTE(M)

1. $B \leftarrow$ REDUCE(M);
2. **if** $n = 1$ **then** output the maximum and **return**;
3. $C \leftarrow B[2, 4, \ldots, 2\lfloor n/2 \rfloor; 1, 2, \ldots, n]$;
4. Call MAX_COMPUTE(C);
5. From the known positions of the maxima in the even rows of B, find the maxima in the odd rows.

The time complexity of this algorithm is determined by the following recurrence:

$$f(n,m) \leq c_1 n + c_2 m + f\,(n/2, n)$$

with $f(0,m) = f(n,1) =$ constant. We therefore have $f(n,m) = O(m)$.

Case (b): Narrow matrix $m < n$.

In this case we decompose the problem into m subproblems each of size $\lfloor n/m \rfloor \times m$ as follows. Let $r_i = \lfloor in/m \rfloor$, for $0 \leq i \leq m$. Apply MAX_COMPUTE to the $m \times m$ submatrix $M[r_1, r_2, \ldots, r_m; 1, 2, \ldots, m]$ to get $c_1, c_2, \ldots, c_m$, where $c_i = j(r_i)$. This takes $O(m)$ time. Let $c_0 = 1$. Consider submatrices $B_i = M[r_{i-1}+1, r_{i-1}+2, \ldots, r_i - 1; c_{i-1}, c_{i-1}+1, \ldots, c_i]$ for $1 \leq i \leq m$ and $r_{i-1} \leq r_i - 2$. Applying the straightforward divide-and-conquer algorithm to the submatrices, B_i, we obtain the column positions of the maxima for all remaining rows. Since each submatrix has at most $\lfloor n/m \rfloor$ rows, the time for finding the maxima is at most $c(p_i - p_{i-1} + 1)\log(n/m)$ for some constant c. Summing over all $1 \leq i \leq m$ we get the total time, which is $O(m(1 + \log(n/m)))$. The bound can be shown to be tight [1].

The applications of the matrix searching algorithm include the problems of finding all farthest neighbors for all vertices of a convex n-gon ($O(n)$ time), and finding the extremal (maximum perimeter or area) polygons (inscribed k-gons) of a convex n-gon ($O(kn + n \log n)$). If one adopts a more recent algorithm, the above problems can be solved in $O(n)$ time [30]. It is also used in solving the largest empty rectangle problem discussed in the section on geometric optimization in the next chapter.

19.5 Decomposition

Polygon decomposition arises in pattern recognition [48] in which recognition of a shape is facilitated by first decomposing it into simpler components, called *primitives,* and comparing them to templates previously stored in a library via some similarity measure. This class of decomposition is called *component-directed* decomposition. The primitives are often convex.

Trapezoidalization

We'll consider first *trapezoidalization* of a polygon P with n vertices, i.e., decomposition of the interior of a polygon into a collection of *trapezoids* with two horizontal sides, one of which may degenerate into a point, reducing a trapezoid to a triangle. Without loss of generality let us assume that no edge of P is horizontal. For each vertex v let us consider the horizontal line passing through v, denoted $\mathcal{H}_v$. The vertices of P are classified into three types. A vertex v is *regular* if the other two vertices adjacent to v lie on different sides of $\mathcal{H}_v$. A vertex v is a V-*cusp* if the two vertices adjacent to v are above $\mathcal{H}_v$, and is a Λ-*cusp* if the two vertices adjacent to v are below $\mathcal{H}_v$. In general the intersection of $\mathcal{H}_v$ and the interior of P consists of a number of horizontal segments, one of which contains v. Let this segment be denoted $\overline{v_l, v_r}$, where v_l and v_r are called the *left* and *right* projections of v on the boundary of P, denoted ∂P, respectively. If v is regular, either $\overline{v, v_l}$ or $\overline{v, v_r}$ lies totally in the interior of P. If v is a V-cusp or Λ-cusp, then $\overline{v_l, v_r}$ either lies totally in the interior of P or degenerates to v itself.

Consider only the segments $\overline{v_l, v_r}$ that are *nondegenerate.* These segments collectively partition the interior of P into a collection of trapezoids, each of which contains no vertex of P in its interior [Fig. 19.10(a)].

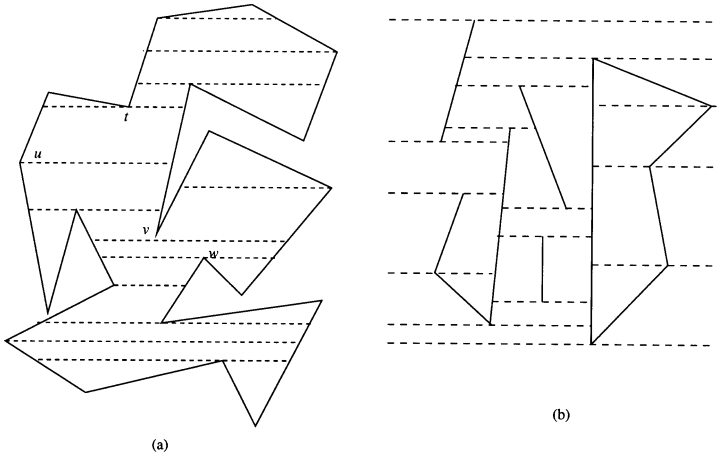


FIGURE 19.10 Trapezoidalization of a polygon.

The trapezoidalization can be generalized to a **planar straight-line graph** $G(V, E)$, where the entire plane is decomposed into trapezoids, some of which are unbounded. This trapezoidalization is sometimes referred to as horizontal **visibility map** of the edges, as the horizontal segments connect two edges of G that are *visible* (horizontally) [Fig. 19.10(b)]. The trapezoidalization of a planar straight-line graph $G(V, E)$ can be computed by plane-sweep technique in $O(n \log n)$ time, where $n = |V|$ [42], while the trapezoidalization of a simple polygon can be found in linear time [12].

The plane-sweep algorithm works as follows. The vertices of the graph $G(V, E)$ are sorted in descending y-coordinates. We'll sweep the plane by a horizontal sweep-line from top down. Associated with this approach there are two basic data structures containing all *relevant* information that should be maintained.

1. *Sweep-line status,* which records the information of the geometric structure that is being swept. In this example the sweep-line status keeps track of the set of edges intersecting the current sweep-line.
2. *Event schedule,* which defines a sequence of *event points* that the sweep-line status will change. In this example, the sweep-line status will change only at the vertices.

The event schedule is normally represented by a data structure, called *priority queue.* The content of the queue may not be available entirely at the start of the plane-sweep process. Instead, the list of events may change dynamically. In this case, the events are static; they are the y-coordinates of the vertices. The sweep-line status is represented by a suitable data structure that supports insertions, deletions, and computation of the left and right projections, v_l and v_r, of each vertex v. In this example a *red-black tree* or any *height-balanced binary search tree* is sufficient for storing the edges that intersect the sweep-line according to the x-coordinates of the intersections. Suppose at event point v_{i-1} we maintain a list of edges intersecting the sweep-line from left to right. Analogous to the trapezoidalization of a polygon, we say that a vertex v is *regular* if there are edges incident on v that lie on different sides of $\mathcal{H}_v$; a vertex v is a V-*cusp* if the all the vertices adjacent to v are above $\mathcal{H}_v$; v is a Λ-*cusp* if all the vertices adjacent to v are below $\mathcal{H}_v$. For each event point v_i we do the following.

1. v_i is regular. Let the leftmost and rightmost edges that are incident on v_i and above $\mathcal{H}_{v_i}$ are $E_l(v_i)$ and $E_r(v_i)$, respectively. The left projection v_{il} of v_i is the intersection of $\mathcal{H}_{v_i}$ and the edge to the left of $E_l(v_i)$ in the sweep-line status. Similarly the right projection v_{ir} of v_i is the intersection of $\mathcal{H}_{v_i}$ and the edge to the right of $E_r(v_i)$ in the sweep-line status. All the edges between $E_l(v_i)$ and $E_r(v_i)$ in the sweep-line status are replaced in an order-preserving manner by the edges incident on v_i that are below $\mathcal{H}_{v_i}$.
2. v_i is a V-cusp. The left and right projections of v_i are computed in the same manner as in Step 1 above. All the edges incident on v_i are then *deleted* from the sweep-line status.
3. v_i is a Λ-cusp. We use binary search in the sweep-line status to look for the two adjacent edges $E_l(v_i)$ and $E_r(v_i)$ such that v_i lies in between. The left projection v_{il} of v_i is the intersection of $\mathcal{H}_{v_i}$ and $E_l(v_i)$ and the right projection v_{ir} of v_i is the intersection of $\mathcal{H}_{v_i}$ and $E_r(v_i)$. All the edges incident on v_i are then inserted in an order-preserving manner between $E_l(v_i)$ and $E_r(v_i)$ in the sweep-line status.

Figure 19.11 illustrates these three cases. Since the update of the sweep-line status for each event point takes $O(\log n)$ time, the total amount of time needed is $O(n \log n)$.

THEOREM 19.8 *Given a planar straight-line graph $G(V, E)$, the horizontal* **visibility map** *of G can be computed in $O(n \log n)$ time, where $n = |V|$. However, if G is a simple polygon then the horizontal visibility map can be computed in linear time.*

Triangulation

In this section we consider triangulating a planar straight-line graph by introducing noncrossing edges so that each face in the final graph is a triangle and the outermost boundary of the graph forms a convex polygon. Triangulation of a set of (discrete) points in the plane is a special case. This is a fundamental problem that arises in computer graphics, geographical information systems, and finite element methods. Let us start with the simplest case.

Polygon Triangulation

Consider a simple polygon P with n vertices. It is obvious that to triangulate the interior of P (into $n - 2$ triangles) one needs to introduce at most $n - 3$ diagonals. A pioneering work is due to Garey et al. [28] who gave an $O(n \log n)$ algorithm and a linear algorithm if the polygon is *monotone.* A polygon is monotone if there exists a straight line $\mathcal{L}$ such that the intersection of ∂P and any line orthogonal to $\mathcal{L}$ consists of no more than two points. The shaded area in Fig. 19.12(a) is a monotone polygon.

The $O(n \log n)$ time algorithm can be illustrated by the following two-step procedure.

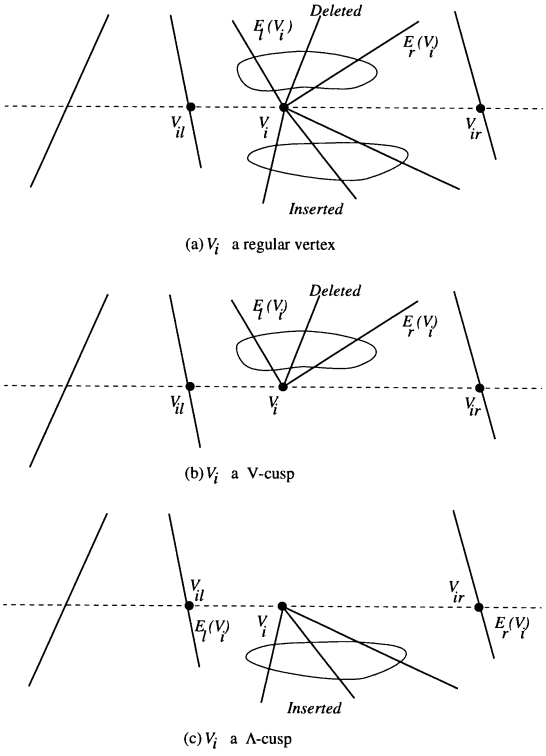


FIGURE 19.11 Updates of sweep-line status.

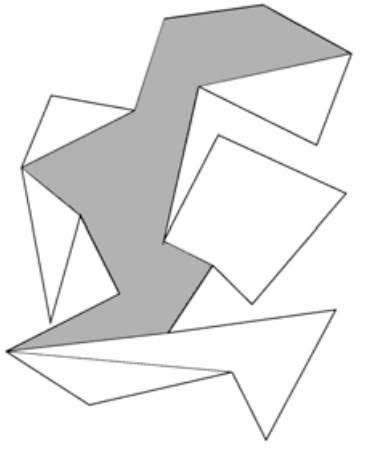


FIGURE 19.12 Decomposition of a simple polygon into monotone subpolygons.

1. Decompose P into a collection of monotone subpolygons with respect to the y-axis in time $O(n \log n)$.
2. Triangulate each monotone subpolygons in linear time.

To find a decomposition of P into a collection of monotone polygons we first obtain the horizontal visibility map described in "Trapezoidalization." In particular we obtain for each cusp v the left and right projections and the associated trapezoid below $\mathcal{H}_v$ if v is a V-cusp, and above $\mathcal{H}_v$ if v is a Λ-cusp. (Recall that $\mathcal{H}_v$ is the horizontal line passing through v.) For each V-cusp v we introduce an edge $\overline{v, w}$ where w

is the vertex through which the other base of the trapezoid below passes. $\overline{t, u}$ and $\overline{v, w}$ in Fig. 19.10(a) illustrate these two possibilities, respectively. For each Λ-cusp we do the same thing. In this manner we convert each vertex into a regular vertex, except the cusps v for which $\overline{v_l, v_r}$ lies totally outside of P, where v_l and v_r are the left and right projections of v in the horizontal visibility map. This process is called *regularization* [42]. Figure 19.12 shows a decomposition of the simple polygon in Fig. 19.10(a) into a collection of monotone polygons.

We now describe an algorithm that triangulates a monotone polygon P in linear time. Assume that the monotone polygon has v_0 as the topmost vertex and v_{n-1} as the lowest vertex. We have two polygonal chains from v_0 to v_{n-1}, denoted $\mathcal{L}$ and $\mathcal{R}$, that define the left and right boundary of P, respectively. Note that vertices on these two polygonal chains are already sorted in descending order of their y-coordinates. The algorithm is based on a *greedy* method, i.e., whenever a triangle can be formed by connecting vertices either on the same chain or on opposite chains, we do so immediately. We shall examine the vertices in order, and maintain a polygonal chain $\mathcal{C}$ consisting of vertices whose internal angles are greater than π. Initially $\mathcal{C}$ consists of two vertices v_0 and v_1 that define an edge $\overline{v_0, v_1}$. Suppose $\mathcal{C}$ consists of vertices $v_{i_0}, v_{i_1}, \ldots, v_{i_k}, k \geq 1$. We distinguish two cases for each vertex v_l examined, $l < n-1$. Without loss of generality we assume $\mathcal{C}$ is a left chain, i.e., $v_{i_k} \in \mathcal{L}$. The other case is treated symmetrically.

1. $v_l \in \mathcal{L}$. Let v_{i_j} be the last vertex on $\mathcal{C}$ that is visible from v_l. That is, the internal angle $\angle(v_{i_j}, v_{i_{j'}}, v_l)$, where $j < j' \leq k$, is less than π, and either $v_{i_j} = v_{i_0}$ or the internal angle $\angle(v_{i_{j-1}}, v_{i_j}, v_l)$ is greater than π. Add diagonals $\overline{v_l, v_{i_{j'}}}$, for $j \leq j' < k$. Update $\mathcal{C}$ to be composed of vertices $v_{i_0}, v_{i_1}, \ldots, v_{i_j}, v_l$.
2. $v_l \in \mathcal{R}$. In this case we add diagonals $\overline{v_l, v_{i_{j'}}}$, for $0 \leq j' \leq k$. $\mathcal{C}$ is updated to be composed of v_{i_k} and v_l and it becomes a right chain.

Figures 19.13(a) and (b) illustrate these two cases, respectively, in which the shaded portion has been triangulated.

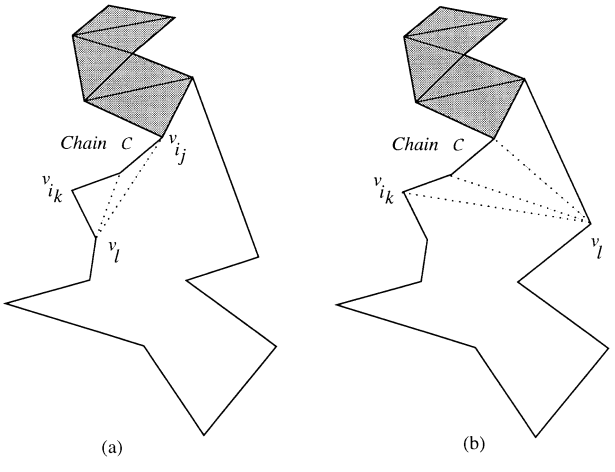


FIGURE 19.13 Triangulation of a monotone polygon.

Fournier and Montuno [26] and independently Chazelle and Incerpi [14] showed that triangulation of a polygon is linear-time equivalent to computing the horizontal visibility map. Based on this result Tarjan and Van Wyk [47] first devised an $O(n \log\log n)$ time algorithm that computes the *horizontal visibility map* and hence, an $O(n \log\log n)$ time algorithm for triangulating a simple polygon. A simpler algorithm of the same complexity was give in [34]. A Las Vegas algorithm with $O(n \log^* n)$ expected time was given

in [18]. A recent breakthrough result of Chazelle [12] finally settled the longstanding open problem, i.e., triangulating a simple polygon in linear time. But the method is quite involved. As a result of this linear triangulation algorithm, a number of problems can be solved asymptotically in linear time. Note that if the polygons have holes, the problem of triangulating the interior is shown to require $\Omega(n \log n)$ time [4].

Planar Straight-Line Graph Triangulation

The triangulation is also known as the *constrained triangulation.* This problem includes the triangulation of a set of points. A triangulation of a given planar straight-line graph $G(V, E)$ with $n = |V|$ vertices is a planar graph $\mathcal{G}(V, \mathcal{E})$ such that $E \subseteq \mathcal{E}$ and each face is a triangle, except the exterior one, which is unbounded. A constrained triangulation $\mathcal{G}(V, \mathcal{E})$ of a $G(V, E)$ can be obtained as follows.

1. Compute the convex hull of the set of vertices, ignoring all the edges. Those edges that belong to the convex hull are necessarily in the constrained triangulation. They define the boundary of the exterior face.
2. Compute the horizontal visibility map for the graph, $G' = G \cup \mathrm{CH}(V)$, where $\mathrm{CH}(V)$ denotes the convex hull of V, i.e., for each vertex, its left and right projections are calculated, and a collection of trapezoids are obtained.
3. Apply the *regularization* process by introducing edges to vertices in the graph G' that are not regular. An isolated vertex requires two edges, one from above and one from below. Regularization will yield a collection of *monotone* polygons that comprise collectively the interior of $\mathrm{CH}(V)$.
4. Triangulate each monotone subpolygon.

It is easily seen that the algorithm runs in time $O(n \log n)$, which is asymptotically optimal. (This is because the problem of sorting is linearly reducible to the problem of constrained triangulation.)

Delaunay and Other Special Triangulations

Sometimes we want to look for *quality* triangulation, instead of just an arbitrary one. For instance, triangles with large or small angles is not desirable. The Delaunay triangulation of a set of points in the plane is a triangulation that satisfies the *empty circumcircle property,* i.e., the circumcircle of each triangle does not contain any other points in its interior. It is well-known that the Delaunay triangulation of points in general position is unique and it will maximize the minimum angle. In fact, the **characteristic angle vector** of the Delaunay triangulation of a set of points is *lexicographically maximum.* The notion of Delaunay triangulation of a set of points can be generalized to a planar straight-line graph $G(V, E)$. That is, we'd like to have G as a subgraph of a triangulation $\mathcal{G}(V, \mathcal{E}')$, $E \subseteq \mathcal{E}'$, such that each triangle satisfies the *empty circumcircle* property: no vertex *visible* from the vertices of triangle is contained in the interior of the circle. This *generalized* Delaunay triangulation is thus a constrained triangulation that maximizes the minimum angle. The generalized Delaunay triangulation was first introduced by the author and an $O(n^2)$ (respectively, $O(n \log n)$) algorithm for constructing the generalized triangulation of a planar graph (respectively a simple polygon) with n vertices was given in [37]. As the generalized Delaunay triangulation (also known as *constrained* Delaunay triangulation) is of fundamental importance, we describe in "Constrained Delaunay Triangulation" an optimal algorithm due to Chew [16] that computes the constrained Delaunay triangulation for a planar straight-line graph $G(V, E)$ with n vertices in $O(n \log n)$ time. Triangulations that minimize the maximum angle or maximum edge length [24] were also studied. But if the constraints are on the measure of the triangles, for instance, each triangle in the triangulation must be nonobtuse, then **Steiner points** must be introduced. See Bern and Eppstein (in [21, p. 23–90]) for a survey of triangulations satisfying different criteria and discussions of triangulations in two and three dimensions. Bern and Eppstein gave an $O(n \log n + \mathcal{F})$ algorithm for constructing a nonobtuse triangulation of polygons using $\mathcal{F}$ triangles. Bern et al. [9] showed that $\mathcal{F}$ is $O(n)$ and gave an $O(n \log n)$

time algorithm for simple polygons without holes, and an $O(n \log^2 n)$ time algorithm for polygons with holes.

The problem of triangulating a set P of points in $\Re^k$, $k \geq 3$ is less studied. In this case the convex hull of P is to be partitioned into $\mathcal{F}$ nonoverlapping simplices, the vertices of which are points in P. A simplex in k-dimensions consists of exactly $k+1$ points, all of which are extreme points. In $\Re^3$ $O(n \log n + \mathcal{F})$ time suffices, where $\mathcal{F}$ is linear if no three points are collinear, and $O(n^2)$ otherwise. See [21] for more references on three dimensional triangulations and Delaunay triangulations in higher dimensions.

Constrained Delaunay Triangulation

Consider a **planar straight-line graph** $G(V, E)$, where V is a set of points in the plane, and edges in E are nonintersecting except possibly at the endpoints. Let $n = |V|$. Without loss of generality we assume that the edges on the convex hull CH(V) are all in E. These edges, if not present, can be computed in $O(n \log n)$ time (cf. "Convex Hulls in Two and Three Dimensions").

In the constrained Delaunay triangulation $G_{DT}(V, \mathcal{E})$ the edges in $\mathcal{E} \setminus E$ are called *Delaunay* edges. It can be shown that two points $p, q \in V$ define a Delaunay edge if there exists a circle $\mathcal{K}$ passing through p and q which does not contain in its interior any other point visible from p and from q.

Let us assume without loss of generality that the points are in general position that no two have the same x-coordinate, and no four are cocircular. Let the points in V be sorted by ascending order of x-coordinate so that $x(p_i) < x(p_j)$ for $i < j$. Let us associate this set V (and graph $G(V, E)$) with it a bounding rectangle R_V with diagonal points $U(u.x, u.y)$, $L(l.x, l.y)$, where $u.x = x(p_n)$, $u.y = \max y(p_i)$, $l.x = x(p_1)$, $l.y = \min y(p_i)$. That is, L is at the lower left corner with x- and y-coordinates equal to the minimum of the x- and y-coordinates of all the points in V, and U is at the upper right corner. Given an edge $\overline{p_i, p_j}$, $i < j$, its x-interval is the interval $(x(p_i), x(p_j))$. The x-interval of the bounding rectangle R_V, denoted by $\mathcal{X}_V$, is the interval $(x(p_1), x(p_n))$. The set V will be recursively divided by vertical lines $\mathcal{L}$'s and so will be the bounding rectangles. We first divide V into two halves V_l and V_r by a line $\mathcal{L}(X = m)$, where $m = \frac{1}{2}(x(p_{\lfloor n/2 \rfloor}) + x(p_{\lfloor n/2 \rfloor + 1}))$. The edges in E that lie totally to the left and to the right of $\mathcal{L}$ are assigned respectively to the left graph $G_l(V_l, E_l)$ and to the right graph $G_r(V_r, E_r)$, which are associated respectively with bounding rectangle R_{v_l} and R_{V_r} whose x-intervals are $\mathcal{X}_{V_l} = (x(p_1), m)$ and $\mathcal{X}_{V_r} = (m, x(p_n))$, respectively. The edges $\overline{p, q} \in E$ that are intersected by the dividing line and that do not *span*[4] the associated x-interval $\mathcal{X}_V$ will each get *cut* into two edges and a *pseudo* point $\epsilon(p, q)$ on the edge is introduced. Two edges, called *half-edges*, $\overline{p, \epsilon(p, q)} \in E_l$ and $\overline{\epsilon(p, q), q} \in E_r$ are created. Figure 19.14(a) shows the creation of half-edges with pseudo points shown in hollow circles. Note that the edge $\overline{p, q}$ in the shaded area is not considered "present" in the associated bounding rectangle, $\overline{u, v}$ spans the x-interval for which no pseudo point is created, and $\overline{s, t}$ is a half-edge that spans the x-interval of the bounding rectangle to the left of the dividing line $\mathcal{L}$.

It can be shown that for each edge $\overline{p, q} \in E$, the number of half-edges so created is at most $O(\log n)$. Within each bounding rectangle the edges that span its x-interval will divide the bounding rectangle into various parts. The constrained Delaunay triangulation gets computed for each part recursively. At the bottom of recursion each bounding rectangle contains at most three vertices of V, the edges incident on them, plus a number of half-edges spanning the x-interval including the pseudo endpoints of half-edges. Figure 19.14(b) illustrates an example of the constrained Delaunay triangulation at some intermediate step. No edges shall intersect one another, except at the endpoints.

As is usually the case for divide-and-conquer paradigm, the **merge** step is the key to the method. We describe below the **merge** step that combines constrained Delaunay triangulations in adjacent bounding rectangles that share a common dividing vertical edge $\mathcal{L}$.

[4] An edge $\overline{p, q}$, $x(p) < x(q)$ is said to span an x-interval (a, b), if $x(p) < a$ and $x(q) > b$.

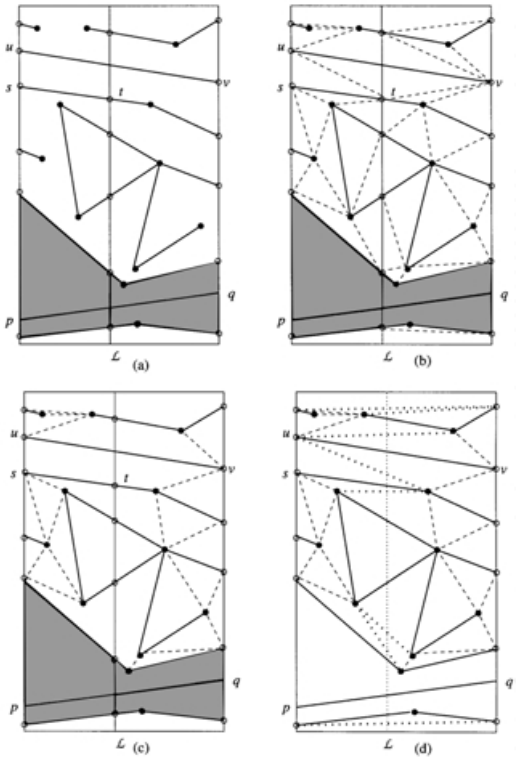


FIGURE 19.14 Computation of constrained Delaunay triangulation for subgraphs in adjacent bounding rectangles.

1. Eliminate the pseudo points along the boundary edge $\mathcal{L}$, including the Delaunay edges incident on those pseudo points. This results in a partial constrained Delaunay triangulation within the union of these two bounding rectangles. Figure 19.14(c) illustrates the partial constrained Delaunay triangulation as a result of the removal of the Delaunay edges incident on the pseudo points on the border of the constrained Delaunay triangulation shown in Fig. 19.14(b).
2. Compute new Delaunay edges that cross $\mathcal{L}$ as follows. Let A and B be the two endpoints of an edge that crosses $\mathcal{L}$ with A on the left and B on the right of $\mathcal{L}$, and $\overline{A, B}$ is known to be part of the desired constrained Delaunay triangulation. That is, either $\overline{A, B}$ is an edge of E or a Delaunay edge just created. Either A or B may be a pseudo point. Let $\overline{A, C}$ be the first edge counterclockwise from edge $\overline{A, B}$. To decide if $\overline{A, C}$ remains to be a Delaunay edge in the desired constrained Delaunay triangulation, we consider the next edge $\overline{A, C_1}$, if it exists. counterclockwise from $\overline{A, C}$. If $\overline{A, C_1}$ does not exist, or $\overline{A, C}$ is in E, $\overline{A, C}$ will remain. Otherwise we test if the circumcircle $\mathcal{K}(A, C, C_1)$ contains B in its interior. If so, $\overline{A, C}$ is eliminated, and the test continues on $\overline{A, C_1}$. Otherwise, $\overline{A, C}$ stays. We do the same thing to test edges incident on B, except that we consider edges incident on B in clockwise direction from $\overline{B, A}$. Assume now we have determined that both $\overline{A, C}$ and $\overline{B, D}$ remain. The next thing to do is to decide which of edge $\overline{B, C}$ and $\overline{A, D}$ should belong to the desired constrained Delaunay triangulation. We apply the circle test: test if circle $\mathcal{K}(A, B, C)$ contains D in the interior. If not, $\overline{B, C}$ is the desired Delaunay edge. Otherwise $\overline{A, D}$ is. We then repeat this step.

Step 2 of the *merge step* is similar to the method is constructing unconstrained Delaunay triangulation given in [37] and can be accomplished in time linear in the number of edges in the combined bounding

rectangle. The dotted lines in Fig. 19.14(d) are the Delaunay edges introduced in Step 2. We therefore conclude with the following theorem.

THEOREM 19.9 *Given a planar straight-line graph $G(V, E)$ with n vertices, the constrained Delaunay triangulation of G can be computed in $O(n \log n)$ time, which is asymptotically optimal.*

An implementation of this algorithm can be found in `http://www.ece.nwu.edu/~theory`.

Other Decompositions

Partitioning a simple polygon into shapes such as convex polygons, **star-shaped polygons,** spiral polygons, etc., has also been investigated. After a polygon has been triangulated one can partition the polygon into star-shaped polygons in linear time. This algorithm provided a very simple proof of the traditional art gallery problem originally posed by Klee, i.e., $\lfloor n/3 \rfloor$ vertex guards are always sufficient to see the entire region of a simple polygons with n vertices. But if a partition of a simple polygon into a minimum number of star-shaped polygons is desired, Keil [33] gave an $O(n^5 N^2 \log n)$ time, where N denotes the number of reflex vertices. However, the problem of *decomposing* a simple polygon into a minimum number of star-shaped parts that may overlap is shown to be **NP-hard** [41]. This problem sometimes is referred to as the *covering* problem, in contrast to the *partitioning* problem, in which the components are not allowed to overlap. The problem of partitioning a polygon into a minimum number of convex parts can be solved in $O(N^2 n \log n)$ time [33]. It is interesting to note that it may not be possible to partition a simple polygon into convex quadrilaterals, but it is always possible for rectilinear polygons. The problem of determining if a convex quadrilateralization of a polygonal region (with holes) exists is NP-complete. It is interesting to note that $\lfloor n/4 \rfloor$ vertex guards are always sufficient for the art gallery problem in a rectilinear polygon. An $O(n \log n)$ algorithm for computing a convex quadrilateralization or positioning at most $\lfloor n/4 \rfloor$ guards is known (see [41] for more information). The minimum covering problem by star-shaped polygons for rectilinear polygons is still *open.* For variations and results of art gallery problems the reader is referred to [41, 46]. Polynomial time algorithms for computing the minimum partition of a simple polygon into simpler parts while allowing Steiner points can be found in [4].

The minimum partition or covering problem for simple polygons becomes NP-hard when the polygons are allowed to have *holes* [33]. Asano et al. [3] showed that the problem of partitioning a simple polygon with h holes into a minimum number of trapezoids with two horizontal sides can be solved in $O(n^{h+2})$ time, and that the problem is NP-complete if h is part of the input. An $O(n \log n)$ time 3-approximation algorithm was presented.

The problem of partitioning a rectilinear polygon with holes into a minimum number of rectangles (allowing Steiner points) arises in VLSI artwork data. Imai and Asano [32] gave an $O(n^{3/2} \log n)$ time and $O(n \log n)$ space algorithm for partitioning a rectilinear polygon with holes into a minimum number of rectangles (allowing Steiner points). The problem of covering a rectilinear polygon (without holes) with a minimum number of rectangles, which is a special case of a *Boolean basis problem,* however, is NP-hard [39].

Given a polyhedron with n vertices and r notches (features causing nonconvexity), $\Omega(r^2)$ convex components are required for a complete convex decomposition in the worst case. Chazelle and Palios [15] gave an $O((n + r^2) \log r)$ time $O(n + r^2)$ space algorithm for this problem. Bajaj and Dey addressed a more general problem where the polyhedron may have holes and internal voids [8]. The problem of minimum partition into convex parts and the problem of determining if a nonconvex polyhedron can be partitioned into tetrahedra without introducing Steiner points are NP-hard [44].

19.6 Research Issues and Summary

We have covered in this chapter a number of topics in computational geometry, including convex hulls, maximal-finding problems, decomposition, and maxima searching in monotone matrices. Results, some of which are classical, and some which represent the state of the art of this field, were presented. More topics will be covered in the next chapter (Chapter 20).

In "Convex Layers of a Planar Set" an optimal algorithm for computing the layers of planar convex hulls is presented. It shows an interesting fact: within the same time as computing the convex hull (the outermost layer) of a point set in two dimensions, one can compute *all* layers of convex hull. Whether or not one can do the same for higher dimensions $\Re^k$, $k > 2$ remains to be seen. It is known that finding a minimum covering of a simple polygon by star-shaped polygons is NP-hard, but the problem for rectilinear polygons, however, is still *open,* Although the triangulation problem of a simple polygon has been solved by Chazelle [12], the algorithm is far from being practical. As this problem is at the heart of this field, a simpler and more practical algorithm is of great interest.

19.7 Defining Terms

Asymptotic time or space complexity: Asymptotic behavior of the time (or space) complexity of an algorithm when the size of the problem approaches infinity. This is usually denoted in big-Oh notation of a function of input size. A time or space complexity $T(n)$ is $O(f(n))$ means that there exists a constant $c > 0$ such that such that $T(n) \leq c \cdot f(n)$ for sufficiently large n, i.e., $n > n_0$, for some n_0.

CAD/CAM: Computer-aided design and computer-aided manufacturing, a discipline that concerns itself with the design and manufacturing of products aided by a computer.

Characteristic angle vector: A vector of minimum angles of each triangle in a triangulation arranged in nondescending order. For a given point set the number of triangles is the same for all triangulations, and therefore each of these triangulation has a characteristic angle vector.

Divide-and-Marriage-before-Conquest: A problem-solving paradigm derived from divide-and-conquer. A term coined by the Kirkpatrick and Seidel [35], authors of this method. After the divide step in a divide-and-conquer paradigm, instead of conquering the subproblems by recursively solving them, a merge operation is performed first on the subproblems. This method is proven more effective than conventional divide-and-conquer for some applications.

Extreme point: A point in S is an extreme point if it cannot be expressed as a convex combination of other points in S. A convex combination of points $p_1, p_2, \ldots, p_n$ is $\Sigma_{i=1}^{n} \alpha_i p_i$, where $\alpha_i, \forall i$ is nonnegative, and $\Sigma_{i=1}^{n} \alpha_i = 1$.

Geometric duality: A transform between a point and a hyperplane in $\Re^k$ that preserves incidence and order relation. For a point $p = (\mu_1, \mu_2, \ldots, \mu_k)$, its dual $\mathcal{D}(p)$ is a hyperplane denoted by $x_k = \Sigma_{j=1}^{k-1} \mu_j x_j - \mu_k$; for a hyperplane $\mathcal{H} : x_k = \Sigma_{j=1}^{k-1} \mu_j x_j + \mu_k$, its dual $\mathcal{D}(\mathcal{H})$ is a point denoted by $(\mu_1, \mu_2, \ldots, -\mu_k)$. See [19, 22] for more information.

Height-balanced binary search tree: A data structure used to support membership, insert/delete operations each in time logarithmic in the size of the tree. A typical example is the *AVL* tree or *red-black* tree.

NP-hard problem: A complexity class of problems that are intrinsically *harder* than those that can be solved by a Turing machine in nondeterministic polynomial time. When a decision version of a combinatorial optimization problem is proven to belong to the class of NP-complete problems, which includes well-known problems such as *satisfiability. traveling salesman problem,* etc., an optimization version is NP-hard. For example, to decide if there exist k star-shaped polygons whose union is equal to a given simple polygon, for some parameter k, is NP-complete. The

optimization version, i.e., finding a minimum number of star-shaped polygons whose union is equal to a given simple polygon, is NP-hard.

On-line algorithm: An algorithm is said to be *on-line* if the input to the algorithm is given one at a time. This is in contrast to the *off-line* case where the input is known in advance. The algorithm that works on-line is similar to the off-line algorithms that work incrementally, i.e., it computes a partial solution by considering input data one at a time.

Planar straight-line graph: A graph that can be embedded in the plane without crossings in which every edge in the graph is a straight line segment. It is sometimes referred to as *planar subdivision* or *map.*

Star-shaped polygon: A polygon P in which there exists an interior point p such that all the boundary points of P are visible from p. That is, for any point q on the boundary of P, the intersection of the line segment $\overline{p, q}$ with the boundary of P is the point q itself.

Steiner point: A point that is not part of the input set. It is derived from the notion of Steiner tree. Consider a set of three points determining a triangle $\Delta(a, b, c)$ all of whose angles are smaller than 120°, in the Euclidean plane, finding a shortest tree interconnecting these three points is known to require a *fourth* point s in the interior such that each side of $\Delta(a, b, c)$ subtends the angle at s equal to 120°. The optimal tree is called the *Steiner tree* of the three points, and the fourth point is called the *Steiner point.*

Visibility map: A planar subdivision that encodes the visibility information. Two points p and q are *visible* if the straight line segment $\overline{p, q}$ does not intersect any other object. A horizontal (or vertical) visibility map of a planar straight-line graph is a partition of the plane into regions by drawing a horizontal (or vertical) straight line through each vertex p until it intersects an edge e of the graph or extends to infinity. The edge e is said to be horizontally (or vertically) visible from p.

References

[1] Aggarwal, A., Klawe, M.M., Moran, S., Shor, P., and Wilber, R., Geometric Applications of a Matrix-Searching Algorithm, *Algorithmica,* 2(2), 195–208, 1987.

[2] Amato, N. and Preparata, F.P., The Parallel 3D Convex Hull Problem Revisited, *Intl. J. Comput. Geom. & Applications,* 2(2), 163–173, Jun. 1992.

[3] Asano, Ta., Asano, Te., and Imai, H., Partitioning a Polygonal Region into Trapezoids, *J. ACM,* 33(2), 290–312, Apr. 1986.

[4] Asano, Ta. Asano, Te., and Pinter, R.Y., Polygon Triangulation: Efficiency and Minimality, *J. Algorithms,* 7, 221–231, 1986.

[5] Atallah, M.J., Parallel Techniques for Computational Geometry, *Proceedings of IEEE,* 80(9), 1435–1448, Sep. 1992.

[6] Atallah, M.J. and Kosaraju, S.R., An Efficient Algorithm for Maxdominance with Applications, *Algorithmica,* 4, 221–236, 1989.

[7] Avis, D., Bremner, D., and Seidel, R., How Good are Convex Hull Algorithms, *Computational Geometry: Theory and Applications,* 7(5/6), 265–301, Apr. 1997.

[8] Bajaj, C. and Dey, T.K., Convex Decomposition of Polyhedra and Robustness, *SIAM J. Comput.,* 21, 339–364, 1992.

[9] Bern, M., Mitchell, S., and Ruppert, J., Linear-Size Nonobtuse Triangulation of Polygons, *Proc. 10th Annual ACM Symp. Comput. Geometry,* 221–230, Jun. 1994.

[10] Chan, T.M., Output-Sensitive Results on Convex Hulls, Extreme Points, and Related Problems, *Proc. 11th ACM Annual Symp. on Computational Geometry,* 10–19, Jun. 1995.

[11] Chazelle, B., On the Convex Layers of a Planar Set, *IEEE Trans. Inform. Theory,* IT-31, 509–517, 1985.

[12] Chazelle, B., Triangulating a Simple Polygon in Linear Time, *Discrete & Comput. Geometry,* 6, 485–524, 1991.
[13] Chazelle, B., An Optimal Convex Hull Algorithm for Point Sets in Any Fixed Dimension, *Discrete & Computational Geometry,* 8, 145–158, 1993.
[14] Chazelle, B. and Incerpi, J., Triangulation and Shape-Complexity, *ACM Trans. Graphics,* 3(2), 135–152, 1984.
[15] Chazelle, B. and Palios, L., Triangulating a Non-Convex Polytope, *Discrete & Comput. Geometry,* 5, 505–526, 1990.
[16] Chew, L.P., Constrained Delaunay Triangulations, *Algorithmica,* 4(1), 97–108, 1989.
[17] Clarkson, K.L. and Shor, P.W., Applications of Random Sampling in Computational Geometry, II, *Discrete & Comput. Geometry,* 4, 387–421, 1989.
[18] Clarkson, K.L., Tarjan, R.E., and Van Wyk, C.J., A Fast Las Vegas Algorithm for Triangulating a Simple Polygon, *Discrete & Comput. Geometry,* 4, 423–432, 1989.
[19] de Berg, M., van Kreveld, M., Overmars, M., and Schwarzkopf, O., *Computational Geometry Algorithms and Applications,* Springer-Verlag, 1997.
[20] Dobkin, D.P. and Kirkpatrick, D.G., Fast Detection of Polyhedral Intersection, *Theoret. Comput. Sci.,* 27, 241–253, 1983.
[21] Du, D.Z. and Hwang, F.K., Eds., *Computing in Euclidean Geometry,* World Scientific, Singapore, 1992.
[22] Edelsbrunner, H., *Algorithms in Combinatorial Geometry,* Springer-Verlag, 1987.
[23] Edelsbrunner, H. and Shi, W., An $O(n \log^2 h)$ Time Algorithm for the Three-Dimensional Convex Hull Problem, *SIAM J. Comput.,* 20(2), 259–269, Apr. 1991.
[24] Edelsbrunner, H and Tan, T.S., A Quadratic Time Algorithm for the Minmax Length Triangulation, *SIAM J. Comput.,* 22, 527–551, 1993.
[25] Fortune, S., Computational Geometry, in *Directions in Computational Geometry,* Martin, R., Ed., Information Geometers, 1993.
[26] Fournier, A. and Montuno, D.Y., Triangulating Simple Polygons and Equivalent Problems, *ACM Trans. Graphics,* 3(2), 153–174, 1984.
[27] Gabow, H.N., Bentley, J.L., and Tarjan, R.E., Scaling and Related Techniques for Geometry Problems, *Proc. 16th Annu. ACM Sympos. Theory Comput.,* 135–143, 1984.
[28] Garey, M.R., Johnson, D.S., Preparata, F.P., and Tarjan, R.E., Triangulating a Simple Polygon, *Info. Proc. Lett.,* 7, 175–179, 1978.
[29] Goodman, J.E. and O'Rourke, J., Eds. *The Handbook of Discrete and Computational Geometry,* CRC Press LLC, Boca Raton, FL, 1997.
[30] Hershberger, J. and Suri, S., Matrix Searching with the Shortest Path Metric, *Proc. 25th ACM Symp. Theory of Comput.,* 485–494, 1993, *SIAM J. Comput.,* (to appear).
[31] Houle, M.E., Imai, H., Imai, K., Robert, J.-M., and Yamamoto, P., Orthogonal Weighted Linear L_1 and L_∞ Approximation and Applications, *Discrete Appl. Math.,* 43, 217–232, 1993.
[32] Imai, H. and Asano, Ta., Efficient Algorithms for Geometric Graph Search Problems, *SIAM J. Comput.,* 15(2), 478–494, May 1986.
[33] Keil, J.M., Decomposing a Polygon into Simpler Components, *SIAM J. Comput.,* 14, 799–817, 1985.
[34] Kirkpatrick, D.G., Klawe, M.M., and Tarjan, R.E., Polygon Triangulation in $O(n \log \log n)$ Time with Simple Data Structures, *Proc. 6th Annual ACM Symp. Comput. Geometry,* 34–43, 1990.
[35] Kirkpatrick, D.G. and Seidel, R., The Ultimate Planar Convex Hull Algorithm? *SIAM J. Comput.,* 15(1), 287–299, Feb. 1986.
[36] Lee, D.T., Computational Geometry, *Computer Science and Engineering Handbook,* Tucker, A., Ed., CRC Press, Boca Raton, FL, 111–140, 1996.

[37] Lee, D.T. and Lin, A.K., Generalized Delaunay Triangulation for Planar Graphs, *Discrete & Comput. Geometry,* 1, 201–217, 1986.
[38] Lee, D.T. and Wu, Y.F., Geometric Complexity of Some Location Problems, *Algorithmica,* 1, 193–211, 1986.
[39] Lubiw, A., The Boolean Basis Problem and How to Cover Some Polygons with Rectangles, *SIAM J. Disc. Math.,* 3(1), 98–115, Feb. 1990.
[40] Matoušek, J. and Schwarzkopf, O., A Deterministic Algorithm for the Three-Dimensional Diameter Problem, *Proc. 25th ACM Symp. on Theory of Comput.,* 478–484, May 1993.
[41] O'Rourke, J., *Art Gallery Theorems and Algorithms,* Oxford University Press, New York, 1987.
[42] Preparata, F.P. and Shamos, M.I., *Computational Geometry: An Introduction,* Springer-Verlag, 1988.
[43] Ramos, E.A., Construction of 1-D Lower Envelopes and Applications, *Proc. 13th Annual ACM Symp. Comput. Geometry,* 57–66, 1997.
[44] Ruppert,J. and Seidel, R., On the Difficulty of Triangulating Three-Dimensional Non-convex Polyhedra. *Discrete & Comput. Geometry,* 7, 227–253, 1992.
[45] Sack, J. and Urrutia, J., *Handbook of Computational Geometry,* Elsevier, Sci. Publishers, Amsterdam, 1997.
[46] Shermer, T.C., Recent Results in Art Galleries, *Proceedings IEEE,* 80(9), 1384–1399, Sep. 1992.
[47] Tarjan, R.E. and Van Wyk, C.J., An $O(n \log\log n)$-time Algorithm for Triangulating a Simple Polygon, *SIAM J. Comput.,* 17(1), 143–178, Feb. 1988. Erratum: 17(5), 1061, 1988.
[48] Toussaint, G.T., New Results in Computational Geometry Relevant to Pattern Recognition in Practice, in *Pattern Recognition in Practice II,* Gelsema, E.S. and Kanal, L.N., Eds., North-Holland, Amsterdam, 135–146, 1986.
[49] Yao, F.F., Computational Geometry, in *Handbook of Theoretical Computer Science,* Vol. A: Algorithms and Complexity, van Leeuwen, J., Ed., 343–389, 1994.
[50] Yap, C., Towards Exact Geometric Computation, *Computational Geometry: Theory and Applications,* 7(3), 3–23, Feb. 1997.

Further Information

For some problems we present efficient algorithms in pseudo code and for others that are of more theoretical interest we only give a sketch of the algorithms and refer the reader to the original articles. A recent textbook by de Berg et al. [19] contains a very nice treatment of this topic. The reader who is interested in *parallel* computational geometry is referred to [5]. For current research results, the reader may consult the Proceedings of the Annual ACM symposium on Computational Geometry, and the following three journals, *Discrete & Computational Geometry, International Journal of Computational Geometry & Applications,* and *Computational Geometry: Theory and Applications.* More references can be found in [29, 36, 45, 49]. The ftp site /pub/geometry/geombib.tar.gz at ftp.cs.usask.ca contains close to 10,000 entries of bibliography in this field.

David Avis announced a convex hull/vertex enumeration code, *lrs,* based on reverse search and made it available at this site ftp://mutt.cs.mcgill.ca/pub/C/lrs.html. It finds all vertices and rays of a polyhedron in $\Re^k$ for any k, defined by a system of inequalities, and finds a system of inequalities describing the convex hull of a set of vertices and rays. For more details consult the user's manual found at the site. See [7] for more information about other convex hull codes. Those who are interested in the implementations or would like to have more information about other software available can consult `http://www.geom.umn.edu/software/cglist/`.

The following WWW page on *Geometry in Action* maintained by David Eppstein at `http://www.ics.uci.edu/~eppstein/geom.html` and computational geometry page by J. Erickson at `http://www.cs.duke.edu/~jeffe/compgeom` give a comprehensive description of research activities of computational geometry.

20

Computational Geometry II

D. T. Lee
Northwestern University

20.1 Introduction

This chapter is a follow-up of the previous Chapter 19, which dealt with geometric problems and their efficient solutions. The classes of problems that we address in this chapter include proximity, optimization, intersection, searching, point location, and some discussions of geometric software that is under development.

20.2 Proximity

Geometric problems abound pertaining to the questions of how *close* two geometric entities are among a collection of objects or how *similar* two geometric patterns match each other. For example, in pattern classification and clustering, features that are *similar* according to some metric are to be clustered in a group. The two aircrafts that are *closest* at any time instant in the air space will have the largest likelihood

of collision with each other. In some cases one may be interested in how *far apart* or how *dissimilar* the objects are. Some of these *proximity*-related problems will be addressed in this section.

Closest Pair

Consider a set S of n points in $\Re^k$. The *closest pair problem* is to find in S a pair of points whose distance is the minimum, i.e., find p_i and p_j, such that $d(p_i, p_j) = \min_{k \neq l}\{d(p_k, p_l)$, for all points $p_k, p_l \in S\}$, where $d(a, b)$ denotes the Euclidean distance between a and b. (The result below holds for any distance metric in Minkowski's norm.) Enumerating all pairs of distances to find the pair with the minimum distance would take $O(d \cdot n^2)$ time. As is well-known, in one dimension one can solve the problem much more efficiently: Since the closest pair of points must occur consecutively on the real line, one can sort these points and then scan them in order to solve the closest pair problem in $O(n \log n)$ time. The time complexity turns out to be best possible, since the problem has a lower bound of $\Omega(n \log n)$, following from a linear time transformation from the *element uniqueness problem* [85].

But unfortunately there is no total ordering for points in $\Re^k$ for $k \geq 2$, and thus, sorting is not applicable. We will show that using *divide-and-conquer* approach one can solve this problem in $O(n \log n)$ optimal time. Let us consider the case when $k = 2$. In the following we only compute the minimum distance of the closest pair; the actual identity of the closest pair that realizes the minimum distance can be found easily by some straightforward bookkeeping operations. Consider a vertical separating line $\mathcal{V}$ that divides S into S_1 and S_2 such that $|S_1| = |S_2| = n/2$. Let δ_i denote the minimum distance defined by the closest pair of points in S_i, $i = 1, 2$. Observe that the minimum distance defined by the closest pair of points in S is either δ_1, δ_2, or $d(p, q)$ for some $p \in S_1$ and $q \in S_2$. In the former case, we are done. In the latter, points p and q must lie in the vertical strip of width $\delta = \min\{\delta_1, \delta_2\}$ on each side of the separating line $\mathcal{V}$ (Fig. 20.1). The problem now reduces to that of finding the closest pair between points in S_1 and S_2 that lie inside the strip $\mathcal{L}$ of width 2δ. This subset $\mathcal{L}$ of points possesses a special property, known as *sparsity*, i.e., for each square box[1] of length 2δ the number of points in $\mathcal{L}$ is bounded by a constant $c = 4 \cdot 3^{k-1}$, since in each set S_i, there exists no point that lies in the interior of the δ-ball centered at each point in S_i, $i = 1, 2$ [85] (Fig. 20.2). It is this sparsity property that enables us to solve the *bichromatic closest pair problem* in $O(n)$ time.

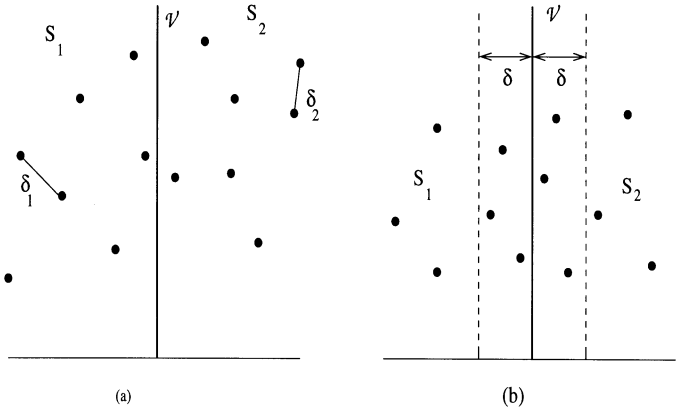


FIGURE 20.1 Divide-and-conquer scheme for closest pair problem.

[1] A box is a hypercube in higher dimensions.

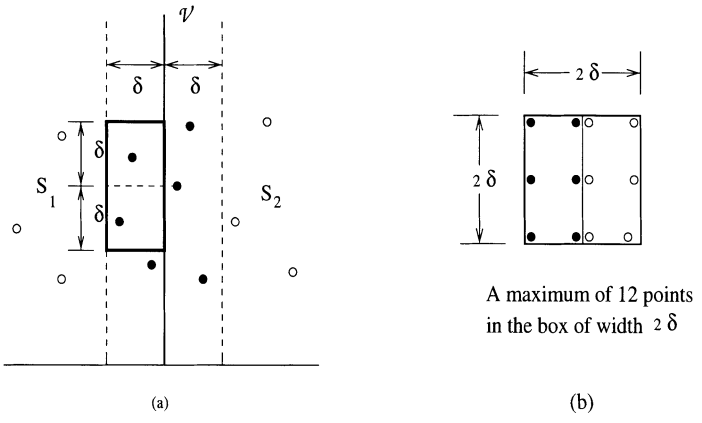


FIGURE 20.2 The box of width 2δ dissected by the separating line has at most 12 points; each point in S_2 needs to examine at most 6 points in S_1 to find its closest neighbor.

The bichromatic closest pair problem is defined as follows. Given two sets of red and blue points, denoted R and B, find the closest pair $r \in R$ and $b \in B$, such that $d(r, b)$ is minimum among all possible distances $d(u, v), u \in R, v \in B$. Let $\overline{S}_i \subseteq S_i$ denote the set of points that lie in the vertical strip. In two dimensions, the sparsity property ensures that for each point $p \in \overline{S}_1$ the number of candidate points $q \in \overline{S}_2$ for the closest pair is at most six (Fig. 20.2). We therefore can scan these points $\overline{S}_1 \cup \overline{S}_2$ in order along the separating line $\mathcal{V}$ and compute the distance between each point in $\overline{S}_1$ (respectively, $\overline{S}_2$) scanned and its six candidate points in $\overline{S}_2$ (respectively, $\overline{S}_1$). The pair that gives the minimum distance δ_3 is the bichromatic closest pair. The minimum distance of all pairs of points in S is then equal to $\delta_S = min\{\delta_1, \delta_2, \delta_3\}$.

Since the merge step takes linear time, the entire algorithm takes $O(n \log n)$ time. This idea generalizes to higher dimensions, except that to ensure the sparsity property of the set $\mathcal{L}$, the separating hyperplane should be appropriately chosen so as to obtain an $O(n \log n)$ time algorithm [85], which is asymptotically optimal.

We note that the bichromatic closest pair problem is in general more difficult than the closest pair problem. Edelsbrunner and Sharir [46] showed that in three dimensions the number of possible closest pairs is $O((|R| \cdot |B|)^{2/3} + |R| + |B|)$. Agarwal et al. [2] gave an $O(n^{2(1-1/(\lceil k/2 \rceil + 1))+\epsilon})$ time algorithm and a randomized algorithm with an expected running time of $O(n^{4/3} \log^c n)$ for some constant c, where $n = |R| + |B|$. Only when the two sets possess the sparsity property defined above can the problem be solved in $O(n \log n)$ time, where $n = |R| + |B|$. A more general problem, known as *fixed radius all nearest-neighbor problem in a sparse set* [85], i.e., given a set M of points in $\Re^k$ that satisfies the sparsity condition, find all pairs of points whose distance is less than a given parameter δ, can be solved in $O(|M| \log |M|)$ time [85].

The closest pair of vertices u and v of a simple polygon P such that $\overline{u, v}$ lies totally within P can be found in linear time [57]; $\overline{u, v}$ is also known as a diagonal of P.

Voronoi Diagrams

The Voronoi diagram $\mathcal{V}(S)$ of a set S of points, called *sites*, $S = \{p_1, p_2, \ldots, p_n\}$ in $\Re^k$ is a partition of $\Re^k$ into Voronoi cells $V(p_i), i = 1, 2, \ldots, n$, such that each cell contains points that are closer to site p_i than to any other site $p_j, j \neq i$, i.e.,

$$V(p_i) = \left\{ x \in R^k \middle| d(x, p_i) \leq d(x, p_j) \forall p_j \in \Re^k, j \neq i \right\}.$$

In two dimensions, $\mathcal{V}(S)$ is a planar graph and is of size linear in $|S|$. In dimensions $k \geq 2$, the total number of d-faces of dimensions $d = 0, 1, \ldots, k-1$, in $\mathcal{V}(S)$ is $O(n^{\lceil d/2 \rceil})$.

Figure 20.3(a) shows the Voronoi diagram of 16 point sites in two dimensions. Figure 20.3(b) shows the straight-line dual graph of the Voronoi diagram, which is called the Delaunay triangulation (cf. "Triangulation" of Chapter 19). In this triangulation the vertices are the sites, and two vertices are connected by an edge if their Voronoi cells are adjacent.

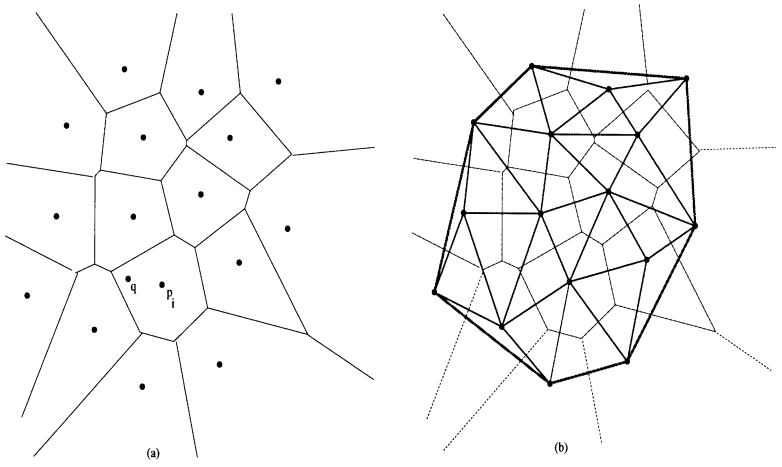


FIGURE 20.3 The Voronoi diagram of a set of 16 points in the plane.

Construction of Voronoi Diagrams in Two Dimensions

The Voronoi diagram possesses many proximity properties. For instance, for each site p_i, the closest site must be among those whose Voronoi cells are adjacent to $V(p_i)$. Thus, the closest pair problem for S in $\Re^2$ can be solved in linear time after the Voronoi diagram has been computed. Since this pair of points must be adjacent in the Delaunay triangulation, all one has to do is to examine all adjacent pairs of points and report the pair with the smallest distance. A divide-and-conquer algorithm to compute the Voronoi diagram of a set of points in the L_p-metric for all $1 \leq p \leq \infty$ is known [64]. There are a rich body of literature concerning the Voronoi diagram. The interested reader is referred to recent surveys [41, 88].

We give below a brief description of a plane-sweep algorithm, known as the *wavefront approach,* due to Dehne and Klein [39]. Let $S = \{p_1, p_2, \ldots, p_n\}$ be a set of point sites in $\Re^2$ sorted in ascending x-coordinate value, i.e., $x(p_1) < x(p_2) < \cdots < x(p_n)$. Consider that we sweep a vertical line $\mathcal{L}$ from left to right and as we sweep $\mathcal{L}$, we compute the Voronoi diagram $\mathcal{V}(S_t)$, where

$$S_t = \{p_i \in S \,|x\,(p_i) \; < t\} \cup \{\mathcal{L}_t\} \; .$$

Here $\mathcal{L}_t$ denotes the vertical line whose x-coordinate equals t. As is well known, $\mathcal{V}(S_t)$ will contain not only straight line segments, which are portions of perpendicular *bisectors* of two point sites, but also parabolic curve segments, which are portions of bisectors of one point site and $\mathcal{L}_t$. The *wavefront* W_t, consisting of a sequence of parabolae, called *waves,* is the boundary of the Voronoi cell $V(\mathcal{L}_t)$ with respect to S_t. Figures 20.4(a) and (b) illustrate two instances, $\mathcal{V}(S_t)$ and $\mathcal{V}(S_{t'})$. Those Voronoi cells that do not contribute to the wavefront are final, whereas those that do will change as $\mathcal{L}$ moves to the right. There are two possible *events* at which the wavefront needs an *update.* One, called *site event,* is when a site is hit by $\mathcal{L}$ and a *new* wave appears. The other, called *spike event,* is when an old wave disappears. Let p_i and p_j be two sites such that the associated waves are adjacent in W_t. The bisector of p_i and p_j defines an

edge of $\mathcal{V}(S_t)$ to the left of W_t. Its extension into the cell $V(\mathcal{L}_t)$ is called a *spike*. The spikes can be viewed as tracks along which two neighboring waves travel. A wave disappears from W_t, once it has reached the point where its two neighboring spikes intersect. In Fig. 20.4(a) dashed lines are spikes and v is a potential spike event point. Without p_7 the wave of p_3 would disappear first and then the wave of p_4. After p_7, a site event, has occurred, a new point v' will be created and it defines an earlier spike event than v. v' will be a spike event point at which the wave of p_4 disappears and waves of p_5 and p_7 become adjacent. Note that the spike event corresponding to v' does not occur at $\mathcal{L}_t$, when $t = x(v')$. Instead, it occurs at $\mathcal{L}_t$, when $t = x(v') + d(v', p_4)$. If there is no site event between $\mathcal{L}_{x(p_7)}$ and $\mathcal{L}_t$, then the wave of p_4 will disappear. It is not difficult to see that after all site events and spikes events have been processed at time τ, $\mathcal{V}(S)$ is identical to $\mathcal{V}(S_\tau)$ with the wavefront removed.

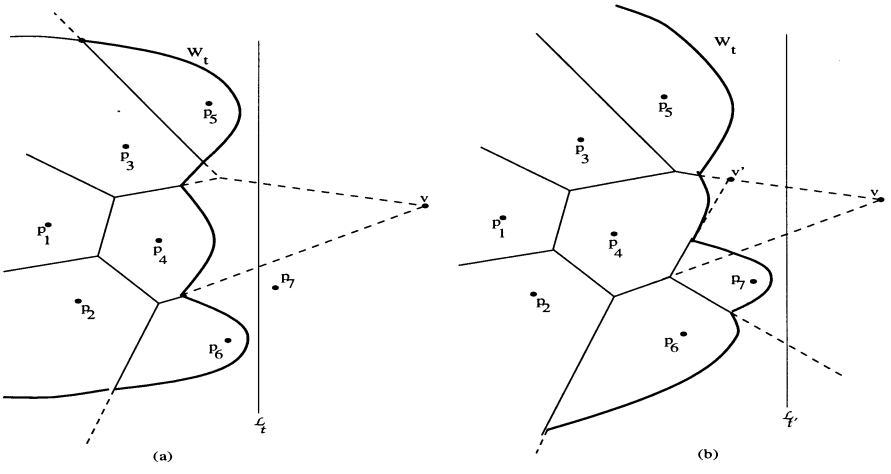


FIGURE 20.4 The Voronoi diagrams of $\mathcal{V}(S_t)$ and $\mathcal{V}(S_{t'})$.

Since the waves in W_t can be stored in a **height-balanced binary search tree** and the site events and spike events can be maintained as a **priority queue,** the overall time and space needed are $O(n \log n)$ and $O(n)$, respectively.

Although $\Omega(n \log n)$ is a lower bound for computing the Voronoi diagram for an arbitrary set of n sites, this lower bound does not apply to special cases, e.g., when the sites are on the vertices of a convex polygon. In fact the Voronoi diagram of a convex polygon can be computed in linear time [6]. This demonstrates further that additional properties of the input can sometimes help reduce the complexity of the problem.

Construction of Voronoi Diagrams in Higher Dimensions

The Voronoi diagrams in $\Re^k$ are related to the convex hulls $\Re^{k+1}$ via a **geometric duality** transformation. Consider a set S of n sites in $\Re^k$, which is the hyperplane $\mathcal{H}^0$ in $\Re^{k+1}$ such that $x_{k+1} = 0$, and a paraboloid $\mathcal{P}$ in $\Re^{k+1}$ represented as $x_{k+1} = x_1^2 + x_2^2 + \cdots + x_k^2$. Each site $p_i = (\mu_1, \mu_2, \ldots, \mu_k)$ is transformed into a hyperplane $\mathcal{H}(p_i)$ in $\Re^{k+1}$ denoted as $x_{k+1} = 2\Sigma_{j=1}^{k} \mu_j x_j - (\Sigma_{j=1}^{k} \mu_j^2)$. That is, $\mathcal{H}(p_i)$ is tangent to the paraboloid $\mathcal{P}$ at point $\mathcal{P}(p_i) = (\mu_1, \mu_2, \ldots, \mu_k, \mu_1^2 + \mu_2^2 + \cdots + \mu_k^2)$, which is just the vertical projection of site p_i onto the paraboloid $\mathcal{P}$. See Fig. 20.5 for an illustration of the transformation in one dimension. The half-space defined by $\mathcal{H}(p_i)$ and containing the paraboloid $\mathcal{P}$ is denoted as $\mathcal{H}^+(p_i)$. The intersection of all half-spaces $\bigcap_{i=1}^{n} \mathcal{H}^+(p_i)$ is a convex body and the boundary of the convex body is denoted $CH(\mathcal{H}(S))$. Any point $q \in \Re^k$ lies in the Voronoi cell $V(p_i)$ if the vertical projection of q onto $CH(\mathcal{H}(S))$ is contained in $\mathcal{H}(p_i)$. The distance between point q and its closest site p_i can be shown to be equal to the square root of the vertical distance between its vertical projection $\mathcal{P}(q)$

on the paraboloid $\mathcal{P}$ and on $CH(\mathcal{H}(S))$. Moreover every κ-face of $CH(\mathcal{H}(S))$ has a vertical projection on the hyperplane $\mathcal{H}^0$ equal to the κ-face of the Voronoi diagram of S in $\mathcal{H}^0$.

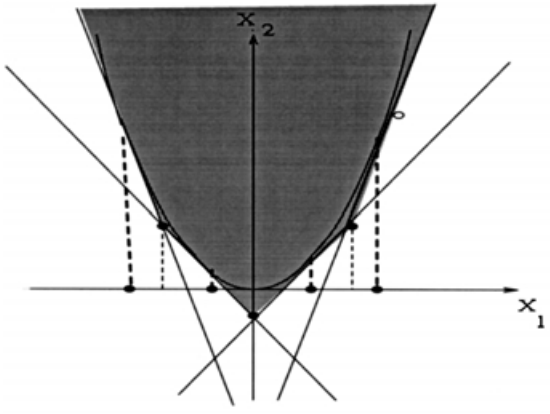


FIGURE 20.5 The paraboloid transformation of a site in one dimension to a line tangent to a parabola.

We thus obtain the result which follows from the theorem for the convex hull in Chapter 19.

THEOREM 20.1 *The Voronoi diagram of a set S of n points in $\Re^k$, $k \geq 2$ can be computed in $O(n \log \mathcal{H})$ time for $k = 2$, and in $O(n \log \mathcal{H} + (n\mathcal{H})^{1-1/(\lfloor (k+1)/2 \rfloor + 1)} \log^{O(1)} n)$ time for $k > 2$, where $\mathcal{H}$ is the number of i-faces, $i = 0, 1, \ldots, k$.*

It has recently been shown that the Voronoi diagram in $\Re^k$, for $k = 3, 4$, can be computed in $O((n + \mathcal{H}) \log^{k-1} \mathcal{H})$ time [11].

Farthest Neighbor Voronoi Diagram

The Voronoi diagram defined in the subsection "Voronoi Diagrams" is also known as the *nearest neighbor* Voronoi diagram. The *nearest neighbor* Voronoi diagram partitions the space into cells such that each site has its own cell, which contains all points that are closer to this site than to any other site. A variation of this partitioning concept is a partition of the space into cells, each of which is associated with a site, and contains all points that are *farther* from the site than from any other site. This diagram is called the *farthest neighbor* Voronoi diagram. Unlike the *nearest neighbor* Voronoi diagram, a farthest neighbor Voronoi diagram only has a subset of sites which have a Voronoi cell associated with them. Those sites that have a nonempty Voronoi cell are those that lie on the convex hull of S. A similar partitioning of the space is known as the order κ*-nearest neighbor* Voronoi diagram, in which each Voronoi cell is associated with a subset of κ sites in S for some fixed integer κ such that these κ sites are the closest among all other sites. For $\kappa = 1$ we have the *nearest neighbor* Voronoi diagram, and for $\kappa = n - 1$ we have the *farthest neighbor* Voronoi diagram. The construction of the order κ*-nearest neighbor* Voronoi diagram in the plane can be found in, e.g., [85]. The order κ Voronoi diagrams in $\Re^k$ are related to the levels of hyperplane arrangements in $\Re^{k+1}$ using the paraboloid transformation discussed in "Construction of Voronoi Diagrams in Higher Dimensions." See, e.g., [1] for details. Below is a discussion of the *farthest neighbor* Voronoi diagram in two dimensions.

Given a set S of sites $s_1, s_2, \ldots, s_n$, the f-neighbor Voronoi cell of site s_i is the locus of points that are farther from s_i than from any other site s_j, $i \neq j$, i.e.,

$$f\_V(s_i) = \left\{ p \in \Re^2 \middle| d(p, s_i) \geq d\left(p, s_j\right), s_i \neq s_j \right\} .$$

The union of these f-neighbor Voronoi cells is called the *farthest neighbor* Voronoi diagram of S. Figure 20.6 shows the *farthest neighbor* Voronoi diagram for a set of 16 sites. Note that only sites that are on the convex hull $CH(S)$ will have a non-empty f-neighbor Voronoi cell [85] and that all the f-neighbor Voronoi cells are unbounded.

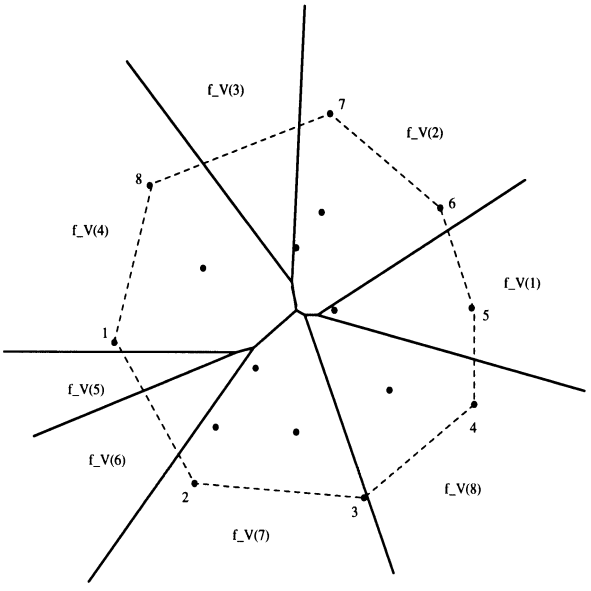


FIGURE 20.6 The farthest neighbor Voronoi diagram of a set of 16 sites in the plane.

Since the *farthest neighbor* Voronoi diagram in the plane is related to the convex hull of the set of sites, one can use the *divide-and-marriage-before-conquest* paradigm to compute the *farthest neighbor* Voronoi diagram of S in two dimensions in time $O(n \log \mathcal{H})$, where $\mathcal{H}$ is the number of sites on the convex hull. Once the convex hull is available, the linear time algorithm [6] for computing the Voronoi diagram for a convex polygon can be applied.

Weighted Voronoi Diagrams

When the sites have weights such that the distance from a point to the sites is weighted, the structure of the Voronoi diagram can be drastically different than the unweighted case. We consider a few examples.

EXAMPLE 20.1: Power Diagrams

Suppose each site s in $\Re^k$ is associated with a nonnegative weight, w_s. For an arbitrary point p in $\Re^k$ the weighted distance from p to s is defined as

$$\delta(s, p) = d(s, p)^2 - w_s^2 .$$

If w_s is positive, and if $d(s, p) \geq w_s$, then $\sqrt{\delta(s, p)}$ is the length of the tangent of p to the ball, $b(s)$, of radius w_s and centered at s. $\delta(s, p)$ is also called the *power of* p with respect to the ball $b(s)$. The locus of points p equidistant from two sites $s \neq t$ of equal weight, will be a hyperplane called the *chordale* of s and t, (see Fig. 20.7). Point q is equidistant to sites a and b and the distance is the length of the tangent line $\overline{q, c} = \overline{q, d}$.

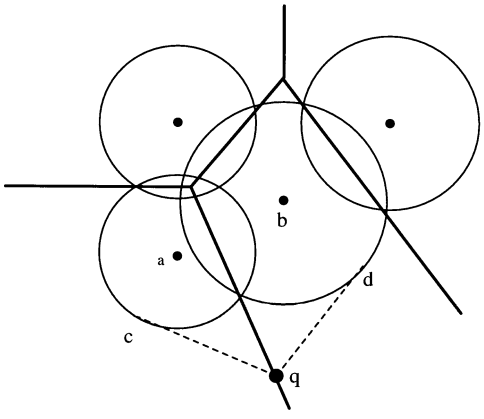


FIGURE 20.7 The power diagram in two dimensions. $\delta(q, a) = \delta(q, b) =$ length of $\overline{q, c}$.

The power diagram in two dimensions can be used to compute the contour of the union of n disks, and the connected components of n disks in $O(n \log n)$ time, and in higher dimensions it can be used to compute the union or intersection of n axis-parallel cones in $\Re^k$ with apexes in a common hyperplane in time $O(CH_{k+1}(n))$, the multiplicative weighted nearest-neighbor Voronoi diagram (defined below) for n points in $\Re^k$ in time $O(CH_{k+2}(n))$, and the Voronoi diagrams for n spheres in $\Re^k$ in time $O(CH_{k+2}(n))$, where $CH_\ell(n)$ denotes the time for constructing the convex hull of n points in $\Re^\ell$ [88]. For the best time bound for $CH_\ell(n)$, consult "Convex Hull" of Chapter 19. For more results on the union of spheres and the volumes see [45].

EXAMPLE 20.2: Multiplicative-Weighted Voronoi Diagrams

Suppose each site $s \in \Re^k$ is associated with a positive weight w_s. The distance from a point $p \in \Re^k$ to s is defined as

$$\delta_{multi-w}(s, p) = d(p, s)/w_s \ .$$

In two dimensions, the locus of points equidistant to two sites $s \neq t$ is a disk, if $w_s \neq w_t$, and a perpendicular bisector of line segment $\overline{s, t}$, if $w_s = w_t$. Each cell associated with a site s consists of all points closer to s than to any other site and may be disconnected. In the worst case the multiplicative-weighted nearest neighbor Voronoi diagram of a set S of n points in two dimensions can have $O(n^2)$ regions and can be computed in $O(n^2)$ time. But in one dimension, the diagram can be computed optimally in $O(n \log n)$ time. On the other hand the multiplicative-weighted farthest neighbor Voronoi diagram has a very different characteristic. Each Voronoi cell associated with a site remains connected, and the size of the diagram is still linear in the number of sites. An $O(n \log^2 n)$ time algorithm for constructing such a diagram is given in [69]. See [89] for more applications of the diagram.

EXAMPLE 20.3: Additive-Weighted Voronoi Diagrams

Suppose each site $s \in \Re^k$ is associated with a positive weight w_s. The distance of a point $p \in \Re^k$ to a site s is defined as

$$\delta_{add-w}(s, p) = d(p, s) - w_s \ .$$

In two dimensions, the locus of points equidistant to two sites $s \neq t$ is a branch of a hyperbola if $w_s \neq w_t$, and a perpendicular bisector of line segment $\overline{s, t}$ if $w_s = w_t$. The Voronoi diagram has properties similar to the ordinary unweighted diagram. For example, each cell is still connected and the size of the diagram is linear. If the weights are positive, the diagram is the same as the Voronoi diagram of a set of spheres

centered at site s and of radius w_s, and in two dimensions this diagram for n disks can be computed in $O(n \log n)$ time [15, 81], and in $k \geq 3$ one can use the notion of power diagram (cf. Example 20.1) to compute the diagram [88].

Generalizations of Voronoi Diagrams

We consider two variations of Voronoi diagrams that are of interest and have applications.

EXAMPLE 20.4: Geodesic Voronoi Diagrams

The nearest neighbor geodesic Voronoi diagram is a Voronoi diagram of sites in the presence of obstacles. The distance from point p to a site s, called the *geodesic* distance between p and s, is the length of the shortest path from p to s avoiding all the obstacles (cf. "Shortest Paths in Two Dimensions"). The locus of points equidistant to two sites s and t is in general a collection of hyperbolic segments. The cell associated with a site is the locus of points whose geodesic distance to the site is shorter than to any other site [84]. The farthest neighbor geodesic Voronoi diagram can be similarly defined. Efficient algorithms for computing either kind of geodesic Voronoi diagram for k point sites in an n-sided simple polygon in $O((n+k)\log(n+k))$ time can be found in [84]. Figure 20.8 illustrates the geodesic Voronoi diagram of a set of point sites within a simple polygon; the whole shaded region is $V(s_i)$.

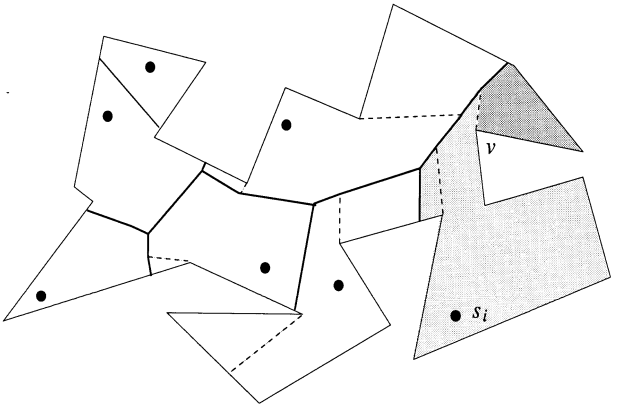


FIGURE 20.8 The geodesic Voronoi diagram within a simple polygon.

EXAMPLE 20.5: Skew Voronoi Diagrams

Most recently a *directional* distance function between two points in the plane is introduced in [8] that models a more realistic distance measure. The distance, called *skew distance,* from point p to point q is defined as

$$\tilde{d}(p,q) = d(p,q) + k \cdot d_y(p,q)\ ,$$

where $d_y(p,q) = y(q) - y(p)$, and $k \geq 0$ is a parameter. This distance function is *asymmetric* and satisfies $\tilde{d}(p,q) + \tilde{d}(q,p) = 2d(p,q)$, and the triangle inequality. Imagine we have a tilted plane $\mathcal{T}$ obtained by rotating the xy-plane by an angle α about the x-axis. The height (z-coordinate) $h(p)$ of a point p on $\mathcal{T}$ is related to its y-coordinate by $h(p) = y(p) \cdot \sin\alpha$.

The distance function defined above reflects the cost that is proportional to the difference of their heights; the distance is *smaller* going *downhill* than going *uphill.* That is, the distance from p to q defined

as $\tilde{d}(p,q) = d(p,q) + \kappa \cdot (h(q) - h(p))$ for $\kappa > 0$ serves this purpose; $\tilde{d}(p,q)$ is less than $\tilde{d}(q,p)$ if $h(q)$ is smaller than $h(p)$.

Because the distance is *directional,* one can define two kinds of Voronoi diagrams defined by the set of sites. A skew Voronoi cell *from* a site p, $\mathcal{V}_{from}(p)$ is defined as the set of points that are closest to p than to any other site. That is,

$$\mathcal{V}_{from}(p) = \left\{ x \,\middle|\, \tilde{d}(p,x) \leq \tilde{d}(q,x) \right\}$$

for all $q \neq p$. Similarly one can define a skew Voronoi cell *to* a site p as follows:

$$\mathcal{V}_{to}(p) = \left\{ x \,\middle|\, \tilde{d}(x,p) \leq \tilde{d}(x,q) \right\}$$

for all $q \neq p$.

The collection of these Voronoi cells for all sites is called the *skew* (or *directional*) Voronoi diagram.

For each site p we define a r-disk centered at p, denoted $from_r(p)$ to be the set of points to which the skew distance from p is r. That is, $from_r(p) = \{x | \tilde{d}(p,x) = r\}$. Symmetrically we can also define a r-disk centered at p, denoted $to_r(p)$ to be the set of points from which the skew distance to p is r. That is, $to_r(p) = \{x | \tilde{d}(x,p) = r\}$. The subscript r is omitted, when $r = 1$. It can be shown that $to_r(p)$ is just a mirror reflection of $from_r(p)$ about the horizontal line passing through p. We shall consider only the skew Voronoi diagram which is the collection of the cells $\mathcal{V}_{from}(p)$ for all $p \in S$.

LEMMA 20.1 For $k > 0$, the unit-disk $from(p)$, is a conic with focus p, directrix the horizontal line at y-distance $1/k$ above p, and eccentricity k. Thus, $from(p)$ is an ellipse for $k < 1$, a parabola for $k = 1$, and a hyperbola for $k > 1$. For $k = 0$, $from(p)$ is a disk with center p (which can be regarded as an ellipse of eccentricity zero).

Note that when k equals 0, the skew Voronoi diagram reduces to the ordinary nearest neighbor Voronoi diagram. When $k < 1$, it leads to known structures: By Lemma 20.1, the skew distance $\tilde{d}$ is a *convex distance function* and the Voronoi diagrams for convex distance functions are well-studied (see, e.g., [88]). They consist of $O(n)$ edges and vertices, and can be constructed in time $O(n \log n)$ by divide-and-conquer.

When $k \geq 1$, since the unit disks are no longer bounded, the skew Voronoi diagrams have different behavior from the ordinary ones. As it turns out, some of the sites do not have nonempty skew Voronoi cells in this case. In this regard, it looks like ordinary farthest neighbor Voronoi diagram discussed earlier.

Let $L_0(p,k)$ denote the locus of points x such that $\tilde{d}(p,x) = 0$. It can be shown that for $k = 1$ $L_0(p,k)$ is a vertical line emanating downwards from p; and for $k > 1$, it consists of two rays, emanating from and extending below p, with slopes $1/(\sqrt{k^2-1})$ and $-1/(\sqrt{k^2-1})$ respectively. Let $N(p,k)$ denote the area below $L_0(p,k)$ (for $k > 1$). Let the *0-envelope,* $E_0(S)$, be the upper boundary of the union of all $N(p,k)$ for $p \in S$. $E_0(S)$ is the upper envelope of the graphs of all $L_0(p,k)$, when being seen as functions of the x-coordinate. For each point u lying above $E_0(S)$, we have $\tilde{d}(p,u) > 0$ for all $p \in S$, and for each point v lying below $E_0(S)$, there is at least one $p \in S$ with $\tilde{d}(p,v) < 0$. See Fig. 20.9 for an example of a 0-envelope (shown as the dashed polygonal line) and the corresponding skew Voronoi diagram. Note that the skew Voronoi cells associated with sites q and t are empty. The following results are obtained [8].

LEMMA 20.2 For $k > 1$, the 0-envelope $E_0(S)$ of a set S of n sites can be computed in $O(n \log \mathcal{H})$ time and $O(n)$ space, where $\mathcal{H}$ is the number of edges of $E_0(S)$.

LEMMA 20.3 Let $p \in S$ and $k > 1$. Then $\mathcal{V}_{from}(p) \neq \emptyset$ if and only if $p \in E_0(S)$. $\mathcal{V}_{from}(p)$ is unbounded if and only if p lies on the upper hull of $E_0(S)$. For $k = 1$ $\mathcal{V}_{from}(p)$ is unbounded for all p.

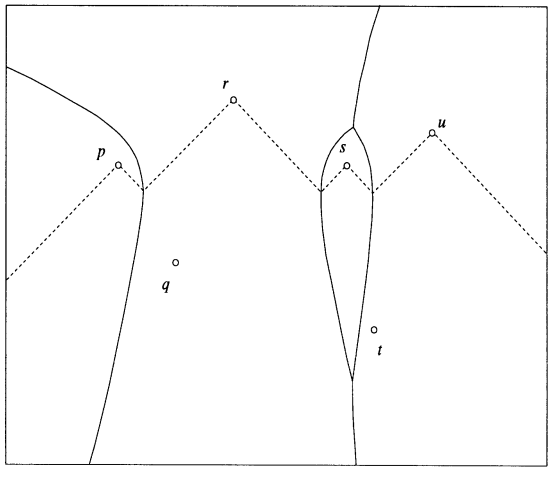


FIGURE 20.9 The 0-envelope and the skew Voronoi diagram when $k = 1.5$.

THEOREM 20.2 *For any $k \geq 0$ the skew Voronoi diagram for n sites can be computed in $O(n \log \mathcal{H})$ time and $O(n)$ space, where $\mathcal{H}$ is the number of non-empty skew Voronoi cells in the resulting Voronoi diagram.*

The sites mentioned so far are point sites. They can be of different shapes. For instance, they can be line segments, or polygonal objects. The Voronoi diagram for the edges of a simple polygon P that divides the interior of P into Voronoi cells is also known as the *medial axis,* or *skeleton* of P [85]. The distance function used can also be convex distance function or other norms.

20.3 Optimization

The geometric optimization problems arise in operations research, VLSI layout, and other engineering disciplines. We give a brief description of a few problems in this category that have been studied in the past.

Minimum Cost Spanning Tree

The minimum (cost) spanning tree, MST, of an undirected, weighted graph $G(V, E)$, in which each edge has a nonnegative weight, is a well studied problem in graph theory and can be solved in $O(|E| \log |V|)$ time [85]. When cast in the Euclidean or other L_p-metric plane in which the input consists of a set S of n points, the complexity of this problem becomes different. Instead of constructing a *complete* graph with edge weight being the distance between its two endpoints, from which to extract an MST, a sparse graph, known as the *Delaunay triangulation* of the point set, is computed. The Delaunay triangulation of S, which is a planar graph, is the straight-line dual of the Voronoi diagram of S. That is, two points are connected by an edge if and only if the Voronoi cells of these two sites share an edge. (cf. Section 19.5 of Chapter 19). It can be shown that the MST of S is a subgraph of the Delaunay triangulation. Since the MST of a planar graph can be found in linear time [85], the problem can be solved in $O(n \log n)$ time. In fact, this is asymptotically optimal, as the closest pair of the set of points must define an edge in the MST, and the closest pair problem is known to have an $\Omega(n \log n)$ lower bound, as mentioned in "Closest Pair."

This problem in dimensions three or higher can be solved in subquadratic time, e.g., in three dimensions, $O((n \log n)^{1.5})$ time is sufficient and in $k \geq 3$ dimensions $O(n^{2(1-1/(\lceil k/2 \rceil + 1))+\epsilon})$ time suffices [2]. Interestingly enough, if we want to find an MST that spans at least k nodes in a planar graph (or in the

Euclidean plane), for some parameter $k \leq n$, then the problem, called k-MST problem, is NP-hard [86]. Approximation algorithms for the k-MST problem can be found in [19, 52].

Steiner Minimum Tree

The *Steiner minimum tree,* SMT, of a set of vertices $S \subseteq V$ in an undirected weighted graph $G(V, E)$ is a spanning tree of $S \cup Q$ for some $Q \subseteq V$ such that the total weight of the spanning tree is minimum. This problem differs from MST in that we need to identify a set $Q \subseteq V$ of so-called *Steiner vertices* so that the total cost of the spanning tree is minimized. Of course if $S = V$, SMT is the same as MST. It is the identification of the Steiner vertices that makes this problem intractable. In the plane, we are given a set S of points and are to find a shortest tree interconnecting points in S, while additional Steiner points are allowed. Both Euclidean and rectilinear (L_1-metric) SMT problems are known to be NP-hard. In the geometric setting, the rectilinear SMT problem arises mostly in VLSI net routing, in which a number of terminals need to be interconnected using horizontal and vertical wire segments using the shortest wire length. As this problem is intractable, heuristics are proposed. For more information, the reader is referred to a recent special issue of *Algorithmica* on Steiner trees, edited by Hwang [59]. Most heuristics for the L_1 SMT problem are based on a classical theorem, known as the *Hanan grid theorem,* which states that the Steiner points of an SMT must be at the grid defined by drawing horizontal and vertical lines through each of the given points. However, when the number of orientations permitted for routing is greater than 2, the Hanan grid theorem no longer holds true. In [68] Lee and Shen established a *multi-level grid* theorem, which states that the Steiner points of an SMT for n points must be at the grid defined by drawing λ lines in the feasible orientation recursively for up to $n - 2$ levels, where λ denotes the number of orientations of the wires allowed in routing. That is, the given points are assumed to be at the 0th level. At each level, λ lines in the feasible orientations are drawn through each *new* grid points created at the previous level. In this λ-geometry plane feasible orientations are assumed to make an angle $i\pi/\lambda$ with the positive x-axis. For the rectilinear case, $\lambda = 2$. Figure 20.10 shows that Hanan grid is insufficient for determining a Steiner SMT for $\lambda = 3$. Steiner point s_3 does not lie on the Hanan grid.

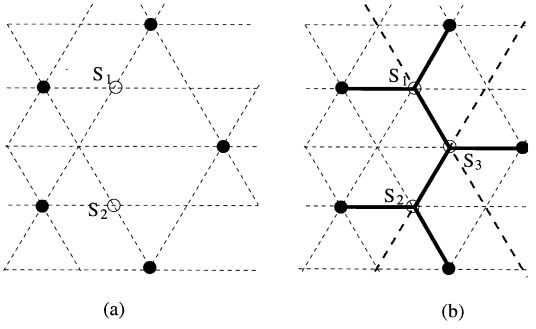


FIGURE 20.10 Hanan grid theorem fails for $\lambda = 3$. Steiner point s_3 does not lie on the Hanan grid.

DEFINITION 20.1 The *performance ratio* of any approximation $\mathcal{A}$ in metric space $\mathcal{M}$ is defined as

$$\rho_{\mathcal{M}}(\mathcal{A}) = \inf_{P \in \mathcal{M}} \frac{L_s(P)}{L_{\mathcal{A}}(P)}$$

where $L_s(P)$ and $L_{\mathcal{A}}(P)$ denote, respectively, the lengths of a Steiner minimum tree and of the approximation $\mathcal{A}$ on P in space $\mathcal{M}$. When the MST is the approximation, the performance ratio is known as the *Steiner ratio,* denoted simply as ρ.

It is well-known that the Steiner ratios for the Euclidean and rectilinear SMTs are $\frac{\sqrt{3}}{2}$ and $\frac{2}{3}$, respectively [59]. The λ-*Steiner ratio*[2] for the λ-geometry SMTs is no less than $\frac{\sqrt{3}\cos(\pi/2\lambda)}{2}$. The following interesting result regarding Steiner ratio is reported in [68], which shows that the Steiner ratio is not an increasing function from $\frac{2}{3}$ to $\frac{\sqrt{3}}{2}$, as λ varies from 2 to ∞.

THEOREM 20.3 *The λ-Steiner ratio is $\frac{\sqrt{3}}{2}$, when λ is multiple of 6, and $\frac{\sqrt{3}\cos(\pi/2\lambda)}{2}$ when λ is multiple of 3 but not multiple of 6.*

Minimum Diameter Spanning Tree

The minimum *diameter* spanning tree MDST of an undirected weighted graph $G(V, E)$ is a spanning tree such that its **diameter,** i.e., total weight of the longest path in the tree is minimum. This arises in applications to communication network where a tree is sought such that the maximum delay, instead of the total cost, is to be minimized. Using a graph-theoretic approach one can solve this problem in $O(|E||V|\log|V|)$ time. However, by the triangle inequality one can show that there exists an MDST such that the longest path in the tree consists of no more than *three* segments [58]. Based on this an $O(n^3)$ time algorithm was obtained.

THEOREM 20.4 *Given a set S of n points, the minimum diameter spanning tree for S can be found in $\theta(n^3)$ time and $O(n)$ space.*

We remark that the problem of finding a spanning tree whose total cost **and** the diameter are both bounded is NP-complete [58]. In [86] the problem of finding a *minimum diameter cost spanning tree* is studied. In this problem for each pair of vertices v_i and v_j there is a weighting function $w_{i,j}$ and the *diameter cost* of a spanning tree is defined to be the maximum over $w_{i,j} * d_{i,j}$, where $d_{i,j}$ denotes the distance between vertices v_i and v_j. To find a spanning tree with minimum diameter cost as defined above is shown to be NP-hard [86].

Another similar problem that arises in VLSI clock tree routing is to find a tree from a source to multiple sinks such that every source-to-sink path is a *shortest* rectilinear path and the total wire length is to be minimized. This problem, also known as *rectilinear Steiner arborescence problem* (see [59]), is still not known to be solvable in polynomial time. The problem is widely believed to be NP-hard. Recently we have shown that the problem of finding a minimum spanning tree such that the longest source-to-sink path is bounded by a given parameter is NP-complete.

Minimum Enclosing Circle

Given a set S of points the problem is to find a smallest disk enclosing the set. This problem is also known as the (unweighted) 1-center problem. That is, find a center such that the maximum distance from the center to the points in S is minimized. More formally, we need to find the center $c \in \Re^2$ such that $\max_{p_j \in S} d(c, p_j)$ is minimized. The weighted 1-center problem, in which, the distance function $d(c, p_j)$ is multiplied by the weight w_j, is a well-known min–max problem, also referred to as the *emergency center problem* in operations research. In two dimensions, the 1-center problem can be solved in $O(n)$ time. The minimum enclosing ball problem in higher dimensions is also solved by using linear programming

[2]The λ-*Steiner ratio* is defined as the greatest lower bound of the length of SMT over the length of MST in the λ-*geometry* plane.

technique [97]. The general p-center problem, i.e., finding p circles whose union contains S such that the maximum radius is minimized, is known to be NP-hard. For a special case when $p = 2$ Eppstein [48] gave an $O(n \log^2 n)$ randomized algorithm based on parametric search technique. For the problem of finding a minimum enclosing ellipsoid for a point set in $\Re^k$ and other types of geometric *location* problem see [47, 97].

Largest Empty Circle

This problem, in contrast to the minimum enclosing circle problem, is to find a circle centered in the interior of the convex hull of the set S of points that does not contain any given point and the radius of the circle is to be maximized. This is mathematically formalized as a max–min problem, the minimum distance from the center to the set is maximized. The weighted version is also known as the *obnoxious center* problem in facility location. For the unweighted version the center must be either at a vertex of the Voronoi diagram for S in the convex hull, or at the intersection of a Voronoi edge and the boundary of the convex hull. $O(n \log n)$ time is sufficient for this problem. Following the same strategy one can solve the largest *empty square* problem for S in $O(n \log n)$ time as well, using the Voronoi diagram in the L_∞-metric [64]. The time complexity of the algorithm is asymptotically optimal, as the maximum gap problem, i.e., finding the maximum gap between two consecutive numbers on the real line, which requires $\Omega(n \log n)$ time, is reducible to this problem [85]. In contrast to the minimum enclosing ellipsoid problem is the largest empty ellipsoid problem, which has also been studied [43].

Largest Empty Rectangle

In "Applications of Convex Hulls" of Chapter 19, we mentioned the *smallest enclosing rectangle* problem. Here we look at the problem of finding a largest rectangle that is empty. That is, given a rectangle containing a set S of n points find the largest area sub-rectangle with sides parallel to those of the original rectangle, whose interior contains no points from S [7]. The problem also arises in document analysis of printed-page layout [16] in which *white* space in the black-and-white image of the form of a maximal empty rectangle is to be recognized. A related problem, called *largest empty corner rectangle* problem is that given two subsets S_l and S_r of S separated by a vertical line, find the largest rectangle containing no other points in S such that lower left corner and upper right corner of the rectangle are in S_l and S_r, respectively. This problem can be solved in $O(n \log n)$ time, where $n = |S|$, using fast matrix searching technique (cf. "Row Maxima Searching in Monotone Matrices" of Chapter 19). With this as a subroutine, one can solve the largest empty rectangle problem in $O(n \log^2 n)$ time [7]. When the points define a rectilinear polygon that is orthogonally convex, the largest empty rectangle that can fit inside the polygon can be found in $O(n\alpha(n))$ time, where $\alpha(n)$ is the slowly growing inverse of Ackermann's function using a result of Klawe and Kleitman [63]. When the polygon P is arbitrary and may contain *holes,* Daniels et al. [37] gave an $O(n \log^2 n)$ algorithm, for finding the largest empty rectangle in P, which matches the best known result when P is a rectilinear polygon [7].

Minimum-Width Annulus

Given a set of S of n points find an annulus (defined by two concentric circles) whose center lies internal to the convex hull of S such that the *width* of the annulus is minimized. This problem arises in dimensional tolerancing and metrology which deals with the specification and measurement of error tolerances in geometric shapes. To measure if a manufactured circular part is *round,* an **ANSI** standard is to use the width of an annulus covering the set of points obtained from a number of measurements. This is known as the *roundness* problem [51, 94]. It can be shown that the center of the annulus can be located at the intersection of the nearest neighbor and the farthest neighbor Voronoi diagrams, as discussed in "Voronoi Diagrams" [51]. The center can be computed in $O(n \log n)$ time [51]. If the input is defined by a

simple polygon P with n vertices, then the problem is to find a minimum-width annulus that contains the boundary of P. The center of the smallest annulus can be located at the medial axis of P [94]. In particular, the problem can be solved in $O(n \log n + k)$, where k denotes the number of intersection points of the medial axis of the simple polygon and the farthest neighbor Voronoi diagram of the vertices of P. In [94] k is shown to be $\theta(n^2)$. However, if the polygon is convex, one can solve this problem in linear time [94]. Note that the minimum-width annulus problem is equivalent to the *best circle approximation* problem, in which a circle approximating a given shape (or a set of points) is sought such that the *error* is minimized. The error of the approximating circle is defined to be the maximum over all distances between points in the set and the approximating circle. To be more precise, the error is equal to one half of width of the smallest annulus. See Fig. 20.11.

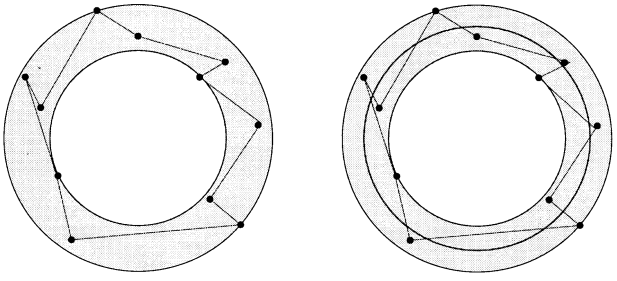


FIGURE 20.11 Minimum-width annulus and best circle approximation.

If the center of the smallest annulus of a point set can be arbitrarily placed, the center *may* lie at infinity and the annulus degenerates to a pair of parallel lines enclosing the set of points. When the center is to be located at infinity, the problem becomes the well-known *minimum-width* problem, which is to find a pair of parallel lines enclosing the set such that the distance between them is minimized. The *width* of a set of n points can be computed in $O(n \log n)$ time, which is optimal [4]. In three dimensions the *width* of a set is also used as a measure for flatness of a *plate,* a so-called *flatness* problem in computational metrology. Chazelle et al. [29] gave an $O(n^{8/5+\epsilon})$ time algorithm for this problem, improving over a previously known algorithm that runs in $O(n^2)$ time.

If, instead of annulus width, the annulus area is to be minimized, then this problem can be solved in linear time via a reduction to fixed-dimensional linear programming [4]. Shermer and Yap [92] introduced the notion of *relative roundness,* where one wants to minimize the ratio of the annulus width and the radius of the inner circle. An $O(n^2)$ algorithm was presented. Duncan et al. [42] define another notion of roundness, called *referenced roundness,* which becomes equivalent to the flatness problem when the radius of the reference circle is set to infinity. Specifically given a reference radius ρ of an annulus A that contains S, i.e., ρ is the mean of the two concentric circles defining the annulus, find an annulus of a minimum width among all annuli with radius ρ containing S, or for a given $\epsilon > 0$, find an annulus containing S whose width is upper bounded by ϵ. They presented an $O(n \log n)$ algorithm for two dimensions and a near quadratic-time algorithm for three dimensions.

20.4 Geometric Matching

Matching in general graphs is one of the classical subjects in combinatorial optimization and has applications in operations research, pattern recognition and VLSI design. Only geometric versions of the matching problem are discussed here. For graph-theoretic matching problems see [82].

Given a weighted undirected complete graph on a set of $2n$ vertices, a *complete matching* is a set of n edges such that each vertex has exactly one edge incident on it. The weight of a matching is the sum of the weights of the edges in the matching. In a metric space, the vertices are points in the plane and the weight of an edge between two points is the distance between them. The *Euclidean minimum weight matching problem* is that given $2n$ points, find n matching pairs of points (p_i, q_i) such that $\Sigma d(p_i, q_i)$ is minimized.

It was not known if geometric properties can be exploited to obtain an algorithm that is faster than the $\theta(n^3)$ algorithm for general graphs (see [82]). Vaidya [96] settled this question in the affirmative. His algorithm is based on a well-studied primal-dual algorithm for weighted matching. Making use of additive weighted Voronoi diagram discussed in "Weighted Voronoi Diagrams" and the range search tree structure (see "Range Searching"), Vaidya solved the problem in $O(n^{2.5} \log^4 n)$ time. This algorithm also generalizes to $\Re^k$ but the complexity is increased by a $\log^k n$ factor.

The *bipartite minimum weight matching problem* is defined similarly, except that we are given a set of red points $R = \{r_1, r_2, \ldots, r_n\}$ and a set of blue points $B = \{b_1, b_2, \ldots, b_n\}$ in the plane, and look for n matching pairs of points $(r, b) \in R \times B$ with minimum cost. In [96] Vaidya gave an $O(n^{2.5} \log n)$ time algorithm for Euclidean metric and an $O(n^2 \log^3 n)$ algorithm for L_1-metric.

If these $2n$ points are given as vertices of a polygon, the problems of minimum weight matching and bipartite matching can be solved in $O(n \log n)$ time if the polygon is convex and in $O(n \log^2 n)$ time if the polygon is simple. In this case the weight of each matching pair of vertices is defined to be the geodesic distance between them [73]. However, if a *maximum* weight matching is sought, an $\log n$ factor can be shaved off [73].

Because of the triangle inequality, one can easily show that in a minimum weight matching the line segments defined by the matched pairs of points cannot intersect one another. Generalizing this *nonintersecting* property the following *geodesic minimum matching* problem in the presence of obstacles can be formulated. Given $2m$ points and polygonal obstacles in the plane, find a matching of these $2m$ points such that the sum of the geodesic distances between matched pairs is minimized. These m paths must not cross each other (they may have portions of the paths overlapping each other). There is no efficient algorithm known to date, except for the obvious method of reducing it to a minimum matching of a complete graph, in which the weight of an edge connecting any two points is the geodesic distance between them. Note that finding a geodesic matching without optimization is trivial, since these m noncrossing paths can always be found. This geodesic minimum matching problem in the general polygonal domain seems nontrivial. The *noncrossing* constraint and the optimization objective function (minimizing total weight) makes the problem hard.

When the matching of these $2m$ points is given *a priori,* finding m noncrossing paths minimizing the total weight seems very difficult. This resembles global routing problem in VLSI for which m 2-terminal nets are given, and a routing is sought that optimizes a certain objective function, including total wire length, subject to some capacity constraints [71]. The noncrossing requirement is needed when single layer routing or planar routing model is used. Global routing problems in general are NP-hard. Since the paths defined by matching pairs in an optimal routing cannot cross each other, paths obtained by earlier matched pairs become *obstacles* for subsequently matched pairs. Thus, the *sequence* in which the pairs of points are matched is very crucial. In fact, the path defined by a matched pair of points need not be the shortest. Thus, to route the matched pairs in a greedy manner sequentially does not give an optimal routing. Consider the configuration shown in Fig. 20.12 in which $R = X, Y, Z$, $B = x, y, z$, and points (X, x), (Y, y) and (Z, z) are to be matched. Note that in this optimal routing, none of the matched pairs is realized by a shortest path, i.e., a straight line. This problem is referred to as the *shortest k-pair noncrossing path* problem. However, if the m matching pairs of points are on the boundary of a simple polygon, and the path must be confined to the interior of the polygon, Papadopoulou [83] gave an $O(n + m)$ algorithm for finding an optimal set of m noncrossing paths, if a solution exists, where n is the number of vertices of the polygon.

Atallah and Chen [14] consider the following bipartite matching problem: Given n red and n blue disjoint isothetic rectangles in the plane, find a matching of these n red-blue pairs of rectangles such that

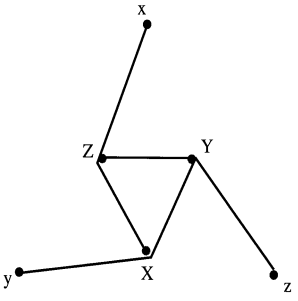


FIGURE 20.12 An instance of 3 noncrossing pair matching problem.

the *rectilinear* paths connecting the matched pairs are noncrossing and monotone. Surprisingly enough, they show that such a matching satisfying the constraints always exists and give an asymptotically optimal $O(n \log n)$ algorithm for finding such a matching.

To conclude this section we remark that the *min–max* versions of the general matching or bipartite matching problems are open. In the red-blue matching if one of the sets is allowed to translate, rotate or scale we have a different matching problem. In this setting we often look for the *best match* according to min-max criterion, i.e., the maximum *error* in the matching is to be minimized. A dual problem can also be defined, i.e., given a maximum error bound, determine if a matching exists, and if so, what kind of *motions* are needed. In [60] Imai et al. called this problem *geometric fitting*.

20.5 Planar Point Location

Planar point location is a fundamental problem in computational geometry. Given a planar subdivision, and a query point, we want to find the region that contains the query point. Figure 20.13 shows an example of a planar subdivision. This problem arises in geographic information systems, in which one often is interested in locating, for example, a certain facility in a map. Consider the skew Voronoi diagram, discussed earlier in "Generalizations of Voronoi Diagrams," for a set S of emergency dispatchers. Suppose an emergency situation arises at a location q and that the nearest dispatcher p is to be called so that the distance $\tilde{d}(p, q)$ is the smallest among all distances $\tilde{d}(r, q)$, for $r \in S$. This is equivalent to locating q in the Voronoi cell $\mathcal{V}_{from}(p)$ of the skew Voronoi diagram that contains q. In situations like this it is vital that the nearest dispatcher be located quickly. We therefore address the point-location problem under the assumption that the underlying planar map is *fixed* and the main objective is to have a *fast* response time to each query. Toward this end we preprocess the planar map into a suitable structure so that it would facilitate the point-location task.

An earlier preprocessing scheme is based on the *slab method* [85], in which parallel lines are drawn through each vertex, thus, partitioning the plane into parallel slabs. Each parallel slab is further divided into subregions by the edges of the subdivision that can be linearly ordered. Any given query point q can thus, be located by two binary searches; one to locate among the $n+1$ horizontal slabs the slab containing q, and followed by another to locate the region defined by a pair of consecutive edges which are ordered from left to right. We use a three tuple, $(P(n), S(n), Q(n))$= (preprocessing time, space requirement, query time) to denote the performance of the search strategy. The slab method gives an $(O(n^2), O(n^2), O(\log n))$ algorithm. Since preprocessing time is only performed once, the time requirement is not as critical as the space requirement, which is permanently engaged. The primary goal of any search strategy is to minimize the query time and the space required. Lee and Preparata [85] first proposed a *chain decomposition* method to decompose a monotone planar subdivision with n points into a collection of $m \leq n$ monotone chains organized in a complete binary tree. Each node in the binary tree is associated with a monotone chain of at most n edges, ordered in y-coordinate. This set of monotone chains forms a totally ordered set partitioning the plane into collections of regions. In particular, between two adjacent chains there are a

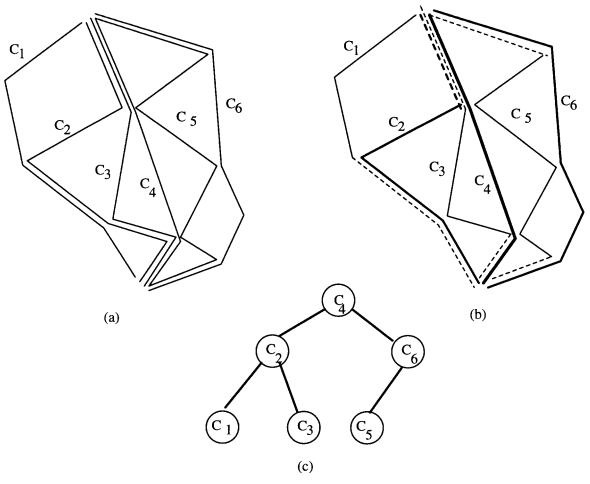


FIGURE 20.13 Chain decomposition method for a planar subdivision.

number of disjoint regions. The point location process begins with the root node of the complete binary tree. When visiting a node, the query point is compared with the node, hence, the associated chain, to decide on which side of the chain the query point lies. Each chain comparison takes $O(\log n)$ time, and total number of nodes visited is $O(\log m)$. The search on the binary tree will lead to two adjacent chains, and hence, identify a region that contains the point. Thus, the query time is $O(\log m \log n) = O(\log^2 n)$. Unlike the slab method in which each edge may be stored as many as $O(n)$ times, resulting in $O(n^2)$ space, it can be shown that with an appropriate chain assignment scheme, each edge in the planar subdivision is stored only once. Thus, the space requirement is $O(n)$. For example, in Fig. 20.13 the edges shared by the root chain C_4 and its descendant chains are assigned to the root chain; in general any edge shared by two nodes on the same root-to-leaf path will be assigned to the node that is an ancestor of the other node. The chain decomposition scheme gives rise to an $(O(n \log n), O(n), O(\log^2 n))$ algorithm. The binary search on the chains is not efficient enough. Recall that after each *chain comparison,* we will move down the binary search tree to perform the next chain comparison and start over another binary search on the *same* y-coordinate of the query point to find an edge of the chain, against which a comparison is made to decide if the point lies to the left or right of the chain. A more efficient scheme is to be able to perform a binary search of the y-coordinate at the root node and to spend only $O(1)$ time per node as we go down the chain tree, shaving off an $O(\log n)$ factor from the query time. This scheme is similar to the ones adopted by Chazelle and Guibas [38, 85] in fractional cascading search paradigm and by Willard [38] in his range tree search method. With the linear time algorithm for triangulating a simple polygon (cf. Section 19.5 of Chapter 19), we conclude with the following optimal search structure for planar point-location.

THEOREM 20.5 *Given a planar subdivision of n vertices, one can preprocess the subdivision in linear time and space such that each point location query can be answered in* $O(\log n)$ *time.*

The point location problem in arrangements of hyperplanes is also of significant interest. See, e.g., [31]. **Dynamic** versions of the point location problem, where the underlying planar subdivision is subject to changes (insertions and deletions of vertices or edges) have also been investigated. See [34] for a survey of dynamic computational geometry.

20.6 Path Planning

This class of problems is mostly cast in the following setting. Given are a set of obstacles O, an object, called *robot,* and an initial and final position, called source and destination, respectively. We wish to find a path for the robot to move from the source to the destination avoiding all the obstacles. This problem arises in several contexts. For instance, in robotics this is referred to as the *piano movers' problem* or *collision avoidance* problem, and in VLSI design this is the *routing* problem for 2-terminal nets. In most applications we are searching for a collision avoidance path that has a *shortest* length, where the distance measure is based on the Euclidean or L_1-metric. For more information regarding motion planning see, e.g., [10].

Shortest Paths in Two Dimensions

In two dimensions, the Euclidean shortest path problem in which the robot is a point, and the obstacles are simple polygons, is well studied. A most fundamental approach is by using the notion of *visibility graph.* Since the shortest path must make turns at polygonal vertices, it is sufficient to construct a graph whose vertices include the vertices of the polygonal obstacles, the source and the destination, and whose edges are determined by vertices that are mutually *visible,* i.e., the segment connecting the two vertices does not intersect the interior of any obstacle. Once the visibility graph is constructed with edge weight equal to the Euclidean distance between the two vertices, one can then apply the Dijkstra's shortest path algorithms [85] to find a shortest path between the source and destination. The Euclidean shortest path between two points is referred to as the *geodesic* path and the distance as the *geodesic* distance. The visibility graph for a set of polygonal obstacles with a total of n vertices can be computed trivially in $O(n^3)$ time. The computation of the visibility graph is the dominating factor for the complexity of any visibility graph based shortest path algorithm. Research results aiming at more efficient algorithms for computing the visibility graph and for computing the geodesic path in time proportional to the size of the graph have been obtained. For example, in [53] Ghosh and Mount gave an output sensitive algorithm that runs in $O(\mathcal{F} + n \log n)$ time for computing the visibility graph, where $\mathcal{F}$ denotes the number of edges in the graph.

Mitchell [75] used the so-called *continuous Dijkstra* wavefront approach to the problem for general polygonal domain of n obstacle vertices and obtained an $O(n^{3/2+\epsilon})$ time algorithm. He constructed a *shortest path map* that partitions the plane into regions such that all points q that lie in the same region have the same vertex sequence in the shortest path from the given source to q. The shortest path map takes $O(n)$ space and enables us to perform shortest path queries, i.e., find a shortest path from the given source to any query points, in $O(\log n)$ time. Hershberger and Suri [56] on the other hand used plane subdivision approach and presented an $O(n \log^2 n)$-time and $O(n \log n)$-space algorithm to compute the *shortest path map* of a given source point. They later improved the time bound to $O(n \log n)$. If the source-destination path is confined in a simple polygon with n vertices, the shortest path can be found in $O(n)$ time [38].

In the context of VLSI routing one is mostly interested in rectilinear paths (L_1-metric) whose edges are either horizontal or vertical. As the paths are restricted to be rectilinear, the shortest path problem can be solved more easily. Lee et al. [70] gave a survey on this topic.

In a two-layer routing model, the number of segments in a rectilinear path reflects the number of *vias,* where the wire segments change layers, which is a factor that governs the fabrication cost. In robotics, a straight line motion is not as costly as *making turns.* Thus, the number of segments (or *turns*) has also become an objective function. This motivates the study of the problem of finding a path with the least number of segments, called the *minimum link path problem* [78, 93].

These two cost measures, length and number of links, are in conflict with each other. That is, a shortest path may have far too many links, whereas a minimum link path may be arbitrarily long compared with a shortest path. A path that is optimal in both criteria is called a *smallest path.* In fact it can be easily

shown that in a general polygonal domain, a smallest path does not exist. However, a smallest rectilinear path in a simple rectilinear polygon exists, and can be found in linear time. Instead of optimizing both measures *simultaneously* one can either seek a path that optimizes a linear function of both length and the number of links, known as the *combined* metric [98] or optimizes them in a lexicographical order. For example, we optimize the length first, and then the number of links, i.e., among those paths that have the same shortest length, find one whose number of links is the smallest and vice versa. In rectilinear case see, e.g., [98]. In [77] algorithms for computing a shortest (in L_2-norm) k-link path in a polygon and for approximating shortest k-link paths in polygons with holes were presented.

A generalization of the collision avoidance problem is to allow collision with a cost. Suppose each obstacle has a weight which represents the cost if the obstacle is *penetrated* [76]. Lee et al. (see [70]) studied this problem in the rectilinear case. They showed that a shortest rectilinear path between two given points in the presence of weighted rectilinear polygons can be found in $O(n \log^{3/2} n)$ time and space. Chen et al. [32] showed that a data structure can be constructed in $O(n \log^{3/2} n)$ time and $O(n \log n)$ space that enables one to find a shortest path from a given source to any query point in $O(\log n + \mathcal{H})$ time, where $\mathcal{H}$ is the number of links in the path. Another generalization is to include in the set of obstacles some subset $F \subset O$ of obstacles, whose vertices are *forbidden* for the solution path to make turns. Of course, when the weight of obstacles is set to be ∞, or the forbidden set $F = \emptyset$, these generalizations reduce to the ordinary collision avoidance problem.

Shortest Paths in Three Dimensions

The Euclidean shortest path problem between two points in a three dimensional polyhedral environment turns out to be much harder than its two dimensional counterpart. Consider a convex polyhedron P with n vertices in three dimensions and two points s, d on the surface of P. A shortest path from s to d on the surface will cross a sequence of edges, denoted $\xi(s, d)$. $\xi(s, d)$ is called the *shortest path edge sequence* induced by s and d and consists of distinct edges. For given s and d, the shortest path from s to d is not unique. However, $\xi(s, d)$ is unique. If $\xi(s, d)$ is known, the shortest path between s and d can be computed by a planar unfolding procedure so that these faces crossed by the path lie in a common plane and the path becomes a straight line segment.

Shortest paths on the surface of a convex polyhedron P possess the following topological properties (i) they do not pass through vertices of P and do not cross an edge of P more than once, (ii) they do not intersect themselves, i.e., they must be simple, and (iii) except for the case of two shortest paths sharing a common subpath, they intersect transversely in at most one point, i.e., they cross each other. If the shortest paths are grouped into *equivalent classes* according to the sequences of edges that they cross, then the number of such equivalent classes, denoted $|\xi(P)|$ is $\theta(n^4)$, where n is the number of vertices of P. These equivalent classes can be computed in $O(|\xi(P)|n^3 \log n)$ time. Chen and Han [33] gave an $O(n^2)$ algorithm for finding a shortest path between a fixed source s and any destination d, where n is the number of vertices and edges of the polyhedron, which may or may not be convex. If s and d lie on the surface of two different polyhedra, $O(N^{O(k)})$ time suffices for computing the shortest path between them amidst a set of k polyhedra, where N denotes the total number of vertices of these obstacles.

The crux of the problem lies in the fact that the number of possible edge sequences may be exponential in the number of obstacles, if s and d lie on the surface of different polyhedra. It was established that the problem of determining the shortest path edge sequence is indeed NP-hard [10]. Figure 20.14 shows an example of a possible shortest path edge sequence induced by s and d in the presence of 3 convex polyhedra. Approximation algorithms for this problem can be found in e.g., Choi et al. [35].

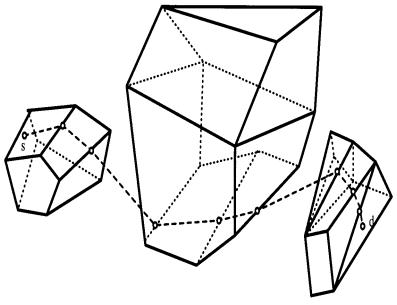


FIGURE 20.14 A possible shortest path edge sequence between points s and d.

20.7 Searching

This class of problems is cast in the form of query-answering. Given is a collection of objects, with preprocessing allowed, one is to find objects that satisfy the queries. The problem can be **static** or dynamic, depending on if the database to be searched is allowed to change over the course of query-answering sessions, and it is studied mostly in two modes, *count mode* and *report mode.* In the former case only the *number* of objects satisfying the query is to be answered, whereas in the latter the actual identity of the objects is to be reported. In the report mode the query time of the algorithm consists of two components, *search time* and *retrieval time,* and expressed as $Q_{\mathcal{A}}(n) = O(f(n) + \mathcal{F})$, where n denotes the size of the database, $f(n)$ a function of n, and $\mathcal{F}$ the size of output. Sometimes we may need to perform some semigroup operations to those objects that satisfy the query. For instance, we may have weights $w(v)$ assigned to each object v, and we want to compute $\Sigma w(v)$ for all $v \cap q \neq \emptyset$. This is referred to as *semigroup range searching* [22]. The semigroup range searching problem is the most general form: if the semigroup operation is set union, we get report-mode range searching problem and if the semigroup operation is just addition (of uniform weight), we have the count-mode range searching problem. We will not discuss the semigroup range searching here. It is obvious that algorithms that handle the report-mode queries can also handle the count-mode queries ($\mathcal{F}$ is the answer). It seems *natural* to expect that the algorithms for count-mode queries would be more efficient (in terms of the order of magnitude of the space required and query time), as they need not search for the objects. However, it was argued that in the report-mode range searching, one could take advantage of the fact that since reporting takes time, the more to report, the *sloppier* the search can be. For example, if we were to know that the ratio $n/\mathcal{F}$ is $O(1)$, we could use a sequential search on a linear list. This notion is known as *filtering search* [85]. In essence more objects than necessary are identified by the search mechanism, followed by a filtering process leaving out unwanted objects. As indicated below the count-mode range searching problem is harder than the report-mode counterpart [23].

Range Searching

This is a fundamental problem in database applications. We'll discuss this problem and the algorithm in the two dimensional space. The generalization to higher-dimensions is straightforward using a known technique, called multidimensional divide-and-conquer [85]. Given is a set of n points in the plane, and the ranges are specified by a product $(l_1, u_1) \times (l_2, u_2)$. We would like to find points $p = (x, y)$ such that $l_1 \leq x \leq u_1$ and $l_2 \leq y \leq u_2$. Intuitively we want to find those points that lie inside a query rectangle specified the range. This is called *orthogonal range searching,* as opposed to other kinds of range searching problems discussed below, e.g., half-space range searching, and simplex range searching, etc. Unless otherwise specified, a range refers to an orthogonal range. We discuss the static case, as this belongs to the class of **decomposable searching problems,** the **dynamization transformation** techniques

can be applied. We note that the range tree structure mentioned below can be made dynamic by using a weight-balanced tree, called $BB(\alpha)$ tree.

For count-mode queries it can be solved by using the locus method as follows. Divide the plane into $O(n^2)$ cells by drawing horizontal and vertical line through each point. The answer to the query q, i.e., find the number of points dominated by q (those points whose x- and y-coordinates are both no greater than those of q), can be found by locating the cell containing q. Let it be denoted by $Dom(q)$. Thus, the answer to the count-mode range-queries can be obtained by some simple arithmetic operations of $Dom(q_i)$ for the four corners, q_1, q_2, q_3, q_4 of the query rectangle. Let q_4 be the northeast corner and q_2 be the southwest corner. The answer will be $Dom(q_4) - Dom(q_1) - Dom(q_3) + Dom(q_2)$. Thus, in $\Re^k$ we have $Q(k,n) = O(k \log n)$, $S(k,n) = P(k,n) = O(n^k)$. To reduce space requirement at the expense of query time has been a goal of further research on this topic. Bentley introduced a data structure, called *range trees* [85]. Using this structure the following results were obtained: for $k \geq 2$, $Q(k,n) = O(\log^{k-1} n)$, $S(k,n) = P(k,n) = O(n \log^{k-1} n)$ (see [67] for more references).

For report-mode queries, by using filtering search technique the space requirement can be further reduced by a $\log \log n$ factor. If the range satisfies additional conditions, e.g., *grounded* in one of the coordinates, say $l_1 = 0$, or the aspect ratio of the intervals specifying the range is fixed, less space is needed. For instance, in two dimensions, the space required is linear (a saving of $\log n / \log \log n$ factor) for these two cases. By using the so-called functional approach to data structures Chazelle [22] developed a *compression* scheme to reduce further the space requirement. Thus, in k-dimensions, $k \geq 2$, for the count-mode range queries we have $Q(k,n) = O(\log^{k-1} n)$ and $S(k,n) = O(n \log^{k-2} n)$ and for report-mode range queries $Q(k,n) = O(\log^{k-1} n + \mathcal{F})$, and $S(k,n) = O(n \log^{k-2+\epsilon} n)$ for some $0 < \epsilon < 1$.

As regards the lower bound of range searching in terms of space-time tradeoffs, Chazelle [23] showed that in k-dimensions, if the query time is $O(\log^c n + \mathcal{F})$ for any constant c, the space required is $\Omega(n(\log n / \log \log n)^{k-1})$ for pointer machine models and the bound is tight for any $c \geq 1$ if $k = 2$, and any $c \geq k - 1 + \epsilon$ (for any $\epsilon > 0$) if $k > 2$. See also [24, 95] for more lower bound results related to orthogonal range searching problems.

Other Range Searching Problems

There are other range searching problems, called simplex range searching problem, and the half-space range searching problems that have been well-studied. A simplex range in $\Re^k$ is a range whose boundary is specified by $k + 1$ hyperplanes. In two dimensions it is a triangle. For this problem there is a lower bound on the query time for simplex range queries: let m denote the space required, $Q(k,n) = \Omega((n/\log n)m^{1/k})$, $k > 2$ and $Q(2,n) = \Omega(n/\sqrt{m})$ [38].

The report-mode half-space range searching problem in the plane can be solved optimally in $Q(n) = O(\log n + \mathcal{F})$ time and $S(n) = O(n)$ space, using geometric duality transform [38]. But this method does not generalize to higher dimensions. In [3] Agarwal and Matoušek obtained a general result for this problem: for $n \leq m \leq n^{\lfloor k/2 \rfloor}$, with $O(m^{1+\epsilon})$ space and preprocessing, $Q(k,n) = O((n/m^{1/\lfloor k/2 \rfloor}) \log n + \mathcal{F})$. As half-space range searching problem is also decomposable, standard dynamization techniques can be applied.

A general method for simplex range searching is to use the notion of *partition tree.* The search space is partitioned in a hierarchical manner using cutting hyperplanes [25], and a search structure is built in a tree structure. Using a **cutting theorem** of hyperplanes Matoušek [74] showed that for k-dimensions, there is a linear space search structure for the simplex range searching problem with query time $O(n^{1-1/k})$, which is optimal in two dimensions and within $O(\log n)$ factor of being optimal for $k > 2$. For more detailed information regarding geometric range searching see [74].

The above discussion is restricted to the case in which the database is a collection of points. One may consider also other kinds of objects, such as line segments, rectangles, triangles, etc., whatever applications may take. The inverse of the orthogonal range searching problem is that of *point enclosure searching problem.* Consider a collection of isothetic rectangles. The point enclosure searching is to find all rectangles that

contain the given query point q. We can cast these problems as *intersection searching* problem, i.e., given a set S of objects, and a query object q, find a subset $\mathcal{F}$ of S such that for any $f \in \mathcal{F}, f \cap q \neq \emptyset$. We have then the rectangle enclosure searching problem, rectangle containment problem, segment intersection searching problem, etc. Janardan and Lopez [61] generalized the intersection searching in the following manner. The database is a collection of *groups* of objects, and the problem is to find all groups of objects intersecting a query object. A group is considered to be intersecting the query object if any object in the group intersects the query object. When each group has only one object, this reduces to the ordinary searching problems.

20.8 Intersection

This class of problems arises in, for example, architectural design, computer graphics, etc. In an architectural design no two objects can share a common region. In computer graphics the well-known hidden-line or hidden-surface elimination problems [40] are examples of intersection problems. This class encompasses two types of problems, *intersection detection* and *intersection computation.*

Intersection Detection

The intersection detection problem is of the form: Given a set of objects, do any two intersect? For instance, given n line segments in the plane, are there two that intersect? The intersection detection problem has a lower bound of $\Omega(n \log n)$ [85].

In two dimensions the problem of detecting if two polygons of r and b vertices intersect was easily solved in $O(n \log n)$ time, where $n = r + b$ using the red-blue segment intersection algorithm [30]. However, this problem can be reduced in linear time to the problem of detecting the self intersection of a polygonal curve. The latter problem is known as the *simplicity* test and can be solved optimally in linear time by Chazelle's linear time triangulation algorithm (cf. Section 19.5 of Chapter 19). If the two polygons are convex, then $O(\log n)$ suffices in detecting if they intersect [27]. Note that although detecting if two convex polygons intersect can be done in logarithmic time, detecting if the boundary of the two convex polygons intersect requires $\Omega(n)$ time [27]. Mount [79] investigated the intersection detection of two simple polygons and computed a separator of m links in $O(m \log^2 n)$ time if they don't intersect.

In three dimensions, detecting if two convex polyhedra intersect can be solved in linear time [27] by using a hierarchical representation of the convex polyhedron, or by formulating it as a linear programming problem in 3 variables.

Intersection Reporting/Counting

One of the simplest such intersecting reporting problems is that of *reporting* pairwise intersection, e.g., intersecting pairs of line segments in the plane. An earlier result due to Bentley and Ottmann [85] used the plane sweep technique that takes $O((n + \mathcal{F}) \log n)$ time, where $\mathcal{F}$ is the output size. This is based on the observation that the line segments intersected by a vertical sweep-line can be ordered according to the y-coordinates of their intersection with the sweep-line, and the sweep-line status can be maintained in logarithmic time per event point, which is either an endpoint of a line segment or the intersection of two line segments. It is not difficult to see that the lower bound for this problem is $\Omega(n \log n + \mathcal{F})$; thus, the above algorithm is $O(\log n)$ factor from the optimal. This segment intersection reporting problem has been solved optimally by Chazelle and Edelsbrunner [28], who used several important algorithm design and data structuring techniques, as well as some crucial combinatorial analysis. In contrast to this asymptotically time-optimal *deterministic* algorithm, a simpler randomized algorithm was obtained [36] for this problem which is both time- and space-optimal. That is, it requires only $O(n)$ space (instead of

$O(n+\mathcal{F})$ as reported in [28]). Balaban [17] most recently reported a deterministic algorithm that solves this problem optimally both in time and space.

On a separate front, the problem of finding intersecting pairs of segments from two different sets was considered. This is called *bichromatic line segment* intersection problem.

Chazelle et al. [30] used *hereditary segment trees* structure and fractional cascading and solved both segment intersection reporting and counting problems optimally in $O(n \log n)$ time and $O(n)$ space. (The term $\mathcal{F}$ should be included in case of reporting.) If the two sets of line segments form connected subdivisions, then merging or overlay of these two subdivisions can be computed in $O(n+\mathcal{F})$ [50]. See [5] for more applications of the bichromatic intersection problem.

The *rectangle intersection reporting* problem arises in the design of VLSI circuitry, in which each rectangle is used to model a certain circuitry component. These rectangles are isothetic, i.e., their sides are all parallel to the coordinate axes. This is a well-studied classical problem, and optimal algorithms ($O(n \log n + \mathcal{F})$ time) have been reported (see [67] for references). The k-dimensional hyperrectangle intersection reporting (respectively, counting) problem can be solved in $O(n^{k-2} \log n + \mathcal{F})$ time and $O(n)$ space (respectively, in time $O(n^{k-1} \log n)$ and space $O(n^{k-2} \log n)$). Gupta et al. [55] gave an $O(n \log n \log\log n + \mathcal{F} \log\log n)$ time and linear space algorithm for the *rectangle enclosure reporting* problem that calls for finding all enclosing pairs of rectangles.

Intersection Computation

Computing the actual intersection is a basic problem, whose efficient solutions often lead to better algorithms for many other problems.

Consider the problem of computing the common intersection of half-planes by divide-and-conquer. Efficient computation of intersection of two convex polygons is required during the merge step. The intersection of two convex polygons can be solved very efficiently by plane-sweep in linear time, taking advantage of the fact that the edges of the input polygons are ordered. Observe that in each vertical strip defined by two consecutive sweep-lines, we only need to compute the intersection of two trapezoids, one derived from each polygon [85].

The problem of intersecting two convex polyhedra was first studied by Muller and Preparata [85], who gave an $O(n \log n)$ algorithm by reducing the problem to the problems of intersection detection and convex hull computation. Following this result one can easily derive an $O(n \log^2 n)$ algorithm for computing the common intersection of n half-spaces in three dimensions by divide-and-conquer. However, using geometric duality and the concept of separating plane, Preparata and Muller [85] obtained an $O(n \log n)$ algorithm for computing the common intersection of n half-spaces, which is asymptotically optimal. There appears to be a difference in the approach to solving the common intersection problem of half-spaces in two and three dimensions. In the latter we resorted to geometric duality instead of divide-and-conquer. This *inconsistency* was later resolved. Chazelle [26] combined the hierarchical representation of convex polyhedra, geometric duality, and other ingenious techniques to obtain a linear time algorithm for computing the intersection of two convex polyhedra. From this result several problems can be solved optimally: (1) the common intersection of half-spaces in three dimensions can now be solved by divide-and-conquer optimally, (2) the merging of two Voronoi diagrams in the plane can be done in linear time by observing the relationship between the Voronoi diagram in two dimensions and the convex hull in three dimensions (cf. Section 19.2 of Chapter 19), and (3) the medial axis of a simple polygon or the Voronoi diagram of vertices of a convex polygon [6] can be solved in linear time.

20.9 Research Issues and Summary

We have covered in this chapter a number of topics in computational geometry, including proximity, optimization, planar point location, geometric matching, path planning, searching, and intersection.

These topics discussed here and in the previous chapter are not meant to be exhaustive. New topics arise as the field continues to flourish.

In Section 20.3 we discussed the problems of smallest enclosing circle and the largest empty circle. These are the two extremes: either the circle is empty or it contains *all* the points. The problem of finding a *smallest* (respectively, *largest*) circle containing at least (respectively, at most) k points for some integer $0 \leq k \leq n$ is also a problem of interest. Moreover, the shape of the object is not limited to circles. A number of open problems remain. What is the complexity of the rectilinear Steiner arborescence problem? Given two points s and t in a simple polygon, is it NP-complete to decide whether there exists a path with at most k links and of length at most L? What is the complexity of the shortest k-pair noncrossing path problem discussed in Section 20.4? How fast can one solve the geodesic minimum matching problem for $2m$ points in the presence of polygonal obstacles? Can one solve the largest empty rectangle problem for a rectilinear polygon in $O(n \log n)$ time? The best known algorithm to date runs in $O(n \log^n)$ time [7]. Is $\Omega(n \log n)$ a lower bound of the minimum-width annulus problem? Can the technique used in [51] be applied to the polygonal case to yield an $O(n \log n)$ time algorithm for the minimum-width annulus problem?

Recently researchers in computational geometry begin to address the issues concerning the actual running times of the algorithms and their robustness when the computations in their implementations are not *exact.* It is understood that the real-RAM computation model with an implicit infinite precision arithmetic is unrealistic in practice. In addition to the robustness issue concerning the accuracy of the output of an algorithm, one needs to find a *new* cost measure to evaluate the efficiency of an algorithm. In the infinite-precision model, the asymptotic time complexity was accepted as an adequate cost measure. However, when the input data have a finite precision representation and computation time varies with the precision required, an alternative cost measure is warranted. The notion of the **degree** of a geometric algorithm could be an important cost measure for comparing the efficiency of algorithms when they are actually implemented [72]. This could play a similar role as the asymptotic time complexity has in the past for the real-RAM computation model.

On the applied side there are new efforts put into development of geometric software. A library of geometric software including visualization tools and application programs is under development. A website containing geometric software is maintained at the Geometry Center, University of Minnesota, at http://www.geom.umn.edu/software/cglist. A project, known as the CGAL project [49], is underway by researchers in Europe to organize a system library containing primitive geometric abstract data types useful for geometric algorithm developers. This is concurrent with the LEDA [80] project at the Max-Planck-Institut für Informatik, Saarbrücken, Germany. Other projects related to the efforts of building geometric software or problem solving environment, include GASP, GeoLab, GeoMAMOS, XYZ Geobench, etc. See [66] for more information.

20.10 Defining Terms

ANSI: American National Standards Institute.

Bisector: A bisector of two elements e_i and e_j is defined to be the locus of points equidistant from both e_i and e_j. That is, $\{p | d(p, e_i) = d(p, e_j)\}$. For instance, if e_i and e_j are two points in the Euclidean plane, the bisector of e_i and e_j is the perpendicular bisector to the line segment $\overline{e_i, e_j}$.

Cutting theorem: This theorem [25] states that for any set $\mathcal{H}$ of n hyperplanes in $\Re^k$, and any parameter r, $1 \leq r \leq n$, there always exists a $(1/r)$-cutting of size $O(r^k)$. In two dimensions, a $(1/r)$-cutting of size s is a partition of the plane into s disjoint triangles, some of which are unbounded, such that no triangle in the partition intersects more than n/r lines in $\mathcal{H}$. In $\Re^k$, triangles are replaced by simplices. Such a cutting can be computed in $O(nr^{k-1})$ time.

Decomposable searching problems: A searching problem with query Q is decomposable if there exists an efficiently computable associative, and commutative binary operator @ satisfying the condition: $Q(x, A \cup B) = @(Q(x, A), Q(x, B))$. In other words, one can decompose the searched domain into subsets, find answers to the query from these subsets, and combine these answers to form the solution to the original problem.

Degree of an algorithm or problem: Assume that each input variable is of arithmetic degree 1 and that the arithmetic degree of a polynomial is the common arithmetic degree of its monomials, whose degree is defined to be the sum of the arithmetic degrees of its variables. An algorithm has degree d if its test computation involves evaluation of multivariate polynomials of arithmetic degree d. A problem has degree d if any algorithm that solves it has degree *at least* d [72].

Diameter of a graph: The distance between two vertices u and v in a graph is the sum of weights of the edges of a shortest path between them. (For unweighted graph, it is the number of edges of a shortest path.) The **diameter of a graph** is the maximum among all distances between all possible pairs of vertices.

Dynamic versus static: This refers to cases when the underlying problem domain can be subject to updates, i.e., insertions and deletions. If no updates are permitted, the problem or data structure is said to be static; otherwise, it is said to be dynamic.

Dynamization transformation: A data structuring technique that can transform a static data structure into dynamic one. In so doing, the performance of the dynamic structure will exhibit certain space-time tradeoffs. See, e.g., [65, 85] for more references.

Geometric duality: A transform between a point and a hyperplane in $\Re^k$, that preserves incidence and order relation. For a point $p = (\mu_1, \mu_2, \ldots, \mu_k)$, its dual $\mathcal{D}(p)$ is a hyperplane denoted by $x_k = \Sigma_{j=1}^{k-1} \mu_j x_j - \mu_k$; for a hyperplane $\mathcal{H} : x_k = \Sigma_{j=1}^{k-1} \mu_j x_j + \mu_k$, its dual $\mathcal{D}(\mathcal{H})$ is a point denoted by $(\mu_1, \mu_2, \ldots, -\mu_k)$. There are other duality transformations. What is described in the text is called the *paraboloid* transform. See [38, 44, 85] for more information.

Height-balanced binary search tree: A data structure used to support membership, insert/delete operations each in time logarithmic in the size of the tree. A typical example is the *AVL* tree or *red-black* tree.

Orthogonally convex rectilinear polygon: A rectilinear polygon P is orthogonally convex if every horizontal or vertical segment connecting two points in P lies totally within P.

Priority queue: A data structure used to support insert and delete operations in time logarithmic in the size of the queue. The elements in the queue are arranged so that the element of the *minimum* priority is always at one end of the queue, readily available for delete operation. Deletions only take place at that end of the queue. Each delete operation can be done is constant time. However, since restoring the above property after each deletion takes logarithmic time, we often say that each delete operation takes logarithmic time. A heap is a well-known priority queue.

References

[1] Agarwal, P.K., de Berg, M., Matoušek, J., and Schwarzkopf, O., Constructing Levels in Arrangements and Higher Order Voronoi Diagrams, *Proc. 10th Annual ACM Symp. Comput. Geometry,* 67–75, 1994.

[2] Agarwal, P.K., Edelsbrunner, H., Schwarzkopf, O., and Welzl, E., Euclidean Minimum Spanning Trees and Bichromatic Closest Pairs, *Discrete & Comput. Geometry,* 6(5), 407–422, 1991.

[3] Agarwal, P.K. and Matoušek, J., Dynamic Half-Space Range Reporting and Its Applications, *Algorithmica,* 13(4) 325–345, Apr. 1995.

[4] Agarwal, P.K., Sharir, M., and Toledo, S., Applications of Parametric Searching in Geometric Optimization, *J. Algorithms,* 17, 292–318, 1994.
[5] Agarwal, P.K. and Sharir, M., Red-Blue Intersection Detection Algorithms, with Applications to Motion Planning and Collision Detection, *SIAM J. Comput.,* 19(2), 297–321, 1990.
[6] Aggarwal, A., Guibas, L.J., Saxe, J., and Shor, P.W., A Linear-Time Algorithm for Computing the Voronoi Diagram of a Convex Polygon, *Discrete & Comput. Geometry,* 4(6), 591–604, 1989.
[7] Aggarwal, A. and Suri, S., Fast Algorithms for Computing the Largest Empty Rectangle, *Proc. 3rd Annu. ACM Sympos. Comput. Geom.,* 278–290, 1987.
[8] Aichholzer, O., Aurenhammer, F., Chen, D.Z., Lee, D.T., Mukhopadhyay, A., and Papadopoulou, E., Voronoi Diagrams for Direction-Sensitive Distances, *Proc. 13th Annu. ACM Sympos. Comput. Geom.,* 418–420, 1997.
[9] Alt, H. and Schwarzkopf, O., The Voronoi Diagram of Curved Objects, *Proc. 11th Annual ACM Symp. Comput. Geometry,* 89–97, 1995.
[10] Alt, H. and Yap, C.K., Algorithmic Aspect of Motion Planning: A Tutorial, Part 1 & 2, *Algorithms Review,* 1, 43–77, 1990.
[11] Amato, N.M. and Ramos, E.A., On Computing Voronoi Diagrams by Divide-Prune-and-Conquer, *Proc. 12th Annual ACM Symp. Comput. Geometry,* 166–175, 1996.
[12] Aonuma, H., Imai, H., Imai, K., and Tokuyama, T., Maximin Location of Convex Objects in a Polygon and Related Dynamic Voronoi Diagrams, *Proc. 6th Annual ACM Symp. Comput. Geometry,* 225–234, 1990.
[13] Atallah, M.J., Parallel Techniques for Computational Geometry, *Proceedings of IEEE,* 80(9), 1435–1448, Sep. 1992.
[14] Atallah, M.J. and Chen, D.Z., Applications of a Numbering Scheme for Polygonal Obstacles in the Plane, *Proc. 7th Intl. Symp. Algorithms and Computation,* 1–24, Dec. 1996.
[15] Aurenhammer, F., A Relationship between Gale Transforms and Voronoi Diagrams, *Discrete Appl. Math.,* 28, 83–91, 1991.
[16] Baird, H.S., Jones, S.E., and Fortune, S.J., Image Segmentation by Shape-Directed Covers, *Proc. 10th Intl. Conf. Pattern Recognition,* 820–825, 1990.
[17] Balaban, I.J., An Optimal Algorithm for Finding Segments Intersections, *Proc. 11th Annual Symp. Comput. Geometry,* 211–219, Jun. 1995.
[18] Bespamyatnikh, S.N., An Optimal Algorithm for Closest Pair Maintenance, *Proc. 11th Annual Symp. Comput. Geometry,* 152–161, Jun. 1995.
[19] Blum, A., Ravi, R., and Vempala, S., A Constant-factor Approximation Algorithm for the k-MST Problem, *Proc. 28th Symp. Theory of Comput.,* May 1996.
[20] Boissonnat, J.-D., Sharir, M., Tagansky, B., and Yvinec, M., Voronoi Diagrams in Higher Dimensions Under Certain Polyhedra Distance Functions, *Proc. 11th Annual ACM Symp. Comput. Geometry,* 79–88, 1995.
[21] Callahan, P. and Kosaraju, S.R., Algorithms for Dynamic Closest Pair and n-Body Potential Fields, *Proc. 6th ACM-SIAM Symp. Discrete Algorithms,* 263–272, 1995.
[22] Chazelle, B., A Functional Approach to Data Structures and Its Use in Multidimensional Searching, *SIAM J. Computing,* 17(3), 427–462, Jun. 1988.
[23] Chazelle, B., Lower Bounds for Orthogonal Range Searching, I: The Reporting Case, *J. ACM,* 37(2), 200–212, Apr. 1990.
[24] Chazelle, B., Lower Bounds for Orthogonal Range Searching, II: The Arithmetic Model, *J. ACM,* 37(3), 439–463, Jul. 1990.
[25] Chazelle, B., Cutting Hyperplanes for Divide-and-Conquer, *Discrete & Comput. Geometry,* 9(2), 145–158, 1993.
[26] Chazelle, B., An Optimal Algorithm for Intersecting Three-Dimensional Convex Polyhedra, *SIAM J. Comput.,* 21(4), 671–696, 1992.

[27] Chazelle, B. and Dobkin, D.P., Intersection of Convex Objects in Two and Three Dimensions, *J. ACM,* 34(1), 1–27, 1987.
[28] Chazelle, B. and Edelsbrunner, H., An Optimal Algorithm for Intersecting Line Segments in the Plane, *J. ACM,* 39(1), 1–54, 1992.
[29] Chazelle, B., Edelsbrunner, H., Guibas, L.J., and Sharir, M., Diameter, Width, Closest Line Pair, and Parametric Searching, *Discrete & Computational Geometry,* 8, 183–196, 1993.
[30] Chazelle, B., Edelsbrunner, H., Guibas, L.J., and Sharir, M., Algorithms for Bichromatic Line-Segment Problems and Polyhedral Terrains, *Algorithmica,* 11(2), 116–132, Feb. 1994.
[31] Chazelle, B. and Friedman, J., Point Location among Hyperplanes and Unidirectional Ray-Shooting, *Comput. Geom. Theory Appl.,* 4, 53–62, 1994.
[32] Chen, D., Klenk, K.S., and Tu, H.-Y. T., Shortest Path Queries among Weighted Obstacles in the Rectilinear Plane, *Proc. 11th Annual ACM Symp. Comput. Geometry,* 370–379, Jun. 1995.
[33] Chen, J. and Han, Y., Shortest Paths on a Polyhedron, Part I: Computing Shortest Paths, *Intl. J. Comput. Geometry & Applications,* 6(2), 127–144, 1996.
[34] Chiang, Y.-J. and Tamassia, R., Dynamic Algorithms in Computational Geometry, *Proceedings of IEEE,* 80(9), 1412–1434, Sep. 1992.
[35] Choi, J., Sellen, J., and Yap, C.K., Approximate Euclidean Shortest Path in 3-Space, *Proc. 10th Symp. on Computational Geometry,* 41–48, 1994.
[36] Clarkson, K.L. and Shor, P.W., Applications of Random Sampling in Computational Geometry II, *Discrete & Computational Geometry,* 4, 387–421, 1989.
[37] Daniels, K., Milenkovic, V., and Roth, D., Finding the Largest Area Axis-Parallel Rectangle in a Polygon, *Computational Geometry: Theory & Appl.,* 7(1-2), 125–148, Jan. 1997.
[38] de Berg, M., van Kreveld, M., Overmars, M., and Schwarzkopf, O., *Computational Geometry Algorithms and Applications,* Springer-Verlag, 1997.
[39] Dehne, F. and Klein, R., The Big Sweep: On the Power of the Wavefront Approach to Voronoi Diagrams, *Algorithmica,* 17(1), 19–32, Jan. 1997.
[40] Dorward, S.E., A Survey of Object-Space Hidden Surface Removal, *Intl. J. Computational Geometry & Applications,* 4(3), 325–362, Sep. 1994.
[41] Du, D.Z. and Hwang, F.K., Eds., *Computing in Euclidean Geometry,* World Scientific, Singapore, 1992.
[42] Duncan, C.A., Goodrich, M.T., and Ramos, E.A., Efficient Approximation and Optimization Algorithms for Computational Metrology, *Proc. 8th ACM-SIAM Symp. Discrete Algorithms,* 121–130, 1997.
[43] Dwyer, R.A. and Eddy, W.F., Maximal Empty Ellipsoids, *Proc. 5th ACM-SIAM Symp. Discrete Algorithms,* 98–102, 1994.
[44] Edelsbrunner, H., *Algorithms in Combinatorial Geometry,* Springer-Verlag, 1987.
[45] Edelsbrunner, H., The Union of Balls and Its Dual Shape, *Proc. 9th Annual ACM Symp. Comput. Geometry,* 218–231, 1993.
[46] Edelsbrunner, H. and Sharir, M., A Hyperplane Incidence Problem with Applications to Counting Distances, *Applied Geometry and Discrete Mathematics. The Victor Klee Festschrift,* 253–263, Gritzmann, P. and Sturmfels, B., Eds., DIMACS Series in Discrete Mathematics and Theoretical Computer Science, 1991.
[47] Efrat, A. and Sharir, M., A Near-Linear Algorithm for the Planar Segment Center Problem, *Proc. 5th ACM-SIAM Symp. Discrete Algorithms,* 87–97, 1994.
[48] Eppstein, D., Fast Construction of Planar Two-Centers, *Proc. 8th ACM-SIAM Symp. Discrete Alg.,* 131–138, 1997.
[49] Fabri, A., Giezeman, G., Kettner, L., Schirra, S., and Schönherr, S., The CGAL Kernel: A Basis for Geometric Computation, *Applied Computational Geometry,* Lin and Manocha, Eds., Springer-Verlag, 1996.

[50] Finkle, U. and Hinrichs, K., Overlaying simply connected planar subdivision in linear time, *Proc. 11th Annu. ACM Sympos. Comput. Geom.,* 119–126, 1995.
[51] Garcia-Lopez, J. and Ramos, P.A., Fitting a Set of Points by a Circle, *Proc. 13th Annu. ACM Sympos. Comput. Geom.,* 139–146, 1997.
[52] Garg, N. and Hochbaum, D.S., An $O(\log k)$ approximation algorithm for the k Minimum Spanning Tree Problem in the Plane, *Proc. 26th Symp. Theory of Comput.,* May 1994.
[53] Ghosh, S.K. and Mount, D.M., An Output-Sensitive Algorithm for Computing Visibility Graphs, *SIAM J. Comput.,* 20(5), 888–910, Oct. 1991.
[54] Goodman, J.E. and O'Rourke, J., Eds., *The Handbook of Discrete and Computational Geometry,* CRC Press LLC, Boca Raton, FL, 1997.
[55] Gupta, P., Janardan, R., Smid, M., and Dasgupta, B., The Rectangle Enclosure and Point-Dominance Problems Revisited, *Proc. 11th Annu. ACM Sympos. Comput. Geom.,* 162–171, 1995.
[56] Hershberger, J. and Suri, S., Efficient Computation of Euclidean Shortest Paths in the Plane, *34th Symp. on Foundations of Computer Science,* 508–517, 1993.
[57] Hershberger, J. and Suri, S., Finding a Shortest Diagonal of a Simple Polygon in Linear Time, *Computational Geometry: Theory and Applications,* 7(3), 149–160, Feb. 1997.
[58] Ho, J.M., Chang, C.H., Lee, D.T., and Wong, C.K., Minimum Diameter Spanning Tree and Related Problems, *SIAM J. Comput.,* 20(5), 987–997, Oct. 1991.
[59] Hwang, F.K., Foreword, *Algorithmica,* 7(2/3), 119–120, 1992.
[60] Imai, K., Sumino, S., and Imai, H., Minimax Geometric Fitting of Two Corresponding Sets of Points, *Proc. 5th Annual ACM Symp. Comput. Geometry,* 266–275, 1989.
[61] Janardan, R. and Lopez, M., Generalized Intersection Searching Problems, *Intl. J. Comput. Geom. & Appl.,* 3(1), 39–69, Mar. 1993.
[62] Kapoor, S. and Smid, M., New Techniques for Exact and Approximate Dynamic Closest-Point Problems, *SIAM J. Computing,* 25(4), 775–796, Aug. 1996.
[63] Klawe, M.M. and Kleitman, D.J., An Almost Linear Time Algorithm for Generalized Matrix Searching, *SIAM J. Discrete Math.,* 3(1), 81–97, Feb. 1990.
[64] Lee, D.T., Two Dimensional Voronoi Diagrams in the L_p-metric, *J. ACM,* 27, 604–618, 1980.
[65] Lee, D.T., Computational Geometry, *Computer Science and Engineering Handbook,* Tucker, A., Ed., CRC Press, Boca Raton, FL, 111–140, 1996.
[66] Lee, D.T., Geometric Algorithm Visualization, Current Status and Future, *Applied Computational Geometry,* Lin and Manocha, Eds., Springer-Verlag, 1996, 45–50.
[67] Lee, D.T. and Preparata, F.P., Computational Geometry: A Survey, *IEEE Trans. Comput.,* C-33(12), 1072–1101, Dec. 1984.
[68] Lee, D.T. and Shen, C.F., The Steiner Minimal Tree Problem in the λ-*geometry* Plane, *Proc. 7th Intl. Symposium on Algorithms and Computation,* Osaka, Japan, 247–255, Dec. 1996.
[69] Lee, D.T. and Wu, V.B., Multiplicative Weighted Farthest Neighbor Voronoi Diagrams in the Plane, *Proc. Intl. Workshop on Discrete Mathematics and Algorithms,* Hong Kong, 154–168, Dec. 1993.
[70] Lee, D.T., Yang, C.D., and Wong, C.K., Rectilinear Paths among Rectilinear Obstacles, *Perspectives in Discrete Applied Math.,* Bogart, K., Ed., 70, 185–215, 1996.
[71] Lengauer, T., *Combinatorial Algorithms for Integrated Circuit Layout,* John Wiley & Sons, 1990.
[72] Liotta, G., Preparata, F.P., and Tamassia, R., Robust Proximity Queries: an Illustration of Degree-driven Algorithm Design, *Proc. 13th Annual ACM Symp. Comput. Geometry,* 156–165, 1997.
[73] Marcotte, O. and Suri, S., Fast Matching Algorithms for Points on a Polygon, *SIAM J. Comput.,* 20(3), 405–422, Jun. 1991.
[74] Matoušek, J., Geometric Range Searching, *ACM Computing Survey,* 26(4), 421–461, 1994.

[75] Mitchell, J.S.B., Shortest Paths among Obstacles in the Plane, *Intl. J. Computational Geometry & Applications,* 6(3), 309–332, Sep. 1996.
[76] Mitchell, J.S.B. and Papadimitriou, C.H., The Weighted Region Problem: Finding Shortest Paths through a Weighted Planar Subdivision, *J. ACM,* 38(1), 18–73, Jan. 1991,
[77] Mitchell, J.S.B., Piatko, C., and Arkin, E.M., Computing a Shortest k-Link Path in a Polygon, *Proc. 33rd Annual Symp. Foundations of Comput. Sci.,* 573–582, Oct. 1992.
[78] Mitchell, J.S.B., Rote, G., and Wōginger, G., Minimum Link Path among Obstacles in the Planes, *Algorithmica,* 8, 431–459, 1992.
[79] Mount, D.M., Intersection Detection and Separators for Simple Polygons, *Proc. 8th Annual ACM Symp. Comput. Geometry,* 303–311, 1992.
[80] Näher, S., LEDA – A Library of Efficient Data Types and Algorithms, Max-Planck-institut für informatik, `http://www.mpi-sb.mpg.de/LEDA/leda.html`.
[81] Okabe, A., Boots, B., and Sugihara, K., *Spatial Tessellations: Concepts and Applications of Voronoi Diagrams,* John Wiley & Sons, Chichester, England, 1992.
[82] Papadimitriou, C.H. and Steiglitz, K., *Combinatorial Optimization: Algorithms and Complexity,* Prentice-Hall, Englewood Cliffs, NJ, 1982.
[83] Papadopoulou, E., k-Pairs Non-Crossing Shortest Paths in a Simple Polygon, *Proc. 7th Intl. Symp. Algorithms and Computation,* 305–314, Dec. 1996.
[84] Papadopoulou, E. and Lee, D.T., Efficient Computation of the Geodesic Voronoi Diagram of Points in a Simple Polygon, *Algorithmica,* (to appear).
[85] Preparata, F.P. and Shamos, M.I., *Computational Geometry: An Introduction,* Springer-Verlag, 1988.
[86] Ravi, R., Sundaram, R., Marathe, M.V., Rosenkrantz, D.J., and Ravi, S.S., Spanning Trees Short or Small, *SIAM J. Discrete Math.,* 9(2), 178–200, May 1996.
[87] Rosenberger, H., Order k Voronoi Diagrams of Sites with Additive Weights in the Plane, *Algorithmica,* 6, 153–181, 1991.
[88] Sack, J. and Urrutia, J., *Handbook of Computational Geometry,* Elsevier, Amsterdam, 1997.
[89] Schaudt, B.F. and Drysdale, R.L., Multiplicatively Weighted Crystal Growth Voronoi Diagrams, *Proc. 7th Annual ACM Symp. Comput. Geometry,* 214–223, 1991.
[90] Schaudt, B.F. and Drysdale, R.L., Higher-Dimensional Voronoi Diagrams for Convex Distance Functions, *Proc. 4th Canad. Conf. Comput. Geometry,* 274–279, 1992.
[91] Schwartz, C., Smid, M., and Snoeyink, J., An Optimal Algorithm for the On-Line Closest-Pair Problem, *Algorithmica,* 12(1), 18–29, Jul. 1994.
[92] Shermer, T. and Yap, C., Probing Near Centers and Estimating Relative Roundness, *Proc. ASME Workshop on Tolerancing and Metrology,* 1995.
[93] Suri, S., On Some Link Distance Problems in a Simple Polygon, *IEEE Trans. Robotics and Automation,* 6(1), 108–113, 1990.
[94] Swanson, K., Lee, D.T., and Wu, V.L., An Optimal Algorithm for Roundness Determination on Convex Polygons, *Comput. Geometry: Theory & Appl.,* 5(4), 225–235, Nov. 1995.
[95] Vaidya, P.M., Space-Time Tradeoffs for Orthogonal Range Queries, *SIAM J. Computing,* 18(4), 748–758, Aug. 1989.
[96] Vaidya, P.M., Geometry Helps in Matching, *SIAM J. Computing,* 18(6), 1201–1225, Dec. 1989.
[97] Welzl, E., Smallest Enclosing Disks, Balls and Ellipsoids, in it New Results and New Trends in Computer Science, Maurer, H., Ed., LNCS, Vol. 555, Springer-Verlag, 359–370, 1991.
[98] Yang, C.D., Lee, D.T., and Wong, C.K., Rectilinear Path Problems Among Rectilinear Obstacles Revisited, *SIAM J. Computing,* 24(3), 457–472, Jun. 1995.
[99] Yao, F.F., Computational Geometry, in *Handbook of Theoretical Computer Science,* Vol. A, Algorithms and Complexity, van Leeuwen, J., Ed., 343–389, 1994.
[100] Yap, C., Exact Computational Geometry and Tolerancing, in *Snapshots of Computational and Discrete Geometry,* Avis and Bose, Eds., School of Comput. Sci., McGill University, 1995.

Further Information

Additional references about various variations of closest pair problems can be found in [18, 21, 62, 91]. For additional results concerning the Voronoi diagrams in higher dimensions and the duality transformation see [15]. For more information about Voronoi diagrams for sites other than points, in various distance functions or norms, see [9, 12, 20, 81, 87, 88, 90]. A recent textbook by de Berg et al. [38] contains a very nice treatment of computational geometry in general. More information can be found in [54, 65, 88, 99]. The reader who is interested in parallel computational geometry is referred to [13]. For current research activities and results, the reader may consult the Proceedings of the Annual ACM symposium on Computational Geometry, and the following three journals: *Discrete & Computational Geometry, International Journal of Computational Geometry & Applications,* and *Computational Geometry: Theory and Applications.* The **ftp** site `/pub/geometry/geombib.tar.gz` at `ftp.cs.usask.ca` contains close to 10,000 entries of bibliography in this field.

Those who are interested in the implementations or would like to have more information about available software can consult `http://www.geom.umn.edu/software/cglist/`.

The following WWW page on *Geometry in Action* maintained by David Eppstein at `http://www.ics.uci.edu/~eppstein/geom.html` and the computational geometry page by J. Erickson at `http://www.cs.duke.edu/~jeffe/compgeom` give a comprehensive description of research activities of computational geometry.

21

Robot Algorithms

Dan Halperin
Tel-Aviv University

Lydia Kavraki
Stanford University

Jean-Claude Latombe
Stanford University

21.1 Introduction

Robots are versatile mechanical devices equipped with actuators and sensors under the control of a computing system. They perform tasks by executing motions in the physical space. This space is populated by various objects and is subject to the laws of nature. A typical task consists of achieving a goal spatial arrangement of objects from a given initial arrangement, for example, assembling a product. Robots are programmable, which means that they can perform a variety of tasks by simply changing the software commanding them. This software embeds robot algorithms, which are abstract descriptions of processes consisting of motions and sensing operations in the physical space.

Robot algorithms differ in significant ways from traditional computer algorithms. The latter have full control over, and perfect access to the data they use, letting aside, for example, problems related to floating-point arithmetic. In contrast, robot algorithms eventually apply to physical objects in the real world, which they attempt to control despite the fact that these objects are subject to the independent and imperfectly modeled laws of nature. Data acquisition through sensing is also local and noisy. Robot algorithms hence raise controllability (or reachability) and observability (or recognizability) issues that are classical in control theory, but not present in computer algorithms.

On the other hand, control theory often deals with well-defined processes in strictly confined environments. In contrast, robot tasks tend to be underspecified, which requires addressing combinatorial issues ignored in control theory. For instance, to reach a goal among **obstacles** that are not represented in the input model, but are sensed during execution, a robot must search "on the fly" for a collision-free **path,** a notoriously hard computational problem.

This blend of control and computational issues is perhaps the main characteristic of robot algorithms. It is presented at greater length in Section 21.2, along with other features of these algorithms. Section 21.3 then surveys specific areas of robotics (e.g., part manipulation, assembly sequencing, motion planning, sensing) and presents algorithmic techniques that have been developed in those areas.

21.2 Underlying Principles

Robot Algorithms Control

The primary goal of a robot algorithm is to describe a procedure for controlling a subset of the real world—the **workspace**—in order to achieve a given goal, say, a spatial arrangement of several physical objects. The real world, which is subject to the laws of nature (such as gravity, inertia, friction), can be regarded as performing its own actions, for instance, applying forces. These actions are not arbitrary, and to some extent, they can be modeled, predicted, and controlled. Thus, a robot algorithm should specify robot's operations whose combination with the (re-)actions of the real world will result in achieving the goal. Note that the robot is itself an important object in the workspace; for example, it should not collide with obstacles. Therefore, the algorithm should also control the relation between the robot and the workspace. The robot's internal controller, which drives the actuators and preprocesses sensory data, defines the primitive operations that can be used to build robot algorithms.

The design of a robot algorithm requires identifying a set of relevant states of the workspace (one being the goal) and selecting operations to take the workspace through a sequence of states ending at the goal. But, due to various inaccuracies (one is in modeling physical laws), an operation may transform a state into one among several possible states. The algorithm can then use sensing to refine its knowledge during execution. In turn, because workspace sensing is imperfect, a state may not be directly recognizable, meaning that no combination of sensors may be capable to return the state's identity. As a result, the three subproblems—choosing pertinent states, selecting operations to transit among these states toward the goal, and constructing state-recognition functions—are strongly interdependent and cannot be solved sequentially.

To illustrate part of the above discussion, consider the task of orienting a convex polygonal part P on a table using a robot arm equipped with a parallel-jaw gripper. This is a typical problem in industrial part feeding ("Part Feeding"). If an overhead vision system is available to measure P's orientation, we can use the following (simplified) algorithm:

ORIENT(P, θ_g)
1 Measure P's initial orientation θ_i
2 Move the gripper to the grasp position of P
3 Close the gripper
4 Rotate the gripper by $\theta_g - \theta_i$
5 Open the gripper
6 Move the gripper to a resting position

The states of interest are defined by the orientations of P, the position of the gripper relative to P, and whether the gripper holds P or does not. (Only the initial and goal orientations, θ_i and θ_g, are explicitly considered.) Step 1 acquires the initial state. Step 4 achieves the goal state. Steps 2 and 3 produce intermediate states. Steps 5 and 6 achieve a second goal not mentioned above, that the robot be away from P at the end of the orientation operation.

A very different algorithm for this same part-orienting task consists of squeezing P several times between the gripper's jaws, at appropriately selected orientations of the gripper (see Fig. 21.1). This algorithm, which requires no workspace sensing, is based on the following principle. Let P be at an arbitrary initial orientation. Any squeezing operation will achieve a new orientation that belongs to a set of $2n$ (n being

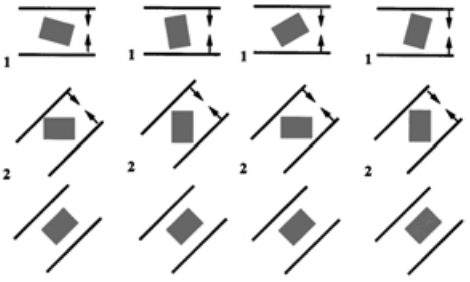


FIGURE 21.1 Orienting a convex polygonal part [20].

the number of sides of P) possible orientations determined by the geometry of P and the orientation of the jaws. If P is released and squeezed again with another orientation of the gripper, the set of possible orientations of P can be reduced further. For any n-sided convex polygon P, there is a sequence of $2n - 1$ squeezes that achieves a single orientation of P (up to symmetries), for an arbitrary initial orientation of P [20].

The states considered by the second algorithm are individual orientations of P and sets of orientations. The state achieved by each squeeze is determined by the jaws' orientation and the previous state. Its prediction is based on understanding the simple mechanics of the operation. The fact that any convex polygon admits a finite sequence of squeezes ending at a unique orientation guarantees that any goal is reachable from any state. However, when the number of parameters that the robot can directly control is smaller than the number of parameters defining a state, the question of whether the goal state is reachable is more problematic (see "Path Planning for Nonholonomic Robots").

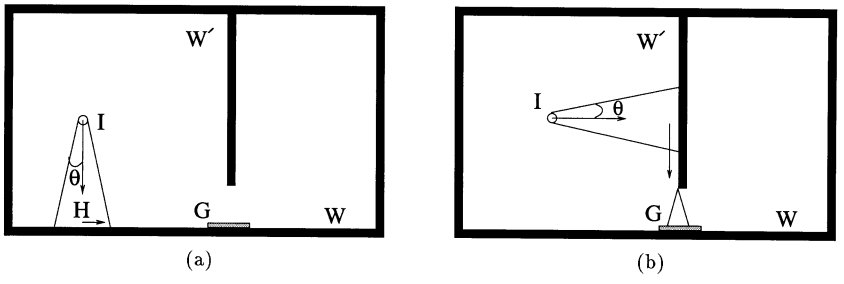


FIGURE 21.2 Goal recognition in mobile robot navigation.

State recognition can also be difficult. To illustrate, consider a mobile robot navigating in an office environment. Its controller uses dead-reckoning techniques to control the motion of the wheels. But these techniques yield cumulative errors in the robot's position with respect to a fixed coordinate system. For better localization, the robot algorithm may sense environmental features (e.g., a wall, a door). However, because sensing is imperfect, a feature may be confused with a similar feature at a different place; this may occasionally cause a major localization mistake. Thus, the robot algorithm must be designed so that enough environmental features will be sensed to make each successive state reliably recognizable.

To be more specific, consider the workspace of Fig. 21.2. Obstacles are shown in bold lines. The robot is modeled as a point with perfect touch sensing. It can be commanded to move along any direction $\phi \in [0, 2\pi)$ in the plane, but imperfect control makes it move within a cone $\phi \pm \theta$, where the angle θ models directional uncertainty. The robot's goal is to move into G, the goal state, which is a subset of the wall W (for instance, G is an outlet for recharging batteries). The robot's initial location is not precisely known: it is anywhere in the disk I, the initial state. One candidate algorithm (illustrated in

Fig. 21.2a) first commands the robot to move perpendicularly to W until it touches it. Despite directional uncertainty, the robot is guaranteed to eventually touch W, somewhere in the region denoted by H. From state H, it can slide along the wall (using touch to maintain contact) toward G. The robot is guaranteed to eventually reach G. But can it reliably recognize this achievement? The answer depends on the growth of the dead-reckoning localization error as the robot moves along W. Clearly, if this error grows by more than half the difference in the size of G and H, G is not reliably recognizable.

An alternative algorithm is possible, using the wall W' (Fig. 21.2b). It commands the robot to first move toward W' until it touches it and then slide along it toward W. At the end of W', it continues heading toward W, but with directional uncertainty. The robot is nevertheless guaranteed to be in G when it senses that it has touched W.

Robot Algorithms Plan

Consider the following variant of the part-orienting task. Parts are successively fed with arbitrary orientations on a table by an independent machine. They have different and arbitrary convex polygonal shape, but whenever a part arrives, the feeding machine provides a geometric model of the part to the robot, along with its goal orientation. In the absence of a vision sensor, the multi-squeeze approach can still be used, but now the robot algorithm must include a planner to compute automatically the successive orientations of the gripper.

As another example, consider the pick-and-place task which requires a robot arm to transfer an object from one position to another. If the obstacles in the workspace are not known in advance, the robot algorithm needs sensing to localize them, as well as a planner to generate a collision-free path. If all obstacles cannot be sensed at once, the algorithm may have to interweave sensing, planning, and acting.

The point is that, for most tasks, the set of states that may have to be considered at execution time is too large to be explicitly anticipated. The robot algorithm must incorporate a planner. In first approximation, the planner can be seen as a separate algorithm that automatically generates a control algorithm for achieving a given task. The robot algorithm is the combination of the planning and the control algorithms. More generally, however, it is not sufficient to invoke the planner once, and planning and control are interleaved. The effect of planning is to dynamically change the control portion of the robot algorithm, by changing the set of states of the workspace that are explicitly considered.

Planning, which often requires exploring large search spaces, raises critical complexity issues. For example, finding a collision-free path for a three-dimensional **linkage** among polyhedral obstacles is PSPACE-hard [47], and the proof of this result provides strong evidence that any complete algorithm will require exponential time in the **number of degrees of freedom**. Planning the motion of a point robot among polyhedral obstacles, with bounded uncertainty in control and sensing, is NEXPTIME-hard [10].

The computational complexity of planning leads to looking for efficient solutions to restricted problems. For example, for part orienting, there exists a complete planning algorithm that computes a sequence of squeezes achieving a single orientation (up to symmetries) of a given convex polygonal part in quadratic time in the number of sides of the part [20]. Another way of dealing with complexity is to trade off completeness against time, by accepting weaker variants of completeness. A **complete planner** is guaranteed to find a solution whenever one exists, and to notify that there exists none otherwise. A weaker variant is probabilistic completeness: if there exists a solution, the planner will find one only with high probability. This variant can be very useful if one can show that the probability of not finding a solution (when one exists) tends rapidly toward 0 as the running time increases. In Section 21.3 we will present planning algorithms that embed similar approaches.

The complexity of a robot algorithm has also some interesting relations with the reachability and recognizability issues introduced in the previous subsection. We will mention several such relations in Section 21.3 (in particular, in "Path Planning for Nonholonomic Robots" and in "Motion Planning with Uncertainty").

The potentially high cost of planning and the fact that it may have to be done on-line raise an additional issue. A robot algorithm must carefully allocate time between computations aimed at planning and computations aimed at controlling and sensing the workspace. If the workspace is changing in an unpredictable way (say, under the influence of other agents), spending too much time on planning may result in obsolete control algorithms; on the other hand, not enough planning may yield irreversible failures. The problem of allocating time between planning and control remains poorly understood, though several promising ideas have been proposed. For example, it has been suggested to develop planners that return a plan in whatever amount of time is allocated to them and can be called back later to incrementally improve the previous plan if more time is allocated to planning [7]. Deliberative techniques have been proposed to decide what amount of time should be given to planning and control and update this decision as more information is collected [43].

Robot Algorithms Reason About Geometry

Imagine a robot whose task is to maintain a botanic garden. To set and update its goal agenda, this robot needs knowledge in domains like botany and fertilization. The algorithms using this knowledge can barely be considered parts of a robot algorithm. But, on the other hand, all robots, including gardener robots, accomplish tasks by eventually moving objects (including themselves) in the real world. Hence, at some point, all robots must reason about the geometry of their workspace. Actually, geometry is not enough, since objects have mass inducing gravitational and inertial forces, while contacts between objects generate frictional forces. All robots must therefore reason with classical mechanics. However, Newtonian concepts of mechanics translate into geometric constructs (e.g., forces are represented as vectors), so that most of the reasoning of a robot eventually involves dealing with geometry.

Computing with continuous geometric models raises discretization issues. For example, a typical planner computing a robot's path first discretizes the robot's free space (the set of collision-free **configurations** of the robot) in order to build a connectivity graph to which well-known search algorithms can be applied. One discretization approach is to place a fine regular grid across **configuration space** and search that grid for a sequence of adjacent points in free space. The grid is just a computational tool and has no physical meaning. Its resolution is arbitrarily chosen despite its critical role in the computation: if it is too coarse, planning is likely to fail; if it is too fine, planning will take too much time. Instead, criticality-driven discretizations have been proposed, whose underlying principle is widely applicable. They consist of partitioning the continuous space of interest into cells, such that some pertinent property remains invariant over each cell and changes when the boundary separating two cells is crossed. The second part-orienting algorithm in "Robot Algorithms Control" is based on such a discretization. The set of all possible orientations of the part is represented as the unit circle (the cyclic interval $[0, 2\pi)$). For a given orientation of the gripper, this circle can be partitioned into arcs such that, for all initial orientations of the part in the same arc, the part's final orientation will be the same after the gripper has closed its jaws. The final orientation is the invariant associated with the cell. From this decomposition, it is a relatively simple matter to plan a squeezing sequence.

Several such criticality-driven discretizations have been proposed for path planning, assembly sequence planning, motion planning with uncertainty, robot localization, object recognition, and so on, as will be described in Section 21.3. Several of them use ideas and tools originally developed in computational geometry, for instance; plane sweep, topological sweep, computing arrangements.

Robot algorithms often require dealing with high-dimensional geometric spaces. Although criticality-based discretization methods apply to such spaces in theory (for instance, see [49]), their computational complexity is then overwhelming. This has led the development of randomized techniques that efficiently approximate the topology and geometry of such spaces by random discretization. Such techniques have been particularly successful for building probabilistically complete path planners ("Probabilistic Algorithms").

Robot Algorithms Have Physical Complexity

Just as the complexity of a computation characterizes the amount of time and memory this computation requires, we can define the physical complexity of a robot algorithm by the amount of physical resources it takes, e.g., the number of "hands," the number of motions, or the number of beacons. Some resources, like the number of motions, relate to the time spent executing the algorithm. Others, like the number of beacons, relate to the engineering cost induced by the algorithm. For example, one complexity measure of the multi-squeeze algorithm to orient a convex polygon ("Robot Algorithms Control") is the maximal number of squeeze operations this algorithm performs. As another example, consider an assembly operation merging several parts into a subassembly. The number of subsets of parts moving relative to each other (each subset moving as a single rigid body) measures the number of hands necessary to hold the parts during the operation. The number of hands required by an assembly sequence is the maximal number of hands needed by an operation, over all the operations in the sequence ("Assembly Sequencing"). The number of fingers to safely grasp or fixture an object is another complexity measure ("Grasping").

Though there is a strong conceptual analogy between computational and physical complexities, there are also major differences between the two notions. Physical complexity must be measured along many more dimensions than computational complexity. Moreover, while computational complexity typically measures an asymptotic trend, a tighter evaluation is usually needed for physical complexity since robot tasks involve relatively few objects.

One may also consider the inherent physical complexity of a task, a notion analogous to the inherent complexity of a computational problem. For example, to orient a convex polygon with n sides, $2n - 1$ squeezes may be needed in the worst case; no correct algorithm can perform better in all cases. By generating all feasible assembly sequences of a product, one could determine the number of hands needed by each sequence and return the smallest number. This number is a measure of the inherent complexity of assembling the product. No robot algorithm to assemble this product can require fewer hands.

Evaluating the inherent physical complexity of a task may lead to redefining the task, if it turns out to be too complex. For example, it has been shown that a product made of n parts may need up to n hands for its assembly ("Grasping"), thus requiring the delicate coordination of $n - 1$ motions. Perhaps a product whose assembly requires several hands could be redesigned so that two hands are sufficient, as is the case for most industrial products. Actually, designers strive to reduce physical complexity along various dimensions. For instance, many mass-produced devices are designed to be assembled with translations only, along very few directions (possibly a single one). The inherent physical complexity of a robot task is not a recent concept, but its formal application to task analysis is [54].

An interesting issue is how the computational and physical complexities of a robot algorithm relate to each other. For example, planning for mobile robot navigation with uncertainty is a provably hard computational problem ("Motion Planning with Uncertainty"). On the other hand, burying wires in the ground or placing enough infrared beacons allows robots to navigate reliably at small computational cost. But isn't it too much? Perhaps the intractability of motion planning with uncertainty can be eliminated with less costly engineering.

21.3 State of the Art and Best Practices

Robotics is a broad domain of research. In this subsection we study a number of specific areas: part manipulation, assembly sequencing, motion planning, and (briefly) sensing. For each area, we introduce problems and survey key algorithmic results.

Although we present the current research according to problem domain, there are several techniques that cross over many domains. One of the most frequently applied methods in robotics is the criticality-based discretization mentioned in "Robot Algorithms Reason About Geometry." This technique allows us to discretize a continuous space without giving up the completeness or exactness of the solution. It is closely related to the study of arrangements in computational geometry [23]. When criticality-based

discretization is done in a space representing all possible motions, it yields the so-called "nondirectional" data structures, which is another prevailing concept in robot algorithms and is exemplified in detail in "Monotone Two-Handed Assembly Sequency."

Randomization is another important paradigm in robotics. Randomized techniques have made it possible to cope practically with robot motion planning with many degrees of freedom (Section on "Probabilistic Algorithms"). Also, randomized algorithms are often simpler than their deterministic counterparts and hence better candidates for efficient implementation. Randomization has recently been applied to solving problems in grasping as well as in many other areas that involve geometric reasoning.

Throughout this section we interchangeably use the terms "body," "physical object," and "part" to designate a rigid physical object modeled as a compact manifold with boundary $B \subset \mathbb{R}^k$ ($k = 2$ or 3). B's boundary is also assumed piecewise-smooth.

Part Manipulation

Part manipulation is one of the most frequently performed operations in industrial robotics: parts are grasped from conveyor belts, they are oriented prior to feeding assembly workcells, and they are immobilized for machining operations.

Grasping

Part grasping has motivated various kinds of research, including the design of versatile mechanical hands, as well as simple, low-cost grippers. From an algorithmic point of view, the main goal is to compute "safe" grasps for an object whose model is given as input.

Force-Closure Grasp Informally, a grasp specifies the positions of "fingers" on a body B. A more formal definition uses the notion of a wrench, a pair $[\boldsymbol{f}, \boldsymbol{p} \times \boldsymbol{f}]$, where $\boldsymbol{p}$ denotes a point in the boundary ∂B of B, represented by its coordinate vector in a frame attached to B, $\boldsymbol{f}$ designates a force applied to B at $\boldsymbol{p}$, and $\times$ is the vector cross-product. If $\boldsymbol{f}$ is a unit vector, the wrench is said to be a unit wrench. A finger is any tool that can apply a wrench.

A grasp of B is a set of unit wrenches $\boldsymbol{w}_i = [\boldsymbol{f}_i, \boldsymbol{p}_i \times \boldsymbol{f}_i]$, $i = 1, \ldots, p$, defined on B. For each $\boldsymbol{w}_i$, if the contact is frictionless, $\boldsymbol{f}_i$ is normal to ∂B at $\boldsymbol{p}_i$; otherwise, it can span a friction cone (Coulomb law of friction).

The notion of a safe grasp is captured by force closure. A force-closure grasp $\{\boldsymbol{w}_i\}_{i=1,\ldots,p}$ on B is such that, for any arbitrary wrench $\boldsymbol{w}$, there exists a set of real values $\{f_1, \ldots, f_p\}$ achieving $\Sigma_{i=1}^{p} f_i \boldsymbol{w}_i = -\boldsymbol{w}$. In other words, a force-closure grasp can resist any external wrench applied to B. If contacts are nonsticky, we require that $f_i \geq 0$, for all $i = 1, \ldots, p$, and the grasp is called positive. Here, we only consider positive grasps. A form-closure grasp is a positive force-closure grasp when all finger-body contacts are frictionless.

Size of a Form/Force-Closure Grasp The following results characterize the physical complexity of achieving a safe grasp [40]:

- Bodies with rotational symmetry (e.g., discs in 2-space, spheres and cylinders in 3-space) admit no form-closure grasps.
- All other bodies admit a form-closure grasp with at most 4 fingers in 2-space and 12 fingers in 3-space.
- All polyhedral bodies have a form-closure grasp with 7 fingers.
- With frictional finger-body contacts, all bodies admit a force-closure grasp that consists of 3 fingers in 2-space and 4 fingers in 3-space.

Testing Force Closure A necessary and sufficient condition for a grasp $\{\boldsymbol{w}_i\}_{i=1,\ldots,p}$ to achieve force closure in 2-space (respectively, 3-space) is that the finger wrenches $\boldsymbol{w}_i$ span a space F of dimension

3 (respectively, 6) and that a strictly positive linear combination of them be zero. In other words, the origin of F (null wrench) should lie in the interior of the convex hull of the finger wrenches [40]. This condition provides an effective test for deciding in constant time whether a given grasp achieves force closure.

Computing Form/Force Closure Grasps Most research has concentrated on computing grasps with 2 to 4 nonsticky fingers. Algorithms that compute a single force-closure grasp of a polygonal/polyhedral part in time linear in the part's complexity have been derived in [40].

Finding the maximal regions on a body where fingers can be positioned independently while achieving force closure makes it possible to accommodate errors in finger placement. Geometric algorithms for constructing such regions are proposed in [42] for grasping polygons with two fingers (with friction) and four fingers (without friction), and for grasping polyhedra with three fingers (with frictional contact capable of generating torques) and seven fingers (without friction). Grasping of curved obstacles is addressed in [45].

Fixturing

Most manufacturing operations require fixtures to hold parts. To avoid the custom design of fixtures for each part, modular reconfigurable fixtures are often used. A typical modular fixture consists of a workholding surface, usually a plane, that has a lattice of holes where locators, clamps, and edge fixtures can be placed. Locators are simple round pins, while clamps apply pressure on the part.

Contacts between fixture elements and parts are generally assumed frictionless. In modular fixturing, contact locations are restricted by the lattice of holes, and form closure cannot always be achieved. In particular, when three locators and one clamp are used on a workholding plane, there exist polygons of arbitrary size for which no form-closure fixture exists; but, if parts are restricted to be rectilinear with all edges longer than four lattice units, a form-closure fixture always exists [56].

When the fixturing kit consists of a latticed workholding plane, three locators, and one clamp, it is possible to find all possible placements of a given part on the workholding surface where form closure can be achieved, along with the corresponding positions of the locators and the clamp [9].

Part Feeding

Part feeders account for a large fraction of the cost of a robotic assembly workcell. A typical feeder must bring parts at subsecond rates with high reliability. An example of a flexible feeder is given in [20] and described in "Robot Algorithms Control" above.

Part feeding often relies on nonprehensile manipulation, which exploits task mechanics to achieve a goal state without grasping and frequently allows accomplishing complex feeding tasks with simple mechanisms [2]. Pushing is one form of nonprehensile manipulation. Work on pushing originated in [38] where a simple rule is established to qualitatively determine the motion of a pushed object. This rule makes use of the position of the center of friction of the object on the supporting surface. Related results include a planning algorithm for a robot that tilts a tray with a planar part of known shape to orient it to a desired orientation [16] and an algorithm that computes the sequence of motions of a single articulated fence on a conveyor belt to achieve a goal orientation of an object [2].

Assembly Sequencing

Most mechanical products consist of multiple parts. The goal of assembly sequencing is to compute both an order in which parts can be assembled and the corresponding required movements of the parts. Assembly sequencing can be used during design to verify that the product will be easy to manufacture and service. An assembly sequence is also a robot algorithm at a high level of abstraction since parts are assumed free-flying, massless geometric objects.

Notion of an Assembly Sequence

An assembly A is a collection of bodies in some given relative placements. Subassemblies are separated if they are arbitrarily far apart from one another. An assembly operation is a motion that merges s separated subassemblies ($s \geq 2$) into a new subassembly, with each subassembly moving as a single body. No overlapping between bodies is allowed during the operation. The parameter s is called the number of hands of the operation. (Hence, a hand is seen here as a grasping or fixturing tool that can hold an arbitrary number of bodies in fixed relative placements.) Assembly partitioning is the reverse of an assembly operation.

An assembly sequence is a total ordering on assembly operations that merges the separated parts composing an assembly into this assembly. The maximum, over all the operations in the sequence, of the number of hands of an operation is the number of hands of the sequence.

A monotone assembly sequence contains no operation that brings a body to an intermediate placement (relative to other bodies), before another operation transfers it to its final placement. Therefore, the bodies in every subassembly produced by such a sequence are in the same relative placements as in the complete assembly. Note that a product may admit no monotone assembly sequence for a given number of hands, while it may admit such sequences if more hands are allowed.

Number of Hands in Assembly

The number of hands needed for various families of assemblies is a measure of the inherent physical complexity of an assembly task ("Robot Algorithms Have Physical Complexity"). It has been shown that an assembly of convex polygons in the plane has a two-handed assembly sequence of translations. In the worst case, s hands are necessary and sufficient for assemblies of s star-shaped polygons/polyhedra [41].

There exists an assembly of six tetrahedra without a two-handed assembly sequence of translations, but with a three-handed sequence of translations. Every assembly of five or fewer convex polyhedra admits a two-handed assembly sequence of translations. There exists an assembly of thirty convex polyhedra that cannot be assembled with two hands [51].

Complexity of Assembly Sequencing

When arbitrary sequences are allowed, assembly sequencing is PSPACE-hard. The problem remains PSPACE-hard even when the bodies are polygons, each with a constant maximal number of vertices [41]. When only two-handed monotone sequences are permitted and rigid motions are allowed, finding a partition of an assembly A into two subassemblies S and $A \setminus S$ is NP-complete. The problem remains NP-complete when both S and $A \setminus S$ are connected and motions are restricted to translations [26]. These latter results were obtained by reducing in polynomial time any instance of the 3-SAT problem to a mechanical assembly such that the partitioning of this assembly gives the solution of the 3-SAT problem instance.

Monotone Two-Handed Assembly Sequencing

A popular approach to assembly sequencing is disassembly sequencing. A sequence that separates an assembly to its individual components is first generated and next reversed. Most existing assembly sequencers can only generate two-handed monotone sequences. Such a sequence is computed by partitioning the assembly and, recursively, the obtained subassemblies into two separated assemblies.

The nondirectional blocking graph (NDBG, for short) is proposed in [54] to represent all the blocking relations in an assembly. It is a subdivision of the space of all allowable motions of separation into a finite number of cells such that within each cell the set of blocking relations between all pairs of parts remain fixed. Within each cell this set is represented in the form of a directed graph, called the directional blocking graph (DBG). The NDBG is the collection of the DBGs over all the cells in the subdivision. The NDBG is one example of a data structure obtained by a criticality-driven discretization technique ("Robot Algorithms Reason About Geometry").

We illustrate this approach for polyhedral assemblies when the allowable motions are infinite translations. The partitioning of an assembly consisting of polyhedral parts into two subassemblies is done as follows. For an ordered pair of parts P_i, P_j, the 3-vector $\boldsymbol{d}$ is a blocking direction if translating P_i to infinity in direction $\boldsymbol{d}$ will cause P_i to collide with P_j. For each ordered pair of parts the set of blocking directions is constructed on the unit sphere $\mathcal{S}^2$ by drawing the boundary arcs of the union of the blocking directions (each arc is a portion of a great circle). The resulting collection of arcs partitions $\mathcal{S}^2$ into maximal regions such that the blocking relation among the parts is the same for any direction inside such a region.

Next, the blocking graph is computed for one such maximal region. The algorithm then moves to an adjacent region and updates the DBG by the blocking relations that change at the boundary between the regions, and so on. After each time the construction of a DBG is completed, this graph is checked for strong connectivity in time linear in the number its edges. The algorithm stops the first time it encounters a DBG that is not strongly connected and it outputs the two subassemblies of the partitioning. The overall sequencing algorithm continues recursively with the resulting subassemblies. If all the DBGs that are produced during a partitioning step are strongly connected, the algorithm notifies that the assembly does not admit a two-handed monotone assembly sequence with infinite translations.

Polynomial time algorithms are proposed in [54] to compute and exploit NDBGs for restricted families of motions. In particular, the case of partitioning a polyhedral assembly by a single translation to infinity, is analyzed in detail, and it is shown that partitioning an assembly of m polyhedra with a total of v vertices takes $O(m^2v^4)$ time. Another case is where the separating motions are infinitesimal rigid motions. Then partitioning the polyhedral assembly takes $O(mc^5)$ time, where m is the number of pairs of parts in contact and c is the number of independent point-plane contact constraints. With the above algorithms, every feasible disassembly sequence can be generated in polynomial time.

Basic Path Planning

Motion planning is aimed at providing robots with the capability of deciding automatically which motions to execute. It arises in a variety of forms. The simplest—basic path planning—requires finding a geometric collision-free path for a single robot in a known static workspace. The path is represented by a curve segment connecting two points in the robot's configuration space [35]. This curve must not intersect a forbidden region, the C-obstacle region, which is the image of the workspace obstacles. Other motion planning problems require dealing with moving obstacles, multiple robots, movable objects, uncertainty, etc.

In this subsection we consider basic path planning. In the next three subsections we will review other motion planning problems and issues.

Configuration Space

A configuration of a robot $\mathcal{A}$ is any mathematical specification of the position and orientation of every body composing $\mathcal{A}$, relative to a fixed coordinate system. The configuration of a single body is also called a placement or a pose.

The robot's configuration space is the set of all its configurations. Usually, it is a smooth manifold. We will always denote the configuration space of a robot by $\mathcal{C}$ and its dimension by m. Given a robot $\mathcal{A}$, we will let $\mathcal{A}(\boldsymbol{q})$ denote the subset of the workspace occupied by $\mathcal{A}$ at configuration $\boldsymbol{q}$.

The number of degrees of freedom of a robot is the dimension m of its configuration space. We abbreviate "degree of freedom" by dof.

Given an obstacle B_i in the workspace, the subset $CB_i \subseteq \mathcal{C}$ such that, for any $\boldsymbol{q} \in CB_i$, $\mathcal{A}(\boldsymbol{q})$ intersects B_i is called a **C-obstacle**. The union $CB = \cup_i CB_i$ plus the configurations that violate the mechanical limits of the robot's joints is called the **C-obstacle region**. The **free space** is the complement of the C-obstacle region in $\mathcal{C}$, that is, $\mathcal{C} \backslash CB$. In most practical cases, C-obstacles are represented as semialgebraic sets with piecewise smooth boundaries.

A robot's path is a continuous map $\tau : [0, 1] \rightarrow \mathcal{C}$. A **free path** is a path that entirely lies in free space. A **semifree path** lies in the closure of free space.

The **basic path planning problem** is to compute a free or semifree path between two input configurations. A **complete motion planner** is guaranteed to find a (semi)free path between two given configurations whenever such a path exists, and to notify that no such path exists otherwise.

Complete Algorithms

Basic path planning for a three-dimensional linkage made of polyhedral links is PSPACE-hard [47]. The proof uses the robot's dofs to both encode the configuration of a polynomial space bounded Turing machine and design obstacles which force the robot's motions to simulate the computation of this machine. It provides strong evidence that any complete algorithm will require exponential time in the number of dofs. This result remains true in more specific cases, for instance when the robot is a planar arm in which all joints are revolute. However, it no longer holds in some very simple settings; for instance, planning the path of a planar arm within an empty circle is in P. For a collection of complexity results on motion planning see [30].

Most complete algorithms first capture the connectivity of the free space into a graph, either by partitioning the free space into a collection of cells (exact cell decomposition techniques), or by extracting a network of curves (roadmap techniques) [30]. General and specific complete planners have been proposed. The general ones apply to virtually any robot with an arbitrary number of dofs. The specific ones apply to a restricted family of robots usually having a fixed small number of dofs.

The general algorithm in [49] computes a cylindrical cell decomposition of the free space using the Collins method. It takes doubly exponential time in the number m of dofs of the robot. The roadmap algorithm in [10] computes a semifree path in time singly exponential in m. Both algorithms are polynomial in the number of polynomial constraints defining the free space and their maximal degree. Specific algorithms have been developed mainly for robots with 2 or 3 dofs. For a k-sided polygonal robot moving freely in a polygonal workspace, the algorithm in [24] takes $O((kn)^{2+\epsilon})$ time, where n is the total number of edges of the workspace, for any $\epsilon > 0$.

Probabilistic Algorithms

The complexity of path planning for robots with many dofs (more than 4 or 5) has led the development of computational schemes that trade off completeness against time. One such scheme, probabilistic planning [4], avoids computing an explicit geometric representation of the free space. Instead, it uses an efficient procedure to compute distances between bodies in the workspace. It samples the configuration space by selecting a large number of configurations at random and retaining only the free configurations as milestones. It then checks if each pair of milestones can be connected by a collision-free straight path in configuration space. This computation yields the graph (V, E), called a probabilistic roadmap, where V is the set of milestones and E is the set of pairs of milestones that have been connected.

Various strategies can be applied to sample the configuration space. The strategy in [27] proceeds as sketched above. Once a roadmap has been computed, it is used to process an arbitrary number of path planning queries.

The results reported in [4] bound the number of milestones generated by the algorithm in [27], under the assumption that the configuration space verifies some simple geometric property. One such property is ϵ-goodness: a set S of volume μ is said to be ϵ-good, if every point in S sees a subset of S of volume at least $\epsilon \times \mu$. Under such an assumption, the number of milestones needed to correctly answer path planning queries with probability $1 - \alpha$ is proportional to $(1/\epsilon)(\log(1/\epsilon) + \log(1/\alpha))$.

Heuristic Algorithms

Several heuristic techniques have been proposed to speedup path planning. Some of them work well in practice, but they usually offer no performance guarantee.

Heuristic algorithms often search a regular grid defined over configuration space and generate a path as a sequence of adjacent grid points. The search can be guided by a potential field, a function over the free space that has a global minimum at the goal configuration. This function may be constructed as the sum of an attractive and a repulsive field [28]. The attractive field has a single minimum at the goal and grows to infinity as the distance to the goal increases. The repulsive field is zero at all configurations where the distance between the robot and the obstacles is greater than some predefined value, and grows to infinity as the robot gets closer to an obstacle.

One may also construct grids at variable resolution. Hierarchical space decomposition techniques such as octrees and boxtrees have been used to that purpose [30].

Path Planning for Nonholonomic Robots

In some cases, the configuration parameters of a robot are not directly controllable. This is in particular the case with nonholonomic robots. Informally, for such robots, the number of parameters that can be controlled is smaller than the number of parameters defining a configuration, which raises controllability issues introduced in "Robot Algorithms Control."

Mathematical Background

The trajectories of a nonholonomic robot are constrained by $p \geq 1$ nonintegrable scalar equality constraints:

$$G(\boldsymbol{q}(t), \dot{\boldsymbol{q}}(t)) = \Big(G^1(\boldsymbol{q}(t), \dot{\boldsymbol{q}}(t)), \ldots, G^p(\boldsymbol{q}(t), \dot{\boldsymbol{q}}(t))\Big) = (0, \ldots, 0) \ ,$$

where $\dot{\boldsymbol{q}}(t) \in T_{\boldsymbol{q}(t)}(\mathcal{C})$ designates the velocity vector along the **trajectory** $\boldsymbol{q}(t)$. At every $\boldsymbol{q}$, the function $G_{\boldsymbol{q}} = G(\boldsymbol{q}, .)$ maps the tangent space[1] $T_{\boldsymbol{q}}(\mathcal{C})$ into $\mathbb{R}^p$. If $G_{\boldsymbol{q}}$ is smooth and its Jacobian has full rank (two conditions that are often satisfied), the constraint $G_{\boldsymbol{q}}(\dot{\boldsymbol{q}}) = (0, \ldots, 0)$ constrains $\dot{\boldsymbol{q}}$ to be in a linear subspace of $T_{\boldsymbol{q}}(\mathcal{C})$ of dimension $m - p$. The nonholonomic robot may also be subject to scalar inequality constraints of the form $H^j(\boldsymbol{q}, \dot{\boldsymbol{q}}) > 0$. The subset of $T_{\boldsymbol{q}}(\mathcal{C})$ that satisfies all the constraints on $\dot{\boldsymbol{q}}$ is called the set $\Omega(\boldsymbol{q})$ of controls at $\boldsymbol{q}$. A feasible path is a piecewise differentiable path whose tangent lies everywhere in the control set.

A car-like robot is a classical example of a nonholonomic robot. It is constrained by one equality constraint (the linear velocity must point along the car's axis). Limits on the steering angle impose two inequality constraints. Other nonholonomic robots include tractor-trailers, airplanes, and satellites.

A key question when dealing with a nonholonomic robot is: Despite the relatively small number of controls, can the robot span its configuration space? The study of this question requires introducing some controllability notions. Given an arbitrary subset $U \subset \mathcal{C}$, the configuration $\boldsymbol{q}_1 \in U$ is said to be U-accessible from $\boldsymbol{q}_0 \in U$ if there exists a piecewise constant control $\dot{\boldsymbol{q}}(t)$ in the control set whose integral is a trajectory joining $\boldsymbol{q}_0$ to $\boldsymbol{q}_1$ that fully lies in U. Let $A_U(\boldsymbol{q}_0)$ be the set of configurations U-accessible from $\boldsymbol{q}_0$. The robot is said to be locally controllable at $\boldsymbol{q}_0$ iff for every neighborhood U of $\boldsymbol{q}_0$, $A_U(\boldsymbol{q}_0)$ is also a neighborhood of $\boldsymbol{q}_0$. It is locally controllable iff this is true for all $\boldsymbol{q}_0 \in \mathcal{C}$. Car-like robots and tractor-trailers that can go forward and backward are locally controllable [5].

Let X and Y be two smooth vector fields on $\mathcal{C}$. The Lie bracket of X and Y, denoted by $[X, Y]$, is the smooth vector field on $\mathcal{C}$ defined by $[X, Y] = dY \cdot X - dX \cdot Y$, where dX and dY, respectively, denote the $m \times m$ matrices of the partial derivatives of the components of X and Y w.r.t. the configuration coordinates in a chart placed on $\mathcal{C}$. The Control Lie Algebra associated with the control set Ω, denoted by $L(\Omega)$, is the space of all linear combinations of vector fields in Ω closed by the Lie bracket operation. The following

[1] The tangent space $T_p(M)$ at a point p of a smooth manifold M is the vector space of all tangent vectors to curves contained in M and passing through p. It has the same dimension as M.

result derives from the Controllability Rank Condition Theorem [5]: *A robot is locally controllable if, for every* $\boldsymbol{q} \in \mathcal{C}$, $\Omega(\boldsymbol{q})$ *is symmetric with respect to the origin of* $T_{\boldsymbol{q}}(\mathcal{C})$ *and the set* $\{X(\boldsymbol{q}) \mid X(\boldsymbol{q}) \in L(\Omega(\boldsymbol{q}))\}$ *has dimension* m.

The minimal number of Lie brackets sufficient to express any vector in $L(\Omega)$ using vectors in Ω is called the degree of nonholonomy of the robot. The degree of nonholonomy of a car-like robot is 2. Except at some singular configurations, the degree of nonholonomy of a tractor towing a chain of s trailers is $2 + s$. Intuitively, the higher the degree of nonholonomy the more complex (and the slower) the robot's maneuvers to perform some motions.

Planning for Controllable Robots

Let $\mathcal{A}$ be a locally controllable nonholonomic robot. A necessary and sufficient condition for the existence of a feasible free path of $\mathcal{A}$ between two given configurations is that they lie in the same connected component of the open free space. Indeed, local controllability guarantees that a possibly nonfeasible path can be decomposed into a finite number of subpaths, each short enough to be replaced by a feasible free subpath [31]. Hence, deciding if there exists a free path for a locally controllable nonholonomic robot has the same complexity as deciding if there exists a free path for the holonomic robot having the same geometry.

Transforming a nonfeasible free path τ into a feasible one can be done by recursively decomposing τ into subpaths. The recursion halts at every subpath that can be replaced by a feasible free subpath. Specific substitution rules (e.g., Reeds and Shepp curves) have been defined for car-like robots [31]. The complexity of transforming a nonfeasible free path τ into a feasible one is of the form $O(\epsilon^d)$, where ϵ is the smallest clearance between the robot and the obstacles along τ and d is the degree of nonholonomy of the robot.

The algorithm in [5] directly constructs a nonholonomic path for a car-like or a tractor-trailer robot by searching a tree obtained by concatenating short feasible paths, starting at the robot's initial configuration. The planner is guaranteed to find a path if one exists, provided that the length of the short feasible paths is small enough. It can also find paths that minimize the number of cusps (changes of sign of the linear velocity).

Planning for Noncontrollable Robots

Path planning for nonholonomic robots that are not locally controllable is much less understood. Research has almost exclusively focused on car-like robots that can only move forward. The algorithm in [17] decides whether there exists such a path between two configurations, but it runs in time exponential in obstacle complexity. The algorithm in [1] computes a path in polynomial time under the assumptions that all obstacles are convex and their boundaries have a curvature radius greater than the minimum turning radius of the point (so called "moderate obstacles"). Other polynomial algorithms [5] require some sort of discretization.

Motion Planning with Uncertainty

In practice, robots deviate from planned paths due to errors in control and position sensing. Such errors raise recognizability issues which make planning more complex.

Problem Formulation

The inputs to a motion planning problem with uncertainty are the initial region $I \subset \mathcal{C}$, in which the robot is known to be prior to moving, the goal region $G \subset \mathcal{C}$, in which it should terminate its motion, and the uncertainty in control and sensing. Uncertainty is specified in the form of regions. For instance, the uncertainty in position sensing is the set of possible actual robot's configurations given the sensor readings. The output is a series of motion commands, if one exists, whose execution enables the robot to reach G

from I. Each command is described by a velocity vector $\boldsymbol{v}$ and a termination condition T. The vector $\boldsymbol{v}$ specifies the desired behavior of the robot over time (with or without compliance). The condition T is a Boolean function of the sensor readings and time which causes the motion to stop as soon as it becomes true. A plan may contain conditional branchings. In [10] this problem is shown NEXPTIME-hard for a point robot moving in 3-space among polyhedral obstacles.

Preimage of a Goal

Given a goal G and a command $(\boldsymbol{v}, T)$, a preimage of G is any region $P \subset \mathcal{C}$ such that executing the command from anywhere in P makes the robot reach and stop in G [36]. One way to compute a (nonmaximal) preimage is to restrict the termination condition so that it recognizes G independently of the region from which the motion started [15]. For example, one may shrink G to a subset K, called the kernel of G, such that whenever the robot is in K, all robot configurations consistent with the current sensor readings are in G. A preimage is then computed as the region from which the robot commanded along $\boldsymbol{v}$ is guaranteed to reach K. This region is called the backprojection of K for $\boldsymbol{v}$. This preimage computation approach has been well studied in a polygonal configuration space when G is a polygon [30].

One-Step Planning

In a polygonal configuration space, the kernel of a polygonal goal is either independent of the selected $\boldsymbol{v}$ or changes at a number of critical orientations of $\boldsymbol{v}$ that is linear in the workspace complexity [30]. Moreover, the backprojection of a polygonal region K, when the orientation of $\boldsymbol{v}$ varies, changes topology only at a quadratic number of critical directions. Its intersection with a polygonal initial region I of constant complexity also changes qualitatively at few directions of $\boldsymbol{v}$. Checking the containment of I by the backprojection at each such direction yields a one-step motion plan, if one exists, in amortized time $O(n^2 \log n)$, where n is the number of edges in $\mathcal{C}$ [8].

Multistep Planning

For multistep planning, algebraic approaches that check the satisfiability of a first-order semi-algebraic formula have been proposed. In [11] it is assumed that all possible trajectories have an algebraic description. The approach there is based on a two-player-game interpretation of planning, where the robot is one player and nature the other. Each step of a plan contributes three quantifiers: one existential quantifier applies to the direction of motion, and corresponds to choosing this direction; another existential quantifier applies to time, and corresponds to choosing when to terminate the motion; one universal quantifier applies to the sensor readings and represents the unknown action of nature. The formula representing an r-step plan thus contains r quantifier alternations; checking its satisfiability takes double-exponential time in r, which is itself polynomial in the total complexity of the robot and the workspace.

Landmark-Based Planning

Often a workspace contains features that can be reliably sensed and used to precisely localize the robot. Each such landmark feature induces a region in configuration space called the landmark area from which the robot can sense the corresponding feature.

The planner in [33] considers a point robot among n circular obstacles and $O(n)$ circular landmark areas. It assumes perfect position sensing and motion control in landmark areas. Outside these areas, it assumes that the robot has no position sensing and that directional errors in control are bounded by the angle θ. Given circular initial and goal regions I and G (with G intersecting at least one landmark area), the planner constructs a motion plan that enables the robot to move from landmark areas to landmark areas, until it reaches the goal. It proceeds backward by computing P_1 – the preimage of the landmark regions intersecting G. Then it computes the preimage P_2 of the landmark regions intersected by P_1, and

so on, until a preimage contains I. The planner runs in $O(n^4 \log n)$ time; it is complete and generates plans that minimize the number of steps to be executed in the worst case.

Other Motion Planning Issues

There are many other useful extensions of the basic path planning problem [30]. Below we briefly present some of them.

Dynamic Workspace

In the presence of moving obstacles, one can no longer plan a robot's motion as a mere geometric path. The path must be indexed by time and is then called a trajectory. It can be represented in the configuration×time space $\mathcal{C} \times [0, +\infty)$ of the robot. All workspace obstacles map to static forbidden regions in that space. A free trajectory is a free path in that space whose tangent at every point points positively along the time axis (or within a more restricted cone, if the robot's velocity modulus is bounded).

Computing a free trajectory for a rigid object in 3-space among arbitrarily moving obstacles (with known trajectories) is PSPACE-hard if the robot's velocity is bounded, and NP-hard otherwise [48]. The problem remains NP-hard for a point robot moving with bounded velocity in the plane among convex polygonal obstacles translating at constant linear velocities [10]. A complete planning algorithm is given in [48] for a polygonal robot that translates in the plane among polygonal obstacles translating at fixed velocities. This algorithm takes time exponential in the number of moving obstacles and polynomial in the total number of edges of the robot and the obstacles.

Coordination of Multiple Robots

The case of multiple robots can be trivially addressed by considering them as the components of a single robot, that is, by planning a path in the cross product of their configuration spaces. This product is called the composite configuration space of the robots and the approach is referred to as centralized planning.

One may try to reduce complexity by separately computing a path for each robot, before tuning the robots' velocities along their respective paths to avoid inter-robot collision (decoupled planning) [25]. Although inherently incomplete, decoupled planning may work well in some practical applications.

Manipulation Planning

Many robot tasks consist of achieving arrangements of physical objects. Such objects, called movable objects, cannot move autonomously; they must be moved by a robot. Planning with movable objects is called manipulation planning.

In [53] the robot $\mathcal{A}$ and the movable object M are both convex polygons in a polygonal workspace. The goal is to bring $\mathcal{A}$ and M to specified positions. $\mathcal{A}$ can only translate. To grasp M, $\mathcal{A}$ must have one of its edges that exactly coincides with an edge of M. While $\mathcal{A}$ grasps M, they move together as one rigid object. An exact cell decomposition algorithm is given that runs in $O(n^2)$ time after $O(n^3 \log^2 n)$ preprocessing, where n is the total number of edges in the workspace, the robot, and the movable object. An extension of this problem allowing an infinite set of grasps is solved by an exact cell decomposition algorithm in Alami et al. [3].

Heuristic algorithms have also been proposed. The planner in [29] first plans the path of the movable object M. During that phase, it only verifies that for every configuration taken by M there exists at least one collision-free configuration of the robot where it can hold M. In the second phase, the planner determines the points along the path of M where the robot must change grasps. It then computes the paths where the robot moves alone to (re)grasp M. The paths of the robot when it carries M are obtained through inverse kinematics. This planner is not complete, but it has solved complex tasks in practice.

Optimal Planning

There has been considerable research in computational geometry on finding shortest Euclidean paths, but minimal Euclidean length is usually not the most suitable criterion in robotics. Rather, one wishes to minimize execution time, which requires taking the robot's dynamics into account.

Optimal-Time Control Planning The input is a geometric free path τ parameterized by $s \in [0, L]$, the distance travelled from the starting configuration. The problem is to find the time parametrization $s(t)$ that minimizes travel time along τ, while satisfying actuator limits.

The dynamic equation of motion of a robot arm with m dofs can be written as $M(\boldsymbol{q})\ddot{\boldsymbol{q}} + V(\dot{\boldsymbol{q}}, \boldsymbol{q}) + G(\boldsymbol{q}) = \Gamma$, where $\boldsymbol{q}, \dot{\boldsymbol{q}}$, and $\ddot{\boldsymbol{q}}$, respectively, denote the robot's configuration, velocity, and acceleration [12]. M is the $m \times m$ inertia matrix of the robot, V the m-vector (quadratic in $\dot{\boldsymbol{q}}$) of centrifugal and Coriolis forces, and G the m-vector of gravity forces. Γ is the m-vector of the torques applied by the joint actuators.

Using the fact that the robot follows τ, this equation can be rewritten in the form: $\boldsymbol{m}\ddot{s} + \boldsymbol{v}\dot{s}^2 + \boldsymbol{g} = \Gamma$, where $\boldsymbol{m}$, $\boldsymbol{v}$, and $\boldsymbol{g}$ are derived from M, V, and G, respectively. Minimum-time control planning becomes a two-point boundary value problem: find $s(t)$ that minimizes $t_f = \int_0^L ds/\dot{s}$, subject to $\Gamma = \boldsymbol{m}\ddot{s} + \boldsymbol{v}\dot{s}^2 + \boldsymbol{g}$, $\Gamma_{min} \leq \Gamma \leq \Gamma_{max}$, $s(0) = 0$, $s(t_f) = L$, and $\dot{s}(0) = \dot{s}(L) = 0$. Numerical techniques solve this problem by finely discretizing the path τ.

Minimal-Time Trajectory Planning Finding a minimal-time trajectory is called kinodynamic motion planning. One approach is to first plan a geometric free path and then iteratively deform this path to reduce travel time. Each iteration requires checking the new path for collision and recomputing the optimal-time control. No bound has been established on the running time of this approach or the goodness of its outcome. Kinodynamic planning is NP-hard for a point robot under Newtonian mechanics in 3-space. The approximation algorithm in [13] computes a trajectory ϵ-close to optimal in time polynomial in both $1/\epsilon$ and the **workspace complexity.**

Discovery and On-Line Planning

On-line planning addresses the case where the workspace is initially unknown or partially unknown. As the robot moves, it acquires new partial information about the workspace through sensing. A motion plan is generated using the partial information that is available and updated as new information is acquired.

Early examples of on-line planners are reviewed in [37]. Most of these planners apply to a point robot in 2-space that must go from a start position s to a goal position g among unknown obstacles, each bounded by a Jordan curve of measurable length. The robot is equipped with perfect position and touch sensors. The planners select a mix of motions following either the line segment connecting s to g or boundaries of hit obstacles. The main consideration is the length of the generated path, expressed as a function of the distance between s and g, the number of obstacles, and their perimeters.

Another way of evaluating an on-line planner is competitive analysis. The competitive ratio of an on-line planner is the maximal ratio (over all possible workspaces) between the length of the path generated by the on-line algorithm and the length of the shortest path [44]. Competitive analysis is not restricted to path length and can be applied to other measures of performance as well.

Sensing

Sensing allows a robot to acquire information about its workspace and localize itself. Here we mention a few selected topics.

Model Building

Consider a mobile robot in an unknown workspace W. A first task for this robot is likely to be the construction of a geometric model (also called a map) of W [55]. This requires the robot to perform a series of sensing operations at different locations. Each operation yields a partial model. The robot must patch

together the successively obtained partial models to eventually form a complete map of the workspace. This problem is complicated by the fact that the robot has imperfect control and cannot accurately keep track of its position in a fixed coordinate system.

Robot Localization

A robot often has to localize itself relative to its workspace W. A model of W is given and localization is done by matching sensory inputs against this model to infer the robot configuration. This problem usually arises for mobile robots. Other types of robots, such as robot arms, often have absolute references (e.g., mechanical stops) and internal sensors (e.g., joint encoders) that provide configurations more directly. Mobile robots have wheel encoders allowing dead-reckoning, but the absence of absolute reference on the one hand and slipping on the ground on the other hand usually require sensor-based localization. GPS (global positioning system) has recently become a widely available alternative, but it does not work in all environments.

Two kinds of sensor-based localization problems can be distinguished, static and dynamic. In the static problem, the robot is placed at an arbitrary unknown configuration and the problem is to compute this configuration. In the dynamic problem, the robot moves continuously and must regularly update its configuration. The second problem consists of refining an available estimate of the current configuration; but the computation must be done in real time. The static problem is usually more complex; but computation time is less restricted. A preprocessing approach to the static localization problem for a point robot equipped with a 360° range sensor is presented in [22]. Practical techniques for localization are described in many papers [52].

Additional Issues in Sensing

Sensor placement is the problem of computing the set of placements from which a sensor can monitor a region within a given workspace [8]. Another problem is to choose a minimal set of sensors and their placement so as to completely cover a given region. Additionally, there has been considerable interest in reconstructing shapes of objects using simple sensors, called probes. See [50] for a review of problems and results in this area. Matching and aspect graphs are two related topics that have been well studied, mainly in computer vision.

21.4 Distance Computation

The efficient computation of (minimum) distances between bodies in 2- and 3-space is a crucial element of many algorithms in robotics.

Algorithms have been proposed to efficiently compute distances between two convex bodies. In [14], an algorithm is given which computes the distance between two convex polygons P and Q (together with the points that realize it) in $O(\log p + \log q)$ time, where p and q denote the number of vertices of P and Q, respectively. This time is optimal in the worst case. The algorithm is based on the observation that the minimal distance is realized between two vertices or between a vertex and an edge. It represents P and Q as sequences of vertices and edges and performs a binary search that eliminates half of the edges in at least one sequence at each step. A widely tested numerical descent technique is described in [18] to compute the distance between two convex polyhedra; extensive experience indicates that it runs in approximately linear time in the total complexity of the polyhedra.

Most robotics applications, however, involve many bodies. Typically, one must compute the minimum distance between two sets of bodies, one representing the robot, the other the obstacles. Each body can be quite complex and the number of bodies forming the obstacles can be large. The cost of accurately computing the distance between every pair of bodies is often prohibitive. In that context, simple bounding volumes, such as parallelepipeds and spheres, have been extensively used to reduce computation time. They are often coupled with hierarchical decomposition techniques, such as octrees, boxtrees, or sphere trees.

(For an example, see [46].) These techniques make it possible to rapidly eliminate pairs of bodies that are too far apart to contribute the minimum distance.

When motion is involved, incremental distance computation has been suggested for tracking the closest points on a pair of convex polyhedra [34]. It takes advantage of the fact that the closest features (faces, edges, vertices) change infrequently as the polyhedra move along finely discretized paths.

21.5 Research Issues and Summary

In this chapter we have introduced robot algorithms as abstract descriptions of processes consisting of motions and sensing operations in the physical space. Robot algorithms send commands to actuators and sensors in order to control a subset of the real world, the workspace, despite the fact that the workspace is subject to the imperfectly modeled laws of nature. Robot algorithms uniquely blend controllability, observability, computational complexity, and physical complexity issues, as described in Section 21.2. Research on robot algorithms is broad and touches many different areas. In Section 21.3 we have surveyed a number of selected areas in which research has been particularly active: part manipulation (grasping, fixturing, feeding), assembly sequencing, motion planning (including basic path planning, nonholonomic planning, planning with uncertainty, dynamic workspaces, and optimal-time planning), and sensing.

Many of the core issues reviewed in Section 21.2 have been barely addressed in currently existing algorithms. There is much more to understand in how controllability, observability, and complexity interact in robot tasks. The interaction between controllability and complexity has been studied to some extent for nonholonomic robots. The interaction between observability (or recognizability) and complexity has been considered in motion planning with uncertainty. But, in both cases, much more remains to be done.

Concerning the areas studied in Section 21.3, several specific problems remain open. We list a few below (by no means is this list exhaustive):

- Given a workspace W, find the optimal design of a robot arm that can reach everywhere in W without collision. The three-dimensional case is largely open. An extension of this problem is to design the layout of the workspace so that a certain task can be completed efficiently.
- Given the geometry of the parts to be manipulated, predict feeders' throughputs to evaluate alternative feeder designs. In relation to this problem, simulation algorithms have been used to predict the pose of a part dropped on a flat surface [39].
- In assembly planning, the complexity of an NDBG grows exponentially with the number of parameters that control the allowable motions. Are there situations where only a small portion of the full NDBG need be constructed?
- Develop efficient sampling techniques for searching the configuration space of robots with many degrees of freedom in the context of the scheme given in [4].
- Establish a nontrivial lower bound on the complexity of planning for a nonholonomic robot that is not locally controllable.

21.6 Defining Terms

Basic path planning problem: Compute a free or semifree path between two input configurations for a robot moving in a known and static workspace.

C-Obstacle: Given an obstacle B_i, the subset CB_i of the configuration space $\mathcal{C}$ such that, for any $\boldsymbol{q} \in CB_i$, $\mathcal{A}(\boldsymbol{q})$ intersects B_i. The union $CB = \cup_i CB_i$ plus the configurations that violate the mechanical limits of the robot's joints is called the **C-obstacle region**.

Complete motion planner: A planner guaranteed to find a (semi)free path between two given configurations whenever such a path exists, and to notify that no such path exists otherwise.

Configuration: Any mathematical specification of the position and orientation of every body composing a robot $\mathcal{A}$, relative to a fixed coordinate system. The configuration of a single body is also called a **placement** or a **pose**.

Configuration space: Set $\mathcal{C}$ of all configurations of a robot. For almost any robot, this set is a smooth manifold.

Free path: A path in free space.

Free space: The complement of the C-obstacle region in $\mathcal{C}$, that is, $\mathcal{C} \setminus \mathcal{CB}$.

Linkage: A collection of rigid objects, called links, in which some pairs of links are connected by joints (e.g., revolute and/or prismatic joints). Most industrial robot arms are serial linkages with actuated joints.

Number of degrees of freedom: The dimension m of $\mathcal{C}$.

Obstacle: The workspace W is often defined by a set of obstacles (bodies) B_i $(i = 1, \ldots, q)$ such that $W = \mathbb{R}^k \setminus \bigcup_1^q B_i$.

Path: A continuous map $\tau : [0, 1] \to \mathcal{C}$.

Semifree path: A path in the closure of free space.

Trajectory: Path indexed by time.

Workspace: A subset of the two- or three-dimensional physical space modeled by $W \subset \mathbb{R}^k, k = 2, 3$.

Workspace complexity: The total number of features (vertices, edges, faces, etc.) on the boundary of the obstacles.

References

[1] Agarwal, P.K., Raghavan, P., and Tamaki, H., Motion Planning for a Steering-Constrained Robot Through Moderate Obstacles. *Proc. 28th ACM STOC,* 343–352, 1995.

[2] Akella, S., Huang, W., Lynch, K., and Mason, M.T., Planar Manipulation on a Conveyor with a One Joint Robot. In *Robotics Research,* Giralt, G. and Hirzinger, G., Eds., Springer, 265–276, 1996.

[3] Alami, R., Laumond, J.P., and Siméon, T., Two Manipulation Algorithms. In *Algorithmic Foundations of Robotics,* Goldberg, K.Y., Wellesley, M.A., Eds., AK Peters, 109–125, 1995.

[4] Barraquand, J., Kavraki, L.E., Latombe, J.C., Li, T.Y., Motwani, R., and Raghavan, P., A Random Sampling Framework for Path Planning in Large-Dimensional Configuration Spaces. *Int. J. of Robotics Research,* 16(6), 759–774, 1997.

[5] Barraquand, J. and Latombe, J.C., Nonholonomic Multibody Mobile Robots: Controllability and Motion Planning in the Presence of Obstacles, *Algorithmica,* 10(2-3-4), 121–155, 1993.

[6] de Berg, M., van Kreveld, M., Overmars, M. and Schwarzkopf, O., *Computational Geometry: Algorithms and Applications.* Springer, New York, 1997.

[7] Boddy M. and Dean T.L., Solving Time-Dependent Planning Problems, *Proc. 11th Int. Joint Conf. on Artificial Intelligence,* 979–984, 1989.

[8] Briggs, A.J., *Efficient Geometric Algorithms for Robot Sensing and Control.* Report No. 95-1480, Dept. of Computer Science, Cornell University, Ithaca, NY, 1995.

[9] Brost, R.C. and Goldberg, K.Y., Complete Algorithm for Designing Planar Fixtures Using Modular Components. *IEEE Tr. on Systems, Man and Cybernetics,* 12, 31–46, 1996.

[10] Canny, J.F., *The Complexity of Robot Motion Planning.* MIT Press, Cambridge, MA, 1988.

[11] Canny, J.F., On Computability of Fine Motion Plans, *Proc. IEEE Int. Conf. on Robotics and Automation,* Scottsdale, AZ, 177–182, 1989.

[12] Craig, J.J., *Introduction to Robotics. Mechanics and Control.* Addison-Wesley, Reading, MA, 1986.

[13] Donald, B.R., Xavier, P., Canny, J.F., and Reif, J.H., Kinodynamic Motion Planning. *J. of the ACM,* 40, 1048–1066, 1993.
[14] Edelsbrunner, H., Computing the Extreme Distances between Two Convex Polygons. *J. of Algorithms,* 6, 213–224, 1985.
[15] Erdmann, M., Using Backprojections for Fine Motion Planning with Uncertainty. *Int. J. of Robotics Research*, 5, 19–45, 1986.
[16] Erdmann, M. and Mason, M.T., An Exploration of Sensorless Manipulation. *IEEE Tr. on Robotics and Automation,* 4(4), 369–379, 1988.
[17] Fortune, S. and Wilfong, G.T., Planning Constrained Motions. In *Proc. ACM Symp. on Theory of Computing,* 445–459, 1988.
[18] Gilbert, E.G., Johnson, D.W., and Keerthi, S.S., A Fast Procedure for Computing Distance Between Complex Objects in Three-Dimensional Space. *IEEE Tr. on Robotics and Automation,* 4, 193–203, 1988.
[19] Giralt, G. and Hirzinger, G., Eds., *Robotics Research,* Springer, 1996.
[20] Goldberg, K.Y., Orienting Polygonal Parts without Sensors. *Algorithmica,* 10(2-3-4), 201–225, 1993.
[21] Goldberg, K.Y., Halperin, D., Latombe, J.C., and Wilson, R.H., Eds., *Algorithmic Foundations of Robotics,* AK Peters, Ltd., Wellesley, MA, 1995.
[22] Guibas, L., Motwani, R., and Raghavan, P., The Robot Localization Problem in Two Dimensions. *SIAM J. on Computing,* 26(4), 1121–1138, 1996.
[23] Halperin, D. Arrangements. In Goodman, J.E. and O'Rourke, J., Eds., *Handbook of Discrete and Computational Geometry,* CRC Press, Boca Raton, FL, 389–412, 1997.
[24] Halperin, D. and Sharir, M., Near-Quadratic Algorithm for Planning the Motion of a Polygon in a Polygonal Environment. *Discrete Computational Geometry,* 16, 121–134, 1996.
[25] Kant, K.G. and Zucker, S.W., Toward Efficient Trajectory Planning: Path Velocity Decomposition. *Int. J. of Robotics Research,* 5, 72–89, 1986.
[26] Kavraki, L.E. and Kolountzakis, M.N., Partitioning a Planar Assembly Into Two Connected Parts is NP-complete. *Information Processing Letters,* 55, 159–165, 1995.
[27] Kavraki, L.E., Švestka, P., Latombe, J.C., and Overmars, M., Probabilistic Roadmaps for Fast Path Planning in High Dimensional Configuration Spaces. *IEEE Tr. on Robotics and Automation,* 12, 566–580, 1996.
[28] Khatib, O. Real-Time Obstacle Avoidance for Manipulators and Mobile Robots. *Int. J. of Robotics Research,* 5, 90–98, 1986.
[29] Koga, Y., Kondo, K., Kuffner, J., and Latombe, J.C., Planning Motions with Intentions. *Proc. ACM SIGGRAPH'94,* 395–408, 1994.
[30] Latombe J.C., *Robot Motion Planning,* Kluwer Academic Publishers, Boston, MA, 1991.
[31] Laumond, J.P., Jacobs, P., Taix, M., and Murray, R., A Motion Planner for Nonholonomic Mobile Robots. *IEEE Tr. on Robotics and Automation,* 10, 577–593, 1994.
[32] Laumond, J.P. and Overmars, M., Eds., *Algorithms for Robot Motion and Manipulation,* AK Peters, Wellesley, MA, 1997.
[33] Lazanas, A. and Latombe, J.C., Landmark-Based Robot Navigation. *Algorithmica,* 13, 472–501, 1995.
[34] Lin, M.C. and Canny, J.F., A Fast Algorithm for Incremental Distance Computation. *Proc. IEEE Int. Conf. on Robotics and Automation,* 1008–1014, 1991.
[35] Lozano-Pérez T., Spatial Planning: A Configuration Space Approach, *IEEE Tr. on Computers,* 32(2), 108–120, 1983.
[36] Lozano-Pérez T., Mason, M.T., and Taylor, R.H., Automatic Synthesis of Fine-Motion Strategies for Robots, *Int. J. of Robotics Research,* 3(1), 3–24, 1984.
[37] Lumelsky, V., A Comparative Study on the Path Length Performance of Maze-Searching and Robot Motion Planning Algorithms. *IEEE Tr. on Robotics and Automation,* 7, 57–66, 1991.

[38] Mason, M.T., Mechanics and Planning of Manipulator Pushing Operations, *Int. J. of Robotics Research,* 5(3), 53–71, 1986.
[39] Mirtich, B., Zhuang, Y., Goldberg, K., Craig, J.J., Zanutta, R., Carlisle, B., and Canny, J.F., Estimating Pose Statistics for Robotic Part Feeders. *Proc. IEEE Int. Conf. on Robotics and Automation,* 1140–1146, 1996.
[40] Mishra B., Schwartz, J.T., and Sharir, M., On the Existence and Synthesis of Multifinger Positive Grips, *Algorithmica,* 2, 541–558, 1987.
[41] Natarajan, B.K., On Planning Assemblies. *Proc. 4th ACM Symp. on Computational Geometry,* 299–308, 1988.
[42] Nguyen, V.D., Constructing Force-Closure Grasps. *Int. J. of Robotics Research,* 7, 3–16, 1988.
[43] Nourbakhsh, I.R., *Interleaving Planning and Execution.* Ph.D. Thesis. Dept. of Computer Science, Stanford University, Stanford, CA, 1996.
[44] Papadimitriou, C.H. and Yannakakis, M., Shortest Paths Without a Map. *Theoretical Computer Science,* 84, 127–150, 1991.
[45] Ponce, J., Sudsang, A., Sullivan, S., Faverjon, B., Boissonnat, J.D., and Merlet, J.P., Algorithms for Computing Force-Closure Grasps of Polyhedral Objects. In *Algorithmic Foundations of Robotics,* Golberg, K.Y and Wellesley, M.A., Eds., AK Peters, 167–184, 1995.
[46] Quinlan, S., Efficient Distance Computation between Non-Convex Objects. *Proc. IEEE Int. Conf. on Robotics and Automation,* 3324–3329, 1994.
[47] Reif J.H., Complexity of the Mover's Problem and Generalizations. *Proc. FOCS,* 421–427, 1979.
[48] Reif, J.H. and Sharir, M., Motion Planning in the Presence of Moving Obstacles. *Journal of the ACM,* 41(4), 764–90, 1994.
[49] Schwartz, J.T. and Sharir, M., On the 'Piano Movers' Problem: II. General Techniques for Computing Topological Properties of Real Algebraic Manifolds, *Advances in Applied Mathematics,* 4, 298–351, 1983.
[50] Skiena, S.S., Geometric reconstruction problems. In Goodman, J.E. and O'Rourke, J., Eds., *Handbook of Discrete and Computational Geometry*, CRC Press, Boca Raton, FL, 481–490, 1997.
[51] Snoeyink, J. and Stolfi, J., Objects That Cannot Be Taken Apart with Two Hands. *Discrete Computational Geometry,* 12, 367–384, 1994.
[52] Talluri, R. and Aggarwal, J.K., Mobile Robot Self-Location Using Model-Image Feature Correspondence. *IEEE Tr. on Robotics and Automation,* 12, 63–77, 1996.
[53] Wilfong, G.T., Motion Planning in the Presence of Movable Objects. *Annals of Mathematics and Artificial Intelligence,* 3, 131–150, 1991.
[54] Wilson, R.H. and Latombe, J.C., Reasoning About Mechanical Assembly. *Artificial Intelligence,* 71, 371–396, 1995.
[55] Zhang, Z. and Faugeras, O., A 3D World Model Builder with a Mobile Robot. *Int. J. of Robotics Research,* 11, 269–285, 1996.
[56] Zhuang, Y., Goldberg, K.Y., and Wong, Y., On the existence of modular fixtures. *Proc. IEEE Int. Conf. on Robotics and Automation,* 543–549.

Further Information

For an introduction to robot arm kinematics, dynamics and control, see [12]. Robot motion planning and its variants are discussed in [30]. Research in all aspects of robotics is published in the IEEE Transactions of Robotics and Automation and the International Journal of Robotics Research, as well as in the proceedings of the IEEE International Conference on Robotics and Automation and the International Symposium on Robotics Research [19]. The Workshop on Algorithmic Foundations of Robotics [21, 32] emphasizes algorithmic issues in robotics. Several computational geometry books contain sections on robotics or motion planning [6].

22

Vision and Image Processing Algorithms

Concettina Guerra
Purdue University and
Universitá di Padova

22.1 Introduction

There is abundance of algorithms developed in the field of image processing and computer vision ranging from simple algorithms that manipulate binary images based on local point operations to complex algorithms for the interpretation of the symbolic information extracted from the images.

Here we concentrate on algorithms for three central problems in image processing and computer vision that are representative of different types of algorithms developed in this area. The first problem, *connectivity analysis,* has been studied since the early days of binary images. It consists of separating the objects from the background by assigning different labels to the **connected components** of an image. The algorithms to identify connected components in binary images are rather straightforward: the first one is an iterative algorithm that performs several scans of the image and uses only local operations; the next two algorithms consist of only two scans of the image and use global information in the form of an equivalence table.

The second problem is that of grouping features extracted from the image (for instance, **edge points**) into parametric curves such as straight lines and circles. An algorithm to detect lines, based on the *Hough transform,* is described that maps the image data into a parameter space that is quantized into an accumulator array. The Hough transform is a robust technique since it is relatively insensitive to noise in the sensory data and to small gaps in a line.

The last problem is that of identifying and locating objects known *a priori* in an image, a problem that is generally called *model-based object recognition*. Our discussion will focus on the matching task, that is, finding correspondences between the image and model descriptions. This correspondence can be used to solve the localization problem, i.e., the determination of a geometrical transformation that maps the model into the observed object. Finding correspondences is difficult, due to the combinatorial

nature of the problem and to noise and uncertainty in the data. We deal mainly with the case of planar objects undergoing rigid transformations in 3D space followed by scaled orthographic projections. We first assume that both the model and the image data are represented in terms of sets of feature points and describe two approaches (*alignment* and *geometric hashing*) to solve the point set matching problem. Then we consider model and image representations in terms of object **boundary** descriptions. We review *dynamic programming* algorithms for matching two sequences of boundary segments, which resemble the algorithm for the string editing problem. For multiresolution boundary representations, a tree dynamic programming algorithm is discussed. Extensions of the *indexing* techniques and of the algorithms based on the *hypothesis-and-test* paradigm to take advantage of the richer boundary descriptions are also described.

22.2 Connected Components

The connected component problem consists of assigning a label to each 1-pixel of a binary image so that two 1-pixels are assigned the same label if and only if they are connected. Two pixels are said to be connected if there is a path of 1-pixels adjacent along the horizontal and vertical directions that links them. (A different definition of connectivity includes the diagonal direction as well.) The set of connected pixels is called a connected component. Figure 22.1 shows a binary image and its connected components labeled with integers. A simple iterative algorithm to determine the connected components performs a sequence

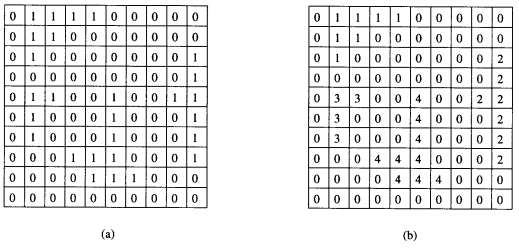


FIGURE 22.1 A 8×8 image (a) and its labeled connected components (b).

of scans over the image propagating a label from each 1-pixel to its adjacent 1-pixels. The algorithm starts with an arbitrary labeling. More precisely, the algorithm works as follows.

CONNECTED COMPONENT ALGORITHM-*Iterative*

Step 1. (*Initialization phase)*
Assign each 1-pixel a unique integer label.

Step 2. Scan the image top-down left-to-right.
Assign each 1-pixel the smallest between its own label and those of the adjacent pixels already examined in the scan sequence.

Step 3. Scan the image bottom-up right-to-left.
Like Step 2 above with a different scan order.

Alternate Step 2 and Step 3 until no changes occur.

Figure 22.2 shows a few steps of the algorithm (b-f) for the input image (a). It is clear that this algorithm is highly inefficient on conventional computers; however, it becomes attractive for implementation on parallel SIMD (single instruction multiple data stream) architectures where all the updates of the pixel's labels can be done concurrently. Another advantage of the iterative algorithm is that it does not require auxiliary memory, unlike the next two algorithms.

A common approach to finding the connected components is based on a two-pass algorithm. First the image is scanned in a top-bottom left-to-right fashion and a label is assigned to each 1-pixel based on

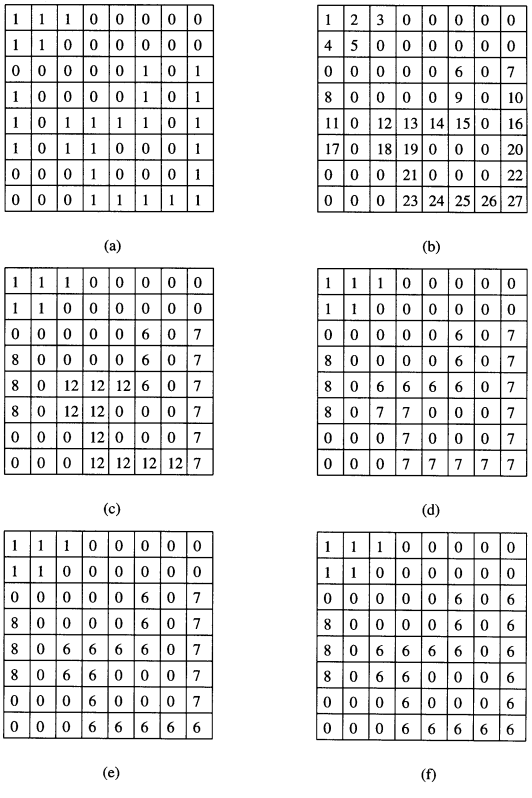


FIGURE 22.2 All the steps of the iterative algorithm for the input image (a).

the value of adjacent 1-pixels already labeled. If there are no adjacent 1-pixels, a new label is assigned. Conflicting situations may arise in which a pixel can be given two different labels. Equivalent classes of labels are then constructed and stored to be used in the second pass of the algorithm to disambiguate such conflicts. During the second scan of the image, each label is replaced by the one selected as the representative of the corresponding equivalence class; for instance, the smallest one in the class if the labels are integers. The details follow.

CONNECTED COMPONENT ALGORITHM-*Equivalence Table*

Step 1. Scan the image top-down left-to-right

for each 1-pixel **do**

if the upper and left adjacent pixels are all 0-pixels

then assign a new label

if the upper and left adjacent pixels have the same label

then assign that label

if only one adjacent pixel has a label

then assign this label

otherwise

Assign the smaller label to the pixel.

Enter the two equivalent labels into the equivalence table.

Step 2.

Find equivalence classes.

Choose a representative of each class

(i.e., the smallest label).

Step 3. Scan the image top-down left-to-right.
Replace the label of each 1-pixel with its smallest equivalent one.

Step 2. can be done using the algorithm for UNION-FIND. The drawback of the above algorithm is that it may require a large number of memory locations to store the equivalent classes.

The next algorithm overcomes this problem by building a local equivalence table that takes into account only two consecutive rows. Thus, as the image is processed the equivalences are found and resolved locally. The algorithm works in two passes over the image.

CONNECTED COMPONENT ALGORITHM-*Local Equivalence Table*

Step 1. Scan the image top-down left-to-right
for each row r of the image **do**
Initialize the local equivalence table for row r.
for each 1-pixel of row r **do**
if the upper and left adjacent pixels are all 0-pixels
then assign a new label
if the upper and left adjacent pixels have the same label
then assign this label
if only one adjacent pixel has a label
then assign this label
otherwise assign the smaller label to the pixel and
enter the two equivalent labels into the local table.
Find equivalence classes of the local table.
Choose a representative of each class.
for each 1-pixel of row r **do**
replace its label with the smallest equivalent label.

Step 2. Scan the image bottom-up right-to-left
for each row r of the image **do**
Initialize the local equivalence table for row r.
for each 1-pixel of row r **do**
for each adjacent pixel with a different label **do**
enter the two equivalent labels into the local table.
Find equivalence classes of the local table.
Choose a representative of each class.
for each 1-pixel of row r **do**
replace its label with the smallest equivalent label.

It has been shown by experiments that as the image size increases this algorithm works better than the classical algorithm on a virtual memory computer.

Modifications of the above algorithms involving different partitioning of the image into rectangular blocks are straightforward.

22.3 The Hough Transform

Line detection

The Hough transform is a powerful technique for the detection of lines in an image [11, 16]. Suppose we are given a set of image points — for instance edge points or some other local feature points — and we

want to determine subsets of them lying on straight lines. A straightforward but inefficient procedure to do that consists of determining for each pair of points the straight line through them and then of counting the number of other points lying on it (or close to it).

The Hough transform formulates the problem as follows. Consider the following parametric representation of a line:

$$y = mx + b$$

where the parameters m, b are the slope and the intercept, respectively. Given a point (x_1, y_1), the equation $b = -x_1 m + y_1$, for varying values of b and m, represents the parameters of all possible lines through (x_1, y_1). This equation in the (b, m) plane is the equation of a straight line. Similarly, point (x_2, y_2) maps into the line $b = -x_2 m + y_2$ of the parameter space. These two lines intersect at a point (b', m') that gives the parameters of the line through the (x_1, y_1) and (x_2, y_2). More generally, points aligned in the (x, y) plane along the line with parameters (b', m') correspond to lines in the (b, m) plane intersecting at (b', m'). Figure 22.3 illustrates this property.

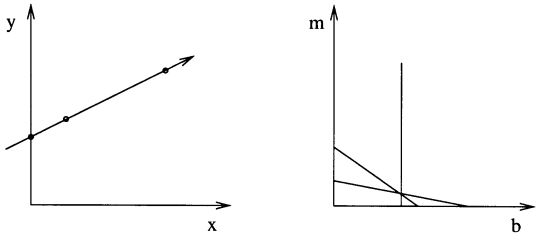


FIGURE 22.3 Mapping collinear image points into parameter space.

Thus the line detection problem is converted into the problem of finding intersections of sets of lines in the parameter space. We next see how to solve this latter problem efficiently in the presence of noise in the image data.

The parameter space is quantized into $h \times k$ cells that form an array A, called *accumulator array*; each entry of the array or cell corresponds to a pair (b, m). In other words, the parameter b is discretized into h values $b_1, b_2, \ldots, b_h$, and m in k values $m_1, m_2, \ldots, m_k$, where b_h and m_k are the largest possible values for the two parameters. See Fig. 22.4.

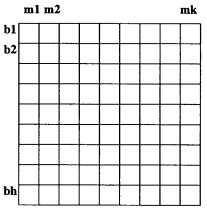


FIGURE 22.4 The parameter space.

The line detection algorithm proceeds in two phases. Phase 1 constructs the accumulator array as follows.

HOUGH ALGORITHM

Initialize all entries of the accumulator array A to zero.
for each image point (x, y) **do**
  for each $m_i, i = 1, k$ **do**

$b \leftarrow -m_i x + y$
round b to the nearest discretized value, say, b_j
$A(b_j, m_i) \leftarrow A(b_j, m_i) + 1$.

At the end of phase 1, the value t at $A(b, m)$ indicates that there are t image points along the line $y = mx + b$. Thus a maximum value in the array corresponds to the best choice of values for the line parameters describing the image data. Phase 2 of the algorithm determines such a peak in the accumulator array A, or more generally, the s largest peaks, for a given s.

Another way of looking at the Hough transform is as a "voting" process where the image points cast votes in the accumulator array.

The above parametric line representation has the drawback that the slope m approaches infinity for vertical lines complicating the construction of the Hough table. Since vertical lines tend to occur frequently in real applications this representation is rarely used. A better representation is

$$\rho = x \cos\theta + y \sin\theta$$

where the two parameters ρ and θ denote the distance of the line from the origin and the angle of the normal line, respectively, as illustrated in Fig. 22.5.

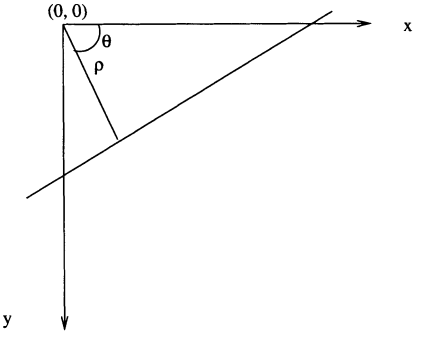


FIGURE 22.5 ρ, θ line representation.

An important property of the Hough transform is its insensitivity to noise and to missing parts of lines.

We now analyze the time complexity of the Hough algorithm. Let n be the number of points (edges or other feature points) and $k \times m$ the size of the accumulator array. Phase 1 of the above algorithm requires $O(nm)$ operations. Phase 2, that is finding the maximum in the array, requires $O(km)$ time for a total of $O(m \max(k, n))$ time.

Detection of Other Parametric Curves

The concept of the Hough transform can be extended in several ways. One extension is to other parametric curves with a reasonably small number of parameters; circles and ellipses are typical examples. Consider the case of circle detection. A circle can be represented by the equation

$$(x - a)^2 + (y - b)^2 = r^2$$

where the parameters a and b are the coordinates of the center and r is the radius. The transform maps the image points into a three-dimensional accumulator array indexed by discretized values of the center coordinates and of the radius. If it is known in advance that only circles with given radii may be present

in an image, which is sometimes true in real applications, only a few two-dimensional subarrays need to be considered. If space is a concern this approach can be used only for few parametric curves, since the array dimensionality grows with the number of the curve parameters.

Another generalization is to arbitrary shapes represented by boundary points [2].

22.4 Model-Based Object Recognition

The problem of object recognition, central to computer vision, is the identification and localization of given objects in a scene. The recognition process generally involves two stages. The first is the extraction from sensed data of information relevant to the problem and the construction of a suitable symbolic representation. The second is that of matching the image and model representations to establish a correspondence between their elements. This correspondence can be used to solve the localization problem, that is the determination of a geometrical transformation that maps the model into the observed object.

Matching is a difficult task, for a number of reasons. First, a brute-force matching is equivalent to a combinatorial search with exponential worst-case complexity. Heuristics may be used to reduce the complexity by pruning the search space that must be explored. Approaches with polynomial time complexity have also been proposed; but the execution time and memory requirements remain high. Second, images do not present perfect data: noise and occlusion greatly complicate the task.

There have been many recognition methods and systems proposed to handle this complexity. Surveys of recognition methods are [1, 3, 4, 8, 34]. The systems differ both in the choice of the representational models and of the matching algorithms used to relate the visual data to the model data. It is obvious that these two aspects influence each other and that the selection of a matching strategy heavily depends on which structures are used to represent data. Some methods use representational models that rely on few high-level features for fast interpretation of the image data. While this facilitates matching, the process of extracting high-level primitives from the raw data may be difficult and time consuming. Moreover, in the presence of occlusion the disappearance of few distinctive features may became fatal. Simpler features are easier to detect and tend to be dense, implying less sensitivity to noise and occlusion. However, low-level features do not have high discriminating power, thus affecting the time for matching.

There are a number of other issues that any recognition system needs to address.

1. *What class of objects is the system able to deal with?*
   The objects can be two-dimensional (2D) or three-dimensional (3D). 3D objects may be smoothly curved or approximated by polyhedra. A restricted class of 3D objects includes flat objects, that is,objects with one dimension much smaller than the other two. Flat objects have received a lot of attention, because a projective transformation applied to this class of objects can be effectively approximated by an **affine transformation,** which is easier to handle. Another important distinction is whether the objects are rigid or deformable or composed of parts that are allowed to move with respect to one another.
2. *What type of geometric transformation is allowed?*
   There is a hierarchy of transformations that has been considered in computer vision: from Euclidean transformations (rotations, translations) to similarity transformations (rotations, translations, and scaling) to affine transformations (represented by a set of linear equations) to the more general projective transformations. To remove the effects of a transformation it is useful to represent the objects by means of features that are invariant under that class of transformations. As the class of transformations becomes more general, the invariant features become more complex and harder to extract. For example, a well-known simple invariant to Euclidean transformations is the distance between two points; however, distance is not invariant to similarity. For a projective transformation four points are needed to define an invariant (cross-ratio).

3. *Robustness*
   Is the recognition system able to deal with real data? Some available systems give good results in controlled environments, with good lighting conditions and with isolated objects. Noisy and cluttered images represent challenging domains for most systems.

This chapter reviews algorithms to match an image against stored object models. Topics that are not covered in this chapter include model representation and organization. The objects are assumed to be given either by means of sets of feature points or by their boundaries (2D or 3D). For an introduction to feature and boundary extraction, see the references listed at the end of this chapter.

The objects may have undergone an affine transformation; thus we do not consider here the more general class of projective transformations.

Matching Sets of Feature Points

In this section we consider recognition of rigid flat objects from an arbitrary viewpoint. Models in the database are represented by sets of feature points. We assume that an image has been preprocessed and feature points have been extracted. Such points might be edge points or corners or correspond to any other relevant feature. The following discussion is independent on the choice of the points and on the method used to acquire them. In fact, as we will see later, similar techniques can be applied when other more complex features, for instance line segments, are used instead of points.

In object recognition to obtain independence from external factors such as viewpoints, it is convenient to represent a shape by means of geometric invariants, that is, shape properties that do not change under a class of transformations. For a flat object a projective transformation can be approximated by an affine transformation. For the definition and properties of affine transformations see Section "Defining Terms."

Affine Matching

There are two main approaches proposed to match objects under affine transformations: *alignment* and *geometric hashing* The first method, *alignment,* uses the hypothesis-and-test paradigm [17]. It computes an affine transformation based on an hypothesized correspondence between an object and model basis and then verifies the hypothesis by transforming the model to image coordinates and determining the fraction of model and image points brought into correspondence. This is taken as a measure of quality of the transformation. The above steps are repeated for all possible groups of three model and image points, since it is known that they uniquely determine an affine transformation.

The geometric hashing or, more generally, indexing methods [20, 21], build at compile time a look-up table that encodes model information in a redundant way. At run time, hypotheses of associations between an observed object and the models can be retrieved from the table and then verified by additional processing. Thus much of the complexity of the task is moved to a preprocessing phase where the work is done on the models alone. Strategies based on indexing do not consider each model separately and are therefore more convenient than search-based techniques in applications involving large databases of models. Crucial to indexing is the choice of image properties to index the table; these can be groups of features or other geometric properties derived from such groups. In the following, we first concentrate on hash methods based on triplets of features points, then on methods based on segments of a boundary object decomposition.

Hypothesize-and-Test (Alignment)

Given an image I containing n feature points, alignment consists of the following steps.

ALIGNMENT
for each model M. Let m be the number of model points
  for each triple of model points **do**
    for each triple of image points **do**
      hypothesize that they are in correspondence and
      compute the affine transformation based on this correspondence.
      for each of the remaining $m - 3$ model points **do**
        apply that transformation.
      Find correspondences between the transformed model points
        and the image points.
      Measure the *quality* of the transformation
      (based on the number of model points that are paired with image points.)

For a given model, these steps are repeated for all triples of model and image features. In the worst-case the number of hypotheses is $O(m^3n^3)$. Thus the total time is $O(T_v m^3 n^3)$, where T_v is the time for the verification of the hypothesized mapping. Since a verification step is likely to take time polynomial in the number of model and image features, this approach takes overall polynomial time which represents a significant improvement over several exponential time approaches to matching proposed in the literature. In the following, a more detailed description of the verification phase is given. Once a transformation is found, it is used to superimpose the model and the object. A distance measure between the two sets of points must be defined so that recognition occurs when such a distance is below a given threshold. One definition of distance is just the number of model and image points that can be brought into correspondence. An approximation to this value can be computed by determining for any transformed model point if an image point can be found in a given small region around it (for instance a small square of pixels of fixed side). In this way an image point can be double-counted if it matches two different model points. Nevertheless, in most cases the above provides a good approximation at a reasonable computational cost.

The search for image points that match each transformed model point can be done sequentially over the image points in time $O(nm)$. Alternatively, it can be made more efficient with some preprocessing of the image points and the use of auxiliary data structures. A practical solution uses a look-up table indexed by the quantized values of the image points coordinates. An entry of the table contains the set of image points located in the small region represented by the cell in the quantized space. Each transformed model point is entered into the table to retrieve, in constant time, possible matching points. Other information besides location of image points can be used to index the table, for instance orientation, which is already available if the features points are computed through an edge operator.

The algorithm above iterates over all the k models in the database leading to a total time complexity $O(kT_v m^3 n^3)$.

Indexing

The second method, *hashing* or *indexing,* is a table look-up method. It consists of representing each model object by storing transformation-invariant information about it in a hash table. This table is compiled off-line. At recognition time, similar invariants are extracted from the sensory data and used to index the table to find possible instances of the model. The indexing mechanism, referred to as *geometric hashing* for point set matching, is based on the following invariant. The coordinates of a point into a reference frame consisting of three noncollinear points are affine invariant. The algorithm consists of a preprocessing phase and a recognition phase.

INDEXING
Preprocessing phase
for each model M. Let m be the number of model points

for each triple of noncollinear model points **do**
form a basis (reference frame)
for each of the $m - 3$ remaining model points **do**
determine the point coordinates in that basis.
Use the triplet of coordinates (after a proper quantization)
as an index to an entry in the hash table, where the pair
(*M, basis*) is stored.

For k models the algorithm has an $O(km^4)$ time complexity. Notice that this process is carried out only once and off-line. At the end of the preprocessing stage an entry of the hash table contains a list of pairs: $(M_{i1}, basis_{i1})$, $(M_{i2}, basis_{i2})$, ..., $(M_{it}, basis_{it})$.

INDEXING
Recognition phase
Initialization
for each entry of the hash table **do**
set a counter to 0.
Choose three noncollinear points of I as a basis.
for each of the $n - 3$ remaining image points **do**
determine its coordinates in that basis.
Use the triplet of coordinates (after a proper quantization)
as an index to an entry in the hash table and
increment the corresponding counter.
Find the pair (*M, basis*) that achieved the maximum value
of the counter when summed over the hash table.

The last process can be seen as one of "voting," with a vote tallied by each image point to all pairs (*M, basis*) that appear at the corresponding entry in the table.

At the end of these steps after all votes have been cast, if the pair (*M, basis*) scores a large number of votes, then there is evidence that the model M is present in the image. If no pair achieves a high vote, then it might be that the selected basis in the image does not correspond to any basis in the model database, and therefore it is convenient to repeat the entire process with another selected basis in the image, until either a match is found or all bases of image points have been explored.

The time complexity of the recognition phase is $O(n)$ if a single selected image basis gives satisfactory results (high scores). Otherwise more bases need to be considered leading, in the worst case, to the time $O(n^4)$. In summary, the time complexity of recognition $T_{\text{recognition}}$ is bound by

$$O(n) \leq T_{\text{recognition}} \leq O\left(n^4\right)$$

There are a few issues that affect the performance of this approach. First, the choice of the basis. The three selected points of the basis should be far away to reduce the numerical error. Other issues are the sensitivity to quantization parameters and to noise. A precise analysis of this approach and a comparison with alignment under uncertainty of sensory data is beyond the scope of this chapter. The interested reader may refer to [15]. Although alignment is computationally more demanding, it is less sensitive to noise than hashing and able to deal with uncertainty under a bounded error noise model. On the other hand, indexing is especially convenient when large databases are involved, since it does not require to match each model separately and its time complexity is therefore not directly dependent on the size of the database.

One way to overcome the limitations of geometric hashing and make it less sensitive to noise is to use more complex global invariants to index the hash table at the expense of preprocessing time to extract such invariants. More complex invariants have a greater discriminating power and generate fewer false positive matches.

Matching Contours of Planar Shapes

It is natural to use contours as a representation for planar objects. Contour information can be in the form of a polygonal approximation, that is, a sequence of line segments of possibly varying lengths approximating the curve. A common boundary description is the eight-direction chain code [39] where all segments of the boundary decomposition have unit length and one of 8 possible directions. Another boundary description is given in terms of concave/convex curve segments. Boundary representations are simple and compact, since a small number of segments suffices to accurately describe most shapes. Techniques to derive boundary descriptions are reviewed in the references listed at the end of this chapter.

Contours have been used in a variety of ways in object recognition to make the process faster and more accurate. First, the availability of contours allows computation of global invariant properties of a shape that can be used for fast removal of candidate objects when searching in the model database; only models found to have similar global properties need be considered for further processing. Global features that can be used include

- The number of segments of a convex/concave boundary decomposition.
- For a close contour, a measure of the area over the length.

A combination of the above features allows a reduction of the candidate objects. An object with extreme values of the global attributes is more easily recognized than an object with average attributes. Obviously there must be a sufficiently high tolerance in the removal process so that good choices are not eliminated.

In the following, we first describe techniques based on dynamic programming that have been developed for contour matching. Then we will see how the methodologies of indexing and alignment based on point sets can take advantage of the contour information in the verification phase for more reliable results. The same methodologies can be further modified so that contour segments become the basic elements of the matching, that is, they are used instead of the feature points to formulate hypotheses.

Dynamic Programming

A number of approaches use dynamic programming to match shape contours. A shape boundary is described by a string of symbols representing boundary segments. For instance, if the segments result from the decomposition of the boundary into convex/concave parts, there might be just three symbols (for convex, concave, and straight) or more if different degrees of convexity or concavity are considered. The matching problem becomes then a string matching problem [39]. We briefly review the string matching algorithm. Then we will discuss adaptations of the basic algorithm to shape recognition that take into account noise and distortion.

Let $A = a_0, \cdots, a_{n-1}$ and $B = b_0, \cdots, b_{m-1}$ be two strings of symbols. Three types of edit operations, namely, *insertion, deletion,* and *change,* are defined to transform A into B.

- insertion: insert a symbol a into a string, denoted as $\lambda \rightarrow a$ where λ is the null symbol;
- deletion: delete a symbol from a string, denoted as $a \rightarrow \lambda$;
- change: change one symbol into another, denoted as $a \rightarrow b$.

A nonnegative real cost function $d(a \rightarrow b)$ is assigned to each edit operation $a \rightarrow b$. The cost of a sequence of edit operations that transforms A into B is given by the sum of the costs of the individual operations. The edit distance $D(A, B)$ is defined as the minimum of such total costs. Let $D(i, j)$ be the

distance between the substrings $a_0, \cdots, a_i$ and $b_0, \cdots, b_j$. It is $D(n, m) = D(A, B)$. Let $D(0, 0) = 0$; then $D(i, j), 0 < i < n, 0 < j < m$ is given by

$$D(i, j) = \min \begin{cases} D(i-1, j) + d(a_i \rightarrow \lambda) \\ D(i, j-1) + d(\lambda \rightarrow b_j) \\ D(i-1, j-1) + d(a_i \rightarrow b_j) \end{cases} \tag{22.1}$$

The matching problem can be seen as one of finding an optimal non-increasing path in the 2D table $D(i, j)$ from the entry $(0, 0)$ to the entry (n, m). If the elements of the table are computed horizontally from each row to the next one, then when computing $D(i, j)$ the values that are needed have already been computed. Since it takes constant time to compute $D(i, j)$, the overall time complexity is given by $O(nm)$. The space complexity is also quadratic.

Using Attributes

A number of variations of the above dynamic programming algorithm have been proposed to adapt it to the shape matching problem. First, attributes are associated to the symbols of the two strings. The choice of the attributes depends both on the type of transformations allowed for the shapes and on the types of boundary segments. Some of the commonly used attributes of concave/convex segments are the normalized length, and the degree of symmetry S_c. Let L_a be the length of segment a and L_A the total length of the segments of A. The normalized length is L_a/L_A. L_A is used as normalization factors to obtain scale invariance.

Let $f(l)$ be the curvature function along segment a.

$$S_a = \int_0^{L_a} \left(\int_0^s f(l)dl - 1/2 \int_0^{L_a} f(l)dl \right) ds$$

If $S_a = 0$ then the segment is symmetric, otherwise it is inclined to the left or to the right depending on whether S_a is positive or negative, respectively.

The attributes are used in the matching process to define the cost $d(i, j)$ of the edit operation that changes segment a_i into b_j. The cost $d(i, j)$ is defined as the weighted sum of the differences between the attributes of a_i and b_j.
Thus the cost function $d(i, j)$ is defined as

$$d(i, j) = w_1 \left| L_{a_i}/L_A - L_{b_j}/L_B \right| + w_2 \sigma \left(S_{a_i}, S_{b_j} \right)$$

where $\sigma(S_a, S_b)$ is taken to be 0 if S_a and S_b are both positive or negative or both close to zero, otherwise is 1. In the above expression w_1 and w_2 are weights used to take into account the different magnitude of the two terms when summed over all edit operations.

Consider now a shape represented by a polygonal approximation. Typical choices for the attributes of a segment a are the normalized length and the angle θ_a that the segment forms with a reference axis. In this case, the cost function can be defined as

$$d(i, j) = w_1 \left| L_{a_i}/L_A - L_{b_j}/L_B \right| + w_2 \gamma \left(\theta_{a_i}, \theta_{b_j} \right)$$

where, again, w_1 and w_2 are weights to make both terms to lie between 0 and 1.

Cyclic Matching

The above algorithm assumes that the correct starting points are known for the correspondences on both shapes. These could be, for instance, the topmost boundary segments when no rotation is present. A more reliable choice that is invariant to Euclidean transformations may be obtained by ranking the

segments according to some criterion, for instance the normalized length or the curvature. The segments on the two curves with highest rank are chosen as the starting points for the dynamic programming algorithm. Alternatively, when the starting points are not available or cannot be computed reliably, cyclic shifts of one of the two strings are needed to try to match segments starting at any point in one of the boundaries. Let B be the shortest string, i.e., $m < n$. To solve the cyclic matching problem a table with n rows and $2m$ columns is built. Paths in the $D(i, j)$ table that correspond to optimal solutions can start at any column $1 \leq j \leq m$ of the first row and end at column $j + m$ of the last row. The dynamic programming algorithm is repeated for each new starting symbol of the shortest string. This brute-force approach solves the cyclic matching problem in time $O(nm^2)$. A more efficient $O(mn \log n)$ solution can be obtained by using divide-and-conquer [24].

Merge Operation

To reduce the effect of noise and distortion and improve matching accuracy, another variant of the above method suggests the use of a fourth operation, called *merge* [37], in addition to the insert, delete, and change operations. Merge allows change of any set of consecutive segments of one string into any set of consecutive segments in the other string. Let $a_{<k,i>}$ denote the sequence of k segments $a_{i-k+1} \cdots a_i$. The merge operation $a_{<k,i>} \rightarrow b_{<h,j>}$ attempts to change the combined segments from the first string into the combined segments of the second. For $k = h = 1$, the merge operation reduces to a change. It has now to be defined what are the attributes associated with the merged segments. First the length of $a_{<k,i>}$ is simply the normalized sum of the lengths of all segments in the sequence. As for the direction, assume $k = 2$. Then the angle $\theta_{a_{<2,i>}}$ can defined as follows:

$$\begin{aligned}
\theta_{a_{2,i}} &= \theta_{a_{i-1}} + \left|\theta_{a_{i-1}} - \theta_{a_i}\right| \\
&\text{if} \quad \theta_{a_{i-1}} < \theta_{a_i} \text{ and } \left|\theta_{a_{i-1}} - \theta_{a_i}\right| \leq 180 \\
&= \theta_{a_i} + \left|\theta_{a_{i-1}} - \theta_{a_i}\right| \\
&\text{if} \quad \theta_{a_{i-1}} \geq \theta_{a_i} \text{ and } \left|\theta_{a_{i-1}} - \theta_{a_i}\right| \leq 180 \\
&= \theta_{a_{i-1}} + \left(360 - \left|\theta_{a_{i-1}} - \theta_{a_i}\right|\right) \\
&\text{if} \quad \theta_{a_{i-1}} \geq \theta_{a_i} \text{ and } \left|\theta_{a_{i-1}} - \theta_{a_i}\right| > 180 \\
&= \theta_{a_i} + \left(360 - \left|\theta_{a_{i-1}} - \theta_{a_i}\right|\right) \\
&\text{if} \quad \theta_{a_{i-1}} < \theta_{a_i} \text{ and } \left|\theta_{a_{i-1}} - \theta_{a_i}\right| > 180
\end{aligned}$$

To include the merge operation in the above algorithm (22.1) has to be replaced by

$$D(i, j) = \min \begin{cases} D(i-1, j) + d\,(a_i \rightarrow \lambda) \\ D(i, j-1) + d\,(\lambda \rightarrow b_j) \\ D_{\text{merge}}(i, j) \end{cases}$$

where

$$D_{\text{merge}}(i, j) = \min_{1 \leq k \leq i, 1 \leq h \leq j} D(i-k, j-h) + d\left(a_{<k,i>}, b_{<h,j>}\right)$$

When there is no limitation on the number of merged segments, the time complexity of the dynamic programming algorithm becomes $O(n^2m^2)$.

In summary, the dynamic programming algorithm is able to deal with rotation, translation, and scaling through the combined use of a distance measure invariant to these transformations and of a cyclic mapping of the two boundaries. Furthermore, noise can be dealt with through the use of concatenated segments in both shapes.

Multiscale Tree Matching

Another approach to match contours of planar shapes uses multiscale data representations. Given a scene containing multiple objects, a full pyramid of images, taken at various levels of resolution, is

first constructed (for instance, a Gaussian pyramid [6]). Then contours are extracted at all levels of resolution and each of them is decomposed into a sequence of segments (either curved or straight line segments) [25, 26].

There are two main ways of matching multiscale objects. One approach, the *coarse-to-fine* matching, tries to find a good initial pair of corresponding segments at the coarsest level of resolution and to expand, at the finer levels, the hypothesized correspondences [29, 30]. The coarse-to-fine approach has generally the disadvantage that mismatches at a coarse scale cause errors from which it is impossible to recover, since the algorithms usually proceed by subdividing the corresponding coarse elements into subelements. This limitation is heavy particularly for highly deformed shapes for which the matching is not very reliable at all levels. On the other hand, this method is fast, because it disregards large portions of the search space. Coarse-to-fine strategies have been successfully used in a variety of image processing and vision applications, including stereo matching, optical flow computation, etc.

Alternative approaches [9, 38] try to overcome the limitations of a coarse-to-fine strategy by processing the multiresolution data starting from the finest level. The shape contours are represented at all scales by sequences of concave/convex segments. In [9], a planar object is modeled as a tree, in which a node corresponds to a multiscale boundary segment and an arc connects nodes at successive levels of resolution. The children of a given node describe the structural changes to that segment at a finer level of resolution. The problem of matching an object against a model is formulated as the one of determining the best mapping between nodes at all levels of the two corresponding trees, according to a given distance between segments. The distance is chosen to be invariant under rotation, translation, and change of scale. The mapping has to satisfy the following constraint: for any path from a leaf node to the root of each of the two trees, there is exactly one node in the mapping. Thus, if a node is in the mapping none of its proper ancestors nor descendants is. Intuitively, such a constraint on the mapped nodes means that for any portion of the boundary at the finest resolution there is one matched segment at some resolution level that covers it.

This method applies to objects for which the entire boundaries are available, implying that it is able to deal with occlusion when this causes alterations in a boundary without breaking it into separate pieces.

We now describe a tree matching algorithm based on dynamic programming that has optimal $O(|T||T'|)$ time complexity, where $|T|$ and $|T'|$ are the number of nodes in the two trees. Let l be the number of leaves of T. $T[i]$ denotes the node of T whose position in the post-order traversal of T is i. Recall that in the post-order traversal the nodes are visited by recursively visiting the first subtree of the root, then the second subtree and so on, and finally visiting the root. In the post-order, $T[1], T[2], \ldots, T[i]$ is in general a forest. $anc(i)$ is the set of ancestor nodes of $T[i]$, including i itself. The postorder number of the father of node $T[i]$ is denoted by $p(i)$. $ll(i)$ and $rl(i)$ denote the postorder number of the leftmost leaf and the rightmost leaf, respectively, of the subtree rooted at $T[i]$; $ll(i) = rl(i) = i$ when $T[i]$ is a leaf node.

A measure of dissimilarity $d(i, j)$ between nodes i and j is defined as above as the weighted sum of the attributes of the segments. In addition $d(i, j)$ must contain a term that rewards matches at higher resolution levels.

The matching problem can be formulated as a minimization problem: find a set of pairs $M = \{(i_h, j_h)\}$ that satisfies the above constraint and minimizes the total cost function, that is,

$$F\left(T, T'\right) = Min_M \sum d\left(i_h, j_h\right)$$

The nodes of the two trees are processed according to the left-to-right post-order numbering. Let $D(i, j)$ be the distance between the two forests $T[1], T[2], \ldots, T[i]$ and $T'[1], T'[2], \ldots, T'[j]$, that is the cost of the best mapping involving nodes up to i and j and covering all leaves $1, \ldots, rl(i)$ and $1, \ldots, rl(j)$. $D(i, j) = F(T, T')$ when $i = |T|$ and $j = |T'|$. Let $D(0, 0) = 0$, $D(0, j) = D(i, 0) = \infty, i, j \neq 0$. $D(i, j)$ is given by the following relations:

- Case 1: if $i \neq p(i-1)$ and $j \neq p(j-1)$

$D(i, j) = D(i-1, j-1) + d(i, j)$;

- Case 2: if $i \neq p(i-1)$ and $j = p(j-1)$
  $D(i, j) = \min\{D(i, j-1), D(i-1, ll(j)-1) + d(i, j)\}$;
- Case 3: if $i = p(i-1)$ and $j \neq p(j-1)$
  $D(i, j) = \min\{D(i-1, j), D(ll(i)-1, j-1) + d(i, j)\}$;
- Case 4: if $i = p(i-1)$ and $j = p(j-1)$
  $D(i, j) = \min\{D(i-1, j-1), D(i-1, j), D(i, j-1), D(ll(i)-1, ll(j)-1) + d(i, j)\}$

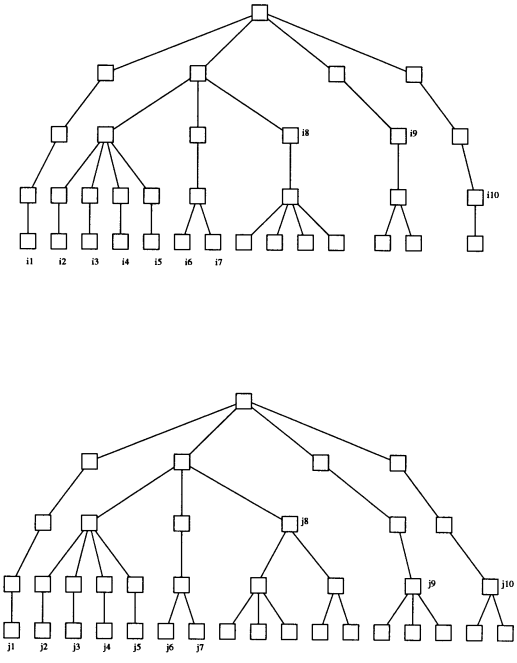


FIGURE 22.6 The matching set $M = \{(i_t, j_t)\}$ of two trees.

The above relations suggest the use of dynamic programming to solve the minimization problem. When computing $D(i, j)$, $1 \leq i \leq |T|$, $1 \leq j \leq |T'|$, all the values on the right side of the recurrence relation above have already been computed. Thus it takes constant time to extend the mapping to nodes i and j leading to a $O(|T||T'|)$ time algorithm. The space complexity is also quadratic.

Hypothesize-and-Test

A common scheme for matching uses three steps: a *hypothesis generation* step during which a few initial matchings are chosen and based on them a transformation is hypothesized, a *prediction* step that determines the matching of image and model features that are compatible with the initial hypothesis, and a *verification* step that evaluates all the resulting possible matchings. This scheme is quite general and can be applied to a variety of data representations. The alignment approach described in the previous section is one such example. The contour information can be effectively used in the alignment method to increase the reliability of the match. As described in the previous section, hypotheses are generated based on small groups of features points brought into correspondence and a geometric transformation is derived for each hypothesis. Then a hierarchical verification process is applied [17]. A preliminary verification phase checks the percentage of model points that lie within reasonable error ranges of the corresponding image

points. This step allows to rapidly eliminate false matches by considering only location and orientation of points (edge points). A more accurate and time-consuming verification procedure based on contour information is applied to the few surviving alignments. Each model segment is mapped through the hypothesized transformation into the image plane and all nearby image segments are determined. Then, for each pair of model and image segments, three cases are considered: (1) the segments are almost parallel and of approximately the same length; (2) the segments are parallel but one is much longer than the other; (3) the segments cross each other. The three cases contribute positive, neutral, or negative evidence to a match, respectively. The basic idea underlying this comparison strategy is that very unlikely two almost coincident segments can be the result of an accidental match. In summary, properties of segments such as length and parallelism can effectively contribute to the evaluation of the quality of a match at the expense of some additional processing.

The next application of the hypothesis-and-test paradigm to contour-based 2D shape descriptions generates hypotheses based on a limited number of corresponding segments. Since the choice of the initial pairing of segments strongly affects the performance of the process, the image segments are first ranked according to a given criterion, for instance length. Then segments in the image and the model are processed in decreasing ranking order of length. Additional constraints can be imposed before a hypothesis is generated. For each pair a and b of such segments, a hypothesis is generated if they are compatible. Compatibility is defined depending on the type of transformation. For similarity transformations a suitable definition can be: segments a and b are compatible (1) if the angle that a forms with its preceding segment along the boundary is close to the angle of b with its neighbor, and (2) assuming the scale factor is known, if the ratio of the lengths of the two segments is close to that value. The verification phase then tries to add more consistent pairings to the initial ones by using an appropriate definition of geometric consistency. The process stops as soon as a reasonably good match is found. We omit here the details of this last phase.

Indexing

The *indexing* methods, as described above, are based on the idea of representing an object by storing invariant information about groups of features in a hash table. Hypotheses of correspondences between an observed object and the models are retrieved from the table by indexing it with groups of features extracted from the object.

One approach to indexing uses information collected locally from the boundary shape in the form of the so-called *super-segment* [33]. A flat object can be approximated by a polygon. Since there exist many polygonal approximations of a boundary with different line fitting tolerances, several of them are used for the purpose of robustness. A super segment is a group of a fixed number of adjacent segments along the boundary, as in Fig. 22.7. Supersegments are the basic elements of the indexing mechanism. The quantized angles between consecutive segments are used to encode each super-segment. Assume that there are n segments in a super segment, then the code is

$$(\alpha_1, \alpha_2, ..., \alpha_{n-1}) \ .$$

Other geometric features of a super segment can be used in the encoding to increase the ability of the system to distinguish between different super segments. One such feature is the eccentricity, that is the ratio of the length of the small and long axis of the ellipse representing the second moment of inertia. A quantized value of the eccentricity is added to the above list of angles. The preprocessing and matching phases of indexing are similar to those described in the section on matching sets of feature points. Briefly, the super-segments of all models are encoded, and each code is used as a key for a table where the corresponding super segment is recorded as an entry. During the matching stage, all encoded super segments extracted from a scene provide indices to the hash table to generate the matching hypotheses.

A *verification phase* is needed to check the consistency of all multiple hypotheses generated by the matching phases. The hypotheses are first divided according to the models they belong to. For a model

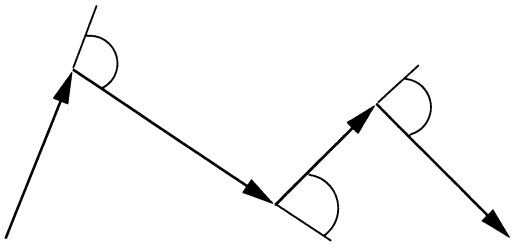


FIGURE 22.7 A super-segment.

M_i, let $h_{i1}, \ldots, h_{it}$ be the hypotheses of associations between super-segments of the image with super-segments of M_i. To check that subsets of such hypotheses are consistent, the following heuristics has been proposed. If three hypotheses are found to be consistent, then the remaining that are found to be consistent with at least one of the three provide an instantiation of the model in the image. Consistency is defined on the basis of few geometric constraints such as the distance, the angle, and the direction. More precisely, the *distances* of corresponding super-segments must be in the same range. The distance between two super-segments is defined as the distance between the locations of the segments taken as the midpoints of the middle segments. Similarly, angles and directions of corresponding super-segments must be in the same range. The orientation of a super-segment is the vector of the predecessor and successor segments of the middle segment.

An alternative approach to indexing suggests the use of information spatially distributed over the object rather than at localized portions of the shape. Groups of points along the curved boundary are collected and indices are computed on the basis of geometrical relationships between the points and local properties at those points. These local measures can include location and tangent or higher order local information as local curve shape.

The difficulty associated with indexing methods arises from the large memory requirements for representing multidimensional tables. Another crucial point is the distribution of data over the table.

Matching Relational Descriptions of Shapes

Graph Matching

An object can be represented by a set of features and their relationships. This representation may take the form of a graph, where nodes correspond to features and arcs represent geometric and topological relationships between features. Similarly, a 3D object can have an object-centered representation consisting of a list of 3D primitives (surface patches, 3D edges, vertices, etc.) and relationships between primitives such as connections between surfaces, edges, etc. Recognition of an object becomes a graph isomorphism problem. Given a graph $G_1 = (V_1, E_1)$ corresponding to a model object and graph $G_2 = (V_2, E_2)$ corresponding to an observed object, the graph isomorphism can be formulated as follows. Find a one-to-one mapping between the vertices of the two graphs $f : V_1 \rightarrow V_2$ such that vertices of V_1 are adjacent iff and only if their corresponding vertices of V_2 are adjacent. In other words, for $u, v \in V_1$, it is $uv \in E_1$ iff $f(u)f(v) \in E_2$. Above we considered only one relation for each graph, but the definition can easily extend to many relations. If an object is only partially visible in an image, a subgraph isomorphism can be used. Graph problems are covered in other chapters of this book. Here we sketch some commonly used heuristics to reduce the complexity of the algorithms for vision applications.

Some researchers [14] have approached the matching problem as one of search, using an *interpretation tree*. A node in the tree represents a pairing between an image feature and a model feature. Nodes at the first level of the tree represent all possible assignments of the first image feature. For each such assignment there a node at the second level for each pairing of the second image feature, and so on.

A path from the root of the tree to a leaf represents a set of consistent pairings that is a solution to

the correspondence problem from which a rigid transformation can be derived. Figure 22.8 shows the interpretation tree for n and m model and object features, respectively. The exponential complexity of the search can be reduced by pruning the interpretation tree using different types of constraints. Mostly geometric constraints involving angles and distances have been considered.

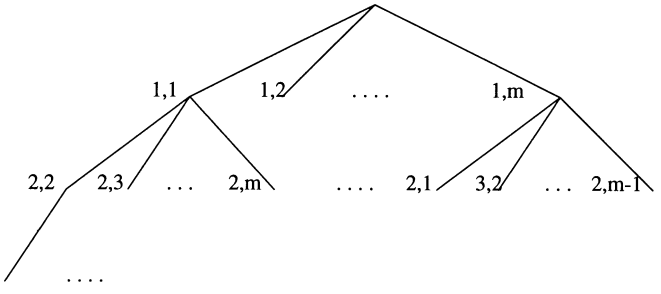


FIGURE 22.8 An interpretation tree with n levels and degree m.

Another approach to cope with the complexity of a brute-force backtracking tree search is the introduction of forward checking and look-ahead functions [31] in the search. Given a partial solution, the idea of forward checking is to find a lower bound on the error of all complete solutions that include the given partial one so that the search can pruned at earlier stages.

22.5 Research Issues and Summary

Most of the work done on image connectivity analysis dates back at least 15 years, and this can be considered a solved problem. Some more recent work has been done on parallel algorithms for image connected component determination for a variety of interconnection networks, including meshes, hypercubes, etc.

The Hough transform is a popular method for line detection, but it is rarely used for more complex parametric curves, due to the high memory requirements. Parallel algorithms for the Hough transform have also been proposed.

Substantial effort has been devoted over the past years to the problem of model-based object recognition, and this is still an area of active research. A number of recognition systems have been designed and built that are successful for limited application domains. The general problem of recognition remains complex, even though polynomial time solutions to matching have been proposed. Practical approaches rely on heuristics based mainly on geometric constraints to reduce the complexity. An interesting issue not covered in this chapter is the effect of noise and spurious elements in the recognition problem. Some recent approaches explicitly take into account a noise error model and formulate the matching problem as one of finding a transformation of model and image data that is consistent with the error model.

Other approaches to match point sets under translation and rotation, developed in the field of computational geometry, are based on the computation of the **Hausdorff distance.**

The problem of finding correspondences is the main component of other important vision tasks, for instance of the *stereo vision problem,* which is the problem of deriving three-dimensional information about an object from different views of the object.

It is also important to observe that some of the matching algorithms used in computer vision can be applied to other research areas, for instance molecular biology. The problem of matching the three-dimensional structure of proteins differs from the point set matching problem mostly in the dimensionality of the data, few tens of feature points for an object and few hundreds of atoms in a protein.

22.6 Defining Terms

Affine transformation: Affine transformations are a subgroup of projective transformations. When the depth of an object is small compared to its distance from the camera, the affine transformation effectively approximates a projective one. For a flat object there is a 2D affine transformation T between two different images of the same object:

$$T : \mathbf{x} \rightarrow A\mathbf{x} + \mathbf{t}$$

where A is 2×2 nonsingular matrix and $\mathbf{t}$ is a 2-vector.
An affine transformation maps parallel lines into parallel lines. Another well-known affine invariant is the following. Given three noncollinear points in the plane, they form a basis or reference frame in which the coordinates of any other point in the plane can be expressed. Such coordinates are affine invariant, meaning that they do not change when the points are transformed under an affine transformation. In other words, let $\mathbf{x_1}, \mathbf{x_2}, \mathbf{x_3}$ be the three noncollinear points, taken as a basis. Let $\mathbf{x}$ be another point and (α, β) its coordinates in the above basis, i.e.,

$$\mathbf{x} = \alpha\,(\mathbf{x_1} - \mathbf{x_3}) + \beta\,(\mathbf{x_2} - \mathbf{x_3}) + \mathbf{x_3}$$

The values α and β are invariant, that is,

$$T(\mathbf{x}) = \alpha\,(T\,(\mathbf{x_1}) - T\,(\mathbf{x_3})) + \beta\,(T\,(\mathbf{x_2}) - T\,(\mathbf{x_3})) + T\,(\mathbf{x_3})$$

The following fact is also known in affine geometry: Given three noncollinear points in the plane $\mathbf{x_1}, \mathbf{x_2}, \mathbf{x_3}$ and three corresponding points $\mathbf{x'_1}, \mathbf{x'_2}, \mathbf{x'_3}$ in the plane, there exists a unique affine transformation T such that $\mathbf{x'_1} = T(\mathbf{x_1})$ $\mathbf{x'_2} = T(\mathbf{x_2})$, and $\mathbf{x'_3} = T(\mathbf{x_3})$.
The affine transformation $T : \mathbf{x} \rightarrow A\mathbf{x} + \mathbf{t}$ can be determined from the set of corresponding points as follows. Let $\mathbf{m}_i$, $i = 1, 2, 3$ and $\mathbf{i}_i$, $i = 1, 2, 3$ be the set of corresponding model and image points, respectively, where

$$\mathbf{m}_i = \begin{pmatrix} m_{i,x} \\ m_{i,y} \end{pmatrix}$$

and

$$\mathbf{i}_i = \begin{pmatrix} i_{i,x} \\ i_{i,y} \end{pmatrix}$$

The six unknown parameters of the 2×2 matrix

$$A = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix}$$

and of the vector

$$\mathbf{t} = \begin{pmatrix} t_1 \\ t_2 \end{pmatrix}$$

can be computed by solving the following system of equations

$$\begin{pmatrix} i_{1,x} & i_{1,y} \\ i_{2,x} & i_{2,y} \\ i_{3,x} & i_{3,y} \end{pmatrix} - \begin{pmatrix} m_{1,x} & m_{1,y} & 1 \\ m_{2,x} & m_{2,y} & 1 \\ m_{3,x} & m_{3,y} & 1 \end{pmatrix} \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \\ t_1 & t_2 \end{pmatrix} = 0\,.$$

Boundary: A closed curve that separates an image component from the background and/or other components.

Connected component: A set C of connected image points. Two points (i, j) and (h, k) are connected if there is a path from (i, j) to (h, k) consisting only of points of C

$$(i, j) = (i_0, j_0), (i_1, j_1), \cdots (i_t, j_t) = (h, k)$$

such that (i_s, j_s) is adjacent to (i_{s+1}, j_{s+1}), $0 \leq s \leq t-1$. Two definitions of adjacency are generally used, 4-adjacency and 8-adjacency. Point (i, j) has four 4-adjacent points, those with coordinates $(i-1, j)$, $(i, j-1)$, $(i, j+1)$, and $(i+1, j+1)$. The 8-adjacent points are, in addition to the four above, the points along the diagonals, namely: $(i-1, j-1)$, $(i-1, j+1)$, $(i+1, j-1)$, and $(i+1, j+1)$.

Digital line: The set of image cells that have a nonempty intersection with a straight-line.

Edge detector: A technique to detect intensity discontinuities. It yields points lying on the boundary between objects and the background. Some of most popular edge detectors are listed. Given an image I, the *gradient operator* is based on the computation of the first-order derivatives, $\delta I/\delta x$ and $\delta I/\delta y$. In a **digital image** there are different ways of approximating such derivatives. The first approach involves only four adjacent pixels.

$$\begin{aligned} \delta I/\delta x &= I(x, y) - I(x-1, y) \\ \delta I/\delta y &= I(x, y) - I(x, y-1) \end{aligned}$$

Alternatively, using 8 adjacent values:

$$\begin{aligned} \delta I/\delta x = & [I(x+1, y-1) + 2I(x+1, y) + I(x+1, y+1)] \\ & - [I(x-1, y-1) + 2I(x-1, y) + I(x-1, y+1)] \end{aligned}$$

The computation of the above expressions can be obtained by applying the following *masks,* known as the *Sobel operators,* to all image points.

-1	-2	-1
0	0	0
1	2	1

-1	0	1
-2	0	2
-1	0	1

The *magnitude* of the gradient, denoted by $G(I(x, y))$, is

$$G(I(x, y)) = \left[(\delta I/\delta x)^2 + (\delta I/\delta y)^2\right]^{1/2}$$

or simply

$$G(I(x, y)) = |\delta I/\delta x| + |\delta I/\delta y|$$

The *Laplacian operator* is a second-order derivative operator defined as

$$L[I(x, y)] = \delta^2 I/\delta x^2 + \delta^2 I/\delta y^2$$

or, in the digital version, as

$$L[I(x, y)] = L[I(x+1, y) + I(x-1, y) + I(x, y+1) + I(x, y-1)] - 4I(x, y)\,.$$

The *zero-crossing operator* determines whether the digital Laplacian has a zero-crossing at a given pixel, that is whether the gradient edge detector has a relative maxima.

Edge point: An image point that is at the border between two image components, that is a point where there is an abrupt change in intensity.

Hausdorff distance: Let $A = \{a_1, a_2, \cdots, a_m\}$ and $B = \{b_1, b_2, \cdots, b_n\}$ two sets of points. The Hausdorff distance between A and B is defined as

$$H(A, B) = \max(h(A, B), h(B, A))) ,$$

where the *one-way Hausdorff distance from A to B* is

$$h(A, B) = \max_{a \in A} \min_{b \in B} d(a, b)$$

and $d(a, b)$ is the distance between two points a and b. The minimum Hausdorff distance is then defined as

$$D(A, B) = \min_{t \in E_2} H(t(A), B)$$

where E_2 is the group of planar motions and $t(A)$ is the transformed of A under motion t. The corresponding *Hausdorff decision problem* for a given ϵ is deciding whether the minimum Hausdorff distance is bounded by ϵ. This last problem is generally solved as a problem of intersection of unions of disks in the *transformation space*.

Image, digital image: A two-dimensional array I of regions or cells each with an assigned integer representing the *intensity value* or *gray level* of the cell. A binary image is an image with only two gray levels: 0 (white), 1 (black).

Polygonal approximation of a curve: A segmentation of a curve into piecewise linear segments that approximates it. One approach to determine it for an open curve is by considering the segment between its endpoints and recursively dividing a segment if the maximum distance between the segment and the curve points is above a given threshold. The segment is divided at the point of maximum distance.

References

[1] Arman, F., Aggarwal, J.K., Model-based object recognition in dense-range images- A review *ACM Computing Surveys,* 25(1), 6–43, 1993.

[2] Ballard, D.H., Generalizing the Hough transform to detect arbitrary shapes. *Pattern Recognition,* 3(2), 11–22, 1981.

[3] Besl, P.J. and Jain, R.C., Three-dimensional object recognition, *Computing Surveys,* 17(1), 75-145, 1985.

[4] Binford, T.O., Survey of model-based image analysis systems. *Inter. J. of Robotics research,* 1(1), 18–64, 1982.

[5] Bruel, T.M., Geometric aspects of visual object recognition. *Ph.D. Dissertation, MIT,* 1992.

[6] Burt, P.S., The Pyramid as a Structure for Efficient Computation. In *Multiresolution Image Processing and Analysis,* Ed., A. Rosenfeld, Springer-Verlag, 6–35, 1984.

[7] Cass, T.A., Polynomial-time geometric matching for object recognition. *Proc. of the European Conf. on Computer Vision,* 1992.

[8] Chin, R.T. and Dyer, C.R., Model-based recognition in robot vision. *ACM Computing Surveys,* 18(1), 66–108, 1986.

[9] Cantoni, V., Cinque, L., Guerra, C., Levialdi, S., and Lombardi, L., Recognizing 2D objects by a multiresolution approach. *12th Int. Conf. on Pattern Recognition,* 310–316, Israel, 1994.

[10] Dasri, R., Costa, L., Geiger, D., and Jacobs, D., Determining the similarity of deformable shapes. *IEEE Workshop on Physics-based Modeling in Computer Vision,* 135–143, 1995.

[11] Duda, R.O. and Hart, P.E., Use of the Hough transformation to detect lines and curves in pictures. *Communications of the ACM,* 15(1), 1972.

[12] Flynn, P.J. and Jain, A.K., 3D object recognition using invariant feature indexing of interpretation tables. *CVIP:Image Understanding,* 55(2), 119–129, 1992.
[13] Gorman, J.R., Mithcell, R., and Kuhl, F., Partial shape recognition using dynamic programming. *IEEE Trans. Pattern Analysis and Machine Intelligence,* 10(2), 257–266, 1988.
[14] Grimson, W.E.L., On the recognition of parameterized 2D objects. *Int. J. of Computer Vision,* 3, 353–372, 1988.
[15] Grimson, W.E., Huttenlocher, D.P., and Jacobs, D., A study of affine matching with bounded sensor error. *Int. J. Comp. Vision,* 13(1), 7–32, 1994.
[16] Hough, P.V., Methods and means to recognize complex patterns. U. S. patent 3.069.654, 1962.
[17] Huttenlocher, D.P. and Ullman, S., Recognizing solid objects by alignment with an image. *Int. J. of Computer Vision,* 5, 195–212, 1990.
[18] Jacobs, D., Optimal matching of planar models in 3D scenes. *IEEE Conf. on Computer Vision and Pattern Recognition,* 269–274, 1991.
[19] Lamdan, Y., Schwartz, J.T., and Wolfson, H.J., On the recognition of 3-D objects from 2-D images. *Proc. of IEEE Int. conf. on Robotics and Application,* 1988.
[20] Lamdan, Y., Schwartz, J.T., and Wolfson, H.J., Affine invariant model-based object recognition. *IEEE Trans. on Robotics and Automation,* 578–589, 1990.
[21] Lamdan, Y. and Wolfson, H., Geometric hashing: a general and efficient model-based recognition scheme. *Proc. Second Int. Conf. on Computer Vision,* 238–249, 1988.
[22] Lowe, D.G., Three-dimensional object recognition from single two-dimensional images. *Artificial Intelligence,* 31, 355–395, 1987.
[23] Lu, S.Y., A tree-to-tree distance and its application to cluster analysis. *IEEE Trans. Pattern Analysis and Machine Intelligence,* 1(1), 219–224, 1971.
[24] Maes, M., On a cyclic string-to-string correction problem. *Information Processing Letters,* 35, 73–78, 1990.
[25] Mokhtarian, F. and Mackworth, A.K., Scale-based descriptions and recognition of planar curves and two dimensional shapes. *IEEE Trans. Pattern Analysis and Machine Intelligence,* 8(1), 34–43, 1986.
[26] Mokhtarian, F., Silhouette-based isolated object recognition through curvature scale space. *IEEE Trans. Pattern Analysis and Machine Intelligence,* 17(5), 539–544, 1995.
[27] Pauwels, E.J., Moons, T., Van Gool, L.J., Kempeners, P., and Oosterlinck, A., Recognition of planar shapes under affine distortion, *Intern. J. of Computer Vision,* 14, 49–65, 1995.
[28] Rosenfeld, A., Ed., *Multiresolution Image Processing and Analysis,* Springer-Verlag, 1984.
[29] Sakou, H., Yoda, H., and Ejiri, M., An algorithm for matching distorted waveforms using a scale-based description. *Proc. IAPR Workshop on Computer Vision,* 329–334, Tokyo, Japan, 1988.
[30] Segen, J., Model learning and recognition of nonrigid objects. *Proc. Computer Vision Pattern Recognition,* 597–602, 1989.
[31] Shapiro, L.G. and Haralick, R.M., Structural descriptions and inexact matching. *IEEE Trans. on Pattern Analysis and Machine Intelligence,* 3, 504–519, 1981.
[32] Shasha, D. and Zhang, K., Fast algorithms for the unit cost editing distance between trees. *J. Algorithms,* 11, 581–621, 1990.
[33] Stein, F. and Medioni, G., Structural hashing: efficient three-dimensional object recognition. *Proc. IEEE Conf. on Computer Vision and Pattern Recognition,* 244–250, 1991.
[34] Suetens, P., Fua, P., and Hanson, A.J., Computational strategies for object recognition. *ACM Computing Surveys,* 24(1), 6–61, 1992.
[35] Tanimoto, S. and Klinger, A., Eds., *Structured Computer Vision,* Academic press, 1980.
[36] Tappert, C., Cursive script recognition by elastic matching. *IBM J. of Res. Develop.,* 26(6), 765–771, 1982.

[37] Tsai, W. and Yu, S., Attributed string matching with merging for shape recognition. *IEEE Trans. Pattern Analysis and Machine Intelligence,* 7(4), 453–462, 1985.
[38] Ueda, N. and Suzuki, S., Learning visual models from shape contours using multi-scale convex/concave structure matching. *IEEE Trans. Pattern Analysis and Machine Intelligence,* 15(4), 337–352, 1993.
[39] Wagner, R.A. and Fischer, M.J., The string-to-string correction problem. *J. Assoc. Comput. Machinery,* 21, 168–173, 1974.

Further Information

A good recent introduction to image processing is presented in *Machine Vision* by R. Jain.

Classical books are: *Computer Vision* by D. H. Ballard, C. M. Brown, and *Digital Picture Processing* by A. Rosenfeld and A. Kak.

A mathematically oriented presentation of the field of computer vision is in *Computer and Robot Vision* by R. M. Haralick and L. G. Shapiro.

Good survey papers on object recognition are [1, 8, 4, 34]. Other papers presenting approaches related to the ones discussed here are [5, 12, 14, 15, 18, 20, 22, 27].

Advances on research on computer vision and image processing are reported in the journals *IEEE Transactions on Pattern Analysis and Machine Intelligence, CVIP: Image Understanding, International Journal on Computer Vision, Pattern Recognition.* This list is by no means exhaustive.

The following is a list of some of the major conference proceedings that publish related work.
IEEE Computer Vision and Pattern Recognition Conference, IAPR International Conference on Pattern Recognition, International Conference on Computer Vision.

23

VLSI Layout Algorithms[1]

Andrea S. LaPaugh
Princeton University

One of the many application areas that has made effective use of algorithm design and analysis is computer-aided design (CAD) of digital circuits. Many aspects of circuit design yield to combinatorial models and the algorithmic techniques discussed in other chapters. In this chapter we focus on one area within the field of computer-aided design: layout of very large scale integrated (VLSI) circuits, which is a particularly good example of the effective use of algorithm design and analysis. We will discuss specific problems in VLSI layout and how algorithmic techniques have been successfully applied. This chapter will not provide a broad survey of CAD techniques for layout, but will highlight a few problems that are important and have particularly nice algorithmic solutions. The reader may find more complete discussions of the field in the references discussed in the *Further Information* section at the end of this chapter.

Integrated circuits are made by arranging active elements (usually transistors) on a planar substrate and interconnecting these elements with conducting wires that are also patterned on the planar substrate [35]. There may be several layers that can be used for wires, but there are restrictions on how wires in different layers can connect, as well as requirements of separations between wires and elements. Therefore, the layout of integrated circuits is usually modeled as a planar embedding problem with several layers in the plane.

VLSI circuits contain hundreds of thousands to millions of transistors. Therefore, it is not feasible to consider the positioning of each transistor separately. Transistors are organized into subcircuits called components; this may be done hierarchically, resulting in several levels of component definition between

[1]Supported in part by the National Science Foundation, FAW award MIP-9023542.

the individual transistors and the complete VLSI circuit. The layout problem for VLSI circuits becomes one of positioning components and their interconnecting wires on a plane, following the design rules, and optimizing some measure such as area or wire length. Within this basic problem structure are a multitude of variations rising from changes in design rules and flexibilities within components as to size, shape, and regions where wires may connect to components. Graph models are used heavily, both to model the components and interconnections themselves and to capture constraint between objects. Geometric aspects of the layout problem must also be modeled. Most layout problems are optimization problems and most are NP-complete. Therefore, heuristics are also employed heavily. Below, we present several of the best known and best understood problems in VLSI layout.

23.1 Background

We will consider a design style known as "general cell." In **general cell layout,** components vary in size and degree of functional complexity. Some components may be from a predesigned component library and have rigidly defined layouts (e.g., a register bank) and others may be full custom designs, in which the building blocks are individual transistors and wires.[2] Components may be defined hierarchically, so that the degree of flexibility in the layout of each component is quite variable. Other design styles, such as standard cell, gate array, and sea-of-gates, are more constrained but share many of the same layout techniques.

The layout problem for a VLSI chip is often decomposed into two stages: placement of components and routing of wires. For this decomposition, the circuit is described as a set of components and a set of interconnections among those components. The components are first placed on the plane based on their size, shape, and interconnectivity. Paths for wires are then found to interconnect specified positions on the components. Thus a placement problem is to position a set of components in a planar region; either the region is bounded, or a measure such as total area of the region used is optimized. The area needed for the yet undetermined routing must be taken into account. A routing problem is, given a collection of sets of points in the plane, to interconnect each sets of points (called a **net**) using paths from an allowable set of paths. The allowable set of paths captures all the constraints on wire routes. In routing problems, the width of wires is abstracted away by representing only the midline of each wire and insuring enough room for the actual wires through the definition of the set of allowable paths.

The above description of the decomposition of a layout problem into placement and routing is meant to be very general. To discuss specific problems and algorithms, we will used a more constrained model. In our model, components will be rectangles. Each component will contain a set of points along its boundary, the **terminals**. Sets of these terminals are the nets, which must be interconnected. A layout consists of a placement of the components in the plane and a set of paths in the plane that do not intersect the components except at terminals and interconnect the terminals as specified by the nets. The paths are composed of segments in various layers of the plane. Further constraints on the set of allowable paths define the routing style, and will be discussed for each style individually. The area of a layout will be the area of the minimum-area rectangle that contains the components and wire paths. (See Fig. 23.1.)

While we still have a fairly general model, we have now restricted our component shapes to be rectangular, our terminals to be single points on component boundaries, and our routing paths to avoid components. (Components are thus assumed to be densely populated with circuitry.) While these assumptions are common and allow us to illustrate important algorithmic results, there is quite a bit of work on nonrectangular components (e.g., [17]), more flexible terminals (see [31]), and "over-the-cell"

[2] We will not discuss special algorithms for laying out transistors, used in tools called "cell generators" or "leaf cell generators" for building components from low-level layout primitives. The reader is referred to [29] as a starting point for an investigation of this topic.

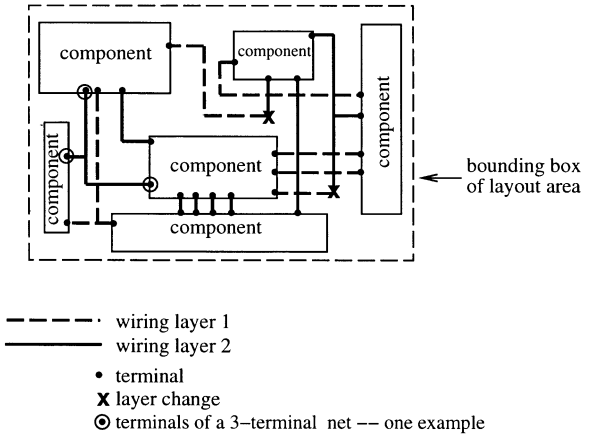


FIGURE 23.1 Example of a layout. This layout is rectilinear and has two layers for wiring.

routing (see [31]). Often, layouts are further constrained to be **rectilinear**. In rectilinear layouts, there is an underlying pair of orthogonal axes defining "horizontal" and "vertical" and the sides of the components are oriented parallel to these axes. The paths of wires are composed of horizontal and vertical segments. In our discussion below, we too will often assume rectilinear layouts.

If a VLSI system is too large to fit on one chip, then it is first partitioned into chip-sized pieces. During partitioning, the goal is to create the fewest chips with the fewest connections between chips. Estimates are used for the amount of space needed by wires to interconnect the components on one chip. The underlying graph problem for this task is **graph partitioning,** which is discussed below.

Closely related to the placement problem is the **floor planning** problem. Floor planning occurs before the designs of components in a general cell design have been completed. The resulting approximate layout is called a **floor plan.** Estimates are used for the size of each component, based on either the functionality of the component or a hierarchical decomposition of the component. Rough positions are determined for the components. These positions can influence the shape and terminal placement within each component as its layout is refined. For hierarchically defined components, one can work bottom up to get rough estimates of size, then top down to get rough estimates of position, then bottom up again to refine positions and sizes.

Once a layout is obtained for a VLSI circuit, either through the use of tools or by hand with a layout editor, there may still be room for improvement. **Compaction** refers to the process of modifying a given layout to remove extra space between features of the layout, space not required by design rules. Humans may introduce such space by virtue of the complexity of the layout task. Tools may place artificial restrictions on layout in order to have tractable models for the main problems of placement and routing. Compaction becomes a postprocessing step to make improvements too difficult to do during placement and routing.

23.2 Placement Techniques

Placement algorithms can be divided into two types: constructive initial placement algorithms and iterative improvement algorithms. A constructive initial placement algorithm has as input a set of components and a set of nets. The algorithm constructs a legal placement with the goal of optimizing some cost function for the layout. Common cost functions measure component area, estimated routing area, estimated total wire length, estimated maximum wire length of a net, or a combination of these. An iterative improvement algorithm has as input the set of components, set of nets, and an initial placement; it modifies the placement, usually repeatedly, to improve a cost function. The initial placement may be a random placement or may be the output of a constructive initial placement algorithm.

Iterative improvement of placements can proceed in a variety of ways. A set of allowable moves, that is, ways in which a placement can be modified, must be identified. These moves should be simple to carry out, including reevaluating the cost function. Typically, a component is rotated or a pair of components are exchanged. An iterative improvement algorithm repeatedly modifies the placement, evaluates the new placement, and decides whether the move should be kept as a starting point for further moves or as a tentative final placement. In the simplest of iterative improvement algorithms, a move is kept only if the placement cost function is improved by it. One more sophisticated paradigm for iterative improvement is simulated annealing (see Chapter 37.) It has been applied successfully to the placement problem, e.g.,[30].

For general cell placement, one of the most widely used initial placement algorithms is a recursive partitioning method (see [19, p. 333]). In this method, a rectangular area for the layout is estimated based on component sizes and connectivity. The algorithm partitions the set of components into two roughly equal-sized subsets such that the number of interconnections between the two subsets is minimized and simultaneously partitions the layout area into two subrectangles of sizes equal to the sizes of the subsets. (See Fig. 23.2.) This partitioning proceeds recursively on the two subsets and subrectangles until each component is in its own subset and the rectangle contains a region for each component.

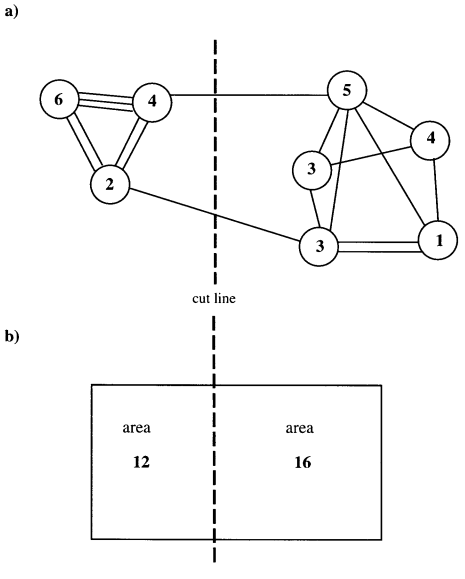


FIGURE 23.2 Partitioning used in placement construction. (a) Partitioning the circuit: Each vertex represents a component; the area of the component is the number inside the vertex. Connections between components are represented by edges. Multiple edges between vertices indicate multiple nets connecting components. (b) Partitioning the layout rectangle proportionally to the partition of component area.

The fundamental problem underlying placement by partitioning is **graph partitioning.**[3] Given a graph $G = (V, E)$, a vertex weight function $w : V \rightarrow N$, an edge cost function $c : E \rightarrow N$, and a balance factor $\beta \epsilon [1/2, 1]$, the graph partitioning problem is to partition V into two subsets V_1 and V_2 such that

$$\sum_{v \epsilon V_1} w(v) \quad \leq \quad \beta \sum_{v \epsilon V} w(v) \tag{23.1}$$

[3]Our definitions follow those in [19].

$$\sum_{v \in V_2} w(v) \quad \leq \quad \beta \sum_{v \in V} w(v) \tag{23.2}$$

and the cost of the partition,

$$\sum_{e \in E \cap (V_1 \times V_2)} c(e) \tag{23.3}$$

is minimized. This problem is NP-complete (see pages 209 and 210 of [7]). Graph partitioning is a well-studied problem. The version we have defined is actually *bipartitioning*. Heuristics for this problem form the core of heuristic algorithms for more general versions of the partition problem where one partitions into more than two vertex sets. The hypergraph version of partitioning, in which each edge is a set of two or more vertices rather than simply a pair of vertices, is a more accurate version for placement, since nets may contain many terminals on many components. But heuristics for the hypergraph version again are based on techniques used for the graph bipartitioning problem.

Among many techniques for graph partitioning, two—the Kernighan–Lin algorithm [13], and simulated annealing—are best known. Both are techniques for iteratively improving a partition. The Kernighan–Lin approach involves exchanging pairs of vertices across the partition. It was improved in the context of layout problems by Fiduccia and Mattheyses [6], who move a single vertex at a time. As applied to graph partitioning, simulated annealing also considers the exchange of vertices across the partition or the movement of a vertex from one side of the partition to the other. The methods of deciding which partitions are altered, which moves are tried, and when to stop the iteration process differentiate the two techniques. Recently, alternatives to iterative improvement have received increased attention for circuit applications, especially as performance-related criteria have been added to the partitioning problem. Examples of these alternatives are spectral methods, based on eigenvectors of the Laplacian, and the use of network flow. The reader is referred to [1] for a more complete discussion of partitioning heuristics.

A second group of algorithms for constructing placements is based on clustering. In this approach, components are selected one at a time and placed in the layout area according to their connectivity to components previously placed. Components are clustered so that highly connected sets of components are close to each other.

When the cost function for a layout involves estimating wire length, several methods can be used. The goal is to define a measure for each net that estimates the length of that net after it is routed. These estimated lengths can then be summed over all nets to get the estimate on the total wire length, or the maximum can be taken over all nets to get maximum net length. Two estimates are commonly used: (1) the half-perimeter of the smallest rectangle containing all terminals of a net; (2) the minimum Euclidean spanning tree of the net. Given a placement, the Euclidean spanning tree of a net is the spanning tree of a graph whose vertices are the terminals of the net, whose edges are all edges between vertices, and whose edge costs are the Euclidean distances in the given placement between the pair of terminals that are endpoints of the edges. Another often-used estimate is the minimum rectilinear spanning tree. This is because rectilinear layouts are common. For a rectilinear layout, the length of the shortest path between a pair of points, (x_a, y_a) and (x_b, y_b), is $|x_a - x_b| + |y_a - y_b|$ (assuming no obstacles). This distance, rather than the Euclidean distance, is used as the distance between terminals. (This is also called the L_1 or Manhattan metric.) A more accurate estimate of the wire length of a net would be the minimum Euclidean (or rectilinear) **Steiner tree** for the net. A Steiner tree for a set of points in the plane is a spanning tree for a superset of the points, i.e., additional points may be introduced to decrease the length of the tree. Finding minimum Steiner trees is NP-hard (see [7]), while finding minimum spanning trees can be done in $O(|E| + |V| \log |V|)$ time for general graphs[4] and in $O(|V| \log |V|)$ time for Euclidean or rectilinear spanning trees (see Chapters 6

[4]Actually, using more sophisticated techniques, finding minimum spanning trees can be done in time almost linear in $|E|$.

and 19). The cost of a minimum spanning tree is an upper bound on the cost of a minimum Steiner tree. For rectilinear Steiner trees, the half-perimeter measure and two thirds the cost of a minimum rectilinear spanning tree are lower bounds on the cost of a Steiner tree [11].

The minimum spanning tree is also useful for estimating routing area. The minimum spanning tree for each net is used as an approximation of the set of paths for the wires of the net. Congested areas of the layout can then be identified and space for routing allocated accordingly.

23.3 Compaction and the Single-Source Shortest Path Problem

Compaction can be done at various levels of design: an entire layout can be compacted at the level of transistors and wires; the layouts of individual components can be compacted; a layout of components can be compacted without changing the layout within components. To simplify our discussion, we will assume that layouts are rectilinear. For compaction, we model a layout as composed entirely of rectangles. These rectangles may represent the most basic geometric building blocks of the circuit: pieces of transistors and segments of wires, or may represent more complex objects such as complex components. We refer to these rectangles as the *features* of the layout. We distinguish two types of features: those that are of fixed size and shape and those that can be stretched or shrunk in one or both dimensions. For example, a wire segment may be able to stretch or shrink in one dimension, representing a lengthening or shortening of the wire, but be fixed in the other dimension, representing a wire of fixed width. We refer to the horizontal dimension of a feature as its *width* and the vertical dimension as its *height.*

Compaction is fundamentally a two-dimensional problem. However, two-dimensional compaction is very difficult. Algorithms based on branch and bound techniques for integer linear programming (see Chapter 32) have been developed, but none is efficient enough to use in practice (see [19], Chapter 6 of [27]). Therefore, the problem is commonly simplify by compacting in each dimension separately: first all features of a layout are pushed together horizontally as much as the design rules will allow, keeping their vertical positions fixed; then all features of a layout are pushed together vertically as much as the design rules will allow, keeping their horizontal positions fixed. The vertical compaction may in fact make possible more horizontal compaction, and so the process may be iterated. This method is not guaranteed to find a minimum area compaction, but, for each dimension, the compaction problem can be solved optimally. We are assuming that we start with a legal layout and that one-dimensional compaction cannot change order relationships in the compaction direction. That is, if two features are intersected by the same horizontal (vertical) line and one feature is to the left of (above) the other, then horizontal (vertical) compaction will maintain this property. The algorithm we present is based on the single-source shortest path algorithm (See Chapter 6). It is an excellent example of a widely used application of this graph algorithm.

The compaction approach we are presenting is called "constraint-based" compaction because it models constraints on and between features explicitly. We shall discuss the algorithm in terms of horizontal compaction, vertical compaction being analogous. We use a graph model in which vertices represent the horizontal positions of features; edges represent constraints between the positions of features. Constraints capture the layout design rules, relationships between features such as connectivity, and possibly other desirable constraints such as performance-related constraints. Design rules are of two types: feature-size rules and separation rules. Feature-size rules give exact sizes or minimum dimensions of features. For example, each type of wire has a minimum width; each transistor in a layout is of a fixed size. Separation rules require that certain features of a layout be at least a minimum distance apart to avoid electrical interaction or problems during fabrication. Connectivity constraints occur when a wire segment is allowed to connect to a component (or another wire segment) anywhere in a given interval along the

component boundary. Performance requirements may dictate that certain elements are not too far apart. A detailed discussion of the issues in the extraction of constraints from a layout can be found in Chapter 6 of [27].

In the simplest case, we start with a legal layout and only consider feature-size rules and separation rules. We assume all wire segments connect at fixed positions on component boundaries and there are no performance constraints. Furthermore, we assume all features that have a variable width attach at their left and right edges to features with fixed width, e.g., a wire segment stretched between two components. Then, we need only represent features with fixed width; we can use one variable for each feature. In this case, any constraints on the width of a variable-width feature are translated into constraints on the positions of the fixed-width features attached to either end (see Fig. 23.3). We are left with only separation

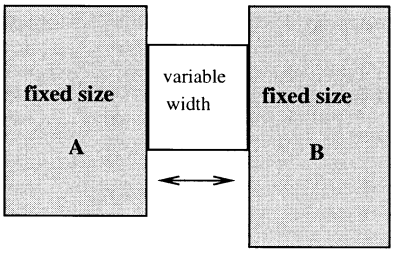


FIGURE 23.3 Generating separation constraints. The constraint on the separation between features A and B is the larger of the minimum separation between them and the minimum width of the flexible feature connecting them.

constraints, which are of the form

$$x_B \geq x_A + d_{\text{min}} \tag{23.4}$$

or equivalently

$$x_B - x_A \geq d_{\text{min}} \tag{23.5}$$

where B is a feature to the right of A, x_A is the horizontal position of A, x_B is the horizontal position of B, and the minimum separation between x_A and x_B is d_{min}. In our graph model, there is an edge from the vertex for feature A to the vertex for feature B with length d_{min}. We add a single extra source vertex to the graph and a 0-length edge from this source vertex to every other vertex in the graph. This source vertex represents the left edge of the layout. Then finding the longest path from this source vertex to every vertex in the graph will give the leftmost legal position of each feature—as if we had pushed each feature as far to the left as possible. Finding the longest path is converted to a single-source shortest path problem by negating all the lengths on edges. This is equivalent to rewriting the constraint as

$$x_A - x_B \leq -d_{\text{min}} \, . \tag{23.6}$$

From now on, we will write constraints in this form. Note that this graph is acyclic. Therefore, as explained below, the single-source shortest path problem can be solved in time $O(n + |E|)$ by a **topological sort,** where n is the number of features and E is the set of edges in the constraint graph.

A topological sorting algorithm is an algorithm for visiting the vertices of a directed acyclic graph (DAG). The edges of a DAG induce a partial order on the vertices: $v < u$ if there is a (directed) path from v to u in the graph. A topological order is any total ordering of the vertices that is consistent with this partial order. A topological sorting algorithm visits the vertices in some topological order. In Chapter 6, a topological sorting algorithm based on depth-first search is presented. For our purposes, a modified breadth-first search approach is more convenient. In this modification, we visit a node as soon as all its immediate predecessors have been visited (rather than as soon as any single immediate predecessor has been visited). For a graph $G = (V, E)$ this algorithm can be expressed as follows:

```
TOPOLOGICAL SORT (G)

1  S ← all vertices with no incoming edges (sources)
2  U ← V
3  while S is not empty
4     do choose any vertex v from S
5        VISIT v
6        for each vertex u such that (v, u)∈E
7           do E ← E − {(v, u)}
8              if u is now a source
9                 then S ← S ∪ {u}
10       U ← U − {v}
11       S ← S − {v}
12 if U is not empty
13    then error ▷ G is not acyclic
```

In our single-source shortest path problem, we start with only one source, s, the vertex representing the left edge of the layout. We compute the length of the shortest path from s to each vertex v, denoted $\ell(v)$. We initialize $\ell(v)$ before line 3 to be 0 for $\ell(s)$ and ∞ for all other vertices. Then for each vertex v we select at line 4, and each edge (v, u) we delete at line 7 we update for all shortest paths that go through v by $\ell(u) \leftarrow \min\{\ell(u), \ell(v) + length(v, u)\}$. When the topological sort has completed, $\ell(v)$ will contain the length of the shortest path from s to v (unless G was not acyclic to begin with). The algorithm takes $O(|V| + |E|)$ time.

In our simplest case, all our constraints were minimum separation constraints. In the general case, we may have maximum separation constraints as well. These occur when connectivity constraints are used and also when performance constraints that limit the length of wires are used. Then we have constraints of the form

$$x_B \leq x_A + d_{\max} \tag{23.7}$$

or equivalently

$$x_B - x_A \leq d_{\max}\,. \tag{23.8}$$

Such a constraint is modeled as an edge from the vertex for feature B to the vertex for feature A with weight $d_{\max}$. For example, to model a horizontal wire W that has an interval from l to r along which it can connect to a component C (see Fig. 23.4), we use the pair of constraints

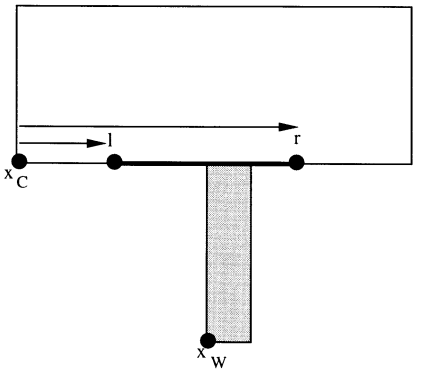


FIGURE 23.4 Modeling a connection that can be made along an interval of a component boundary.

$$x_C - x_W \leq -l \tag{23.9}$$

and

$$x_W - x_C \leq r\ . \tag{23.10}$$

Once we allow both minimum and maximum separation constraints, we have a much more general linear constraint system. All constraints are still of the form

$$x - y \leq d\ , \tag{23.11}$$

but the resulting constraint graph need not be acyclic. (Note that equality constraints $y - x = d$ may be expressed as $y - x \leq d$ and $x - y \leq -d$. Equality constraints may also be handled in a preprocessing step that merges vertices that are related by equality constraints.) To solve the single-source shortest path algorithm, we now need the $O(|V||E|)$-time Bellman–Ford algorithm (see [3]). This algorithm only works if the graph contains no negative cycle. If the constraint graph is derived from a layout that satisfies all the constraints, this will be true, since a negative cycle represents an infeasible set of constraints. However, if the initial layout does not satisfy all the constraints, for example, the designer adds constraints for performance to a rough layout, then the graph may be cyclic. The Bellman–Ford algorithm can detect this condition, but since the set of constraints is infeasible, no layout can be produced.

If the constraint graph is derived from a layout that satisfies all the constraints, an observation by Maley [21] allows us to use Dijkstra's algorithm to compute the shortest paths more efficiently. To use Dijkstra's algorithm, the weights on all the edges must be positive (see Chapter 6). Maley observed that when an initial layout exists, the initial positions of the features can be used to convert all lengths to positive lengths as follows. Let p_A and p_B be initial positions of features A and B. The constraint graph is modified so that the length of an edge (v_A, v_B) from the vertex for A to the vertex for B becomes $length(v_A, v_B) + p_B - p_A$. Since the initial layout satisfies the constraint $x_A - x_B \leq length(v_A, v_B)$, we have $p_B - p_A \geq -length(v_A, v_B)$ and $p_B - p_A + length(v_A, v_B) \geq 0$. Maley shows that this transformation of the edge lengths preserves the shortest paths. Since all edge weights have been converted to positive lengths, Dijkstra's algorithm can be used, giving a running time of $O(|V| \log |V| + |E|)$ or $O(|E| \log |V|)$, depending on the implementation used.[5]

Even when the constraint graph is not acyclic and an initial feasible layout is not available, restrictions on the type or structure of constraints can be use to get faster algorithms. For example, Lengauer and Mehlhorn give an $O(|V| + |E|)$-time algorithm when the constraint graph has a special structure called a "chain DAG" that is found when the only constraints other than minimum separation constraints are those coming from flexible connections such as those modeled by Eqs. (23.9) and (23.10) above (see [19, p. 614]). Liao and Wong [20] and Mata [23][6] present $O(|E_x| \times |E|)$-time algorithms, where E_x is the set of edges derived from constraints other than the minimum-separation constraints. These algorithms are based on the observation that $E - E_x$ is a directed acyclic graph (as in our simple case above). Topological sort is used as a subroutine to solve the single-source shortest path problem with edges $E - E_x$. The solution to this problem may violate constraints represented by E_x. Therefore, after finding the shortest paths for $E - E_x$, positions are modified in an attempt to satisfy the other constraints (represented by E_x), and the single-source shortest path algorithm for $E - E_x$ is run again. This technique is iterated until it converges to a solution for the entire set of constraints or the set of constraints is shown to be infeasible,

[5]The $O(|V| \log |V| + |E|)$ running time depends on using Fibonacci heaps for a priority queue. If the simpler binary heap is used, the running time is $O(|E| \log |V|)$. This comment also holds for finding minimum spanning trees using Prim's algorithm. See Chapter 6 for a discussion of the running times of Dijkstra's algorithm and Prim's algorithm.

[6]The technique used by Mata is the same as that by Liao and Wong, but Mata has a different stopping condition for his search, which can lead to more efficient execution.

which is proven to be within $|E_x|$ iterations. If $|E_x|$ is small, this algorithm is more efficient than using Bellman–Ford.

The single-dimensional compaction that we have discussed ignores many practical issues. One major issue is the introduction of bends in wires. The fact that a straight wire segment connects two components may be an artifact of the layout, but it puts the components in lock-step during compaction. Adding a bend to the wire would allow the components to move independently, stretching the bend accordingly. Although the bend may require extra area, the overall area might improve through compaction with the components moving separately.

Another issue is allowing components to change their order from left to right or top to bottom. This change might allow for a smaller layout, but the compaction problem becomes much more difficult. In fact, a definition of one-dimensional compaction that allows for such exchanges is NP-complete (see [19, p. 587]). Practically speaking, such exchanges may cause problems for wire routing. The compaction problem we have presented requires that the topological relationships between the layout features remain unchanged while space is compressed.

23.4 Floor Plan Sizing and Classic Divide and Conquer

The problem we will now consider, called **floor plan sizing,** is one encountered during certain styles of placement or floor planning. With some reasonable assumptions about the form of the layout, the problem can be solved optimally by a polynomial-time algorithm that is an example of classic divide and conquer.

Floor plan sizing occurs when a floor plan is initially specified as a partitioning of a rectangular layout area, representing the chip, into subrectangles, representing components [see Fig. 23.5(a)]. Each subrectangle corresponds to some component. We assume that the rectangular partitioning is rectilinear. For this discussion, given a set of components C, by "a floor plan for C," we shall mean such a rectangular partition of a rectangle into $|C|$ subrectangles, and a one-to-one mapping from C to the subrectangles. This partition indicates the relative position of components, but the subrectangles are constrained only to have approximately the same area as the components (possibly with some bloating to account for routing), not to have the same aspect ratio as the components. When the actual components are placed in the locations, the layout will change [see Fig. 23.5(b)]. Furthermore, it may be possible to orient each component so that the longer dimension may be either horizontal or vertical, affecting the ultimate size of the layout. In fact, if the component layouts have not been completed, the components may be able to assume many shapes while satisfying an area bound that is represented by the corresponding subrectangle. We will formalize this through the use of a **shape function.**

Definition: A **shape function** for a component is a mapping $s : [w_{min}, \infty] \rightarrow [h_{min}, \infty]$ such that s is monotonically decreasing, where $[w_{min}, \infty]$ and $[h_{min}, \infty]$ are intervals of $\Re^+$.

The interpretation of $s(w)$ is that it is the minimum height (vertical dimension) of any rectangle of width w that contains a layout of the component. w_{min} is the minimum width of any rectangle that contains a layout and h_{min} is the minimum height. The monotonicity requirement represents the fact that if there is a layout for a component that fits in a $w \times s(w)$ rectangle, it certainly must fit in a $(w+d) \times s(w)$ rectangle for any $d \geq 0$; therefore, $s(w+d) \leq s(w)$. In this discussion we will restrict ourselves to *piecewise linear* shape functions.

Given an actual shape (width and height) for each component, determining the minimum width and minimum height of the rectangular layout area becomes a simple compaction problem as discussed above. Each dimension is done separately, and two constraint graphs are build. We will discuss the horizontal constraint graph; the vertical constraint graph is analogous. The reader should refer to Fig. 23.5(c). The horizontal constraint graph has a vertex for each vertical side of the layout rectangle and each vertical side of a subrectangle (representing a component) in the rectangular partition. There is a directed edge from each vertex representing the left side of a subrectangle to each vertex representing the right side of a subrectangle; this edge has a length that is the width of the corresponding component. There are also

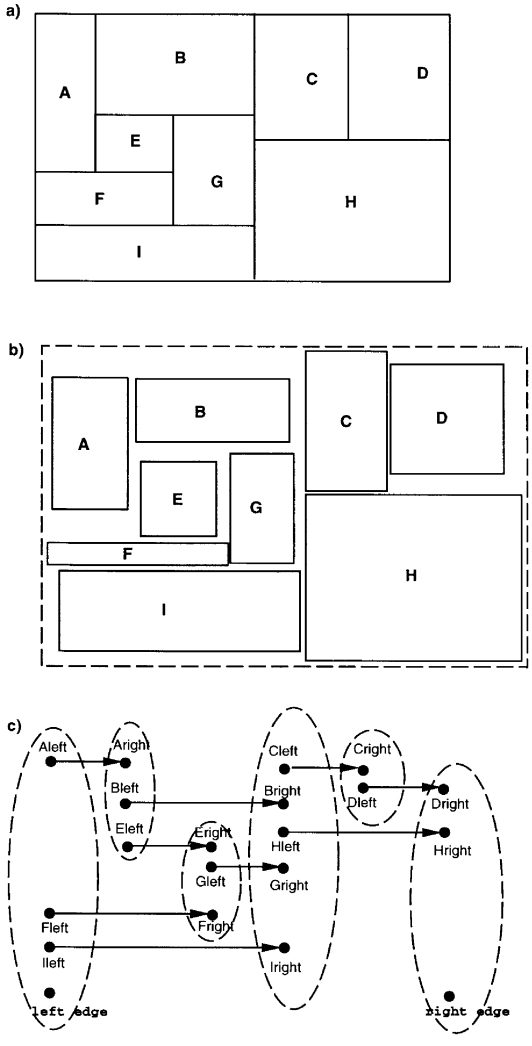


FIGURE 23.5 A floor plan and the derived layout. (a) A partition of a rectangle representing a floor plan. (b) A layout with actual components corresponding to the floor plan. (c) The horizontal constraint graph for the layout. Bidirectional 0-length edges are not shown. Instead, vertices constrained to have the same horizontal position due to chains of 0-length edges are shown in the same oval. They are treated as a single vertex.

two directed edges (one in each direction) between the vertices representing any two overlapping sides of the layout rectangle or subrectangles; the length of these edges is 0. Thus the vertex representing the left side of the layout rectangle has 0-length edges between it and the vertices representing the left sides of the leftmost subrectangles. Note that these constraints force two subrectangles that do not touch but are related through a chain of 0-length edges between one's left side and the other's right side to lie on opposite sides of a vertical line in the layout [e.g., components B and H in Fig. 23.5(a).]. This is an added restriction to the layout, but an important one for the correctness of the algorithm presented below for the sizing of **slicing floor plans.**

Given an actual width for each component and having constructed the horizontal constraint graph as described in the preceding paragraph, to determine the minimum width of the rectangular layout area, one simply finds the longest path from the vertex representing the left side of the layout rectangle to the

vertex representing the right side of the layout rectangle. To simplify the problem to one in an acyclic graph, vertices connected by pairs of 0-length edges can be collapsed into a single vertex; only the longest edge between each pair of (collapsed) vertices is needed. Then topological sort can be used to find the longest path between the left side and the right side of the floor plan, as discussed in the preceding section of this chapter.

We now have the machinery to state the problem of interest:

Floor plan sizing: Given a set C of components, a piecewise linear shape function for each component, and a floor plan for C, find an assignment of specific shapes to the components so that the area of the layout is minimized.

Stockmeyer [33] showed that for general floor plans, the floor plan sizing problem is NP-complete. This holds even if the components are of fixed shape, but can be rotated 90°. In this case, the shape function of each component is a step function with at most two steps: $s(x) = d_2$ for $d_1 \leq x < d_2$ and $s(x) = d_1$ for $d_2 \leq x$, where the dimensions of the component are d_1 and d_2 with $d_1 \leq d_2$. However, for floor plans of a special form, called **slicing floor plans,** Stockmeyer gave a polynomial-time algorithm for the floor plan sizing problem when components are of fixed shape but can rotate. Otten [24] generalized this result to any piecewise-linear shape function. A slicing floor plan is one in which the partition of the layout rectangle can be constructed by a recursive cutting of a rectangular region into two subregions using either a vertical or horizontal line segment (see Fig. 23.6). The rectangular regions that are not further partitioned are the subrectangles corresponding to components. The recursive partitioning method of constructing a placement discussed above produces a slicing floor plan.

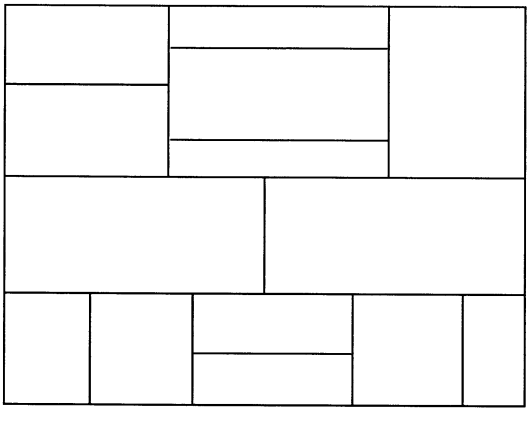


FIGURE 23.6 A slicing floor plan.

A slicing floor plan can be represented by a binary tree. The root of the tree represents the entire layout rectangle and is labeled with the direction of the first cut. Other interior vertices represent rectangular subregions that are to be further subdivided, and the label on any such vertex is the direction of the cut used to subdivide it. The two children of any vertex are the rectangular regions resulting from cutting the rectangle represented by the vertex. The leaves of the binary tree represent the rectangles corresponding to components.

The algorithm for the sizing of a slicing floor plan uses the binary tree representation in a fundamental way. The key observation is that one only needs the shape functions of the two subregions represented by the children of a vertex to determine the shape function of a vertex. If the shape functions can be represented succinctly and combined efficiently for each vertex, the shape function of the root can be determined efficiently. We will present the combining step for shape functions that are step functions (i.e., piecewise constant) following the description in [19], since it illustrates the technique but is a straightforward

calculation. Otten [24] shows how to combine piecewise linear slicing functions, but Lengauer [19] comments that arithmetic precision can become an issue in this case.

We shall represent a shape function that is a step function by a list of pairs $(w_i, s(w_i))$ for $0 \leq i \leq b_s$ and $w_0 = w_{min}$. The interpretation is that for all x, $w_i \leq x < w_{i+1}$ (with $w_{b_s+1} = \infty$), $s(x) = s(w_i)$. Parameter b_s is the number of *breaks* in the step function. The representation of the function is linear in the number of breaks. (This represents a step function whose constant intervals are left closed and right open and is the most logical form of step function for shape functions. However, other forms of step functions also can be represented in size linear in the number of breaks.)

Given step functions, s_l and s_r, for the shapes of two children of a vertex, the shape function, s, for the vertex will also be a step function. When the direction of the cut is horizontal, the shape functions can simply be added, that is, $s(x) = s_l(x) + s_r(x)$. w_{min} for s is the maximum of $w_{min,l}$ for s_l and $w_{min,r}$ for s_r. Each subsequent break point for s_l or s_r is a break point for s, so that $b_s \leq b_{s_l} + b_{s_r}$. Combining the shape functions takes time $O(b_{s_l} + b_{s_r})$. When the direction is vertical, the step functions must first be inverted, the inverted functions combined, and then the combined function inverted back. The inversion of a step function s is straightforward and can be done in $O(b_s)$ time.

To compute the shape function for the root of a slicing floor plan, one simply does a postorder traversal of the binary tree (see Chapter 6), computing the shape function for each vertex from the shape functions for the children of the vertices. The number of breaks in the shape function for any vertex is no more than the number of breaks in the shape functions for the leaves of the subtree rooted at that vertex. Let b be the maximum number of breaks in any shape function of a component. Then the running time of this algorithm for a slicing floor plan with n components is

$$\begin{aligned} T(n) &\leq T(n_l) + T(n_r) + bn & (23.12)\\ &\leq dbn\ , & (23.13) \end{aligned}$$

where d is the depth of the tree (the length of the longest path from the root to a leaf). We have the following:

> Given an instance of the floor plan sizing problem that has a slicing floor plan and step functions as shape functions for the components, there is an $O(dbn)$-time algorithm to compute the shape function of the layout rectangle.

Given the shape function for the layout rectangle, the minimum area shape can be found by computing the area at each break in time linear in the number of breaks, which is at most $O(bn)$.

Recently, Shi [32] presented a modification to this technique that improves the running time for imbalanced slicing trees. For general (nonslicing) floor plans, many heuristics have been proposed: for example, Pan and Liu [25] present a generalization of the slicing floor plan technique to general floor plans.

23.5 Routing Problems

We shall only discuss the most common routing model for general cell placement—the rectilinear **channel routing** model. In this model, the layout is rectilinear. The regions of the layout that are not covered by components are partitioned into nonoverlapping rectangles, called **channels.** The allowed partitions are restricted so that each channel only has components touching its horizontal edges (a horizontal channel) or its vertical edges (a vertical channel). These edges will be referred to as the "top" and "bottom" of the channel, regardless of whether the channel is horizontal or vertical. The orthogonal edges, referred to as the "left edge" and "right edge" of the channel, can only touch another channel. These channels compose the area for the wire routing. There are several strategies for partitioning the routing area into channels, i.e., "defining" the channels, but most use maximal rectangles where possible (i.e., no two channels can be merged to form a larger channel). The layout area becomes a rectangle that is partitioned into subrectangles of two types: components and channels. (See Fig. 23.7.)

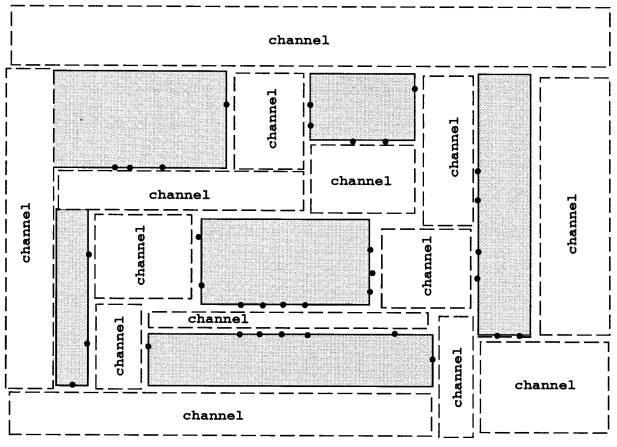


FIGURE 23.7 The decomposition of a layout into routing channels and components.

Given a layout with channels defined, the routing problem can be decomposed into two subproblems: **global routing** and local or detailed routing. **Global routing** is the problem of choosing which channels will be used to make the interconnections for each net. Actual paths are not produced. By doing global routing first, one can determine the actual paths for wires in each channel separately. The problem of detailed routing is to determine these actual paths and is more commonly referred to as "**channel routing.**" Of course, the segments of a wire path in each channel must join at the edges of the channel to produce an actual interconnection of terminals. To understand the approaches for handling this interfacing, we must have a more detailed definition of the channel routing problem, which we give next.

The channel routing problem is defined so that there are initial positional constraints in only one dimension. Recall that we define channels to abut components on only two parallel sides, the "top" and "bottom." This is so that the routes in the channel will only be constrained by terminal positions on two sides. The standard channel routing problem has the following input: a rectangle (the channel) containing points (terminals) at fixed positions along its top and bottom edges, a set of nets that must be routed in the channel, and an assignment of each of the terminals on the channel to one of these nets. Also, two (possibly empty) subsets of nets are specified: one containing nets whose routing must interface with the left edge of the channel and one containing nets whose routing must interface with the right edge of the channel. The positions at which wires must intersect the left and right edges of the channel are not specified. The dimension of the channel from the left edge to the right edge is called the *length* of the channel; the dimension from the top edge to the bottom edge is the *width* of the channel (see Fig. 23.8). Since there are no terminals on the left and right edges, the width is often taken to be variable, and the goal is to find routing paths achieving the connections specified by the nets and minimizing the width of the channel. In this case, the space needed for the wires determines the width of the channel. The length of the channel is more often viewed as fixed, although there are channel routing models in which routes are allowed to cross outside the left and right edges of the channel.

We can now discuss the problem of interfacing routes at channel edges. There are two main approaches to handling the interfacing of channels. One is to define the channels so that all adjacent channels form '$\top$'s (no '+'s). Then if the channel routing is done for the base of the '$\top$' first, the positions of paths leaving the base of the '$\top$' and entering the cross piece of the '$\top$' are fixed by the routing of the base and can be treated as terminal positions for the cross piece of the '$\top$'. Using this approach places constraints on the routing order of the channels. We can model these constraints using a directed graph: there is a vertex, v_C, for each channel C and an edge from v_A to v_B if channel A and channel B abut with channel A as the base of the '$\top$' and channel B as the cross piece of the '$\top$'. The edge from v_A to v_B represents that channel A must be routed before channel B. This graph must be acyclic for the set of constraints on the order of routing to be feasible. If the graph is not acyclic, another method must be used to deal with some

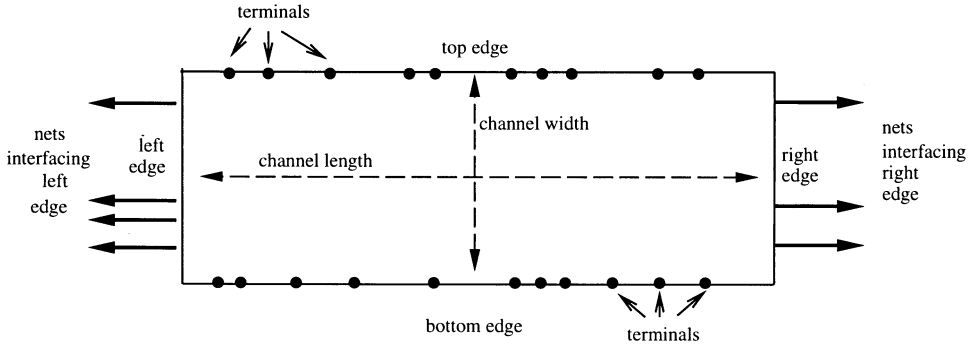


FIGURE 23.8 A channel. Nets interfacing at the left and right edges are not in a given order.

of the channel interfaces. Slicing floor plans are very good for this approach because if each slice is defined to be a channel, then the channel order constraint graph will be acyclic.

The second alternative for handling the interfaces of channels is to define a special kind of channel, called a **switch box,** that is constrained by terminal locations on all four sides. In this alternative, some or all of the standard channels abut switch boxes. Special algorithms are used to do the switch box routing, since, practically speaking, this is a more difficult problem than standard channel routing.

23.6 Global Routing

Since global routing need not produce actual paths for wires, the channels can be modeled by a graph, called the *channel intersection graph.* For each channel, project the terminals lying on the top and bottom of the channel onto the midline of the channel. Each channel can be divided into segments by the positions of these projected terminals. The channel intersection graph is an undirected graph with one vertex for each projected terminal and one vertex for each intersection of channels. There is one edge for each segment of a channel, which connects the pair of vertices representing the terminals and/or channel intersections bounding the segment. A length and a capacity can be assigned to each edge, representing, respectively, the length between the ends of the channel segment and the width of the channel. Different versions of the global routing problem use one or both of the length and capacity parameters.

Given the channel intersection graph, the problem of finding a global routing for a set of nets becomes the problem of finding a **Steiner tree** in the channel intersection graph for the terminals of each net. Earlier in this chapter, we defined Steiner trees for points in the plane. For a general graph $G = (V, E)$, a Steiner tree for a subset of vertices $U \subset V$ is a set of edges of G that form a tree in G and whose set of endpoints contains the set U. Various versions of the global routing problem are produced by the constraints and optimization criteria used. For example, one can simply ask for the minimum length Steiner tree for each net, or one can ask for a set of Steiner trees that does not violate capacity constraints and has total minimum length. For each edge, the capacity used by a set of Steiner trees is the number of Steiner trees containing that edge; this must be no greater than the capacity of the edge. Another choice is not to have constraints on the capacity of edges but to minimize the maximum capacity used for any edge. In general, any combination of constraints and cost functions on length and capacity can be used. However, regardless of the criteria, the global routing problem is invariably NP-complete. A more detailed discussion of variations of the problem and their complexity can be found in [19]. There, a number of sophisticated algorithms for Steiner tree problems are also discussed. Here we will only discuss two techniques based on basic graph algorithms: breadth-first search and Dijkstra's single-source shortest path algorithm.

The minimum Steiner tree problem is itself NP-complete (see pages 208-209 in [7]). Therefore, one approach to global routing is to avoid finding Steiner trees by breaking up each net into a collection of point-to-point connections. One way to do this is to find the minimum Euclidean or rectilinear spanning

tree for the terminals belonging to each net (ignoring the channel structure), and use the edges of this tree to define the point-to-point connections. Then one can use Dijkstra's single-source shortest path algorithm on the channel intersection graph to find a shortest path for each point-to-point connection. Paths for connections of the same net that share edges can then be merged, yielding a Steiner tree. If there are no capacity constraints on edges, the quality of this solution is only limited by the quality of the approximation of a minimum Steiner tree by the chosen collection of point-to-point paths. If there are capacity constraints, then after solving each shortest path problem, one must remove from the channel intersection graph the edges whose used capacity already equals the edge capacity. In this case, the order in which nets and terminals within a net are routed is significant. Heuristics are used to choose this order. One can better approximate Steiner trees for the nets by, at each iteration for connections within one net, choosing a terminal not yet connected to any other terminals in the net as the source of Dijkstra's algorithm. Since this algorithm computes the shortest path to every other vertex in the graph from the source, the shortest path which connects to any other vertex in the channel intersection graph that is already on a path connecting terminals of the net can be used. Of course there are variations on this idea.

For any graph, breadth-first search from a vertex v will find a shortest path from v to every other vertex in the graph when the length of each edge is 1. Breadth-first search takes time $O(|V| + |E|)$ compared to the best worst-case running time known for Dijkstra's algorithm: $O(|V| \log |V| + |E|)$ time. It is also very straightforward to implement. It is easy to incorporate heuristics that take into account the capacity of an edge already used and bias the search towards edges with little capacity used. Furthermore, breadth-first search can be started from several vertices simultaneously, so that all terminals of a net could be starting points of the search simultaneously. If it is adequate to view all channel segments as being of equal length, then the edge lengths can all be assigned value 1 and breadth-first search can be used. This might occur when the terminals are uniformly distributed and so divide channels into approximately equal-length segments. Alternatively, one can add new vertices to the channel intersection graph to further decompose the channel segments into unit-length segments. This can substantially increase $|V|$ and $|E|$ so that they are proportional to the dimensions of the underlying grid defining the unit of length rather than the number of channels and terminals in the problem. However, this allows one to use breadth-first search to compute shortest paths while modeling the actual lengths of channels. In fact, breadth-first search was developed by Lee [16][7] for routing of circuit boards in exactly this manner. He modeled the entire board by a grid graph, and modeled obstacles to routing paths as grid vertices that were missing from the graph. Each wire route was found by doing a breadth-first search in the grid graph.

23.7 Channel Routing

Channel routing is not one single problem, but rather a family of problems based on the allowable paths for wires in the channel. We will limit our discussion to grid-based routing. While both grid-free rectilinear and nonrectilinear routing techniques exist, the most basic techniques are grid-based. We assume that there is a grid underlying the channel, the sides of the channel lie on grid edges, terminals lie on grid points, and all routing paths must follow edges in the grid. For ease of discussion, we shall refer to channel directions as though the channel were oriented with its length running horizontally. The vertical segments of the grid that run from the top to the bottom of the channel are referred to as *columns.* The horizontal segments of the grid that run from the left edge to the right edge of the channel are referred to as *tracks.* We will consider channel routing problems that allow the width of the channel to vary. Therefore, the

[7] The first published description of breadth-first search was by E. F. Moore for finding a path in a maze. Lee developed the algorithm for routing in grid graphs under a variety of path costs. See the discussion on page 394 of [19].

number of columns, determining the channel length, will be fixed, but the number of tracks, determining the channel width, will be variable. The goal is to minimize the number of tracks used to route the channel.

The next distinction is based on how many routing layers are presumed. If there are ℓ routing layers, then there are ℓ overlaid copies of the grid, one for each layer. Routes that use the same edge on different layers do not interact and are considered disjoint. Routes change layer at grid points. The exact rules for how routes can change layers vary, but the most common is to view a route that goes from layer i to layer j ($j > i$) at a grid point as using layers $i + 1, \ldots, j - 1$ as well at the grid point.

One can separate the channel routing problem into two subproblems: finding Steiner trees in the grid that achieve the interconnections, and finding a **layer assignment** for each edge in each Steiner tree so that the resulting set of routes is legal. One channel routing model for which this separation is made is knock-knee routing. (See Section 9.5 of [19].) Given any set of trees in the grid (not only those for knock-knee routes), a legal layer assignment can be determined efficiently for two layers. Maximal single-layer segments in each tree can be represented as vertices of a conflict graph, with segments that must be on different layers related by conflict edges. Then finding a legal two-layer assignment is equivalent to two-coloring the conflict graph. A more challenging problem is **via minimization,** which is to find a legal layer assignment that minimizes the number of grid points at which layer change occurs. This problem is also solvable in polynomial time as long as none of the Steiner trees contain a four-way split (a technical limitation of the algorithm). For more than two layers, both layer assignment and via minimization are NP-complete. (See Section 9.5.3 of [19]).

We have chosen to discuss two routing models in detail: single-layer routing and two-layer **Manhattan routing.** Routing models that allow more than two layers, called multilayer models, are becoming more popular as technology is able to provide more layers for wires. However, many of the multilayer routing techniques are derived from single-layer or two-layer Manhattan techniques (e.g.,[8]), so we focus this restricted discussion on those models. A more detailed review of channel routing algorithms can be found in [15]. Neither model requires layer assignment. In single-layer routing there is no issue of layer assignment; in Manhattan routing, the model is such that the layer assignment is automatically derived from the paths. Therefore, for each model, our discussion need only address how to find a collection of Steiner trees that satisfy the restrictions of the routing model.

Manhattan Routing

Manhattan routing is the dominant 2-layer routing model. It dates back to printed circuit boards [10]. It dominates because it finesses the issue of layer assignment by defining all vertical wire segments to be on one layer and all horizontal wire segments to be on the other layer. Therefore, a horizontal routing path and a vertical routing path can cross without interacting, but any path that bends at a grid point is on both layers at that point and no path for a disjoint net can use the same point. Thus, under the Manhattan model, the problem of routing can be stated completely in terms of finding a set of paths such that paths for distinct nets may cross but do not share edges or bend points.

Although Manhattan routing provides a simple model of legal routes, the resulting channel routing problem is NP-complete. An important lower bound on the width of a channel is the **channel density.** The density at any vertical line cutting the channel (not necessarily a column) is the number of nets that have terminals both to the left and right of the vertical line. The interpretation is that each net must have at least one wire crossing the vertical line, and thus a number of tracks equal to the density at the vertical line is necessary. For columns, nets that contain a terminal on the column are counted in the density unless the net contains exactly two terminals and these are at the top and bottom of the column. (Such a net can be routed with a vertical path the extent of the column.) The channel density is the maximum density of any vertical cut of the channel. In practice, the channel routing problem is solved with heuristic algorithms that find routes giving a channel width within one or two of density, although such performance is not provably achievable.

If a channel routing instance has no top terminal and bottom terminal on the same column, then a number of tracks equal to the channel density suffices and a route achieving this density can be solved in $O(m + n \log n)$ time, where n is the number of nets and m is the number of terminals. Under this restriction, the channel routing problem becomes equivalent to the problem of **interval graph coloring.** The equivalence is based on the observation that if no column contains terminals at its top and bottom, then any terminal can connect to any track by a vertical segment that does not conflict with any other path (see Fig. 23.9). Each net can use one horizontal segment that spans all the columns containing its terminals. The issue is to pack these horizontal segments into the minimum number of tracks so that no segments intersect. Equivalently, the goal is to color the segments (or intervals) with the minimum number of colors so that no two segments that intersect are the same color. Hence we have the relationship to interval graphs, which have a set of vertices that represent intervals on a line and edges between vertices of intersecting intervals.

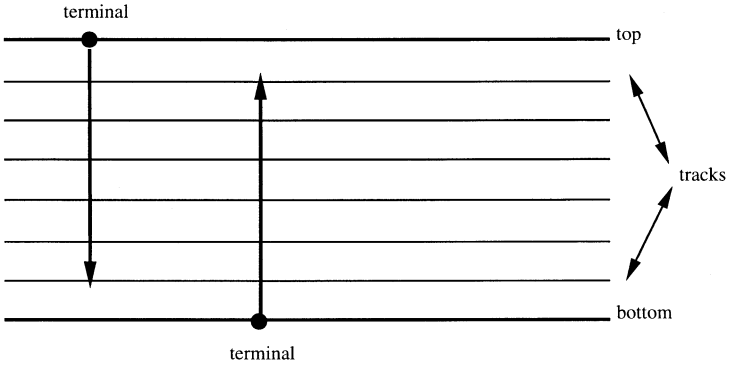


FIGURE 23.9 Connecting to tracks without conflicts. Each of the nets can reach any of the tracks with a vertical segment in the column containing the terminal.

A classic greedy algorithm can assign a set I of intervals to d tracks, where d is the **channel density.** Intervals are placed in tracks in order of their left endpoints; all tracks are filled with intervals from left to right concurrently (the actual position of each track does not matter). The set of interval endpoints is first sorted so that the combined order of left (starting) endpoints and right (ending) endpoints is known. At any point in the algorithm, there are tracks containing intervals that have not yet ended and tracks that are free to receive new intervals. A queue F is used to hold the free tracks; the function FREE inserts (enqueues) a track in F. The unprocessed endpoints are stored in a queue of points, P, sorted from left to right. The function DEQUEUE is the standard deletion operation for a queue [3]. (See Chapter 4.) The algorithm starts with only one track and adds tracks as needed; variable t holds the number of tracks used.

INTERVAL-BY-INTERVAL ASSIGNMENT (I)

```
1  Sort the endpoints of the intervals in I and build queue P
2  t ← 1
3  FREE track 1
4  while P is not empty
5     do p ← DEQUEUE(P)
6        if p is the left endpoint of an interval i
7           then do  if F is empty
```

8 **then do** $t \leftarrow t + 1$
9 Put i in track t
10 **else do** $track \leftarrow$ DEQUEUE(F)
11 Put i in $track$
12 **else do** FREE the track containing the interval whose right endpoint is p

To see that this algorithm never uses more than a number of tracks equal to the density d of the channel, note that when t is increased at line 8 it is because the current t tracks all contain intervals that have not yet ended when the left endpoint p obtained in line 5 is considered. Therefore, when t is increased for the last time to value w, all tracks numbered less than w contain intervals that span the current p; hence $d \geq w$. Since no overlapping intervals are places in the same track, $w = d$. Therefore, the algorithm finds an optimal track assignment. Preprocessing to find the interval for each net takes time $O(m)$ and INTERVAL-BY-INTERVAL ASSIGNMENT has running time $O(|I| \log |I|) = O(n \log n)$, due to the initial sorting of the endpoints of I.

Once one allows terminals at both the top and bottom of a column (except when all such pairs of terminals belong to the same net), one introduces a new set of constraints called *vertical constraints.* These constraints capture the fact that if net i has a terminal at the top of column c and net j has a terminal at the bottom of column c, then to connect these terminals to horizontal segments using vertical segments at column c, the horizontal segment for i must be above the horizontal segment for j. On can construct a vertical constraint graph that has a vertex v_i for each net i and a directed edge between v_i and v_j if there is a column that contains a terminal in i at the top and a terminal in j at the bottom. If one considers only routes that use at most one horizontal segment per net, then the constraint graph represents order constraints on the tracks used by the horizontal segments. If the vertical constraint graph is cyclic, then the routing cannot be done with one horizontal segment per net. If the vertical constraint graph is acyclic, it is NP-complete to determine if the routing can be achieved in a given number of tracks (see [19, p. 547]). Furthermore, even if an optimal or good routing using one horizontal segment per net is found, the number of tracks required is often substantially larger than what could be obtained using more horizontal segments. For these reasons, practical channel routing algorithms allow the route for each net to traverse portions of several tracks. Each time a route changes from one track to another, it uses a section of a column; this is called a **jog** (see Fig. 23.10).

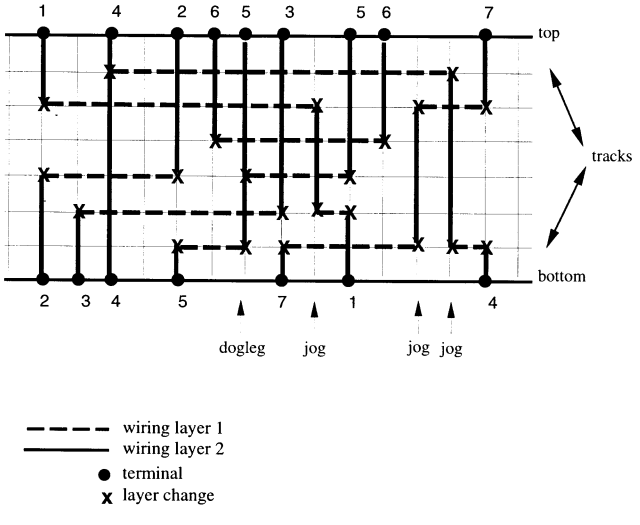


FIGURE 23.10 A channel routing in the 2-layer Manhattan model showing jogs.

Manhattan channel routing remains NP-complete even if unrestricted jogs are allowed (see [19, p. 541]). Therefore, the practical routing algorithms for this problem use heuristics. The earliest of these is by Deutsch [4]. Deutsch allows the route for a net to jog only in a column that contained a terminal of the net; he calls these jogs "doglegs" (see Fig. 23.10). This approach effectively breaks up each net into two-point subnets, and one can then define a vertical constraint graph in which each vertex represents a subnet. Deutsch's algorithm is based on a modification of INTERVAL-BY-INTERVAL ASSIGNMENT called *track-by-track assignment*. Track-by-track assignment also fills tracks greedily from left to right but fills one track to completion before starting the next. Deutsch's basic algorithm does track-by-track assignment but does not assign an interval for a subnet to a track if the assignment would violate a vertical constraint. Embellishments on the basic algorithm try to improve its performance and minimize the number of doglegs. Others have also modified the approach. (See the discussion in Section 9.6.1.4 of [19].) The class of algorithms based on greedy assignment of sorted intervals is also known as "left-edge algorithms."

Manhattan channel routing is arguably the most widely used detailed routing model and many algorithms have been developed. The reader is referred to [31] or [15] for a survey of algorithms. In this chapter, we will discuss only one more algorithm—an algorithm that proceeds column-by-column in contrast to the track-by-track algorithm. The column-by-column algorithm was originally proposed by Rivest and Fiduccia [28] and was called by them a "greedy" router. This algorithm routes all nets simultaneously. It starts at the leftmost column of the channel and proceeds to the right, considering each column in order. As it proceeds left to right it creates, destroys and continues horizontal segments for nets in the channel. Using this approach, it is easy to introduce a jog for a net in any column. At each column, the algorithm connects terminals at the top and bottom to horizontal segments for their nets, starting new segments when necessary, and ending segments when justified. At each column, for each continuing net with terminals to the right, it may also create a jog to bring a horizontal segment of the route closer to the channel edge containing the next terminal to the right. Thus, the algorithm employs some "look ahead" in determining what track to use for each net at each column. The algorithm is actually a framework with many parameters that can be adjusted. It may create, for one net, multiple horizontal segments that cross the same column and may extend routes beyond the left and right edges of the channel. It is a very flexible framework that allows many competing criteria to be considered and allows the interaction of nets more directly than strategies that route one net at a time. Many routing tools have adopted this approach.

Single-Layer Routing

Although single-layer channel routing plays a role in the routing of multilayer channels (e.g.,[8]), its greatest significance comes from the fact that even in its most general form, it can be solved optimally in polynomial time. There is a rich theory of single-layer detailed routing that has been developed not only for channel routing, but for routing in more general regions (see [22]). The first algorithmic results for single-layer channel routing were by Tompa [34], who considered **river routing** problems. A river routing problem is a single-layer channel routing problem in which each net contains exactly two terminals, one at the top edge of the channel and one at the bottom edge of the channel. The nets have terminals in the same order along the top and bottom—a requirement if the problem is to be routable in one layer. Tompa considered unconstrained (versus rectilinear) wire paths and gave a $O(n^2)$-time algorithm for n nets to test routability and find the route that minimizes both the individual wire lengths and the total wire length when the width of the channel is fixed. This algorithm can be used as a subroutine within binary search to find the minimum-width channel in $O(n^2 \log n)$ time. Tompa also suggested how to modify his algorithm for the rectilinear case. Dolev et al. [5] built upon Tompa's theory for the rectilinear case and presented an $O(n)$-time algorithm to compute the minimum width of the channel and an $O(n^2)$-time algorithm to actually produce the rectilinear routing. The difference in running times comes from the fact that the routing may actually have n segments per net, and thus would take $O(n^2)$-time to generate (see Fig. 23.11). In contrast, the testing for routability can be done by examining a set of constraints for the

channel. The results were generalized to multi-terminal nets by Greenberg and Maley [9], where the time to calculate the minimum width remains linear in the number of terminals.

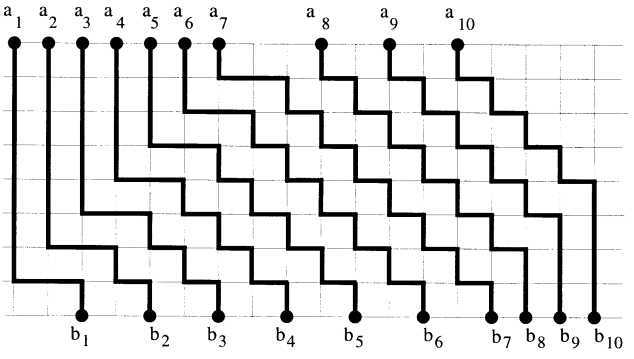


FIGURE 23.11 A river routing. The route for a single net may bend $O(n)$ times.

We now present the theory that allows the width of a river-routing channel to be computed in linear time. Our presentation is an amalgam of the presentations in [5] and [18]. The heart of the theory is the observation that for river routing, cut lines other than the vertical lines that define channel density also contribute to a lower bound for the width of the channel. This lower bound is then shown to be an upper bound as well. Indexing from left to right, let the ith net of a routing channel, $1 \leq i \leq n$, be denoted by the pair (a_i, b_i) where a_i is the horizontal position of its terminal on the top of the channel and b_i is the horizontal position of its terminal on the bottom of the channel. Consider a line from b_i to a_{i+k}, $k > 0$, cutting the channel. There are $k + 1$ nets that must cross this line. Measuring slopes from 0° to 180°, if the line has slope $\geq 90^\circ$, then $k + 1$ nets must cross the vertical (90°) line at b_i, and there must be $k + 1$ tracks. If the line has slope $< 90^\circ$ and $> 45^\circ$, then each vertical grid segment that crosses the line can be paired with a horizontal grid segment that must also cross the line and which cannot be used by a different net. Therefore, the line must cross $k + 1$ tracks. Finally, if the line has slope $\leq 45^\circ$, then each horizontal grid segment that crosses the line can be paired with a vertical grid segment that must also cross the line and which cannot be used by a different net. Therefore, there must be $k + 1$ columns crossing the line, i.e., $a_{i+k} - b_i \geq k$. Similarly, by considering a line from b_{i+k} to a_i, we conclude $k + 1$ tracks are necessary unless $b_{k+i} - a_i \geq k$. In the case of $k = 0$, a_i must equal b_i for no horizontal track to be required. Based on this observation, it can be proved that the minimum number of tracks t required by an instance of river routing is the least t such that for all $1 \leq i \leq n - t$

$$b_{i+t} - a_i \geq t \tag{23.14}$$

and

$$a_{i+t} - b_i \geq t \; . \tag{23.15}$$

To find the minimum such t in linear time, observe that $a_{i+k+1} \geq a_{i+k} + 1$ and $b_{i+k+1} \geq b_{i+k} + 1$. Therefore,

$$\text{if } b_{i+k} - a_i \geq k \text{ then } b_{i+k+1} - a_i \geq b_{i+k} + 1 - a_i \geq k + 1 \tag{23.16}$$

and

$$\text{if } a_{i+k} - b_i \geq k \text{ then } a_{i+k+1} - b_i \geq a_{i+k} + 1 - b_i \geq k + 1 \; . \tag{23.17}$$

Therefore, we can start with $t = 0$ and search for violated constraints from $i = 1$ to $n - t$; each time a constraint of the form of (23.14) or (23.15) above is violated, we increase t by one and continue the search; t can be no larger than n. Let N denote the set of nets in a river routing channel, $|N| = n$. The following algorithm calculates the minimum number of tracks needed to route this channel.

RIVER-ROUTING WIDTH (N)

```
1  i ← 1
2  t ← 0
3  while i ≤ |N| − t
4     do if b_{i+t} − a_i ≥ t and a_{i+t} − b_i ≥ t
5        then do i ← i + 1
6        else do t ← t + 1
7  return t
```

The actual routing for a river-routing channel can be produced in greedy fashion by routing one net at a time from left to right, and routing each net beginning with its left terminal. The route of any net travels vertically whenever it is not blocked by a previously routed net, travels horizontally right until it can travel vertically again or until it reaches the horizontal position of the right terminal of the net. This routing takes worst-case time $O(n^2)$, as the example in Fig. 23.11 illustrates.

23.8 Research Issues and Summary

This chapter has given an overview of the design problems arising from the computer-aided layout of VLSI circuits and some of the algorithmic approaches used. The algorithms presented in this chapter draw upon the theoretical foundations discussed in other chapters. Graph models are predominant and are frequently used to capture constraints. Since many of the problems are NP-complete, heuristics are used. Research continues in this field, both to find better methods of solving these difficult problems—both in efficiency and quality of solution—and to model and solve new layout problems arising from the ever-changing technology of VLSI fabrication and packaging. Layout techniques particular to increasingly popular technologies such as field-programmable gate arrays (FPGAs) and multichip modules (MCMs) have been and continue to be developed.

A major theme of current research in VLSI layout is the consideration of circuit performance as well as layout area. As feature sizes continue to shrink, wire delay is becoming an increasingly large fraction of total circuit delay. Therefore, the delay introduced by routing is increasingly important. The consideration of performance has necessitated new techniques, not only for routing but for partitioning and placement as well. In some cases, the techniques for area-based layout have been extended to consider delay. For example, the simulated annealing approach to placement has been modified to consider delay on critical paths. However, many researchers are developing new approaches for performance-based layout. For example, one finds the use of the techniques of linear programming, integer programming, and even specialized higher-order programming (see Chapters 31 and 32) in recent work on performance-driven layout. Clock-tree routing, to minimize clock delay and clock skew, is also receiving attention as an important component of performance-driven layout.

The reader is referred to the references given in *Further Information* for broader descriptions of the field and, in particular, for more thorough treatments of current research.

23.9 Defining Terms

Channel: A rectangular region for routing wires, with terminals lying on two opposite edges, called the "top" and "bottom." The other two edges contain no terminals, but wires may cross these edges for nets that enter the channel from other channels. The routing area of a layout is decomposed into several channels.

Channel density: Orient a channel so that the top and bottom are horizontal edges. Then the density at any vertical line cutting the channel is the number of nets that have terminals both to the left and right of the vertical line. Nets with a terminal on the vertical line contribute to the density unless all of the terminals of the net lie on the vertical line. The channel density is the maximum density of any vertical cut of the channel.

Channel routing: The problem of determining the routes, i.e., paths and layers, for wires in a routing channel.

Compaction: The process of modifying a given layout to remove extra space between features of the layout.

Floor plan: An approximate layout of a circuit that is made before the layouts of the components composing the circuit have been completely determined.

Floor plan sizing: Given a floor plan and a set of components, each with a shape function, finding an assignment of specific shapes to the components so that the area of the layout is minimized.

Floor planning: Designing a floor plan.

General cell layout: A style of layout in which the components may be of arbitrary height and width and functional complexity.

Global routing: When a layout area is decomposed into channels, global routing is the problem of choosing which channels will be used to make the interconnections for each net.

Graph partitioning: Given a graph with weights on its vertices and costs on its edges, the problem of partitioning the vertices into some given number k of approximately equal-weight subsets such that the cost of the edges that connect vertices in different subsets is minimized.

Interval graph coloring: Given a finite set of n intervals $\{[l_i, r_i], 1 \leq i \leq n\}$, for integer l_i and r_i, color the intervals with a minimum number of colors so that no two intervals that intersect are the same color. The graph representation is direct: each vertex represents an interval, and there is an edge between two vertices if the corresponding intervals intersect. Then coloring the interval graph corresponds to coloring the intervals.

Jog: In a rectilinear routing model, a vertical segment in a path that is generally running horizontally, or vice versa.

Layer assignment: Given a set of trees in the plane, each interconnecting the terminals of a net, an assignment of a routing layer to each segment of each tree so that the resulting wiring layout is legal under the routing model.

Manhattan routing: A popular rectilinear channel routing model in which paths for disjoint nets can cross (a vertical segment crosses a horizontal segment) but cannot contain segments that overlap in the same direction at even a point.

Net: A set of terminals to be connected together.

Rectilinear: With respect to layouts, describes a layout for which there is an underlying pair of orthogonal axes defining "horizontal" and "vertical;" the features of the layout, such as the sides of the components and segments of the paths of wires, are horizontal and vertical line segments.

River routing: a single-layer channel routing problem in which each net contains exactly two terminals, one at the top edge of the channel and one at the bottom edge of the channel. The nets have terminals in the same order along the top and bottom—a requirement if the problem is to be routable in one layer.

Shape function: A function that gives the possible dimensions of the layout of a component with a flexible (or not yet completely determined) layout. For a shape function $s : [w_{min}, \infty] \rightarrow [h_{min}, \infty]$ with $[w_{min}, \infty]$ and $[h_{min}, \infty]$ subsets of $\Re^+$, $s(w)$ is the minimum height of any rectangle of width w that contains a layout of the component.

Slicing floor plan: A floor plan which can be obtained by the recursive bipartitioning of a rectangular layout area using vertical and horizontal line segments.

Steiner tree: Given a graph $G = (V, E)$ a Steiner tree for a subset of vertices U of V is a subset of edges of G that form a tree and contain among their endpoints all the vertices of U. The tree may contain other vertices than those in U. For a Euclidean Steiner tree, U is a set of points in the Euclidean plane, and the tree interconnecting U can contain arbitrary points and line segments in the plane.

Switch box: A rectangular routing region containing terminals to be connected on all four sides of the rectangle boundary and for which the entire interior of the rectangle can be used by wires (contains no obstacles).

Terminal: A position within a component where a wire attaches. Usually a terminal is a single point on the boundary of a component, but a terminal can be on the interior of a component and may consist of a set of points, any of which may be used for the connection. A typical set of points is an interval along the component boundary.

Topological sort: Given a directed, acyclic graph, a topological sort of the vertices of the graph is a total ordering of the vertices such that if vertex u comes before vertex v in the ordering, there is no directed path from v to u.

Via minimization: Given a set of trees in the plane, each interconnecting the terminals of a net, determining a layer assignment that minimizes the number of points (vias) at which a layer change occurs.

Acknowledgments

I would like to thank Ron Pinter for his help in improving this chapter.

References

[1] Alpert, C.J. and Kahng, A.B., Recent directions in netlist partitioning: a survey, *Integration: the VLSI Journal.* 19, 1–81, 1995.

[2] Asano, T., Sato, M., and Ohtsuki, T., In *Layout Design and Verification,* Ohtsuki, T., Ed., 295–347. North-Holland, Amsterdam, 1986.

[3] Cormen, T.H., Leiserson, C.E., and Rivest, R.L., *Introduction to Algorithms,* MIT Press, Cambridge, MA, 1990.

[4] Deutsch, D.N., A dogleg channel router. In *Proc. of the 13th ACM/IEEE Design Automation Conf.,* 425–433. IEEE, 1976.

[5] Dolev, D., Karplus, K., Siegel, A., Strong, A., and Ullman, J.D., Optimal wiring between rectangles. In *Proc. of the 13th Annual ACM Symp. on Theory of Computing,* 312–317. ACM, 1981.

[6] Fiduccia, C.M. and Mattheyses, R.M., A linear-time heuristic for improving network partitions. In *Proc. of the 19th ACM/IEEE Design Automation Conf.,* 175–181. IEEE, 1982.

[7] Garey, M.R. and Johnson, D.S., *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W.H. Freeman, San Francisco, 1979.

[8] Greenberg, R.I., Ishii, A.T., and Sangiovanni-Vincentelli, A.L., MulCh: A multi-layer channel router using one, two and three layer partitions. In *IEEE Inter. Conf. on Computer-Aided Design,* 88–91. IEEE, 1988.

[9] Greenberg, R.I. and Maley, F.M., Minimum separation for single-layer channel routing. *Information Processing Letters.* 43, 201–205, 1992.

[10] Hashimoto, A. and Stevens, J., Wire routing by optimizing channel assignment within large apertures. In *Proc. of the 8th IEEE Design Automation Wkshp.,* 155–169. IEEE, 1971.

[11] Hwang, F.K., On Steiner minimal trees with rectilinear distance. *SIAM J. on Applied Math.,* 30(1), 104–114, 1976.
[12] Joy, D. and Ciesielski, M., Layer assignment for printed circuit boards and integrated circuits. *Proceedings of the IEEE.* 80(2), 311–331, 1992.
[13] Kernighan, W. and Lin, S., An efficient heuristic procedure for partitioning graphs. *Bell System Tech. J.,* 49, 291–307, 1970.
[14] Kuh, E.S. and Ohtsuki, T., Recent advances in VLSI layout. *Proceedings of the IEEE.* 78(2), 237–263, 1990.
[15] LaPaugh, A.S. and Pinter, R.Y., Channel routing for integrated circuits. In *Annual Review of Computer Science,* vol. 4, J. Traub, Ed., 307–363. Annual Reviews Inc., Palo Alto, CA, 1990.
[16] Lee, C.Y., An algorithm for path connection and its applications. *IRE Trans. on Electronic Computers.* EC-10(3), 346–365, 1961.
[17] Lee, T.-C., A bounded 2D contour searching algorithm for floor plan design with arbitrarily shaped rectilinear and soft modules. In *Proc. of the 30th ACM/IEEE Design Automation Conf.,* 525–530. ACM, 1993.
[18] Leiserson, C.E. and Pinter, R.Y., Optimal placement for river routing. *SIAM J. Computing.* 12(3), 447–462, 1983.
[19] Lengauer, T., *Combinatorial Algorithms for Integrated Circuit Layout.* John Wiley & Sons, West Sussex, England, 1990.
[20] Liao, Y.Z. and Wong, C.K., An algorithm to compact a VLSI symbolic layout with mixed constraints. *IEEE Trans. on Computer-Aided Design.* CAD-2(2), 62–69, 1983.
[21] Maley, F.M., An observation concerning constraint-based compaction. *Information Processing Letters.* 25(2), 119–122, 1987.
[22] Maley, F.M., *Single-Layer Wire routing.* Ph.D. Thesis, Department of Electrical Engineering and Computer Science, Mass. Inst. of Tech, Cambridge, MA, 1987.
[23] Mata, J.M., Solving systems of linear equalities and inequalities efficiently. In *15th Southeastern Conference on Combinatorics, Graph Theory and Computing.* 1984.
[24] Otten, R.H.J.M.. Efficient floor plan optimization. In *Proc. International Conf. on Computer Design: VLSI in Computers.* 499–502. IEEE, 1983.
[25] Pan, P. and Liu, C.L., Area minimization for floor plans. *IEEE Transactions on Computer-Aided Design.* CAD-14(1), 123–132, 1995.
[26] Preas, B.T. and Karger, P.G., Automatic placement: a review of current techniques. In *Proc. of the 23rd ACM/IEEE Design Automation Conf.,* 622–629. IEEE, 1986.
[27] Preas, B.T. and Lorenzetti, M.J., Ed., *Physical Design Automation of VLSI Systems.* Benjamins/Cummings, Menlo Park, CA, 1988.
[28] Rivest, R.L. and Fiduccia, C.M., A "greedy" channel router. In *Proc. 19th ACM/IEEE Design Automation Conf.,* 418–422. IEEE, 1982.
[29] Sarrafzadeh, M. and Wong, C.K., *An Introduction to VLSI Physical Design.* McGraw-Hill, New York, 1996.
[30] Sechen, C., Chip-planning, placement, and global routing of macro/custom cell integrated circuits using simulated annealing. In *Proc. of the 25th ACM/IEEE Design Automation Conf.,* 73–80. IEEE, 1988.
[31] Sherwani, N., *Algorithms for VLSI Physical Design Automation,* 2nd ed. Kluwer Academic, Norwell, MA, 1995.
[32] Shi, W., An optimal algorithm for area minimization of slicing floor plans. In *IEEE/ACM Inter. Conf. on Computer-Aided Design,* 480–484. IEEE, 1995.
[33] Stockmeyer, L., Optimal orientations of cells in slicing floor plan designs. *Information and Control.* 57, 91–101, 1983.
[34] Tompa, M., An optimal solution to a wire-routing problem. *Journal of Computer and System Sciences.* 23(2), 127–150, 1981.

[35] Wolf, W.H., *Modern VLSI Design: A Systems Approach.* Prentice-Hall, Englewood Cliffs, NJ, 1994.

Further Information

This chapter has given several examples of the successful application of the theory of combinatorial algorithms to problems in VLSI layout. It is by no means a survey of all the important problems and algorithms in the area. Several textbooks have been written on algorithms for VLSI layout, such as [19, 27, 29, 31], and the reader is referred to these for more complete coverage of the area. Other review articles of interest are [26] on placement, [2] on geometric algorithms for layout, [14] on layout, [15] on channel routing, [12] on layer assignment, and [1] on netlist partitioning. There are many conferences and workshops on computer-aided design that contain papers presenting algorithms for layout. The *ACM/IEEE Design Automation Conference* and the *IEEE/ACM International Conference on Computer-Aided Design* are the richest sources of algorithms among these conferences. The premier journal on the topic is *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems.* Other journals include *Integration, the VLSI Journal* published by North-Holland Publishing and *IEEE Transaction on VLSI Systems.* The ACM has just begun a journal *Transactions on Design Automation of Electronic Systems* that has the potential to become another good source of papers on layout algorithms. Layout algorithms also appear in journals focusing on general algorithm development such as *SIAM Journal on Computing,* published by the Society of Industrial and Applied Mathematics, *Journal of Algorithms,* published by Academic Press, and *Algorithmica,* published by Springer-Verlag.

24

Basic Notions in Computational Complexity

Tao Jiang
McMaster University

Ming Li
University of Waterloo

Bala Ravikumar
University of Rhode Island

24.1 Introduction

Computational complexity is aimed at measuring how complex a computational solution is. There are many ways to measure the complexity of a solution: how hard it is to understand it, how hard to express it, how long the solution process will take, and more. The last criterion—time—is most widely taken as the definition of complexity. This is because the agent that implements an **algorithm** is usually a computer—and from a user's point of view, the most important issue is how long one should wait to see the solution. However, there are other important measures of complexity, such as the amount of memory, hardware, information, communication, knowledge, or energy needed for the solution. Complexity theory is aimed at quantifying these resources precisely and studying the amounts of them required to accomplish computational tasks.

This technical sense of the word "complexity" did not take root until the mid-1960s. Today, complexity theory is not only a vibrant subfield of computer science, but also has direct or metaphoric impact on other fields such as dynamical systems and chaos theory. The seed of computational complexity is the formalization of the concept of an **algorithm.** Algorithms in turn must be planted in some *computational model,* ideally one that abstracts important features of real computing machines and processes. In this chapter we consider the **Turing machine,** a computer model created and studied by the British mathematician Alan Turing in the 1930s [39]. Chapter 26 will discuss different (and equivalent) ways of formalizing algorithms.

With the advent of commercial computers in the 1950s and 1960s, when processor speed was much lower and memory cost unimaginably higher than today, it became critical to design *efficient* algorithms for solving large classes of problems. Just knowing that a problem was solvable, which had been the main concern of computability theory since the 1930s, was no longer enough. Turing machines provided a basis

for Hartmanis and Stearns [14] to formally define the measures of **time complexity** and **space complexity** for computations. The latter refers to the amount of memory needed to execute the computation. They defined measures for other resources as well, and Blum [1] found a precise definition of *complexity measure* that was not tied to any particular resource or machine model. Together with earlier work by Rabin [30], these papers marked the beginning of *computational complexity theory* as an important new discipline. A closely related development was drawn together by Knuth, whose work on algorithms and data structures [19] created the field of *algorithm design and analysis.* All of this work has been recognized in annual Turing Awards given by the Association for Computing Machinery. Hartmanis' Turing Award lecture [13] recounts the origins of computational complexity theory and speculates on its future development.

The power of a computing machine depends on its internal structure as well as on the amount of time, space, or other resources it is allowed to consume. Restricting our model of choice, the Turing machine, in various internal ways yields progressively simpler and weaker computing machines. These machines correspond to a natural hierarchy of grammars defined and advanced by Chomsky [5], which we describe in Chapter 25. In this chapter we present the most basic computational models, and use these models to classify problems first by solvability and then by complexity.

Central issues studied by researchers in computational complexity include the following.

- For a given amount of resources, or for a given type of resource, what problems can and cannot be solved?
- What is the relationship between problems requiring essentially the same amount of the resource or resources? May they be equivalent in some intrinsic sense?
- What connections are there between different kinds of resources? Given more time, can we reduce the demand on memory storage, or vice-versa?
- What general limits can be set on the kind of problems that can be solved when resources are limited?

The end thrust of Turing's famous paper [39] was to demonstrate rigorously that several fundamental problems in logic cannot be solved by algorithms at all. Complexity theory aims to show similar results for many more problems in the presence of resource limits. Such an unsolvability result, although "bad news" in most contexts (cryptographic security is an exception) can have practical benefits: it may lead to alternative models, goals, or solution strategies. As will come out in Chapters 27 and 28, complexity theory has so far been much more successful at drawing relative conclusions and relationships between problems than in proving absolute statements about (un)solvability.

24.2 Computational Models

Throughout this chapter, Σ denotes a finite alphabet of symbols; unless otherwise specified, $\Sigma = \{0, 1\}$. Then Σ^* denotes the set of all finite strings, including the empty string ϵ, over Σ. A **language** over Σ is simply a subset of Σ^*.

We use **regular expressions** in specifying some languages. In advance of the formal definition to come in Chapter 25, we define them as one kind of "patterns" for strings to "match." The basic patterns are ϵ and the characters in Σ, which match only themselves. The null pattern $\emptyset$ matches no strings. Two patterns joined by "$+$" match any string that matches either of them. Two or more patterns written in sequence match any string composed of substrings that match the respective patterns. A pattern followed by a "$*$" matches any string that can be divided into zero or more successive substrings, each of which matches the pattern—here the "zero" case applies to ϵ, which matches any starred pattern. For example, the pattern $0(0+1)^*1$ matches any string that begins with a 0 and ends with a 1, with zero or more binary characters in between. The pattern can be used as a *name* for the language of strings that match it. In like manner, the pattern $(0^*10^*1)^*0^*$ stands for the language of binary strings that have an even number of 1s. This

pattern *says* that trailing 0s are immaterial to any such string, and the rest of the string can be broken into zero or more substrings, each of which ends in a 1 and has exactly one previous 1.

Finite Automata

The **finite automaton** (in its **deterministic** version) was introduced by McCulloch and Pitts in 1943 as a logical model for the behavior of neural systems [28]. Rabin and Scott introduced the **nondeterministic** finite automaton (NFA) in [31], and showed that NFAs are equivalent to deterministic finite automata, in the sense that they recognize the same class of languages. This class of languages, called the **regular languages,** had already been characterized by Kleene [18] and Chomsky and Miller [6] in terms of regular expressions and regular grammars, which will be described formally in Chapter 25.

In addition to their original role in the study of neural nets, finite automata have enjoyed great success in many fields such as the design and analysis of sequential circuits [21], asynchronous circuits [3], text-processing systems [23], and compilers. They also led to the design of more efficient algorithms. One excellent example is the development of linear-time string-matching algorithms, as described in [20]. Other applications of finite automata can be found in computational biology [35], natural language processing, and distributed computing [27].

A **finite automaton,** pictured in Fig. 24.1, consists of an *input tape* and a *finite control.* The input tape contains a finite string of input symbols, and is read one symbol at a time from left to right. The finite control is connected to an *input head* that reads each symbol, and can be in one of a finite number of *states.* The input head is *one-way,* meaning that it cannot "backspace" to the left, and *read-only,* meaning that it cannot alter the tape. At each *step,* the finite control may change its state according to its current state and the symbol read, and the head advances to the next tape cell. In an NFA there may be more than one choice of next state in a step. Figure 24.1 also shows the second step of a computation on an input string beginning *aabab* . . . When the input head reaches the right end of the input tape, if the machine is in a state designated "final" (or "accepting"), we say that the input string is *accepted;* else we say it is *rejected.* The following is the formal definition.

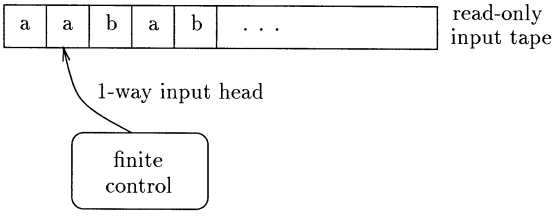


FIGURE 24.1 A finite automaton.

DEFINITION 24.1 A **nondeterministic finite automaton** (NFA) is a quintuple $(Q, \Sigma, \delta, q_0, F)$, where

- Q is a finite set of *states;*
- Σ is a finite set of *input symbols;*
- δ, the *state transition function,* is a mapping from $Q \times \Sigma$ to subsets of Q;
- $q_0 \in Q$ is the *start state* of the NFA;
- $F \subseteq Q$ is the set of *final states.*

If δ maps $|Q| \times \Sigma$ to singleton subsets of Q, then we call such a machine a **deterministic finite automaton** (DFA).

Note that a DFA is treatable as a special case of an NFA, where the next state is always uniquely determined by the current state and the symbol read. On any input string $x \in \Sigma^*$, a DFA follows a unique *computation path,* starting in state q_0. If the final state in the path is in F, then x is *accepted,* and the path is an *accepting path.* An NFA, however, may have multiple computation paths on the same input x. It is useful to imagine that when an NFA has more than one next state, all options are taken in parallel, so that the "super-computation" is a tree of branching computation paths. Then the NFA is said to *accept* x if at least *one* of those paths is an accepting path.

REMARKS: The concept of a nondeterministic automaton, and especially the notion of acceptance, can be nonintuitive and confusing at first. We can, however, explain them in terms that should be familiar, namely those of a *solitaire* game such as "Klondike." The game starts with a certain arrangement of cards, which we can regard as the input, and has a desired "final" position—in Klondike, when all the cards have been built up by suit from ace to king. At each step, the rules of the game dictate how a new arrangement of cards can be reached from the current one—and the element of nondeterminism is that there is often more than one choice of move (otherwise the game would be little fun!). Some positions have no possible move, and lose the game. Most crucially, some positions have moves that unavoidably lead to a loss, and other moves that keep open the possibility of winning. Now for a given position, the important analytical question is, "Is there a way to win?" The answer is "yes" so long as there is at least one sequence of moves that ends in a (or the) desired final position. For the starting position, this condition is much the same as for an input to be accepted by an NFA. (Practically speaking, some winnable starting positions may give so many chances to go wrong along the way that a player may have little chance to find a winning sequence of moves. That, however, is beside the point in defining which positions are *winnable.* If one can always (efficiently!) answer the yes/no question of whether a given position is winnable, then one can always avoid losing moves—and win—so long as the start position is winnable to begin with.)

In any event, the set of strings accepted by a (deterministic or nondeterministic) finite automaton M is denoted by $L(M)$. When we say that M *accepts a language* L, we mean that M accepts all strings in L *and nothing else,* i.e., $L = L(M)$. Two machines are **equivalent** if they accept the same language.

Nondeterminism is capable of modeling many important situations other than solitaire games. Concurrent computing offers some examples. Suppose a device or resource (such as a printer or a network interface) is controllable by more than one process. Each process could change the state of the device in a different way. Since there may be no way to predict the order in which processes may be given control in any step, the evolution of the device may best be regarded as nondeterministic.

Sometimes a state change can occur without input stimulus. This can be modeled by allowing an NFA to make ϵ*-transitions,* which change state without advancing the read head. Then the second argument of δ can be ϵ instead of a character in Σ, and these transitions can even be nondeterministic, for instance if $\delta(q_0, \epsilon) = \{q_1, q_2\}$. It is not hard to see that by suitable "lookahead" on states reachable by ϵ*-transitions,* one can always convert such a machine into an equivalent NFA that does not use them.

EXAMPLE 24.1:

We design an NFA to accept the language $0(0 + 1)^*1$. Recall that this regular expression defines those strings in $\{0, 1\}^*$ that begin with a 0 and end with a 1. A standard way to draw finite automata is exemplified by Fig. 24.2. As a convention, each circle represents a state, and an unlabeled arrow points to the start state (here, state "a"). Final states such as "c" have darker (or double) circles. The transition function δ is represented by the labeled edges. For example, $\delta(\mathrm{a}, 0) = \{\mathrm{b}\}$, and $\delta(\mathrm{b}, 1) = \{\mathrm{b}, \mathrm{c}\}$. When a transition is missing, such as on input 1 from state "a" and on both inputs from state "c," the transition is assumed to lead to an implicit nonaccepting "dead" state, which has transitions to itself on all inputs. In a DFA such a dead state must be included when counting the number of states, while in an NFA it can be left out.

The machine in Fig. 24.2 is nondeterministic since from "b" on input 1 the machine has two choices: stay at "b" or go to "c." Figure 24.3 gives a DFA that accepts the same language. The DFA has four, not three, states, since a dead state reached by an initial '1' is not shown.

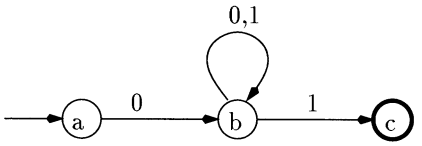


FIGURE 24.2 An NFA accepting $0(0+1)^*1$.

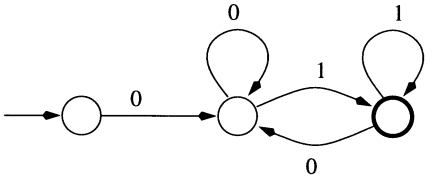


FIGURE 24.3 A DFA accepting $0(0+1)^*1$.

EXAMPLE 24.2:

The DFA in Fig. 24.4 accepts the language of all strings in $\{0, 1\}^*$ with an even number of 1's, which has the regular expression $(0^*10^*1)^*0^*$.

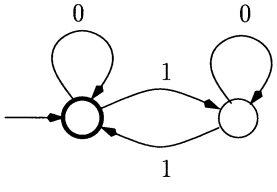


FIGURE 24.4 A DFA accepting $(0^*10^*1)^*0^*$.

EXAMPLE 24.3:

For a final example of a regular language, we introduce a general "tiling problem" to be discussed further in Chapter 26, and then strip it down to a simpler problem. A *tile* is a unit-sized square divided into four quarters by joining two diagonals. Each quarter has a color chosen from a finite set C of colors. Suppose you are given a set T of different types of tiles, and have an unlimited supply of each type. A $k \times n$ rectangle is said to be *tiled* using the tiles in T if the rectangle can be filled with exactly kn unit tiles (with no overlaps) such that at every edge between two tiles, the quarters of the two tiles sharing that edge have the same color. The tile set T is said to *tile* an entire plane if the plane can be covered with tiles subject to the color compatibility stated above. As a standard application of König's infinity lemma (for which see [19, Chapter 2, p. 381]), it can be shown that the entire plane can be tiled with a tile set T if and only if all finite integer sided rectangles can be tiled with T. We will see in Chapter 26 that the problem of whether a given tile set T can tile the entire plane is *unsolvable.* Chapter 26 will also say more about the meaning and implications of this tiling problem.

Here we will consider a simpler problem: Let k be a fixed positive integer. Given a set T of unit tiles, we want to know whether T can tile an infinite *strip* of width k. The answer is yes if T can tile any $k \times n$ rectangle for all n. It turns out that this problem is solvable by an efficient algorithm. One way to design such an algorithm is based on finite automata. We present the solution for $k = 1$ and leave the generalization for other values of k as an exercise. Number the quarters of each tile as in Fig. 24.5. Given a tile set T, we want to know whether for all n, the $1 \times n$ rectangle can be tiled using T.

FIGURE 24.5 Numbering the quarters of a tile.

To use finite automata, we define a language that corresponds to valid tilings. Define Σ, the input alphabet, to be T, the tile set. Each tile in T can be described by a 4-tuple $[A, B, P, Q]$ where A, B, P, and Q are (possibly equal) members of the color set C. Next we define a language L over Σ to be the set of strings $T_1T_2...T_n$ such that (i) each T_i is in Σ, and (ii) for each i, $1 \leq i \leq n-1$, T_i's third-quadrant color is the same as T_{i+1}'s first-quadrant color. These two conditions say that $T_1T_2...T_n$ is a valid $1 \times n$ tiling.

We will now informally describe a DFA M_L that recognizes the language L. Basically, M_L "remembers" (using the current state as the memory) the relevant information—for this problem, it need only remember the third-quadrant color Q of the *most recently seen* tile. Suppose the DFA's current state is Q. If the next (input) tile is $[X, Y, W, Z]$, it is consistent with the last tile if and only if $Q = X$. In this case, the next state will be W. Otherwise, the tile sequence is inconsistent, so M_L enters a "dead state" from which it can never leave and rejects. All other states of M_L are accepting states. Then the infinite strip of width 1 can be tiled if and only if the language L accepted by M_L contains strings of all lengths. There are standard algorithms to determine this property for a given DFA.

The next three theorems show the satisfying result that all the following language classes are identical:

- The class of languages accepted by DFAs;
- The class of languages accepted by NFAs;
- The class of languages generated by regular expressions;
- The class of languages generated by the right-linear, or Type-3, grammars, which are formally defined in Chapter 25 and informally used here.

This class of languages is called the **regular languages.**

THEOREM 24.1 *For every NFA, there is an equivalent DFA.*

PROOF An NFA might look more powerful since it can carry out its computation in parallel with its nondeterministic branches. But since we are working with a *finite number* of states, we can simulate an NFA $M = (Q, \Sigma, \delta, q_0, F)$ by a DFA $M' = (Q', \Sigma, \delta', q_0', F')$, where

- $Q' = \{[S] : S \subseteq Q\}$;
- $q_0' = [\{q_0\}]$;
- $\delta'([S], a) = [S'] = [\cup_{q_l \in S} \delta(q_l, a)]$;
- F' is the set of all subsets of Q containing a state in F.

Here square brackets have been placed around sets of states to help one view these sets as being states of the DFA M'. The idea is that whenever M' has read some initial segment y of its input x, its current state equals the set of states q such that M has a computation path reading y that leads to state q. When all of x is read, this means that M' is in an accepting state if and only if M has an accepting computation path. Hence $L(M) = L(M')$.

EXAMPLE 24.4:

Example 24.1 contains an NFA and an equivalent DFA accepting the same language. In fact the above proof provides an effective procedure for converting an NFA to a DFA. Although each NFA can be converted

to an equivalent DFA, the resulting DFA may require exponentially many more states, since the above procedure may assign a state for every eligible subset of the states of the NFA. For any $k > 0$, consider the language $L_k = \{x \mid x \in \{0, 1\}^*$ and the kth letter from the right of x is a 1$\}$. An NFA of $k + 1$ states (for $k = 3$) accepting L_k is given in Fig. 24.6. Now we claim that any DFA M accepting L_k needs a separate state for every possible value $y \in \{0, 1\}^k$ of the last k bits read. Take any distinct $y_1, y_2 \in \{0, 1\}^k$ and let i be a position in which they differ. Let $z = 0^{k-i}$. Then M must accept one of the strings y_1z, y_2z and reject the other. This is possible only if the state M is in after processing y_1 (with z to come) is different from that after y_2, and thus M needs a different state for each string in $\{0, 1\}^k$. The 2^k required states are also sufficient, as the reader may verify.

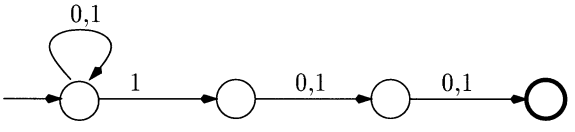


FIGURE 24.6 An NFA accepting L_3.

The remaining results of this section point forward to the **formal-language** models defined in Chapter 25. The point of including them here is to show the power of the finite automaton model. Regular expressions have been defined above, while a *regular grammar* over Σ consists of a set V of variable symbols, a starting variable $S \in V$, and a set P of substring-rewrite rules of the forms $A \to cB$, $A \to \epsilon$, or $A \to c$, where $A, B \in V$ and $c \in \Sigma$.

THEOREM 24.2 *A language L is generated by a regular grammar if and only if L is accepted by an NFA.*

PROOF Let L be accepted by an NFA $M = (Q, \Sigma, \delta, q_0, F)$. We define an equivalent regular grammar $G = (\Sigma, V, S, P)$ by taking $V = Q$ with $S = q_0$, adding a rule $q_i \to cq_j$ whenever $q_j \in \delta(q_i, c)$, and adding rules $q_j \to \epsilon$ for all $q_j \in F$. Then the grammar simulates computations by the NFA in a direct manner, giving $L(G) = L(M)$.

Conversely, suppose L is the language of a regular grammar $G = (\Sigma, V, S, P)$. We design an NFA $M = (Q, \Sigma, \delta, q_0, F)$ by taking $Q = V \cup \{f\}$, $q_0 = S$, and $F = \{f\}$. To define the δ function, we have $B \in \delta(A, c)$ iff $A \to cB$. For rules $A \to c$, $\delta(A, c) = \{f\}$. Then $L(M) = L(G)$—if this is not clear already, the treatment of grammars in Chapter 25 will make it so.

THEOREM 24.3 *A language L is specified by a regular expression if and only if L is accepted by an NFA.*

PROOF PROOF SKETCH. *Part 1.* We inductively convert a regular expression to an NFA that accepts the same language as follows.

- The pattern ϵ converts to the NFA $M_\epsilon = (\{q\}, \Sigma, \emptyset, q, \{q\})$, which accepts only the empty string.
- The pattern $\emptyset$ converts to the NFA $M_\emptyset = (\{q\}, \Sigma, \emptyset, q, \emptyset)$, which accepts no strings at all.
- For each $c \in \Sigma$, the pattern c converts to the NFA $M_c = (\{q, f\}, \Sigma, \delta(q, c) = \{f\}, q, \{f\})$, which accepts only the string c.
- A pattern of the form $\alpha + \beta$, where α and β are regular expressions that (by induction hypothesis) have corresponding NFAs M_α and M_β, converts to an NFA M that connects M_α and M_β in parallel: M includes all the states and transitions of M_α and M_β and has an extra start state q_0, which is connected by ϵ-transitions to the start states of M_α and M_β.

- A pattern of the form $\alpha \cdot \beta$, where α and β have the corresponding NFAs M_α and M_β, converts to an NFA M that connects M_α and M_β in series: M includes all the states and transitions of M_α and M_β and has extra ϵ-transitions from the final states of M_α to the start state of M_β. The start state of M is that of M_α, while the final states of M are those of M_β.
- A pattern of the form α^*, where α has the corresponding NFA M_α, converts to an NFA M that figuratively feeds M_α back into itself. M includes the states and transitions of M_α, plus ϵ-transitions from the final states of M_α back to its start state. This state is not only the start state of M but the only final state of M as well.

Part 2. We now show how to convert an NFA to an equivalent regular expression. The idea used here is based on [2]; see also [3] and [41].

Given an NFA M, add a new final state t, and add ϵ-transitions from each old final state of M to t. Also add a new start state s with an ϵ-transition to the old start state of M. The idea is to eliminate all states p other than s and t as follows. To eliminate a state p, we eliminate each arc coming in to p from some other state q as follows: For each triple of states q, p, q' as shown in Fig. 24.7(a), add the transition(s) shown in Fig. 24.7(b). (Note that if p does not have a transition leading back to p, then $\beta = \beta^* = \epsilon$.) After we have considered all such triples, we can delete state p and transitions related to p. Finally, we obtain Fig. 24.8, and the final α is a regular expression for $L(M)$.

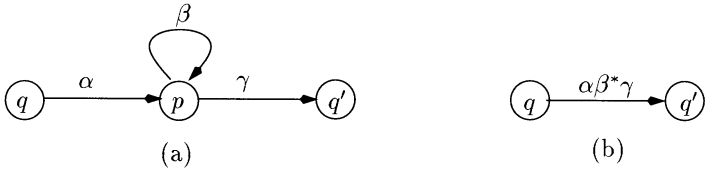


FIGURE 24.7 Converting an NFA to a regular expression.

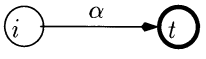


FIGURE 24.8 The reduced NFA.

The last three theorems underline the importance of the class of regular languages, since it connects to notions of automata, grammars, patterns, and much else. However, regular languages and finite automata are not powerful enough to serve as our model for a modern computer. Many extremely simple languages cannot be accepted by DFAs. For example, $L = \{xx \mid x \in \{0, 1\}^*\}$ cannot be accepted by a DFA. To show this, we can argue similarly to the "L_k" languages in Example 24.4 that for any two strings $y_1 = 0^m1$ and $y_2 = 0^n1$ with $n \neq m$, a hypothetical DFA M would need to be in a different state after processing y_1 from that after y_2, because with $z = 0^m1$ it would need to accept y_1z and reject y_2z. However, in this case we would conclude that M needs *infinitely* many states, contradicting the definition of a *finite* automaton. Hence the language L is not regular. Other ways to prove assertions of this kind include so-called "pumping lemmas" or a direct argument that some strings y contain more information than a *finite* state machine can *remember* [26]. We refer the interested readers to Chapter 25 and textbooks such as [10, 11, 15, 41].

One can also try to generalize the DFA to allow the input head to move backwards as well as forwards, in order to review earlier parts of the input string, while keeping it read-only. However, such "two-way DFAs" are not more powerful—they can be simulated by normal DFAs. The point of departure for a

more-powerful model is to allow the machine to *write* on its tape and later review what it has written. Then the tape becomes a *storage* medium, not just a sequence of events to react to. This ability to write down intermediate results for future reference makes DFAs into full-blown general-purpose computers.

Turing Machines

A **Turing machine** (TM), pictured in Fig. 24.9, consists of a *finite control,* an infinite *tape* divided into cells, and a read/write *head* on the tape. The finite control can be in any one of a finite set Q of states. Each cell can contain one symbol from the *tape alphabet* Γ, which contains the *input alphabet* Σ and a special character B called the *blank.* Γ may contain other characters besides B and those in Σ, but most often $\Sigma = \{0, 1\}$ and $\Gamma = \{0, 1, B\}$. We refer to the two directions on the tape as *left* and *right,* and abbreviate them by L and R. At any time, the head is positioned over a particular cell that it is said to *scan.* Initially, the head scans a distinguished cell on the tape called the *start cell,* the finite control is in the start state q_0, and all cells contain B except for a contiguous finite sequence of cells, extending from the start cell to the right, that contain characters in Σ. These cells hold the *input string* x; in case $x = \epsilon$ the whole tape is blank. The machine is said to begin its computation on input x at time 0, and computation unfolds in discrete time steps numbered $1, 2, \ldots$.

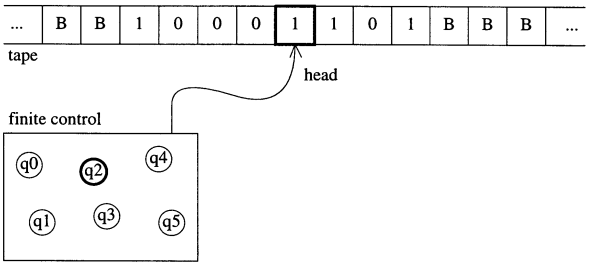


FIGURE 24.9 A Turing machine.

In any step, contingent on its current state and the character being scanned, the device is allowed to perform one the following two basic operations:

1. Write a symbol from the tape alphabet Γ into the scanned cell, or
2. Shift the head one cell left or right.

Then it may change its internal state in the same step. The allowed actions of a particular machine are specified by a finite set δ of *instructions.* Each instruction has the form (q, c, d, r) with $q, r \in Q, c \in \Gamma$, and either $d \in \Gamma$ or $d \in \{L, R\}$. This means that if the machine is in state q scanning character c on the tape, it may either change c to d (if $d \in \Gamma$) or move its head (if $d \in \{L, R\}$), and it enters state r. Either $c = d$ or $q = r$ is allowed. (Many texts use an alternate formalism in which *both* basic operations may be performed in the same step, so that instructions have the form (q, c, d, D, r) with $q, r \in Q, c, d \in \Gamma$, and $D \in \{L, R\}$, sometimes adding the option $D = S$ of keeping the head stationary. The differences do not matter for our purposes.)

If for every combination of q and c there is at most one instruction (q, c, d, r) that the machine can execute, then the machine is *deterministic.* Otherwise, the machine is *nondeterministic.* In order for computations to possibly halt, there must be some combinations q, c for which δ has *no* instruction (q, c, d, r). If a computation reaches such a state q while scanning c, the device is said to *halt.* Then if q is designated as a final state, we say the machine *accepts* its input string x; if q is not a final state, we say that the machine *rejects* the input. We adopt the convention that there is only one final state labeled q_f, and that q_f is also a halting state, meaning that there is no instruction (q, c, d, r) with source state $q = q_f$ at all.

DEFINITION 24.2 A **Turing machine** is a 7-tuple $M = (Q, \Sigma, \Gamma, \delta, B, q_0, q_f)$, where each of the components has been described above.

Given an input x, a deterministic Turing machine M carries out a uniquely determined succession of operations, which may or may not terminate in a finite number of steps. If it terminates, then the *output* $M(x)$ is determined to be the longest string of characters over Σ beginning in the cell in which the head halted and extending to the right. (If the scanned cell holds B or some other character in $\Gamma \setminus \Sigma$, then the output is ϵ.) A function $f : \Sigma^* \to \Sigma^*$ is **computable** if there is a Turing machine M such that for all inputs $x \in \Sigma^*$, $M(x) = f(x)$.

A nondeterministic Turing machine is analogous to an NFA. One may imagine that it carries out its computation in parallel. The computation may be viewed as a (possibly infinite) tree, each of whose nodes is labeled by a configuration of the machine. A *configuration* specifies the current state q, the position of the tape head, and the contents of the tape, including the character c currently being scanned. Each node has as many children as there are different instructions $(q, c, \cdot, \cdot)$ to execute, and each child is the configuration that results from executing the corresponding instruction. The root of the tree is the starting configuration of the machine. If any of the branches terminates in the final state q_f, we say the machine accepts the input. Note that this is the same "benefit of all doubt" criterion for acceptance that we discussed above for NFAs. Note also that a deterministic Turing machine always defines a "tree" with a single branch that forms a simple (possibly infinite) path.

A language L is said to be **recursively enumerable** (**r.e.**) if there is a Turing machine M such that $L = \{x : M \text{ accepts } x\}$. Furthermore, if M is *total,* i.e., if every computation of M on every input x halts, then $L(M)$ is **recursive.** These terms reflect an important correspondence between languages and functions. For any language $L \subseteq \Sigma^*$, define the *characteristic function* f_L by $f_L(x) = 1$ of $x \in L$, $f(x) = 0$ otherwise. Then a language L is recursive if and only if f_L is computable—and **computable functions** were originally defined with regard to formalisms that used *recursion.* Note that having L be acceptable by a Turing machine M is *not* enough for f_L to be computable, because there may be inputs $x \notin L$ for which the computation of M on x never halts.

Conversely, given a function $f : \Sigma^* \to \Sigma^*$, and letting # be a new input symbol not in Σ, one can define the language $L_f = \{x\#y : y \text{ is an initial segment of } f(x)\}$. Recognizing L_f allows one to find successive bits of the value $f(x)$. Hence it is common in the field to identify function problems with language problems and concentrate on the latter. A language can also be identified with a property of strings and with the associated **decision problem** "given a string x, does x have the property?" For instance, the problem of deciding whether a given number is prime is identifiable with the language of (binary string encodings of) prime numbers. The problem is **decidable** if the associated language is recursive, and a total Turing machine accepting the language is said to *decide* the problem. The term "decidable language" is a synonym for "**recursive language,**" and "recursive function" is a synonym for "computable function." A Turing machine M that does not halt on all inputs computes a **partial recursive function** (whose domain is a proper subset of Σ^*), and $L(M)$ is a **partially decidable** language (or problem, or property). Any problem or language that is not decidable by a Turing machine is called **undecidable,** and any (partial or total) function that is not computable by a Turing machine is called **uncomputable.**

Now when we say that a Turing machine M' *simulates* another Turing machine M, we usually mean more than saying they accept the same language or compute the same (partial) function. Usually there is some overt correspondence between computations of M and those of M'. This is so in the simulations claimed by the following theorem, which says that many variations in the basic machine model do not alter the notion of computability.

THEOREM 24.4 *All the following generalizations of Turing machines can be simulated by the one-tape deterministic Turing machine model defined in Definition 24.2, with tape alphabet* $\{0, 1, B\}$.

- *Enlarging the tape alphabet* Γ*;*

- *Adding more tapes;*
- *Adding more read/write heads or other access points on each tape;*
- *Having two- or higher-dimensional grids in place of tapes, where the head may move to any adjacent grid cell;*
- *Allowing nondeterminism.*

Extra tapes after the input tape are called *worktapes* provided they allow read-write access. A two-tape Turing machine, or alternately a one-tape machine with two heads, has instructions with six components: current state, two characters read by the heads, two head actions, next state. Although these generalizations do not make a Turing machine compute more, they do make Turing machines more efficient and easier to **program.** Many more variants of Turing machines have been studied and used in the literature.

Of all the simulations in Theorem 24.4, the last one needs special comment. A nondeterministic computation branches like a tree. The easiest way for a deterministic Turing machine to simulate it is by traversing the tree in a breadth-first manner, which is the same as trying all possibilities at any step with nondeterministic choices. However, even if there are at most two choices at any step, simulating n steps of the NTM could take on the order of 2^n steps by the DTM. Whether a more-efficient simulation is possible is bound up with the famous "P vs. NP" problem, to be discussed below and in Chapter 27.

EXAMPLE 24.5:

A DFA can be regarded as the special case of a Turing machine in which every instruction moves the head right. Turing machines naturally accept more languages than DFAs can. For example, a Turing machine can accept the non-regular language $L = \{xx \mid x \in \{0, 1\}^*\}$ as follows. Given an input string $w \in \{0, 1\}^*$:

- First find the middle point. A TM with two tape heads can do this efficiently by moving one head twice for every move of the other, until the further-advanced head sees the blank that marks the end of w. This stage can also tell whether w has even or odd length and immediately *reject* in the latter case.
- Then check whether the scanned characters match while moving both heads one cell left until the leftmost head sees the blank to the left of the beginning of w. If all pairs match, *accept,* else *reject.*

For a TM with only one head the strategy is more cumbersome. One way is to use "alias" characters a for 0 and b for 1, aliasing first the leftmost 0/1 character on the tape, then the rightmost, then the next-leftmost,... until finding the character just left of middle (if w has even length). Then "un-alias" it and check that the rightmost aliased character matches it, un-aliasing the latter as well. By looking for cases of an a or b immediately to the left of an un-aliased 0 or 1, the TM can repeat this check until all of the left half is compared with the right half. Whereas the two-head TM needs only $3n/2$ steps to decide whether a string w of length n belongs to L, the one-head TM takes about n^2 steps. It is known that n^2 steps are necessary (asymptotically) for any one-head TM to accept the language L (see, for instance, [15] or [26]).

Three restrictions of the notion of a Turing machine tape merit special mention.

- A *pushdown store* (or *stack*) is a semi-infinite worktape with one head such that each time the head moves to the left, it erases the symbol scanned previously. This is a last-in first-out storage.
- A *queue* is a semi-infinite work tape with two heads that only move to the right, the leading head is write-only and the trailing head is read-only. This is a first-in first-out storage.
- A *counter* is a pushdown store with a single-letter alphabet, except for a special bottom-of-stack marker that allows testing the counter for zero. Then a push increments the counter by one, and a pop decrements it by one.

EXAMPLE 24.6:

A **pushdown automaton** (PDA) has one read-only input tape and one pushdown store. A PDA can be identified with a two-tape Turing machine whose tape-1 head can never move left and whose tape-2 head can move left only while scanning a blank (combined with a previous step that writes a blank, this simulates popping the stack).

Pushdown automata have been thoroughly studied because nondeterministic PDAs accept precisely the class of context-free languages, to be defined in Chapter 25. Various types of PDAs have fundamental applications in compiler design.

The PDA has less power than a Turing machine. For example, $L = \{xx \mid x \in \{0, 1\}^*\}$ cannot be accepted by a PDA, but it can be accepted by a Turing machine as in Example 24.5. However, a PDA is more powerful than a DFA. For example, a PDA can accept the nonregular language $L' = \{0^n 1^n \mid n \geq 0\}$ easily: For each initial 0 read, push a 0 onto the stack; then pop a 0 for each 1 read, and accept if and only if a blank is read after the block of 1s exactly when the stack is empty. Indeed, this PDA is a counter machine with a single counter.

Two pushdown stores can easily be used to simulate a tape—let one stack represent the part of the tape currently to the left of the input head, and let the other stack represent the rightward portion. Much more subtle is the fact that two *counters* can simulate a tape; unlike the two-pushdown case this takes exponentially more time. Finally, a single queue can simulate a tape: send the lead head to the right end so that it can write the next-step update of the configuration that the trailing head is reading. This involves encoding the current state of the TM being simulated onto the tape of the queue machine that is simulating it. Hence a single-queue machine, with the input initially resting in the queue, is as powerful as a Turing machine, although it may require the square of the running time. For comparisons of powers of pushdown stores, queues, counters, and tapes, see [40, 26].

Much more important is the fact that there are single Turing machines that are capable of simulating *any* Turing machine. Formally, a **universal Turing machine** U takes an encoding $\langle M, w\rangle$ of a (deterministic) Turing machine M and a string w as input, and simulates M on input w. U accepts $\langle M, w\rangle$ if and only if M accepts w. Intuitively, U models a general-purpose computer that takes a "program" M and "data" w, and executes M on input w. Universal Turing machines have many applications. For example, the definition of Kolmogorov complexity (see [26]) fundamentally relies on them.

EXAMPLE 24.7:

Let $L_u = \{\langle M, w\rangle | M \text{accepts} w\}$. Then L_u is *the* language accepted by a universal Turing machine, so it is recursively enumerable. We shall see in Chapter 26, however, that L_u is not recursive. The same properties hold for the language $L_h = \{\langle M, w\rangle : M \text{ on input } w \text{ halts}\}$, which is the language of the so-called **Halting Problem.**

Oracle Turing Machines

In order to study the comparative hardness of computational problems, we often need to extend the power of Turing machines by adding oracles to them.

Informally, a Turing machine T with an *oracle* A operates similarly to a normal Turing machine, with the exception that it can write down a string z and ask whether z is in the language A. The machine gets the correct yes/no answer from the oracle in one step, and can branch its computation accordingly. This feature can be used as often as desired. We now give the definition precisely.

DEFINITION 24.3 An **oracle Turing machine** is a normal Turing machine T with an extra *oracle query tape,* a special state $q_?$, and two distinguished states labeled q_y and q_n. Let A be any language over an

alphabet Σ. Whenever T enters state $q_?$ with some string $z \in \Sigma^*$ on the query tape, control passes to state q_y if $z \in A$, or to q_n if $z \notin A$. The computation continues normally until the next time the machine enters $q_?$. The machine T with a given choice of oracle A is denoted by T^A.

EXAMPLE 24.8:

In Example 24.7, we know that the universal language $L_u = \{\langle M, w\rangle | M \text{accepts } w\}$ is not Turing decidable. But if we can use L_u as the oracle set, there is a trivial oracle TM T such that T with oracle L_u decides L_u. T simply copies its input x onto the query tape and enters $q_?$. If control passes to q_y, T accepts; otherwise from q_n it rejects.

For something less trivial, suppose we have $L_h = \{\langle M, w\rangle : M \text{ on input } w \text{ halts}\}$ as the oracle set. Given a (non-oracle) Turing machine M, there is a standard way to modify M to the code of an equivalent Turing machine M' in which the accepting state q_f is the only place where M' can halt. This is done by making every other combination q, c where M might halt send control to an extra state that causes an infinite loop. Thus M' halts on w if and only if M accepts w. Now design an oracle Turing machine T' that on any input x of the form $\langle M, w\rangle$ (rejecting if x does not have the form) writes $x' = \langle M', w\rangle$ on the query tape and enters $q_?$, accepting if control goes to q_y and rejecting from q_n. Then T' with oracle set L_h decides whether $x \in L_u$, since $x \in L_u \iff x' \in L_h$. This is a simple case where an oracle for one problem helps one decide a different problem.

A language A is **Turing-reducible** to a language B, written $A \leq_T B$, if there is an oracle Turing machine that with oracle B decides A. For example, we have just shown that $L_u \leq_T L_h$. The important special case in which the oracle TM T makes exactly one query, accepting from q_y and rejecting from q_n, gets its own notation: $A \leq_m B$. Equivalently, $A \leq_m B$ if there is a computable function f such that for all $x \in \Sigma^*, x \in A \iff f(x) \in B$. The function f represents the computation done by T prior to making the sole query. This case is called a **many-one reduction** (hence the subscript "m") for the arcane historical reason that f need not be a one-to-one function. The term "many-one reduction" is standard now. The above example actually shows that $L_u \leq_m L_h$. It is not hard to show that also $L_h \leq_m L_u$, so that the Halting Problem and the membership problem for a universal Turing machine are *many-one equivalent.*

Alternating Turing Machines

Turing machines can be naturally generalized to model parallel computation. A nondeterministic Turing machine accepts an input if there exists a move sequence leading to acceptance. We can call any nondeterministic state entered along this sequence an *existential state.* We can naturally add another type of state, a *universal state.* When a machine enters a universal state, the machine will accept if and only if *all* moves from this state lead to acceptance. These machines are called **alternating Turing machines.**

Let us describe the computation of alternating Turing machines formally and precisely. An alternating Turing machine is simply a nondeterministic Turing machine with the extra power that some states can be universal. A *configuration* of an alternating Turing machine A has the same form as was described for a deterministic Turing machine, namely,

```
(current state, tape contents, head positions).
```

We write

$$\alpha \vdash \beta$$

if, in one step, A can move from configuration α to configuration β. A configuration with current state q is *accepting* if

- q is an accepting state (i.e., $q = q_f$); or
- q is existential, and there exists an accepting configuration β such that $\alpha \vdash \beta$; or

- q is universal, and or each configuration β such that $\alpha \vdash \beta$, β is an accepting configuration.

This definition may seem circular, but by working backwards from configurations with q_f in the current-state field, one may verify that it inductively defines the set of accepting configurations in a natural manner. Then A accepts an input x if its initial configuration (with current state q_0, x on the input tape, heads at initial positions) is accepting.

Alternating Turing machines were first proposed by [4] for the purpose of modeling parallel computation. In order to allow sublinear computation times, a random-access model is used to read the input. When in a special "read" state, the alternating Turing machine is allowed to write a number in binary which is then interpreted as the address of a location on the input tape, whose symbol is then read in unit time. By using universal states to relate different branches of the computation, one can effectively read the whole input in as little as logarithmic time.

Just as a nondeterministic Turing machine is a model for solitaire games, an alternating Turing machine is a model for general two-person games. Alternating Turing machines have been successfully used to provide a theoretical foundation of parallel computation as well as to establish the complexity of various two person games. For example, a chess position with White to move can be modeled from White's point of view as a configuration α whose first component is an existential state. The position is winning if there exists a move for White such that the resulting position β is winning. Here β with Black to move has a universal state q, and is a winning position (for White) if and only if either Black is checkmated (this is the base case $q = q_f$) or for all moves by Black to a position γ, γ is a winning position for White. Chapter 28, Sections 28.5 and 28.6, will demonstrate the significance of games for time and space complexity, to which we now turn.

24.3 Time and Space Complexity

With Turing machines, we can now formally define what we mean by time and space complexity. The formal investigation by [1, 14] in the 1960s marked the beginning of the field of *computational complexity.*

An important point with space complexity is that a machine should be charged only for those cells it uses for calculation, and not for read-only input, which might be provided on cheaper non-writable media such as CD-ROM or accessed piecemeal over a network. Hence we modify the Turing machine of Fig. 24.9 by making the tape containing the input read-only, and giving it one or more worktapes.

DEFINITION 24.4 Let M be a Turing machine. If for all n, every sequence of legal moves on an input x of length n halts within $t(n)$ steps, we say that M is of **time complexity** $t(n)$. Similarly, if every such sequence uses at most $s(n)$ *worktape* cells, then M is of **space complexity** $s(n)$.

THEOREM 24.5 *Fix a number $c > 0$, a space bound $s(n)$, and a time bound $t(n)$.*

(a) Any Turing machine of $s(n)$ space complexity, using any number of tapes or grids of any dimension, can be simulated by a Turing machine with a single (one-dimensional) worktape that has space complexity $s(n)/c$.

(b) Any Turing machine of $t(n)$ time complexity can be simulated by a Turing machine, with the same number and kinds of worktapes, that has time complexity $n + t(n)/c$.

The proof of these so-called *linear speed-up theorems* involves enlarging the original TM's worktape alphabet Γ to an alphabet Γ' large enough that one character in Γ' can encode a block of c consecutive characters on a tape of the original machine (see [15]). The extra "$n+$" in the time for part (b) is needed to read and translate the input into the "compressed" alphabet Γ'. If we think of memory in units of bits the idea that this saves space and time is illusory, but if we regard the machine with Γ' as having a larger

word size, the savings make sense. Definition 24.4 is phrased in a way that applies also to nondeterministic and alternating TMs, and the two statements in Theorem 24.5 hold for them as well.

In Theorem 24.5, if $s(n) \geq n$, then we do not need to separate the input tape from the worktape(s). For any Turing machine M of linear space complexity, part (a) implies that we can simulate M by a one-tape TM M' that on any input x uses only the cells initially occupied by x (except for one visit to the blank cell to the right of x to tell where x ends). Then M' is called a *linear bounded automaton.*

The main import and convenience of Theorem 24.5 is that one does not need to use "$O()$-notation" to define complexity classes: space complexity $O(s(n))$ is no different from space complexity $s(n)$, and similarly for time. As we shall see, it is not always possible to reduce the number of tapes and run in the same time complexity, so researchers have settled on Turing machines with any finite number of tapes as the bench model for time complexity.

DEFINITION 24.5

- DTIME$[t(n)]$ is the class of languages accepted by multitape deterministic TMs in time $t(n)$;
- NTIME$[t(n)]$ is the class of languages accepted by multitape nondeterministic TMs in time $t(n)$;
- DSPACE$[s(n)]$ is the class of languages accepted by deterministic TMs in space $s(n)$;
- NSPACE$[s(n)]$ is the class of languages accepted by multitape nondeterministic TMs in space $O(s(n))$;
- P is the complexity class $\bigcup_{c\in\mathcal{N}}$ DTIME$[n^c]$;
- NP is the complexity class $\bigcup_{c\in\mathcal{N}}$ NTIME$[n^c]$;
- PSPACE is the complexity class $\bigcup_{c\in\mathcal{N}}$ DSPACE$[n^c]$;
- ATIME$[s(n), t(n)]$ is the class of languages accepted by alternating Turing machines operating simultaneously in time $t(n)$ and space $s(n)$.

EXAMPLE 24.9:

In Example 24.5 we demonstrated how the language $L = \{xx | x \in \{0, 1\}^*\}$ can be decided by a Turing machine. We gave a two-head, one-tape TM running in time $3n/2$, and it is easy to design a two-tape, one-head-per-tape TM that executes the same strategy in time $2n$. Theorem 24.5 says that by using a larger tape alphabet, one can push the time down to $(1 + \epsilon)n$ for any fixed $\epsilon > 0$. However, our basic one-tape, one-head Turing machine model can do no better than time on the order of n^2.

EXAMPLE 24.10:

Any multitape, multihead Turing machine, not just the one accepting L in the last example, can be simulated by our basic one-tape, one-head model in at most the square of the original's running time. For example, a two-tape machine M with tape alphabet Γ can be simulated by a one-tape machine M' with a Γ' large enough to encode all pairs of characters over Γ. Then M' can regard its single tape as having two "tracks," one for each tape of M. M' also needs to mark the locations of the two heads of M, one on each track—this can be facilitated by adding more characters to Γ'. Now to simulate one step of M, the one-tape machine M' must use its single head to update the computation at the two locations with the two head markers. If M runs in time $t(n)$, then the two head markers cannot be more than $t(n)$ cells apart. Thus to simulate each step by M, M' moves its head for at most $t(n)$ distance. Hence M' runs in time at most $t(n)^2$.

The simulation idea in Example 24.10 does not work if M uses two- or higher-dimensional tapes, or a more-general "random-access" storage (see the next section). However, a one-tape M' can be built to simulate it whose running time is still no worse than a polynomial in the time of M. It is important to note that our basic one-tape deterministic Turing machine is known to simulate all of the extended models we offer above and below—except nondeterministic and alternating TMs—with at most polynomial slowdown. This is a key point in taking the class P, defined as above in terms of Turing machine time, as the benchmark for which languages and functions are considered *feasibly computable* for general computation. (See Chapter 27 for more discussion of this point.) Polynomial time for nondeterministic Turing machines defines the class NP. Interestingly, polynomial space for nondeterministic TMs does equal polynomial space for deterministic TMs [34], and polynomial *time* for *alternating* TMs equals the same class, namely PSPACE.

EXAMPLE 24.11:

All of the basic arithmetical operations—plus, minus, times, and division—belong to P. Given two n-digit integers x and y, we can easily add or subtract them in $O(n)$ steps. We can multiply or divide them in $O(n^2)$ steps using the standard algorithms learned in school. Actually, by grouping blocks of digits in x and y and using some clever tricks one can bring the time down to $O(n^{1+\epsilon})$ bit-operations for any desired fixed $\epsilon > 0$, and the asymptotically fastest method known takes time $O(n \log n \operatorname{loglog} n)$ [37]. Computing x^y is technically not in P because the sheer length of the output may be exponential in n, but if we measure time as a function of output length as well as input length, it is in P. However, the operation of *factoring* a number into primes, which is a kind of inverse of multiplication, is commonly believed *not* to belong to P. The language associated to the factoring function (refer to "L_f" before Theorem 24.4 above) does belong to NP.

EXAMPLE 24.12:

There are many other important problems in NP that are not known to be in P. For example, consider the following "King Arthur" problem, which is equivalent to the problem called HAMILTONIAN CIRCUIT in Chapter 28. King Arthur plans to have a round table meeting. By one historical account he had 150 knights, so let $n = 150$. It is known that some pairs of knights hate each other, and some do not. King Arthur's problem is to arrange the knights around a round table so that no pair of knights who sit side by side hate each other. King Arthur can solve this problem by enumerating all possible permutations of n knights. But even at $n = 150$, there are 150! permutations. All the computers in the whole world, even if they started a thousand years ago and worked non-stop, would still be going on today, having examined only a tiny fraction of the 150! permutations. However, this problem is in NP because a nondeterministic Turing machine can just guess an arrangement and verify the correctness of the solution—by checking if any two neighboring knights are enemies—in polynomial time. It is currently unknown if every problem in NP is also in P. This problem has a special property—namely, if it is in P then every problem in NP is also in P.

The following relationships are true:

$$\mathrm{P} \subseteq \mathrm{NP} \subseteq \mathrm{PSPACE}\ .$$

Whether or not either of the above inclusions is proper is one of the most fundamental open questions in computer science and mathematics. Research in computational complexity theory centers around these questions. The first step in working on these questions is to identify the hardest problems in NP or PSPACE.

DEFINITION 24.6 Given two languages A and B over an alphabet Σ, a function $f : \Sigma^* \to \Sigma^*$ is

called a **polynomial-time many-one reduction** from A to B if

(a) f is polynomial-time computable, and
(b) for all $x \in \Sigma^*$, $x \in A$ if and only if $f(x) \in B$.

One also writes $A \leq_m^p B$.

The only change from the definition of "many-one reduction" at the end of "Oracle Turing Machines" is that we have inserted "polynomial-time" before "computable." There is also a polynomial-time version of Turing reducibility as defined there, which gets the notation $A \leq_T^p B$.

DEFINITION 24.7 A language B is called **NP-complete** if

1. B is in NP;
2. for every language $A \in$ NP, $A \leq_m^p B$.

In this definition, if only the second item holds, then we say the language B is *NP-hard.* (Curiously, while "NP-complete" is always taken to refer by default to polynomial-time many-one **reductions,** "NP-hard" is usually extended to refer to polynomial-time **Turing reductions.**) The upshot is that if a language B is NP-complete and B is in P, then NP = P. An **NP-complete language** is in this sense a hardest language in the class NP. Working independently, Cook [7] and Levin [24] introduced NP-completeness, and Karp [16] further demonstrated its importance. PSPACE and other classes also have complete languages.

Chapters 27 and 28 of this *Handbook* develop the topics of this section in much greater detail. We also refer the interested reader to the textbooks [10, 15, 25, 41, 42].

24.4 Other Computing Models

Over the years, many alternative computing models have been proposed. Under reasonable definitions of running time for these models, they can all be simulated by Turing machines with at most a polynomial slow-down. The reference [40] provides a nice survey of various computing models other than Turing machines. We will discuss a few such alternatives very briefly and refer our readers to Chapter 45 and [40] for more information.

Random Access Machines

The *random access machine* [8] consists of a finite control where a program is stored, several arithmetic registers, and an infinite collection of memory registers $R[1], R[2], \ldots$. All registers have an unbounded word length. The basic instructions for the program are LOAD, STORE, ADD, MUL, GOTO, and conditional-branch instructions. The LOAD and STORE commands can use indirect addressing. Compared to Turing machines, this appears to be a closer but more complicated approximation of modern computers. There are two standard ways for measuring time complexity of the model:

- The *Unit-cost RAM:* Here each instruction takes one unit of time, no matter how big the operands are. This measure is convenient for analyzing many algorithms.
- The *Log-cost RAM:* Here each instruction is charged for the sum of the lengths of all data manipulated by the instruction. Equivalently, each use of an integer i is charged $\log i$ time units, since $\log i$ is approximately the length of i. This is a more realistic model, but sometimes less convenient to use.

Log-cost RAMs and Turing machines can simulate each other with polynomial overheads. The unit-cost assumption becomes unrealistic when the MUL instruction is used repeatedly to form exponentially large

numbers. Taking MUL out, however, makes the unit-cost RAM polynomially equivalent to the Turing machine as well.

Pointer Machines

Pointer machines were introduced by [22] in 1958 and in modified form by Schönhage in the 1970s. Schönhage called his form the "storage modification machine" [36], and both forms are sometimes named for their authors. We informally describe Schönhage's form here. A pointer machine is similar to a RAM, but instead of having an unbounded array of registers for its memory structure, it has modifiable pointer links that form a Δ-*structure.* A Δ-structure S, for a finite alphabet Δ of size k, is a finite directed graph in which each node has k out-edges labeled by the k symbols in Δ. Every node also has a cell holding an integer, as with a RAM. At every step of the computation, one node of S is distinguished as the *center,* which acts as a starting point for addressing. A word $w \in \Delta^*$ addresses the cell of the node formed by following the path of pointer links selected by the successive characters in w. Besides having all the RAM instructions, the pointer machine has various instructions and rules for moving its center and redirecting pointer links, thus modifying the storage structure. Under the log-cost criterion, pointer machines are polynomially equivalent to RAMs and Turing machines. There are many interesting studies on the precise efficiency of the simulations among these models, and we refer to the reader to the survey [40] as a center for further pointers on them.

Circuits and Nonuniform Models

A *Boolean circuit* is a finite, labeled, directed acyclic graph. Input nodes are nodes without ancestors; they are labeled with input variables $x_1, \ldots, x_n$. The internal nodes are labeled with functions from a finite set of Boolean operations such as {AND, OR, NOT} or {NAND}. The number of ancestors of an internal node is precisely the number of arguments of the Boolean function that the node is labeled with. A node without successors is an output node. The circuit is naturally evaluated from input to output: at each node the function labeling the node is evaluated using the results of its ancestors as arguments. Two cost measures for the circuit model are

- *depth:* the length of a longest path from an input node to an output node.
- *size:* the number of nodes in the circuit.

These measures are applied to a family $\{C_n \mid n \geq 1\}$ of circuits for a particular problem, where C_n solves the problem of size n. Subject to the *uniformity condition* that the layout of C_n be computable given n (in time polynomial in n), circuits are (polynomially) equivalent to Turing machines. Chapters 27 and 45 give full presentations of circuit complexity, while [40, 17] have more details and pointers to the literature.

24.5 Defining Terms

Alternating turing machine: A generalization of a nondeterministic Turing machine. In the latter, every state can be called an existential state since the machine accepts if one of the possible moves leads to acceptance. In an alternating Turing machine there are also universal states, from which the machine accepts only if all possible moves out of that state lead to acceptance.

Algorithm: A finite sequence of instructions that is supposed to solve a particular problem.

Complexity class NP: The class of languages that can be accepted by a nondeterministic Turing machine in polynomial time.

Complexity class P: The class of languages that can be accepted by a deterministic Turing machine in polynomial time.

Complexity class PSPACE: The class of languages that can be accepted by a Turing machine in polynomial space.

Computable function: A function that can be computed by an algorithm—equivalently, by a Turing machine.

Decidable problem/language: A problem that can be decided by an algorithm—equivalently, whose associated language is accepted by a Turing machine that halts for all inputs.

Deterministic: Permitting at most one next move at any step in a computation.

Finite automaton or finite-state machine: A restricted Turing machine where the head is read-only and shifts only from left to right.

(Formal) language: A set of strings over some fixed alphabet.

Halting Problem: The problem of deciding whether a given program (or Turing machine) halts on a given input.

Many-one reduction: A reduction that maps an instance of one problem into an equivalent instance of another problem.

Nondeterministic: Permitting more than one choice of next move at some step in a computation.

NP-complete language: A language in NP such that every language in NP can be reduced to it in polynomial time.

Oracle Turing machine: A Turing machine with an extra oracle tape and three extra states $q_?$, q_y, q_n. When the machine enters $q_?$, control goes to state q_y if the oracle tape content is in the oracle set; otherwise control goes to state q_n.

Partial recursive function: A partial function computed by a Turing machine that need not halt for all inputs.

Partially decidable problem: One whose associated language is recursively enumerable. Equivalently, there exists a program that halts and outputs 1 for every instance having a yes answer, but is allowed not to halt or to halt and output 0 for every instance with a no answer.

Polynomial time reduction: A reduction computable in polynomial time.

Program: A sequence of instructions that can be executed, such as the code of a Turing machine or a sequence of RAM instructions.

Pushdown automaton: A restricted Turing machine where the tape acts as a pushdown store (or a stack), with an extra one-way read-only input tape.

Recursive language: A language accepted by a Turing machine that halts for all inputs.

Recursively enumerable (r.e.) language: A language accepted by a Turing machine.

Reduction: A computable transformation of one problem into another.

Regular language: A language which can be described by some right-linear/regular grammar (or equivalently by some regular expression).

Time/space complexity: A function describing the maximum time/space required by the machine on any input of length n.

Turing machine: A simplest formal model of computation consisting a finite-state control and a semi-infinite sequential tape with a read-write head. Depending on the current state and symbol read on the tape, the machine can change its state and move the head to the left or right. Unless otherwise specified, a Turing machine is deterministic.

Turing reduction: A reduction computed by an oracle Turing machine that halts for all inputs with the oracle used in the reduction.

Uncomputable function: A function that cannot be computed by any algorithm—equivalently, not by any Turing machine.

Undecidable problem/language: A problem that cannot be decided by any algorithm—equivalently, whose associated language cannot be recognized by a Turing machine that halts for all inputs.

Universal Turing machine: A Turing machine that is capable of simulating any other Turing machine if the latter is properly encoded.

References

[1] Blum, M., A machine independent theory of complexity of recursive functions. *J. Assoc. Comput. Mach.,* 14, 322–336, 1967.

[2] Brzozowski, J. and McCluskey Jr., E., Signal flow graph techniques for sequential circuit state diagram. *IEEE Trans. on Electronic Computers,* EC-12(2), 67–76, 1963.

[3] Brzozowski, J.A. and Seger, C.-J.H., *Asynchronous Circuits,* Springer-Verlag, New York, 1994.

[4] Chandra, A.K., Kozen, D.C., and Stockmeyer, L.J., Alternation, *Journal of the Assoc. Comput. Mach.,* 28, 114–133, 1981.

[5] Chomsky, N., Three models for the description of language. *IRE Trans. on Information Theory,* 2(2), 113–124, 1956.

[6] Chomsky, N. and Miller, G., Finite-state languages. *Information and Control,* 1, 91–112, 1958.

[7] Cook, S., The complexity of theorem-proving procedures. *Proc. 3r ACM Symp. Theory of Comput.,* 151–158, 1971.

[8] Cook, S. and Reckhow, R., Time bounded random access machines. *J. Comput. Syst. Sci.,* 7, 354–375, 1973.

[9] Davis, M., What is computation? In *Mathematics Today–Twelve Informal Essays,* Steen, L., Ed., 241–259, 1980.

[10] Floyd, R.W. and Beigel, R., *The Language of Machines: an Introduction to Computability and Formal Languages,* Computer Science Press, New York, 1994.

[11] Gurari, E., *An Introduction to the Theory of Computation.* Computer Science Press, Rockville, MD, 1989.

[12] Harel, D., *Algorithmics: The spirit of Computing.* Addison-Wesley, Reading, MA, 1992.

[13] Hartmanis, J., On computational complexity and the nature of computer science. *CACM,* 37(10), 37–43, 1994.

[14] Hartmanis, J. and Stearns, R., On the computational complexity of algorithms. *Trans. Amer. Math. Soc.,* 117, 285–306, 1965.

[15] Hopcroft, J. and Ullman, J., *Introduction to Automata Theory, Languages and Computation.* Addison-Wesley, Reading, MA, 1979.

[16] Karp, R.M., 1972. Reducibility among combinatorial problems. In *Complexity of Computer Computations,* Miller, R.E. and Thatcher, J.W., Eds., Plenum Press, 85–104, 1972.

[17] Karp, R.M. and Ramachandran, V., Parallel algorithms for shared-memory machines. In *Handbook of Theoretical Computer Science,* van Leeuwen, J., Ed., Elsevier/MIT Press, 869–941, 1990.

[18] Kleene, S., Representation of events in nerve nets and finite automata. In *Automata Studies.* Princeton University Press, NJ, 3–41, 1956.

[19] Knuth, D.E., *Fundamental Algorithms,* Vol. 1 of *The Art of Computer Programming,* Addison-Wesley, Reading, MA, 1969.

[20] Knuth, D., Morris, J., and Pratt, V., Fast pattern matching in strings. *SIAM J. Comput.,* 6, 323–350, 1977.

[21] Kohavi, Z., *Switching and Finite Automata Theory,* McGraw-Hill, 1978.

[22] Kolmogorov, A. and Uspenskii, V., On the definition of an algorithm. *Uspekhi Mat. Nauk.,* 13, 3–28, 1958.

[23] Lesk, M., LEX–a lexical analyzer generator. *Technical Report 39,* Bell Laboratories, Murray Hill, NJ, 1975.

[24] Levin, L., Universal sorting problems. *Problemi Peredachi Informatsii,* 9(3), 265–266, 1973. (In Russian.)
[25] Lewis, H. and Papadimitriou, C.H., *Elements of the Theory of Computation.* Prentice-Hall, Englewood Cliffs, NJ, 1981.
[26] Li, M. and Vitányi, P., *An Introduction to Kolmogorov Complexity and Its Applications,* Springer-Verlag, 1993; 2nd edition, 1997.
[27] Lynch, N., *Distributed Algorithms,* Morgan Kaufmann, 1996.
[28] McCulloch, W. and Pitts, W., A logical calculus of ideas immanent in nervous activity. *Bull. Math. Biophysics,* 5, 115–133, 1943.
[29] Post, E., Formal reductions of the general combinatorial decision problems. *Amer. J. Math.,* 65, 197–215, 1943.
[30] Rabin, M.O., Real time computation. *Israel J. Math.,* 1(4), 203–211, 1963.
[31] Rabin, M.O. and Scott, D., Finite automata and their decision problems. *IBM J. Res. Dev.,* 3, 114–125, 1959.
[32] Robinson, R., Minsky's small universal Turing machine. *International Journal of Mathematics,* 2(5), 551–562, 1991.
[33] Ruzzo, W.L., On uniform circuit complexity. *J. Comput. System Sci.,* 22, 365–383, 1981.
[34] Savitch, J., Relationships between nondeterministic and deterministic tape complexities. *J. Comput. Syst. Sci.,* 4(2), 177–192, 1970.
[35] Searls, D., The computational linguistics of biological sequences. In *Artificial Intelligence and Molecular Biology.* Hunter, L., Ed., MIT Press, 47–120, 1993.
[36] Schönhage, A., Storage modification machines. *SIAM J. Comput.,* 9, 490–508, 1980.
[37] Schönhage, A. and Strassen, V., Schnelle Multiplikation grosser Zahlen, *Computing,* 7, 281–292, 1971.
[38] Sipser, M., *Introduction to the Theory of Computation,* 1st ed. PWS Publishing Company, Boston, MA, 1997.
[39] Turing, A., On computable numbers with an application to the Entscheidungsproblem. *Proc. London Math. Soc., series 2,* 42, 230–265, 1936.
[40] van Emde Boas, P., Machine models and simulations. In *Handbook of Theoretical Computer Science,* van Leeuwen, J., Ed., Elsevier/MIT Press, 1–66, 1990.
[41] Wood, D., *Theory of Computation,* Harper and Row, 1987.
[42] Yap, C., *Introduction to Complexity Classes.* Oxford University Press, 1997. (To appear).

Further Information

The fundamentals of the theory of computation, automata theory, and formal languages can be found in Chapters 25 through 27 and in many text books including [10, 11, 12, 15, 25, 38, 41, 42]. One central focus of research in this area is to understand the relationships between different resource complexity classes. This work is motivated in part by some major open questions about the relationships between resources (such as time and space) and the role of control mechanisms (such as nondeterminism or randomness). At the same time, new computational models are being introduced and studied. One recent model that has led to the resolution of a number of interesting problems is the *interactive proofs* (IP) model. IP is defined in terms of two Turing machines that communicate with each other. One of them has unlimited power and the other (called the *verifier*) is a probabilistic Turing machine whose time complexity is bounded by a polynomial. The study of IP has led to new ways to encrypt information as well as to the proof of some unexpected results about the difficulty of solving NP-hard problems (such as coloring, clique etc.) even approximately. See Chapter 29, Sections 3 and 5. Another new model is the *quantum Turing machine,* which can solve in polynomial time some problems such as factoring that are believed to require

exponential time on any hardware that follows the laws of "classical" (pre-quantum) physics. There are also attempts to use molecular or cell-level interactions as the basic operations of a computer.

The following annual conferences present the leading research work in computation theory: ACM Annual Symposium on Theory of Computing (STOC), IEEE Symposium on the Foundations of Computer Science (FOCS), IEEE Conference on Computational Complexity (CCC, formerly Structure in Complexity Theory), International Colloquium on Automata, Languages and Programming (ICALP), Symposium on Theoretical Aspects of Computer Science (STACS), Mathematical Foundations of Computer Science (MFCS), and Fundamentals of Computation Theory (FCT). There are many related conferences in the following areas: computational learning theory, computational geometry, algorithms, principles of distributed computing, computational biology, and database theory. In each case, specialized computational models and concrete algorithms are studied for a specific application area. There are also conferences in both pure and applied mathematics that admit topics in computation theory and complexity. We conclude with a partial list of major journals whose primary focus is in theory of computation: *Journal of the ACM, SIAM Journal on Computing, Journal of Computer and System Sciences, Information and Computation, Theory of Computing Systems* (formerly *Mathematical Systems Theory*), *Theoretical Computer Science, Computational Complexity, Journal of Complexity, Information Processing Letters, International Journal of Foundations of Computer Science,* and *Acta Informatica.*

25

Formal Grammars and Languages

Tao Jiang
McMaster University

Ming Li
University of Waterloo

Bala Ravikumar
University of Rhode Island

Kenneth W. Regan
State University of New York at Buffalo

25.1 Introduction

Formal language theory as a discipline is generally regarded as growing from the work of linguist Noam Chomsky in the 1950s, when he attempted to give a precise characterization of the structure of natural languages. His goal was to define the syntax of languages using simple and precise mathematical rules. Later it was found that the syntax of programming languages can be described using one of Chomsky's grammatical models called **context-free grammars.** Much earlier, the Norwegian mathematician Axel Thue studied sequences of binary symbols subject to interesting mathematical properties, such as not having the same substring three times in a row. His work influenced Emil Post, Stephen Kleene, and others to study the mathematical properties of strings and collections of strings.

Soon after the advent of modern electronic computers, people realized that all forms of information—whether numbers, names, pictures, or sound waves—can be represented as strings. Then collections of strings known as *languages* became central to computer science. This chapter is concerned with fundamental mathematical properties of languages and language generating systems, such as grammars. Every programming language from Fortran to Java can be precisely described by a grammar. Moreover, the grammar allows us to write a computer program (called the *syntax analyzer* in a compiler) to determine whether a string of statements is syntactically correct in the programming language. Many people would wish that natural languages such as English could be analyzed as precisely, that we could write computer programs to tell which English sentences are grammatically correct. Despite recent advances in *natural language processing*, many of which have been spurred by formal grammars and other theoretical tools, today's commercial products for grammar and style fall well short of that ideal. The main problem is that *there is no common agreement* on what are grammatically correct (English) sentences; nor has anyone yet been able to offer a grammar precise enough to propose as definitive. And style is a matter of taste(!) such as not beginning sentences with "and" or using interior exclamations. Formal languages and grammars

have many applications in other fields, including molecular biology (see [17]) and symbolic dynamics (see [14]).

In this chapter, we will present some formal systems that define families of formal languages arising in many computer science applications. Our primary focus will be on **context-free languages,** since they are most widely used to describe the syntax of programming languages. In the rest of this section, we present some basic definitions and terminology.

DEFINITION 25.1 An **alphabet** is a finite nonempty set of *symbols.* Symbols are assumed to be indivisible.

For example, an alphabet for English can consist of as few as the 26 lower-case letters $a, b, \ldots, z$, adding some punctuation symbols if sentences rather than single words will be considered. Or it may include all of the symbols on a standard North American typewriter, which together with terminal control codes yields the 128-symbol ASCII alphabet, in which much of the world's communication takes place. The new world standard is an alphabet called *UNICODE,* which is intended to provide symbols for all the world's languages—as of this writing, over 38,000 symbols have been assigned. But most important aspects of formal languages can be modeled using the simple two-letter alphabet $\{0, 1\}$, over which ASCII and UNICODE are encoded to begin with. We usually use the symbol Σ to denote an alphabet.

DEFINITION 25.2 A **string** over an alphabet Σ is a finite sequence of symbols of Σ.

The number of symbols in a string x is called its *length,* denoted by $|x|$. It is convenient to introduce a notation ϵ for the empty string, which contains no symbols at all. The length of ϵ is 0.

DEFINITION 25.3 Let $x = a_1 a_2 \cdots a_n$ and $y = b_1 b_2 \cdots b_m$ be two strings. The *concatenation* of x and y, denoted by xy, is the string $a_1 a_2 \cdots a_n b_1 b_2 \cdots b_m$.

Then for any string x, $\epsilon x = x\epsilon = x$. For any string x and integer $n \geq 0$, we use x^n to denote the string formed by sequentially concatenating n copies of x.

DEFINITION 25.4 The set of all strings over an alphabet Σ is denoted by Σ^*, and the set of all nonempty strings over Σ is denoted by Σ^+. The empty set of strings is denoted by $\emptyset$.

DEFINITION 25.5 For any alphabet Σ, a **language** over Σ is a set of strings over Σ. The members of a language are also called the *words* of the language.

EXAMPLE 25.1:

The sets $L_1 = \{01, 11, 0110\}$ and $L_2 = \{0^n 1^n | n \geq 0\}$ are two languages over the binary alphabet $\{0, 1\}$. L_1 has three words, while L_2 is infinite. The string 01 is in both languages while 11 is in L_1 but not in L_2.

Since languages are just sets, standard set operations such as union, intersection, and complementation apply to languages. It is useful to introduce two more operations for languages: *concatenation* and *Kleene closure.*

DEFINITION 25.6 Let L_1 and L_2 be two languages over Σ. The concatenation of L_1 and L_2, denoted by $L_1 L_2$, is the language $\{xy | x \in L_1, y \in L_2\}$.

DEFINITION 25.7 Let L be a language over Σ. Define $L^0 = \{\epsilon\}$ and $L^i = LL^{i-1}$ for $i \geq 1$. The *Kleene closure* of L, denoted by L^*, is the language

$$L^* = \bigcup_{i \geq 0} L^i .$$

The *positive closure* of L, denoted by L^+, is the language

$$L^+ = \bigcup_{i \geq 1} L^i .$$

In other words, the Kleene closure of a language L consists of all strings that can be formed by concatenating zero or more words from L. For example, if $L = \{0, 01\}$, then $LL = \{00, 001, 010, 0101\}$, and L^* comprises all binary strings in which every 1 is preceded by a 0. Note that concatenating zero words always gives the empty string, and that a string with no 1s in it still makes the condition on "every 1" true. L^+ has the meaning "concatenate *one* or more words from L," and satisfies the properties $L^* = L^+ \cup \{\epsilon\}$ and $L^+ = LL^*$. Furthermore, for any language L, L^* always contains ϵ, and L^+ contains ϵ if and only if L does. Also note that Σ^* is in fact the Kleene closure of the alphabet Σ when Σ is viewed as a language of words of length 1, and Σ^+ is just the positive closure of Σ.

25.2 Representation of Languages

In general a language over an alphabet Σ is a subset of Σ^*. How can we describe a language rigorously so that we know whether a given string belongs to the language or not? As shown in Example 25.1, a finite language such as L_1 can be explicitly defined by enumerating its elements. An infinite language such as L_2 cannot be exhaustively enumerated, but in the case of L_2 we were able to give a simple rule characterizing all of its members. In English, the rule is, "some number of 0s followed by an equal number of 1s." Can we find systematic methods for defining rules that characterize a wide class of languages? In the following we will introduce three such methods: **regular expressions, pattern systems,** and **grammars.** Interestingly, only the last is capable of specifying the simple rule for L_2, although the first two work for many intricate languages. The term **formal languages** refers to languages that can be described by a body of systematic rules.

Regular Expressions and Languages

Let Σ be an alphabet.

DEFINITION 25.8 The **regular expressions** over Σ and the languages they represent are defined inductively as follows.

1. The symbol $\emptyset$ is a regular expression, and represents the empty language.
2. The symbol ϵ is a regular expression, and represents the language whose only member is the empty string, namely $\{\epsilon\}$.
3. For each $c \in \Sigma$, c is a regular expression, and represents the language $\{c\}$, whose only member is the string consisting of the single character c.
4. If r and s are regular expressions representing the languages R and S, then $(r + s)$, (rs) and (r^*) are regular expressions that represent the languages $R \cup S$, RS, and R^*, respectively.

For example, $((0(0+1)^*) + ((0+1)^*0))$ is a regular expression over $\{0, 1\}$ that represents the language consisting of all binary strings that begin or end with a 0. Since the set operations union and concatenation

are both associative, and since we can stipulate that Kleene closure takes precedence over concatenation and concatenation over union, many parentheses can be omitted from regular expressions. For example, the above regular expression can be written as $0(0+1)^* + (0+1)^*0$. We will also abbreviate the expression rr^* as r^+. Let us look at a few more examples of regular expressions and the languages they represent.

EXAMPLE 25.2:

The expression $0(0 + 1)^*1$ represents the set of all strings that begin with a 0 and end with a 1.

EXAMPLE 25.3:

The expression $0 + 1 + 0(0 + 1)^*0 + 1(0 + 1)^*1$ represents the set of all nonempty binary strings that begin and end with the same bit. Note the inclusion of the strings 0 and 1 as special cases.

EXAMPLE 25.4:

The expressions 0^*, 0^*10^*, and $0^*10^*10^*$ represent the languages consisting of strings that contain no 1, exactly one 1, and exactly two 1's, respectively.

EXAMPLE 25.5:

The expressions $(0+1)^*1(0+1)^*1(0+1)^*$, $(0+1)^*10^*1(0+1)^*$, $0^*10^*1(0+1)^*$, and $(0+1)^*10^*10^*$ all represent the same set of strings that contain at least two 1's.

Two or more regular expressions that represent the same language, as in Example 25.5, are called *equivalent.* It is possible to introduce algebraic identities for regular expressions in order to construct equivalent expressions. Two such identities are $r(s + t) = rs + rt$, which says that concatenation distributes over union the same way "times" distributes over "plus" in ordinary algebra (but taking care that concatenation isn't commutative), and $r^* = (r^*)^*$. These two identities are easy to prove; the reader seeking more detail may consult [16].

EXAMPLE 25.6:

Let us construct a regular expression for the set of all strings that contain no consecutive 0s. A string in this set may begin and end with a sequence of 1s. Since there are no consecutive 0s, every 0 that is not the last symbol of the string must be followed by a 1. This gives us the expression $1^*(01^+)^*1^*(\epsilon + 0)$. It is not hard to see that the second 1^* is redundant and thus the expression can in fact be simplified to $1^*(01^+)^*(\epsilon + 0)$.

Regular expressions were first introduced by [13] for studying the properties of neural nets. The above examples illustrate that regular expressions often give very clear and concise representations of languages. The languages represented by regular expressions are called the **regular languages.** Fortunately or unfortunately, not every language is regular. For example, there are no regular expressions that represent the languages $\{0^n1^n | n \geq 1\}$ or $\{xx \mid x \in \{0, 1\}^*\}$; the latter case is proved in Section 24.2 in Chapter 24.

Pattern Languages

Another way of representing languages is to use *pattern systems* [2] (see also [12]).

DEFINITION 25.9 A **pattern system** is a triple (Σ, V, p), where Σ is the alphabet, V is the set of *variables* with $\Sigma \cap V = \emptyset$, and p is a string over $\Sigma \cup V$ called the *pattern.*

DEFINITION 25.10 The language generated by a pattern system (Σ, V, p) consists of all strings over Σ that can be obtained from p by replacing each variable in p with a string over Σ.

An example pattern system is $(\{0, 1\}, \{v_1, v_2\}, v_1 v_1 0 v_2)$. The language it generates contains all words that begin with a 0 (since v_1 can be chosen as the empty string, and v_2 as an arbitrary string), and contains some words that begin with a 1, such as 110 (by taking $v_1 = 1$, $v_2 = \epsilon$) and 101001 (by taking $v_1 = 10$, $v_2 = 1$). However, it does not contain the strings ϵ, 1, 10, 11, 100, 101, etc. The pattern system $(\{0, 1\}, \{v_1\}, v_1 v_1)$ generates the set of all strings that are the concatenation of two equal substrings, namely the set $\{xx | x \in \{0, 1\}^*\}$. The languages generated by pattern systems are called *pattern languages.*

Regular languages and pattern languages are really different. We have noted that the pattern language $\{xx | x \in \{0, 1\}^*\}$ is not a regular language, and one can prove that the set represented by the regular expression 0^*1^* is not a pattern language. Although it is easy to write an algorithm to decide whether a given string is in the language generated by a given pattern system, such an algorithm would most likely have to be very inefficient [2].

General Grammars

Perhaps the most useful and general system for representing languages is based on the formal notion of a *grammar.*

DEFINITION 25.11 A **grammar** is a quadruple (Σ, V, S, P), where

1. Σ is a finite nonempty set called the **terminal alphabet.** The elements of Σ are called the **terminals.**
2. V is a finite nonempty set disjoint from Σ. The elements of V are called the **nonterminals** or **variables.**
3. $S \in V$ is a distinguished nonterminal called the **start symbol.**
4. P is a finite set of **productions** (or **rules**) of the form

$$\alpha \rightarrow \beta$$

where $\alpha \in (\Sigma \cup V)^* V (\Sigma \cup V)^*$ and $\beta \in (\Sigma \cup V)^*$, i.e., α is a string of terminals and nonterminals containing at least one nonterminal and β is a string of terminals and nonterminals.

EXAMPLE 25.7:

Let $G_1 = (\{0, 1\}, \{S, T, O, I\}, S, P)$, where P contains the following productions

$$\begin{aligned} S &\rightarrow TO \\ S &\rightarrow OI \\ T &\rightarrow SI \\ O &\rightarrow 0 \\ I &\rightarrow 1 \end{aligned}$$

As we shall see, the grammar G_1 can be used to describe the set $\{0^n 1^n | n \geq 1\}$.

EXAMPLE 25.8:

Let $G_2 = (\{0, 1, 2\}, \{S, A\}, S, P)$, where P contains the following productions

$$\begin{aligned} S &\rightarrow 0SA2 \\ S &\rightarrow \epsilon \\ 2A &\rightarrow A2 \\ 0A &\rightarrow 01 \\ 1A &\rightarrow 11 \end{aligned}$$

This grammar G_2 can be used to describe the set $\{0^n 1^n 2^n \geq n \geq 0\}$.

EXAMPLE 25.9:

To construct a grammar G_3 to describe English sentences, one might let the alphabet Σ comprise all English *words* rather than letters. V would contain nonterminals that correspond to the structural components in an English sentence, such as <sentence>, <subject>, <predicate>, <noun>, <verb>, <article>, and so on. The start symbol would be <sentence>. Some typical productions are as follows:

$$\begin{aligned} \text{<sentence>} &\rightarrow \text{<subject><predicate>} \\ \text{<subject>} &\rightarrow \text{<noun>} \\ \text{<predicate>} &\rightarrow \text{<verb><article><noun>} \\ \text{<noun>} &\rightarrow \text{mary} \\ \text{<noun>} &\rightarrow \text{algorithm} \\ \text{<verb>} &\rightarrow \text{wrote} \\ \text{<article>} &\rightarrow \text{an} \end{aligned}$$

The rule <sentence> $\rightarrow$ <subject><predicate> models the fact that a sentence can consist of a subject phrase and a predicate phrase. The rules <noun> $\rightarrow$ mary and <noun> $\rightarrow$ algorithm mean that both "mary" and "algorithm" are possible nouns. This approach to grammar, stemming from Chomsky's work, has influenced even elementary-school teaching.

To explain how a grammar represents a language, we need the following concepts.

DEFINITION 25.12 Let (Σ, V, S, P) be a grammar. A **sentential form** of G is any string of terminals and nonterminals, i.e., a string over $\Sigma \cup V$.

DEFINITION 25.13 Let (Σ, V, S, P) be a grammar, and let γ_1, γ_2 be two sentential forms of G. We say that γ_1 **directly derives** γ_2, written $\gamma_1 \Rightarrow \gamma_2$, if $\gamma_1 = \sigma\alpha\tau$, $\gamma_2 = \sigma\beta\tau$, and $\alpha \rightarrow \beta$ is a production in P.

For example, the sentential form $00S11$ directly derives the sentential form $00OT11$ in grammar G_1, and $A2A2$ directly derives $AA22$ in grammar G_2.

DEFINITION 25.14 Let γ_1 and γ_2 be two sentential forms of a grammar G. We say that γ_1 **derives** γ_2, written $\gamma_1 \Rightarrow^* \gamma_2$, if there exists a sequence of (zero or more) sentential forms $\sigma_1, \ldots, \sigma_n$ such that

$$\gamma_1 \Rightarrow \sigma_1 \Rightarrow \cdots \Rightarrow \sigma_n \Rightarrow \gamma_2 \, .$$

The sequence $\gamma_1 \Rightarrow \sigma_1 \Rightarrow \cdots \Rightarrow \sigma_n \Rightarrow \gamma_2$ is called a **derivation** of γ_2 from γ_1.

For example, in grammar G_1, $S \Rightarrow^* 0011$ because

$$S \Rightarrow \underline{O}T \Rightarrow 0\underline{T} \Rightarrow 0S\underline{I} \Rightarrow 0\underline{S}1 \Rightarrow 0\underline{O}I1 \Rightarrow 00\underline{I}1 \Rightarrow 0011$$

and in grammar G_2, $S \Rightarrow^* 001122$ because

$$S \Rightarrow 0\underline{S}A2 \Rightarrow 00\underline{S}A2A2 \Rightarrow 00\underline{A}2A2 \Rightarrow 0012\underline{A}2 \Rightarrow 0011\underline{A}22 \Rightarrow 001122\,.$$

Here the left-hand side of the relevant production in each derivation step is underlined for clarity.

DEFINITION 25.15 Let (Σ, V, S, P) be a grammar. The language generated by G, denoted by $L(G)$, is defined as

$$L(G) = \left\{ x \,|\, x \in \Sigma^*, S \Rightarrow^* x \right\}\,.$$

The words in $L(G)$ are also called the *sentences* of $L(G)$.

Clearly, $L(G_1)$ contains all strings of the form $0^n 1^n$, $n \geq 1$, and $L(G_2)$ contains all strings of the form $0^n 1^n 2^n$, $n \geq 0$. Although only a partial definition of G_3 is given, we know that $L(G_3)$ contains sentences like "mary wrote an algorithm" and "algorithm wrote an algorithm," but does not contain strings like "an wrote algorithm."

Formal grammars were introduced as such by [15], and had antecedents in work by Thue and others. However, the study of their rigorous use in describing formal (and natural) languages did not begin until the mid-1950s [3]. In the next section, we consider various restrictions on the form of productions in a grammar, and see how these restrictions can affect its power to represent languages. In particular, we will show that regular languages and pattern languages can all be generated by grammars under different restrictions.

25.3 Hierarchy of Grammars

Grammars can be divided into four classes by gradually increasing the restrictions on the form of the productions. Such a classification is due to Chomsky [3, 4] and is called the *Chomsky hierarchy.*

DEFINITION 25.16 Let $G = (\Sigma, V, S, P)$ be a grammar.

1. G is also called a **Type**-0 grammar or an **unrestricted** grammar.
2. G is a **Type-1** or **context-sensitive grammar** if each production $\alpha \rightarrow \beta$ in P satisfies $|\alpha| \leq |\beta|$. By "special dispensation," we also allow a Type-1 grammar to have the production $S \rightarrow \epsilon$, provided S does not appear on the right-hand side of any production.
3. G is a **Type-2** or **context-free** grammar if each production $\alpha \rightarrow \beta$ in P satisfies $|\alpha| = 1$; i.e., α is a single nonterminal.
4. G is a **Type**-3 or **right-linear** or **regular** grammar if each production has one of the following three forms:
$$A \rightarrow cB, \quad A \rightarrow c, \quad A \rightarrow \epsilon\,,$$
where A, B are nonterminals (with $B = A$ allowed) and c is a terminal.

The language generated by a Type-i grammar is called a Type-i language, $i = 0, 1, 2, 3$. A Type-1 language is also called a **context-sensitive language** (CSL), and a Type-2 language is also called a **context-free language** (CFL). The "special dispensation" allows a CSL to contain ϵ, and thus allows one to say that

every CFL is also a CSL. Many sources allow "right-linear" grammars to have productions of the form $A \to xB$, where x is any string of terminals, and/or exclude one of the forms $A \to c$, $A \to \epsilon$ from their definition of "regular" grammar (perhaps allowing $S \to \epsilon$ in the latter case). Regardless of the choice of definitions, every Type-3 grammar generates a regular language, and every regular language has a Type-3 grammar; we have proved this using finite automata in Chapter 24. Stated in other words:

THEOREM 25.1 *The class of Type-3 languages and the class of regular languages are equal.*

The grammars G_1 and G_3 given in the last section are context-free and the grammar G_2 is context-sensitive. Now we give some examples of unrestricted and right-linear grammars.

EXAMPLE 25.10:

Let $G_4 = (\{0, 1\}, \{S, A, O, I, T\}, S, P)$, where P contains

$$\begin{array}{rcl@{\qquad}rcl} S & \to & AT & & & \\ A & \to & 0AO & A & \to & 1AI \\ O0 & \to & 0O & O1 & \to & 1O \\ I0 & \to & 0I & I1 & \to & 1I \\ OT & \to & 0T & IT & \to & 1T \\ A & \to & \epsilon & T & \to & \epsilon \end{array}$$

Then G_4 generates the set $\{xx | x \in \{0, 1\}^*\}$. To understand how this grammar works, think of the nonterminal O as saying, "I must ensure that the right half gets a terminal 0 in the same place as the terminal 0 in the production $A \to 0AO$ that introduced me." The nonterminal I eventually forces the precise placement of a terminal 1 in the right-hand side in the same way. The nonterminal T makes sure that O and I place their 0 and 1 on the right-hand side rather than prematurely. Only after every O and I has moved right past any earlier-formed terminals 0 and 1 and been eliminated "in the context of" T, and the production $A \to \epsilon$ is used to signal that no additional O or I will be introduced, can the endmarker T be dispensed with via $T \to \epsilon$. For example, we can derive the word 0101 from S as follows:

$$S \Rightarrow \underline{A}T \Rightarrow 0\underline{A}OT \Rightarrow 01\underline{A}IOT \Rightarrow 01I\underline{OT} \Rightarrow 01\underline{I0}T \Rightarrow 010\underline{IT} \Rightarrow 0101\underline{T} \Rightarrow 0101 .$$

Only the productions $A \to \epsilon$ and $T \to \epsilon$ prevent this grammar from being Type-1. The interested reader is challenged to write a Type-1 grammar for this language.

EXAMPLE 25.11:

We give a right-linear grammar G_5 to generate the language represented by the regular expression in Example 25.3, i.e., the set of all nonempty binary strings beginning and ending with the same bit. Let $G_5 = (\{0, 1\}, \{S, O, I\}, S, P)$, where P contains

$$\begin{array}{rcl@{\qquad}rcl} S & \to & 0O & S & \to & 1I \\ S & \to & 0 & S & \to & 1 \\ O & \to & 0O & O & \to & 1O \\ I & \to & 0I & I & \to & 1I \\ O & \to & 0 & I & \to & 1 \end{array}$$

Here O means to remember that the last bit must be a 0, and 1 similarly forces the last bit to be a 1. Note again how the grammar treats the words 0 and 1 as special cases.

Every regular grammar is a context-free grammar, but not every context-free grammar is context-sensitive. However, every context-free grammar G can be transformed into an equivalent one in which every production has the form $A \to BC$ or $A \to c$, where A, B, and C are (possibly identical) variables, and c is a terminal. If the empty string is in $L(G)$, then we can arrange to include $S \to \epsilon$ under the same "special dispensation" as for CSLs. This form is called **Chomsky normal form** [4], where it was used to prove the case $i = 1$ of the next theorem. The grammar G_1 in the last section is an example of a context-free grammar in Chomsky normal form.

THEOREM 25.2 *For each $i = 0, 1, 2$, the class of Type-i languages properly contains the class of Type-$(i+1)$ languages.*

The containments are clear from the above remarks. For the proper containments, we have already seen that $\{0^n 1^n | n \geq 0\}$ is a Type-2 language that is not regular, and Chapter 26 will show that the language of the Halting Problem is Type-0 but not Type-1. One can prove by a technique called "pumping" that the Type-1 languages $\{0^n 1^n 2^n | n \geq 0\}$ and $\{xx | x \in \{0, 1\}^*\}$ are not Type-2. See [11] for this, and for a presentation of the algorithm for converting a context-free grammar into Chomsky normal form.

The four classes of languages in the Chomsky hierarchy have also been completely characterized in terms of Turing machines (see Chapter 24) and natural restrictions on them. We mention this here to make the point that these characterizations show that these classes capture fundamental properties of computation, not just of formal languages. A *linear bounded automaton* is a possibly nondeterministic Turing machine that on any input x uses only the cells initially occupied by x, except for one visit to the blank cell immediately to the right of x (which is the initially scanned cell if $x = \epsilon$). Pushdown automata may also be nondeterministic and were likewise introduced in Chapter 24.

THEOREM 25.3

(a) The class of Type-0 languages equals the class of languages accepted by Turing machines.
(b) The class of Type-1 languages equals the class of languages accepted by linear bounded automata.
(c) The class of Type-2 languages equals the class of languages accepted by pushdown automata.
(d) The class of Type-3 languages equals the class of languages accepted by finite automata.

PROOF (a) Given a Type-0 grammar G, one can build a nondeterministic Turing machine M that accepts $L(G)$ by having M first write the start symbol S of G on a second tape. M always nondeterministically chooses a production and chooses a place (if any) on its second tape where it can be applied. If and when the second tape becomes an all-terminal string, M compares it to its input, and if they match, M accepts. Then $L(M) = L(G)$, and by Theorem 24.4 of Chapter 24, M can be converted into an equivalent deterministic single-tape Turing machine.

For the reverse simulation of a TM by a grammar we give full details. Given any TM M_0, we may modify M_0 into an equivalent TM $M = (Q, \Sigma, \Gamma, \delta, B, q_0, q_f)$ that has the following five properties: (i) M never writes a blank; (ii) M when reading a blank always converts it to a nonblank symbol on the current step; (iii) M begins with a transition from q_0 that overwrites the first input cell (remembering what it was) by a special symbol $\wedge$ that is never altered; (iv) M never reenters state q_0 or moves left of $\wedge$; and (v) whenever M is about to accept, M moves left to the $\wedge$, where it executes an instruction that moves right and enters a distinguished state q_e. In state q_e it overwrites any nonblank character by a special new symbol # and moves right; when it hits the blank after having #-ed out the rightmost nonblank symbol on its tape, M finally goes to q_f and accepts.

Given M with these properties, take $V = \{S, A, \} \cup (Q \times \Gamma) \cup (\Gamma \setminus \Sigma)$. A single symbol in $Q \times \Gamma$ is written using square brackets; e.g., $[q, c]$ means that M is in state q scanning character c. The grammar

G has the following productions, which intuitively can simulate any accepting computation by M in reverse:

(1) $S \to \wedge S_0$; $\quad S_0 \to \#S_0 \mid [q_e, \#]$;
(2) $[r, d] \to [q, c]$, for all instructions $(q, c, d, r) \in \delta$ with $q, r \in Q$ and $c, d \in \Gamma$;
(3) $c[r, B] \to [q, c]A$, for all $(q, c, R, r) \in \delta$;
(4) $c[r, d] \to [q, c]d$, for all $(q, c, R, r) \in \delta$ and $d \in \Gamma, d \neq B$;
(5) $[r, d]c \to d[q, c]$, for all $(q, c, L, r) \in \delta$ and $d \in \Gamma, d \neq B$;
(6) $[q_0, c] \to c$ for all $c \in \Sigma$, and
(7) $A \to \epsilon$.

For all $x \in L(M)$, G can generate x by first using the productions in (1) to lay down a # for every cell *used* during the computation, using the productions (2)–(5) to simulate the computation in reverse, using (6) to restore the first bit of x (blank if $x = \epsilon$) one step after having eliminated the nonterminal $\wedge$, and using (7) to erase each A marking an initially-blank cell that M used. Conversely, the only way G can eliminate $\wedge$ and reach an all-terminal string is by winding back an accepting computation of M all the way to state q_0 scanning the first cell. Hence $L(G) = L(M)$.

(b) If the given TM M_0 is a linear bounded automaton, then we can patch the last construction to eliminate the productions in (3) and (7), yielding a context-sensitive grammar G. To do this, we need to make M postpone its one allowed visit to the blank cell after the input until the last step of an accepting computation. To do this, we make M nondeterministically guess which bit of its input x is the last one, and overwrite it by an immutable right endmarker \$ the same way it did with $\wedge$ on the left. Then we arrange that from state q_e, M will accept only if it sees a blank immediately to the right of the \$, meaning that its initial guess delimited exactly the true input x. (Technically this needs another state q'_e.) Now M never even scans a blank in the middle of an accepting computation, and we can delete the productions in (3) as well as (7). Moreover, if M_0 accepts ϵ, we can add the production $S \to \epsilon$ allowed by the "special dispensation" for context-sensitive grammars above.

Going the other way, if the grammar G in the first paragraph of this proof is context-sensitive, then the resulting TM M uses only $O(n)$ space, and can be converted to an equivalent linear bounded automaton by Theorem 24.5 of Chapter 24.

(c) Given a context-free grammar G, we may assume that G is in Chomsky normal form. We can build a nondeterministic PDA M whose initial moves lay down a bottom-of-stack marker $\wedge$ and the start symbol S of G, and go to a "central" state q. For every production of the form $A \to BC$ in G, M has moves that pop the stack if A is uppermost and push C and then B, returning to state q. For every production of the form $A \to c$, M can pop an uppermost A from its stack if the currently-scanned input symbol is c; then it moves its input head right. If G has the production $S \to \epsilon$ as a special case, then M can pop the initial S. A computation path accepts if and only if the stack gets down to $\wedge$ precisely when M reaches the blank at the end of its input x. Then accepting paths of M on an input x are in 1–1 correspondence with **leftmost derivations** (see below) of x in G, so $L(M) = L(G)$.

Going from a PDA M to an equivalent CFG G is much trickier, and is covered well in [11].

(d) This has been proved in Chapter 24, Theorem 24.2.

Since $\{xx | x \in \{0, 1\}^*\}$ is a pattern language, we know from discussions above that the class of pattern languages is not contained in the class of context-free languages. It is contained in the class of context-sensitive languages, however.

THEOREM 25.4 *Every pattern language is context-sensitive.*

This was proved by showing that every pattern language is accepted by a linear bounded automaton [2], whereupon it is a corollary of Theorem 25.3(b).

Given a class of languages, we are often interested in the so-called *closure properties* of the class.

DEFINITION 25.17 A class of languages is said to be *closed under* a particular operation (such as union, intersection, complementation, concatenation, or Kleene closure) if every application of the operation on language(s) of the class yields a language of the class.

Closure properties are often useful in constructing new languages from existing languages, and for proving many theoretical properties of languages and grammars. The closure properties of the four types of languages in the Chomsky hierarchy are summarized below. Proofs may be found in [8, 10, 11], the closure of the CSLs under complementation is the famous Immerman–Szelepcsényi Theorem, which is given as Theorem 27.4(c) in Chapter 27.

THEOREM 25.5

1. *The class of Type-0 languages is closed under union, intersection, concatenation, and Kleene closure, but not under complementation.*
2. *The class of context-free languages is closed under union, concatenation and Kleene closure, but not under intersection or complementation.*
3. *The classes of context-sensitive and regular languages are closed under all of the five operations.*

For example, let $L_1 = \{0^m 1^n 2^p | m = n\}$, $L_2 = \{0^m 1^n 2^p | n = p\}$, and $L_3 = \{0^m 1^n 2^p | m = n \text{ or } n = p\}$. Now L_1 is the concatenation of the context-free languages $\{0^n 1^n | n \geq 0\}$ and 2^*, so L_1 is context-free. Similarly L_2 is context-free. Since $L_3 = L_1 \cup L_2$, L_3 is context-free. However, intersecting L_1 with L_2 gives the language $\{0^m 1^n 2^p | m = n = p\}$, which is not context-free.

We will look at context-free grammars more closely in the next section and introduce the concepts of parsing and ambiguity.

25.4 Context-Free Grammars and Parsing

From a practical point of view, for each grammar $G = (\Sigma, V, S, P)$ representing some language, the following two problems are important:

1. **Membership problem:** Given a string over Σ, does it belong to $L(G)$?
2. **Parsing problem:** Given a string in $L(G)$, how can it be derived from S?

The importance of the membership problem is quite obvious—given an English sentence or computer program, we wish to know if it is grammatically correct or has the right format. Solving the membership problem for context-free grammars is an integral step in the *lexical analysis* of computer programs, namely the stage of decomposing each statement into *tokens,* prior to fully parsing the program. For this reason, the membership problem is also often referred to as lexical analysis (cf. [6]). Parsing is important because a derivation usually brings out the "meaning" of the string. For example, in the case of a Pascal program, a derivation of the program in the Pascal grammar tells the compiler how the program should be executed. The following theorem qualifies the decidability of the membership problem for the four classes of grammars in the Chomsky hierarchy. Proofs of the first assertion can be found in [4, 10, 11], while the second assertion is treated below. Decidability and time complexity were defined in Chapter 24.

THEOREM 25.6 *The membership problem for Type-0 grammars is undecidable in general, but it is decidable given any context-sensitive grammar. For context-free grammars the problem is decidable in polynomial time, and for regular grammars, linear time.*

Since context-free grammars play a very important role in describing computer programming languages, we discuss the membership and parsing problems for context-free grammars in more detail in this and the next section. First, let us look at another example of a context-free grammar. For convenience, let us abbreviate a set of productions

$$A \to \alpha_1, \ldots, A \to \alpha_n$$

with the same left-hand side nonterminal as

$$A \to \alpha_1 | \ldots | \alpha_n \ .$$

EXAMPLE 25.12:

We construct a context-free grammar G_6 for the set of all valid real-number literals in Pascal. In general, a real constant in Pascal has one of the following forms:

$$m.n, \quad m\mathbf{e}q, \quad m.n\mathbf{e}q \ ,$$

where m, q are signed or unsigned integers and n is an unsigned integer. Let Σ comprise the digits 0–9, the decimal point '.', the $+$ and $-$ signs, and the **e** for scientific notation. Let the set V of variables be $\{S, M, N, D\}$ and let the set P of the productions be

$$\begin{aligned} S &\to M.N | M\mathbf{e}M | M.N\mathbf{e}M \\ M &\to N | +N | -N \\ N &\to DN | D \\ D &\to 0|1|2|3|4|5|7|8|9 \end{aligned}$$

Then the grammar generates all valid Pascal real values (allowing redundant leading 0s). For instance, the value 12.3**e**-4 can be derived via

$$\begin{aligned} &S \Rightarrow \underline{M}.N\mathbf{e}M \Rightarrow \underline{N}.N\mathbf{e}M \Rightarrow \underline{D}N.N\mathbf{e}M \Rightarrow 1\underline{N}.N\mathbf{e}M \Rightarrow 1\underline{D}.N\mathbf{e}M \Rightarrow \\ &12.\underline{N}\mathbf{e}M \Rightarrow 12.\underline{D}\mathbf{e}M \Rightarrow 12.3\mathbf{e}\underline{M} \Rightarrow 12.3\mathbf{e} - \underline{N} \Rightarrow 12.3\mathbf{e}\text{-}\underline{D} \Rightarrow 12.3\mathbf{e}\text{-}4 \end{aligned}$$

Perhaps the most natural representation of derivations in a context-free grammar is a **derivation tree** or a **parse tree.** Every leaf of such a tree corresponds to a terminal (or to ϵ), and every internal node corresponds to a nonterminal. If A is an internal node with children $B_1, \ldots, B_n$, ordered from left to right, then $A \to B_1 \cdots B_n$ must be a production. The concatenation of all leaves from left to right yields the string being derived. For example, the derivation tree corresponding to the above derivation of 12.3**e**-4 is given in Fig. 25.1. Such a tree also makes possible the extraction of the parts 12, 3 and -4, which are useful in the storage of the real value in a computer memory.

DEFINITION 25.18 A context-free grammar G is **ambiguous** if there is a string $x \in L(G)$ that has two distinct derivation trees. Otherwise G is **unambiguous.**

Unambiguity is a very desirable property because it promises a unique interpretation of each sentence in the language. It is not hard to see that the grammar G_6 for Pascal real values and the grammar G_1 defined in Example 25.7 are both unambiguous. The following example shows an ambiguous grammar.

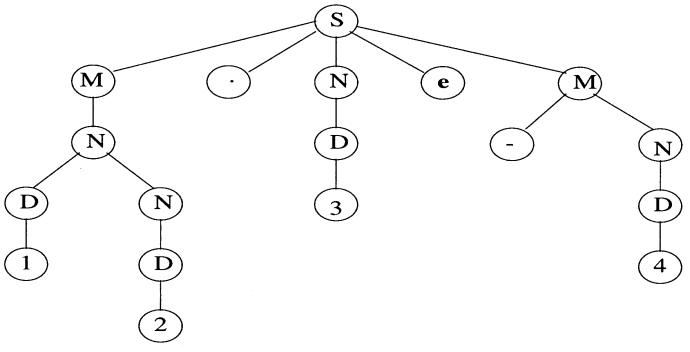


FIGURE 25.1 The derivation tree for $12.3\mathbf{e} - 4$.

EXAMPLE 25.13:

Consider a grammar G_7 for all valid arithmetic expressions that are composed of unsigned positive integers and symbols $+, *, (,)$. For convenience, let us use the symbol **n** to denote any unsigned positive integer—it is treated as a terminal. This grammar has the productions

$$\begin{aligned} S &\rightarrow T + S | S + T | T \\ T &\rightarrow F * T | T * F | F \\ F &\rightarrow \mathbf{n} | (S) \end{aligned}$$

Two possible different derivation trees for the expression $1 + 2 * 3 + 4$ are shown in Fig. 25.2. Thus G_7 is ambiguous. The left tree means that the first addition should be done before the second addition, while the right tree says the opposite.

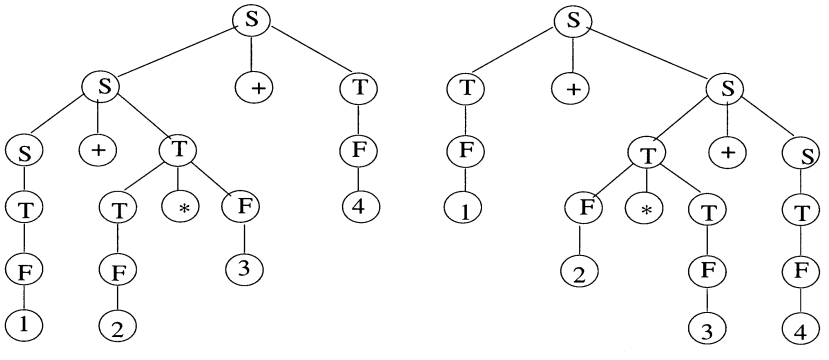


FIGURE 25.2 Different derivation trees for the expression $1 + 2 * 3 + 4$.

Although in the above example different derivations/interpretations of any expression always result in the same value because the operations addition and multiplication are associative, there are situations where the difference in the derivation can affect the final outcome. Actually, the grammar G_7 can be made unambiguous by removing the redundant productions $S \rightarrow T + S$ and $T \rightarrow F * T$. This corresponds to the convention that a sequence of consecutive additions or multiplications is always evaluated from left to right. Deleting the two productions does not change the language of strings generated by G_7, but it does fix unique interpretations of those strings.

It is worth noting that there are context-free languages that cannot be generated by any unambiguous context-free grammar. Such languages are said to be *inherently ambiguous.* An example taken from [11]

(where this fact is proved) is

$$\left\{ 0^m 1^m 2^n 3^n \,\middle|\, m, n > 0 \right\} \cup \left\{ 0^m 1^n 2^n 3^m \,\middle|\, m, n > 0 \right\} .$$

The reason is that every context-free grammar G must yield two parse trees for some strings of the form $x = 0^n 1^n 2^n 3^n$, where one tree intuitively expresses that x is a member of the first set of the union, and the other tree expresses that x is in the second set.

We end this section by presenting an efficient algorithm for the membership problem for context-free grammars, following the treatment in [11]. The algorithm is due to Cocke, Younger, and Kasami, and is often called the CYK algorithm. Let $G = (\Sigma, V, S, P)$ be a context-free grammar in Chomsky normal form.

EXAMPLE 25.14:

If we use the algorithm in [11] to convert the grammar G_7 from Example 25.13 into Chomsky normal form, we are led to introduce new "alias variables" A, B, C, D for the operators and parentheses, and "helper variables" S_1, T_1, T_2, F_1, F_2 to break up the productions in G_7 with right-hand sides of length > 2 into length-2 pieces. The resulting grammar is:

$$\begin{array}{rcl}
S & \rightarrow & T_1 S | S T_2 | F_1 T | T F_2 | C S_1 | \mathbf{n} \\
T_1 & \rightarrow & T A \\
T_2 & \rightarrow & A T \\
T & \rightarrow & F_1 T | T F_2 | C S_1 | \mathbf{n} \\
F_1 & \rightarrow & F B \\
F_2 & \rightarrow & B F \\
F & \rightarrow & \mathbf{n} | C S_1 \\
S_1 & \rightarrow & S D \\
A & \rightarrow & + \\
B & \rightarrow & * \\
C & \rightarrow & (\\
D & \rightarrow &)
\end{array}$$

While this grammar is much less intuitive to read than G_7, having it in Chomsky normal form facilitates the description and operation of the CYK algorithm.

Now suppose that $x = a_1 \cdots a_n$ is a string of n terminals that we want to test for membership in $L(G)$. The basic idea of the CYK algorithm is a form of *dynamic programming.* For each pair i, j, where $1 \leq i \leq j \leq n$, define a set $X_{i,j} \subseteq V$ by

$$X_{i,j} = \left\{ A \,|\, A \Rightarrow^* a_i \cdots a_j \right\} .$$

Then $x \in L(G)$ if and only if $S \in X_{1,n}$. The sets $X_{i,j}$ can be computed inductively in ascending order of $j - i$. It is easy to figure out $X_{i,i}$ for each i since $X_{i,i} = \{A | A \rightarrow a_i \in P\}$. Suppose that we have computed all $X_{i,j}$ where $j - i < d$ for some $d > 0$. To compute a set $X_{i,j}$, where $j - i = d$, we just have to find all the nonterminals A such that there exist some nonterminals B and C satisfying $A \rightarrow BC \in P$ and for some k, $i \leq k < j$, $B \in X_{i,k}$ and $C \in X_{k+1,j}$. A rigorous description of the algorithm in a Pascal-style pseudocode is given below.

Algorithm CYK($x = a_1 \cdots a_n$)
1 **for** $i \leftarrow 1$ **to** n **do**
2 $\quad X_{i,i} \leftarrow \{A | A \rightarrow a_i \in P\}$;
3 **for** $d \leftarrow 1$ **to** $n-1$ **do**
4 $\quad$ **for** $i \leftarrow 1$ **to** $n-d$ **do**
5 $\quad\quad X_{i,i+d} \leftarrow \emptyset$;
6 $\quad$ **for** $t \leftarrow 0$ **to** $d-1$ **do**
7 $\quad\quad X_{i,i+d} \leftarrow X_{i,i+d} \cup \{A | A \rightarrow BC \in P$ for some $B \in X_{i,i+t}$ and $C \in X_{i+t+1,i+d}\}$;

Table 25.1 shows the sets $X_{i,j}$ for the grammar G_1 and the string $x = 000111$. In this run it happens that every $X_{i,j}$ is either empty or a singleton. The computation proceeds from the main diagonal toward the upper-right corner.

TABLE 25.1 An Example Execution of the CYK Algorithm

		0	0	0	1	1	1
				$j \rightarrow$			
		1	2	3	4	5	6
	1	O					S
	2		O			S	T
i	3			O	S	T	
$\downarrow$	4				I		
	5					I	
	6						I

We now analyze the asymptotic time complexity of the CYK algorithm. Step 2 is executed n times. Step 5 is executed $\Sigma_{d=1}^{n-1} n-d = (n-1)(n-1+n-(n-1))/2 = n(n-1)/2 = O(n^2)$ times. Step 7 is repeated for $\Sigma_{d=1}^{n-1} d(n-d) = O(n^3)$ times. Therefore, the algorithm requires asymptotically $O(n^3)$ time to decide the membership of a string length n in $L(G)$, for any grammar G in Chomsky normal form.

25.5 More Efficient Parsing for Context-Free Grammars

The CYK algorithm presented in the last section can be easily extended to solve the parsing problem for context-free grammars: In step 7, we also record a production $A \rightarrow BC$ and the corresponding value of t for any nonterminal A that gets added to $X_{i,i+d}$. Thus a derivation tree for x can be constructed by starting from the nonterminal S in $X_{1,n}$ and repeatedly applying the productions recorded for appropriate nonterminals in appropriate sets $X_{i,j}$. However, the cubic running time of this algorithm is generally too high for parsing applications. In practice, with compilation modules thousands of lines long, people seek grammars in other forms besides Chomsky's that permit parsing in linear or nearly linear time.

Before we present some of these forms, we discuss parsing strategies in general. Parsing algorithms fall into two basic types, called **top-down parsers** and **bottom-up parsers.** As indicated by their names, a top-down parser builds derivation trees from the top (root) to the bottom (leaves), while a bottom-up parser starts from the leaves and works up to the root. Although neither method is good for handling all context-free grammars, each provides efficient parsing for many important subclasses of the context-free grammars, including those used in most programming languages.

We will only consider unambiguous grammars. To simplify the description of the parsers, we will assume that each string to be parsed ends with a special delimiter \$ that does not appear anywhere else in the string. This assumption makes the detection of the end of the string easy in a left-to-right scan. The assumption does not put any serious restriction on the range of languages that can be parsed—the \$ is just like the end-of-file marker in a real input file. The following definition will be useful.

DEFINITION 25.19 A derivation from a sentential form to another is said to be **leftmost** (or **rightmost**) if at each step the leftmost (or rightmost, respectively) nonterminal is replaced.

For example, Example 25.14 gave a leftmost derivation of the word 12.3**e**-4 in the grammar G_6. For a given word x, leftmost derivations are in 1–1 correspondence with derivation trees, since we can find the leftmost derivation specified by a derivation tree by tracing the tree down from the root going from left to right. Rightmost derivations are likewise in 1–1 correspondence with derivation trees. Hence, in an unambiguous context-free grammar, every derivable string has a unique leftmost derivation and a unique rightmost derivation. The parsing methods considered next find one or the other.

Top-Down Parsing

An important member of the top-down parsers is the **LL parser** (see [1, 6]). Here, the first "L" means scanning the input from left to right, and the second means leftmost derivation. In other words, for any input string x, the parser intends to find the sequence of productions used in the leftmost derivation of x.

Let $G = (\Sigma, V, S, P)$ be a context-free grammar. A *parsing table* T for G has rows indexed by members of V and columns indexed by members of Σ and \$. Each entry $T[A, c]$ is either blank or contains one or more productions of the form $A \to \alpha$. Here we will suppose that G allows the construction of a parsing table T such that every non-blank entry $T[A, c]$ contains only *one* production. Then the LL parser for G is a device very similar to a pushdown automaton as described in Chapter 24. The parser has an input buffer, a pushdown stack, a parsing table, and an output stream. The input buffer contains the string to be parsed followed by the delimiter \$. The stack contains a sequence of terminals or nonterminals, with another delimiter # that marks the bottom of the stack. Initially, the input pointer points to the first symbol of the input string, and the stack contains the start nonterminal S on top of #. Figure 25.3 illustrates schematically the components of the parser. As usual, the input pointer will only move to the right, while the stack pointer is allowed to move up and down.

The parser is controlled by an algorithm that behaves as follows. At any instant of time, the algorithm considers the symbol X on top of the stack and the current input symbol c pointed by the input pointer, and makes one of the following moves.

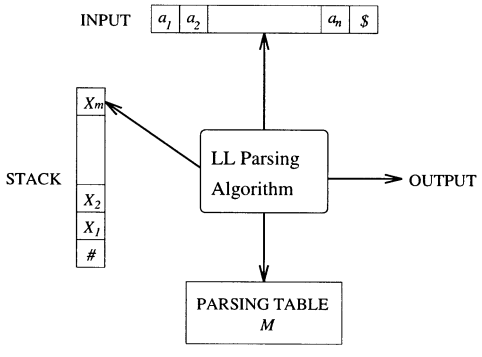


FIGURE 25.3 A schematic illustration of the LL parser.

1. If X is a nonterminal, the algorithm consults the entry $T[X, a]$ of the parsing table T. If the entry is blank, the parser halts and states that the input string x is not in the language $L(G)$. If not, the entry is a production of the form $X \rightarrow Y_1 \cdots Y_k$. Then the algorithm replaces the top stack symbol X with the string $Y_1 \cdots Y_k$ (with Y_1 on top), and outputs the production.
2. If X is a terminal, X is compared with c. If $X = c$, the algorithm pops X off the stack and shifts the input pointer to the next input symbol. Otherwise, the algorithm halts and states that $x \notin L(G)$.
3. If $X = \#$, then provided $c = \$$, the algorithm halts and declares the successful completion of parsing. Otherwise the algorithm halts and states that $x \notin L(G)$.

Intuitively, the parser reconstructs the derivation of a string $x = a_1 \cdots a_n$ as follows. Suppose that the leftmost derivation of x is

$$S = \gamma_0 \Rightarrow \gamma_1 \Rightarrow \cdots \Rightarrow \gamma_i \Rightarrow \gamma_{i+1} \Rightarrow \cdots \Rightarrow \gamma_m = x \ ,$$

where each γ_j is a sentential form. Suppose, moreover, that the derivation step $\gamma_i \Rightarrow \gamma_{i+1}$ is the result of applying a production $X \rightarrow Y_1 \cdots Y_k$. This means that $\gamma_i = \alpha X \beta$ for some string α of terminals and sentential form β. Since no subsequent derivation will change α, this string must match a leading substring $a_1 \cdots a_j$ of x for some j. In other words, $\gamma_i = a_1 \cdots a_j X \beta$ and $\gamma_{i+1} = a_1 \cdots a_j Y_1 \cdots Y_k \beta$. Suppose that the parser has successfully reconstructed the derivation steps up to γ_i. To complete the derivation, the parser must transform the tail end of γ_i into $a_{j+1} \cdots a_n$. Thus, it keeps the string $X\beta$ on the stack and repeatedly replaces the top stack symbol (i.e., replaces the leftmost nonterminal) until a_{j+1} appears on top. At this point, a_{j+1} is removed from the stack, and the remainder of the stack must be transformed to match $a_{j+2} \cdots a_n$. The procedure is repeated until all the input symbols are matched.

The following example illustrates the parsing table for a simple context-free grammar, and how the parser operates.

EXAMPLE 25.15:

Consider again the language of valid arithmetic expressions from Example 25.13, where an ambiguous grammar G_7 was given that could be made unambiguous by removing two productions. Let us remove the ambiguity in a different way. The new grammar is called G_8 and has the following productions

$$\begin{aligned} S &\rightarrow TS' \\ S' &\rightarrow +S|\epsilon \\ T &\rightarrow FT' \\ T' &\rightarrow *T|\epsilon \\ F &\rightarrow \mathbf{n}|(S) \end{aligned}$$

It is easy to see that grammar G_8 is unambiguous. A parsing table for this grammar is shown in Table 25.2. We will discuss how such a table can be constructed shortly.

To demonstrate how the parser operates, consider the input string $(\mathbf{n} + \mathbf{n}) * \mathbf{n}$. Table 25.3 shows the content of the stack, the remaining input symbols, and the output after each step. If we trace the actions of the parser carefully, we see that the sequence of productions it outputs constitutes the leftmost derivation of $(\mathbf{n} + \mathbf{n}) * \mathbf{n}$.

Now we turn to the question of how to construct an LL parser for a given grammar $G = (\Sigma, V, S, P)$. It suffices to show how to compute the entries $T[A, c]$, where $A \in V$ and $c \in \Sigma \cup \{\$\}$. We first need to introduce two functions $FIRST(\alpha)$ and $FOLLOW(A)$. The former maps a sentential form to a terminal or ϵ, and the latter maps a nonterminal to a terminal or \$.

TABLE 25.2 An LL Parsing Table for Grammar G_8

NONTER-MINAL	INPUT SYMBOL					
	n	+	*	(	)	$
S	$S \to TS'$			$S \to TS'$		
S'		$S' \to +S$			$S' \to \epsilon$	$S' \to \epsilon$
T	$T \to FT'$			$T \to FT'$		
T'		$T' \to \epsilon$	$T' \to *T$		$T' \to \epsilon$	$T' \to \epsilon$
F	$F \to \mathbf{n}$			$F \to (S)$		

TABLE 25.3 The Steps in the LL Parsing of $(\mathbf{n} + \mathbf{n}) * \mathbf{n}$

STACK	INPUT	OUTPUT
$S\#$	$(\mathbf{n} + \mathbf{n}) * \mathbf{n}\$$	
$TS'\#$	$(\mathbf{n} + \mathbf{n}) * \mathbf{n}\$$	$S \to TS'$
$FT'S'\#$	$(\mathbf{n} + \mathbf{n}) * \mathbf{n}\$$	$T \to FT'$
$(S)T'S'\#$	$(\mathbf{n} + \mathbf{n}) * \mathbf{n}\$$	$F \to (S)$
$S)T'S'\#$	$\mathbf{n} + \mathbf{n}) * \mathbf{n}\$$	
$TS')T'S'\#$	$\mathbf{n} + \mathbf{n}) * \mathbf{n}\$$	$S \to TS'$
$FT'S')T'S'\#$	$\mathbf{n} + \mathbf{n}) * \mathbf{n}\$$	$T \to FT'$
$\mathbf{n}T'S')T'S'\#$	$\mathbf{n} + \mathbf{n}) * \mathbf{n}\$$	$F \to \mathbf{n}$
$T'S')T'S'\#$	$+\mathbf{n}) * \mathbf{n}\$$	
$S')T'S'\#$	$+\mathbf{n}) * \mathbf{n}\$$	$T' \to \epsilon$
$+S)T'S'\#$	$+\mathbf{n}) * \mathbf{n}\$$	$S' \to +S$
$S)T'S'\#$	$\mathbf{n}) * \mathbf{n}\$$	
$TS')T'S'\#$	$\mathbf{n}) * \mathbf{n}\$$	$S \to TS'$
$FT'S')T'S'\#$	$\mathbf{n}) * \mathbf{n}\$$	$T \to FT'$
$\mathbf{n}T'S')T'S'\#$	$\mathbf{n}) * \mathbf{n}\$$	$F \to \mathbf{n}$
$T'S')T'S'\#$	$) * \mathbf{n}\$$	
$S')T'S'\#$	$) * \mathbf{n}\$$	$T' \to \epsilon$
$)T'S'\#$	$) * \mathbf{n}\$$	$T' \to \epsilon$
$T'S'\#$	$*\mathbf{n}\$$	
$*TS'\#$	$*\mathbf{n}\$$	$T' \to *T$
$TS'\#$	$\mathbf{n}\$$	
$FT'S'\#$	$\mathbf{n}\$$	$T \to FT'$
$\mathbf{n}T'S'\#$	$\mathbf{n}\$$	$F \to \mathbf{n}$
$T'S'\#$	$\$$	$T' \to \epsilon$
$S'\#$	$\$$	$S' \to \epsilon$
$\#$	$\$$	

DEFINITION 25.20 For each sentential form $\alpha \in \{\Sigma \cup V\}^*$, and for each nonterminal $A \in V$,

$$\begin{aligned} FIRST(\alpha) &= \left\{ c \in \Sigma \,|\, \text{for some } \beta \in \{\Sigma \cup V\}^*, \alpha \Rightarrow^* c\beta \right\} \cup \left\{ \epsilon \,|\, \alpha \Rightarrow^* \epsilon \right\} \\ FOLLOW(A) &= \left\{ c \in \Sigma \,|\, \text{for some } \alpha, \beta \in \{\Sigma \cup V\}^*, S \Rightarrow^* \alpha A c \beta \right\} \\ &\cup \left\{ \$ \,|\, \text{for some } \alpha \in \{\Sigma \cup V\}^*,\ S \Rightarrow^* \alpha A \right\} . \end{aligned}$$

Intuitively, for any sentential form α, $FIRST(\alpha)$ consists of all the terminals that appear as the first symbol of some sentential form derivable from α. The empty string ϵ is included in $FIRST(\alpha)$ as a special case if α derives ϵ. On the other hand, for any nonterminal A, $FOLLOW(A)$ consists of all the terminals that immediately follow an occurrence of A in some sentential form derivable from the start symbol S.

The end delimiter \$ is included in $FOLLOW(A)$ as a special case if A appears at the end of some sentential form derivable from S.

Algorithms for computing the $FIRST()$ and $FOLLOW()$ functions are fairly straightforward and can be found in [1, 6]. It turns out that to construct the parsing table for a grammar G, we only need to know the values of $FIRST(\alpha)$ for those sentential forms α appearing on the right-hand sides of the productions in G.

EXAMPLE 25.16:

The following illustrate the functions $FIRST(\alpha)$ and $FOLLOW(A)$ for the grammar G_8 described in the above example. For the former, only those sentential forms appearing on the right-hand sides of the productions in G_8 are considered.

$$\begin{aligned}
FIRST(TS') &= \{(, \mathbf{n}\} \\
FIRST(+S) &= \{+\} \\
FIRST(FT') &= \{(, \mathbf{n}\} \\
FIRST(*T) &= \{*\} \\
FIRST((S)) &= \{(\} \\
FIRST(\mathbf{n}) &= \{\mathbf{n}\} \\
FIRST(\epsilon &= \{\epsilon\} \\
FIRST(S) &= \{), \$\} \\
FIRST(S') &= \{), \$\} \\
FIRST(T) &= \{+,), \$\} \\
FIRST(T') &= \{+,), \$\} \\
FIRST(F') &= \{*, +,), \$\}
\end{aligned}$$

Given the functions $FIRST(\alpha)$ and $FOLLOW(A)$ for a grammar G, we can easily construct the **LL parsing** table $T[A, c]$ for G. The basic idea is as follows. Suppose that $A \rightarrow \alpha$ is a production and $c \in FIRST(\alpha)$. Then, the parser will replace A with α when A is on top of the stack and c is the current input symbol. The only complication occurs when α may derive ϵ. In this case, the parser should still replace A with α if the current input symbol is a member of $FOLLOW(A)$. The detailed algorithm is given below.

Algorithm LL-Parsing-Table($G = (\Sigma, V, S, P)$)
1 Initialize each entry of the table to blank.
2 **for** each production $A \rightarrow \alpha$ in P **do**
3 **for** each terminal $a \in FIRST(\alpha)$ **do**
4 add $A \rightarrow \alpha$ to $T[A, a]$;
5 **if** $\epsilon \in FIRST(\alpha)$ **then**
6 **for** each terminal or delimiter $a \in FOLLOW(A)$ **do**
7 add $A \rightarrow \alpha$ to $T[A, a]$;

The above algorithm can be applied to any context-free grammar to produce a parsing table. However, for some grammars the table may have entries containing multiple productions. Multiply defined entries

in a parsing table, however, would present our parsing algorithm with an unwelcome choice. It would be possible for it to make a wrong choice and incorrectly report a string as not being derivable, and backtracking to the last choice to try another would blow up the running time unacceptably.

EXAMPLE 25.17:

Recall that we could make the grammar G_7 of Example 25.13 unambiguous by deleting two unnecessary productions. The resulting grammar, which we call G_9, has the following productions:

$$\begin{aligned} S &\rightarrow S + T | T \\ T &\rightarrow T * F | F \\ F &\rightarrow \mathbf{n} | (S) \end{aligned}$$

It is easy to see that both $FIRST(S + T)$ and $FIRST(T)$ contain the terminal **n**. Hence, the entry $T[S, \mathbf{n}]$ of the parsing table is multiply defined, so this table is not well-conditioned for LL parsing.

A context-free grammar whose parsing table has no multiply defined entries is called an **LL(1) grammar.** Here, the "1" signifies the fact that the LL parser uses one input symbol of lookahead to decide its next move. For example, G_8 is an LL(1) grammar, while G_9 is not. It is easy to show that our LL parser runs in linear time for any LL(1) grammar.

What can we do for grammars that are not LL(1), such as G_9? The first idea is to extend the LL parser to use more input symbols of lookahead. In other words, we will allow the parser to see the next several input symbols before it makes a decision. For one more symbol of lookahead, this requires expanding the parsing table to have a column for every *pair* of symbols in Σ (plus \$ as a possible second symbol), but so doing may separate and/or eliminate multiply defined entries in the original parsing table. The *FIRST*() and *FOLLOW*() functions have to be modified to take two (or more) lookahead symbols into consideration. For any constant $k > 1$, a grammar is said to be an **LL(k) grammar** if its parsing table using k lookahead symbols has no multiply defined entries. For example, the grammar G_1 given in Example 25.7 is not LL(1), but it is LL(2).

Although LL(k) grammars form a larger class than LL(1) grammars, there are still grammars that are not LL(k) for any constant k. The grammar G_7 and G_9 are examples. The texts [1, 6] provide several techniques for dealing with non-LL(k) grammars, such as grammar transformations and backtracking. When backtracking is used, the parsing process is often called *recursive-descent parsing,* and can be very time consuming due to the use of many recursive calls.

Bottom-Up Parsing

The most popular **bottom-up parsing** technique is **LR parsing.** Here, the "L" again means scanning the input from left to right, while the "R" means constructing the rightmost derivation. For any input string x, the LR parser scans x from left to right and tries to find the *reverse* of the sequence of productions used in the rightmost derivation of x. It turns out that in bottom-up parsing rightmost derivations are easier to deal with than leftmost derivations. LR parsing is especially attractive in practice for many reasons summarized in [1]: (i) it can handle virtually all programming language constructs; (ii) it has very efficient implementations; (iii) it is more powerful than LL parsing; and (iv) it detects syntactic errors quickly. The principal drawback of the method is that constructing an LR parser is very involved. Fortunately, there exist efficient algorithms that can automatically generate LR parsers from certain context-free grammars. Because of space limitations, we describe only the operation of an LR parser here, and refer the reader to [1] for the construction of such a parser.

Similar to an LL parser, an LR parser has an input buffer, a pushdown stack, a parsing table, and an output stream, and is controlled by an algorithm that is the same for all LR parsers. The input string is

again assumed to have an end delimiter \$. At any time during parsing, the stack stores a string of the form $q_m X_m q_{m-1} \cdots X_1 q_0$ (with q_0 at bottom), where each X_i is a grammar symbol (i.e., a terminal or nonterminal of the grammar involved) and q_i is a *state* symbol. The number of distinct states is finite, and each state symbol intends to summarize the information contained in the stack below it. The combination of the state on top of the stack and the current input symbol are used to index the parsing table and determine the move of the parser. It will be seen that the state symbols subsume all information in the grammar symbols, and a real parser omits the latter. However, we retain the grammar symbols $X_1, \ldots, X_m$ to make our illustration easier to follow, and for consistency with previous examples.

The parsing table consists of two parts: a parsing action function $ACTION(q, c)$, which maps a state and an input symbol to a move, and a function $GOTO(q, X)$, which maps a state and a grammar symbol to a state. For each state q and each input symbol c, the value of the function $ACTION(q, c)$ can be one of the following:

1. *shift,*
2. *reduce by* $A \to \alpha$, where $A \to \alpha$ is a production in the grammar,
3. *accept,* and
4. *blank.*

The algorithm controlling the LR parser operates as follows. Suppose that the state on top of the stack is q and the current input symbol is c. It consults $ACTION(q, c)$ and makes one of the four types of moves as below.

1. If $ACTION(q, c) =$ *shift,* the parser pushes the string $GOTO(q, c)c$ on the stack and shifts its input pointer to the next input symbol.
2. If $ACTION(q, c) =$ *reduce by* $A \to \alpha$, the parser applies the production $A \to \alpha$ as follows. Let $k = |\alpha|$, and let the current stack content be $q_m X_m q_{m-1} \cdots X_1 q_0$. The parser first pops the top $2k$ symbols $q_m, X_m, \ldots, q_{m-k+1}, X_{m-k+1}$ off the stack. It then consults $GOTO(q_{m-k}, A)$ and pushes the string $GOTO(q_{m-k}, A)A$ onto the stack, resulting in a stack with content $GOTO(q_{m-k}, A)Aq_{m-k}X_{m-k} \cdots X_1 q_0$. The parser also outputs the production $A \to \alpha$. It is always guaranteed in the above that $X_{m-k+1} \cdots X_m = \alpha$.
3. If $ACTION(q, a) =$ *accept,* the parser successfully terminates.
4. If $ACTION(q, a) =$ *blank,* the parser terminates and declares that the input string is not a member of the language.

Intuitively, the LR parser reconstructs the rightmost derivation of a string $x = a_1 \cdots a_n$ as follows. Suppose that the rightmost derivation of x is

$$S = \gamma_0 \Rightarrow \gamma_1 \Rightarrow \cdots \Rightarrow \gamma_i \Rightarrow \gamma_{i+1} \Rightarrow \cdots \Rightarrow \gamma_m = x \ ,$$

where each γ_j is a sentential form. Furthermore, suppose that the derivation step $\gamma_i \Rightarrow \gamma_{i+1}$ is the result of applying a production $A \to Y_1 \cdots Y_k$. This means that $\gamma_i = \alpha Az$ for some sentential form $\alpha = X_1 \cdots X_t$ and string z of terminals, and $\gamma_{i+1} = \alpha Y_1 \cdots Y_k z = X_1 \cdots X_t Y_1 \cdots Y_k z$. Since no subsequent derivation will change z, this string must match a trailing substring $a_j \cdots a_n$ of x for some j. In other words, $\gamma_i = X_1 \cdots X_t A a_j \cdots a_n$ and $\gamma_{i+1} = X_1 \cdots X_t Y_1 \cdots Y_k a_j \cdots a_n$.

Suppose that the parser has successfully reconstructed the derivation steps in reverse from γ_m back to γ_{i+1}. At this point, the stack must be holding a string of the form

$$q_{t+h} Y_h \cdots q_{t+1} Y_1 q_t X_t \cdots q_1 X_1 q_0 \ ,$$

where $h \leq k$ and $q_0, q_1, \ldots q_{t+h}$ are some states, and the input pointer is pointing at a_{j+h-k}. Moreover, it must be that $Y_{h+1} = a_{j+h-k}, \ldots, Y_k = a_{j-1}$. To recover γ_i, the parser consults the state q_{t+h} on top

of stack and the current input symbol a_{j+h-k}. It then shifts the $h-k$ input symbols $a_{j+h-k}, \ldots, a_{j-1}$ and $h-k$ appropriate state symbols onto the stack. It also advances the input pointer to a_j. Then, the parser *reduces* the string $Y_1 \cdots Y_k$ to the nonterminal A by replacing the top $2k$ stack symbols with A and an appropriate state symbol.

The above shift-and-reduce process is repeated until the sentential form $\gamma_0 = S$ is obtained. For this reason, the LR parser is sometimes called a *shift-reduce parser.*

Clearly, the state symbols stored on the stack play a key role in dictating the actions of the parser. Below we first give an example of LR parsing tables and show exactly how the parser operates on a specific input. Then we will briefly sketch how the states are chosen for a grammar and what they represent.

EXAMPLE 25.18:

Consider again the unambiguous grammar G_9 given in Example 25.17. For convenience, let us number the productions as follows.

$$
\begin{array}{lllll}
(1) & S & \rightarrow & S + T \\
(2) & S & \rightarrow & T \\
(3) & T & \rightarrow & T * F \\
(4) & T & \rightarrow & F \\
(5) & F & \rightarrow & \mathbf{n} \\
(6) & F & \rightarrow & (S)
\end{array}
$$

Tables 25.4 and 25.5 illustrate the functions $ACTION(q, c)$ and $GOTO(q, X)$ for the grammar. In the first table, shf means shift, pi means reduce by production i, acc means accept, and blank means reject. The states are numbered $0, 1, \ldots, 11$.

TABLE 25.4 The Function $ACTION(q, c)$ for the Unambiguous Grammar G_9

STATE	**n**	+	*	(	)	$
0	shf			shf		
1		shf				acc
2		p2	shf		p2	p2
3		p4	p4		p4	p4
4	shf			shf		
5		p6	p6		p6	p6
6	shf			shf		
7	shf			shf		
8		shf			shf	
9		p1	shf		p1	p1
10		p3	p3		p3	p3
11		p5	p5		p5	p5

Now we demonstrate how the parser operates on the string $(\mathbf{n} + \mathbf{n}) * \mathbf{n}$. Table 25.6 shows the content of the stack, the remaining input symbols, and the output after each step. It is easy to see that the reverse sequence of the productions in the reduce steps constitute the rightmost derivation of $(\mathbf{n} + \mathbf{n}) * \mathbf{n}$.

There are several techniques for constructing an LR parsing table, such as simple-LR (SLR), canonical-LR, and lookahead-LR (LALR), as described by [1]. In general, these techniques all use states that are sets

TABLE 25.5 The Function $GOTO(q, X)$ for the Unambiguous Grammar G_9

STATE	**n**	+	*	(	)	$	S	T	F
0	5			4			1	2	3
1		6							
2			7						
3									
4	5			4			8	2	3
5									
6	5			4				9	3
7	5			4					10
8		8			11				
9		7							
10									
11									

TABLE 25.6 The Steps in the LR Parsing of $(\mathbf{n} + \mathbf{n}) * \mathbf{n}$

STACK	INPUT	ACTION
0	$(\mathbf{n} + \mathbf{n}) * \mathbf{n}\$$	shift
$4(0$	$\mathbf{n} + \mathbf{n}) * \mathbf{n}\$$	shift
$5\mathbf{n}4(0$	$+\mathbf{n}) * \mathbf{n}\$$	reduce by $F \to \mathbf{n}$
$3F4(0$	$+\mathbf{n}) * \mathbf{n}\$$	reduce by $T \to F$
$2T4(0$	$+\mathbf{n}) * \mathbf{n}\$$	reduce by $S \to T$
$8S4(0$	$+\mathbf{n}) * \mathbf{n}\$$	shift
$6 + 8S4(0$	$\mathbf{n}) * \mathbf{n}\$$	shift
$5\mathbf{n}6 + 8S4(0$	$) * \mathbf{n}\$$	reduce by $F \to \mathbf{n}$
$3F6 + 8S4(0$	$) * \mathbf{n}\$$	reduce by $T \to F$
$9T6 + 8S4(0$	$) * \mathbf{n}\$$	reduce by $S \to S + T$
$8S4(0$	$) * \mathbf{n}\$$	shift
$11)8S4(0$	$*\mathbf{n}\$$	reduce by $F \to (S)$
$3F0$	$*\mathbf{n}\$$	reduce by $T \to F$
$2T0$	$*\mathbf{n}\$$	shift
$7 * 2T0$	$\mathbf{n}\$$	shift
$5\mathbf{n}7 * 2T0$	$\$$	reduce by $F \to \mathbf{n}$
$10F7 * 2T0$	$\$$	reduce by $T \to T * F$
$2T0$	$\$$	reduce by $S \to T$
$1S0$	$\$$	accept

of *items* of the form $A \to \alpha \cdot \beta$, where $A \to \alpha\beta$ is a production and the $\cdot$ marks a place in the right-hand side. Such items are commonly known as the *LR items.* Each item expresses the assertion that the part α has already been obtained by previous shift/reduce steps and pushed on the stack, and the part β is expected to be obtainable from the next few input symbols by some shift/reduce steps. Since at any given time the parser may not be able to predict what input symbols should follow, it has to maintain a set of LR items to deal with all possibilities.

Again, not all context-free grammars have effective LR parsers. For example, the grammar with productions

$$S \rightarrow 0S0|1S1|0|1|\epsilon$$

cannot be handled by LR parsing. This grammar generates the set of all palindromes. The grammars that have effective LR parsers are called *LR grammars.* In fact, there are context-free languages that cannot be represented by any LR grammars. The set of palindromes is one such language.

25.6 Defining Terms

Ambiguous context-free grammar: A context-free grammar in which some derivable terminal strings have two distinct derivation trees.

Bottom-up parsing: A process of building a derivation tree from the leaves up to the root.

Chomsky normal form: A form of context-free grammar in which every rule has the form $A \rightarrow BC$ or $A \rightarrow a$, where A, B, C are nonterminals and a is a terminal.

Context-free grammar: A grammar whose rules have the form $A \rightarrow \beta$, where A is a nonterminal and β is a string of nonterminals and terminals.

Context-free language: A language that can be described by some context-free grammar.

Context-sensitive grammar: A grammar whose rules have the form $\alpha \rightarrow \beta$, where α, β are strings of nonterminals and terminals, and $|\alpha| \leq |\beta|$.

Context-sensitive language: A language that can be described by some context-sensitive grammar.

Derivation or parsing: A sequence of applications of rules of a grammar that transforms the start symbol into a given terminal string or sentential form.

Derivation tree or parse tree: A rooted, ordered tree that describes a particular derivation of a string with respect to some context-free grammar.

(Formal) language: A set of strings over some fixed alphabet.

(Formal) grammar: A description of some language, typically consisting of a set of terminals, a set of nonterminals, a distinguished nonterminal called the start symbol, and a set of rules (or productions) of the form $\alpha \rightarrow \beta$, which determine which substrings α of a sentential form can be replaced by some another string β.

Leftmost (or rightmost) derivation: A derivation in which at each step, the leftmost (respectively, rightmost) nonterminal is rewritten.

LL parsing: A type of top-down parsing in which one reads the input from left to right in order to reconstruct a leftmost derivation.

LL(k) grammar: A context-free grammar whose LL(k) parsing table has no multiply defined entries.

LL(k) parsing: An LL parsing that uses k symbols of lookahead.

LR parsing: A type of bottom-up parsing in which one reads the input from left to right in order to reconstruct a rightmost derivation in reverse order of steps.

LR grammar: A context-free grammar that has an effective LR parser.

Membership problem (or lexical analysis): The problem or process of deciding whether a given string is generated by a given grammar.

Parsing problem: The problem of reconstructing a derivation of a given input string in a given grammar.

Regular expression: A description of some language using the operators union, concatenation, and Kleene closure.

Regular language: A language that can be described by some regular expression, or equivalently, by some right-linear/regular grammar.

Right-linear or regular grammar: A grammar whose rules have the form $A \to cB$, $A \to c$, or $A \to \epsilon$, where A, B are nonterminals, c is a terminal, and ϵ is the empty string.

Sentential form: A string of terminals and nonterminals obtained at some step of a derivation in a grammar.

Top-down parsing: A process of building derivation trees from the top (root) down to the bottom (leaves).

References

[1] Aho, A.V., Ullman, J.D., and Sethi, I., *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, Reading, MA, 1985.

[2] Angluin, D., Finding patterns common to a set of strings. *Journal of Computer and System Sciences.* 21, 46–62, 1980.

[3] Chomsky, N., Three models for the description of language. *IRE Trans. on Information Theory.* 2(2), 113–124, 1956.

[4] Chomsky, N., Formal properties of grammars. In *Handbook of Mathematical Psychology,* Vol. 2, 323–418, John Wiley & Sons, New York, 1963.

[5] Chomsky, N. and Miller, G., Finite-state languages. *Information and Control.* 1, 91–112, 1958.

[6] Drobot, V., *Formal Languages and Automata Theory.* Computer Science Press, Rockville, MD, 1989.

[7] Floyd, R.W. and Beigel, R., *The Language of Machines: An Introduction to Computability and Formal Languages.* Computer Science Press, New York, 1994.

[8] Gurari, E., *An Introduction to the Theory of Computation.* Computer Science Press, Rockville, MD, 1989.

[9] Harel, D., *Algorithmics: The Spirit of Computing.* Addison-Wesley, Reading, MA, 1992.

[10] Harrison, M., *Introduction to Formal Language Theory.* Addison-Wesley, Reading, MA, 1978.

[11] Hopcroft, J. and Ullman, J., *Introduction to Automata Theory, Languages and Computation.* Addison-Wesley, Reading, MA, 1979.

[12] Jiang, T., Salomaa, A., Salomaa, K., and Yu, S., Decision problems for patterns. *Journal of Computer and System Sciences.* 50(1), 53–63, 1995.

[13] Kleene, S., Representation of events in nerve nets and finite automata. In *Automata Studies,* 3–41. Princeton University Press, NJ, 1956.

[14] Lind, D. and Marcus, *Symbolic Dynamics,* Academic Press, 1995.

[15] Post, E., Formal reductions of the general combinatorial decision problems. *Amer. J. Math.,* 65, 197–215, 1943.

[16] Salomaa, A., Two complete axiom systems for the algebra of regular events. *J. ACM.* 13(1), 158–169, 1966.

[17] Searls, D., The computational linguistics of biological sequences. In *Artificial Intelligence and Molecular Biology.* L. Hunter, Ed., MIT Press, 1993, 47–120, 1993.

[18] Wood, D., *Theory of Computation.* Harper and Row, 1987.

Further Information

The fundamentals of formal languages and grammars can be found in many text books including [6, 7, 8, 9, 10, 11, 18]. The central focus of research in this area has been to find formal grammatical representations of languages that are very expressive and are yet easy to parse. The research results have

greatly benefited many fields of computer science, including programming languages, compiler design, and natural language processing. Chapter 24 presents the machine model counterparts of regular grammars, context-free grammars, context-sensitive grammars, and unrestricted grammars, and Chapter 26 introduces the concepts of decidability and undecidability, which has a close relation to formal grammars. The following annual conferences present the leading research work in formal languages and grammars: International Colloquium on Automata, Languages and Programming (ICALP), ACM Annual Symposium on Theory of Computing (STOC), IEEE Symposium on the Foundations of Computer Science (FOCS), ACM Symposium on Principles of Programming Languages (POPL), Symposium on Theoretical Aspects of Computer Science (STACS), Mathematical Foundations of Computer Science (MFCS), Fundamentals of Computation Theory (FCT), Foundation of Software Technology and Theoretical Computer Science (FSTTCS), and Conference on Developments in Language Theory (DLT). There are many related conferences, including Computational Learning Theory (COLT), Colloquium on Trees in Algebra and Programming (CAAP), and International Conference on Concurrency Theory (CONCUR), where either specific issues concerning formal grammars are considered or specialized grammatical systems are studied for a specific application area. We conclude with a list of major journals that publish papers in formal language theory: *Journal of the ACM, SIAM Journal on Computing, Journal of Computer and System Sciences, Information and Computation, Theory of Computing Systems* (formerly *Mathematical Systems Theory*), *Theoretical Computer Science, Information Processing Letters, International Journal of Foundations of Computer Science,* and *Acta Informatica.*

26

Computability

Tao Jiang
McMaster University

Ming Li
University of Waterloo

Bala Ravikumar
University of Rhode Island

Kenneth W. Regan
State University of New York at Buffalo

26.1 Introduction

In the last two chapters, we have introduced several important computational models, including Turing machines and Chomsky's hierarchy of formal grammars. In this chapter, we will explore the limits of mechanical computation as defined by these models. We begin with a list of fundamental problems for which automatic computational solution would be very useful. One of these is the *universal simulation problem:* can one design a single algorithm that is capable of simulating *any* algorithm? Turing's demonstration that the answer is *yes* [28] supplied the proof for Babbage's dream of a single machine that could be programmed to carry out any computational task. We introduce a simple Turing machine programming language called "GOTO" in order to facilitate our own design of a universal machine. Next, we describe the schemes of *primitive recursion* and μ*-recursion,* which enable a concise, mathematical description of computable functions that is independent of any machine model. We show that the μ-recursive functions are the same as those computable on a Turing machine, and describe some computable functions, including one that solves a second problem on our list.

The success in solving ends there, however. We show in the last section of this chapter that all of the remaining problems on our list are *unsolvable* by Turing machines, and subject to the *Church–Turing thesis,* have no mechanical or human solver at all. That is to say, there is no Turing machine or physical device, no stand-alone product of human invention, that is capable of giving the correct answer to all—or even most—instances of these problems. The implication we draw is that in order to solve *some* important instances of these problems, human ingenuity is needed to guide powerful computers down the paths felt most likely to yield the answers. To cite Raymond Smullyan quoting P. Rosenbloom, the results on unsolvability imply that "man can never eliminate the necessity of using his own cleverness, no matter how cleverly he tries."

Almost the first consequence of formalizing computation was that we can formally establish its limits. Kurt Gödel showed that the process of proof in any formal axiomatic system of logic can be simulated by the basic arithmetical functions that computation is made of. Then he proved that any sound formal system that is capable of stating the grade-school rules of arithmetic can make statements that are neither provable nor disprovable in the system. Put another way, every sound formal system is *incomplete* in the sense that there are mathematical truths that cannot be proved in the system. Turing realized that Gödel's basic method could be applied to computational models themselves, and thus proved the first computational unsolvability results. Since then problems from many areas, including group theory, number theory, combinatorics, set theory, logic, cellular automata, dynamical systems, topology, and knot theory, have been shown to be unsolvable. In fact, proving unsolvability is now an accepted "solution" to a problem. It is just a way of saying that the problem is too general for a computer to handle—that supplementary information is needed to enable a mechanical solution.

Since Turing machines capture the power of mechanical computability, our study will be based on Turing's model. In the next section, we describe a Turing machine as a computer that can run programs written in a very simple language we call the "GOTO Language." This formalism is equivalent to Chapter 24's description of Turing machines using the standard 7-tuple notation. Our language provides an alternate way to write programs and makes proofs about Turing machines more intuitive.

26.2 Computability and a Universal Program

Turing's notion of mechanical computation was based on identifying the *basic steps* in any mechanical computation. He reasoned that an operation such as numerical multiplication is not primitive, because it can be divided into simpler steps such as using the times-table on individual pairs of digits, shifting, and adding. Addition itself can be broken down into simpler steps such as adding the lowest digits, computing the carry, and moving to the next digit. Turing concluded that the most basic features of mechanical computation are the ability to read and write on a storage medium, the ability to move about on that medium, and the ability to make simple logical decisions. Turing chose the storage medium to be a single linear *tape* divided into cells. He showed that such a tape could model spatial memory in three (or any number of) dimensions through the use of indexed co-ordinates. With much care he argued that human sensory input could be encoded by strings over a finite alphabet of cell symbols called the *tape alphabet.* (This bold discretization of sensory experience now seems a harbinger of the digital revolution that was to follow.) A decision step enables the computer to exert local control over the sequence of actions. Turing restricted the next action performed to be in a cell neighboring the one on which the current action occurred, and showed how nonlocal actions can be simulated by successions of steps of this kind. He also introduced an instruction to tell the computer to stop.

In summary, Turing proposed a model to characterize mechanical computation as being carried out as a sequence of instructions. Our "GOTO" formalism provides the following five kinds of instructions. Here i stands for a tape symbol and j stands for a line number.

PRINT i
MOVE RIGHT
MOVE LEFT
IF i IS SCANNED GOTO LINE j
STOP

When we speak about programs recognizing languages rather than computing functions, we replace STOP by statements ACCEPT and REJECT, each of which need occur only once in a program.

A program in this language is a sequence of instructions or "lines" numbered 1 to k. The input to the program is a string over a designated *input alphabet* Σ, which we take to be $\{0, 1\}$ throughout this chapter. The tape alphabet includes Σ and a special *blank* character B representing an empty cell, and may (but

need not) contain other symbols. The input is stored on the tape, with the read head scanning the first symbol (or B if the input is empty), before the computation begins.

How much memory should we allow the computer to use? Rather than postulate that the tape is actually infinite—an unrealistic assumption—we prefer here to say that the tape has expandable boundaries. Initially the input defines the two boundaries of the tape. Whenever the machine moves left of the left boundary or right of the right boundary, a new memory cell containing the blank is attached. This convention clarifies what we mean by saying that if and when the machine halts by reaching the STOP instruction, the "result" of the computation is the entire content of the tape.

We present an example program written in the GOTO language. This program accomplishes the simple task of doubling the number of initial 1s in the input string. Informally, the program achieves its goal as follows: When it reads a 1, it changes the 1 to a 0, moves left looking for a new cell, and writes a 1 in that cell. Then it returns rightward to the 0 that marks where it had been, rewrites it as a 1, and moves right to look for more 1s. If it immediately finds another 1 it repeats the process from line 1, while if it doesn't, it halts right there. This program even has a "bug"—it "should" leave strings that do not begin with a 1 unchanged, but instead it alters them.

```
1 PRINT 0
2 MOVE LEFT
3 IF 1 IS SCANNED GOTO LINE 2
4 PRINT 1
5 MOVE RIGHT
6 IF 1 IS SCANNED GOTO LINE 5
7 PRINT 1
8 MOVE RIGHT
9 IF 1 IS SCANNED GOTO LINE 1
10 STOP
```

FIGURE 26.1 The doubling program in the GOTO language.

The main change from the traditional Turing machine formalism of Chapter 24 is that we have replaced "states" by line numbers. A Turing machine of the former kind can always be simulated in our GOTO language by making blocks of successive lines, themselves divided into sub-blocks (for each character) that are headed by "IF" statements, carry out the instructions for each state. Our formalism makes many programs more succinct and closer to programmers' experience, and highlights the role of (conditional) GOTO instructions in setting up loops and enabling statements to be repeated. Despite the popular scorn of `goto` statements, this feature is ultimately the most important aspect of programming and can be found in every imperative-style programming language—at least in the code produced by the compiler if the language has no `goto` instruction itself. Indeed, the above example could be rendered into a structured programming language such as C as follows,

```
do {
   do { PRINT 0; MOVE LEFT; } while (1 is scanned);
   PRINT 1;
   do MOVE RIGHT; while (1 is scanned);
   PRINT 1; MOVE RIGHT; }
while (1 is scanned);
```

and a C compiler (using a character array `tape[i]` and `++i`, `--i` for the moves) might plausibly convert this into something exactly like our GOTO program!

The simplicity of the GOTO language is rather deceptive. As the above example hints, any program in any known high-level programming language can be converted into an equivalent GOTO program, under suitable conventions on how inputs and outputs are represented on the tape. (If the program only reads from the standard input stream and writes to the standard output stream, then no such conventions are necessary.) There is strong reason to believe that any mechanical computation of any future kind can be expressed by a suitable GOTO program. Note, however, that a program written in the GOTO language need not always halt; i.e., on certain inputs the program may never reach a STOP instruction. On such inputs we say that the output of the program is *undefined.*

Now we can give a precise definition of what we mean by an *algorithm,* attempting to rule out this last situation. An algorithm is *any program written in the* GOTO *language that has the additional property of halting on all inputs.* Such programs will be called *halting programs,* and correspond to "total" deterministic Turing machines in Chapter 24. When we consider **decision problems,** which have yes/no answers, halting programs are required to end their computation with either an ACCEPT or a REJECT statement, on any input.

Some Computational Problems

We begin by listing a collection of computational problems for which a mechanical solution can be very helpful. By a mechanical solution, we mean a step-by-step process that takes into account all possible inputs, and that can be executed without any human assistance once a certain input is provided. An algorithm is required to work correctly on all instances.

We now list some problems that are fundamental either because they are inherently important or because they played a historical role in the development of computation theory. For the first four, P stands for a program in our GOTO language, and x is a string over the input alphabet, which we fix to be $\{0, 1\}$.

1. *Universal simulation.* Given a program P and an input x to P, determine the output (if any) that P would produce on input x.
2. *Halting problem.* Given P and x, output 1 (for yes) if P would halt when given input x, and 0 (for no) if P would not halt.
3. *Type-0 grammar membership.* Given a type-0 grammar G and a string x, determine whether x can be derived from the start symbol of G.
4. *String compression.* Given a string x, find the shortest program P such that when P is started with empty tape, P eventually halts with x as its output. Here "shortest" means that the total number of symbols in the program's instructions is as small as possible.
5. *Tiling.* Given a finite set T of tile types, where all tiles of a type are unit squares with the same four colors on their four edges, determine whether every finite rectangle can be *tiled* by T. If k and n are the integer sides of the rectangle, being tiled means that kn tiles drawn from T can be arranged so that every two tiles that share an edge have the same color at that edge.
6. *Linear programming.* Given some number k of linear inequalities in n unknowns, determine whether there is an assignment of n values to the unknowns that satisfies all the inequalities.
7. *Integer equations.* Given k-many polynomial equations in n unknowns, determine whether there is an assignment of n integers to the unknowns that satisfies all the equations.

SOME REMARKS ABOUT THE ABOVE PROBLEMS: A solution to Problem 1 realizes Babbage's objective of a single program or machine capable of simulating all programs P. For cases where P run on x would never halt and produce output, we have left open whether we require the solution itself to halt and detect this fact—i.e., to be an algorithm. For any such algorithm to exist, there must be an

algorithm to solve Problem 2, which is a yes/no *decision problem.* An algorithm for Problem 2 would be a boon to reliable software design, since it could be used to test whether a given block of code can cause infinite loops. Problem 3 is another decision problem; its solution would be useful for natural-language processing and much more. Problem 4 is a function-computation problem of central importance in information theory. For illustration, think of x as a large amount of scientific data for which we seek a concise theory P that can generate and hence explain it. A famous example is Kepler's laws, which explained Tycho Brahe's voluminous and meticulous observational data. Problem 4 thus asks whether the heart of science (to paraphrase Occam's Razor, "finding the simplest explanation that fits the facts") can be done automatically on a computer.

The tiles in Problem 5, which we introduced in detail in Chapter 24, are sometimes named after Hao Wang who wrote the first research paper about them [30]. Figure 26.2(a) shows an example of a set T of tile types, and Fig. 26.2(b) shows how tiles drawn from T can be used to tile a 5×5 square area. The tiling

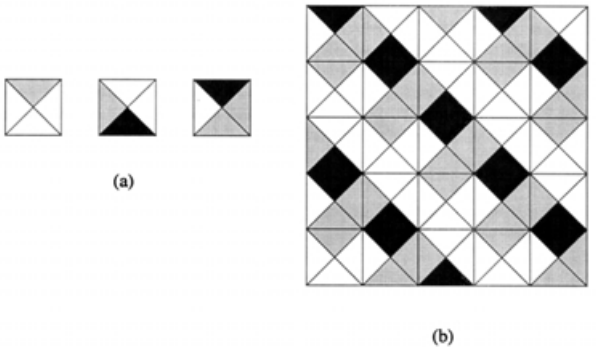


FIGURE 26.2 An example of tiling.

problem is not merely an interesting puzzle. It has been an art form pursued by artists from many cultures for centuries. Tiling problems have deep significance in combinatorics, algebra, and formal languages. Note that our decision problem does not ask simply whether a given $k \times n$ rectangle can be tiled, but whether—given T—*all* $k \times n$ rectangles can be tiled via T. The full problem of linear programming adds to Problem 6 a clause saying: if there exist *feasible solutions,* i.e., assignments that satisfy all the so-called linear *constraints,* find one that maximizes (or minimizes) a given *objective function* (or *cost function*). This problem has central importance in economics, game theory, and operations research. Problem 7 is called *Hilbert's tenth problem,* and was one of twenty-four that David Hilbert posed as challenges for the new century at the International Congress of Mathematicians in 1900. It goes back two thousand years to the mathematician Diophantus' study of these so-called *Diophantine equations.* Actually, Hilbert posed the "meta-problem" of finding an algorithm that can solve any Diophantine equation, or at least tell whether it has a solution.

Recall from Chapter 24 that a decision problem is *decidable* if it has an algorithm, and *undecidable* otherwise. If our program correctly evaluates all instances for which the answer is "yes," but may fail to halt on some instances for which the answer is "no," then the program is a *partial decision procedure,* and the problem is *partially decidable.* A partially **decidable problem,** however, is undecidable—unless you can find an algorithm that removes the word "partially." Likewise, if our program correctly outputs $f(x)$ whenever $f(x)$ is defined, but may fail to halt when $f(x)$ is undefined, then the partial function f is *partial computable.*

In the remainder of the subsection, we present some simple algorithm design techniques and sketch how they make progress on solving some of these problems and special cases of them. These techniques may seem too obvious to warrant explicit description. However, we feel that such a description will help new readers to appreciate the limits on information processing that make certain problems undecidable.

Table Look-Up

For certain functions g it can be advantageous to create a table with one column for inputs x and one for values $g(x)$, looking up the value in the table whenever an evaluation $g(x)$ is needed. A function f that is defined on an infinite set such as Σ^* cannot have its values enumerated in a finite table in this manner, but sometimes the infinite table for f can be described in a finite way that constitutes an algorithm for f. Moreover, tables for other functions g may help the task of computing f, such as the digit-by-digit times-table used in multiplying integers of arbitrary size. These ideas come into play next.

Bounding the Search Domain

Many solutions to decision problems involve finding a *witness* that proves a "yes" or "no" answer for a given instance. The term reflects an analogy to a criminal trial where a key witness may determine the guilt or innocence of the defendant. Thus the first step in solving many decision problems is to identify the right kind of witness to look for. For example, consider the problem of determining whether a given number N is prime. Here a (counter-) witness would be a factor of N (other than 1 and N itself). If N is composite, it is easy to prove by simple division that the witness' claim is correct.

In cases where the given number is prime, a witness of a different kind needs to be searched for. This search may involve integers larger than N, and trying to summon every integer sequentially as a witness would violate the requirement of an algorithm to terminate in finite number of steps. *This is often the main challenge in establishing decidability*. The difficulty can be surmounted if, based on the structure of the problem, we can establish ahead of time an upper bound such that if any witness exists at all, one exists that meets the bound. Then a sufficient body of potential witnesses can be examined in a finite number of steps. In the case of composite N, the bound is N itself. For prime N, there is a known polynomial p such that a witness exists in the numbers between 1 and $2^{p(n)}$, where n is the number of digits in N, according to a certain witnessing scheme whose test for correct claims is easy to compute. This kind of "polynomial size-bounded witnessing scheme" characterizes the important complexity class NP, and is discussed much further in Chapter 27.

For another example, let us consider the special case of Problem 3 where the given G is a Type-1 grammar, and we wish to determine whether a given string x can be generated from the start symbol S of G. A witness in this case can be a sequence of sentential forms starting from S and ending with x that forms a valid derivation in G. The length of x imposes a limit on the size of such sentential forms because G has no length-decreasing productions, and this in turn defines a (much larger) limit on the number of sequences that need be considered before all possibilities are exhausted. Readers may find the details in a standard text such as [12].

For one more example, consider the full version of the linear programming problem where one wishes to maximize a linear objective function f over the set of feasible solutions s. This set may be infinite, and so a table-lookup through all values $f(s)$ cannot be used. However, it is possible to reduce the search domain to a finite set as follows. The feasible solutions form a collection in n-dimensional space (where n is the number of variables plus the number of constraints), known as a *convex polytope.* Unless the polytope is empty or unbounded—cases that can be detected and resolved—the polytope has a finite number of "corner" points, which are similar to the vertices of a polygon, and which are easily computed. In this case, it is known that a linear objective function attains its maximum value at one (or more) of these corner points. Thus we know the problem is decidable via table-lookup of values at the corner points. In practice, there are intelligent algorithms that find a maximum-giving corner point after searching (usually) only a small part of this table.

Use of Subroutines

This is more a programming technique than an algorithm design tool. The idea is to use one program P as a single step in another program Q. Building programs from simpler programs is a natural way to deal with the complexity of the programmer's task. A simple example is using a lookup to the times

table as a subroutine in multiplying two integers i and j. Let us examine this in the context of designing Turing machines, where i and j are represented on the tape by the string 1^i01^j (namely, i 1s followed by a 0 and then by j 1s). The basic idea of our GOTO program is to duplicate the string of i 1s $j-1$ times, meanwhile erasing the string 1^j bit-by-bit to count the iterations. A little thought reveals that our earlier GOTO program in Fig. 26.1 can *almost* be used verbatim as a subroutine to call $j-1$ times. The only hitch is that the first call would run $2i$-many 1s together so that further calls would duplicate too many 1s. To fix the problem, we introduce a new tape symbol 2, using two initial steps to convert the tape to 21^i01^j, and "patch" the subroutine so that it will not overwrite this 2. The new subroutine can be called by a line "k: IF 2 IS SCANNED GOTO m," where m is the number of the first line in the subroutine, and can return control to the point of call by replacing its STOP instruction by "IF 0 IS SCANNED GOTO $k+1$." Careful writing will ensure that this latter 0 is the one initially separating 1^i from 1^j. The remaining details are left to the interested reader, while performing a similar patch without using a new symbol "2" is left to the obsessive reader. This subroutine mechanism is in fact no different from the one programmers in BASIC have used for decades.

A Universal Program

We will now solve Problem 1 by arguing the existence of a program U written in the GOTO language that takes as input a program P (also written in the GOTO language) and data x for P, and that produces the same output as P does on input x, *if* $P(x)$ halts and produces output at all. The last caveat hints that we shall only achieve a partial solution, formally showing only that the function $U(P,x) = P(x)$ is partial computable.

For convenience, we assume that all programs written in the GOTO language use the fixed alphabet $\{0, 1, B\}$. Since we have thus far used the full English alphabet for the notation of our GOTO programs, we must first address the issue of what the formal input to the program U will look like. This problem can be circumvented by *encoding* each instruction using only 0 and 1. The idea of such an encoding should not be mysterious—we could refer to the 0-1 encoding defined by the ASCII standard, which the terminal used to type this chapter has already carried out for these example programs. However, we prefer the more-succinct encoding defined by Table 26.1.

TABLE 26.1 Encoding GOTO Instructions

Instruction	Code
PRINT i	0001^{i+1}
MOVE LEFT	001
MOVE RIGHT	010
IF i IS SCANNED GOTO j	0111^j01^{i+1}
STOP	100

To encode an entire program, we simply write down in order (without the line numbers) the code for each instruction as given in the table. For example, here is the code for the doubling program shown in Fig. 26.1:

$$000100101111011000110100111111101100011010011101 1100 .$$

Note that the encoded string preserves all the information about the program, so that one can easily reverse the process to decode the string into a GOTO program. From now on, if P is a program in the GOTO language, *code* (P) will denote its binary encoding. When there is no confusion, we will identify P and

code (P). We may also assume that all programs P have a unique STOP instruction that comes last. This convention ensures that a input string to U of the form $w = code\,(P)x$ can be parsed into its P and x components. (When we consider decision problems we will use the code 100 for a unique final ACCEPT instruction, and assign some other code to REJECT.) Before proceeding further, readers may test their understanding of the encoding/decoding process by decoding the following string: 0100111011001001.

The basic idea behind the construction of a universal program is simple, although the details involved in actually constructing one are substantial. Turing in his original paper [28] exhibited a universal program in glorious gory detail, while simpler constructions may be found in more-recent sources such as [22]. Here we will content ourselves with a sketch that conveys the central ideas.

U has as its input a string w of the form *code* $(P)x$. (If U is given an input string not of this form, it can detect the flaw and immediately stop.) To simulate the computational steps of P on input x, U divides its tape into two segments, one containing the program P, and one modeling the contents of the tape of P as it changes with successive moves. The computation by U consists of a sequence of *cycles,* each of which simulates one step by P and is analogous to an *REW cycle* (for "read-evaluate-write") in many real computer systems.

To execute a cycle, U first needs to know the cell that the "virtual" tape head of P is currently scanning, and the instruction P is currently executing. We can assist U by extending its own work alphabet to include new "alias" symbols $0'$, $1'$, B' for the characters of P. U maintains the condition that there is exactly one aliased symbol in the "P" segment of its tape that marks the encoding of the current instruction, and exactly one in the other segment that marks the cell currently scanned by P. For example, suppose that after thirty-nine steps, P is reading the fourth symbol from the left on its tape containing 01001001. Then the second tape segment of U after thirty-nine cycles consists of the string $0100'1001$. We can further assist U by adding a symbol $\wedge$ to divide the two segments, although the unique STOP instruction itself could serve as the divider. The computation by U on an input $w = code\,(P)x$ can begin with some steps that prime the first symbol of *code* (P), insert a $\wedge$ before the first symbol of x (caterpillaring x one cell to the right), and prime the first symbol of x. We may suppose that each cycle by U begins with its own head scanning the $\wedge$.

At the beginning of a new cycle, U moves its head left to find the current instruction, and begins decoding it. The only information U needs to retain is which type of instruction it is, and in the case of a PRINT i or IF $i\ldots$ instruction, which character i is involved. To execute a PRINT i, MOVE RIGHT, or MOVE LEFT instruction, P unprimes the instruction, primes the next one, and marches down its tape to find the primed cell on its copy of P's tape and execute the action. It is possible that a MOVE LEFT instruction may bump into the $\wedge$, in which case U makes another call to its "caterpillar" subroutine to move P's tape over, and inserts a B' for the blank P would scan after that move. The only case that requires cumbersome action by U is an instruction IF i IS SCANNED GOTO j, when U finds that P really is scanning character i. Then U needs to find the jth instruction in the "P" part of its tape. Because we have used a unary encoding 1^j of the required line number j, it is not too difficult to write a subroutine that counts off the 1s in 1^j and advances an instruction marker each time beginning from line 1, knowing to stop when the jth instruction has been located. Finally, if the current instruction is STOP, U gleefully *erases* P, erases the $\wedge$, and unprimes the scanned symbol, leaving exactly the final output $P(x)$.

One last refinement is needed to answer the objection that U is using extra tape symbols $0'$, $1'$, B', $\wedge$ that we have expressly forbidden to GOTO programs. This use can be eliminated by one more level of encoding. Give each of the seven tape symbols its own three-bit code, and make U treat blocks of three cells as single cells in the simulation that was described above. U itself can be programmed to convert its input *code* $(P)x$ to this encoding before the first cycle, and to invert it when restoring the final output $P(x)$. Then U is a bona-fide GOTO program that meets all our requirements. It is even possible to run U on input *code* $(U)w$ where $w = code\,(P)x$, producing (more slowly) the same output $P(x)$. It is important to note that the code of U itself is completely independent of any program P that might be simulated. The code of U itself is not long—a reader with good programming skill can make it shorter than the prose description we have just given.

Besides solving what was asked for in Problem 1, we have also shown that Problem 2 is partially decidable. Namely, for any "yes"-instance $w = code(P)x$ where P on input x halts, U on input w will eventually detect that fact—and the slight edit of changing U's own STOP instruction to ACCEPT will make U halt and accept w. However, on a "no"-instance where $P(x)$ does not halt, our U will blindly follow P and not halt either. The question is whether we can improve U so that it will *detect* every case in which $P(x)$ does not halt, and signal this by executing a REJECT instruction. We will see in Section 26.5 that all the programming skill in the world cannot produce such a U—the halting problem is *un*decidable.

Before presenting undecidability, however, we develop a fundamentally different way to formalize the notion of mechanical computability in the next section.

26.3 Recursive Function Theory

The main advantage of using the class of *μ-recursive* functions to define computation is their mathematical elegance. Proofs about this class can be presented in a rigorous and concise way, without long prose descriptions or complicated programs that are hard to verify. These functions need and make no reference to any computational machine model, so it is remarkable that they characterize "mechanical" computability.

An analogy to the two broad families of programming languages is in order. We have already discussed how Turing machines and our particular "GOTO" formalism abstract the essence of *imperative* programming languages, in which a program is a sequence of operational commands and the major program structures are subroutines and loops and other forms of *iteration.* By contrast, specifications in recursive function theory are *declarative,* and the major structures are forms of *recursion.* "Declarative" means that a function f is specified by a direct description of the value $f(x)$ on a general argument x, as opposed to giving steps to compute $f(x)$ on input x. Often this description is *recursive,* meaning that $f(x)$ is defined in terms of values $f(y)$ on other (usually smaller) arguments y. Programming languages built on declarative principles include Lisp, ML, and Haskell, which are known as *functional languages.* These languages have recursion syntax that is not greatly different from the recursion schemes presented here. They also draw upon Church's *lambda calculus,* which can be called the world's first general programming language. A formal proof of equivalence between lambda calculus and the Turing machine model (via a programming language called I) can be found in [13], which presents computability theory from a programming perspective.[1]

In this section, we will describe this functional approach to computation and code some simple functions using recursion. Owing to space limitation, we will not present a complete proof that the class of *μ-recursive* functions is the same as the class of (partial) computable functions on a Turing machine. The full proof can be found in standard texts such as [27]. All the functions we consider have one or more nonnegative integers as arguments, and produce a single nonnegative integer value.

Before presenting formal definitions, we qualify the above ideas with a few examples. Consider first the simple definition of a two-variable linear function by

$$h(y, z) = z + 2 * y + 1 \,. \tag{26.1}$$

Here $h(y, z)$ is defined with the aid of other functions (here, plus and times) and quantities (here, 2 and 1) that presumably have already been defined or given. This is an example of an *explicit definition* because all entities on the right-hand side are known—in particular, this definition does not involve recursion. If we

[1] Turing created an addendum to his seminal paper [28] showing that his definition of a (partial) computable function was equivalent to the one proposed by Church. The lambda calculus uses essentially a single execution scheme called *reduction* to govern its computations, and by suitable conditioning one can make this scheme carry out recursion. Another declarative language, Prolog, also fixes a single execution scheme that tries to limit the operational decisions the programmer needs to make, and also relies upon recursion.

rewrite the infix functions + and ∗ in prefix-function style as "*plus* " and "*times* ," the expression becomes

$$h(y, z) = plus\ (z, plus\ (times\ (2, y), 1))\ , \tag{26.2}$$

and we can glimpse another hallmark of functional languages: function names can be regarded as parameters the same way that variable names can. Now consider the somewhat-similar definition of a one-variable function by

$$f(x) = f(x - 1) + 2 * (x - 1) + 1\ , \tag{26.3}$$

together with a base case such as $f(0) = 0$. Here not every quantity on the right-hand side is known—one must first know $f(x - 1)$ to compute $f(x)$. However, this is still "declarative" insofar as $f(x)$ is defined in terms of known quantities and values $f(y)$ for other (smaller) arguments y. The reader may check that this is a recursive definition of the squaring function.

Why use recursion? One reason is that explicit definition by itself is known not to be powerful enough to capture the essence of mechanical computation. The next two sections define the two principal schemes of recursion in recursive function theory.

Primitive Recursive Functions

The class of *primitive recursive functions* is built up from the following set of *basic functions,* which are the only ones we need to presuppose are "known:"

1. The *successor function* S is defined for all x by $S(x) = x + 1$.
2. The *zero function* Z is defined for all x by $Z(x) = 0$. The constant 0 is also provided here.
3. For all fixed numbers i and n with $1 \leq i \leq n$, the *projection function* p_i^n is defined for all n-tuples $(x_1, x_2, \ldots, x_n)$ by $p_i^n(x_1, x_2, \ldots, x_n) = x_i$.

The primitive recursive functions are constructed from the basic functions by applications of the following two operations. The case $n = 0$ is allowed in them; a 0-variable function is the same as a constant, and a 0-tuple is the empty list.

1. *Functional composition:* Given k-many functions $g_1, \ldots, g_k$ that each take n variables, and a function h that takes k variables, one can define a function f of n variables by

$$f\ (x_1, \ldots, x_n) = h\ (g_1\ (x_1, \ldots, x_n)\ , g_2\ (x_1, \ldots, x_n)\ , \ldots, g_k\ (x_1, \ldots, x_n))\ . \tag{26.4}$$

   If $g_1, \ldots, g_k$ and h are primitive recursive, then f is defined to be primitive recursive.
2. *Primitive recursion:* Given a function g that takes n variables, and a function h that takes $n + 2$ variables, one can define a function f of $n + 1$ variables by

$$f\ (x_1, \ldots, x_n, 0) = g\ (x_1, \ldots, x_n)\ ; \tag{26.5}$$

$$f\ (x_1, \ldots, x_n, S(y)) = h\ (x_1, \ldots, x_n, y, f\ (x_1, \ldots, x_n, y))\ . \tag{26.6}$$

   If g and h are primitive recursive, then f is defined to be primitive recursive.

Here (26.5) is the *basis* and (26.6) is the *recursion step.* It is conventional to call $x_1, \ldots, x_n$ the *parameters* and y the *recursion variable.* From a computational viewpoint, the scheme is easy to interpret. Given integer values for variables $x_1, \ldots, x_n$ and z, how can we evaluate $f(x_1, \ldots, x_n, z)$? We start building a table T in which each row y contains the value of $f(x_1, \ldots, x_n, y)$. The basis step gives us the top row via $T[0] = f(x_1, \ldots, x_n, 0) = g(x_1, \ldots, x_n)$. Whenever we have filled a row y, we can fill the next row via the recursion step, via $T[y + 1] = f(x_1, \ldots, x_n, S(y)) = h(x_1, \ldots, x_n, y, T[y])$. As soon as row z is

filled, using y such that $z = S(y)$, we are done. The point is that provided g and h are computable, the function f is also computable. Functional composition likewise preserves computability. Moreover, since the basic functions are all total and produce nonnegative values, every function that we can build up in this manner is also total and produces nonnegative values.

DEFINITION 26.1 A function is said to be **primitive recursive** if it can be built up from the successor, zero and projection functions by a finite number of applications of composition and primitive recursion.

EXAMPLE 26.1:

To show how the scheme of primitive recursion models the informal recursion defining the function f of one variable (so we have $n = 0$) in Eq. (26.3), take "$g()$" to be the constant 0, and take h to be the two-variable function $h(y, z) = z + 2y + 1$, which happens to be our example of "explicit definition" in (26.1). Then we have $f(0) = 0$ and

$$f(S(y)) = h(y, f(y)) = f(y) + 2 * y + 1 .$$

With "$x - 1$" in place of "y" and "x" in place of "$S(y)$," this is the same as (26.3). We will return to this notational difference later.

As the prefix form (26.2) indicates, h itself can be built up via functional composition from the *plus* and *times* functions. It is interesting to see how the usual functions of arithmetic can themselves be constructed from the rather Spartan basis we have been given. To begin with, the constants $1, 2, \ldots$ are formally introduced by functional composition, with "$g_1()$" as the constant 0 and "h" as the successor function, via $1 = S(0)$, $2 = S(1) = S(S(0))$, $3 = S(2)$, and so on.

EXAMPLE 26.2:

Addition. Take $g(x) = x$ and $h(x, y, z) = S(z)$. Formally, g is the basis function p_1^1, and h is the functional composition of the successor function with p_3^3. Then primitive recursion gives us $plus\,(x, 0) = g(x) = x$ and

$$plus\,(x, S(y)) = h(x, y, plus\,(x, y)) = S(plus\,(x, y)) = S(x + y) = x + y + 1 ,$$

as we would demand. Hence this formal definition of *plus* correctly computes addition, and we may use the standard "+" notation in the formal examples that follow.

EXAMPLE 26.3:

Multiplication. Take $g(x) = 0$ and $h(x, y, z) = x + z$. Formally, g is the zero function (of one variable rather than the constant zero), and h is the functional composition of *plus* with the two functions p_1^3 and p_3^3 (so $k = 2$ here). Then primitive recursion gives us $times\,(x, 0) = g(x) = 0$ and

$$times\,(x, S(y)) = h(x, y, times\,(x, y)) = x + times\,(x, y) = x * (y + 1) ,$$

again as we would demand. Hence this formal definition of *times* correctly computes multiplication. Note that we had to go to some length (of making h a function of 3 variables) so that our definition exactly agrees with the formal requirements in Eq. (26.6).

EXAMPLE 26.4:

Exponentiation. Take $g(x) = 1$ and $h(x, y, z) = x * z$. Formally, g is the one-variable function that always outputs 1 and is defined by composing S and the zero function Z, while h is the same as in Example 26.3 but with *times* in place of *plus*. Then primitive recursion gives us $exp\,(x, 0) = g(x) = 1$ (note that even 0^0 equals 1) and

$$exp\,(x, S(y)) = h(x, y, exp\,(x, y)) = x * exp\,(x, y) = x^{y+1}\ .$$

Once again the correctness of this definition for all values of x and y is easy to verify, via a simple proof by induction that follows the recursion.

It is now straightforward to omit some of the formal apparatus and write the definitions more succinctly. For instance, the last example becomes

$$\begin{aligned} exp\,(x, 0) &= 1 \\ exp\,(x, y+1) &= x * exp\,(x, y)\ . \end{aligned}$$

This resembles a program one would actually write, especially in a language like C that does not provide exponentiation as a built-in operator.

At this point the alert reader, noting the way our schemes all involve nonnegative numbers, will first wonder how on earth we can ever define *subtraction* this way. The key is that the syntax of primitive recursion allows us to define a function $P(y)$ that computes "proper subtraction by 1," and then use P to define *proper subtraction* itself. The word "proper" here means that any negative value is replaced by 0, in order to maintain our restriction to the nonnegative numbers. The definitions are

$$\begin{aligned} P(0) &= 0 \\ P(S(y)) &= y \\ sub\,(x, 0) &= x \\ sub\,(x, S(y)) &= P(sub\,(x, y)) \end{aligned}$$

For P we took $h(y, z) = y$, i.e., $h = p_1^2$, and for *sub* we took $h(y, z) = P(z)$. To trace this out, $sub\,(3, 2) = P(sub\,(3, 1)) = P(P(sub\,(3, 0)) = P(P(3)) = P(2) = 1$, and $sub\,(2, 3) = P(P(P(2))) = P(0) = 0$, which is the "proper" value.

Second, the reader may have felt uncomfortable defining functions in terms of "$S(y)$" rather than "y." For example, the primitive recursion for the factorial function, with 0! standardly defined to be 1, gives us

```
fact(0) = 1 | fact(y+1) = (y+1)*fact(y);
```

here "`|`" separates the base and recursion cases. This would actually be valid syntax in the programming language ML *except* that "`fact(y+1)`" is an illegal function header. The syntax of ML forces one to write it this way:

```
fact(0) = 1 | fact(y) = y*fact(y-1);
```

this is literally the example used in many texts. To make the formal Eq. (26.6) for primitive recursion reflect the syntax of programming languages, we can use P in place of S to change it to

$$f\,(x_1, \ldots, x_n, y) = h\,(x_1, \ldots, x_n, y, f\,(x_1, \ldots, x_n P(y)))\ , \tag{26.7}$$

and alternately make the middle argument of h be $P(y)$ instead of y. Either way, one might then expect to be able to recover the function S by defining it in terms of P and the other two basis functions, just as we defined P in terms of S above. However, this is impossible—one could never define any increasing functions at all. This curious asymmetry partly explains why primitive recursion was defined the way

it is. Nevertheless, if S as well as P is provided in the basis, then one can use the modified definition and obtain exactly the same class of primitive recursive functions. For instance, addition is definable by $plus\,(x, 0) = x \mid plus\,(x, y) = S(plus\,(x, P(y)))$, and so on. Hence primitive recursion is for the most part exactly what ML and other functional languages *do*.[2]

Finally, the reader may wonder what has become of functions defined on *strings.* A string over an alphabet Σ can always be identified with its number in the standard lexicographic enumeration of Σ^*, with ϵ corresponding to 0. Then a string function $f : \Sigma^* \to \Sigma^*$ can be called primitive recursive if the corresponding numerical function (of one variable) is primitive recursive. For instance, the function that appends a '1' to a binary string x corresponds to $2x + 2$. Cutting the other way, under some transparent encoding of negative and rational and complex numbers (etc.) by strings, one can extend the concept of primitive recursion to define addition and multiplication and nearly all familiar mathematical functions in their full generality. The meaning and proof of the following statement should now be clear; full detail can be found in [27].

THEOREM 26.1 *Every primitive recursive function is computable by a Turing machine.*

The converse is false, however. A famous example of a computable total function that is not primitive recursive is *Ackermann's function;* this and other examples may be found in [19]. To obtain all computable functions we need to introduce one more scheme of recursion—at the inevitable cost, however, of opening a Pandora's box of functions that are no longer total.

μ-Recursive Functions

We will add a new operation called *minimalization* that does not preserve totality. Again we restrict numerical arguments to be non-negative integers.

DEFINITION 26.2 A possibly-partial function f of n variables is defined by **μ-recursion** from a function g of $n + 1$ variables, written

$$f\,(x_1, \ldots, x_n) = \mu y.g\,(x_1, \ldots, x_n, y)\ ,$$

if whenever $f(x_1, \ldots, x_n)$ is defined, it equals the least number y such that $g(x_1, \ldots, x_n, y) = 1$. If $f(x_1, \ldots, x_n)$ is undefined, there must be no y such that $g(x_1, \ldots, x_n, y) = 1$. The class of **$\mu$-recursive functions** is the class of all functions that can be built up from the successor, zero, and projection functions by the operations of composition, primitive recursion, and μ-recursion.

The computation of $f(x_1, \ldots, x_n)$ that is implicit in Definition 26.2 can be described by building a table as before. First fill in the row $T[0] = g(x_1, \ldots, x_n, 0)$, then $T[1] = g(x_1, \ldots, x_n, 1)$, and so on. If and when one finds a y whose value $T[y]$ equals 1, halt and output y. The "if" is the big difference from the algorithm for primitive recursion, because if $g(x_1, \ldots, x_n, y)$ never takes the value 1, this procedure will never halt. This procedure is called an *unbounded search.* Compared another way to primitive recursion, μ-recursion increments its recursion variable rather than decrement it.

[2] Primitive recursion has its counterpart in imperative languages as well, aside from the fact that most of them support recursion directly. The "table $T[y]$" computation above shows how primitive recursion can be simulated by a simple for-loop **for** $y = 0$ **to** z **do**. . . **end** that fixes its bounds and never alters y in the loop body. A theorem [18] in programming languages states that the primitive recursive functions are exactly the total functions computable by programs that use only if-then-else and simple nested for-loops.

There is nothing special about "= 1" here: zero or any other constant could be used instead. Our use of 1 suggests the special case in which g is a total function that takes on only the values 0 and 1. Then we can regard its output as a Boolean truth value, with 1 = true and 0 = false, and call g a *predicate.* The class of μ-recursive functions is not changed under the restriction that g be a predicate. Then we can read the syntax "$\mu y.g(x_1, \ldots, x_n, y)$" in English as "the least y such that $g(x_1, \ldots, x_n, y)$ is true." From all this we can see that whereas primitive recursion corresponds to a **for** loop, μ-recursion corresponds to a **while** loop, with $g(\ldots)$ as the test condition.

EXAMPLE 26.5:

Partial square-root function. Define the predicate $g(x, y)$ to hold if and only if $x = y^2$. Then the function f defined for all x by $f(x) = \mu y.g(x, y)$ computes the square root of x when x is a perfect square. When x is not a perfect square, however, the recursion is undefined, so f is a partial recursive function.

EXAMPLE 26.6:

Linear programming. The standard *simplex algorithm* uses a **while** loop that executes a basic *pivot step* until a predicate expressing optimality holds. Hence the function that embodies the solution to a linear programming problem is μ-recursive. In point of fact, because a bound on the number of polytope corner points is explicitly definable from the problem instance, the same function can be computed via a simple **for** loop, so it is primitive recursive. However, the former method is usually much faster.

Part (a) of the next theorem expresses the fact that **while** loops, together with **if-then-else,** suffice to make a general-purpose programming language. Part (b) is the gist of the famous theorem, credited in various forms to various sources, that *at most one* **while** *loop is needed in any program.*

THEOREM 26.2

(a) A (partial) function is μ-recursive if and only if it is a Turing-computable (partial) function.

(b) Moreover, given any Turing machine T, we can find a primitive recursive function u and a primitive recursive predicate t such that for all x, $T(x) = u(\mu y.t(x, y))$.

In the standard proof of part (b), the predicate $t(x, y)$ is designed to hold if and only if y encodes the sequence of configurations of a halting computation of T on input x, and the function u picks off the output from the final configuration. To complete the proof of (a), all one needs to show is that given a Turing machine that computes g in Definition 26.2, one can build a Turing machine that computes f. This is done by following the unbounded-search procedure sketched above.

The corresponding theorem for formal languages also merits mention here. In Chapter 24 we defined the *characteristic function* of a language L to be the function f_L defined for all x by $f_L(x) = 1$ if $x \in L$, $f_L(x) = 0$ if $x \notin L$. This is simply the predicate corresponding to membership in L. The *partial characteristic function* still takes the value 1 when $x \in L$, but is undefined when $x \notin L$.

THEOREM 26.3

(a) A language is recursive if and only if its characteristic function is μ-recursive.

(b) A language is r.e. if and only if its partial characteristic function is μ-recursive.

Part (a) explains how the term "recursive" became applied to languages and predicates as a synonym for

"decidable." It is important to recall that not all languages L accepted by Turing machines have computable characteristic functions (i.e., are decidable); unless we find a Turing machine accepting L that halts for all inputs, all we know is that the *partial* characteristic function of L is (partial-) computable. Before proceeding to *un*decidable languages, we take time to interpret these two theorems and others presented in Chapters 24 and 25.

26.4 Equivalence of Computational Models and the Church–Turing Thesis

In Chapter 24 we introduced various machine models, the most important of which is the Turing machine. In Chapter 25 we introduced the grammar hierarchy of Chomsky, of which the most powerful was the Type-0 grammar. Here we have presented the purely mathematical model of μ-recursive functions. Although these models were defined over different domains for different purposes, they are all equivalent in a precise technical sense—they all define the same class of computable functions and decidable languages, and the same class of partial computable functions and partially decidable languages. We can summarize all this by saying that Turing machines, type-0 grammars, and μ-recursive functions have the same problem-solving power.

This equivalence extends to vastly many other computational models, of which we mention a few:

(1) *Cellular Automata.* Cellular automata are intended to model the evolution of a colony of microorganisms. Each cell is a deterministic finite automaton that receives its input in discrete time steps from neighboring cells, so that its current state is defined by its own previous state and the previous states of its neighbors. All the cells execute the same DFA. There are different schemes for specifying the representation of the input to a cellular automaton and its output. But under any reasonable scheme, the largest class of problems that can be solved on cellular automata coincides with the class of **solvable problems** on a Turing machine.

(2) *String-Rewriting Systems.* A string-rewriting system is similar to a grammar. The main difference is that there are no nonterminals. Let the input alphabet be Σ. The production rules of a rewriting system T will be of the form $\alpha \rightarrow \beta$ where α and β are strings over Σ. One can apply such a rule by replacing any occurrence of α in a string by β. T is defined as a finite set of rewrite rules, along with a finite set of initial strings. The language generated by T is defined as the set of strings that can be obtained from an initial string by applying the rewrite rules a finite number of times. The systems proposed before 1930 by Thue and Post fall roughly into this category. It turns out that the class of string-rewriting languages is the same as the r.e. languages (see [1]).

(3) *Tree-Rewriting Systems.* These are similar to string-rewriting systems except that the local edits are done on subtrees of a tree, and rules may have more than one argument. The subtrees typically represent *terms* in algebraic or logical expressions that are being operated on. Under reasonable schemes for encoding numbers or strings by trees, all known tree-rewriting systems generate r.e. languages or compute partial recursive functions. Church's λ-*calculus* and most formal systems of logic fall into this category.

(4) *Extensions of Turing's Model.* As mentioned in Chapter 24, one can also create numerous modifications to the basic Turing machine model, such as having multi-dimensional tapes or binary trees with MOVE UP, MOVE DOWN LEFT, and MOVE DOWN RIGHT instructions (the latter are tantamount to having random-access to stored values), allowing nondeterminism or alternation, making computation probabilistic (see Chapter 29, Section 29.2), and so on. All of these machines compute the same functions as the simple one-tape Turing machine.

(5) *Random-Access Machines and High-Level Programming Languages.* These can be mentioned in tandem because a RAM, as described in Chapter 24, is just an idealization of assembly or

machine language. Every high-level language yet devised can be compiled into some machine language. Even the standard *Java Virtual Machine* is little more than a RAM, with some added handling of class objects via pointers that is not unlike the workings of a pointer machine, and some hooks to enable the host system to control physical devices and network communications. Without excessive effort one can extend the construction of a **universal Turing machine** in Section 26.2 to handle the case where P is a RAM program rather than a GOTO program. The registers of P can be simulated on the tape by adding one more tape symbol # and using strings of the form $\#i\#j\#$, where i is the register's number and j is its contents. As stated in Chapter 24, Section 24.4, this simulation is even fairly efficient. Hence all these high-level languages have the same problem-solving power as the lowly one-tape Turing machine.

The convergence of so many disparate formal models on the same class of languages or functions is the main evidence for the assertion that they all exactly capture the informal notion of what is mechanically or humanly computable. This assertion is called the **Church–Turing thesis.** In one form, it asserts that every problem that is humanly solvable is solvable by a Turing machine. Put more precisely, any cognitive process that a human being could or will ever use to distinguish certain numbers or strings as "good" defines an r.e. language—and if it also would determine that any other given number or string is "bad," it defines a **recursive language.** An extension of the thesis claims that no one will ever design a physical device to compute functions that are not μ-recursive. The Church–Turing thesis is not a mathematical conjecture and is not subject to mathematical proof; it is not even clear whether the extension is resolvable scientifically.

26.5 Undecidability

The Church–Turing thesis implies that if a language is undecidable in the formal sense defined above, then the problem it represents is really, humanly, physically undecidable. The existence of languages that are not even partially decidable can be established by a counting argument: Turing machines can be counted 1, 2, 3, . . ., but the mathematician Georg Cantor proved that the totality of all sets of integers cannot be so counted. Hence there are sets left over that are not accepted, let alone decided, by any program. This argument, however, does not apply to languages or problems *that one can state,* since these are also countable. The remarkable fact is that many easily-stated problems of high practical relevance are undecidable. This section shows that the five remaining problems on our list in Section 26.2, namely 2 through 5 and 7, are all unsolvable.

Diagonalization and Self-Reference

Undecidability is inextricably tied to the concept of *self-reference,* and so we begin by looking at this perplexing and sometimes paradoxical concept. The simplest examples of self-referential paradox are statements such as "This statement is false" and "Right now I am lying." If the former statement is true, then by what it says, it is false; and if false, it is true. . . . The idea and effects of self-reference go back to antiquity; a version of the latter "liar" paradox ascribed to the Cretan poet Epimenides even found its way into the New Testament, Titus 1:12–13. For a more colorful example, picture a barber of Seville hanging out an advertisement reading, "I shave those who do not shave themselves." When the statement is applied to the barber himself, we need to ask: Does he shave himself? If *yes,* then he is one of those who do shave themselves, which are not the people his statement says he shaves. The contrary answer *no* is equally untenable. Hence the statement can be neither true nor false (it may be good ad copy), and this is the essence of the paradox. Such paradoxes have made entry into modern mathematics in various forms. We will present some more examples in the next few paragraphs. Many variations on the theme of self-reference can be found in the books of the logician and puzzlist Raymond Smullyan, including [25] and [26].

Berry's paradox concerns English descriptions of natural numbers. For example, the number 24 can be described by many different phrases: "twenty-four," "six times four," "four factorial," etc. We are interested in the *shortest* of such descriptions, namely one(s) having the fewest letters. Here, "two dozen" beats all of the above. Clearly there are (infinitely) many positive integers whose shortest descriptions require one hundred letters or more. (A simple counting argument can be used to show this. The set of positive integers is infinite, but the set of positive integers with English descriptions of fewer than one hundred letters is finite.) Let D denote the set of positive integers that do not have English descriptions of fewer than one hundred letters. Thus D is not empty. It is a well-known fact in set theory that any nonempty subset of positive integers has a smallest integer. Let x be the smallest integer in D. Does x have an English description of fewer than one hundred letters? By the definition of the set D and x, the answer is *yes:* such a description of x is, "the smallest positive integer that cannot be described in English in fewer than one hundred letters." This is an absurdity, because the quoted part of the last sentence is clearly a description of x, and it contains fewer than one hundred letters.

Russell's paradox similarly turns on issues in defining sets. In formal mathematics, we can perfectly easily describe "the set of all sets that do not include themselves as elements" by the definition $S = \{x | x \notin x\}$. The question "Is $S \in S$?" leads to a real conundrum. This also resembles the barber paradox, with "$\notin$" read as "does not shave." This paradox forced the realization that the formal notion of a *set,* and importantly the formal rules that apply to sets, do not and cannot apply to everything that we informally regard as being a "set."

Our last example is a charming paradox named for the mathematician William Zwicker. Consider the collection of all two-person games that are *normal* in the sense that every play of the game must end after a finite number of moves. Tic-tac-toe is normal since it always ends within nine moves, while chess is normal because the official "fifty move rule" prevents games from going on forever. Now here is *hypergame.* In the first move of hypergame, the first player calls out a normal game—and then the two players go on to play that game, with the second player making the first move. Now we need to ask, "Is hypergame normal?" If *yes,* then it is legal for the first player to call out "hypergame!"—since it is a normal game. By the rules, the second player must then play the first move of hypergame—and this move can be calling out "hypergame!" Thus the players can keep saying "hypergame" without end, but this contradicts the definition of a normal game. On the other hand, suppose hypergame is not normal. Then in the first move, player 1 cannot call out hypergame and must call a normal game instead—so that the infinite move sequence given above is not possible and hypergame is normal after all!

Let us try to implement Zwicker's paradox. To play hypergame, we need a way of formalizing and encoding the rules of a game as a string x, and we need a decision procedure `isNormal(x)` to tell if the game is normal. Then the rules of hypergame are easily formalized: pick a string x, verify `isNormal(`x`)`, and play game x. Let h be the string encoding of these rules. Now we get a real contradiction when `isNormal(`h`)` is run. We must conclude that either (i) our formalization of games is inadequate or inconsistent, or (ii) a decision procedure `isNormal` simply cannot exist. Now (i) is the way out for Russell's paradox with "sets" in place of "games." For *computation,* however, we know that our formalization is adequate and consistent—and hence we will be faced with conclusions of (ii), namely that our corresponding computational problems are unsolvable.

Before showing how the above paradoxes can be modified and ingrained into our problems, we need to review the 0-1 encoding of GOTO programs from Section "A Universal Program," including the conventions that ACCEPT has the same code 100 as STOP for programs that accept languages, and that such an ACCEPT statement be last and unique. We may assign the code 101 to REJECT, which may appear anywhere. If a binary string x encodes a program P, it is easy to decode x into P, and we may identify x with P. If x does not encode a legal GOTO program, this fact is easy to detect. Then we may choose to treat x as an alternate code for the trivial GOTO program that consists of a single REJECT statement.

Now we can define the so-called "diagonal language" L_d as follows:

$$L_d = \{x \mid x \text{ is a GOTO program that does not accept the string } x\} \tag{26.8}$$

This language consists of all programs in the GOTO language that do not halt in the ACCEPT statement when given their own encoding as input—they may either REJECT or not halt at all on that input. For example, consider $x = 01111101101100$, which encodes a program that accepts any string beginning with 1 and rejects any string beginning with 0. Then $x \in L_d$ since the program does not accept 01111101101100. Note the self-reference in (26.8). Although the definition of L_d seems artificial, its importance will become clear when we use it to show the undecidability of other problems. First we prove that L_d is not even accepted by any Turing machine, let alone decided by one.

THEOREM 26.4 *L_d is not recursively enumerable.*

PROOF Suppose for the sake of contradiction that L_d is r.e. Then there is a GOTO program that accepts L_d—call it P. Now what does P do on input $x = code\,(P)$? If P accepts x, then x is not in L_d, but this contradicts $L(P) = L_d$. But if P does not accept x, then x *is* in L_d, and this also contradicts $L(P) = L_d$. Hence a program P such that $L(P) = L_d$ cannot exist, and so L_d is not r.e.

The definition of L_d is motivated by Russell's paradox, reading "$\notin$" as "does not accept." Whereas in Russell's paradox we had to conclude that S is not a *set,* here we conclude that L_d is not a *Turing-acceptable* set.

We can similarly carry over Zwicker's paradox by treating a given string x as formally defining "Game-x" as follows: The first player decodes x into a GOTO program P, and then tries to choose some string x' in the language $L(P)$. If $L(P)$ is empty, in particular if x decodes to the trivial program "1. REJECT" as stipulated above, then the game ends then and there. But if the first player finds such an x', then the second player must play the same way with x'. Then we can say that x is *normal* if every play of Game-x must terminate (by reaching a GOTO program that accepts the empty language) in a finite number of steps. Finally define L_Z to be the set of normal strings. By applying the reasoning from Zwicker's paradox, one can imitate the above proof to show that L_Z is not recursively enumerable.

Reductions and More Undecidable Problems

Recall from Chapter 24 (Section 24.3) the notion of Turing reducibility. Basically, a language L_1 is Turing reducible to L_2 if there is a halting Turing machine for language L_1 using an oracle for language L_2. If L_1 is reducible to L_2 and L_2 is decidable, then so is L_1. This is because one can replace queries to oracles by executing a halting computation for L_2. The contrapositive of this statement can be used to show undecidability. If L_1 is undecidable, then so is L_2. We will first express Problem (2) as a language:

$$L_U = \{code(P)111x \,|\, P \text{accepts the string } x\}.$$

Thus L_U takes as input a program in GOTO, and a binary string x, and accepts the encoded pair (P, x) if and only if P accepts x. (Note 111 is used as a separator between P and x.) The universal program presented in "A Universal Program" accepts the language L_U hence it is recursively enumerable. We will show that L_U is not recursive. First, we will show a simple fact about recursive languages.

THEOREM 26.5 *Recursive languages are closed under complement.*

PROOF Let P be a GOTO program for language L. The program P' obtained by interchanging the ACCEPT and REJECT instructions is easily seen to accept the language $\overline{L}$. This standard trick works to complement the computations of most of the deterministic devices (such as DFA).

Now we show that L_U is not recursive.

THEOREM 26.6 *L_U is not recursive.*

PROOF Consider the language $L'_U = \{x|\ x$ when interpreted as a GOTO program accepts its own encoding$\}$. Obviously, $L'_U = \overline{L}_d$. Since L_d is not recursively enumerable, it is not recursive. (Recall that the set of recursive languages is a subset of **recursively enumerable languages.**) By the above theorem, L'_U is not recursive. Finally, note that L'_U can be reduced to L_U as follows. Given an algorithm for L_U, we can construct an algorithm for L'_U as follows: Let P be an algorithm for L_U. To construct an algorithm for L'_U simply note the connection between the two problems. An input string x belongs to L'_U if and only if $x111x$ belongs to L_U. Thus, a simple copy program (similar to one presented in Section 26.2) can be first used to convert the input x into $x111x$. Move the scanning head back to the leftmost character of the first copy of x. Now simply run the program P. Note that the program P' described above is being constructed using only P, not x. This reduction shows that L_U is not recursive.

Next we consider problem (3) in our list. Earlier we showed that a special case of this problem (when the input is restricted to type-1 grammar) is totally solvable. It is not hard to see that the general problem is partially solvable. (To see this, suppose there is a derivation for a string x starting from S, the start symbol of the grammar. Suppose the length of one such derivation is k. A program can try all possible derivations of length 1, 2, etc., until it succeeds. Such a program will always halt on strings x generated by the grammar G. Thus the language

$$L_0 = \{G\#x|G \text{is a type-0 grammar and } x \text{ can be generated by } G\}$$

is recursively enumerable. A standard result from formal language theory [12] is that for every Turing machine M, there is a type-0 grammar G such that $L(M) = T(G)$. This conversion from M to G is the reduction that shows that the language is not recursive.

The string compression problem, numbered 4 on our list, is not a decision problem, but reduction techniques can still be used to show that it is unsolvable. We refer the reader to [17] for details.

By a fairly elaborate reduction (from L_d), it can be shown that the tiling problem (5) in our list is also not partially decidable. We will not do it here and refer the interested reader to [8]. But we would like to point out how the undecidability result can be used to infer a result about *aperiodic tilings.* This deduction is interesting because the result appears to have some deep implications and is hard to deduce directly. We need the following discussion before we can state the result. A different way to pose the tiling problem is whether a given set of tiles can tile *an entire plane* in such a way that all the adjacent tiles have the same color on the meeting quarter. (Note that this question is different from the way we originally posed it: Can a given set of tiles tile any *finite* rectangular region? Interestingly, the two problems are identical in the sense that the answer to one version is "yes" if and only if it is "yes" for the other version.) Call a tiling of the plane *periodic* if one can identify a $k \times k$ square such that the entire tiling is made by repeating this $k \times k$ square tile. Otherwise, call it *aperiodic.* Consider the question: Is there a (finite) set of unit tiles that can tile the plane, but only aperiodically? The answer is "yes" and it can be shown from the total undecidability of the tiling problem. Suppose the answer is "no." Then, for any given set of tiles, the entire plane can be tiled if and only if the plane can be tiled periodically. But such a periodic tiling can be found, if one exists, by trying to tile a $k \times k$ region for successively increasing values of k. This process will eventually succeed (in a finite number of steps) if the tiling exists. This would make the tiling problem partially decidable, which contradicts the total undecidability of the problem. This means that the assumption that the entire plane can be tiled if and only if some $k \times k$ region can be tiled is wrong. Thus there must exist a finite set of tiles that can tile the entire plane, but only aperiodically.

We conclude with a brief remark about problem 7 in our list. After many years of effort by several mathematicians and computer scientists (including Davis and Robinson), Matiyasevich found an effective way to transform a given Turing machine T into a set of equations in variables $x, y_1, \ldots, y_m$ such that for any x, T on input x halts if and only if the other m variables can be set to solve the equations. This reduction shows that Hilbert's tenth problem is undecidable. Details behind this reduction can be found in [6].

26.6 Defining Terms

Decision problem: A computational problem with a yes/no answer. Equivalently, a function whose range consists of two values {0, 1}.

Decidable problem: A decision problem that can be solved by a GOTO program that halts on all inputs in a finite number of steps. For emphasis, the equivalent term *totally decidable* problem is used. The associated language is called *recursive.*

Partially decidable problem: A decision problem that can be solved by a GOTO program that halts (and outputs ACCEPT) on all yes-instances. The program may or may not halt on no-instances. Equivalently, the collection of yes-instance strings forms a type-0 language. (See Chapter 25.)

Recursively enumerable language: Same as partially decidable language.

μ-recursive function: A function that is a basic function (Zero, Successor or Projection), or one that can be obtained from other μ-recursive functions using composition and μ-recursion.

Recursive language: A language that can be accepted by a GOTO program that halts on all inputs. The associated problem is called *decidable.*

Solvable problem: A computational problem that can be solved by a halting GOTO program. The problem may have a nonbinary output.

Totally undecidable problem: A problem that cannot be solved by a GOTO program. Equivalently, one for which the set of yes-instance strings is not a type-0 language.

Undecidable problem: A decision problem that is not (totally) decidable. It could be partially decidable or totally undecidable.

Universal Turing machine: A Turing machine that can simulate any other Turing machine.

Unsolvable problem: A computational problem that is not solvable. The associated function is called an uncomputable function.

References

[1] Book, R. and Otto, F., *String Rewriting Systems,* Springer-Verlag, Berlin, 1993.

[2] Chomsky, N., Three models for the description of language, *IRE Trans. on Information Theory,* 2(2), 113–124, 1956.

[3] Chomsky, N., Formal properties of grammars. In *Handbook of Math. Psych.,* Vol. 2, 323–418. John Wiley & Sons, New York, 1963.

[4] Davis, M., *Computability and Unsolvability,* McGraw-Hill, New York, 1958.

[5] Davis, M., What is computation? In *Mathematics Today–Twelve Informal Essays,* Steen, L., Ed. 241–259, 1980.

[6] Floyd, R.W. and Beigel, R., *The Language of Machines: An Introduction to Computability and Formal Languages,* Computer Science Press, New York, 1994.

[7] Gurari, E., *An Introduction to the Theory of Computation,* Computer Science Press, Rockville, MD, 1989.

[8] Harel, D., 1992. *Algorithmics: The Spirit of Computing,* Addison-Wesley, Reading, MA, 1992.
[9] Harrison, M., *Introduction to Formal Language Theory,* Addison-Wesley, Reading, MA, 1978.
[10] Hartmanis, J., On computational complexity and the nature of computer science. *Communications of the ACM,* 37(10), 37–43, 1994.
[11] Hartmanis, J. and Stearns, R., On the computational complexity of algorithms. *Trans. Amer. Math. Soc.,* 117, 285–306, 1965.
[12] Hopcroft, J. and Ullman, J., *Introduction to Automata Theory, Languages and Computation.* Addison-Wesley, Reading, MA, 1979.
[13] Jones, N.D., *Computability and Complexity from a Programming Perspective.* MIT Press, Cambridge, MA, 1997.
[14] Kleene, S., Representation of events in nerve nets and finite automata. In *Automata Studies,* 3-41. Princeton University Press, NJ, 1956.
[15] Kohavi, Z., *Switching and Finite Automata Theory.* McGraw-Hill, 1978.
[16] Kolmogorov, A. and Uspenskii, V., On the definition of an algorithm. *Uspekhi Mat. Nauk.,* 13, 3–28, 1958.
[17] Li, M. and Vitányi, P., *An Introduction to Kolmogorov Complexity and Its Applications.* Springer-Verlag, 1993; 2nd edition, 1997.
[18] Meyer, A. and Ritchie, D., The complexity of LOOP programs. *Proc. 22nd ACM National Conf.,* 465–469, 1967.
[19] McNaughton, R., *Elementary Computability, Formal Languages and Automata.* ZB Publishing Industries, Lawrence, KS, 1993,
[20] Minsky, M., *Computation: Finite and Infinite Machines.* Prentice Hall, Englewood Cliffs, NJ, 1967.
[21] Post, E., Formal reductions of the general combinatorial decision problems. *Amer. J. Math.,* 65, 197–215, 1943.
[22] Robinson, Minsky's small universal Turing machine. *International Journal of Mathematics,* 2(5), 551–562, 1991.
[23] Rogers, H., *Theory of Recursive Functions and Effective Computability.* MIT Press, Cambridge, MA, 1967.
[24] Sipser, M., *Introduction to the Theory of Computation,* 1st ed., PWS, Boston, MA, 1996.
[25] Smullyan, R., *What is the Name of this Book?* Prentice-Hall, Englewood Cliffs, NJ, 1978.
[26] Smullyan, R., *Satan, Cantor and Infinity.* Alfred A. Knopf, New York, 1992.
[27] Sudkamp, T., *Languages and Machines An Introduction to the Theory of Computer Science,* Addison-Wesley Longman, 1997.
[28] Turing, A., On computable numbers with an application to the Entscheidungsproblem. *Proc. London Math. Soc., series 2,* 42, 230–265, 1936.
[29] Wood, D., *Theory of Computation,* Harper and Row, 1987.
[30] Wang, H., Proving theorems by pattern recognition. *Bell System Technical Journal,* 40, 1–42, 1961.

Further Information

The fundamentals of computability can be found in many books including the classic texts [4, 20, 23]. More-recent books on automata and formal languages have also devoted at least a few chapters to computability [6, 7, 8, 9, 12, 24, 29]. Early work on computability was motivated by a quest to address profound questions about the basis of logical reasoning, mathematical proofs and automatic computation. Various formalisms discussed in this chapter were proposed at around the same time, and soon thereafter, their equivalence was tested. Thus, in a short time, the Church-Turing thesis took deep roots. Subsequent work has focused on whether specific problems are decidable or not. Another direction of research has been

to make finer distinctions among unsolvable problems by introducing degrees of unsolvability. Recursive function theory and lambda calculus also led to the development of functional programming languages such as Lisp, Scheme, Haskell, and ML. Computability theory is also closely related to logic, formal deductive systems, and complexity theory. Logic and deductive systems are of interest to philosophers and researchers in artificial intelligence, as well as to computation theorists. Although there are no journals devoted exclusively to computability, many theory journals (such as those listed at the end of Chapters 24 and 25) publish papers on this topic. In addition, *Annals of Pure and Applied Logic* publishes papers on logic and computability.

27

Complexity Classes[1]

Eric Allender
Rutgers University

Michael C. Loui
University of Illinois at Urbana-Champaign

Kenneth W. Regan
State University of New York at Buffalo

27.1 Introduction

The purposes of complexity theory are to ascertain the amount of computational resources required to solve important computational problems, and to classify problems according to their difficulty. The resource most often discussed is computational time, although memory (space) and circuitry (or hardware) have also been studied. The main challenge of the theory is to prove lower bounds, i.e., that certain problems cannot be solved without expending large amounts of resources. Although it is easy to prove that inherently difficult problems exist, it has turned out to be much more difficult to prove that any *interesting* problems are hard to solve. There has been much more success in providing strong evidence of intractability, based on plausible, widely held conjectures. In both cases, the mathematical arguments of intractability rely on the notions of *reducibility* and *completeness*—which are the topics of Chapter 28. Before one can understand reducibility and completeness, however, one must grasp the notion of a *complexity class*—and that is the topic of this chapter.

First, however, we want to demonstrate that complexity theory really can prove—to even the most skeptical practitioner—that it is hopeless to try to build programs or **circuits** that solve certain problems.

[1]Eric Allender — Supported by the National Science Foundation under Grant CCR-9509603. Portions of this work were performed while a visiting scholar at the Institute of Mathematical Sciences, Madras, India.
Michael C. Loui — Supported by the National Science Foundation under Grant CCR-9315696.
Kenneth W. Regan — Supported by the National Science Foundation under Grant CCR-9409104.

As our example, we consider the manufacture and testing of logic circuits and communication protocols. Many problems in these domains are solved by building a logical formula over a certain vocabulary, and then determining whether the formula is logically valid, or whether counterexamples (that is, bugs) exist. The choice of vocabulary for the logic is important here, as the next paragraph illustrates.

One particular logic that was studied by Stockmeyer [42] is called WS1S. (We need not be concerned with any details of this logic.) Stockmeyer showed that any circuit that takes as input a formula with up to 616 symbols and produces as output a correct answer saying whether the formula is valid, requires at least 10^{123} gates. According to Stockmeyer [43], "Even if gates were the size of a proton and were connected by infinitely thin wires, the network would densely fill the known universe."

Of course, Stockmeyer's theorem holds for one particular sort of circuitry, but the awesome size of the lower bound makes it evident that, no matter how innovative the architecture, no matter how clever the software, no computational machinery will enable us to solve the validity problem in this logic. For the practitioner testing validity of logical formulas, the lessons are (1) be careful with the choice of the logic, (2) use small formulas, and often (3) be satisfied with something less than full validity testing.

In contrast to this result of Stockmeyer, most lower bounds in complexity theory are stated asymptotically. For example, one might show that a particular problem requires time $\Omega(t(n))$ to solve on a Turing machine, for some rapidly growing function t. For the Turing machine model, no other type of lower bound is possible, because Turing machines have the linear-speed-up property (see Chapter 24, Theorem 24.5). This property makes Turing machines mathematically convenient to work with, since constant factors become irrelevant, but it has the by-product—which some find disturbing—that for any n there is a Turing machine that handles inputs of length n in just n steps by looking up answers in a big table. Nonetheless, these asymptotic lower bounds essentially always can be translated into concrete lower bounds on, say, the number of components of a particular technology, or the number of clock cycles on a particular vendor's machine, that are required to compute a given function on a certain input size.[2]

Sadly, to date, few general complexity–theoretic lower bounds are known that are interesting enough to translate into concrete lower bounds in this sense. Even worse, for the vast majority of important problems that are believed to be difficult, no nontrivial lower bound on complexity is known today. Instead, complexity theory has contributed (1) a way of dividing the computational world up into *complexity classes,* and (2) evidence suggesting that these complexity classes are probably distinct. If this evidence can be replaced by mathematical proof, then we will have an abundance of interesting lower bounds.

What is a Complexity Class?

Typically, a complexity class is defined by (1) a model of computation, (2) a resource (or collection of resources), and (3) a function known as the *complexity bound* for each resource.

The models used to define complexity classes fall into two main categories: (1) machine-based models, and (2) circuit-based models. Turing machines (TMs) and random-access machines (RAMs) are the two principal families of machine models; they were described in Chapter 24. We describe circuit-based models later, in Section 27.3. Other kinds of (Turing) machines were also introduced in Chapter 24, including deterministic, nondeterministic, alternating, and oracle machines.

When we wish to model real computations, deterministic machines and circuits are our closest links to reality. Then why consider the other kinds of machines? There are two main reasons.

---

[2]The skeptical practitioner can still argue that these lower bounds hold only for the worst-case behavior of an algorithm, and that these bounds are irrelevant if the worst case arises very rarely in practice. There is a complexity theory of problems that are hard on average (as a counterpoint to the average case analysis of algorithms considered in Chapter 14), but to date only a small number of natural problems have been shown to be hard in this sense, and this theory is beyond the scope of this volume. See **Further Information** at the end of this chapter.

The most potent reason comes from the computational problems whose complexity we are trying to understand. The most notorious examples are the hundreds of natural NP-complete problems [15]. To the extent that we understand anything about the complexity of these problems, it is because of the model of *nondeterministic* Turing machines. Nondeterministic machines do not model physical computation devices, but they do model real computational problems. There are many other examples where a particular model of computation has been introduced in order to capture some well-known computational problem in a complexity class. This phenomenon is discussed at greater length in Chapter 29.

The second reason is related to the first. Our desire to understand real computational problems has forced upon us a repertoire of models of computation and resource bounds. In order to understand the relationships between these models and bounds, we combine and mix them and attempt to discover their relative power. Consider, for example, nondeterminism. By considering the complements of languages accepted by nondeterministic machines, researchers were naturally led to the notion of alternating machines. When alternating machines and deterministic machines were compared, a surprising virtual identity of deterministic space and alternating time emerged. Subsequently, alternation was found to be a useful way to model efficient parallel computation. (See "Alternating Complexity Classes" and "Circuits and Parallel Classes" below.) This phenomenon, whereby models of computation are generalized and modified in order to clarify their relative complexity, has occurred often through the brief history of complexity theory, and has generated some of the most important new insights.

Other underlying principles in complexity theory emerge from the major theorems showing *relationships* between complexity classes. These theorems fall into two broad categories. **Simulation theorems** show that computations in one class can be simulated by computations that meet the defining resource bounds of another class. The containment of nondeterministic logarithmic space (NL) in polynomial time (P), and the equality of the class P with alternating logarithmic space, are simulation theorems. **Separation theorems** show that certain complexity classes are distinct. Complexity theory currently has precious few of these. The main tool used in those separation theorems we have is called **diagonalization.** We illustrate this tool by giving proofs of some separation theorems in this chapter. In the next chapter, however, we show some apparently severe limitations of this tool. This ties in to the general feeling in computer science that *lower bounds are hard to prove.* Our current inability to separate many complexity classes from each other is perhaps the greatest challange posed by complexity theory.

27.2 Time and Space Complexity Classes

We begin by emphasizing the fundamental resources of time and space for deterministic and nondeterministic Turing machines. We concentrate on resource bounds between logarithmic and exponential, because those bounds have proved to be the most useful for understanding problems that arise in practice.

Time complexity and **space complexity** were defined in Chapter 24, Definition 24.4. We repeat Definition 24.5 of that chapter to define the following **fundamental time classes** and **fundamental space classes**, given functions $t(n)$ and $s(n)$:

- $\mathsf{DTIME}[t(n)]$ is the class of languages decided by deterministic Turing machines of time complexity $t(n)$.
- $\mathsf{NTIME}[t(n)]$ is the class of languages decided by nondeterministic Turing machines of time complexity $t(n)$.
- $\mathsf{DSPACE}[s(n)]$ is the class of languages decided by deterministic Turing machines of space complexity $s(n)$.
- $\mathsf{NSPACE}[s(n)]$ is the class of languages decided by nondeterministic Turing machines of space complexity $s(n)$.

We sometimes abbreviate $\mathsf{DTIME}[t(n)]$ to $\mathsf{DTIME}[t]$ (and so on) when t is understood to be a function, and when no reference is made to the input length n.

Canonical Complexity Classes

The following are the **canonical complexity classes:**

- $\mathsf{L} = \mathsf{DSPACE}[\log n]$ (deterministic log space)
- $\mathsf{NL} = \mathsf{NSPACE}[\log n]$ (nondeterministic log space)
- $\mathsf{P} = \mathsf{DTIME}[n^{O(1)}] = \bigcup_{k\geq 1} \mathsf{DTIME}[n^k]$ (polynomial time)
- $\mathsf{NP} = \mathsf{NTIME}[n^{O(1)}] = \bigcup_{k\geq 1} \mathsf{NTIME}[n^k]$ (nondeterministic polynomial time)
- $\mathsf{PSPACE} = \mathsf{DSPACE}[n^{O(1)}] = \bigcup_{k\geq 1} \mathsf{DSPACE}[n^k]$ (polynomial space)
- $\mathsf{E} = \mathsf{DTIME}[2^{O(n)}] = \bigcup_{k\geq 1} \mathsf{DTIME}[k^n]$
- $\mathsf{NE} = \mathsf{NTIME}[2^{O(n)}] = \bigcup_{k\geq 1} \mathsf{NTIME}[k^n]$
- $\mathsf{EXP} = \mathsf{DTIME}[2^{n^{O(1)}}] = \bigcup_{k\geq 1} \mathsf{DTIME}[2^{n^k}]$ (deterministic exponential time)
- $\mathsf{NEXP} = \mathsf{NTIME}[2^{n^{O(1)}}] = \bigcup_{k\geq 1} \mathsf{NTIME}[2^{n^k}]$ (nondeterministic exponential time)
- $\mathsf{EXPSPACE} = \mathsf{DSPACE}[2^{n^{O(1)}}] = \bigcup_{k\geq 1} \mathsf{DSPACE}[2^{n^k}]$ (exponential space)

The space classes PSPACE and EXPSPACE are defined in terms of the DSPACE complexity measure. By Savitch's Theorem (see Theorem 27.3 below) the NSPACE measure with polynomial bounds also yields PSPACE, and with $2^{n^{O(1)}}$ bounds yields EXPSPACE.

Why Focus on These Classes?

The class P contains many familiar problems that can be solved efficiently, such as finding shortest paths in networks, parsing context-free grammars, sorting, matrix multiplication, and linear programming. By definition, in fact, P contains all problems that can be solved by (deterministic) programs of reasonable worst-case time complexity.

But P also contains problems whose best algorithms have time complexity $n^{10^{500}}$. It seems ridiculous to say that such problems are computationally feasible. Nevertheless, there are four important reasons to include these problems:

1. For the main goal of proving lower bounds, it is sensible to have an overly generous notion of the class of feasible problems. That is, if we show that a problem is *not* in P, then we have shown in a very strong way that solution via deterministic algorithms is impractical.
2. The theory of complexity-bounded reducibility (Chapter 28) is predicated on the simple notion that if functions f and g are both easy to compute, then the composition of f and g should also be easy to compute. If we want to allow algorithms of time complexity n^2 to be considered feasible (and certainly many algorithms of this complexity are used daily), then we are immediately led to regard running times n^4, n^8, ... as also being feasible. Put another way, the choice is either to lay down an arbitrary and artificial limit on feasibility (and to forgo the desired property that the composition of easy functions be easy), or to go with the natural and overly generous notion given by P.
3. Polynomial time has served well as the intellectual boundary between feasible and infeasible problems. Empirically, problems of time complexity $n^{10^{500}}$ do not arise in practice, while problems of time complexity $O(n^4)$, and those proved or believed to be $\Omega(2^n)$, occur often. Moreover, once a polynomial-time algorithm for a problem is found, the foot is in the door, and an armada of mathematical and algorithmic techniques can be used to improve the algorithm. Linear programming may be the best known example. The breakthrough $O(n^8)$ time algorithm of Khachiyan [29], for $n \times n$ instances, was impractical in itself, but it prompted an innovation by Karmarkar [27] that produced an algorithm whose running time of about

$O(n^3)$ on all cases competes well commercially with the simplex method, which runs in $O(n^3)$ time in most cases but takes 2^n time in some. Of course, if it should turn out that the Hamiltonian circuit problem (or some other NP-complete problem) has complexity $n^{10^{500}}$, then the theory would need to be overhauled. For the time being, this seems unlikely.

4. We would like our fundamental notions to be independent of arbitrary choices we have made in formalizing our definitions. There is much that is arbitrary and historically accidental in the prevalent choice of the Turing machine as the standard model of computation. This choice does not affect the class P itself, however, because the natural notions of polynomial time for essentially all models of sequential computation that have been devised yield the same class. The random-access and pointer machine models described in Section 4 of Chapter 30 can be simulated by Turing machines with at most a cubic increase in time. Many feel that our "true" experience of running time on real sequential computers falls midway between Turing machines and these more-powerful models, but this only bolsters our conviction that the class P gives the "true" notion of polynomial time.

By analogy to the famous *Church–Turing thesis* (see Chapter 26, Section 26.4), which states that the definition of a (partial) recursive function captures the intuitive notion of a computable process, several authorities have proposed the following

Polynomial-Time Church-Turing Thesis

The class P captures the true notion of those problems that are computable in polynomial time by sequential machines, and is the same for any physically relevant model and minimally reasonable time measure of sequential computation that will ever be devised.

This thesis extends also to parallel models if "time" is replaced by the technologically important notion of parallel *work* (see Chapter 45, on parallel computation).

Another way in which the concept of P is robust is that P is characterized by many concepts from logic and mathematics that do not mention machines or time. Some of these characterizations are surveyed in Chapter 29.

The class NP can also be defined by means other than nondeterministic Turing machines. NP equals the class of problems whose solutions can be *verified* quickly, by deterministic machines in polynomial time. Equivalently, NP comprises those languages whose membership proofs can be checked quickly.

For example, one language in NP is the set of composite numbers, written in binary. A proof that a number z is composite can consist of two factors $z_1 \geq 2$ and $z_2 \geq 2$ whose product $z_1 z_2$ equals z. This proof is quick to check if z_1 and z_2 are given, or guessed. Correspondingly, one can design a nondeterministic Turing machine N that on input z branches to write down "guesses" for z_1 and z_2, and then deterministically multiplies them to test whether $z_1 z_2 = z$. Then $L(N)$, the language accepted by N, equals the set of composite numbers, since there exists an accepting computation path if and only if z really is composite. Note that N does not really solve the problem—it just checks the candidate solution proposed by each branch of the computation.

Another important language in NP is the set of satisfiable Boolean formulas, called SAT. A Boolean formula ϕ is satisfiable if there exists a way of assigning `true` or `false` to each variable such that under this truth assignment, the value of ϕ is `true`. For example, the formula $x \wedge (\overline{x} \vee y)$ is satisfiable, but $x \wedge \overline{y} \wedge (\overline{x} \vee y)$ is not satisfiable. A nondeterministic Turing machine N, after checking the syntax of ϕ and counting the number n of variables, can nondeterministically write down an n-bit 0-1 string a on its tape, and then deterministically (and easily) evaluate ϕ for the truth assignment denoted by a. The computation path corresponding to each individual a accepts if and only if $\phi(a) =$ `true`, and so N itself accepts ϕ if and only if ϕ is satisfiable; i.e., $L(N) =$ SAT. Again, this checking of given assignments differs significantly from trying to find an accepting assignment.

The above characterization of NP as the set of problems with easily verified solutions is formalized as follows: $A \in \mathsf{NP}$ if and only if there exist a language $A' \in \mathsf{P}$ and a polynomial p such that for every x, $x \in A$ if and only if there exists a y such that $|y| \leq p(|x|)$ and $(x, y) \in A'$. Here, whenever x belongs to A, y is interpreted as a positive solution to the problem represented by x, or equivalently, as a proof that x belongs to A. The difference between P and NP is that between solving and checking, or between finding a proof of a mathematical theorem and testing whether a candidate proof is correct. In essence, NP represents all sets of theorems with proofs that are short (i.e., of polynomial length), while P represents those statements that can proved or refuted quickly from scratch.

The theory of NP-completeness, together with the many known NP-complete problems, is perhaps the best justification for interest in the classes P and NP. All of the other canonical complexity classes listed above have natural and important problems that are complete for them (under various reducibility relations, the subject of the next chapter). Further motivation for studying L, NL, and PSPACE, comes from their relationships to P and NP. Namely, L and NL are the largest space-bounded classes known to be contained in P, and PSPACE is the smallest space-bounded class known to contain NP. (It is worth mentioning here that NP does not stand for "nonpolynomial time"; the class P is a subclass of NP.)

Similarly, EXP is of interest primarily because it is the smallest deterministic time class known to contain NP. The closely-related class E is not known to contain NP; we will see in "Padding Arguments" the main reason for interest in E.

Constructibility

Before we go further, we need to introduce the notion of *constructibility.* Without it, no meaningful theory of complexity is possible.

The most basic theorem that one should expect from complexity theory would say, "If you have more resources, you can do more." Unfortunately, if we aren't careful with our definitions, then this claim is false:

THEOREM 27.1 **(Gap Theorem)** *There is a computable time bound $t(n)$ such that* $\mathsf{DTIME}[t(n)] = \mathsf{DTIME}[2^{2^{t(n)}}]$.

That is, there is an empty gap between time $t(n)$ and time doubly exponentially greater than $t(n)$, in the sense that anything that can be computed in the larger time bound can already be computed in the smaller time bound. That is, even with much more time, you can't compute more. This gap can be made much larger than doubly exponential; for any computable r, there is a computable time bound t such that $\mathsf{DTIME}[t(n)] = \mathsf{DTIME}[r(t(n))]$. Exactly analogous statements hold for the NTIME, DSPACE, and NSPACE measures.

Fortunately, the gap phenomenon cannot happen for time bounds t that anyone would ever be interested in. Indeed, the proof of the Gap Theorem proceeds by showing that one can define a time bound t such that no machine has a running time that is between $t(n)$ and $2^{2^{t(n)}}$. This theorem indicates the need for formulating only those time bounds that actually describe the complexity of some machine.

A function $t(n)$ is **time-constructible** if there exists a deterministic Turing machine that halts after exactly $t(n)$ steps for every input of length n. A function $s(n)$ is **space-constructible** if there exists a deterministic Turing machine that uses exactly $s(n)$ worktape cells for every input of length n. (Most authors consider only functions $t(n) \geq n + 1$ to be time-constructible, and many limit attention to $s(n) \geq \log n$ for space bounds. There do exist sublogarithmic space-constructible functions, but we prefer to avoid the tricky theory of $o(\log n)$ space bounds.)

For example, $t(n) = n+1$ is time-constructible. Furthermore, if $t_1(n)$ and $t_2(n)$ are time-constructible, then so are the functions $t_1 + t_2$, $t_1 t_2$, $t_1^{t_2}$, and c^{t_1} for every integer $c > 1$. Consequently, if $p(n)$ is a polynomial, then $p(n) = \Theta(t(n))$ for some time-constructible polynomial function $t(n)$. Similarly, $s(n) = \log n$ is space-constructible, and if $s_1(n)$ and $s_2(n)$ are space-constructible, then so are the functions

$s_1 + s_2$, $s_1 s_2$, $s_1^{s_2}$, and c^{s_1} for every integer $c > 1$. Many common functions are space-constructible: e.g., $n \log n$, n^3, 2^n, $n!$.

Constructibility helps eliminate an arbitrary choice in the definition of the basic time and space classes. For general time functions t, the classes $\mathsf{DTIME}[t]$ and $\mathsf{NTIME}[t]$ may vary depending on whether machines are required to halt within t steps on all computation paths, or just on those paths that accept. If t is time-constructible and s is space-constructible, however, then $\mathsf{DTIME}[t]$, $\mathsf{NTIME}[t]$, $\mathsf{DSPACE}[s]$, and $\mathsf{NSPACE}[s]$ can be defined without loss of generality in terms of Turing machines that always halt.

As a general rule, any function $t(n) \geq n + 1$ and any function $s(n) \geq \log n$ that one is interested in as a time or space bound, is time- or space-constructible, respectively. As we have seen, little of interest can be proved without restricting attention to constructible functions. This restriction still leaves a rich class of resource bounds.

The Gap Theorem is not the only case where intuitions about complexity are false. Most people also expect that a goal of algorithm design should be to arrive at an optimal algorithm for a given problem. In some cases, however, no algorithm is remotely close to optimal.

THEOREM 27.2 **(Speed-Up Theorem)** *There is a decidable language A such that for every machine M that decides A, with running time $u(n)$, there is another machine M' that decides A much faster: its running time $t(n)$ satisfies $2^{2^{t(n)}} \leq u(n)$ for all but finitely many n.*

This statement, too, holds with any computable function $r(t)$ in place of 2^{2^t}. Put intuitively, the program M' running on an old IBM PC is better than the program M running on the fastest hardware to date. Hence A has no best algorithm, and no well-defined time-complexity function. Unlike the case of the Gap Theorem, the speed-up phenomenon may hold for languages and time bounds of interest. For instance, a problem of time complexity bounded by $t(n) = n^{\log n}$, which is just above polynomial time, may have arbitrary polynomial speed-up—i.e., may have algorithms of time complexity $t(n)^{1/k}$ for all $k > 0$.

One implication of the Speed-Up Theorem is that the complexities of some problems need to be sandwiched between upper and lower bounds. Actually, there is a sense in which every problem has a well defined lower bound on time. For every language A there is a computable function t_0 such that for every time-constructible function t, there is some machine that accepts A within time t if and only if $t = \Omega(t_0)$ [31]. A catch, however, is that t_0 itself may not be time-constructible.

Basic Relationships

Clearly, for all time functions $t(n)$ and space functions $s(n)$, $\mathsf{DTIME}[t(n)] \subseteq \mathsf{NTIME}[t(n)]$ and $\mathsf{DSPACE}[s(n)] \subseteq \mathsf{NSPACE}[s(n)]$, because a deterministic machine is a special case of a nondeterministic machine. Furthermore, $\mathsf{DTIME}[t(n)] \subseteq \mathsf{DSPACE}[t(n)]$ and $\mathsf{NTIME}[t(n)] \subseteq \mathsf{NSPACE}[t(n)]$, because at each step, a k-tape Turing machine can write on at most $k = O(1)$ previously unwritten cells. The next theorem presents additional important relationships between classes.

THEOREM 27.3 *Let $t(n)$ be a* **time-constructible function,** *and let $s(n)$ be a* **space-constructible function,** *$s(n) \geq \log n$.*

(a) $\mathsf{NTIME}[t(n)] \subseteq \mathsf{DTIME}[2^{O(t(n))}]$.

(b) $\mathsf{NSPACE}[s(n)] \subseteq \mathsf{DTIME}[2^{O(s(n))}]$.

(c) $\mathsf{NTIME}[t(n)] \subseteq \mathsf{DSPACE}[t(n)]$.

(d) **(Savitch's Theorem)** $\mathsf{NSPACE}[s(n)] \subseteq \mathsf{DSPACE}[s(n)^2]$.

As a consequence of the first part of this theorem, $\mathsf{NP} \subseteq \mathsf{EXP}$. No better general upper bound on deter-

ministic time is known for languages in NP, however. See Fig. 27.1 for other known inclusion relationships between canonical complexity classes. (Classes AC^0, TC^0, and NC^1 are defined in Section 27.3.)

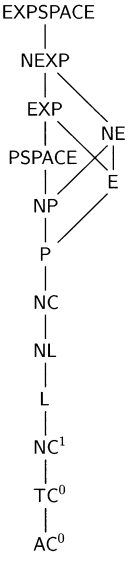


FIGURE 27.1 Inclusion relationships between the canonical complexity classes.

Although we do not know whether allowing nondeterminism strictly increases the class of languages decided in polynomial time, Savitch's Theorem says that for space classes, nondeterminism does not help by more than a polynomial amount.

Complementation

For a language A over an alphabet Σ, define $\overline{A}$ to be the complement of A in the set of words over Σ: $\overline{A} = \Sigma^* - A$. For a class of languages $\mathcal{C}$, define co-$\mathcal{C} = \{\overline{A} : A \in \mathcal{C}\}$. If $\mathcal{C} =$ co-$\mathcal{C}$, then $\mathcal{C}$ is **closed under complementation.**

In particular, co-NP is the class of languages that are complements of languages in NP. For the language SAT of satisfiable Boolean formulas, $\overline{\text{SAT}}$ is the set of unsatisfiable formulas, whose value is `false` for every truth assignment, together with the syntactically incorrect formulas. A closely related language in co-NP is the set of Boolean tautologies, namely, those formulas whose value is `true` for every truth assignment. The question of whether NP equals co-NP comes down to whether every tautology has a short (i.e., polynomial-sized) proof. The only obvious general way to prove a tautology ϕ in m variables is to verify all 2^m rows of the truth table for ϕ, taking exponential time. Most complexity theorists believe that there is no general way to reduce this time to polynomial, hence that NP $\neq$ co-NP.

Questions about complementation bear directly on the P vs. NP question. It is easy to show that P is closed under complementation (see the next theorem). Consequently, if NP $\neq$ co-NP, then P $\neq$ NP.

THEOREM 27.4 **(Complementation Theorems)** *Let t be a time-constructible function, and let s be a space-constructible function, with $s(n) \geq \log n$ for all n. Then*

(a) DTIME$[t]$ *is closed under complementation.*
(b) DSPACE$[s]$ *is closed under complementation.*

(c) **(Immerman–Szelepcsényi Theorem)** $\mathsf{NSPACE}[s]$ *is closed under complementation.*

The Complementation Theorems are used to prove the Hierarchy Theorems in the next section.

Hierarchy Theorems and Diagonalization

Diagonalization is the most useful technique for proving the existence of computationally difficult problems. In this section, we will see examples of two rather different types of arguments, both of which can be called "diagonalization," and we will see how these are used to prove hierarchy theorems in complexity theory.

A hierarchy theorem is a theorem that says "If you have more resources, you can compute more." As we saw in "Constructibility", this theorem is possible only if we restrict attention to constructible time and space bounds. Next, we state hierarchy theorems for deterministic and nondeterministic time and space classes. In the following, $\subset$ denotes *strict* inclusion between complexity classes.

THEOREM 27.5 **(Hierarchy Theorems)** *Let t_1 and t_2 be time-constructible functions, and let s_1 and s_2 be space-constructible functions, with $s_1(n), s_2(n) \geq \log n$ for all n.*

(a) If $t_1(n) \log t_1(n) = o(t_2(n))$, then $\mathsf{DTIME}[t_1] \subset \mathsf{DTIME}[t_2]$.
(b) If $t_1(n+1) = o(t_2(n))$, then $\mathsf{NTIME}[t_1] \subset \mathsf{NTIME}[t_2]$.
(c) If $s_1(n) = o(s_2(n))$, then $\mathsf{DSPACE}[s_1] \subset \mathsf{DSPACE}[s_2]$.
(d) If $s_1(n) = o(s_2(n))$, then $\mathsf{NSPACE}[s_1] \subset \mathsf{NSPACE}[s_2]$.

As a corollary of the Hierarchy Theorem for DTIME,

$$\mathsf{P} \subseteq \mathsf{DTIME}[n^{\log n}] \subset \mathsf{DTIME}[2^n] \subseteq \mathsf{E};$$

hence we have the strict inclusion $\mathsf{P} \subset \mathsf{E}$. Although we do not know whether $\mathsf{P} \subset \mathsf{NP}$, there exists a problem in E that cannot be solved in polynomial time. Other consequences of the Hierarchy Theorems are $\mathsf{NE} \subset \mathsf{NEXP}$ and $\mathsf{NL} \subset \mathsf{PSPACE}$.

In the Hierarchy Theorem for DTIME, the hypothesis on t_1 and t_2 is $t_1(n) \log t_1(n) = o(t_2(n))$, instead of $t_1(n) = o(t_2(n))$, for technical reasons related to the simulation of machines with multiple worktapes by a single universal Turing machine with a fixed number of worktapes. Other computational models, such as random access machines, enjoy tighter time hierarchy theorems.

All proofs of the Hierarchy Theorems use the technique of **diagonalization**. For example, the proof for DTIME constructs a Turing machine M of time complexity t_2 that considers all machines $M_1, M_2, \ldots$ whose time complexity is t_1; for each i, the proof finds a word x_i that is accepted by M if and only if $x_i \notin L(M_i)$, the language decided by M_i. Consequently, $L(M)$, the language decided by M, differs from each $L(M_i)$, hence $L(M) \notin \mathsf{DTIME}[t_1]$. The diagonalization technique resembles the classic method used to prove that the real numbers are uncountable, by constructing a number whose jth digit differs from the jth digit of the jth number on the list. To illustrate the diagonalization technique, we outline proofs of the Hierarchy Theorems for DSPACE and for NTIME. In this subsection, $\langle i, x \rangle$ stands for the string $0^i 1 x$, and *zeroes*(y) stands for the number of 0's that a given string y starts with. Note that *zeroes*$(\langle i, x \rangle) = i$.

PROOF (of the DSPACE Hierarchy Theorem)

We construct a deterministic Turing machine M that decides a language A such that $A \in \mathsf{DSPACE}[s_2] - \mathsf{DSPACE}[s_1]$.

Let U be a deterministic universal Turing machine, as described in Chapter 26, Section 26.2. On input x of length n, machine M performs the following:

1. Lay out $s_2(n)$ cells on a worktape.
2. Let $i = zeroes(x)$.
3. Simulate the universal machine U on input $\langle i, x\rangle$. Accept x if U tries to use more than s_2 worktape cells. (We omit some technical details, such as interleaving multiple worktapes onto the fixed number of worktapes of M, and the way in which the constructibility of s_2 is used to ensure that this process halts.)
4. If U accepts $\langle i, x\rangle$, then reject; if U rejects $\langle i, x\rangle$, then accept.

Clearly, M always halts and uses space $O(s_2(n))$. Let $A = L(M)$.

Suppose $A \in \mathsf{DSPACE}[s_1(n)]$. Then there is some Turing machine M_j accepting A using space at most $s_1(n)$. The universal Turing machine U can easily be given the property that its space needed to simulate a given Turing machine M_j is at most a constant factor higher than the space used by M_j itself. More precisely, there is a constant k depending only on j (in fact, we can take $k = |j|$), such that U, on inputs z of the form $z = \langle j, x\rangle$, uses at most $ks_1(|x|)$ space.

Since $s_1(n) = o(s_2(n))$, there is an n_0 such that $ks_1(n) \leq s_2(n)$ for all $n \geq n_0$. Let x be a string of length greater than n_0 such that the first $j + 1$ symbols of x are $0^j 1$. Note that the universal Turing machine U, on input $\langle j, x\rangle$, simulates M_j on input x and uses space at most $ks_1(n) \leq s_2(n)$. Thus, when we consider the machine M defining A, we see that on input x the simulation does not stop in Step 3, but continues on to Step 4, and thus $x \in A$ if and only if U rejects $\langle j, x\rangle$. Consequently, M_j does not accept A, contrary to our assumption. Thus $A \notin \mathsf{DSPACE}[s_1(n)]$.

A more sophisticated argument is required to prove the Hierarchy Theorem for NTIME. To see why, note that it is necessary to diagonalize against *nondeterministic* machines, and thus it is necessary to use a nondeterministic universal Turing machine as well. In the deterministic case, when we simulated an accepting computation of the universal machine, we would reject, and if we simulated a rejecting computation of the universal machine, we would accept. That is, we would do exactly the opposite of what the universal machine does, in order to "fool" each simulated machine M_i. If the machines under consideration are *nondeterministic,* then M_i can have both an accepting path and a rejecting path on input x, in which case the universal nondeterministic machine would accept input $\langle i, x\rangle$. If we simulate the universal machine on an input and accept upon reaching a rejecting leaf and reject if upon reaching an accepting leaf, then this simulation would still accept (because the simulation that follows the rejecting path now accepts). Thus, we would fail to do the opposite of what M_i does.

The following careful argument guarantees that each machine M_i is fooled on some input. It draws on a result of Book et al. [6] that every language in $\mathsf{NTIME}[t(n)]$ is accepted by a two-tape nondeterministic Turing machine that runs in time $t(n)$.

PROOF (of the NTIME Hierarchy Theorem)

Let $M_1, M_2, \ldots$ be an enumeration of two-tape nondeterministic Turing machines running in time $t_1(n)$. Let f be a rapidly growing function such that time $f(i, n, s)$ is enough time for a deterministic machine to compute the function

$$(i, n, s) \mapsto \begin{cases} 1 & \text{if } M_i \text{ accepts } 1^n \text{ in } \leq s \text{ steps} \\ 0 & \text{otherwise} \end{cases}$$

Letting $f(i, n, s)$ be greater than $2^{2^{i+n+s}}$ is sufficient.

Now divide Σ^* into regions, so that in region $j = \langle i, y\rangle$, we try to "fool" machine M_i. Note that each M_i is considered infinitely often. The regions are defined by functions $start(j)$ and $end(j)$, defined as follows: $start(1) = 1$, $start(j + 1) = end(j) + 1$, where taking $i = zeroes(j)$, we have $end(j) = f(i, start(j), t_2(start(j)))$. The important point is that, on input $1^{end(j)}$, a deterministic machine can, in time $t_2(end(j))$, determine whether M_i accepts $1^{start(j)}$ in at most $t_2(start(j))$ steps.

By picking f appropriately easy to invert, we can guarantee that, on input 1^n, we can in time $t_2(n)$ determine which region j contains n.

Now it is easy to verify that the following routine can be performed in time $t_2(n)$ by a nondeterministic machine. (In the pseudo-code below, U is a "universal" nondeterministic machine with 4 tapes, which is therefore able to simulate one step of machine M_i in $O(i^3)$ steps.)

1. On input 1^n, determine which region j contains n. Let $j = \langle i, y \rangle$.
2. If $n = end(j)$, then accept if and only if M_i does *not* accept $1^{start(j)}$ within $t_2(start(j))$ steps.
3. Otherwise, accept if and only if U accepts $\langle i, 1^{n+1} \rangle$ within $t_2(n)$ steps. (Here, it is important that we are talking about $t_2(n)$ steps of U, which may be only about $t_2(n)/i^3$ steps of M_i.)

Let us call the language accepted by this procedure A. Clearly $A \in \mathsf{NTIME}[t_2(n)]$. We now claim that $A \notin \mathsf{NTIME}[t_1(n)]$.

Assume otherwise, and let M_i be the nondeterministic machine accepting A in time $t_1(n)$. Recall that M_i has only two tapes. Let c be a constant such that $i^3 t_1(n+1) < t_2(n)$ for all $n \geq c$. Let y be a string such that $|y| \geq c$, and consider stage $j = \langle i, y \rangle$. Then for all n such that $start(j) \leq n < end(j)$, we have $1^n \in A$ if and only if $1^{n+1} \in A$. However this contradicts the fact that $1^{start(j)} \in A$ if and only if $1^{end(j)} \notin A$.

Although the diagonalization technique successfully separates some pairs of complexity classes, diagonalization does not seem strong enough to separate P from NP. (See Theorem 28.15 in Chapter 28.)

Padding Arguments

A useful technique for establishing relationships between complexity classes is the **padding argument.** Let A be a language over alphabet Σ, and let # be a symbol not in Σ. Let f be a numeric function. The **f-padded version of** A is the language

$$A' = \left\{ x\#^{f(n)} : x \in A \text{ and } n = |x| \right\}$$

That is, each word of A' is a word in A concatenated with $f(n)$ consecutive # symbols. The padded version A' has the same information content as A, but because each word is longer, the computational complexity of A' is smaller!

The proof of the next theorem illustrates the use of a padding argument.

THEOREM 27.6 *If* $\mathsf{P} = \mathsf{NP}$, *then* $\mathsf{E} = \mathsf{NE}$.

PROOF Since $\mathsf{E} \subseteq \mathsf{NE}$, we prove that $\mathsf{NE} \subseteq \mathsf{E}$.

Let $A \in \mathsf{NE}$ be decided by a nondeterministic Turing machine M in at most $t(n) = k^n$ time for some constant integer k. Let A' be the $t(n)$-padded version of A. From M, we construct a nondeterministic Turing machine M' that decides A' in linear time: M' checks that its input has the correct format, using the time-constructibility of t; then M' runs M on the prefix of the input preceding the first # symbol. Thus, $A' \in \mathsf{NP}$.

If $\mathsf{P} = \mathsf{NP}$, then there is a deterministic Turing machine D' that decides A' in at most $p'(n)$ time for some polynomial p'. From D', we construct a deterministic Turing machine D that decides A, as follows. On input x of length n, since $t(n)$ is time-constructible, machine D constructs $x\#^{t(n)}$, whose length is $n + t(n)$, in $O(t(n))$ time. Then D runs D' on this input word. The time complexity of D is at most $O(t(n)) + p'(n + t(n)) = 2^{O(n)}$. Therefore, $\mathsf{NE} \subseteq \mathsf{E}$.

A similar argument shows that the $\mathsf{E} = \mathsf{NE}$ question is equivalent to the question of whether $\mathsf{NP} - \mathsf{P}$ contains a subset of 1^*, that is, a language over a single-letter alphabet.

Padding arguments sometimes can be used to give tighter hierarchies than can obtained by straightforward diagonalization. For instance, Theorem 27.5 leaves open the question of whether, say, $\mathsf{DTIME}[n^3 \log^{1/2} n] = \mathsf{DTIME}[n^3]$. We can show that these classes are not equal, by using a padding argument. We will need the following lemma, whose proof is similar to that of Theorem 27.6.

LEMMA 27.1 (**Translational Lemma**) Let t_1, t_2, and f be time-constructible functions. If $\mathsf{DTIME}[t_1(n)] = \mathsf{DTIME}[t_2(n)]$, then $\mathsf{DTIME}[t_1(f(n))] = \mathsf{DTIME}[t_2(f(n))]$.

THEOREM 27.7 *For any real number $a > 0$ and natural number $k \geq 1$, $\mathsf{DTIME}[n^k] \subset \mathsf{DTIME}[n^k \log^a n]$.*

PROOF Suppose for contradiction that $\mathsf{DTIME}[n^k] = \mathsf{DTIME}[n^k \log^a n]$. For now let us also suppose that $a > 1/2$. Taking $f(n) = 2^{n/k}$, and using the linear speed-up property, we obtain from the Translational Lemma the identity $\mathsf{DTIME}[2^n n^a] = \mathsf{DTIME}[2^n]$. This does not yet give the desired contradiction to the DTIME Hierarchy Theorem—but it is close. We'll need to use the Translational Lemma twice more.

Assume that $\mathsf{DTIME}[2^n n^a] = \mathsf{DTIME}[2^n]$. Using the Translational Lemma with $f(n) = 2^n$ yields $\mathsf{DTIME}[2^{2^n} 2^{an}] = \mathsf{DTIME}[2^{2^n}]$. Applying the Lemma once again on the classes $\mathsf{DTIME}[2^n n^a] = \mathsf{DTIME}[2^n]$, this time using $f(n) = 2^n + an$, we obtain $\mathsf{DTIME}[2^{2^n} 2^{an} f(n)^a] = \mathsf{DTIME}[2^{2^n} 2^{an}]$. Combining these two equalities yields $\mathsf{DTIME}[2^{2^n} 2^{an} f(n)^a] = \mathsf{DTIME}[2^{2^n}]$. Since $f(n)^a > 2^{an}$, we have that $2^{an} f(n)^a > 2^{2an} = 2^n 2^{bn}$ for some $b > 0$ (since $a > 1/2$). Thus $\mathsf{DTIME}[2^{2^n} 2^n 2^{bn}] = \mathsf{DTIME}[2^{2^n}]$, and this result contradicts the DTIME Hierarchy Theorem, since $2^{2^n} \log 2^{2^n} = o(2^{2^n} 2^n 2^{bn})$.

Finally, for any fixed $a > 0$, not just $a > 1/2$, we need to apply the Translational Lemma several more times.

One consequence of this theorem is that within P, there can be no "complexity gaps" of size $(\log n)^{\Omega(1)}$.

Alternating Complexity Classes

In this section, we define time and space complexity classes for alternating Turing machines, and we show how these classes are related to the classes introduced already. Alternating Turing machines and their configurations are defined in Chapter 24.

The possible computations of an alternating Turing machine M on an input word x can be represented by a tree T_x in which the root is the initial configuration, and the children of a nonterminal node C are the configurations reachable from C by one step of M. For a word x in $L(M)$, define an **accepting subtree** S of T_x as follows:

- S is finite.
- The root of S is the initial configuration with input word x.
- If S has an existential configuration C, then S has exactly one child of C in T_x; if S has a universal configuration C, then S has all children of C in T_x.
- Every leaf is a configuration whose state is the accepting state q_A.

Observe that each node in S is an accepting configuration.

We consider only alternating Turing machines that always halt. For $x \in L(M)$, define the time taken by M to be the height of the shortest accepting tree for x, and the space to be the maximum number

of nonblank worktape cells among configurations in the accepting tree that minimizes this number. For $x \notin L(M)$, define the time to be the height of T_x, and the space to be the maximum number of non-blank worktape cells among configurations in T_x.

Let $t(n)$ be a time-constructible function, and let $s(n)$ be a space-constructible function. Define the following complexity classes:

- $\mathsf{ATIME}[t(n)]$ is the class of languages decided by alternating Turing machines of time complexity $O(t(n))$.
- $\mathsf{ASPACE}[s(n)]$ is the class of languages decided by alternating Turing machines of space complexity $O(s(n))$.

Because a nondeterministic Turing machine is a special case of an alternating Turing machine, for every $t(n)$ and $s(n)$, $\mathsf{NTIME}(t) \subseteq \mathsf{ATIME}(t)$ and $\mathsf{NSPACE}(s) \subseteq \mathsf{ASPACE}(s)$. The next theorem states further relationships between computational resources used by alternating Turing machines, and resources used by deterministic and nondeterministic Turing machines.

THEOREM 27.8 **(Alternation Theorems)** *Let $t(n)$ be a time-constructible function, and let $s(n)$ be a space-constructible function, $s(n) \geq \log n$.*

(a) $\mathsf{NSPACE}[s(n)] \subseteq \mathsf{ATIME}[s(n)^2]$

(b) $\mathsf{ATIME}[t(n)] \subseteq \mathsf{DSPACE}[t(n)]$

(c) $\mathsf{ASPACE}[s(n)] \subseteq \mathsf{DTIME}[2^{O(s(n))}]$

(d) $\mathsf{DTIME}[t(n)] \subseteq \mathsf{ASPACE}[\log t(n)]$

In other words, space on deterministic and nondeterministic Turing machines is polynomially related to time on alternating Turing machines. Space on alternating Turing machines is exponentially related to time on deterministic Turing machines. The following corollary is immediate.

THEOREM 27.9

(a) $\mathsf{ASPACE}[O(\log n)] = \mathsf{P}$.

(b) $\mathsf{ATIME}[n^{O(1)}] = \mathsf{PSPACE}$.

(c) $\mathsf{ASPACE}[n^{O(1)}] = \mathsf{EXP}$.

Note that Theorem 27.8(a) says, for instance, that NL is contained in $\mathsf{ATIME}(\log^2(n))$. For this to make sense, it is necessary to modify the definition of alternating Turing machines to allow them to read individual bits of the input in constant time, rather than requiring n time units to traverse the entire input tape. This has become the standard definition of alternating Turing machines, because it is useful in establishing relationships between Turing machine complexity and **circuit complexity,** as explained in the upcoming section.

27.3 Circuit Complexity

Up to now, this chapter has been concerned only with complexity classes that were defined in order to understand the nature of sequential computation. Although we called them "machines," the models discussed here and in Chapter 24 are closer in spirit to software, namely to sequential algorithms or to single-processor machine-language programs. Circuits were originally studied to model hardware. The hardware of electronic digital computers is based on digital gates, connected into combinational and

sequential networks. Whereas a software program can branch and even modify itself while running, hardware components on today's typical machines are fixed and cannot reconfigure themselves. Also, circuits capture well the notion of nonbranching, straight-line computation.

Furthermore, circuits provide a good model of parallel computation. Many machine models, complexity measures, and classes for parallel computation have been devised, but the circuit complexity classes defined here coincide with most of them. Chapter 45 in this volume surveys parallel models and their relation to circuits in more detail.

Kinds of Circuits

A *circuit* can be formalized as a directed graph with some number n of sources, called *input nodes* and labeled $x_1, \ldots, x_n$, and one sink, called the *output node.* The edges of the graph are called *wires.* Every noninput node v is called a *gate,* and has an associated *gate function* g_v that takes as many arguments as there are wires coming into v. In this survey we limit attention to Boolean circuits, meaning that each argument is 0 or 1, although arithmetical circuits with numeric arguments and $+$, $*$ (etc.) gates have also been studied in complexity theory. Formally g_v is a function from $\{0, 1\}^r$ to $\{0, 1\}$, where r is the *fan-in* of v. The value of the gate is transmitted along each wire that goes out of v. The *size* of a circuit is the number of nodes in it.

We restrict attention to circuits C in which the graph is acyclic, so that there is no "feedback." Then every Boolean assignment $x \in \{0, 1\}^n$ of values to the input nodes determines a unique value for every gate and wire, and the value of the output gate is the output $C(x)$ of the circuit. The circuit *accepts* x if $C(x) = 1$.

The sequential view of a circuit is obtained by numbering the gates in a manner that respects the edge relation, meaning that for all edges (u, v), g_u has a lower number than g_v. Then the gate functions in that order become a sequence of basic instructions in a straight-line program that computes $C(x)$. The size of the circuit becomes the number of steps in the program. However, this view presumes a single processing unit that evaluates the instructions in sequence, and ignores information that the graphical layout provides. A more powerful view regards the gates as simple processing units that can act in parallel. Every gate whose incoming wires all come from input nodes can act and compute its value at step 1, and every other gate can act and transmit its value at the first step after all gates on its incoming wires have computed their values. The number of steps for this process is the *depth* of the circuit. Depth is a notion of *parallel time complexity.* A circuit with small depth is a fast circuit. The circuit *size* in this view is the amount of hardware needed. Chapter 45 gives much more information on the correspondence between circuits and parallel machines, and gives formal definitions of size and depth.

A *circuit family* $\mathbf{C}$ consists of a sequence of circuits $\{C_1, C_2, \ldots\}$, where each C_n has n input nodes. The language accepted by the family is $L(\mathbf{C}) = \{x : C_{|x|} \text{ accepts } x\}$. (Circuit families computing functions $f : \{0, 1\}^* \to \{0, 1\}^*$ are defined in Chapter 45.)

The *size complexity* of the family is the function $z(n)$ giving the number of nodes in C_n. The *depth complexity* is the function $d(n)$ giving the depth of C_n.

Another aspect of circuits that must be specified in order to define complexity classes is the underlying technology. By technology we mean the types of gates that are used as components in the circuits. Three types of technology are considered in this chapter:

(1) Bounded fan-in gates, usually taken to be the "standard basis" of binary $\wedge$ (AND), binary $\vee$ (OR) and unary $\neg$ (NOT) gates. A notable alternative is to use NAND gates.

(2) Unbounded fan-in $\wedge$ and $\vee$ gates (together with unary $\neg$ gates).

(3) Threshold gates. For our purposes, it suffices to consider the simplest kind of threshold gate, called the MAJORITY gate, which also uses the Boolean domain. A MAJORITY gate outputs 1 if and only if at least $r/2$ of its r incoming wires have value 1. These gates can

simulate unbounded fan-in $\wedge$ and $\vee$ with the help of "dummy wires." Threshold circuits also have unary $\neg$ gates.

The difference between (1) and (2) corresponds to general technological issues about high-bandwidth connections, whether they are feasible and how powerful they are. Circuits of type (1) can be converted to equivalent circuits that also have bounded fan-*out,* with only a constant-factor penalty in size and depth. Thus the difference also raises issues about one-to-many broadcast and all-to-one reception.

Threshold gates model the technology of *neural networks,* which were formalized in the 1940s. The kind of threshold gate studied most often in neural networks uses Boolean arguments and values, with '1' for "firing" and '0' for "off." It has numerical *weights* $w_1, \ldots, w_r$ for each of the r incoming wires and a *threshold* t. Letting $a_1, \ldots, a_r$ stand for the incoming 0-1 values, the gate outputs 1 if $\Sigma_{i=1}^{r} a_i w_i \geq t$, 0 otherwise. Thus the MAJORITY gate is the special case with $w_1 = \cdots = w_r = 1$ and $t = r/2$. A depth-2 (sub-)circuit of MAJORITY gates can simulate this general threshold gate.

Uniformity and Circuit Classes

One tricky aspect of circuit complexity is the fact that many functions that are *not computable* have trivial circuit complexity! For instance, let K be a noncomputable set of numbers, such as the indices of halting Turing machines, and let A be the language $\{x : |x| \in K\}$. For each n, if $n \in K$, then define C_n by attaching a $\neg$ gate to input x_1 and an OR gate whose two wires come from the $\neg$ gate and x_1 itself. If $n \notin K$, then define C_n similarly but with an AND gate in place of the OR. The circuit family $[C_n]$ so defined accepts A and has size and depth 2. The rub, however, is that there is no algorithm to tell *which* choice for C_n to define for each n. A related anomaly is that there are uncountably many circuit families. Indeed, every language is accepted by some circuit family $[C_n]$ with size complexity $2^{O(n)}$ and depth complexity 3 (unbounded fan-in) or $O(n)$ (bounded fan-in). Consequently, for general circuits, size complexity is at most exponential, and depth complexity is at most linear.

The notion of **uniform circuit complexity** avoids both anomalies. A circuit family $[C_n]$ is *uniform* if there is an easy algorithm Q that, given n, outputs an encoding of C_n. Either the adjacency-matrix or the edge-list representation of the graphs of the circuits C_n, together with the gate type of each node, may serve for our purposes as the *standard encoding scheme* for circuit families. If Q runs in polynomial time, then the circuit family is P-*uniform,* and so on.

P-uniformity is natural because it defines those families of circuits that are feasible to construct. However, we most often use circuits to model computation in *subclasses* of P. Allowing powerful computation to be incorporated into the step of *building* $C_{|x|}$ may overshadow the computation done by the circuit $C_{|x|}$ itself. The following much more stringent condition has proved to be most useful for characterizing these subclasses, and also works well for circuit classes at the level of polynomial time.

DEFINITION 27.1 A circuit family $[C_n]$ is DLOGTIME-*uniform* if there is a Turing machine M that can answer questions of the forms "Is there a path of edges from node u to node v in C_n?" and "What gate type does node u have?" in $O(\log n)$ time.

This uniformity condition is sufficient to build an encoding of C_n in sequential time roughly proportional to the size of C_n, and even much faster in parallel time. We will not try to define DLOGTIME as a complexity class, but note that since the inputs u, v to M can be presented by strings of length $O(\log n)$, the computation by M takes linear time in the (scaled down) input length. This definition presupposes that the size complexity $z(n)$ of the family is polynomial, which will be our chief interest here. The definition can be modified for $z(n)$ more than polynomial by changing the time limit on M to $O(\log z(n))$. Many central results originally proved using L-uniformity extend without change to DLOGTIME-uniformity, as explained later in this section. Unless otherwise stated, "uniform" means DLOGTIME-uniform throughout this and the next two chapters. We define the following circuit complexity classes:

DEFINITION 27.2 Given complexity functions $z(n)$ and $d(n)$,

- SIZE$[z(n)]$ is the class of all languages accepted by DLOGTIME-uniform bounded fan-in circuit families whose size complexity is at most $z(n)$;
- DEPTH$[d(n)]$ is the class of all languages accepted by DLOGTIME-uniform bounded fan-in circuit families whose depth complexity is at most $d(n)$;
- SIZE,DEPTH$[z(n), d(n)]$ is the class of all languages accepted by DLOGTIME-uniform bounded fan-in circuit families whose size complexity is at most $z(n)$ and whose depth complexity is at most $d(n)$.

Non-uniform circuit classes can be approached by an alternative view introduced by Karp and Lipton [28], by counting the number of bits of information needed to set up the preprocessing. For integer-valued functions t, a, define DTIME$[t(n)]$/ADV$[a(n)]$ to be the class of languages accepted by Turing machines M as follows: for all n there is a word y_n of length at most $a(n)$ such that for all x of length n, on input (x, y_n), M accepts if and only if $x \in L$, and M halts within $t(n)$ steps. Here y_n is regarded as "advice" on how to accept strings of length n. The class DTIME$[n^{O(1)}]$/ADV$[n^{O(1)}]$ is called P/poly. Karp and Lipton observed that P/poly is equal to the class of languages accepted by polynomial-sized circuits. Indeed, P/poly is now the standard name for this class.

Circuits and Sequential Classes

The importance of P/poly and uniformity is shown by the following basic theorem. We give the proof since it is used often in the next chapter.

THEOREM 27.10 *Every language in* P *is accepted by a family of polynomial-sized circuits that is* DLOGTIME*-uniform. Conversely, every language with* P*-uniform polynomial-sized circuits belongs to* P.

PROOF Let $A \in$ P. By Example 24.10 in Chapter 24, A is accepted by a Turing machine M with just one tape and tape head that runs in polynomial time $p(n)$. Let δ be the transition function of M, whereby for all states q of M and characters c in the worktape alphabet Γ of M, $\delta(q, c)$ specifies the character written to the current cell, the movement of the head, and the next state of M. We build a circuit of "δ-gates" that simulates M on inputs x of a given length n as follows, and then show how to simulate δ-gates by Boolean gates.

Lay out a $p(n) \times p(n)$ array of cells. Each cell (i, j) $(0 \leq i, j \leq p(n))$ is intended to hold the character on tape cell j after step i of the computation of M, and if the tape head of M is in that cell, also the state of M after step i. Cells $(0, 0)$ through $(0, n-1)$ are the input nodes of C_n, while cells $(0, n)$ through $(0, p(n))$ can be treated as "dummy wires" whose value is the blank B in the alphabet Γ. The key idea is that the value in cell (i, j) for $i \geq 1$ depends only on the values in cells $(i-1, j-1)$, $(i-1, j)$, and $(i-1, j+1)$. Cell $(i-1, j-1)$ is relevant in case its value includes the component for the tape head being there, and the head moves right at step i; cell $(i-1, j+1)$ similarly for a left move.

When the boundary cases $j = 0$ or $j = p(n)$ are handled properly, each cell value is computed by the same finite function of the three cells above, and this function defines a "δ-gate" for each cell. (See Fig. 27.2.) Finally, we may suppose that M is coded to signal acceptance by moving its tape head to the left end and staying in a special state q_a. Thus node $(i, 0)$ becomes the output gate of the circuit, and the accepting output values are those with q_a in the state component. Since in $p(n)$ steps M can visit at most $p(n)$ tape cells, the array is large enough to hold all the computations of M on inputs of length n.

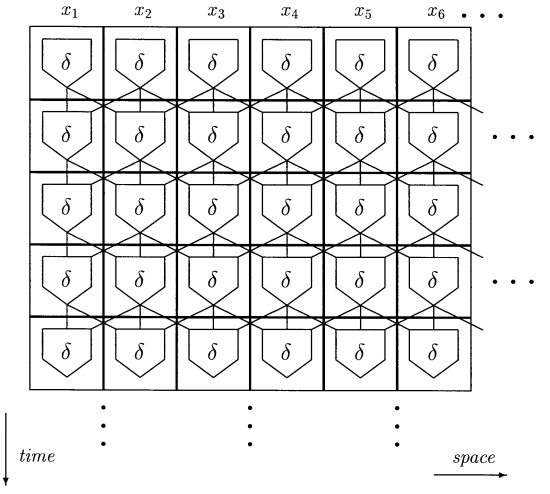


FIGURE 27.2 Conversion from Turing machine to Boolean circuits.

Since each argument and value of a δ-gate comes from a finite domain, we may take an (arbitrary) binary encoding of the domain, and replace all δ-gates by identical fixed-size subcircuits of Boolean gates that compute δ under the encoding. If the alphabet Σ over which A is defined is $\{0, 1\}$ then no recoding need be done at the inputs; otherwise, we similarly adopt a binary encoding of Σ. The Boolean circuits C_n thus obtained accept A. They also are DLOGTIME-uniform, intuitively by the very regular structure of the identical δ-gates.

Conversely, given a P-uniform family $\mathbf{C}$, a Turing machine can accept $L(\mathbf{C})$ in polynomial time given any input x by first constructing $C_{|x|}$ in polynomial time, and then evaluating $C_{|x|}(x)$.

A caching strategy that works for Turing machines with any fixed number of tapes yields the following improvement:

THEOREM 27.11 *If $t(n)$ is a time-constructible function, then* $\mathsf{DTIME}(t) \subseteq \mathsf{SIZE}(t \log t)$.

Connections between space complexity and circuit depth are shown by the next result.

THEOREM 27.12

(a) If $d(n) \geq \log n$, then $\mathsf{DEPTH}[d(n)] \subseteq \mathsf{DSPACE}[d(n)]$.
(b) If $s(n)$ is a space-constructible function and $s(n) \geq \log n$, then $\mathsf{NSPACE}[s(n)] \subseteq \mathsf{DEPTH}[s(n)^2]$.

Circuits and Parallel Classes

Since the 1970s, research on circuit complexity has focused on problems that can be solved quickly in parallel, with feasible amounts of hardware—circuit families of polynomial size and depth as small as possible. Note, however, that the meaning of the phrase "as small as possible" depends on the technology used. With unbounded fan-in gates, depth $O(1)$ is sufficient to carry out interesting computation, whereas with fan-in two gates, depth less than $\log n$ is impossible if the value at the output gate depends on all of the input bits. In any technology, however, a circuit with depth nearly logarithmic is considered to be very fast. This observation motivates the following definitions. Let $\log^k n$ stand for $(\log n)^k$.

DEFINITION 27.3
For all $k \geq 0$,

(a) NC^k denotes the class of languages accepted by DLOGTIME-uniform bounded fan-in circuit families of polynomial size and $O(\log^k n)$ depth. In other words, $\mathsf{NC}^k = \mathsf{SIZE,DEPTH}[n^{O(1)}, O(\log^k n)]$. NC denotes $\cup_{k\geq 0}\mathsf{NC}^k$.

(b) AC^k denotes the class of languages accepted by DLOGTIME-uniform families of circuits of unbounded fan-in $\wedge$, $\vee$, and $\neg$ gates, again with polynomial size and $O(\log^k n)$ depth.

(c) TC^k denotes the class of languages accepted by DLOGTIME-uniform families of circuits of MAJORITY and $\neg$ gates, again with polynomial size and $O(\log^k n)$ depth.

The case $k = 0$ in these definitions gives constant-depth circuit families. A function f is said to belong to one of these classes if the language $A_f = \{\langle x, i, b\rangle : 1 \leq i \leq |f(x)| \text{ and bit } i \text{ of } f(x) \text{ is } b\}$ belongs to the class. NC^0 is not studied as a language class in general, since the output gate can depend on only a constant number of input bits, but NC^0 is interesting as a function class.

Some notes on the nomenclature are in order. Nicholas Pippenger was one of the first to study polynomial-size, polylog-depth circuits in the late 1970s, and NC was dubbed "Nick's Class." There is no connotation of nondeterminism in NC. The "A" in AC^k connotes both alternating circuits and alternating Turing machines for reasons described below. The "T" in TC^k stands for the presence of threshold gates.

The following theorem expresses the relationships at each level of the hierarchies defined by these classes.

THEOREM 27.13 *For each $k \geq 0$,*

$$\mathsf{NC}^k \subseteq \mathsf{AC}^k \subseteq \mathsf{TC}^k \subseteq \mathsf{NC}^{k+1}.$$

PROOF The first inclusion is immediate (for each k), and the second conclusion follows from the observation noted above that MAJORITY gates can simulate unbounded fan-in AND and OR gates. The interesting case is $\mathsf{TC}^k \subseteq \mathsf{NC}^{k+1}$. For this, it suffices to show how to simulate a single MAJORITY gate with a fan-in two circuit of logarithmic depth. To simulate $\mathsf{MAJORITY}(w_1, \ldots, w_r)$, we add up the one-bit numbers $w_1, \ldots, w_r$ and test whether the sum is at least $r/2$. We may suppose for simplicity that the fan-in r is a power of 2, $r = 2^m$. The circuit has m distinguished nodes that represent the sum written as an m-bit binary number. Then the sum is at least $r/2 = 2^{m-1}$ if and only if the node representing the most significant bit of the sum has value 1.

To compute the sum efficiently, we use a standard "carry-save" technique: There is a simple $O(1)$ depth fan-in two circuit that takes as input three b-bit binary numbers a_1, a_2, a_3 and produces as output two $(b+1)$-bit numbers b_1, b_2 such that $a_1 + a_2 + a_3 = b_1 + b_2$. Thus in one phase, the original sum of r bits is reduced to taking the sum of $\frac{2}{3}r$ numbers, and after $O(\log r)$ additional phases, the problem is reduced to taking the sum of two $\log r$-bit numbers, and this sum can be produced by a full carry-lookahead adder circuit of $O(\log r)$ depth. Finally, since the circuits have polynomial size, r is polynomial in n, and so $O(\log r) = O(\log n)$.

Thus in particular, $\cup_k \mathsf{AC}^k = \cup_k \mathsf{TC}^k = \mathsf{NC}$. The only proper inclusion known, besides the trivial case $\mathsf{NC}^0 \subset \mathsf{AC}^0$, is $\mathsf{AC}^0 \subset \mathsf{TC}^0$, discussed below. For all we know at this time, TC^0 may be equal not only to NC, but even to NP!

Several relationships between complexity classes based on circuits and classes based on Turing machines are known:

THEOREM 27.14 $\mathsf{NC}^1 \subseteq \mathsf{L} \subseteq \mathsf{NL} \subseteq \mathsf{AC}^1$.

In fact, the connection with Turing machines is much closer than this theorem suggests. Using *alternating* Turing machines, we define the following complexity classes:

- $\mathsf{ASPACE,TIME}[s(n), t(n)]$ is the class of languages recognized by alternating Turing machines that use space at most $s(n)$ and also run in time at most $t(n)$.
- $\mathsf{ASPACE,ALTS}[s(n), a(n)]$ is the class of languages recognized by alternating Turing machines that use space at most $s(n)$ and make at most $a(n)$ alternations between existential and universal states.
- $\mathsf{ATIME,ALTS}[s(n), a(n)]$ is the class of languages recognized by alternating Turing machines that run in time $t(n)$ and make at most $a(n)$ alternations between existential and universal states.

THEOREM 27.15

(a) *For all* $k \geq 1$, $\mathsf{NC}^k = \mathsf{ASPACE,TIME}[O(\log n), O(\log^k n)]$.
(b) *For all* $k \geq 1$, $\mathsf{AC}^k = \mathsf{ASPACE,ALTS}[O(\log n), O(\log^k n)]$.
(c) $\mathsf{NC}^1 = \mathsf{ATIME}[O(\log n)]$.
(d) $\mathsf{AC}^0 = \mathsf{ATIME,ALTS}[O(\log n), O(1)]$.

For AC^1 and the higher circuit classes, changing the uniformity condition to L-uniformity does not change the class of languages. However, it is not known whether L-uniform NC^1 differs from NC^1, or L-uniform AC^0 from AC^0. Thus the natural extension (c,d) of the results in (a,b) is another advantage of $\mathsf{DLOGTIME}$-uniformity. Insofar as the containment of NC^1 in L is believed to be proper by many researchers, the definition of L-uniform NC^1 may allow more computing power to the "preprocessing stage" than to the circuits themselves. Avoiding this anomaly is a reason to adopt $\mathsf{DLOGTIME}$-uniformity.

As discussed in Chapter 45, many other models of parallel computation can be used to define NC. This robustness of NC supports the belief that NC is not merely an artifact of some arbitrary choices made in formulating the definitions, but instead captures a fundamental aspect of parallel computation. The criticism has been made that NC is overly generous in allowing polynomial size. Again, the justification in complexity theory is that the ultimate goal is to prove lower bounds, and a lower bound proved against a generous upper-bound notion is impervious to this criticism.

Why Focus on These Circuit Classes?

The class AC^0 is particularly important for the following reasons:

- It captures the complexity of important basic operations such as integer addition and subtraction.
- It corresponds closely to first-order logic, as described in Section 29.4.
- Most important, it is one of the few complexity classes for which lower bounds are actually known, instead of merely being conjectured.

It is known that AC^0 circuits, even non-uniform ones, cannot recognize the language PARITY of strings that have an odd number of 1's. Consequently, constant depth unbounded fan-in AND/OR/NOT circuits for PARITY must have super-polynomial size. However, PARITY does have constant-depth polynomial-size threshold circuits; indeed, it belongs to TC^0.

Note that this also implies that AC^0 is somehow "finer" than the notion of constant space, because the class of regular languages, which includes PARITY, can be decided in constant space. There has been

much progress on proving lower bounds for classes of constant-depth circuits. Still, the fact that TC^0 is not known to differ from NP is a wide gulf in our knowledge. Separating NC from P, or L from P, or L from NP would imply separating TC^0 from NP.

TC^0 is important because it captures the complexity of important basic operations such as integer multiplication and sorting. Further, integer division is known to be in P-uniform TC^0, and many suspect that DLOGTIME-uniformity would also be sufficient. Also, TC^0 is a good complexity-theoretic counterpart to popular models of neural networks.

NC^1 is important because it captures the complexity of the basic operation of evaluating a Boolean formula on a given assignment. The problem of whether NC^1 equals TC^0 thus captures the question of whether basic calculations in logic are harder than basic operations in arithmetic, or harder than basic neural processes. Several other characterizations of NC^1 besides the one given for $\mathsf{ATIME}[O(\log n)]$ are known. NC^1 equals the class of languages definable by polynomial-size Boolean *formulas* (as opposed to polynomial-sized *circuits;* a formula is equivalent to a circuit of fan-out 1). Also, NC^1 equals the class of languages recognized by bounded-width *branching programs* [3]. Finally, NC^1 captures the circuit complexity of regular expressions.

27.4 Research Issues and Summary

The complexity class is the fundamental notion of complexity theory. What makes a complexity class useful to the practitioner is the close relationship between complexity classes and real computational problems. The strongest such relationship comes from the concept of completeness, which is a chief subject of the next chapter. Even in the absence of lower bounds separating complexity classes, the apparent fundamental difference between models such as deterministic and nondeterministic Turing machines, for example, provides insight into the nature of problem solving on computers.

The initial goal when trying to solve a computational problem is to find an efficient polynomial-time algorithm. If this attempt fails, then one could attempt to prove that no efficient algorithm exists, but to date nobody has succeeded doing this for any problem in PSPACE. With the notion of a complexity class to guide us, however, we can attempt to discover the complexity class that exactly captures our current problem. A main theme of the next chapter is the surprising fact that most natural computational problems are *complete* for one of the canonical complexity classes. When viewed in the abstract setting provided by the model that defines the complexity class, the aspects of a problem that make an efficient algorithm difficult to achieve are easier to identify. Often this perspective leads to a redefinition of the problem in a way that is more amenable to solution.

Figure 27.1 shows the known inclusion relationships between canonical classes. Perhaps even more significant is what is currently not known. Although AC^0 differs from TC^0, TC^0 (let alone P!) is not known to differ from NP, or NP from EXP, or EXP from EXPSPACE. The only other proper inclusions known are (immediate consequences of) $\mathsf{L} \neq \mathsf{PSPACE} \neq \mathsf{EXPSPACE}$, $\mathsf{P} \neq \mathsf{E} \neq \mathsf{EXP}$, and $\mathsf{NP} \neq \mathsf{NE} \neq \mathsf{NEXP}$—and these follow simply from the hierarchy theorems proved in this chapter.

We have given two examples of diagonalization arguments. Diagonalization is still the main tool for showing the existence of hard-to-compute problems inside a complexity class. Unfortunately, the languages constructed by diagonalization arguments rarely correspond to computational problems that arise in practice. In some cases, however, one can show that there is an efficient reduction from a difficult problem (shown to exist by diagonalization) to a more natural problem—with the consequence that the natural problem is also difficult to solve. Thus diagonalization inside a complexity class (the topic of this chapter) can work hand in hand with reducibility (the topic of the next chapter) to produce intractability results for natural computational problems.

27.5 Defining Terms

Canonical complexity classes: The classes defined by logarithmic, polynomial, and exponential bounds on time and space, for deterministic and nondeterministic machines. These are the most central to the field, and classify most of the important computational problems.

Circuit: A network of input, output, and logic gates, contrasted with a **Turing machine** in that its hardware is static and fixed.

Circuit complexity: The study of the size, depth, and other attributes of circuits that decide specified languages or compute specified functions.

Diagonalization: A proof technique for showing that a given language does not belong to a given complexity class, used in many **separation theorems**.

Padding argument: A method for transferring results about one complexity bound to another complexity bound, by padding extra dummy characters onto the inputs of the machines involved.

Polynomial-time Church–Turing Thesis: An analogue of the classical **Church–Turing Thesis**, for which see Chapter 26, stating that the class P captures the true notion of feasible (polynomial time) sequential computation.

Separation theorems: Theorems showing that two complexity classes are distinct. Most known separation theorems have been proved by **diagonalization**.

Simulation theorems: Theorems showing that one kind of computation can be simulated by another kind within stated complexity bounds. Most known containment or equality relationships between complexity classes have been proved this way.

Space-constructible function: A function $s(n)$ that gives the actual space used by some Turing machine on all inputs of length n, for all n.

Time-constructible function: A function $t(n)$ that is the actual running time of some Turing machine on all inputs of length n, for all n.

Uniform circuit family: A sequence of circuits, one for each input length n, that can be efficiently generated by a Turing machine.

Uniform circuit complexity: The study of complexity classes defined by uniform circuit families.

References

[1] Ajtai, M., Σ_1^1 formulae on finite structures. *Annals of Pure and Applied Logic,* 24, 1–48, 1983.

[2] Barrington, D.M., Immerman, N., and Straubing, H., On uniformity within NC^1. *J. Comp. Sys. Sci.,* 41, 274–306, 1990.

[3] Barrington, D.M., Bounded-width polynomial-size branching programs recognize exactly those languages in NC^1. *J. Comp. Sys. Sci.,* 38, 150–164, 1989.

[4] Berthiaume, A., Quantum computation. In *Complexity Theory Retrospective II,* L. Hemaspaandra and A. Selman, Eds., 23–51. Springer-Verlag, 1997.

[5] Blum, M., A machine-independent theory of the complexity of recursive functions. *J. Assn. Comp. Mach.,* 14, 322–336, 1967.

[6] Book, R., Greibach, S., and Wegbreit, B., Time- and tape-bounded turing acceptors and afls. *J. Comp. Sys. Sci.,* 4, 606–621, 1970.

[7] Book, R., Comparing complexity classes. *J. Comp. Sys. Sci.,* 9, 213–229, 1974.

[8] Boppana, R. and Sipser, M., The complexity of finite functions. In *Handbook of Theoretical Computer Science,* J. Van Leeuwen, Ed., Vol. A, 757–804. Elsevier and MIT Press, 1990.

[9] Borodin, A., Computational complexity and the existence of complexity gaps. *J. Assn. Comp. Mach.,* 19, 158–174, 1972.

[10] Borodin, A., On relating time and space to size and depth. *SIAM J. Comput.,* 6, 733–744, 1977.
[11] Chandra, A., Kozen, D., and Stockmeyer, L., Alternation. *J. Assn. Comp. Mach.,* 28, 114–133, 1981.
[12] Chandra, A., Stockmeyer, L., and Vishkin, U., Constant-depth reducibility. *SIAM J. Comput.,* 13, 423–439, 1984.
[13] Cook, S., A taxonomy of problems with fast parallel algorithms. *Inform. and Control,* 64, 2–22, 1985.
[14] Furst, M., Saxe, J., and Sipser, M., Parity, circuits, and the polynomial-time hierarchy. *Math. Sys. Thy.,* 17, 13–27, 1984.
[15] Garey, M. and Johnson, D.S., *Computers and Intractability: A Guide to the Theory of NP-Completeness,* Freeman, 1988. First edition was 1979.
[16] Gurevich, Y., Average case completeness. *J. Comp. Sys. Sci.,* 42, 346–398, 1991.
[17] Hartmanis, J. and Stearns, R., On the computational complexity of algorithms. *Transactions of the AMS,* 117, 285–306, 1965.
[18] Håstad, J., Almost optimal lower bounds for small-depth circuits. In *Randomness and Computation,* S. Micali, Ed., Vol. 5, *Advances in Computing Research,* 143–170. JAI Press, Greenwich, CT, 1989.
[19] Hofmeister, T., A note on the simulation of exponential threshold weights. In *Proc. 2nd International Computing and Combinatorics Conference,* Vol. 1090, *Lect. Notes in Comp. Sci.,* 136–141. Springer-Verlag, 1996.
[20] Hoover, H., Klawe, M., and Pippenger, N., Bounding fan-out in logical networks. *J. Assn. Comp. Mach.,* 31, 13–18, 1984.
[21] Hopcroft, J. and Ullman, J., *Introduction to Automata Theory, Languages, and Computation,* Addison–Wesley, Reading, MA, 1979.
[22] Ibarra, O., A note concerning nondeterministic tape complexities. *J. Assn. Comp. Mach.,* 19, 608–612, 1972.
[23] Immerman, N. and Landau, S., The complexity of iterated multiplication. *Inform. and Control,* 116, 103–116, 1995.
[24] Immerman, N., Nondeterministic space is closed under complementation. *SIAM J. Comput.,* 17, 935–938, 1988.
[25] Impagliazzo, R., A personal view of average-case complexity. In *Proc. 10th Annual IEEE Conference on Structure in Complexity Theory,* 134–147, 1995.
[26] Johnson, D.S., A catalog of complexity classes. In *Handbook of Theoretical Computer Science,* J. Van Leeuwen, Ed., Vol. A, 67–161. Elsevier and MIT Press, 1990.
[27] Karmarkar, N., A new polynomial-time algorithm for linear programming. *Combinatorica,* 4, 373–395, 1984.
[28] Karp, R. and Lipton, R., Turing machines that take advice. *L'Enseignement Mathématique,* 28, 191–210, 1982.
[29] Khachiyan, L., A polynomial algorithm in linear programming. *Soviet Mathematics Doklady,* 20(1), 191–194, 1979. English translation.
[30] Kurtz, S., Mahaney, S., Royer, J., and Simon, J., Biological computing. In *Complexity Theory Retrospective II,* L. Hemaspaandra and A. Selman, Eds., 179–195. Springer-Verlag, 1997.
[31] Levin, L.A., Computational complexity of functions. *Theor. Comp. Sci.,* 157(2), 267–271, 1996.
[32] Lewis II, P., Stearns, R., and Hartmanis, J., Memory bounds for recognition of context-free and context-sensitive languages. In *Proceedings, Sixth Annual IEEE Symposium on Switching Circuit Theory and Logical Design,* 191–202, 1965.
[33] Papadimitriou, C., *Computational Complexity,* Addison-Wesley, Reading, MA, 1994.
[34] Parberry, I., *Circuit Complexity and Neural Networks,* M.I.T. Press, Cambridge, MA, 1994.
[35] Pippenger, N. and Fischer, M., Relations among complexity measures. *J. Assn. Comp. Mach.,* 26, 361–381, 1979.

[36] Reif, J. and Tate, S., On threshold circuits and polynomial computation. *SIAM J. Comput.,* 21, 896–908, 1992.
[37] Ruby, S. and Fischer, P., Translational methods and computational complexity. In *Proceedings, Sixth Annual IEEE Symposium on Switching Circuit Theory and Logical Design,* 173–178, 1965.
[38] Ruzzo, W., On uniform circuit complexity. *J. Comp. Sys. Sci.,* 22, 365–383, 1981.
[39] Savitch, W., Relationship between nondeterministic and deterministic tape complexities. *J. Comp. Sys. Sci.,* 4, 177–192, 1970.
[40] Seiferas, J., Fischer, M., and Meyer, A., Separating nondeterministic time complexity classes. *J. Assn. Comp. Mach.,* 25, 146–167, 1978.
[41] Sipser, M., Borel sets and circuit complexity. In *Proc. 15th Annual ACM Symposium on the Theory of Computing,* 61–69, 1983.
[42] Stockmeyer, L., The complexity of decision problems in automata theory and logic. Technical Report MAC-TR-133, Project MAC, M.I.T., Cambridge, MA, 1974.
[43] Stockmeyer, L., Classifying the computational complexity of problems. *J. Symb. Logic,* 52, 1–43, 1987.
[44] Szelepcsényi, R., The method of forced enumeration for nondeterministic automata. *Acta Informatica,* 26, 279–284, 1988.
[45] Trakhtenbrot, B., Turing computations with logarithmic delay. *Algebra i Logika,* 3, 33–48, 1964.
[46] van Emde Boas, P., Machine models and simulations. In *Handbook of Theoretical Computer Science,* J. Van Leeuwen, Ed., Vol. A, 1–66, Elsevier and MIT Press, 1990.
[47] von zur Gathen, J., Efficient exponentiation in finite fields. In *Proc. 32nd Annual IEEE Symposium on Foundations of Computer Science,* 384–391, 1991.
[48] Wang, J., Average-case computational complexity theory. In *Complexity Theory Retrospective II,* L. Hemaspaandra and A. Selman, Eds., 295–328. Springer-Verlag, 1997.
[49] Zak, S., A turing machine time hierarchy. *Theor. Comp. Sci.,* 26, 327–333, 1983.

Further Information

Primary sources for the results presented in this chapter are: Theorem 27.1 [9, 45]; Theorem 27.2 [5]; Theorems 27.3 and 27.4 [17, 24, 32, 39, 44]; Theorem 27.5 [17, 22, 40]; Theorem 27.6 [7]; Lemma 27.1 [37]; Theorems 27.8 and 27.9 [11]; Theorem 27.10 [39]; Theorem 27.11 [35]; Theorem 27.12 [10]; Theorem 27.15 [2, 12, 38, 41]. Theorems 27.13 and 27.14 are a combination of results in the last four papers; see also the influential survey by Cook [13]. Our proof of Theorem 27.5(b) follows [49].

For Section 27.3, a comparison of arithmetical circuits with Boolean circuits may be found in [47], the result that bounded fan-in circuits can be given bounded fan-out is due to [20], and the sharpest simulation of general weighted threshold gates by MAJORITY gates is due to [19]. The theorem in "Why Focus on These Circuit Classes?" that PARITY is not in AC^0 is due to [1, 14], and the strongest lower bounds known on the size of constant-depth circuits for PARITY are those in [18]. The results mentioned for TC^0 may be found in [2, 23, 36].

The texts [21] and [33] present many of these results in greater technical detail. Three chapters of the *Handbook of Theoretical Computer Science,* respectively [26], [46], and [8], describe more complexity classes, compare complexity measures for more machine models, and present more information on circuit complexity. Relationships between circuits and parallel and neural models are covered very accessibly in [34]. Average-case complexity is discussed by [16, 25, 48]. See also Chapter 29 and the notes at the end of that chapter for further sources.

Two important new research areas that challenge our arguments about feasible computation are *quantum computing* and *DNA computing.* Two new survey articles on these fields are [4] and [30].

28

Reducibility and Completeness[1]

Eric Allender
Rutgers University

Michael C. Loui
University of Illinois at Urbana-Champaign

Kenneth W. Regan
State University of New York at Buffalo

28.1 Introduction

There is little doubt that the notion of *reducibility* is the most useful tool that complexity theory has delivered to the rest of the computer science community.

For most computational problems that arise in real-world applications, such as the Traveling Salesperson Problem, we still know little about their deterministic time or space complexity. We cannot now tell whether classes such as **P** and **NP** are distinct. And yet, even without such hard knowledge, it has been useful in practice to take some new problem A whose complexity needs to be analyzed, and announce that A has roughly the same complexity as Traveling Salesperson, by exhibiting efficient ways of reducing each problem to the other. Thus we can say a lot about problems being equivalent in complexity to each other, even if we cannot pinpoint what that complexity is.

[1] Eric Allender — Supported by the National Science Foundation under Grant CCR-9509603. Portions of this work were performed while a visiting scholar at the Institute of Mathematical Sciences, Madras, India.
Michael C. Loui — Supported by the National Science Foundation under Grant CCR-9315696.
Kenneth W. Regan — Supported by the National Science Foundation under Grant CCR-9409104.

One reason this has succeeded is that, when one partitions the many thousands of real-world computational problems into equivalence classes according to the reducibility relation, there is a surprisingly small number of classes of this partition. Thus, the complexity of almost any problem arising in practice can be classified by showing that it is equivalent to one of a short list of representative problems. It was not originally expected that this would be the case.

Even more amazingly, these "representative problems" correspond in a very natural way to abstract models of computation—that is, they correspond to complexity classes. These classes were defined in Chapter 27 using a small set of abstract machine concepts: Turing machines, nondeterminism, alternation, time, space. With this and a few simple functions that define time and space bounds, we are able to characterize the complexity of the overwhelming majority of natural computational problems—most of which bear no topical resemblance to any question about Turing machines. This tool has been much more successful than we had any right to expect it would be.

All this leads us to believe that it is no mere accident that problems easily lend themselves to being placed in one class or another. That is, we are disposed to think that these classes really *are* distinct, that the classification is *real,* and that the mathematics developed to deal with them really does describe some important aspect of nature. Nondeterministic Turing machines, with their ability to magically soar through immense search spaces, *seem* to be much more powerful than our mundane deterministic machines, and this reinforces our belief. However, until P vs. NP and similar longstanding questions of complexity theory are completely resolved, our best method of understanding the complexity of real-world problems is to use the classification provided by reducibility, and to trust in a few plausible conjectures.

In this chapter, we discuss reducibility. We will learn about different types of reducibility, and the related notion of *completeness.* It is especially useful to understand NP-completeness. We define NP-completeness precisely, and give examples of NP-complete problems. We show how to prove that a problem is NP-complete, and give some help for coping with NP-completeness. After that, we describe problems that are complete for other complexity classes, under the most efficient reducibility relations. Finally, we cover two other important topics in complexity theory that are motivated by reducibility: relativized computation and the study of **sparse languages.**

28.2 Reducibility Relations

In mathematics, as in everyday life, a typical way to solve a new problem is to reduce it to a previously solved problem. Frequently, an instance of the new problem is expressed completely in terms of an instance of the prior problem, and the solution is then interpreted in the terms of the new problem. This kind of **reduction** is called **many-one reducibility**, and is defined below.

A different way to solve the new problem is to use a subroutine that solves the prior problem. For example, we can solve an optimization problem whose solution is feasible and maximizes the value of an objective function g by repeatedly calling a subroutine that solves the corresponding decision problem of whether there exists a feasible solution x whose value $g(x)$ satisfies $g(x) \geq k$. This kind of reduction is called **Turing reducibility,** and is also defined below.

Let A_1 and A_2 be languages. A_1 is many-one reducible to A_2, written $A_1 \leq_m A_2$, if there exists a total recursive function f such that for all x, $x \in A_1$ if and only if $f(x) \in A_2$. The function f is called the **transformation function.** A_1 is Turing reducible to A_2, written $A_1 \leq_T A_2$, if A_1 can be decided by a deterministic oracle Turing machine M using A_2 as its oracle, i.e., $A_1 = L(M^{A_2})$. (Recursive functions are defined in Chapter 26, and oracle machines are defined in Chapter 24. The oracle for A_2 models a hypothetical efficient subroutine for A_2.)

If f or M above consumes too much time or space, the reductions they compute are not helpful. To study complexity classes defined by bounds on time and space resources, it is natural to consider resource-bounded reducibilities. Let A_1 and A_2 be languages.

- A_1 is **Karp reducible** to A_2, written $A_1 \leq_m^p A_2$, if A_1 is many-one reducible to A_2 via a transformation function that is computable deterministically in polynomial time.
- A_1 is **Cook reducible** to A_2, written $A_1 \leq_T^p A_2$, if A_1 is Turing reducible to A_2 via a deterministic **oracle Turing machine** of polynomial time complexity.

The term "polynomial-time reducibility" usually refers to Karp reducibility. If $A_1 \leq_m^p A_2$ and $A_2 \leq_m^p A_1$, then A_1 and A_2 are **equivalent** under Karp reducibility. Equivalence under Cook reducibility is defined similarly.

Karp and Cook reductions are useful for finding relationships between languages of high complexity, but they are not useful at all for distinguishing between problems in P, because all problems in P are equivalent under Karp (and hence Cook) reductions. (Here and later we ignore the special cases $A_1 = \emptyset$ and $A_1 = \Sigma^*$, and consider them to reduce to any language.) To investigate the many interesting complexity classes inside P, we will want to define more restrictive reducibilities, and this we do beginning in Section 28.5. For the time being, however, we focus on Cook and Karp reducibility.

The key property of Cook and Karp reductions is that they preserve polynomial-time feasibility. Suppose $A_1 \leq_m^p A_2$ via a transformation f. If M_2 decides A_2, and M_f computes f, then to decide whether an input word x is in A_1, we may use M_f to compute $f(x)$, and then run M_2 on input $f(x)$. If the time complexities of M_2 and M_f are bounded by polynomials t_2 and t_f, respectively, then on inputs x of length $n = |x|$, the time taken by this method of deciding A_1 is at most $t_f(n) + t_2(t_f(n))$, which is also a polynomial in n. In summary, if A_2 is feasible, and there is an efficient reduction from A_1 to A_2, then A_1 is feasible. Although this is a simple observation, this fact is important enough to state as a theorem. First, though, we need the concept of "closure."

A class of languages $\mathcal{C}$ is **closed under a reducibility** $\leq_r$ if for all languages A_1 and A_2, whenever $A_1 \leq_r A_2$ and $A_2 \in \mathcal{C}$, necessarily $A_1 \in \mathcal{C}$.

THEOREM 28.1 P *is closed under both Cook and Karp reducibility.*

Note that this is an instance of an idea that motivated our identification of P with the class of "feasible" problems in Chapter 27, namely that the composition of two feasible functions should be feasible. Similar considerations give us the following theorem.

THEOREM 28.2 *Karp reducibility and Cook reducibility are transitive; i.e.,*

1. *If* $A_1 \leq_m^p A_2$ *and* $A_2 \leq_m^p A_3$, *then* $A_1 \leq_m^p A_3$.
2. *If* $A_1 \leq_T^p A_2$ *and* $A_2 \leq_T^p A_3$, *then* $A_1 \leq_T^p A_3$.

We shall see the importance of closure under a reducibility in conjunction with the concept of completeness, which we define in the next section.

28.3 Complete Languages and Cook's Theorem

Let $\mathcal{C}$ be a class of languages that represent computational problems. A language A_0 is $\mathcal{C}$-**hard** under a reducibility $\leq_r$ if for all A in $\mathcal{C}$, $A \leq_r A_0$. A language A_0 is $\mathcal{C}$-**complete** under $\leq_r$ if A_0 is $\mathcal{C}$-hard, and $A_0 \in \mathcal{C}$. Informally, if A_0 is $\mathcal{C}$-hard, then A_0 represents a problem that is at least as difficult to solve as any problem in $\mathcal{C}$. If A_0 is $\mathcal{C}$-complete, then in a sense, A_0 is one of the most difficult problems in $\mathcal{C}$.

There is another way to view completeness. Completeness provides us with tight lower bounds on the complexity of problems. If a language A is complete for complexity class $\mathcal{C}$, then we have a lower bound on its complexity. Namely, A is as hard as the most difficult problem in $\mathcal{C}$, assuming that the complexity

of the reduction itself is small enough not to matter. The lower bound is tight because A is in $\mathcal{C}$; that is, the upper bound matches the lower bound.

In the case $\mathcal{C} = \mathsf{NP}$, the reducibility $\leq_r$ is usually taken to be Karp reducibility unless otherwise stated. Thus we say:

- A language A_0 is NP-**hard** if A_0 is NP-hard under Karp reducibility.
- A_0 is NP-**complete** if A_0 is NP-complete under Karp reducibility.

However, many sources take the term "NP-hard" to refer to Cook reducibility.

Many important languages are now known to be NP-complete. Before we get to them, let us discuss some implications of the statement "A_0 is NP-complete," and also some things this statement doesn't mean.

The first implication is that *if* there exists a deterministic Turing machine that decides A_0 in polynomial time—that is, if $A_0 \in \mathsf{P}$—then because P is closed under Karp reducibility (Theorem 28.1 in Section 28.2), it would follow that $\mathsf{NP} \subseteq \mathsf{P}$, hence $\mathsf{P} = \mathsf{NP}$. In essence, the question of whether P is the same as NP comes down to the question of whether any particular NP-complete language is in P. Put another way, *all* of the NP-complete languages stand or fall together: if one is in P, then all are in P; if one is not, then all are not. Another implication, which follows by a similar closure argument applied to co-NP, is that if $A_0 \in$ co-NP then $\mathsf{NP} =$ co-NP. It is also believed unlikely that $\mathsf{NP} =$ co-NP, as was noted in connection with whether all tautologies have short proofs in Chapter 27.

A common misconception is that the above property of NP-complete languages is actually their definition, namely: if $A \in \mathsf{NP}$, and $A \in \mathsf{P}$ implies $\mathsf{P} = \mathsf{NP}$, then A is NP-complete. This "definition" is wrong. A theorem due to Ladner [19] shows that $\mathsf{P} \neq \mathsf{NP}$ if and only if there exists a language A' in $\mathsf{NP} - \mathsf{P}$ such that A' is not NP-complete. Thus, if $\mathsf{P} \neq \mathsf{NP}$, then A' is a counterexample to the "definition."

Another common misconception arises from a misunderstanding of the statement "If A_0 is NP-complete, then A_0 is one of the most difficult problems in NP." This statement is true on one level: if there is any problem at all in NP that is not in P, then the NP-complete language A_0 is one such problem. However, note that there are NP-complete problems in $\mathsf{NTIME}[n]$—and these problems are, in some sense, much *simpler* than many problems in $\mathsf{NTIME}[n^{10^{500}}]$. We discuss the difficulty of NP-complete problems in more detail after studying several examples.

We now prove **Cook's Theorem,** which established the first important NP-complete problem. Recall the definition of SAT, the language of satisfiable Boolean formulas, from Section 27.2 of Chapter 27. In this and later Karp-reduction proofs, we highlight the *construction* of the transformation f, check that the *complexity* of f is polynomial, and verify the *correctness* of the reduction.

THEOREM 28.3 **(Cook's Theorem)** *SAT is* NP*-complete.*

PROOF Let $A \in \mathsf{NP}$. Without loss of generality we may assume that $A \subseteq \{0, 1\}^*$. There is a polynomial q and a polynomial-time computable relation R such that for all x,

$$x \in A \iff (\exists y : |y| = q(|x|))\ R(x, y)\ .$$

By the construction of the proof of Theorem 27.10 in Chapter 27, there is a polynomial p such that for all n, we can build in time $p(n)$ a Boolean circuit C_n, using only binary NAND gates, that decides R on inputs of length $n + q(n)$. C_n has n input nodes labeled $x_1, \ldots, x_n$ and $q = q(n)$ more input nodes labeled $y_1, \ldots, y_q$. C_n has at most $p(n)$ *wires,* which we label by $w_1, \ldots, w_m$, where $m \leq p(n)$ and w_m is a special wire leading out of the output gate.

Construction. We first write a Boolean formula ϕ_n in the x, y, *and* w variables to express that every gate in C_n functions correctly and C_n outputs 1. For every NAND gate in C_n with incoming wires u, v, and

for each outgoing wire w of the gate, we add to ϕ_n the following conjunction of three *clauses*

$$(u \vee w) \wedge (v \vee w) \wedge (\bar{u} \vee \bar{v} \vee \bar{w}).$$

These clauses are satisfied by those assignments to u, v, w such that $w = \neg(u \wedge v)$. Intuitively, they assert that the given NAND gate functions correctly.

Thus ϕ_n has three clauses for every wire w except those wires leading from the inputs, each of which carries the label of the corresponding input variable. Finally for the output wire, ϕ_n has the singleton clause (w_m). So ϕ_n has at most $3p(n) + 1$ clauses in all.

Now given x, we form the desired formula $f(x) = \phi_x$ by building ϕ_n, where $n = |x|$, and simply appending n singleton clauses that force the corresponding assignment to the $x_1, \ldots, x_n$ variables. (For example, if $x = 1001$, append $x_1 \wedge \bar{x}_2 \wedge \bar{x}_3 \wedge x_4$.)

Complexity. C_n is built up in roughly $O(p(n))$ time. Building ϕ_n from C_n, and appending the singleton clauses for x, takes a similar polynomial amount of time.

Correctness. Formally, we need to show that for all x, $x \in A \iff f(x) \in \text{SAT}$. By construction, for all x, $x \in A$ if and only if there exists an assignment to the y variables and to the w variables that satisfies ϕ_x. Hence the reduction is correct.

A glance at the proof shows that ϕ_x is always a Boolean formula in conjunctive normal form (CNF) with clauses of one, two, or three literals each. By introducing some new "dummy" variables, we can arrange that each clause has exactly three literals. Thus we have actually shown that the following *restricted form* of the satisfiability problem is NP-complete:

3-SATISFIABILITY (3SAT)
Instance: A Boolean expression ϕ in conjunctive normal form with three literals per clause.
Question: Is ϕ satisfiable?

One concrete implication of Cook's Theorem is that if deciding SAT is easy (i.e., in polynomial time), then factoring integers is likewise easy, because the decision version of factoring belongs to NP. (See Chapter 39.) This is a surprising connection between two ostensibly unrelated problems.

The main impact, however, is that once one language has been proved complete for a class such as NP, others can be proved complete by constructing transformations. If A_0 is NP-complete, then to prove that another language A_1 is NP-complete, it suffices to prove that $A_1 \in \text{NP}$, and to construct a polynomial-time transformation that establishes $A_0 \leq_m^p A_1$. Since A_0 is NP-complete, for every language A in NP, $A \leq_m^p A_0$, hence by transitivity (Theorem 28.2 in Section 28.2), $A \leq_m^p A_1$.

Hundreds of computational problems in many fields of science and engineering have been proved to be NP-complete, almost always by reduction from a problem that was previously known to be NP-complete. We give some practically-motivated examples of these reductions, and also some advice on how to cope with NP-completeness.

28.4 NP-Complete Problems and Completeness Proofs

This and the next two sections are directed toward practitioners who have a computational problem, don't know how to solve it, and want to know how hard it is—specifically, is it NP-complete, or NP-hard? The following step-by-step procedure will help in answering these questions, and may help in identifying cases of the problem that are tractable even if the problem is NP-hard for general cases. In brief, the steps are:

1. State the problem in general mathematical terms, and formalize the statement.
2. Ascertain whether the problem belongs to NP.
3. If so, try to find it in a compendium of known NP-complete problems.

4. If you cannot find it, try to construct a *reduction* from a related problem that is known to be NP-complete or NP-hard.
5. Try to identify special cases of your problem that are (i) hard, (ii) easy, and/or (iii) the ones you need. Your work in Steps 1–4 may help you here.
6. Even if your cases are NP-hard, they may still be amenable to direct attack by sophisticated methods on high-powered hardware.

These steps are interspersed with a traditional "theorem–proof" presentation and several long examples, but the same sequence is maintained. We emphasize that trying to do the formalization and proofs asked for in these steps may give you useful positive information about your problem.

Step 1. Give a *formal statement* of the problem. State it without using terms that are specific to your own particular discipline. Use common terms from mathematics and data objects in computer science, e.g., graphs, trees, lists, vectors, matrices, alphabets, strings, logical formulas, mathematical equations. For example, a problem in evolutionary biology that a phylogenist would state in terms of "species" and "characters" and "cladograms" can be stated in terms of trees and strings, using an alphabet that represents the taxonomic characters. Standard notions of size, depth, and distance in trees can express the objectives of the problem.

If your problem involves computing a function that produces a lot of output, look for associated yes/no decision problems, because decision problems have been easier to characterize and classify. For instance, if you need to compute matrices of a certain kind, see whether the essence of your problem can be captured by yes/no questions about the matrices, perhaps about individual entries of them. Many optimization problems looking for a solution of a certain minimum cost or maximum value can be turned into decision problems by including a target cost/value "k" as an input parameter, and phrasing the question of whether a solution exists of cost less than (or value greater than) the target k. Several problems given in the examples below have this form.

It may also help to simplify, even over-simplify, your problem by removing or ignoring some particular elements of it. Doing so may make it easier to ascertain what general category of decision problem yours is in or closest to. In the process, you may learn useful information about the problem that tells you what the effects of those specific elements are—we say some more about this in "Significance of NP-Completeness".

Step 2. When you have an adequate formalization, ask first, does your decision problem belong to NP? This is true if and only if candidate solutions that would bring about a "yes" answer can be tested in polynomial time—see the extended discussion in Chapter 27, Section 27.2. If it does belong to NP, that's good news for now! Even if not, you may proceed to determine whether it is NP-hard. The problem may be complete for a class such as PSPACE that contains NP. Examples of such problems are given later in this chapter.

Step 3. See whether your problem is already listed in a compendium of (NP-)complete problems. The book [11] lists several hundred NP-complete problems arranged by category. The following is intended as a small representative sample. The first five (together with 3SAT) receive extended treatment in [11], while the last five receive comparable treatment here. (The language corresponding to each problem is the set of instances whose answers are "yes.")

VERTEX COVER
Instance: A graph G and an integer k.
Question: Does G have a set W of k vertices such that every edge in G is incident on a vertex in W?

CLIQUE
Instance: A graph G and an integer k.
Question: Does G have a set K of k vertices such that every two vertices in K are adjacent in G?

HAMILTONIAN CIRCUIT
Instance: A graph G.
Question: Does G have a circuit that includes every vertex exactly once?

3-DIMENSIONAL MATCHING
Instance: Sets W, X, Y with $|W| = |X| = |Y| = q$ and a subset $S \subseteq W \times X \times Y$.
Question: Is there a subset $S' \subseteq S$ of size q such that no two triples in S' agree in any coordinate?

PARTITION
Instance: A set S of positive integers.
Question: Is there a subset $S' \subseteq S$ such that the sum of the elements of S' equals the sum of the elements of $S - S'$?

INDEPENDENT SET
Instance: A graph G and an integer k.
Question: Does G have a set U of k vertices such that no two vertices in U are adjacent in G?

GRAPH COLORABILITY
Instance: A graph G and an integer k.
Question: Is there an assignment of colors to the vertices of G so that no two adjacent vertices receive the same color, and at most k colors are used overall?

TRAVELING SALESPERSON (TSP)
Instance: A set of m "cities" $C_1, \ldots, C_m$, with a distance $d(i, j)$ between every pair of cities C_i and C_j, and an integer D.
Question: Is there a tour of the cities whose total length is at most D, i.e., a permutation $c_1, \ldots, c_m$ of $\{1, \ldots, m\}$, such that $d(c_1, c_2) + \cdots + d(c_{m-1}, c_m) + d(c_m, c_1) \leq D$?

KNAPSACK
Instance: A set $U = \{u_1, \ldots, u_m\}$ of objects, each with an integer size *size* (u_i) and an integer profit *profit* (u_i), a target size s_0, and a target profit p_0.
Question: Is there a subset $U' \subseteq U$ whose total cost and total profit satisfy

$$\sum_{u_i \in U'} \textit{size}\ (u_i) \leq s_0 \qquad \textit{and} \qquad \sum_{u_i \in U'} \textit{profit}\ (u_i) \geq p_0?$$

The languages of all of these problems are easily seen to belong to NP. For example, to show that TSP is in NP, one can build a nondeterministic Turing machine that simply guesses a tour and checks that the tour's total length is at most D.

Some comments on the last two problems are relevant to Steps 1 and 2 above. TRAVELING SALESPERSON provides a single abstract form for many concrete problems about sequencing a series of test examples so as to minimize the variation between successive items. The KNAPSACK problem models the filling of a knapsack with items of various sizes, with the goal of maximizing the total value (profit) of the items. Many scheduling problems for multiprocessor computers can be expressed in the form of KNAPSACK instances, where the "size" of an item represents the length of time a job takes to run, and the size of the knapsack represents an available block of machine time.

If yours is on the list of NP-complete problems, you may skip Step 4, and the compendium may give you further information for Steps 5 and 6. You may still wish to pursue Step 4 if you need more study of particular transformations to and from your problem.

If your problem is not on the list, it may still be close enough to one or more problems on the list to help with the next step.

Step 4. Construct a reduction from an already-known NP*-complete problem.* Broadly speaking, Karp reductions come in three kinds.

- A *restriction* from your problem to a special case that is already known to be NP-complete.
- A *minor adjustment* of an already-known problem.
- A *combinatorial transformation.*

The first two kinds of reduction are usually quite easy to do, and we give several examples before proceeding to the third kind.

EXAMPLE 28.1:

PARTITION $\leq_m^p$ KNAPSACK, *by restriction:* Given a PARTITION instance with integers s_i, the corresponding instance of KNAPSACK takes *size* (u_i) = *profit* $(u_i) = s_i$ (for all i), and sets the targets s_0 and p_0 both equal to $(\Sigma_i s_i)/2$. The condition in the definition of the KNAPSACK problem of not exceeding s_0 nor being less than p_0 requires that the sum of the selected items meet the target $(\Sigma_i s_i)/2$ exactly, which is possible if and only if the original instance of PARTITION is solvable.

In this way, the PARTITION problem can be regarded as a *restriction* or special case of the KNAPSACK problem. Note that the reduction itself goes *from* the more-special problem *to* the more-general problem, even though one thinks of the more-general problem as the one being restricted. The implication is that if the restricted problem is NP-hard, then the more-general problem is NP-hard as well, not vice-versa.

EXAMPLE 28.2:

HAMILTONIAN CIRCUIT $\leq_m^p$ TSP *by restriction:* Let a graph G be given as an instance of the HAMILTONIAN CIRCUIT problem, and let G have m vertices $v_1, \ldots, v_m$. These vertices become the "cities" of the TSP instance that we build. Now define a distance function d as follows:

$$d(i, j) = \begin{cases} 1 & \text{if } (v_i, v_j) \text{ is an edge in } G \\ m+1 & \text{otherwise .} \end{cases}$$

Set $D = m$. Clearly, d and D can be computed in polynomial time from G. If G has a Hamiltonian circuit, then the length of the tour that corresponds to this circuit is exactly m. Conversely, if there is a tour whose length is at most m, then each step of the tour must have distance 1, not $m+1$. Then each step corresponds to an edge of G, so the corresponding sequence of vertices forms a Hamiltonian circuit in G. Thus the function f defined by $f(G) = (\{d(i, j) : 1 \leq i, j \leq m\}, D)$ is a polynomial-time transformation from HAMILTONIAN CIRCUIT to TSP.[2]

Minor Adjustments. Here we consider cases where two problems look different but are really closely connected. Consider CLIQUE, INDEPENDENT SET, and VERTEX COVER. A graph G has a clique of size k if and only if its complementary graph G' has an independent set of size k. It follows that the function f defined by $f(G, k) = (G', k)$ is a Karp reduction from INDEPENDENT SET to CLIQUE. To forge a link to the VERTEX COVER problem, note that all vertices *not* in a given vertex cover form an independent set, and vice versa. Thus a graph G on n vertices has a vertex cover of size at most k if and only if G has an

[2]Technically we need f to be a function from Σ^* to Σ^*. However, given a string x we can decide in polynomial time whether x encodes a graph G that can be given as an instance of HAMILTONIAN CIRCUIT. If x does not encode a well-formed instance, then define $f(x)$ to be a fixed instance I_0 of TSP for which the answer is "no." Because this sort of thing can generally always be done, we are free to regard the domain of a reduction function f to be the set of "well-formed instances" of the problem we are reducing from. Henceforth we try to ignore such encoding details.

independent set of size at least $n - k$. Hence the function $g(G, k) = (G, n - k)$ is a Karp reduction from INDEPENDENT SET to VERTEX COVER. (Note that the same f and g also provide reductions from CLIQUE to INDEPENDENT SET and from VERTEX COVER to INDEPENDENT SET, respectively. This does not happen for all reductions, and gives a sense in which these three problems are unusually close to each other.)

NP-Completeness by Combinatorial Transformation

The following examples show how the combinatorial mechanism of one problem (here, 3SAT) can be *transformed* by a reduction into the seemingly much different mechanism of another problem.

THEOREM 28.4 INDEPENDENT SET *is* NP*-complete. Hence also* CLIQUE *and* VERTEX COVER *are* NP*-complete.*

PROOF We have remarked already that the languages of these three problems belong to NP, and shown already that INDEPENDENT SET $\leq_m^p$ CLIQUE and INDEPENDENT SET $\leq_m^p$ VERTEX COVER. It suffices to show that 3SAT $\leq_m^p$ INDEPENDENT SET.

Construction. Let the Boolean formula ϕ be a given instance of 3SAT with variables $x_1, \ldots, x_n$ and clauses $C_1, \ldots, C_m$. The graph G_ϕ we build consists of a "ladder" on $2n$ vertices labeled $x_1, \bar{x}_1, \ldots, x_n, \bar{x}_n$, with edges $(x_i, \bar{x}_i)$ for $1 \leq i \leq n$ forming the "rungs," and m "clause components." Here the component for each clause C_j has one vertex for each literal x_i or $\bar{x}_i$ in the clause, and all pairs of vertices within each clause component are joined by an edge. Finally, each clause-component node with a label x_i is connected by a "crossing edge" to the node with the opposite label $\bar{x}_i$ in the ith "rung," and similarly each occurrence of $\bar{x}_i$ in a clause is joined to the rung node x_i. This finishes the construction of G_ϕ. See Fig. 28.1.

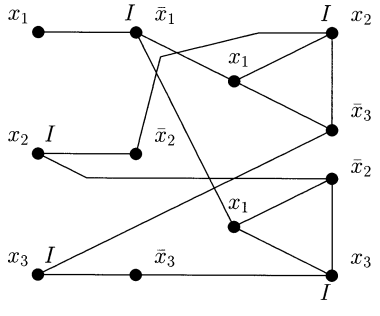


FIGURE 28.1 Construction in the proof of NP-completeness of INDEPENDENT SET for the formula $(x_1 \vee x_2 \vee \bar{x}_3) \wedge (x_1 \vee \bar{x}_2 \vee x_3)$. The independent set of size 5 corresponding to the satisfying assignment $x_1 =$ `false`, $x_2 =$ `true`, and $x_3 =$ `true` is shown by nodes marked I.

Also set $k = n + m$. Then the reduction function f is defined for all arguments ϕ by $f(\phi) = (G_\phi, k)$.

Complexity. It is not hard to see that f is computable in polynomial time given (a straightforward encoding of) ϕ.

Correctness. To complete the proof, we need to argue that ϕ is satisfiable if and only if G_ϕ has an independent set of size $n + m$. To see this, first note that any independent set I of that size must contain exactly one of the two nodes from each "rung," and exactly one node from each clause component—because the edges in the rungs and the clause component prevent any more nodes from being added. And if I selects a node labeled x_i in a clause component, then I must also select x_i in the ith rung. If I

selects $\bar{x}_j$ in a clause component, then I must also select $\bar{x}_j$ in the rung. In this manner I induces a truth assignment in which $x_i = \texttt{true}$ and $x_j = \texttt{false}$, and so on for all variables. This assignment satisfies ϕ, because the node selected from each clause component tells how the corresponding clause is satisfied by the assignment. Going the other way, if ϕ has a satisfying assignment, then that assignment yields an independent set I of size $n + m$ in like manner.

Since the ϕ in this proof is a 3SAT instance, every clause component is a triangle. The idea, however, also works for CNF formulas with any number of variables in a clause, such as the ϕ_x in the proof of Cook's Theorem.

Now we modify the above idea to give another example of an NP-completeness proof by combinatorial transformation.

THEOREM 28.5 GRAPH COLORABILITY *is* NP*-complete*

PROOF *Construction.* Given the 3SAT instance ϕ, we build G_ϕ similarly to the last proof, but with several changes. See Fig. 28.2. On the left, we add a special node labeled "B" and connect it to all $2n$ rung nodes. On the right we add a special node "G" with an edge to B. In any possible 3-coloring of G_ϕ, without loss of generality B will be colored "blue" and the adjacent G will be colored "green." The third color "red" stands for literals made true, whereas green stands for falsity.

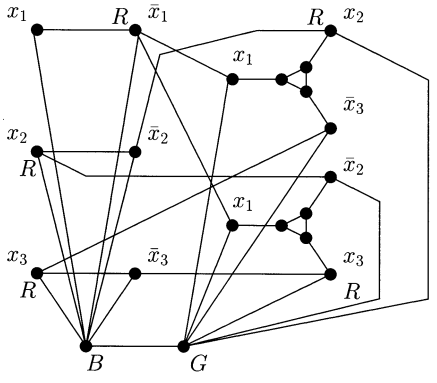


FIGURE 28.2 Construction in the proof of NP-completeness of GRAPH COLORABILITY for the formula $(x_1 \vee x_2 \vee \bar{x}_3) \wedge (x_1 \vee \bar{x}_2 \vee x_3)$. The nodes shown colored R correspond to the satisfying assignment $x_1 = \texttt{false}$, $x_2 = \texttt{true}$, and $x_3 = \texttt{true}$, and these together with G and B essentially force a 3-coloring of the graph, which the reader may complete. Note the resemblance to Fig. 28.1.

Now for each occurrence of a positive literal x_i in a clause, the corresponding clause component has two nodes labeled x_i and x_i' with an edge between them; and similarly an occurrence of a negated literal $\bar{x}_j$ gives nodes $\bar{x}_j$ and $\bar{x}_j'$ with an edge between them. The primed ("inner") nodes in each component are connected by edges into a triangle, but the unprimed ("outer") nodes are not. Each outer node of each clause component is instead connected by an edge to G. Finally, each outer node x_i is connected by a "crossing edge" to the rung node $\bar{x}_i$, and each outer node $\bar{x}_j$ to rung node x_j, exactly as in the INDEPENDENT SET reduction. This finishes the construction of G_ϕ.

Complexity. The function f that given any ϕ outputs G_ϕ, also fixing $k = 3$, is clearly computable in polynomial time.

Correctness. The key idea is that every three-coloring of B, G, and the rung nodes, which corresponds to a truth assignment to the variables of ϕ, can be extended to a 3-coloring of a clause component *if and*

only if at least one of the three crossing edges from the component goes to a green rung node. If all three of these edges go to red nodes, then the links to G force each outer node in the component to be colored blue—but then it is impossible to three-color the inner triangle since blue cannot be used. Conversely, any crossing edge to a green node allows the outer node x_i or $\bar{x_j}$ to be colored red, so that one red and two blues can be used for the outer nodes, and this allows the inner triangle to be colored as well. Hence G_ϕ is 3-colorable if and only if ϕ is satisfiable.

Note that we have also shown that the restricted form of GRAPH COLORABILITY with k fixed to be 3 (i.e., given a graph G, is G 3-colorable?) is NP-complete. Had we stated the problem this way originally, we would now conclude instead that the more-general graph-colorability problem is NP-complete, similarly to the KNAPSACK and TSP examples above.

Many other reductions from 3SAT use the same basic pattern of a truth-assignment selection component for the variables, components for the clauses (whose behavior depends on whether a variable in the clause is satisfied), and links between these components that make the reduction work correctly. For another example of this pattern, a standard proof that HAMILTONIAN CIRCUIT is NP-complete uses subgraphs V_i for each pair $x_i, \bar{x}_i$ and C_j for each clause. There are two possible ways a circuit can enter V_i, and these correspond to the choices of $x_i =$ `true` or $x_i =$ `false` in an assignment. The whole graph is built so that if the circuit enters V_i on the "$x_i =$ `true`" side, then the circuit has the opportunity to visit all nodes in the C_j components for all clauses in which x_i occurs positively, and similarly for occurrences of $\bar{x}_i$ if the circuit enters on the negative side. Hence the circuit can run through every C_j if and only if ϕ is satisfiable. Full details may be found in the text by Papdimitriou [22]. For our last fully worked-out example, we show a somewhat different pattern in which the individual variables as well as the clauses correspond to top-level components of the following problem.

DISJOINT CONNECTING PATHS

Instance: A graph G with two disjoint sets of distinguished vertices $s_1, \ldots, s_k$ and $t_1, \ldots, t_k$, where $k \geq 1$.

Question: Does G contain paths $P_1, \ldots, P_k$, with each P_i going from s_i to t_i, such that no two paths share a vertex?

THEOREM 28.6 DISJOINT CONNECTING PATHS *is* NP*-complete.*

PROOF First, it is easy to see that DISJOINT CONNECTING PATHS belongs to NP: one can design a polynomial-time nondeterministic Turing machine that simply guesses k paths and then deterministically checks that no two of these paths share a vertex. Now let ϕ be a given instance of 3SAT with n variables and m clauses. Take $k = n + m$.

Construction and complexity. The graph G_ϕ we build has distinguished path-origin vertices $s_1, \ldots, s_n$ for the variables and $S_1, \ldots, S_m$ for the clauses of ϕ. G_ϕ also has corresponding sets of path-destination nodes $t_1, \ldots, t_n$ and $T_1, \ldots, T_m$. The other vertices in G_ϕ are nodes u_{ij} for each occurrence of a positive literal x_i in a clause C_j, and nodes v_{ij} for each occurrences of a negated literal $\bar{x}_i$ in C_j. For each i, $1 \leq i \leq n$, G_ϕ is given the edges for a directed path from s_i through all u_{ij} nodes to t_i, and another from s_i through all v_{ij} nodes to t_i. (If there are no occurrences of the positive literal x_i in any clause then the former path is just an edge from s_i right to t_i, and likewise for the latter path if the negated literal $\bar{x}_i$ does not appear in any clause.) Finally, for each j, $1 \leq j \leq m$, G_ϕ has an edge from S_j to every node u_{ij} or v_{ij} for the jth clause, and edges from those nodes to T_j. Clearly these instructions can be carried out to build G_ϕ in polynomial time given ϕ. (See Fig. 28.3.)

Correctness. The first point is that for each i, no path from s_i to t_i can go through both a "u-node" and a "v-node." Setting x_i *true* corresponds to avoiding u-nodes, and setting x_i *false* entails avoiding v-nodes. Thus the choices of such paths for all i represent a truth assignment. The key point is that for each j, one of the three nodes between S_j and T_j will be free for the taking if and only if the corresponding positive or negative literal was made *true* in the assignment, thus satisfying the clause. Hence G_ϕ has the $n + m$ required paths if and only if ϕ is satisfiable.

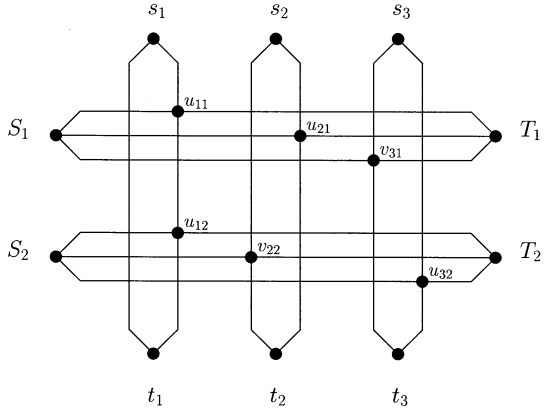


FIGURE 28.3 Construction in the proof of NP-completeness of DISJOINT CONNECTING PATHS for the formula $(x_1 \vee x_2 \vee \bar{x}_3) \wedge (x_1 \vee \bar{x}_2 \vee x_3)$.

Significance of NP-Completeness

Suppose that you have proved that your problem is NP-complete. What does this mean, and how should you approach the problem now?

Exactly what it means is that your problem does not have a polynomial-time algorithm, *unless* every problem in NP has a polynomial-time algorithm; i.e., unless NP $\neq$ P. We have discussed above the reasons for believing that NP $\neq$ P. In practical terms, you can draw one definite conclusion: Don't bother looking for a "magic bullet" to solve the problem. A simple formula or an easily tested deciding condition will not be available; otherwise it probably would have been spotted already during the thousands of person-years that have been spent trying to solve similar problems. For example, the NP-completeness of GRAPH 3-COLORABILITY effectively ended hopes that an efficient mathematical formula for deciding the problem would pop out of research on "chromatic polynomials" associated to graphs. Notice that NP-hardness does not say that one needs to be "extra clever" to find a feasible solving algorithm—it says that one probably does not exist at all.

The proof itself means that the combinatorial mechanism of the problem is rich enough to simulate Boolean logic. The proof, however, may also unlock the door to finding saving graces in Steps 5 and 6.

Step 5. Analyze the instances of your problem that are in the range of the reduction. You may tentatively think of these as "hard cases" of the problem. If these differ markedly from the kinds of instances that you expect to see, then this difference may help you refine the statement and conditions of your problem in ways that may actually define a problem in P after all.

To be sure, avoiding the range of one reduction still leaves wide open the possibility that another reduction will map into your instances of interest. However, it often happens that special cases of NP-complete problems belong to P—and often the boundary between these and the NP-complete cases is

sudden and sharp. For one example, consider SAT. The restricted case of three variables per clause is NP-complete, but the case of two variables per clause belongs to P.

For another example, note that the proof of NP-completeness for DISJOINT CONNECTING PATHS given above uses instances in which $k = n + m$; i.e., in which k depends on the number of variables. The case $k = 2$, where you are given G and s_1, s_2, t_1, t_2 and need to decide whether there are vertex-disjoint paths from s_1 to t_1 and from s_2 to t_2, belongs to P. (The polynomial-time algorithm for this case is nontrivial and was not discovered until 1978 by Garey and Johnson [11].)

However, one must also be careful in one's expectations. Suppose we alter the statement of DISJOINT CONNECTING PATHS by requiring also that no two vertices in two different paths may have an edge between them. Then the case $k = 2$ of the new problem is NP-complete. (Showing this is a nice exercise; the idea is to make one path climb the "variable ladder" and send the other path through all the clause components.)

Strong NP-Completeness for Numerical Problems

An important difference between hard and easy cases applies to certain NP-complete problems that involve *numbers.* For example, above we stated that the PARTITION problem is NP-complete; thus, it is unlikely to be solvable by an efficient algorithm. Clearly, however, we can solve the PARTITION problem by a simple dynamic programming algorithm, as follows.

For an instance of PARTITION, let S be a set of positive integers $\{s_1, \ldots, s_m\}$, and let s^* be the total, $s^* = \Sigma_{i=1}^{m} s_i$. Initialize a linear array B of Boolean values so that $B[0] =$ `true`, and each other entry of B is `false`. For $i = 1$ to m, and for $t = s^*$ down to 0, if $B[t] =$ `true`, then set $B[t + s_i]$ to `true`. After the ith iteration, $B[t]$ is `true` if and only if a subset of $\{s_1, \ldots, s_i\}$ sums to t. The answer to this instance of PARTITION is "yes" if $B[s^*/2]$ is ever set to `true`.

The running time of this algorithm depends critically on the representation of S. If each integer in S is represented in binary, then the running time is exponential in the total length of the representation. If each integer is represented in unary—that is, each s_i is represented by s_i consecutive occurrences of the same symbol—then total length of the representation would be greater than s^*, and the running time would be only a polynomial in the length. Put another way, if the *magnitudes* of the numbers involved are bounded by a polynomial in m, then the above algorithm runs in time bounded by a polynomial in m. Since the length of the encoding of such a low-magnitude instance is $O(m \log m)$, the running time is polynomial in the length of the input. The bottom line is that these cases of the PARTITION problem *are* feasible to solve completely.

A problem is **NP-complete in the strong sense** if there is a fixed polynomial p such that for each instance x of the problem, the value of the largest number encoded in x is at most $p(|x|)$. That is, the integer values are polynomial in the length of the standard representation of the problem. By definition, the 3SAT, VERTEX COVER, CLIQUE, HAMILTONIAN CIRCUIT, and 3-DIMENSIONAL MATCHING problems defined in Section 28.4 are NP-complete in the strong sense, but PARTITION and KNAPSACK are not. The PARTITION and KNAPSACK problems can be solved in polynomial time if the integers in their statements are bounded by a polynomial in n—for instance, if numbers are written in unary rather than binary notation.

The concept of strong NP-completeness reminds us that the representation of information can have a major impact on the computational complexity of a problem.

Coping with NP-Hardness

Step 6. Even if you cannot escape NP-hardness, the cases you need to solve may still respond to sophisticated algorithmic methods, possibly needing high-powered hardware.

There are two broad families of direct attack that have been made on hard problems. *Exact solvers* typically take exponential time in the worst case, but provide feasible runs in certain concrete cases.

Whenever they halt, they output a correct answer—and some exact solvers also output a proof that their answer is correct. *Heuristic algorithms* typically run in polynomial time in all cases, and often aim to be correct only most of the time, or to find approximate solutions (see Sections 6.1 and 6.2 in the next chapter). They are more common. Popular heuristic methods include genetic algorithms, simulated annealing, neural networks, relaxation to linear programming, and stochastic (Markov) process simulation. Experimental systems dedicated to certain NP-complete problems have recently yielded some interesting results—an extensive survey on solvers for TRAVELING SALESPERSON is given by Johnson and McGeogh [15].

There are two ways to attempt to use this research. One is to find a problem close to yours for which people have produced solvers, and try to carry over their methods and heuristics to the specific features of your problem. The other (much more speculative) is to construct a Karp reduction from your problem to their problem, ask to run their program or machine itself on the transformed instance, and then try to map the answer obtained back to a solution of your problem. The hitches are (1) that the currently-known Karp reductions f tend to lose much of the potentially helpful structure of the source instance x when they form $f(x)$, and (2) that approximate solutions for $f(x)$ may map back to terribly suboptimal or even infeasible answers to x. (See, however, the notion of **L-reductions** in Section 29.6 of Chapter 29. All of this indicates that there is much scope for further research on important practical features of relationships between NP-complete problems. See also Chapter 34.

Beyond NP-Hardness

If your problem belongs to NP and you cannot prove that it is NP-hard, it may be an "NP-intermediate" problem; i.e., neither in P nor NP-complete. The theorem of Ladner mentioned in Section 28.3 shows that NP-intermediate problems exist, assuming NP $\neq$ P. However, very few natural problems are currently counted as good candidates for such intermediate status: *factoring, discrete logarithm, graph-isomorphism,* and several problems relating to *lattice bases* form a very representative list. For the first two, see Chapter 39. The vast majority of natural problems in NP have resolved themselves as being either in P or NP-complete. Unless you uncover a specific connection to one of those four intermediate problems, it is more likely offhand that your problem simply needs more work.

The observed tendency of natural problems in NP to "cluster" as either being in P or NP-complete, with little in between, reinforces the arguments made early in this chapter that P is really different from NP.

Finally, if your problem seems not to be in NP, or alternatively if some more stringent notion of feasibility than polynomial time is at issue, then you may desire to know whether your problem is *complete for some other complexity class.* We now turn to this question.

28.5 Complete Problems for NL, P, and PSPACE

We first investigate the log-space analogue of the P vs. NP question, namely whether NL = L. We show that there are natural computational problems that are NL-complete. The question is, under which reducibility? Polynomial-time reducibility is too blunt an instrument here, because NL is contained in P, and so all languages in NL are technically complete for NL under both $\leq_m^p$ and $\leq_T^p$ reductions. We need a reducibility that is fine enough to preserve the distinction between deterministic and nondeterministic log-space that we are attempting to establish and study. The simplest way is to replace the polynomial-time bound in $\leq_m^p$ reductions by a log-space bound.

- A language A_1 is **log-space reducible** to a language A_2, written $A_1 \leq_m^{log} A_2$, if A_1 is many-one reducible to A_2 via a transformation function that is computable by a deterministic Turing machine in $O(\log n)$ space.

There is a log-space analogue of $\leq_T^p$ reducibility, but we do not use it here. Now we show that $\leq_m^{log}$ reductions have the properties we desire:

THEOREM 28.7

(a) (Closure) If $A_1 \leq_m^{log} A_2$ *and* $A_2 \in \mathsf{L}$, *then* $A_1 \in \mathsf{L}$.
(b) (Transitivity) If $A_1 \leq_m^{log} A_2$ *and* $A_2 \leq_m^{log} A_3$, *then* $A_1 \leq_m^{log} A_3$.
(c) (Refinement of $\leq_m^p$ *reductions) If* $A_1 \leq_m^{log} A_2$, *then* $A_1 \leq_m^p A_2$.

The proof of (a) and (b) is somewhat tricky and rests on the fact that if two functions f and g from strings to strings are computable in log space, then so is the function h defined by $h(x) = g(f(x))$. The hitch is that a log space Turing machine M_f computing f can output the characters of $f(x)$ serially but does not have space to store them. This becomes a problem whenever the machine M_g computing g, whose input is the output from M_f, requests the ith character of $f(x)$, where i may be less than the index j of the previous request. The solution is that since only space and not time is constrained, we may *restart* the computation of $M_f(x)$ from scratch upon the request, and let M_g count the characters that M_f outputs serially until it sees the ith one. Such a counter, and similar ones tracking the movements of M_f's actual input head and M_g's "virtual" input head, can be maintained in $O(\log n)$ space. Thus we need no physical output tape for M_f or input tape for M_g, and we obtain a tandem machine that computes $g(f(x))$ in log space. Part (c) is immediate by the function-class counterpart of the inclusion $\mathsf{L} \subseteq \mathsf{P}$.

The definition of "NL-complete" is an instance of the general definition of completeness at the beginning of Section 28.3: A language A_1 is NL-complete ("under $\leq_m^{log}$ reductions" is assumed) if $A_1 \in \mathsf{NL}$ and for every language $A_2 \in \mathsf{NL}$, $A_2 \leq_m^{log} A_1$. One defines "P-complete" in a similar manner—again with $\leq_m^{log}$ reductions assumed. Together with the observation that whenever $A_1, A_2 \in \mathsf{L}$ we have $A_1 \leq_m^{log} A_2$ (ignoring technicalities for A_1 or A_2 equal to $\emptyset$ or Σ^*), we obtain a similar state of affairs to what is known about NP-completeness and the P vs. NP question:

THEOREM 28.8 *Let A be* NL*-complete. Then the following statements are equivalent:*

- $\mathsf{NL} = \mathsf{L}$.
- $A \in \mathsf{L}$.
- *Some* NL*-complete language belongs to* L.
- *All* NL*-complete languages belong to* L.
- *All languages in* L *are* NL*-complete.*

Substitute "P" for "NL" and the same equivalence holds. Note that here we are applying completeness to a class, namely P itself, whose definition does not involve nondeterminism.

Cook's Theorem provides two significant inferences: evidence of intractability, and a connection between computation and Boolean logic. NL-completeness is not to any comparable degree a notion of intractability, but does provide a fundamental link between computations and *graphs,* via the following important problem.

GRAPH ACCESSIBILITY PROBLEM (GAP)
Instance: A directed graph G, and two nodes s and t of G.
Question: Does G have a directed path from node s to node t?

Other names are the *s-t connectivity problem* and the *reachability problem.*

The link involves the concept of an **instantaneous description** (ID) of a Turing machine M. Let us suppose for simplicity that M has just two tapes: one read-only input tape that holds the input x, and one

work tape with alphabet $\{0, 1, B\}$, where B is the blank character. Let us also suppose that M never writes a B. Then any step of a computation of M on the fixed input x is describable by giving

- The current state q of M,
- The contents y of the work tape,
- The position i of the input tape head, and
- The position j of the work tape head.

Then the 4-tuple (q, y, i, j) is called an **ID** of M on input x. The restriction on writing B allows us to identify y with a string in $\{0, 1\}^*$. Without loss of generality, we always have $1 \leq i \leq n+1$, where $n = |x|$, and if $s(n)$ is a space bound on M, also $1 \leq j \leq s(n)$. An ID is also called a **configuration.**

Now define G_x to be the graph whose nodes are all possible IDs of M on input x, and whose directed edges comprise all pairs (I, J) such that M, if set up in configuration I, has a transition that takes it to configuration J in one step. If M is deterministic, then every node in G_x has at most one outgoing arc. Nondeterministic TMs, however, give rise to directed graphs G_x of out-degree more than one. Note that G_x does depend on x, since the step(s) taken from an ID (q, y, i, j) may depend on bit x_i of x.

THEOREM 28.9 GAP *is* NL*-complete.*

PROOF GAP belongs to NL because guessing successive edges in a path from 1 to R (when one exists) needs only $O(\log n)$ space to store the label of the current node, and to locate where on the input tape the adjacency information for the current node is stored. To show NL-hardness, let $A \in \mathsf{NL}$. Then A is accepted by a nondeterministic $O(\log n)$ space bounded Turing machine M. We prove that $A \leq_m^{log} GAP$.

Construction: It is easy to modify M to have the properties supposed in the above discussion of IDs and still run in $O(\log n)$ space. We may also code M so that any accepting computation has a final phase that blanks out all used work tape cells, leaves the input head in cell $n+1$, and halts in a special accepting state q_a. This ensures that every accepting computation (if any) ends in the unique ID $I_t = (q_a, \lambda, n+1, 1)$.

Now given any x, define G_x as above. Let node s be the unique starting ID $I_s = (q_0, \lambda, 1, 1)$, and let node t be I_t. Note that the size of G_x is polynomial—if M runs in $k \log n$ space, then the size is $O(n^{k+2})$.

Complexity: We show that the transformation f that takes a string x as input and produces the list of edges in G_x as output can be computed by a machine M_f that uses $O(\log n)$ space. For each node $I = (q, y, i, j)$ in turn, M_f reads the ith symbol of x and then produces an edge (I, J) for each J such that M can move from I to J when reading that symbol. The only memory space that M_f needs is the space to step through each I in turn, and count up to the ith input position, and produce each J. $O(\log n)$ space is sufficient for all of this.

Correctness: By the construction, paths in G_x correspond to valid sequences of transitions by M on input x. Hence there exists a path from s to t in G_x if and only if M has an accepting computation on input x.

To see an example of a reduction between two NL-complete problems, consider the related problem SC of whether a given directed graph is *strongly connected,* meaning that there is a path from every node u to every other node v. Then GAP $\leq_m^{log}$ SC: Take an instance graph G with distinguished nodes s and t and add an edge from every node to s and from t to every node. Computing this transformation needs only $O(\log n)$ space to store the labels of nodes s and t and find their adjacency information on the input tape, changing '0' for "non-edge" to '1' for "edge" as appropriate while writing to the output tape. This transformation is correct because the new edges cannot cause a path from s to t to exist when there wasn't one beforehand, but do allow any such path to be extended to and from any other pair of nodes. SC belongs to NL since a log-space machine can cycle through all pairs (u, v) and nondeterministically guess a path in each case, so SC is NL-complete.

Further variations of the connectivity theme and many other problems are NL-complete. For an interesting contrast to the current situation with NP-completeness, the *complements* of all these problems are also NL-complete under $\leq_m^{log}$ reductions! This is true because NL is closed under complementation (Theorem 27.4(c) in Chapter 27). The next problem, however, is apparently harder than the NL-complete problems.

CIRCUIT VALUE PROBLEM (CVP)
Instance: A Boolean circuit C (see Section 3 of Chapter 33) and an assignment I to the inputs of C.
Question: Does $C(I)$ evaluate to `true`?

THEOREM 28.10 CVP *is* P*-complete under* $\leq_m^{log}$ *reductions.*

PROOF That CVP $\in$ P is clear, and completeness is essentially proved by the construction in the proof of Theorem 27.10 in Chapter 27, which gives a polynomial-size circuit family $\{C_n\}$ that accepts any given language in P.

Thus CVP belongs to L if and only if P = L, to NL if and only if P = NL, and (owing to NC likewise being closed under $\leq_m^{log}$ reductions), to NC if and only if P = NC. For more P-complete problems, more detail, and discussion of the kind of "intractability" that P-completeness is evidence for, see Chapter 45 in this volume.

The last problem we consider here is a generalization of SAT. *Quantified Boolean formulas* may use the quantifiers $\forall$ and $\exists$ as well as $\{\wedge, \vee, \neg\}$. The formula is *closed* if every variable is quantified. For example, an instance $\phi(x_1, \ldots, x_n)$ of SAT is satisfiable if and only if the closed quantified Boolean formula $\phi' = (\exists x_1)(\exists x_2)\ldots(\exists x_n)\phi$ is true. Another example of a quantified Boolean formula is $\forall x \forall y \exists z(x \wedge (\overline{y} \vee z))$, and this one happens to be false.

QUANTIFIED BOOLEAN FORMULAS (QBF)
Instance: A closed quantified Boolean formula ϕ.
Question: Is ϕ true?

THEOREM 28.11 QBF *is* PSPACE*-complete under* $\leq_m^{log}$ *reductions, hence also under* $\leq_m^{p}$ *reductions.*

PROOF Given an n-variable instance ϕ, $O(n)$ space suffices to maintain a stack with current assignments to each variable, and with this stack one can evaluate ϕ by unwinding one quantifier at a time. So QBF $\in$ PSPACE. For hardness, let $A \in$ PSPACE, and let M be a Turing machine that accepts A in polynomial space. We prove that $A \leq_m^{log}$ QBF.

Construction: Given an input x to M, define the ID graph of G_x as before the proof of Theorem 28.9, but for a polynomial rather than logarithmic space bound. The size of G_x is bounded by $2^{s(n)}$ for some polynomial s, where $n = |x|$. We first define, by induction on r, formulas $\Phi_r(I, J)$ expressing that M started in configuration I can reach configuration J in at most 2^r transitions. The base case formula $\Phi_0(I, J)$ asserts that (I, J) is an edge of G_x.

The idea of the induction is to assert that there is an ID K that is "halfway between" I and J. The straightforward definition $\Phi_r(I, J) := (\exists K)[\Phi_{r-1}(I, K) \wedge \Phi_{r-1}(K, J)]$, however, blows Φ_r up to size exponential in r because of the two occurrences of "Φ_{r-1}" on the right-hand side. The trick is to define, for $r \geq 1$,

$$\Phi_r(I, J) := (\exists K)\left(\forall I', J'\right) : \left[\left(I' = I \wedge J' = K\right) \vee \left(I' = K \wedge J' = J\right)\right] \rightarrow \Phi_{r-1}\left(I', J'\right) .$$

The single occurrence of "Φ_{r-1}" makes the size of $\Phi_r(I, J)$ roughly proportional to r. Now let I_s be the starting ID of M on input x, and I_t the unique accepting ID, from the proof of Theorem 28.9. Then M accepts x if and only if $\Phi_{s(n)}(I_s, I_t)$ is true.

To convert $\Phi_{s(n)}(I_s, I_t)$ into an equivalent instance ϕ_x of QBF, we can represent IDs by blocks of $s(n)$ Boolean variables. All we need to do is code up polynomially-many instances of the predicates "$I = J$" and "M has a transition from I to J." This is similar to the coding done in the proof of Theorem 28.3, since levels of the circuits C_n in that proof are essentially IDs.

Complexity and Correctness. The only real care needed in the straightforward buildup of $\Phi_{s(n)}(I_1, I_R)$ and then ϕ_x is keeping track of the variables. Since there are only polynomially many of them, each has a tag of length $O(\log n)$, and the housekeeping can be done by a deterministic log-space machine. The reduction is correct since $x \in A$ if and only if $\phi_x \in$ QBF.

Remarks. This construction is unaffected if M is nondeterministic, and works for any constructible space bound $s(n) \geq \log n$, producing a formula ϕ_x with $O(s(n)^2)$ Boolean variables. Since ϕ_x can be evaluated deterministically in $O(s(n)^2)$ space, we have also proved Savitch's Theorem (Theorem 27.3(d) in Chapter 27), namely that $\mathsf{NSPACE}[s(n)] \subseteq \mathsf{DSPACE}[s(n)^2]$. We have also essentially proved that PSPACE equals alternating polynomial time (Theorem 27.9(b) in Chapter 27), since ϕ_x can be evaluated in $O(s(n)^2)$ *time* by an ATM M that makes existential and universal moves corresponding to the leading "$\exists$" and "$\forall$" quantifier blocks in ϕ_x. This also yields an alternative reduction from any language A in PSPACE to QBF, since the proof method for Cook's Theorem (28.3) extends to convert this M directly into a quantified Boolean formula.

One family of PSPACE-complete problems consists of connectivity problems for graphs that, although of exponential size, are specified by a "hierarchical," recursive, or some other scheme that enables one to test whether (u, v) is an edge in time polynomial in the length of the labels of u and v. The graph G_x in the last proof is of this kind, since its edge relation $\Phi_1(I, J)$ is polynomial-time decidable. Another family comprises many two-player combinatorial games, where the question is whether the player to move has a winning strategy. A reduction from QBF to the game question transforms a formula such as $(\exists x)(\forall y)(\exists z) \ldots B$ into reasoning of the form "there exists a move for Black such that for all moves by White, there exists a move for Black such that . . . Black wins," starting from a carefully constructed position. Decision problems that exhibit this kind of "there exists. . . for all. . . " alternation (with polynomially many turns) are often PSPACE-complete.

All PSPACE-complete problems are NP-hard, since $\mathsf{NP} \subseteq \mathsf{PSPACE}$; hence $\mathsf{PSPACE} = \mathsf{P}$ if and only if any one of them belongs to P. The only definite lower bound that follows from the results in this section is that no problem that is PSPACE-complete under $\leq_m^{log}$ reductions belongs to L or even NL, because these classes are closed under $\leq_m^{log}$ reductions and $\mathsf{PSPACE} \neq \mathsf{NL}$. It is, however, still possible to have a problem that is PSPACE-complete under $\leq_m^p$ reductions belong to L, since if $\mathsf{PSPACE} = \mathsf{P}$ then all languages in P are PSPACE-complete under $\leq_m^p$ reductions.

To investigate problems and classes within L, we need even finer reducibility relations than $\leq_m^{log}$. Recent results have brought the reductions defined in the next section to the fore. Amazingly, these new reductions, which are based on our tiniest canonical complexity class, are effective not just within L but for natural problems in all the complexity classes in these chapters, including all the problems defined above.

28.6 AC^0 Reducibilities

Recall that a function f belongs to AC^0 if and only if the language $\{\langle x, i, b\rangle : i \leq |f(x)| \wedge \text{bit } i \text{ of } f(x) \text{ equals } b\}$ belongs to AC^0.

- A language A_1 is AC^0 **reducible** to a language A_2, written $A_1 \leq_m^{AC^0} A_2$, if A_1 is many-one reducible to A_2 via a transformation in AC^0.

- A_1 is AC^0-**Turing reducible** to A_2, written $A_1 \leq_T^{AC^0} A_2$, if A_1 is recognized by a DLOGTIME-uniform family of circuits of polynomial size and constant depth, consisting of $\neg$ gates, unbounded fan-in $\wedge$ and $\vee$ gates, and oracle gates for A_2. (An oracle gate for A_2 takes m inputs $x_1, \ldots, x_m$ and outputs 1 if $x_1 \ldots x_m$ is in A_2, and outputs 0 otherwise.)

The next theorem summarizes basic relationships among the five reducibility relations defined thus far.

THEOREM 28.12 *For any languages A_1, A_2, and A_3:*
(Transitivity)

(a) *If $A_1 \leq_m^{AC^0} A_2$ and $A_2 \leq_m^{AC^0} A_3$, then $A_1 \leq_m^{AC^0} A_3$.*
(b) *If $A_1 \leq_T^{AC^0} A_2$ and $A_2 \leq_T^{AC^0} A_3$, then $A_1 \leq_T^{AC^0} A_3$.*

(Refinement)

(c) $A_1 \leq_m^{AC^0} A_2 \Longrightarrow A_1 \leq_m^{log} A_2 \Longrightarrow A_1 \leq_m^{p} A_2 \Longrightarrow A_1 \leq_T^{p} A_2$.
(d) $A_1 \leq_m^{AC^0} A_2 \Longrightarrow A_1 \leq_T^{AC^0} A_2 \Longrightarrow A_1 \leq_T^{p} A_2$.

In prose, (c) says that AC^0 reducibility implies log-space reducibility, which implies Karp reducibility, which implies Cook reducibility; and (d) says that AC^0 reducibility implies AC^0-Turing reducibility, which implies Cook reducibility. However, AC^0-Turing reducibility is known *not* to imply log-space reducibility or even Karp reducibility—any language that does not many-one reduce to its complement shows this.

Next, we list which of our canonical complexity classes are closed under which reducibilities. Note that the subclasses of P are not known to be closed under the more powerful reducibilities, and that the nondeterministic time classes are not known to be closed under the Turing reducibilities—mainly because they are not known to be closed under complementation.

THEOREM 28.13

(a) P, PSPACE, EXP, *and* EXPSPACE *are closed under Cook reducibility, and hence under* AC^0 *reducibility,* AC^0*-Turing reducibility, log-space reducibility, and Karp reducibility as well.*
(b) NP *and* NEXP *are closed under Karp reducibility, hence also under* AC^0 *reducibility and log-space reducibility.*
(c) L, NL, *and* NC *are closed under both log-space reducibility and* AC^0*-Turing reducibility, hence also under* AC^0 *reducibility.*
(d) NC^1, TC^0, *and* AC^0 *are closed under* AC^0*-Turing reducibility, hence also under* AC^0 *reducibility.*

For contrast, note that E and NE are *not* closed under AC^0 reducibility—hence they are not closed under any of the other reducibilities, either. To see this, let A be any language in $\mathsf{EXP} - \mathsf{E}$; such languages exist by the time hierarchy theorem (Theorem 27.5 in Chapter 27). Then for some $k > 0$, the language $A_k = \{x10^{|x|^k} \mid x \in A\}$ belongs to E, and it is easy to see that $A \leq_m^{AC^0} A_k$. If E were closed under $\leq_m^{AC^0}$ reductions, A would be in E, a contradiction. The same can be done for NE using NEXP in place of EXP.

Note that this gives an easy proof that $\mathsf{E} \neq \mathsf{NP}$, because NP has a closure property that E does not share. On the other hand, although this inequality tells us that exactly one of the following must hold,

- $\mathsf{NP} \subset \mathsf{E}$
- $\mathsf{E} \subset \mathsf{NP}$
- $\mathsf{NP} \not\subseteq \mathsf{E}$ and $\mathsf{E} \not\subseteq \mathsf{NP}$,

it is not known *which* of these is true.

Why Have So Many Kinds of Reducibility?

We have already discussed one reason to consider different kinds of reducibility: in order to explore the important subclasses of P, more restrictive notions such as $\leq_m^{AC^0}$ and $\leq_m^{log}$ are required. However, that does not explain why we study both Karp *and* Cook reducibility, or both AC^0 and AC^0-Turing reductions. It is worth taking a moment to explain this.

If our goal were merely to classify the deterministic complexity of problems, then Cook reducibility ($\leq_T^p$) would be the most natural notion to study. The class of problems that are Cook reducible to A (usually denoted by P^A) characterizes what can be computed quickly if A is easy to compute. Note in particular that A is Cook-reducible to its complement $\overline{A}$, and A and $\overline{A}$ have the same deterministic complexity.

However, if A is an NP-complete language, then A probably does *not* have the same *nondeterministic* complexity as $\overline{A}$. That is, $\overline{A} \in \mathsf{NP}$ only if $\mathsf{NP} = \mathsf{co\text{-}NP}$. It is worth emphasizing that, if we know only that A is complete for NP under $\leq_T^p$ reductions, the hypothesis $\overline{A} \in \mathsf{NP}$ does not allow us to conclude $\mathsf{NP} = \mathsf{co\text{-}NP}$. That is, we get *stronger* evidence that $\overline{A}$ has high nondeterministic complexity, if we know that A is complete for NP under the more restrictive kind of reducibility.

This is a general phenomenon: if we know that a language is complete under more restrictive kind of reducibility, then we know more about its complexity. We have already seen one other example: knowing a language A is complete for PSPACE under $\leq_m^{log}$ reductions tells you that $A \notin \mathsf{NL}$, whereas completeness under $\leq_m^p$ reductions does not even entail that $A \notin \mathsf{L}$. In the latter case we must appeal to the unproven conjecture that $\mathsf{P} \neq \mathsf{PSPACE}$ to infer that A is not in L. For another example, if A is complete for NP under $\leq_m^{AC^0}$ reductions, then we know that $A \notin \mathsf{AC}^0$, whereas if we know only that A is NP-complete under $\leq_m^{log}$ reductions, then we cannot conclude anything about the complexity of A, because we cannot yet rule out the possibility that $\mathsf{L} = \mathsf{NP}$.

Canonical Classes and Complete Problems

It is an amazing and surprising fact that most computational problems that arise in practice turn out to be complete for some natural complexity class—and complete under some extremely restrictive reducibility such as $\leq_m^{AC^0}$. Indeed, the lion's share of those natural problems known to be complete for NP under $\leq_m^p$ reductions, and for P under $\leq_m^{log}$ reductions, etc., are in fact complete under $\leq_m^{AC^0}$ reductions, too. We observe this after filling out our spectrum of complexity classes with some more decision problems.

Integer Multiplication
Instance: The binary representation of integers x and y, and a number i.
Question: Is the ith bit of the binary representation of $x \cdot y$ equal to 1?

Boolean Formula Value Problem (BFVP)
Instance: A Boolean formula ϕ and a 0-1 assignment I to the variables in ϕ.
Question: Does $\phi(I)$ evaluate to `true`?

Degree-One Connectivity (GAP_1)
Instance: A directed graph in which each node has at most one outgoing edge, and nodes s, t of G.
Question: Is there a path from node s to node t in G?

Regular Expressions with $(\cup, \cdot, ^*)$
Instance: A regular expression α with the standard union, concatenation, and Kleene-star operations (see Chapter 25).
Question: Is there a string that does *not* match α?

REGULAR EXPRESSIONS WITH $(\cup, \cdot, ^2)$
Instance: A regular expression α with union, concatenation, and "squaring" operators (where α^2 denotes $\alpha \cdot \alpha$).
Question: Is there a string that does not match α?

REGULAR EXPRESSIONS WITH $(\cup, \cdot, ^*, ^2)$
Instance: A regular expression α composed of the union, concatenation, "squaring," and Kleene star operators.
Question: Is there a string that does not match α?

N × N CHECKERS
Instance: A position in checkers played on an $N \times N$ board, with Black to move.
Question: Is this a winning position for Black?

THEOREM 28.14 *The following problems are complete for the given complexity classes under $\leq_m^{AC^0}$ reductions, except that* INTEGER MULTIPLICATION *is only known to be complete under $\leq_T^{AC^0}$ reductions.*

- TC^0 : INTEGER MULTIPLICATION.
- NC^1 : BFVP.
- L : GAP_1.
- NL : GAP.
- P : CVP.
- NP : SAT, CLIQUE, VERTEX COVER, *and so on.*
- PSPACE : QBF, REGULAR EXPRESSIONS WITH $(\cup, \cdot, ^*)$.
- EXP : N × N CHECKERS.
- NEXP : REGULAR EXPRESSIONS WITH $(\cup, \cdot, ^2)$.
- $\mathsf{EXPSPACE}$: REGULAR EXPRESSIONS WITH $(\cup, \cdot, ^*, ^2)$.

The last three problems also belong to E, NE, and $\mathsf{DSPACE}[2^{O(n)}]$, respectively (under suitable encodings), and so they are complete for these respective classes as well. Note that a "tiny" reducibility still gives complete problems for a big class! However, TC^0 is not known to have any complete problems under $\leq_m^{AC^0}$ reductions.

Note that the class NC does not appear anywhere in the list above. NC is not known (or generally believed) to have any complete language under log-space reductions. In fact, if NC does have a language that is complete under $\leq_m^{log}$ reducibility, then there is some k such that $\mathsf{NC}^k = \mathsf{NC}^{k+1} = \cdots = \mathsf{NC}$. This is considered unlikely. This behavior is typical of certain "hierarchy classes," and the *polynomial hierarchy* class PH (defined in the next chapter) behaves similarly with regard to $\leq_m^p$ reductions.

To (im)prove the claim about $\leq_m^{AC^0}$ reductions in Theorem 28.14, we can show that all the reductions in this chapter are computable by uniform AC^0 circuits *without any* $\wedge$ *or* $\vee$ *gates at all!* The circuits have only the constants 0 and 1, the inputs $x_1, \ldots, x_n$, and $\neg$ gates. A function $f : \{0,1\}^* \to \{0,1\}^*$ computed by circuits of this kind is called a *projection.* In a projection, every bit j of the output depends on at most one bit i of the input: it is either always 0, always 1, always x_i, or always the complement of x_i. An AC^0 projection f can be defined without reference to circuits: there is a deterministic Turing machine M that, given binary numbers n and j, decides in $O(\log n)$ time which of these four cases holds, also computing i in the latter two cases. (Note that $m = |f(x)|$ depends only on $n = |x|$; M must also test whether $j > m$ in $O(\log n)$ time.) Now observe:

- Most of the construction of ϕ in our proof of Cook's Theorem depended only on the length n of the argument x. The only dependence on x itself was in the very last piece of ϕ, and under the encoding, x was basically copied bit by bit as the signs of the literals. The construction for P-completeness of CVP has the same property.
- In the reductions shown from SAT to graph problems, each *edge* of the target graph depended on only one bit of information of the form, "is variable x_i in clause C_j?" (To meet the technical requirements for projections we must use a convoluted encoding of formulas and graphs, but this is the essential idea.)
- In the construction for NL-completeness of GAP, each edge of G_x depended on what the machine N could do while reading just one bit of x.
- Even in the trickiest proof in this chapter, for PSPACE-completeness of QBF, the only ID whose dependence on x needs to be made explicit is the starting ID I_s, and for this x is just copied bit by bit.

Similar ideas work for BFVP, and GAP_1; the reader is invited to investigate the remaining problems.

Our point is not to emphasize projections at the expense of other reductions, but to show that the reductions themselves can be incredibly easy to compute. Thus the complexity levels shown in these completeness results are entirely intrinsic to the target problems. In practice, with classes around P or NP, finding and proving a $\leq_m^p$ or $\leq_m^{log}$ reduction is usually easier, free of encoding fuss, and sufficient for one's purposes.

The list in Theorem 28.14 only begins to illustrate the phenomenon of completeness. Take a computational problem from the practical literature, and chances are it is complete for one of our short list of canonical classes. Here are some more examples, all complete under AC^0 reductions: Does a given deterministic finite automaton M accept a given input x?—L-complete. For certain fixed M, however, the problem is NC^1-complete, and since every regular language belongs to NC^1, this gives a sense in which NC^1 characterizes the complexity of regular languages. Do $N-1$ pairs (i, j) of numbers in $\{1, \ldots, N\}$ form one linked list starting from 1 and ending at N?—L-complete. Various related problems about permutations, list-ranking, depth-first search, and breadth-first search are also L-complete. Satisfiability for 2CNF formulas?—NL-complete. Is $L(G) = \emptyset$ for a given context-free grammar G?—P-complete. REGULAR EXPRESSIONS WITH $(\cup, \cdot)$ only?—NP-complete. Can a multithreaded finite-state program avert deadlock?—PSPACE-complete. Game problems in which play must halt after polynomially many moves tend to be PSPACE-complete, but if exponentially long games are possible, as with suitably generalized versions of Chess and Go as well as Checkers to arbitrarily large boards, they tend to be EXP-complete—and hence intractable to solve!

There are some exceptional problems: connectivity for undirected graphs (L-hard, not known to be in L), matrix determinant, matrix permanent (see Chapter 29), and the "NP-intermediate" problems mentioned at the end of the Section 28.4. But overall, no one would have expected thirty years ago that so many well-studied problems would quantize into so few complexity levels under efficient reductions.

Changing the conditions on a problem also often makes it jump into a new canonical completeness level. The regular-expression problems show this amply. Special cases of NP-complete problems overwhelmingly tend either to remain NP-hard or jump all the way down to P. BFVP is the special case of CVP where every gate in the circuit has fanout 1. Even SAT itself is a restricted case of QBF.

Problems complete for a given class share an underlying mathematical structure that is brought out by the reductions between them. Note that the transformations map tiny local features of one instance x

to tiny local features of $f(x)$—particularly when f is a projection! How such local transformations can propagate global decision properties between widely varying problems is a scientific phenomenon that has been studied for itself.

The main significance of completeness, however, is the evidence of intractability it provides. Although in many cases this evidence is based on an unproven conjecture, sometimes it is absolute. Consider the problem REGULAR EXPRESSIONS WITH $(\cup, ^2, \cdot)$, which is complete for NEXP. If this problem were in P, then by closure under Karp reducibility (Theorem 28.1 in Section 28.2), we would have $\mathsf{NEXP} \subseteq \mathsf{P}$, a contradiction of the Hierarchy Theorems (Theorem 27.5 in Chapter 27). Therefore, this decision problem is infeasible: it has no polynomial-time algorithm. In contrast, decision problems in $\mathsf{NEXP} - \mathsf{P}$ that have been constructed by diagonalization are artificial problems that nobody would want to solve anyway. It is an important point that although diagonalization produces unnatural problems by itself, the combination of diagonalization and completeness shows that *natural* problems are intractable.

However, the next section points out some limitations of current diagonalization techniques.

28.7 Relativization of the P vs. NP Problem

Let A be a language. Define P^A (respectively, NP^A) to be the class of languages accepted in polynomial time by deterministic (nondeterministic) oracle Turing machines with oracle A.

Proofs that use the diagonalization technique on Turing machines without oracles generally carry over to oracle Turing machines. Thus, for instance, the proof of DTIME hierarchy theorem also shows that, for *any* oracle A, $\mathsf{DTIME}^A[n^2]$ is properly contained in $\mathsf{DTIME}^A[n^3]$. This can be seen as a *strength* of the diagonalization technique, since it allows an argument to "relativize" to computation carried out relative to an oracle. In fact, there are examples of lower bounds (for deterministic, "unrelativized" circuit models) that make crucial use of the fact that the time hierarchies relativize in this sense.

But it can also be seen as a weakness of the diagonalization technique. The following important theorem demonstrates why.

THEOREM 28.15 *There exist languages A and B such that $\mathsf{P}^A = \mathsf{NP}^A$, and $\mathsf{P}^B \neq \mathsf{NP}^B$.*

This shows that resolving the P vs. NP question requires techniques that do not relativize, i.e., that do not apply to oracle Turing machines too. Thus, diagonalization as we currently know it is unlikely to succeed in separating P from NP, because the diagonalization arguments we know (and in fact *most* of the arguments we know) relativize. The only major nonrelativizing proof technique in complexity theory appears to be the technique used to prove that $\mathsf{IP} = \mathsf{PSPACE}$. (See Section 29.5 and the end notes to Chapter 29.)

28.8 Sparse Languages

Despite their variety, the known NP-complete languages are similar in the following sense. Two languages A and B are **P-isomorphic** if there exists a function h such that

- For all x, $x \in A$ if and only if $h(x) \in B$,
- h is bijective (i.e., one-to-one and onto), and
- Both h and its inverse h^{-1} are computable in polynomial time.

All known NP-complete languages are P-isomorphic. Thus, in some sense, they are merely different encodings of the same problem. This is yet another example of the "amazing fact" alluded to in Section 28.6 that natural NP-complete languages exhibit unexpected similarities.

Because of this and other considerations, Berman and Hartmanis [5] conjectured that *all* NP-complete languages are P-isomorphic. This conjecture implies that $\mathsf{P} \neq \mathsf{NP}$, because if $\mathsf{P} = \mathsf{NP}$, then there are finite NP-complete languages, and no infinite language (such as SAT) can be isomorphic to a finite language.

Between the finite languages and the infinite languages lie the sparse languages, which are defined as follows. For a language A over an alphabet Σ, the **census function** of A, denoted $c_A(n)$, is the number of words x in A such that $|x| \leq n$. Clearly, $c_A(n) < |\Sigma|^{n+1}$. If $c_A(n)$ is bounded by a polynomial in n, then A is **sparse**. From the definitions, it follows that if A is sparse, and A is P-isomorphic to B, then B is sparse.

If a sparse NP-complete language S exists, then we could use S to solve NP-complete problems efficiently, by the following method. Let A be a language in NP, and let f be a transformation function that reduces A to S in polynomial time $t_f(n)$. To quickly decide membership in A for every word x whose length is at most n, deterministically, compute $f(x)$ and check whether $f(x) \in S$ by looking up $f(x)$ in a table. The table would consist of all words in S whose length is at most $t_f(n)$. The number of entries in this table would be $c_S(t_f(n))$, which is polynomial in n, and hence the total space occupied by the table would be bounded by a polynomial in n.

The Berman–Hartmanis conjecture implies that there is no sparse NP-complete language, however, because SAT is not sparse. A stronger reason for believing there are no such languages is:

THEOREM 28.16 *If a sparse* NP*-complete language exists, then* $\mathsf{P} = \mathsf{NP}$.

Very recently, it has been shown that other complexity classes are similarly unlikely to possess sparse complete languages.

THEOREM 28.17

1. *If there is a sparse language that is complete for* P *under log-space reducibility, then* $\mathsf{L} = \mathsf{P}$.
2. *If there is a sparse language that is complete for* NL *under log-space reducibility, then* $\mathsf{L} = \mathsf{NL}$.

28.9 Advice, Circuits, and Sparse Oracles

In the computation of an oracle Turing machine, the oracle language provides assistance in the form of answers to queries, which may depend on the input word. A different kind of assistance, called "advice," depends only on the length of the input word.

Recall the following definitions from Section 27.3: A function α is an **advice function** if for every nonnegative integer n, $\alpha(n)$ is a binary word whose length is bounded by a polynomial in n. (An advice function need not be total recursive.)

- $\mathsf{P/poly}$ is the class of languages $A = \{x : \langle x, \alpha(|x|)\rangle \in A'\}$ for some advice function α and some language A' in P.

As was pointed out in that section, $\mathsf{P/poly}$ comprises those problems that can be solved by (nonuniform) circuit families of polynomial size.

Berman and Hartmanis also pointed out the following connection between sparse languages and circuit complexity.

THEOREM 28.18 *The following are equivalent:*

1. $A \in \mathsf{P/poly}$.

2. *A is decided by a circuit family of polynomial size complexity.*
3. *$A \in \mathsf{P}^S$ for some sparse language S, that is, A is Cook reducible to a sparse language.*

In Section 28.8 we discussed the consequences of having a sparse language complete for NP under Karp reducibility. Namely, for any n and for any problem A in NP, there would exist a small table that one could use to efficiently solve A on instances of length at most n. This would also be true if there were a sparse language complete for NP under Cook reducibility. This is equivalent to having $\mathsf{NP} \subseteq \mathsf{P/poly}$. But there is one notable difference between these two situations. By Theorem 28.16, if there is a sparse language complete under Karp reductions, then $\mathsf{P} = \mathsf{NP}$. It is not known if we can conclude that $\mathsf{P} = \mathsf{NP}$ assuming only that there is a sparse language complete under Cook reductions. The reasons for believing that $\mathsf{NP} \not\subseteq \mathsf{P/poly}$ are nearly as strong as those for believing that $\mathsf{NP} \neq \mathsf{P}$; for one, $\mathsf{NP} \subseteq \mathsf{P/poly}$ would imply that there are polynomial-size lookup tables to help one efficiently solve instances of SAT or factor integers, etc. There is other evidence against $\mathsf{NP} \subseteq \mathsf{P/poly}$ that we present in Section 29.2.

These connections to circuit complexity and to the isomorphism conjecture have motivated a great deal of research into the complexity of sparse languages under various types of reducibility.

28.10 Research Issues and Summary

Thanks to the notions of reducibility and completeness, it is possible to give "tight lower bounds" on the complexity of many natural problems, even without yet knowing whether $\mathsf{P} = \mathsf{NP}$. In this chapter, we have seen some examples showing how to prove that problems are NP-complete. We have also explored some of the other notions to which reducibility gives rise, including the notions of relativized computation, P-isomorphism, and the complexity of sparse languages.

There are many natural and important problems that are complete for complexity classes that do not appear in our list of "canonical" complexity classes. In order that these problems can be better understood, it is necessary to introduce some additional complexity classes. That is the topic of Chapter 29.

28.11 Defining Terms

Configuration: For a Turing machine, synonymous with **instantaneous description**.

Cook reduction ($\leq_T^p$): A **reduction** computed by a deterministic polynomial time **oracle Turing machine**.

Cook's theorem: The theorem that the language SAT of satisfiable Boolean formulas (defined in Chapter 33) is **NP-complete.**

Instantaneous description (ID): A string that encodes the current state, head position, and (work) tape contents at one step of a Turing machine computation.

Karp reduction ($\leq_m^p$): A **reduction** given by a polynomial-time computable **transformation function**.

NP: The class of languages accepted by **N**ondeterministic **P**olynomial-time Turing machines. The acronym does not stand for "nonpolynomial"—every problem in P belongs to NP.

NP-complete: A language A is NP-complete if A belongs to NP and every language in NP **reduces** to A. Usually this term refers to **Karp reducibility**.

NP-hard: A language A is NP-hard if every language in NP **reduces** to A. Usually this term refers to **Cook reducibility**.

Oracle Turing machine: A Turing machine that may write "query strings" y on a special tape and learn instantly whether y belongs to a language A_2 given as its *oracle.* These are defined in more detail in Chapter 24.

P: The class of languages accepted by (equivalently, decision problems solved by) deterministic polynomial-time Turing machines. Less technically, the class of feasibly solvable problems.

Reduction: A function or algorithm that maps a given instance of a (decision) problem A_1 into one or more instances of another problem A_2, such that an efficient solver for A_2 could be plugged in to yield an efficient solver for A_1.

Sparse language: A language with a polynomially bounded number of strings of any given length.

Transformation function: A function f that maps instances x of one decision problem A_1 to those of another problem A_2 such that for all such x, $x \in A_1 \iff f(x) \in A_2$. (Here we identify a decision problem with the language of inputs for which the answer is "yes.")

References

[1] Allender, E., Oracles versus proof techniques that do not relativize. In *Proc. 1st Annual International Symposium on Algorithms and Computation,* 1990.

[2] Baker, T., Gill, J., and Solovay, R., relativizations of the P=NP? question. *SIAM J. Comput.,* 4, 431–442, 1975.

[3] Balcázar, J., Díaz, J., and Gabarró, J., *Structural Complexity I,II,* Springer Verlag, 1990. Part I published in 1988.

[4] Barrington, D.M., Immerman, N., and Straubing, H., On uniformity within NC^1. *J. Comp. Sys. Sci.,* 41, 274–306, 1990.

[5] Berman, L. and Hartmanis, J., On isomorphisms and density of NP and other complete sets. *SIAM J. Comput.,* 6, 305–321, 1977.

[6] Cai, J.-Y. and Ogihara, M., Sparse hard sets. In *Complexity Theory Retrospective II,* L. Hemaspaandra and A. Selman, Eds., 53–80. Springer-Verlag, 1997.

[7] Cai, J. and Sivakumar, D., The resolution of a Hartmanis conjecture. To appear in the *J. Comp. Sys. Sci.*

[8] Cai, J., Naik, A., and Sivakumar, D., On the existence of hard sparse sets under weak reductions. In *Proc. 13th Annual Symposium on Theoretical Aspects of Computer Science,* Vol. 1046, *Lect. Notes in Comp. Sci.,* 307–318, Springer-Verlag, 1996.

[9] Cook, S., The complexity of theorem-proving procedures. In *Proc. 3rd Annual ACM Symposium on the Theory of Computing,* 151–158, 1971.

[10] Fortnow, L., The role of relativization in complexity theory. *Bull. EATCS,* 52, 229–244, 1994.

[11] Garey, M. and Johnson, D.S., *Computers and Intractability: A Guide to the Theory of NP-Completeness,* Freeman, 1988. First edition was 1979.

[12] Greenlaw, R., Hoover, J., and Ruzzo, W.L., *Limits to Parallel Computation: P-Completeness Theory,* Oxford University Press, 1995. Excerpts available at `http://web.cs.ualberta.ca:80/~hoover/P-complete/`.

[13] Hartmanis, J., New directions in structural complexity theory. In *Proc. 15th Annual International Conference on Automata, Languages, and Programming,* Vol. 317, *Lect. Notes in Comp. Sci.,* 271–286, Springer-Verlag, 1988.

[14] Hopcroft, J. and Ullman, J., *Introduction to Automata Theory, Languages, and Computation,* Addison–Wesley, Reading, MA, 1979.

[15] Johnson, D.S. and McGeogh, L., The traveling salesman problem: a case study in local optimization. In *Local Search in Combinatorial Optimization,* E.H.L. Aarts and J.K. Lenstra, Eds., John Wiley & Sons, New York, 1997.

[16] Jones, N., Space-bounded reducibility among combinatorial problems. *J. Comp. Sys. Sci.,* 11, 68–85, 1975. Corrigendum *J. Comp. Sys. Sci.,* 15, 241, 1977.

[17] Karp, R., Reducibility among combinatorial problems. In *Complexity of Computer Computations,* R.E. Miller and J.W. Thatcher, Eds., 85–104, Plenum Press, 1972.
[18] Ladner, R., The circuit value problem is log-space complete for P. *SIGACT News,* 7, 18–20, 1975.
[19] Ladner, R., On the structure of polynomial-time reducibility. *J. Assn. Comp. Mach.,* 22, 155–171, 1975.
[20] Levin, L., Universal sequential search problems. *Problems of Information Transmission,* 9, 265–266, 1973.
[21] Mahaney, S., Sparse complete sets for NP: solution of a conjecture of Berman and Hartmanis. *J. Comp. Sys. Sci.,* 25, 130–143, 1982.
[22] Papadimitriou, C., *Computational Complexity,* Addison-Wesley, Reading, MA, 1994.
[23] Savitch, W., Relationship between nondeterministic and deterministic tape complexities. *J. Comp. Sys. Sci.,* 4, 177–192, 1970.
[24] Schnorr, C., Satisfiability is quasilinear complete in NQL. *J. Assn. Comp. Mach.,* 25, 136–145, 1978.
[25] Stockmeyer, L., The complexity of decision problems in automata theory and logic. Technical Report MAC-TR-133, Project MAC, M.I.T., Cambridge, MA, 1974.
[26] van Melkebeek, D. and Ogihara, M., Sparse hard sets for P. In *Advances in Complexity and Algorithms,* Du, D. and Ko, K., Eds., Kluwer Academic Press, 1997. In press.
[27] Wagner, K. and Wechsung, G., *Computational Complexity,* D. Reidel, 1986.

Further Information

Cook's Theorem was originally stated and proved for Cook reductions [9], and later for Karp reductions [17]. Independently, Levin [20] proved an equivalent theorem using a variant of Karp reductions; sometimes Theorem 28.3 is called the *Cook–Levin theorem.* Our circuit-based proof stems from Schnorr [24], where SAT is shown to be complete for nondeterministic quasi-linear time under deterministic quasi-linear time reductions. Karp [17] showed the large-scale impact of NP-completeness, and Theorems 28.4, 28.5, and 28.6 come from there.

A much more extensive discussion of NP-completeness and techniques for proving problems to be NP-complete may be found in [11]. This classic reference also contains a list of hundreds of NP-complete problems. An analogous treatment of problems complete for P can be found in [12]—see also Chapter 48 in this volume. Most textbooks on algorithm design or complexity theory also contain a discussion of NP-completeness.

Primary sources for other completeness results in this chapter include: Theorem 28.9 [23]; Theorem 28.10 [18]; Theorem 28.11 [25]. Jones [16] studied log-space reductions in detail, and also introduced $\leq_m^{AC^0}$ reductions under the name "log-bounded rudimentary reductions." An important paper for the theory of AC^0 is [4], which also discusses the complete problems in for TC^0 and NC^1 in Theorem 28.14. The remaining problems in Theorem 28.14 may be found in [27]. The texts by Hopcroft and Ullman [14] and Papadimitriou [22] give more examples of problems complete for other classes.

There are many other notions of reducibility, including *truth-table, randomized,* and *truth-table* reductions. A good treatment of this material can be found in the two volumes of [3].

The first relativization results, including Theorem 28.15, are due to Baker et al. [2], and many papers have proved more of them. The role of relativization in complexity theory (and even the question of what constitutes a nonrelativizing proof technique) is fraught with controversy. Longer discussions of the issues involved may be found in [1, 10, 13].

Sparse languages came to prominence in connection with the Berman–Hartmanis conjecture [5], where Theorem 28.18 is ascribed to Albert Meyer. Theorem 28.16 is from [21], and Theorem 28.17 from [7, 8]. Two new surveys of sparse languages and their impact on complexity theory are [6] and [26].

Information about practical efforts to solve instances of NP-complete and other hard problems is fairly easy to find on the World Wide Web, by searches on problem names such as `Traveling Salesman` (note that variants such as "Salesperson" and the British "Travelling" are also used). Three helpful sites with further links are the Center for Discrete Mathematics and Computer Science (DIMACS), the *TSP Library* (TSPLIB), and the *Genetic Algorithms Archive;*

`http://dimacs.rutgers.edu/`
`http://www.iwr.uni-heidelberg.de/iwr/comopt/soft/TSPLIB95/TSPLIB.html`
`http://www.aic.nrl.navy.mil:80/galist/`

are the current URLs. TSPLIB has downloadable test instances of the TSP problem drawn mostly from practical sources. There is also an extensive bibliography on the TSP problem called TSPBIB, maintained by P. Moscato at: `http://www.ing.unlp.edu.ar/cetad/mos/TSPBIB_home.html`.

See also the **Further Information** section of Chapter 29.

29

Other Complexity Classes and Measures[1]

Eric Allender
Rutgers University

Michael C. Loui
University of Illinois at Urbana-Champaign

Kenneth W. Regan
State University of New York at Buffalo

29.1 Introduction

In the previous two chapters, we have

- Introduced the basic complexity classes,
- Summarized the known relationships between these classes, and
- Seen how reducibility and completeness can be used to establish tight links between natural computational problems and complexity classes.

Some natural problems seem not to be complete for any of the complexity classes we have seen so far. For example, consider the problem of taking as input a graph G and a number k, and deciding whether k

[1] Eric Allender — Supported by the National Science Foundation under Grant CCR-9509603. Portions of this work were performed while a visiting scholar at the Institute of Mathematical Sciences, Madras, India.
Michael C. Loui — Supported by the National Science Foundation under Grant CCR-9315696.
Kenneth W. Regan — Supported by the National Science Foundation under Grant CCR-9409104.

is exactly the length of the shortest traveling salesperson's tour. This is clearly related to the TSP problem discussed in Chapter 28, but in contrast to TSP, it seems not to belong to NP, and also seems not to belong to co-NP.

To classify and understand this and other problems, we will introduce a few more complexity classes. We cannot discuss all of the classes that have been studied—there are further pointers to the literature at the end of this chapter. Our goal is to describe some of the most important classes, such as those defined by probabilistic and interactive computation.

A common theme is that the new classes arise from the interaction of complexity theory with other fields, such as randomized algorithms, formal logic, combinatorial optimization, and matrix algebra. Complexity theory provides a common formal language for analyzing computational performance in these areas. Other examples can be found in other chapters of this *Handbook.*

29.2 The Polynomial Hierarchy

Recall from Theorem 27.9(b) in Chapter 27 that PSPACE is equal to the class of languages that can be recognized in polynomial time on an alternating Turing machine, and that NP corresponds to polynomial time on a nondeterministic Turing machine, which is just an alternating Turing machine that uses only existential states. Thus, in some sense, NP sits near the very "bottom" of PSPACE, and as we allow more use of the power of alternation, we slowly climb up toward PSPACE.

Many natural and important problems reside near the bottom of PSPACE in this sense, but are neither known nor believed to be in NP. (We shall see some examples later in this chapter.) Most of these problems can be accepted quickly by alternating Turing machines that make only two or three alternations between existential and universal states. This observation motivates the definition in the next paragraph.

With reference to Chapter 24, define a k-**alternating Turing machine** to be a machine such that on every computation path, the number of changes from an existential state to universal state, or from a universal state to an existential state, is at most $k-1$. Thus, a nondeterministic Turing machine, which stays in existential states, is a 1-alternating Turing machine.

It turns out that the class of languages recognized in polynomial time by 2-alternating Turing machines is precisely $\mathsf{NP}^{\mathrm{SAT}}$. This is a manifestation of something more general, and it leads us to the following definitions.

Let $\mathcal{C}$ be a class of languages. Define

- $\mathsf{NP}^{\mathcal{C}} = \bigcup_{A\in\mathcal{C}} \mathsf{NP}^{A}$,
- $\Sigma_0^P = \Pi_0^P = \mathsf{P}$;

and for $k \geq 0$, define

- $\Sigma_{k+1}^P = \mathsf{NP}^{\Sigma_k^P}$,
- $\Pi_{k+1}^P = \text{co-}\Sigma_{k+1}^P$.

Observe that $\Sigma_1^P = \mathsf{NP}^{\mathsf{P}} = \mathsf{NP}$, because each of polynomially many queries to an oracle language in P can be answered directly by a (nondeterministic) Turing machine in polynomial time. Consequently, $\Pi_1^P = \text{co-NP}$. For each k, $\Sigma_k^P \subseteq \Sigma_{k+1}^P$, and $\Pi_k^P \subseteq \Sigma_{k+1}^P$, but these inclusions are not known to be strict. See Fig. 29.1.

The classes Σ_k^P and Π_k^P constitute the **polynomial hierarchy.** Define

$$\mathsf{PH} = \bigcup_{k\geq 0} \Sigma_k^P .$$

It is straightforward to prove that $\mathsf{PH} \subseteq \mathsf{PSPACE}$, but it is not known whether the inclusion is strict. In fact, if $\mathsf{PH} = \mathsf{PSPACE}$, then the polynomial hierarchy collapses to some level, i.e., $\mathsf{PH} = \Sigma_m^P$ for some m.

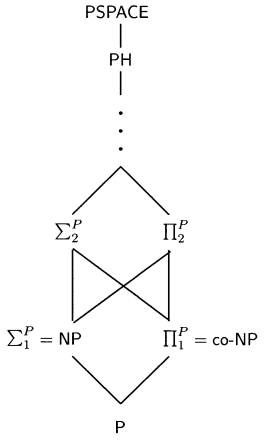


FIGURE 29.1 The polynomial hierarchy.

We have already hinted that the levels of the polynomial hierarchy correspond to k-alternating Turing machines. The next theorem makes this correspondence explicit, and also gives us a third equivalent characterization.

THEOREM 29.1 *For any language A, the following are equivalent:*

1. $A \in \Sigma_k^P$.
2. *A is decided in polynomial time by a k-alternating Turing machine that starts in an existential state.*
3. *There exists a language $B \in \mathsf{P}$ and a polynomial p such that for all x, $x \in A$ if and only if*

$$(\exists y_1 : |y_1| \leq p(|x|))\ (\forall y_2 : |y_2| \leq p(|x|)) \cdots (Q y_k : |y_k| \leq p(|x|))\ [(x, y_1, \ldots, y_k) \in B]\ ,$$

where the quantifier Q is $\exists$ if k is odd, $\forall$ if k is even.

In Section 28.9 we discussed some of the startling consequences that would follow if NP were included in P/poly, but observed that this inclusion was not known to imply P = NP. It is known, however, that if NP $\subseteq$ P/poly, then PH collapses to its second level, Σ_2^P [32]. It is generally considered likely that PH does not collapse to any level, and hence that all of its levels are distinct. Hence this result is considered strong evidence that NP is not a subset of P/poly.

Also inside the polynomial hierarchy is the important class BPP of problems that can be solved efficiently and reliably by probabilistic algorithms, to which we now turn.

29.3 Probabilistic Complexity Classes

Since the 1970s, with the development of randomized algorithms for computational problems (see Chapter 15), complexity theorists have placed randomized algorithms on a firm intellectual foundation. In this section, we outline some basic concepts in this area.

A **probabilistic Turing machine** M can be formalized as a nondeterministic Turing machine with exactly two choices at each step. During a computation, M chooses each possible next step with independent probability 1/2. Intuitively, at each step, M flips a fair coin to decide what to do next. The probability of a computation path of t steps is $1/2^t$. The probability that M accepts an input string x, denoted by $p_M(x)$, is the sum of the probabilities of the accepting computation paths.

Throughout this section, we consider only machines whose time complexity $t(n)$ is time-constructible. Without loss of generality, we may assume that every computation path of such a machine halts in exactly t steps.

Let A be a language. A probabilistic Turing machine M decides A with

		for all $x \in A$	for all $x \notin A$
unbounded two-sided error	if	$p_M(x) > 1/2$	$p_M(x) \leq 1/2$
bounded two-sided error	if	$p_M(x) > 1/2 + \epsilon$	$p_M(x) < 1/2 - \epsilon$
		for some constant ϵ	
one-sided error	if	$p_M(x) > 1/2$	$p_M(x) = 0$

Many practical and important probabilistic algorithms make one-sided errors. For example, in the Solovay–Strassen primality testing algorithm of Chapter 15 (on randomized algorithms), when the input x is a prime number, the algorithm *always* says "prime;" when x is composite, the algorithm *usually* says "composite," but may occasionally say "prime." Using the definitions above, this means that the Solovay–Strassen algorithm is a one-sided error algorithm for the set A of composite numbers. It also is a bounded two-sided error algorithm for $\overline{A}$, the set of prime numbers.

These three kinds of errors suggest three complexity classes:

- PP is the class of languages decided by probabilistic Turing machines of polynomial time complexity with unbounded two-sided error.
- BPP is the class of languages decided by probabilistic Turing machines of polynomial time complexity with bounded two-sided error.
- RP is the class of languages decided by probabilistic Turing machines of polynomial time complexity with one-sided error.

In the literature, RP is also called R.

A probabilistic Turing machine M is a **PP-machine** (respectively, a **BPP-machine,** an **RP-machine**) if M has polynomial time complexity, and M decides with two-sided error (bounded two-sided error, one-sided error).

Through repeated Bernoulli trials, we can make the error probabilities of BPP-machines and RP-machines arbitrarily small, as stated in the following theorem. (Among other things, this theorem implies that $\mathsf{RP} \subseteq \mathsf{BPP}$.)

THEOREM 29.2 *If $L \in \mathsf{BPP}$, then for every polynomial $q(n)$, there exists a BPP-machine M such that $p_M(x) > 1 - 1/2^{q(n)}$ for every $x \in L$, and $p_M(x) < 1/2^{q(n)}$ for every $x \notin L$.*

If $L \in \mathsf{RP}$, then for every polynomial $q(n)$, there exists an RP-machine M such that $p_M(x) > 1 - 1/2^{q(n)}$ for every x in L.

It is important to note just how minuscule the probability of error is (provided that the coin flips are truly random). If the probability of error is less than $1/2^{5000}$, then it is less likely that the algorithm produces an incorrect answer than that the computer will be struck by a meteor. An algorithm whose probability of error is $1/2^{5000}$ is essentially as good as an algorithm that makes no errors. For this reason, many computer scientists consider BPP to be the class of practically feasible computational problems.

Next, we define a class of problems that have probabilistic algorithms that make no errors. Define

- $\mathsf{ZPP} = \mathsf{RP} \cap \text{co-RP}$.

The letter Z in ZPP is for zero probability of error, as we now demonstrate. Suppose $A \in \mathsf{ZPP}$. Here is an algorithm that checks membership in A. Let M be an RP-machine that decides A, and let M' be

an RP-machine that decides $\overline{A}$. For an input string x, alternately run M and M' on x, repeatedly, until a computation path of one machine accepts x. If M accepts x, then accept x; if M' accepts x, then reject x. This algorithm works correctly because when an RP-machine accepts its input, it does not make a mistake. This algorithm might not terminate, but with very high probability, the algorithm terminates after a few iterations.

The next theorem expresses some known relationships between probabilistic complexity classes and other complexity classes, such as classes in the polynomial hierarchy (see Section 29.2).

THEOREM 29.3

(a) $\mathsf{P} \subseteq \mathsf{ZPP} \subseteq \mathsf{RP} \subseteq \mathsf{BPP} \subseteq \mathsf{PP} \subseteq \mathsf{PSPACE}$.
(b) $\mathsf{RP} \subseteq \mathsf{NP} \subseteq \mathsf{PP}$.
(c) $\mathsf{BPP} \subseteq \Sigma_2^P \cap \Pi_2^P$.
(d) $\mathsf{PH} \subseteq \mathsf{P}^{\mathsf{PP}}$.
(e) $\mathsf{TC}^0 \subset \mathsf{PP}$.

(Note that the last inclusion is strict! TC^0 is not known to be different from NP, but it is a proper subset of PP.) Figure 29.2 illustrates many of these relationships. PP is not considered to be a feasible class because it contains NP.

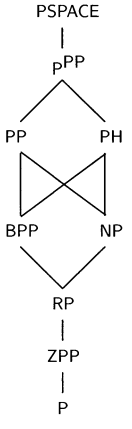


FIGURE 29.2 Probabilistic complexity classes.

Even though it is not clear that there is a good physical source of randomness that can be used to execute probabilistic algorithms and obtain the desired low error bounds, pseudo-random generators are often used and seem to work well. There is currently great interest in de-randomizing probabilistic algorithms, but that topic is beyond the scope of this chapter. There is a simple sense in which a probabilistic algorithm can be de-randomized, however. If an algorithm has very small error probability (in particular, if it has error probability a little less than $1/2^n$), then there is one sequence of coin flips that gives the right answer on all inputs of length n, and this sequence can be hard-wired into the algorithm to yield a deterministic (but *nonuniform*) circuit family. More formally:

THEOREM 29.4 $\mathsf{BPP} \subseteq \mathsf{P/poly}$.

There is another important way in which BPP, RP, and ZPP differ from PP (as well as from NP and

all of the other complexity classes we have discussed thus far): BPP, RP, and ZPP are not known to have any complete languages. Intuitively, BPP is believed to lack complete sets because there is no computable way to weed out those polynomial-time probabilistic Turing machines that are not BPP-machines from those that are. The same goes for RP and ZPP—a more detailed discussion of this point may be found in [6, 45]. To be sure, if these classes equal P then trivially they have complete languages. Recent work [30] proves that a highly plausible hardness assertion for languages in exponential time implies P = BPP.

Log-space analogues of these probabilistic classes have also been studied, of which the most important is RL, defined by probabilistic TMs with one-sided error that run in log space and may use polynomially many random bits in any computation. An important problem in RL that is not known to be in L is that of whether there is a path from node s to node t in an *un*directed graph, or much the same thing, whether an undirected graph is connected.

29.4 Formal Logic and Complexity Classes

There is a surprisingly close connection between important complexity classes and natural notions that arise in the study of formal logic. This connection has led to important applications of complexity theory to logic, and vice-versa. Below, we present some basic notions from formal logic, and then we show some of the connections between logic and complexity theory.

Descriptive complexity refers to the ability to describe and characterize individual problems and whole complexity classes by certain kinds of formulas in formal logic. These descriptions do not depend on an underlying machine model—they are machine-independent. Furthermore, computational problems can be described in terms of their native data structures, rather than under *ad hoc* string encodings.

A **relational structure** consists of a set V (called the *universe*), a tuple $E_1, \ldots, E_k$ of relations on V, and a tuple $c_1, \ldots, c_\ell$ of elements of V ($k, \ell \geq 0$). Its *type* τ is given by the tuple $(a_1, \ldots, a_k)$ of arities of the respective relations, together with ℓ. In this chapter, V is always finite. For example, directed graphs $G = (V, E)$ are relational structures with the one binary relation E, and their type has $k = 1, a_1 = 2$, and $\ell = 0$, the last since there are no distinguished vertices. For another example, instances of the GRAPH ACCESSIBILITY PROBLEM (GAP) from Section 28.5 consist of a directed graph $G = (V, E)$ along with two distinguished vertices $s, t \in V$, so they have $\ell = 2$.

An ordinary binary string x can be regarded as a structure $(V, X, \leq)$, where $\leq$ is a total order on V that sequences the bits, and for all i ($1 \leq i \leq |x|$), $x_i = 1$ if and only if $X(u_i)$ holds. Here u_i is the ith element of V under the total order, and x_i is the ith bit of x. It is often desirable to regard the ordering $\leq$ as fixed, and focus attention on the single unary relation $X(\cdot)$ as the essence of the string.

Systems of Logic

For our purposes, a **system of logic** (or *logic language*) $\mathcal{L}$ consists of the following:

1. A tuple $(\mathbf{E}_1, \ldots, \mathbf{E}_k)$ of *relation symbols,* with corresponding arities $a_1, \ldots, a_k \geq 1$, and a tuple $(\mathbf{c}_1, \ldots, \mathbf{c}_\ell)$ of *constant symbols* ($k, \ell \geq 0$). These symbols constitute the *vocabulary* of $\mathcal{L}$, and can be identified with the corresponding type τ of relational structures.
2. Optionally, a further finite collection of relation and constant symbols whose interpretations are fixed in all universes V under consideration. By default this collection contains the symbol $=$, which is interpreted as the equality relation on V.
3. An unbounded supply of variable symbols $u, v, w, \ldots$ ranging over elements of V, and optionally, an unbounded supply of variable relation symbols $R_1, R_2, R_3, \ldots$, each with an associated arity and ranging over relations on V.
4. A complete set of Boolean connectives, for which we use $\wedge, \vee, \neg, \rightarrow$, and $\leftrightarrow$, and the quantifiers $\forall, \exists$. Additional kinds of operators for building up formulas are discussed later.

The *well-formed formulas* of $\mathcal{L}$, and the *free, bound, positive,* and *negative* occurrences of symbols in a formula, are defined in the usual inductive manner. A *sentence* is a formula ϕ with no free variables. A formula, or a whole system, is called **first-order** if it has no relation *variables* R_i; otherwise it is **second-order.**

Just as machines of a particular type define complexity classes, so also do logical formulas of a particular type define important classes of languages. The most common nomenclature for these classes begins with a prefix such as FO or F_1 for first-order systems, and SO or F_2 for second-order. SO$\exists$ denotes systems whose second-order formulas are restricted to the form $(\exists R_1)(\exists R_2)\ldots(\exists R_k)\psi$ with ψ first-order. After this prefix, in parentheses, we list the vocabulary, and any extra fixed-interpretation symbols or additions to formulas. For instance, SO$\exists$(*Graphs,* $\leq$) stands for the second-order existential theory of graphs whose nodes are labeled and ordered. (The predicate $=$ is always available in the logics we study, and thus it is not explicitly listed with the other fixed-interpretation symbols such as $\leq$.)

The fixed-interpretation symbols deserve special mention. Many authorities treat them as part of the vocabulary. A finite universe V may without loss of generality be identified with the set $\{1, \ldots, n\}$, where $n \in \mathbb{N}$. Important fixed-interpretation symbols for these sets, besides $=$ and $\leq$, are *suc* , $+$, and $*$, respectively standing for the successor, addition, and multiplication relations. (Here $+(i, j, k)$ stands for $i + j = k$, etc.) Insofar as they deal with the numeric coding of V and do not depend on any structures that are being built on V, such fixed-interpretation symbols are commonly called *numerical predicates.*

Languages, Logics, and Complexity Classes

Let us see how a logical formula describes a language, just as a Turing machine or a program does. A formal inductive definition of the following key notion, and much further information on systems of logic, may be found in the standard text [19].

DEFINITION 29.1 Let ϕ be a sentence in a system $\mathcal{L}$ with vocabulary τ. A relational structure $\mathcal{R}$ of type τ *satisfies* (or *models*) ϕ, written $\mathcal{R} \models \phi$, if ϕ becomes a true statement about $\mathcal{R}$ when the elements of $\mathcal{R}$ are substituted for the corresponding vocabulary symbols of ϕ. The **language of** ϕ is $L_\phi = \{\mathcal{R} : \mathcal{R} \models \phi\}$.

We say that ϕ *describes* L_ϕ, or describes the property of belonging to L_ϕ. Finally, given a system $\mathcal{L}$ of vocabulary τ, $\mathcal{L}$ itself stands for the class of structures of type τ that are described by formulas in $\mathcal{L}$. If τ is the vocabulary *Strings* of binary strings, then L_ϕ is a language in the familiar sense of a subset of $\{0, 1\}^*$, and systems $\mathcal{L}$ over τ define ordinary classes of languages. Thus defining sets of structures over τ generalizes the notion of defining languages over an alphabet.

For example, the formula $(\forall u)X(u)$ over binary strings describes the language 1^*, while $(\forall v, w)[v \neq w \leftrightarrow E(v, w)]$ defines complete (loop-free) graphs. The formula

$$Undir = (\forall v, w)[E(v, w) \rightarrow E(w, v)] \wedge (\forall u)\neg E(u, u)$$

describes the property of being an undirected simple graph, treating an undirected edge as a pair of directed edges, and ruling out "self-loops." Given unary relation symbols $X_1, \ldots, X_k$, the formula

$$Uniq_{X_1,\ldots,X_k} = (\forall v)\left[\bigvee_{1\leq i\leq k} X_i(v) \wedge \bigwedge_{1\leq i<j\leq k} \neg\left(X_i(v) \wedge X_j(v)\right)\right]$$

expresses that every element v is assigned exactly one i such that $X_i(v)$ holds. Given an arbitrary finite alphabet $\Sigma = \{c_1, \ldots, c_k\}$, the vocabulary $\{X_1, \ldots, X_k\}$, together with this formula, enables us to define languages of strings over Σ. (Since the presence of *Uniq* does not affect any of the syntactic characterizations that follow, we may now regard *Strings* as a vocabulary over any Σ.) Given a unary relation symbol R and

the numerical predicate *suc* on V, the formula

$$Alts_R = (\exists s, t)(\forall u, v)[\neg Suc(u, s) \wedge \neg Suc(t, u) \wedge R(s) \wedge \neg R(t) \wedge (Suc(u, v) \rightarrow (R(u) \leftrightarrow \neg R(v))]$$

says that R is true of the first element s, false of the last element t, and alternates true and false in-between. This requires $|V|$ to be even. The following examples are used again below.

(1) The regular language $(10)^*$ is described by the first-order formula $\phi_1 = Alts_X$.

(2) $(11)^*$ is described by the second-order formula $\phi_2 = (\exists R)(\forall u)[X(u) \wedge Alts_R]$.

(3) GRAPH THREE-COLORABILITY:

$$\phi_3 = Undir \wedge (\exists R_1, R_2, R_3)\left[Uniq_{R_1,R_2,R_3} \wedge (\forall v, w)(E(v, w) \rightarrow \bigvee_{1 \le i \le 3} R_i(v) \wedge \neg R_i(w))\right].$$

(4) GAP (i.e., s-t connectivity for directed graphs):

$$\phi_4 = (\forall R)\neg(\forall u, v)[R(s) \wedge \neg R(t) \wedge (R(u) \wedge E(u, v) \rightarrow R(v))]\,.$$

Formula ϕ_4 says that there is no set $R \subseteq V$ that is closed under the edge relation and contains s but doesn't contain t, and this is equivalent to the existence of a path from s to t. Much trickier is the fact that deleting "$Uniq_{R_1,R_2,R_3}$" from ϕ_3 leaves a formula that still defines exactly the set of undirected 3-colorable graphs. This fact hints at the delicacy of complexity issues in logic.

Much of this study originated in research on database systems, because data base query languages correspond to logics. First-order logic is notoriously limited in expressive power, and this limitation has motivated the study of extensions of first-order logic, such as the following *first-order operators.*

DEFINITION 29.2

(a) *Transitive closure* (TC): Let ϕ be a formula in which the first-order variables $u_1, \ldots, u_k$ and $v_1, \ldots, v_k$ occur freely, and regard ϕ as implicitly defining a binary relation S on V^k. That is, S is the set of pairs $(\vec{u}, \vec{v})$ such that $\phi(\vec{u}, \vec{v})$ holds. Then $\mathrm{TC}_{(u_1,\ldots,u_k,v_1,\ldots,v_k)}\,\phi$ is a formula, and its semantics is the reflexive-transitive closure of S.

(b) *Least fixed point* (LFP): Let ϕ be a formula with free first-order variables $u_1, \ldots, u_k$ and a free k-ary relation symbol R that occurs only positively in ϕ. In this case, for any relational structure $\mathcal{R}$ and $S \subseteq V^k$, the mapping $f_\phi(S) = \{(e_1, \ldots, e_k) : \mathcal{R} \models \phi(S, e_1, \ldots, e_k)\}$ is monotone. That is, if $S \subseteq T$, then for every tuple of domain elements $(e_1, \ldots, e_k)$, if $\phi(R, u_1, \ldots, u_k)$ evaluates to `true` when R is set to S and each u_i is set to e_i, then ϕ also evaluates to `true` when R is set to T, because R appears positively. Thus the mapping f_ϕ has a least fixed point in V^k. Then $\mathrm{LFP}_{(R,u_1,\ldots,u_k)}\,\phi$ is a formula, and its semantics is the least fixed point of f_ϕ, i.e., the smallest S such that $f_\phi(S) = S$.

(c) *Partial fixed point* (PFP): Even if f_ϕ above is not monotone, $\mathrm{PFP}_{(R,u_1,\ldots,u_k)}\,\phi$ is a formula whose semantics is the first fixed point found in the sequence $f_\phi(\emptyset)$, $f_\phi(f_\phi(\emptyset))$, ..., if it exists, $\emptyset$ otherwise.

The first-order variables $u_1, \ldots, u_k$ remain free in these formulas. The relation symbol R is bound in $\mathrm{LFP}_{(R,u_1,\ldots,u_k)}\,\phi$, but since this formula is fixing R uniquely rather than quantifying over it, the formula $\mathrm{LFP}_{(R,u_1,\ldots,u_k)}\,\phi$ is still regarded as first-order (provided ϕ is first-order).

A somewhat less natural but still useful operation is the "deterministic transitive closure" operator. We write "DTC" for the restriction of (a) above to cases where the implicitly defined binary relation S is a partial function. The DTC restriction is enforcible syntactically by replacing any (sub)-formula ϕ to which

TC is applied by $\phi'' = \phi \wedge (\forall w_1, \ldots, w_k)[\phi' \rightarrow \wedge_{i=1}^{k} v_i = w_i]$, where ϕ' is the result of replacing each v_i in ϕ by w_i, $1 \leq i \leq k$.

For example, s-t connectivity is definable by the FO(TC) and FO(LFP) formulas

$$\begin{aligned} \phi_4' &= (\exists u, v)\left[u = s \wedge v = t \wedge \mathrm{TC}_{(u,v)}\, E(u, v)\right], \\ \phi_4'' &= (\exists u, v)\left[u = s \wedge v = t \wedge \mathrm{LFP}_{(R,u,v)}\, \psi\right], \end{aligned}$$

where $\psi = (u = v \vee E(u, v) \vee (\exists w)[R(u, w) \wedge R(w, v)])$. To understand how ϕ_4'' works, starting with S as the empty binary relation and substituting the current S for R at each turn, the first iteration yields $S = \{(u, v) : u = v \vee E(u, v)\}$, the second iteration gives pairs of vertices connected by a path of length at most 2, then 4, ..., and the fixed-point is the reflexive-transitive closure E^* of E. Then ϕ_4'' is read as if it were $(\exists u, v)(u = s \wedge v = t \wedge E^*(u, v))$, or more simply, as if it were $E^*(s, t)$.

Note however, that writing DTC ... in place of TC ... in ϕ_4' changes the property defined by restricting it to directed graphs in which each non-sink vertex has out-degree 1. It is not known whether s-t connectivity can be expressed using the DTC operator. This question is equivalent to whether L = NL.

Logical Characterizations of Complexity Classes

As discussed by [21], there is a uniform encoding method *Enc* such that for any vocabulary τ and (finite) relational structure $\mathcal{R}$ of type τ, *Enc* ($\mathcal{R}$) is a standard string encoding of $\mathcal{R}$. For instance with $\tau = \mathit{Graphs}$, an n-vertex graph becomes the size-n^2 binary string that lists the entries of its adjacency matrix in row-major order. Thus one can say that a language L_ϕ over any vocabulary belongs to a complexity class $\mathcal{C}$ if the string language $\mathit{Enc}\,(L_\phi) = \{\mathit{Enc}\,(\mathcal{R}) : \mathcal{R} \models \phi\}$ is in $\mathcal{C}$.

The following theorems of the form "$\mathcal{C} = \mathcal{L}$" all hold in the following strong sense: for every vocabulary τ and $\mathcal{L}(\tau)$-formula ϕ, $\mathit{Enc}\,(L_\phi) \in \mathcal{C}$; and for every language $A \in \mathcal{C}$, there is a $\mathcal{L}(\mathit{Strings})$-formula ϕ such that $L_\phi = A$. Although going to strings via *Enc* may seem counter to the motivation expressed in the first paragraph of this whole section, the generality and strength of these results has a powerful impact in the desired direction: they define the right notion of complexity class $\mathcal{C}$ for any vocabulary τ. Hence we omit the vocabulary τ in the following statements.

THEOREM 29.5

(a) PSPACE = FO(PFP, $\leq$).
(b) PH = SO.
(c) **(Fagin's Theorem)** NP = SO$\exists$.
(d) P = FO(LFP, $\leq$).
(e) NL = FO(TC, $\leq$).
(f) L = FO(DTC, $\leq$).
(g) AC^0 = FO(+, $*$).

One other result should be mentioned with the above. Define the *spectrum* of a formula ϕ by $S_\phi = \{n :$ for some $\mathcal{R}$ with n elements, $\mathcal{R} \models \phi\}$. Jones and Selman [31] proved that a language A belongs to NE if and only if there is a vocabulary τ and a sentence $\phi \in \mathsf{FO}(\tau)$ such that $A = S_\phi$ (identifying numbers and strings). Thus spectra characterize NE.

The ordering $\leq$ is needed in results (a), (d), (e), and (f). Chandra and Harel [15] proved that FO(LFP) without $\leq$ cannot even define (11)* (and their proof works also for FO(PFP)). Put another way, without an (ad-hoc) ordering on the full database, one cannot express queries of the kind "Is the number of widgets in Toledo even?" even in the powerful system of first-order logic with PFP. Note that, as a consequence of what we know about complexity classes, it follows that FO(PFP, $\leq$) is more expressive

than FO(TC, $\leq$). This result is an example of an application of complexity theory to logic. In contrast, when the ordering is not present, it is much easier to show directly that FO(PFP) is more powerful than FO(TC) than to use the tools of complexity theory. Furthermore, the hypotheses FO(LFP) = FO(PFP) and FO(LFP, $\leq$) = FO(PFP, $\leq$) are *both* equivalent to P = PSPACE [2]. This shows how logic can apply to complexity theory.

A Short Digression: Logic and Formal Languages

There are two more logical characterizations that seem at first to have little to do with complexity theory. Characterizations such as these have been important in circuit complexity, but those considerations are beyond the scope of this chapter.

Let SF stand for the class of *star-free regular languages,* which are defined by regular expressions without Kleene stars, but with $\emptyset$ as an atom and complementation ($\sim$) as an operator. For example, $(10)^* \in$ SF via the equivalent expression $\sim[(\sim\emptyset)(00+11)(\sim\emptyset)+0(\sim\emptyset)+(\sim\emptyset)1]$.

A formula is *monadic* if each of its relation symbols is unary. A second-order system is *monadic* if every relation variable is unary. Let mSO denote the monadic second-order formulas. The formula ϕ_2 above defines $(11)^*$ in mSO$\exists$(*suc*). The following results are specific to the vocabulary of strings.

THEOREM 29.6

(a) REG = mSO(*Strings,* $\leq$) = mSO$\exists$(*Strings,suc*).
(b) SF = FO(*Strings,* $\leq$).

Theorem 29.6, combined with Theorem 29.5 (b) and (c), shows that SO is much more expressive than mSO, and SO$\exists(\leq)$ is similarly more expressive than mSO$\exists(\leq)$. A seemingly smaller change to mSO$\exists$ also results in a leap of expressiveness from the regular languages to the level of NP. Lynch [39] showed that if we consider mSO$\exists(+)$ instead of mSO$\exists(\leq)$ (for strings), then the resulting class contains NTIME$[n]$, and hence contains many NP-complete languages, such as GRAPH THREE-COLORABILITY.

29.5 Interactive Models and Complexity Classes

Interactive Proofs

In Section 27.2, we characterized NP as the set of languages whose membership proofs can be checked quickly, by a deterministic Turing machine M of polynomial time complexity. A different notion of proof involves interaction between two parties, a prover P and a verifier V, who exchange messages. In an **interactive proof system,** the prover is an all-powerful machine, with unlimited computational resources, analogous to a teacher. The verifier is a computationally limited machine, analogous to a student. Interactive proof systems are also called "Arthur–Merlin games:" the wizard Merlin corresponds to P, and the impatient Arthur corresponds to V.

Formally, an **interactive proof system** comprises the following:

- A read-only input tape on which an input string x is written.
- A *prover* P, whose behavior is not restricted.
- A *verifier* V, which is a probabilistic Turing machine augmented with the capability to send and receive messages. The running time of V is bounded by a polynomial in $|x|$.
- A tape on which V writes messages to send to P, and a tape on which P writes messages to send to V. The length of every message is bounded by a polynomial in $|x|$.

A computation of an interactive proof system (P, V) proceeds in rounds, as follows. For $j = 1, 2, \ldots$, in round j, V performs some steps, writes a message m_j, and temporarily stops. Then P reads m_j and responds with a message m'_j, which V reads in round $j + 1$. An interactive proof system (P, V) **accepts** an input string x if the probability of acceptance by V satisfies $p_V(x) > 1/2$.

In an interactive proof system, a prover can convince the verifier about the truth of a statement without exhibiting an entire proof, as the following example illustrates.

EXAMPLE 29.1:

Consider the graph non-isomorphism problem: the input consists of two graphs G and H, and the decision is "yes" if and only if G is not isomorphic to H. Although there is a short proof that two graphs *are* isomorphic (namely: the proof consists of the isomorphism mapping G onto H), nobody has found a general way of proving that two graphs are *not* isomorphic that is significantly shorter than listing all $n!$ permutations and showing that each fails to be an isomorphism. (That is, the graph nonisomorphism problem is in co-NP, but is not known to be in NP.) In contrast, the verifier V in an interactive proof system is able to take statistical evidence into account, and determine "beyond all reasonable doubt" that two graphs are nonisomorphic, using the following protocol.

In each round, V randomly chooses either G or H with equal probability; if V chooses G, then V computes a random permutation G' of G, presents G' to P, and asks P whether G' came from G or from H (and similarly if V chooses H). If P gave an erroneous answer on the first round, and G is isomorphic to H, then after k subsequent rounds, the probability that P answers all the subsequent queries correctly is $1/2^k$. (To see this, it is important to understand that the prover P does not see the coins that V flips in making its random choices; P sees only the graphs G' and H' that V sends as messages.) V accepts the interaction with P as "proof" that G and H are nonisomorphic if P is able to pick the correct graph for 100 consecutive rounds. Note that V has ample grounds to accept this as a convincing demonstration: if the graphs are indeed isomorphic, the prover P would have to have an incredible streak of luck to fool V.

The complexity class IP comprises the languages A for which there exists a verifier V and an ϵ such that

- There exists a prover $\hat{P}$ such that for all x in A, the interactive proof system $(\hat{P}, V)$ accepts x with probability greater than $1/2 + \epsilon$; and
- For every prover P and every $x \notin A$, the interactive proof system (P, V) rejects x with probability greater than $1/2 + \epsilon$.

By substituting random choices for existential choices in the proof that $\mathsf{ATIME}(t) \subseteq \mathsf{DSPACE}(t)$ (Theorem 27.8(b) in Chapter 27), it is straightforward to show that $\mathsf{IP} \subseteq \mathsf{PSPACE}$. It was originally believed likely that IP was a small subclass of PSPACE. Evidence supporting this belief was the construction by Fortnow and Sipser [24] of an oracle language B for which $\text{co-NP}^B - \mathsf{IP}^B \neq \emptyset$, so that IP^B is strictly included in PSPACE^B. Using a proof technique that does not relativize, however, Shamir [44] (building on the work of Lund et al. [37]) proved that in fact, IP and PSPACE are the same class.

THEOREM 29.7 $\mathsf{IP} = \mathsf{PSPACE}$.

If NP is a proper subset of PSPACE, as is widely believed, then Theorem 29.7 says that interactive proof systems can decide a larger class of languages than NP.

Probabilistically Checkable Proofs

In an interactive proof system, the verifier does not need a complete conventional proof to become convinced about the membership of a word in a language, but uses random choices to query parts of

a proof that the prover may know. This interpretation inspired another notion of "proof": a proof consists of a (potentially) large amount of information that the verifier need only inspect in a few places in order to become convinced. The following definition makes this idea more precise.

A language L has a **probabilistically checkable proof** if there exists an oracle BPP-machine M such that

- For all $x \in L$, there exists an oracle language B_x such that M^{B_x} accepts x.
- For all $x \notin L$, and for every language B, machine M^B rejects x.

Intuitively, the oracle language B_x represents a proof of membership of x in L. Notice that B_x can be finite since the length of each possible query during a computation of M^{B_x} on x is bounded by the running time of M. The oracle language takes the role of the prover in an interactive proof system—but in contrast to an interactive proof system, the prover cannot change strategy adaptively in response to the questions that the verifier poses. This change results in a potentially stronger system, since a machine M that has bounded error probability relative to all languages B might not have bounded error probability relative to some adaptive prover. Although this change to the proof system framework may seem modest, it leads to a characterization of a class that seems to be much larger than PSPACE.

THEOREM 29.8 *A has a probabilistically checkable proof if and only if* $A \in \mathsf{NEXP}$.

Although the notion of probabilistically checkable proofs seems to lead us away from feasible complexity classes, by considering natural restrictions on how the proof is accessed, we can obtain important insights into familiar complexity classes.

Let $\mathsf{PCP}(r(n), q(n))$ denote the class of languages with probabilistically checkable proofs in which the probabilistic oracle Turing machine M makes $O(r(n))$ random binary choices, and queries its oracle $O(q(n))$ times. (For this definition, we assume that M has either one or two choices for each step.) It follows from the definitions that $\mathsf{BPP} = \mathsf{PCP}(n^{O(1)}, 0)$, and $\mathsf{NP} = \mathsf{PCP}(0, n^{O(1)})$.

THEOREM 29.9 $\mathsf{NP} = \mathsf{PCP}(\log n, 1)$.

Theorem 29.9 asserts that for every language L in NP, a proof that $x \in L$ can be encoded so that the verifier can be convinced of the correctness of the proof (or detect an incorrect proof) by using only $O(\log n)$ random choices, and inspecting only a *constant* number of bits of the proof!

This surprising characterization of NP has important applications to the complexity of finding approximate solutions to **optimization problems,** as discussed in the next section.

29.6 Classifying the Complexity of Functions

Up to now, we have considered only the complexity of decision problems. (Recall that a decision problem is a problem in which, for every input, the output is either "yes" or "no.") Most of the functions that we actually compute are functions that produce more than one bit of output. For example, instead of merely deciding whether a graph has a clique of size m, we often want to *find* a clique. Problems in NP are naturally associated with this kind of search problem.

Of course, any function f can be analyzed in terms of a decision problem in a straightforward way by considering the decision problem A_f that takes as input x and i, and answers "yes" if the ith bit of $f(x)$ is 1. But there are other ways of formulating functions as decision problems, and sometimes it is instructive to study the complexity of functions directly instead of their associated decision problems. In this section and the sections that follow, we will discuss some of the more useful classifications.

The most important class of functions is the class that we can compute quickly.

- FP is the set of functions computable in polynomial time by deterministic Turing machines.

In an analogous way, we define FL, FNC^k, etc., to be the set of functions computable by deterministic log-space machines, by NC^k circuits, etc. We also define FPSPACE to be the class of functions f computable by deterministic machines in polynomial space, such that also $|f(x)|$ is bounded by a polynomial in $|x|$. This restriction is essential because a machine that uses polynomial space could run for an exponential number of steps, producing an exponentially long output.

To study functions that appear to be difficult to compute, we again use the notions of reducibility and completeness. Analogous to Cook reducibility to oracle *languages,* we consider Cook reducibility to a *function* given as an oracle. For a function f whose length $|f(x)|$ is bounded by a polynomial in $|x|$, we say that a language A is Cook reducible to f if there is a polynomial-time oracle Turing machine M that accepts A, where the oracle is accessed as follows: M writes a string y on the query tape, and in the next step y is replaced by $f(y)$. As usual, we let P^f and FP^f denote the class of languages and functions computable in polynomial time with oracle f, respectively.

Let $\mathcal{C}$ be a class of functions. When $\mathcal{C}$ is at least as big as FP, then we will use Cook reducibility to define completeness. That is, a function f is $\mathcal{C}$-complete, if f is in $\mathcal{C}$ and $\mathcal{C} \subseteq \mathsf{FP}^f$. When we are discussing smaller classes $\mathcal{C}$ (where polynomial-time is too powerful to give a meaningful notion of reducibility), then when we say that a function f is $\mathcal{C}$-complete, it refers to completeness under AC^0-Turing reducibility, which was defined in Section 28.6. In this chapter, we consider only these two variants of Turing reducibility. There are many other ways to reduce one function to another, just as there are many kinds of reductions between languages.

We use these notions to study optimization problems in this section and counting problems in Section 29.7.

Optimization Classes

Given an optimization (minimization) problem, we most often study the following associated decision problem:

> "Is the optimal value at most k"?

Alternatively, we could formulate the decision problem as the following:

> "Is the optimal value exactly k?"

For example, consider the TRAVELING SALESPERSON problem (TSP) again. TSP asks whether the length of the optimal tour is at most d_0. Define EXACT TSP to be the decision problem that asks whether the length of the optimal tour is exactly d_0. It is not clear that EXACT TSP is in NP or in co-NP, but EXACT TSP *can* be expressed as the intersection of TSP and its complement $\overline{\text{TSP}}$: the length of the optimal tour is d_0 if there is a tour whose length is at most d_0, and no tour whose length is at most $d_0 - 1$. Similar remarks apply to the optimization problem MAX CLIQUE: given an undirected graph G, find the maximum size of a clique in G.

Exact versions of many optimization problems can be expressed as the intersection of a language in NP and a language in co-NP. This observation motivates the definition of a new complexity class:

- D^P is the class of languages A such that $A = A_1 \cap A_2$ for some languages A_1 in NP and A_2 in co-NP.

The letter D in D^P means difference: $A \in \mathsf{D}^P$ if and only if A is the difference of two languages, i.e., $A = A_1 - A_3$ for some languages A_1 and A_3 in NP.

Not only is EXACT TSP in D^P, but also EXACT TSP is D^P-complete. Exact versions of many other NP-complete problems, including CLIQUE, are also D^P-complete [41].

Although it is not known whether D^P is contained in NP, it is straightforward to prove that

$$\mathsf{NP} \subseteq \mathsf{D}^P \subseteq \mathsf{P}^{\mathsf{NP}} \subseteq \Sigma_2^P \cap \Pi_2^P .$$

Thus, D^P lies between the first two levels of the polynomial hierarchy.

We have characterized the complexity of computing the optimal value of an instance of an optimization problem, but we have not yet characterized the complexity of computing the optimal solution itself. An optimization algorithm produces not only a "yes" or "no" answer, but also, when feasible solutions exist, an optimal solution.

First, for a maximization problem, suppose that we have a subroutine that solves the decision problem "Is the optimal value at least k?" Sometimes, with repeated calls to the subroutine, we can construct an optimal solution. For example, suppose subroutine S solves the CLIQUE problem; for an input graph G and integer k, the subroutine outputs "yes" if G has a clique of k (or more) vertices. To construct the largest clique in an input graph, first, determine the size K of the largest clique by binary search on $\{1, \ldots, n\}$ with $\log_2 n$ calls to S. Next, for each vertex v, in sequence, determine whether deleting v produces a graph whose largest clique has size K by calling S. If so, then delete v and continue with the remaining graph. If not, then look for a clique of size $K - 1$ among the neighbors of v.

The method outlined in the last paragraph uses S in the same way as an oracle Turing machine queries an oracle language in NP. With this observation, we define the following classes:

- $\mathsf{FP}^{\mathsf{NP}}$ is the set of functions computable in polynomial time by deterministic oracle Turing machines with oracle languages in NP.
- $\mathsf{FP}^{\mathsf{NP}[\log n]}$ is the set of functions computable in polynomial time by deterministic oracle Turing machines with oracle languages in NP that make $O(\log n)$ queries during computations on inputs of length n

$\mathsf{FP}^{\mathsf{NP}}$ and $\mathsf{FP}^{\mathsf{NP}[\log n]}$ contain many well-studied optimization problems [33]. The problem of producing the optimal tour in the TRAVELING SALESPERSON problem is $\mathsf{FP}^{\mathsf{NP}}$-complete. The problem of determining the size of the largest clique subgraph in a graph is $\mathsf{FP}^{\mathsf{NP}[\log n]}$-complete.

Approximability and Complexity

As discussed in Chapter 34 because polynomial-time algorithms for NP-complete optimization problems are unlikely to exist, we ask whether a polynomial-time algorithm can produce a feasible solution that is *close* to optimal.

Fix an optimization problem Π with a positive integer-valued objective function g. For each problem instance x, let $\mathsf{OPT}(x)$ be the optimal value, that is, $g(z)$, where z is a feasible solution to x that achieves the best possible value of g. Let M be a deterministic Turing machine that on input x produces as output a feasible solution $M(x)$ for Π. We say M is an **ϵ-approximation algorithm** if for all x,

$$\frac{|g(M(x)) - \mathsf{OPT}(x)|}{\max\{g(M(x)), \mathsf{OPT}(x)\}} \leq \epsilon .$$

(This definition handles both minimization and maximization problems.) The problem Π has a **polynomial-time approximation scheme** if for every fixed ϵ, there is a polynomial-time ϵ-approximation algorithm. Although the running time is polynomial in $|x|$, the time could be exponential in $1/\epsilon$.

Several NP-complete problems, including KNAPSACK, have polynomial-time approximation schemes. It is natural to ask whether *all* NP-complete optimization problems have polynomial-time approximation schemes. We define an important class of optimization problems, $\mathsf{MAX\text{-}SNP}$, whose complete problems apparently do not.

First, we define a reducibility between optimization problems that preserves the quality of solutions. Let Π_1 and Π_2 be optimization problems with objective functions g_1 and g_2, respectively. An **L-reduction** from Π_1 to Π_2 is defined by a pair of polynomial-time computable functions f and f' that satisfy the following properties:

1. If x is an instance of Π_1 with optimal value $\mathsf{OPT}(x)$, then $f(x)$ is an instance of Π_2 whose optimal value satisfies $\mathsf{OPT}(f(x)) \leq c \cdot \mathsf{OPT}(x)$ for some constant c.
2. If z is a feasible solution of $f(x)$, then $f'(z)$ is a feasible solution of x, such that

$$\left|\mathsf{OPT}(x) - g_1(f'(z))\right| \leq c' \left|\mathsf{OPT}(f(x)) - g_2(z)\right|$$

   for some constant c'.

The second property implies that if z is an optimal solution to $f(x)$, then $f'(z)$ is an optimal solution to x. From the definitions, it follows that if there is an L-reduction from Π_1 to Π_2, and there is a polynomial-time approximation scheme for Π_2, then there is a polynomial-time approximation scheme for Π_1.

To define MAX-SNP, it will help to recall the characterization of NP as SO$\exists$ in Section 29.4. This characterization says that for any A in NP, there is a first-order formula ψ such that $x \in A$ if and only if

$$\exists S_1 \ldots \exists S_l \; \psi(x, S_1, \ldots, S_l) \; .$$

For many important NP-complete problems, it is sufficient to consider having only a single second-order variable S, and to consider formulas ψ having only universal quantifiers. Thus, for such a language A we have a quantifier-free formula ϕ such that $x \in A$ if and only if

$$\exists S \forall u_1 \ldots \forall u_k \; \phi \, (S, u_1, \ldots, u_k) \; .$$

Now define MAX-SNP$_0$ to be the class of optimization problems mapping input x to

$$\max_S \left|\{(y_1, \ldots, y_k) : \phi \, (S, y_1, \ldots, y_k)\}\right| \; .$$

For example, we can express in this form the MAX CUT problem, the problem of finding the largest cut in an input graph $G = (V, E)$ with vertex set V and edge set E. A set of vertices S is the optimal solution if it maximizes

$$\left|\{(v, w) : E(v, w) \wedge S(v) \wedge \neg S(w)\}\right| \; .$$

That is, the optimal solution S maximizes the number of edges (v, w) between vertices v in S and vertices w in $V - S$.

Define MAX-SNP to be the class of all optimization problems that are L-reducible to a problem in MAX-SNP$_0$. MAX-SNP contains many natural optimization problems. MAX CUT is MAX-SNP-complete, and MAX CLIQUE is MAX-SNP-hard, under L-reductions.

A surprising connection between the existence of probabilistically checkable proofs (Section 29.5) and the existence of approximation algorithms comes out in the next major theorem.

THEOREM 29.10 *If there is a polynomial-time approximation scheme for some* MAX-SNP*-hard problem, then* P = NP.

In particular, unless P = NP, there is no polynomial-time approximation scheme for MAX CUT or MAX CLIQUE. To prove this theorem, all we need to do is show its statement for a particular problem that is MAX-SNP-complete under L-reductions. However, we prefer to show the *idea* of the proof for the MAX CLIQUE problem, which although MAX-SNP-hard is not known to belong to MAX-SNP. It gives a strikingly different kind of reduction from an arbitrary language A in NP to CLIQUE over the

reduction from A to SAT to CLIQUE in Section 28.4, and its discovery by Feige et al. [22] [23] stimulated the whole area.

PROOF Let $A \in \mathsf{NP}$. By Theorem 29.9, namely $\mathsf{NP} = \mathsf{PCP}[O(\log n), O(1)]$, there is a probabilistic oracle Turing machine M constrained to use $r(n) = O(\log n)$ random bits and make at most a constant number ℓ of queries in any computation path, such that

- For all $x \in A$, there exists an oracle language B_x such that $\text{Prob}_{s \in \{0,1\}^{r(n)}}[M^{B_x}(x, s) = 1] > 3/4$;
- For all $x \notin A$, and for every language B, $\text{Prob}_{s \in \{0,1\}^{r(n)}}[M^{B}(x, s) = 1] < 1/4$.

Now define a *transcript* of M on input x to consist of a string $s \in \{0, 1\}^{r(n)}$ together with a sequence of ℓ pairs (w_i, a_i), where w_i is an oracle query and $a_i \in \{0, 1\}$ is a possible yes/no answer. In addition, a transcript must be *valid:* for all i, $0 \leq i < \ell$, on input x with random bits s, having made queries $w_1, \ldots, w_i$ to its (unspecified) oracle and received answers $a_1, \ldots, a_i$, machine M writes w_{i+1} as its next query string. Thus a transcript provides enough information to determine a full computation path of M on input x, and the transcript is *accepting* if and only if this computation path accepts. Finally, call two transcripts *consistent* if whenever a string w appears as "w_i" in one transcript and "w_j" in the other, the corresponding answer bits a_i and a_j are the same.

Construction: Let G_x be the undirected graph whose node set V_x is the set of all accepting transcripts, and whose edges connect pairs of transcripts that are consistent.

Complexity: Since $r(n) + \ell = O(\log n)$, there are only polynomially many transcripts, and since consistency is easy to check, G_x is constructed in polynomial time.

Correctness: If $x \in A$, then take the oracle B_x specified above and let C be the set of accepting transcripts whose answer bits are given by B_x. These transcripts are consistent with each other, and there are at least $(3/4)2^{r(n)}$ such accepting transcripts, so C forms a clique of size at least $(3/4)2^{r(n)}$ in G_x. Now suppose $x \notin A$, and suppose C' is a clique of size greater than $(1/4)2^{r(n)}$ in G_x. Because the transcripts in C' are mutually consistent, there exists a single oracle B that produces all the answer bits to queries in transcripts in C'. But then $\text{Prob}_s[M^B(x, s) = 1] > 1/4$, contradicting the PCP condition on M.

Thus we have proved the statement of the theorem for MAX CLIQUE. The proof of the general statement is similar.

Note that the cases $x \in A$ and $x \notin A$ in this proof lead to a "(3/4,1/4) gap" in the maximum clique size ω of G_x. If there were a polynomial-time algorithm guaranteed to determine ω within a factor better than 3, then this algorithm could tell the "3/4" case apart from the "1/4" case, and hence decide whether $x \in A$. Since G_x can be constructed in polynomial time (in particular, G_x has size at most $2^{r(n)+\ell} = n^{O(1)}$), $\mathsf{P} = \mathsf{NP}$ would follow. Hence we can say that CLIQUE is NP-*hard to approximate within a factor of 3.* A long sequence of improvements to this basic construction has pushed the hardness-of-approximation not only to any fixed constant factor, but also to factors that increase with n. Moreover, approximation-preserving reductions have extended this kind of hardness result to many other optimization problems.

29.7 Counting

Two other important classes of functions deserve special mention:

- $\#\mathsf{P}$ is the class of functions f such that there exists a nondeterministic polynomial-time Turing machine M with the property that $f(x)$ is the number of accepting computation paths of M on input x.
- $\#\mathsf{L}$ is the class of functions f such that there exists a nondeterministic log-space Turing machine M with the property that $f(x)$ is the number of accepting computation paths of M on input x.

Some functions in #P are clearly at least as difficult to compute as some NP-complete problems are to decide. For instance, consider the following problem.

NUMBER OF SATISFYING ASSIGNMENTS TO A 3CNF FORMULA (#3CNF)
Instance: A Boolean formula in conjunctive normal form with at most three variables per clause.
Output: The number of distinct assignments to the variables that cause the formula to evaluate to `true`.

Note that #3CNF is in #P, and note also that the NP-complete problem of determining whether $x \in 3\mathrm{SAT}$ is merely the question of whether $\#3\mathsf{CNF}(x) = 0$.

In apparent contrast to #P, all functions in #L can be computed by NC circuits.

THEOREM 29.11 *Relationships between counting classes.*

- $\mathsf{FP} \subseteq \#\mathsf{P} \subseteq \mathsf{FPSPACE}$
- $\mathsf{P}^{\mathsf{PP}} = \mathsf{P}^{\#\mathsf{P}}$ *(and thus also* $\mathsf{FP}^{\mathsf{NP}} \subseteq \mathsf{FP}^{\#\mathsf{P}}$*)*
- $\mathsf{FL} \subseteq \#\mathsf{L} \subseteq \mathsf{FNC}^2$.

It is not surprising that #P and #L capture the complexity of various functions that involve counting, but as the following examples illustrate, it sometimes is surprising which functions are difficult to compute.

The proof of Cook's Theorem that appears in Chapter 28 also proves that #3CNF is complete for #P, because it shows that for every nondeterministic polynomial-time machine M and every input x, one can efficiently construct a formula with the property that each accepting computation of M on input x corresponds to a distinct satisfying assignment, and vice versa. Thus the number of satisfying assignments equals the number of accepting computation paths. A reduction with this property is called *parsimonious.*

Most NP-complete languages that one encounters in practice are known to be complete under parsimonious reductions. (The reader may wish to check which of the reductions presented in Chapter 28 are parsimonious.) For any such complete language, it is clear how to define a corresponding complete function in #P.

Similarly, for the GRAPH ACCESSIBILITY PROBLEM (GAP), which is complete for NL, we can define the function that counts the number of paths from the start vertex s to the terminal vertex t. For reasons that will become clear soon, we consider two versions of this problem: one for general directed graphs, and one for directed acyclic graphs. (The restriction of GAP to acyclic graphs remains NL-complete.)

NUMBER OF PATHS IN A GRAPH (#PATHS)
Instance: A directed graph on n vertices, with two vertices s and t.
Output: The number of simple paths from s to t.
(A path is a *simple path* if it visits no vertex more than once.)

NUMBER OF PATHS IN A DIRECTED ACYCLIC GRAPH (#DAG-PATHS)
Instance: A directed acyclic graph on n vertices, with two vertices s and t.
Output: The number of paths from s to t.
(In an acyclic graph, all paths are simple.)

As one might expect, the problem #DAG-PATHS is complete for #L, but it may come as a surprise that #PATHS is complete for #P [52]! That is, although it is easy to decide whether there is a path between two vertices, it seems quite difficult to count the number of distinct paths, unless the underlying graph is acyclic.

As another example of this phenomenon, consider the problem 2SAT, which is the same as 3SAT except that each clause has at most two literals. 2SAT is complete for NL, but the problem of counting the number of satisfying assignments for these formulas is complete for #P.

A striking illustration of the relationship between #P and #L is provided by the following two important problems from linear algebra.

DETERMINANT
Instance: An integer matrix.
Output: The determinant of the matrix.

Recall that the determinant of a matrix M with entries $M_{i,j}$ is given by

$$\sum_{\pi} \text{sign}(\pi) \prod_{i=1}^{n} M_{i,\pi(i)} ,$$

where the sum is over all permutations π on $\{1, \ldots, n\}$, and $\text{sign}(\pi)$ is -1 if π can be written as the composition of an odd number of transpositions, and $\text{sign}(\pi)$ is 1 otherwise.

PERMANENT
Instance: An integer matrix.
Output: The permanent of the matrix. The permanent of a matrix is given by

$$\sum_{\pi} \prod_{i=1}^{n} M_{i,\pi(i)} .$$

The reader is probably familiar with the determinant function, which can be computed efficiently by Gaussian elimination. The permanent may be less familiar, although its definition is formally simpler. Nobody has ever found an efficient way to compute the permanent, however.

We need to introduce slight modification of our function classes to classify these problems, however, because #L and #P consist of functions that take only non-negative values, whereas both the permanent and determinant can be negative.

Define GapL to be the class of functions that can be expressed as the difference of two #L functions, and define GapP to be the difference of two #P functions.

THEOREM 29.12

(a) PERMANENT *is complete for* GapP.
(b) DETERMINANT *is complete for* GapL

The class of problems that are AC^0-Turing reducible to DETERMINANT is one of the most important subclasses of NC, and in fact contains most of the natural problems for which NC algorithms are known.

29.8 Kolmogorov Complexity

Until now, we have considered only dynamic complexity measures, namely, the time and space used by Turing machines. **Kolmogorov complexity** is a static complexity measure that captures the difficulty of describing a string. For example, the string consisting of three million zeroes can be described with fewer than three million symbols (as in this sentence). In contrast, for a string consisting of three million randomly generated bits, with high probability there is no shorter description than the string itself.

Let U be a universal Turing machine (see Section 26.2 of Chapter 26). Let ϵ denote the empty string. The **Kolmogorov complexity** of a binary string y with respect to U, denoted by $K_U(y)$, is the length of the shortest binary string i such that on input $\langle i, \epsilon \rangle$, machine U outputs y. In essence, i is a description of y, for it tells U how to generate y.

The next theorem states that different choices for the universal Turing machine affect the definition of Kolmogorov complexity in only a small way.

THEOREM 29.13 **(Invariance Theorem)** *There exists a universal Turing machine U such that for every universal Turing machine U', there is a constant c such that for all y, $K_U(y) \leq K_{U'}(y) + c$.*

Henceforth, let K be defined by the universal Turing machine of Theorem 29.13. For every integer n and every binary string y of length n, because y can be described by giving itself explicitly, $K(y) \leq n + c'$ for a constant c'. Call y **incompressible** if $K(y) \geq n$. Since there are 2^n binary strings of length n, and only $2^n - 1$ possible shorter descriptions, there exists an **incompressible string** for every length n.

Kolmogorov complexity gives a precise mathematical meaning to the intuitive notion of "randomness." If someone flips a coin fifty times and it comes up "heads" each time, then intuitively, the sequence of flips is not random—although from the standpoint of probability theory the all-heads sequence is precisely as likely as any other sequence. Probability theory does not provide the tools for calling one sequence "more random" than another; Kolmogorov complexity theory does.

Kolmogorov complexity provides a useful framework for presenting combinatorial arguments. For example, when one wants to prove that an object with some property P exists, then it is sufficient to show that any object that does *not* have property P has a short description; thus any incompressible (or "random") object must have property P. This sort of argument has been useful in proving lower bounds in complexity theory. For example, Dietzfelbinger et al. [17] use Kolmogorov complexity to show that no Turing machine with a single worktape can compute the transpose of a matrix in less than time $\Omega(n^{3/2}/\sqrt{\log n})$.

29.9 Research Issues and Summary

As stated in the introduction to Chapter 27, the goals of complexity theory are (1) to ascertain the amount of computational resources required to solve important computational problems, and (2) to classify problems according to their difficulty. The preceding two chapters have explained how complexity theory has devised a classification scheme in order to meet the second goal. The present chapter has presented a few of the additional notions of complexity that have been devised in order to capture more problems in this scheme. Progress toward the first goal (proving lower bounds) depends on knowing that levels in this classification scheme are in fact distinct. Thus the core research questions in complexity theory are expressed in terms of separating complexity classes:

- Is L different from NL?
- Is P different from RP or BPP?
- Is P different from NP?
- Is NP different from PSPACE?

Motivated by these questions, much current research is devoted to efforts to understand the power of nondeterminism, randomization, and interaction. In these studies, researchers have gone well beyond the theory presented in Chapters 27, 28, and 29:

- Beyond Turing machines and Boolean circuits, to restricted and specialized models in which nontrivial lower bounds on complexity can be proved;

- Beyond deterministic reducibilities, to nondeterministic and probabilistic reducibilities, and refined versions of the reducibilities considered here;
- Beyond worst-case complexity, to average-case complexity.

We have illustrated how recent research in complexity theory has had direct applications to other areas of computer science and mathematics. Probabilistically checkable proofs were used to show that obtaining approximate solutions to some optimization problems is as difficult as solving them exactly. Complexity theory provides new tools for studying questions in finite model theory, a branch of mathematical logic. Fundamental questions in complexity theory are intimately linked to practical questions about the use of cryptography for computer security, such as the existence of one-way functions and the strength of public key cryptosystems.

This last point illustrates the urgent practical need for progress in computational complexity theory. Many popular cryptographic systems in current use are based on unproven assumptions about the difficulty of computing certain functions (such as the factoring and discrete logarithm problems; see Chapters 38–44 of this *Handbook* for more background on cryptography). All of these systems are thus based on wishful thinking and conjecture. The need to resolve these open questions and replace conjecture with mathematical certainty should be self-evident. (In the brief history of complexity theory, we have learned that many popular conjectures turn out to be incorrect.)

With precisely defined models and mathematically rigorous proofs, research in complexity theory will continue to provide sound insights into the difficulty of solving real computational problems.

29.10 Defining Terms

Descriptive complexity: The study of classes of languages described by formulas in certain systems of logic.

Incompressible string: A string whose **Kolmogorov complexity** equals its length, so that it has no shorter encodings.

Interactive proof system: A protocol in which one or more *provers* try to convince another party called the *verifier* that the prover(s) possess certain true knowledge, such as the membership of a string x in a given language, often with the goal of revealing no further details about this knowledge. The prover(s) and verifier are formally defined as **probabilistic Turing machines** with special "interaction tapes" for exchanging messages.

Kolmogorov complexity: The minimum number of bits into which a string can be compressed without losing information. This is defined with respect to a fixed but *universal* decompression scheme, given by a universal Turing machine.

L-reduction: A Karp reduction that preserves approximation properties of optimization problems.

Optimization problem: A computational problem in which the object is not to decide some yes/no property, as with a decision problem, but to find the best solution in those "yes" cases where a solution exists.

Polynomial hierarchy: The collection of classes of languages accepted by k-alternating Turing machines, over all $k \geq 0$ and with initial state existential or universal. The bottom level ($k = 0$) is the class P, and the next level ($k = 1$) comprises NP and co-NP.

Polynomial time approximation scheme (PTAS): A meta-algorithm that for every $\epsilon > 0$ produces a polynomial time ϵ-approximation algorithm for a given optimization problem.

Probabilistic Turing machine: A Turing machine in which some transitions are random choices among finitely many alternatives.

Probabilistically checkable proof: An interactive proof system in which provers follow a fixed strategy, one not affected by any messages from the verifier. The prover's strategy for a given instance

x of a decision problem can be represented by a finite oracle language B_x, which constitutes a proof of the correct answer for x.

Relational structure: The counterpart in formal logic of a data structure or class instance in the object-oriented sense. Examples are strings, directed graphs, and undirected graphs. Sets of relational structures generalize the notion of languages as sets of strings.

References

[1] Abiteboul, S. and Vianu, V., Datalog extensions for database queries and updates. *J. Comp. Sys. Sci.,* 43, 62–124, 1991.

[2] Abiteboul, S. and Vianu, V., Computing with first-order logic. *J. Comp. Sys. Sci.,* 50, 309–335, 1995.

[3] Adleman, L., Two theorems on random polynomial time. In *Proc. 19th Annual IEEE Symposium on Foundations of Computer Science,* 75–83, 1978.

[4] Allender, E., The permanent requires large uniform threshold circuits. Technical Report TR 97-51, DIMACS, Sep. 1997. Submitted for publication.

[5] Alvarez, C. and Jenner, B., A very hard log-space counting class. *Theor. Comp. Sci.,* 107, 3–30, 1993.

[6] Ambos-Spies, K., A note on complete problems for complexity classes. *Inf. Proc. Lett.,* 23, 227–230, 1986.

[7] Arora, S. and Safra, S., Probabilistic checking of proofs. In *Proc. 33rd Annual IEEE Symposium on Foundations of Computer Science,* 2–13, 1992.

[8] Arora, S., Lund, C., Motwani, R., Sudan, M., and Szegedy, M., Proof verification and hardness of approximation problems. In *Proc. 33rd Annual IEEE Symposium on Foundations of Computer Science,* 14–23, 1992.

[9] Babai, L. and Moran, S., Arthur-Merlin games: A randomized proof system, and a hierarchy of complexity classes. *J. Comp. Sys. Sci.,* 36, 254–276, 1988.

[10] Babai, L., Fortnow, L., and Lund, C., Nondeterministic exponential time has two-prover interactive protocols. *Computational Complexity,* 1, 3–40, 1991. Addendum in Vol. 2 of same journal.

[11] Balcázar, J., Díaz, J., and Gabarró, J., *Structural Complexity I, II.* Springer Verlag, 1990. Part I published in 1988.

[12] Barrington, D.M., Immerman, N., and Straubing, H., On uniformity within NC^1. *J. Comp. Sys. Sci.,* 41, 274–306, 1990.

[13] Bovet, D. and Crescenzi, P., *Introduction to the Theory of Complexity.* Prentice Hall International (UK) Limited, Hertfordshire, U.K., 1994.

[14] Büchi, J., Weak second-order arithmetic and finite automata. *Zeitschrift für Mathematische Logik und Grundlagen der Mathematik,* 6, 66–92, 1960.

[15] Chandra, A. and Harel, D., Structure and complexity of relational queries. *J. Comp. Sys. Sci.,* 25, 99–128, 1982.

[16] Chandra, A., Kozen, D., and Stockmeyer, L., Alternation. *J. Assn. Comp. Mach.,* 28, 114–133, 1981.

[17] Dietzfelbinger, M., Maass, W., and Schnitger, G., The complexity of matrix transposition on one-tape off-line Turing machines. *Theor. Comp. Sci.,* 82, 113–129, 1991.

[18] Downey, R. and Fellows, M., Fixed-parameter tractability and completeness I: Basic theory. *SIAM J. Comput.,* 24, 873–921, 1995.

[19] Enderton, H.B., *A Mathematical Introduction to Logic.* Academic Press, New York, 1972.

[20] Fagin, R., Generalized first-order spectra and polynomial-time recognizable sets. In R. Karp, Ed., *Complexity of Computation: Proceedings of Symposium in Applied Mathematics of the American Mathematical Society and the Society for Industrial and Applied Mathematics, Vol. VII,* 43–73. SIAM-AMS, 1974.
[21] Fagin, R., Finite model theory—a personal perspective. *Theor. Comp. Sci.,* 116, 3–31, 1993.
[22] Feige, U., Goldwasser, S., Lovász, L., Safra, S., and Szegedy, M., Approximating clique is almost NP-complete. In *Proc. 32nd Annual IEEE Symposium on Foundations of Computer Science,* 2–12, 1991.
[23] Feige, U., Goldwasser, S., Lovász, L., Safra, S., and Szegedy, M., Interactive proofs and the hardness of approximating cliques. *Journal of the ACM,* 43, 268–292, 1996.
[24] Fortnow, L. and Sipser, M., Are there interactive protocols for co-NP languages? *Inf. Proc. Lett.,* 28, 249–251, 1988.
[25] Gill, J., Computational complexity of probabilistic Turing machines. *SIAM J. Comput.,* 6, 675–695, 1977.
[26] Goldwasser, S., Micali, S., and Rackoff, C., The knowledge complexity of interactive proof systems. *SIAM J. Comput.,* 18, 186–208, 1989.
[27] Hartmanis, J., Ed., *Computational Complexity Theory.* American Mathematical Society, 1989.
[28] Hartmanis, J., On computational complexity and the nature of computer science. *Comm. Assn. Comp. Mach.,* 37, 37–43, 1994.
[29] Immerman, N., Descriptive and computational complexity. In J. Hartmanis, Ed., *Computational Complexity Theory,* volume 38 of *Proc. Symp. in Applied Math.,* 75–91. American Mathematical Society, 1989.
[30] Impagliazzo, R. and Wigderson, A., P = BPP if E requires exponential circuits: Derandomizing the XOR Lemma. In *Proc. 29th Annual ACM Symposium on the Theory of Computing,* 220–229, 1997.
[31] Jones, N. and Selman, A., Turing machines and the spectra of first-order formulas. *J. Assn. Comp. Mach.,* 39, 139–150, 1974.
[32] Karp, R. and Lipton, R., Turing machines that take advice. *L'Enseignement Mathématique,* 28, 191–210, 1982.
[33] Krentel, M., The complexity of optimization problems. *J. Comp. Sys. Sci.,* 36, 490–509, 1988.
[34] Lautemann, C., BPP and the polynomial hierarchy. *Inf. Proc. Lett.,* 17, 215–217, 1983.
[35] Li, M. and Vitányi, P., *An Introduction to Kolmogorov Complexity and its Applications.* Springer-Verlag, 1993.
[36] Lindell, S., How to define exponentiation from addition and multiplication in first-order logic on finite structures, 1994. Unpublished manuscript. The main result will appear in a forthcoming text by N. Immerman, *Descriptive and Computational Complexity,* In the Springer-Verlag "Graduate Texts in Computer Science" series.
[37] Lund, C., Fortnow, L., Karloff, H., and Nisan,N., Algebraic methods for interactive proof systems. *J. Assn. Comp. Mach.,* 39, 859–868, 1992.
[38] Lutz, J., The quantitative structure of exponential time. In L. Hemaspaandra and A. Selman, Eds., *Complexity Theory Retrospective II,* 225–260. Springer-Verlag, 1997.
[39] Lynch, J., Complexity classes and theories of finite models. *Math. Sys. Thy.,* 15, 127–144, 1982.
[40] McNaughton, R. and Papert, S., *Counter-Free Automata.* MIT Press, Cambridge, MA, 1971.
[41] Papadimitriou, C. and Yannakakis, M., The complexity of facets (and some facets of complexity). *J. Comp. Sys. Sci.,* 28, 244–259, 1984.
[42] Papadimitriou, C., *Computational Complexity.* Addison-Wesley, Reading, MA, 1994.
[43] Schützenberger, M.P., On finite monoids having only trivial subgroups. *Inform. and Control,* 8, 190–194, 1965.
[44] Shamir, A., IP = PSPACE. *J. Assn. Comp. Mach.,* 39, 869–877, 1992.

[45] Sipser, M., On relativization and the existence of complete sets. In *Proc. 9th Annual International Conference on Automata, Languages, and Programming,* volume 140 of *Lect. Notes in Comp. Sci.,* 523–531. Springer-Verlag, 1982.
[46] Sipser, M., Borel sets and circuit complexity. In *Proc. 15th Annual ACM Symposium on the Theory of Computing,* 61–69, 1983.
[47] Sipser, M., The history and status of the P versus NP question. In *Proc. 24th Annual ACM Symposium on the Theory of Computing,* 603–618, 1992.
[48] Stearns, R., Juris Hartmanis: the beginnings of computational complexity. In A. Selman, Ed., *Complexity Theory Retrospective,* 5–18. Springer-Verlag, New York, 1990.
[49] Stockmeyer, L., The polynomial time hierarchy. *Theor. Comp. Sci.,* 3, 1–22, 1976.
[50] Stockmeyer, L., Classifying the computational complexity of problems. *J. Symb. Logic,* 52, 1–43, 1987.
[51] Toda, S., PP is as hard as the polynomial-time hierarchy. *SIAM J. Comput.,* 20, 865–877, 1991.
[52] Valiant, L., The complexity of computing the permanent. *Theor. Comp. Sci.,* 8, 189–201, 1979.
[53] van Leeuwen, J., Ed., *Handbook of Theoretical Computer Science,* volume A. Elsevier and MIT Press, 1990.
[54] Vinay, V., Counting auxiliary pushdown automata and semi-unbounded arithmetic circuits. In *Proc. 6th Annual IEEE Conference on Structure in Complexity Theory,* 270–284, 1991.
[55] Wagner, K. and Wechsung, G., *Computational Complexity.* D. Reidel, 1986.
[56] Wang, J., Average-case computational complexity theory. In L. Hemaspaandra and A. Selman, Eds., *Complexity Theory Retrospective II,* 295–328. Springer-Verlag, 1997.
[57] Wrathall, C., Complete sets and the polynomial-time hierarchy. *Theor. Comp. Sci.,* 3, 23–33, 1976.

Further Information

Primary sources for major theorems presented in this chapter include Theorem 29.1 [16, 49, 57]; Theorem 29.3(a,b) [25], (c) [34, 46], (d) [51], (e) [4]; Theorem 29.4 [3]; Theorem 29.5(a) [1], (b) [49], (c) [20], (d,e,f) [29], (g) ([36], cf. [12]); Theorem 29.6(a) [14], (b) [40, 43]; Theorem 29.7 [44]; Theorem 29.8 [10]; Theorem 29.9 [7]; Theorem 29.10 [8]; Theorem 29.12(a) [52], (b) [54]. The operators in Definition 29.2 are from [29] and [1]. Interactive proof systems were defined by Goldwasser et al. [26], and in the "Arthur-Merlin" formulation, by Babai and Moran [9]. A large compendium of optimization problems and hardness results collected by P. Crescenzi and V. Kann is available at:

http://www.nada.kth.se/~viggo/index-en.html

The class #P was introduced by Valiant [52], and #L by Alvarez and Jenner [5]. Li and Vitányi [35] give a far-reaching and comprehensive scholarly treatment of Kolmogorov complexity, with many applications, as well as the source of Theorem 29.13.

Three contemporary textbooks on complexity theory are [11], [13], and [42]. Wagner and Wechsung [55] provide is an exhaustive survey of complexity theory that covers work published before 1986. Another perspective of some of the issues covered in these three chapters may be found in the survey [50].

A good general reference is the *Handbook of Theoretical Computer Science* [53], volume A. The following chapters in the *Handbook* are particularly relevant: "Machine Models and Simulations," by P. van Emde Boas, pp. 1–66; "A Catalog of Complexity Classes," by D.S. Johnson, pp. 67–161; "Machine-Independent Complexity Theory," by J.I. Seiferas, pp. 163–186; "Kolmogorov Complexity and Its Applications," by M. Li and P.M.B. Vitányi, pp. 187–254; and "The Complexity of Finite Functions," by R.B. Boppana and M. Sipser, pp. 757–804, which covers circuit complexity.

A collection of articles edited by Hartmanis [27] includes an overview of complexity theory, and chapters on sparse complete languages, on relativizations, on interactive proof systems, and on applications of complexity theory to cryptography. For historical perspectives on complexity theory, see [28], [47], and [48].

There are many areas of complexity theory that we have not been able to cover in these chapters. Some of them cross-pollinate with other fields of computer science and are reflected in other chapters of this *Handbook*. Three others are *average-case complexity, resource-bounded measure theory,* and *parameterized complexity.* Recent surveys on the first two are by Lutz [38] and Wang [56], while the third stems from Downey and Fellows [18] and has its own Web site, currently maintained by M. Hallett at

`http://csr.uvic.ca/home/mhallett/W.hier/compendium.html`

Surveys and lecture notes on complexity theory that can be obtained via WWW are maintained by A. Czumaj and M. Kutylowski at

`http://www.uni-paderborn.de/fachbereich/AG/agmadh/WWW/english/scripts.html`

As usual with the WWW, these links are subject to change. A good stem page to begin searches is the site for SIGACT, the ACM Special Interest Group on Algorithms and Computation Theory:

`http://sigact.acm.org/`

This has a pointer to a "Virtual Address Book" that indexes the personal Web pages of over 1,000 computer scientists, including all three authors of these chapters. Many of these pages have downloadable papers and links to further research resources.

Research papers on complexity theory are presented at several annual conferences, including the annual ACM Symposium on Theory of Computing; the annual International Colloquium on Automata, Languages, and Programming, sponsored by the European Association for Theoretical Computer Science (EATCS); and the annual Symposium on Foundations of Computer Science, sponsored by the IEEE. The annual Conference on Computational Complexity (formerly Structure in Complexity Theory), also sponsored by the IEEE, is entirely devoted to complexity theory. Research articles on complexity theory regularly appear in the following journals, among others: *Chicago Journal on Theoretical Computer Science, Computational Complexity, Information and Computation, Journal of the ACM, Journal of Computer and System Sciences, Mathematical Systems Theory, SIAM Journal on Computing,* and *Theoretical Computer Science.* Each issue of *ACM SIGACT News* and *Bulletin of the EATCS* contains a column on complexity theory.

30

Computational Learning Theory

Sally A. Goldman
Washington University,
St. Louis Missouri

30.1 Introduction

Since the late 1950s, computer scientists (particularly those working in the area of artificial intelligence) have been trying to understand how to construct computer programs that perform tasks we normally think of as requiring human intelligence, and which can improve their performance over time by modifying their behavior in response to experience. In other words, one objective has been to design computer programs that can learn. For example, Samuels designed a program to play checkers in the early 1960s that could improve its performance as it gained experience playing against human opponents. More recently, research on artificial neural networks has stimulated interest in the design of systems capable of performing tasks that are difficult to describe algorithmically (such as recognizing a spoken word or identifying an object in a complex scene), by exposure to many examples.

As a concrete example consider the task of hand-written character recognition. A learning algorithm is given a set of examples where each contains a hand-written character as specified by a set of *attributes* (e.g., the height of the letter) along with the name (*label*) for the intended character. This set of examples is often called the *training data*. The goal of the learner is to efficiently construct a rule (often referred to as a *hypothesis* or *classifier*) that can be used to take some previously unseen character and with high accuracy determine the proper label.

Computational learning theory is a branch of theoretical computer science that formally studies how to design computer programs that are capable of learning, and identifies the computational limits of learning by machines. Historically, researchers in the artificial intelligence community have judged learning algorithms empirically, according to their performance on sample problems. While such evaluations provide much useful information and insight, often it is hard using such evaluations to make meaningful comparisons among competing learning algorithms.

Computational learning theory provides a formal framework in which to precisely formulate and address questions regarding the performance of different learning algorithms so that careful comparisons of both the predictive power and the computational efficiency of alternative learning algorithms can be made. Three key aspects that must be formalized are the way in which the learner interacts with its environment, the definition of successfully completing the learning task, and a formal definition of efficiency of both data usage (*sample complexity*) and processing time (*time complexity*). It is important to remember that the theoretical learning models studied are abstractions from real-life problems. Thus close connections with experimentalists are useful to help validate or modify these abstractions so that the theoretical results help to explain or predict empirical performance. In this direction, computational learning theory research has close connections to machine learning research. In addition to its predictive capability, some other important features of a good theoretical model are simplicity, robustness to variations in the learning scenario, and an ability to create insights to empirically observed phenomena.

The first theoretical studies of machine learning were performed by inductive inference researchers (see [10]) beginning with the introduction of the first formal definition of learning given by Gold [24]. In Gold's model, the learner receives a sequence of examples and is required to make a sequence of guesses as to the underlying rule (*concept*) such that this sequence converges at some finite point to a single guess that correctly names the unknown rule. A key distinguishing characteristic is that Gold's model does not attempt to capture any notion of the efficiency of the learning process, whereas the field of computational learning theory emphasizes the computational feasibility of the learning algorithm.

Another closely related field is that of pattern recognition (see [21] and [20]). Much of the theory developed by learning theory researchers to evaluate the amount of data needed to learn directly adapts results from the fields of statistics and pattern recognition. A key distinguishing factor is that learning theory researchers study both the data (information) requirements for learning *and* the time complexity of the learning algorithm. (In contrast, pattern recognition researchers tend to focus on issues related to the data requirements.) Finally, there are a lot of close relations to work on artificial neural networks. While much of neural network research is empirical, there is also a good amount of theoretical work (see [20]).

This chapter is structured as follows. In Section 30.2 we describe the basic framework of concept learning and give notation that we use throughout the chapter. Next, in Section 30.3 we describe the *PAC* (*distribution-free*) model that began the field of computational learning theory. An important early result in the field is the demonstration of the relationships between the *VC-dimension,* a combinatorial measure, and the data requirements for **PAC learning.** We then discuss some commonly studied noise models and general techniques for PAC learning from noisy data.

In Section 30.4 we cover some of the other commonly studied formal learning models. First we study an **on-line learning model.** Unlike the PAC model in which there is a training period, in the on-line learning model the learner must improve the quality of its predictions as it functions in the world. Next we study the query model which is very similar to the on-line model except that the learner plays a more active role.

As in other theoretical fields, along with having techniques to prove positive results, another important component of learning theory research is to develop and apply methods to prove when a learning problem is hard. In Section 30.5 we describe some techniques used to show that learning problems are hard. Next, in Section 30.6, we explore a variation of the PAC model called the **weak learning model,** and study techniques for boosting the performance of a mediocre learning algorithm. Finally, we close with a brief overview of some of the many current research issues being studied within the field of computational learning theory.

30.2 General Framework

For ease of exposition, we initially focus on *concept learning* in which the learner's goal is to infer how an unknown target function classifies (as positive or negative) examples from a given domain. In the character recognition example, one possible task of the learner would be to classify each character as a numeral or non-numeral. Most of the definitions given here naturally extend to the general setting of learning functions with multiple-valued or real-valued outputs. Later in Section 30.4 we briefly discuss the more general problem of learning a real-valued function.

Notation

The *instance space* (*domain*) $\mathcal{X}$ is the set of all possible objects (instances) to be classified. Two very common domains are the Boolean hypercube $\{0, 1\}^n$ and continuous n-dimensional space $\Re^n$. For example, if there are n Boolean attributes then each example can be expressed as an element of $\{0, 1\}^n$. Likewise, if there are n real-valued attributes, then $\Re^n$ can be used. A *concept* f is a Boolean function over domain $\mathcal{X}$. Each $x \in \mathcal{X}$ is referred to as an *example* (*point*).

A **concept class** $\mathcal{C}$ is a collection of subsets of $\mathcal{X}$. That is $\mathcal{C} \subseteq 2^{\mathcal{X}}$. Typically (but not always), it is assumed that the learner has prior knowledge of $\mathcal{C}$. Examples are classified according to membership of a *target concept* $f \in \mathcal{C}$. Often, instead of using this set theoretic view of $\mathcal{C}$, a functional view is used in which $f(x)$ gives the classification of concept $f \in \mathcal{C}$ for each example $x \in \mathcal{X}$. An example $x \in \mathcal{X}$ is a *positive example* of f if $f(x) = 1$ (equivalently $x \in f$), or a *negative example* of f if $f(x) = 0$ (or equivalently $x \notin f$). Often $\mathcal{C}$ is decomposed into subclasses $\mathcal{C}_n$ according to some natural size measure n for encoding an example. For example, in the Boolean domain, n is the number of Boolean attributes. Let $\mathcal{X}_n$ denote the set of examples to be classified for each problem of size n, $\mathcal{X} = \bigcup_{n \geq 1} \mathcal{X}_n$.

We use the class $\mathcal{C}_{\text{halfspace}}$, the set of halfspaces in $\Re^n$, and the class $\mathcal{C}^{\cap_s}_{\text{halfspace}}$, the set of all the intersections of up to s halfspaces in $\Re^d$, to illustrate some of the basic techniques for designing learning algorithms[1] (see Fig. 30.1). We now formally define a halfspace in $\Re^n$ and describe how the points in $\Re^n$ are classified

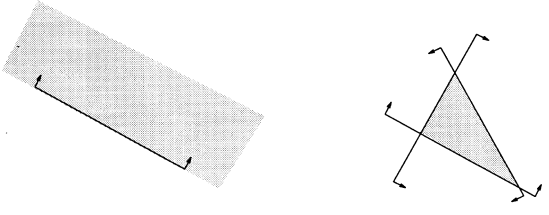


FIGURE 30.1 The left figure shows (in $\Re^2$) a concept from $\mathcal{C}_{\text{halfspace}}$ and the right figure shows a concept for $\mathcal{C}^{\cap_s}_{\text{halfspace}}$ for $s = 3$. The points from $\Re^2$ that are classified as positive are shaded. The unshaded points are classified as negative.

by it. Let $\vec{x} = (x_1, \ldots, x_n)$ denote an element of $\Re^n$. A halfspace defined by $\vec{a} \in \Re^n$ and $b \in \Re$ classifies as positive the set of points $\{\vec{x} \mid \vec{a} \cdot \vec{x} \geq b\}$. For example, in two dimensions (i.e., $n = 2$), $5x_1 - 2x_2 \geq -3$

[1] As a convention, when the number of dimensions can be arbitrary we use n, and when the number of dimensions must be a constant we use d.

defines a halfspace. The point (0, 0) is a positive example and (2, 10) is a negative example. For $\mathcal{C}^{\cap_s}_{\text{halfspace}}$, an example is positive exactly when it is classified as positive by each of the halfspaces forming the intersection. Thus the set of positive points forms a (possibly open) convex polytope in $\Re^n$.

We also study some Boolean concepts. For these concepts the domain is $\{0, 1\}^n$. Let $x_1, \ldots, x_n$ denote the n Boolean attributes. A *literal* is either x_i or $\overline{x_i}$ where $i = 1, \ldots, n$. A term is a conjunction of literals. Finally, a DNF formula is a disjunction of terms. One of the biggest open problems of computational learning theory is whether or not the concept class of DNF formulas is efficiently PAC learnable. Since the problem of learning general DNF formulas is a long-standing open problem, several subclasses have been studied. A *monotone DNF formula* is a DNF formula in which there are no negated variables. A *read-once DNF formula* is a DNF formula in which each variable appears at most once. In addition to adding a restriction that the formula be monotone and/or read-once, one can also limit either the size of each term or the number of terms. A *k-term DNF formula* is a DNF formula in which there are at most k terms. Finally, a *k-DNF formula* is a DNF formula in which at most k literals are used by each term.

As one would expect, the time and sample complexity of the learning algorithm depends on the complexity of the underlying target concept. For example, the complexity for learning a monotone DNF formula is likely to depend on the number of terms (conjuncts) in the formula. To give a careful definition of the size of a concept $f \in \mathcal{C}$, we associate a language $\mathcal{R}_\mathcal{C}$ with each concept class $\mathcal{C}$ that is used for representing concepts in $\mathcal{C}$. Each $r \in \mathcal{R}_\mathcal{C}$ denotes some $f \in \mathcal{C}$, and every $f \in \mathcal{C}$ has at least one representation $r \in \mathcal{R}_\mathcal{C}$. Each concept $f \in \mathcal{C}_n$ has a *size* denoted by $|f|$, which is the representation length of the shortest $r \in \mathcal{R}_\mathcal{C}$ that denotes f. For ease of exposition, in the remainder of this chapter we use $\mathcal{C}$ and $\mathcal{R}_\mathcal{C}$ interchangeably.

To appreciate the significance of the choice of the representation class, consider the problem of learning a regular language.[2] The question as to whether an algorithm is efficient depends heavily on the representation class. As defined more formally below, an *efficient* learning algorithm must have time and sample complexity polynomial in $|f|$ where f is the target concept. The target regular language could be represented as a deterministic finite-state automaton (DFA) or as a nondeterministic finite-state automaton (NFA). However, the length of the representation as a NFA can be exponentially smaller than the shortest representation as a DFA. Thus, the learner may be allowed exponentially more time when learning the class of regular languages as represented by DFAs versus when learning the class of regular languages as represented by NFAs. Often to make the representation class clear, the concept class names the representation. Thus instead of talking about learning a regular language, we talk about learning a DFA or an NFA. Thus the problem of learning a DFA is easier than learning an NFA. Similar issues arise when learning Boolean functions. For example, whether the function is represented as a decision tree, a DNF formula, or a Boolean circuit greatly affects the representation size.

30.3 PAC Learning Model

The field of computational learning theory began with Valiant's seminal work [48] in which he defined the *PAC*[3] (*distribution-free*) learning model. In the PAC model examples are generated according to an unknown probability distribution $\mathcal{D}$, and the goal of a learning algorithm is to classify with high accuracy (with respect to the distribution $\mathcal{D}$) any further (unclassified) examples.

We now formally define the PAC model. To obtain information about an unknown target function $f \in \mathcal{C}_n$, the learner is provided access to labeled (positive and negative) examples of f, drawn randomly according to some unknown target distribution $\mathcal{D}$ over $\mathcal{X}_n$. The learner is also given as input ϵ and δ

[2] See the section on "Notation" of Chapter 30 and Section 31.2 of Chapter 31 in this *Handbook* for background on regular languages, DFAs and NFAs.

[3] PAC is an acronym coined by Dana Angluin for probably approximately correct.

such that $0 < \epsilon, \delta < 1$, and an upper bound k on $|f|$. The learner's goal is to output, with probability at least $1 - \delta$, a *hypothesis* $h \in \mathcal{C}_n$ that has probability at most ϵ of disagreeing with f on a randomly drawn example from $\mathcal{D}$ (thus, h has *error* at most ϵ). If such a learning algorithm $\mathcal{A}$ exists (that is, an algorithm $\mathcal{A}$ meeting the goal for any $n \geq 1$, any target concept $f \in \mathcal{C}_n$, any target distribution $\mathcal{D}$, any $\epsilon, \delta > 0$, and any $k \geq |f|$), then $\mathcal{C}$ is **PAC learnable.** A PAC learning algorithm is a polynomial-time (*efficient*) algorithm if the number of examples drawn and the computation time are polynomial in n, k, $1/\epsilon$, and $1/\delta$. We note that most learning algorithms are really functions from samples to hypotheses (i.e., given a sample S the learning algorithm produces a hypothesis h). It is only for the analysis in which we say that for a given ϵ and δ, the learning algorithm is guaranteed with probability at least $1 - \delta$ to output a hypothesis with error at most ϵ given a sample whose size is a function of ϵ, δ, n and k. Thus, empirically, one can generally run a PAC algorithm on provided data and then empirically measure the error of the final hypothesis. One exception is when trying to empirically use statistical query algorithms since, for most of these algorithms, the algorithm uses ϵ for more than just determining the desired sample size. (See "Gaining Noise Tolerance in the PAC Model" for a discussion of statistical query algorithms, and Goldman and Scott [27] for a discussion of their empirical use).

As originally formulated, PAC learnability also required the hypothesis to be a member of the concept class $\mathcal{C}_n$. We refer to this more stringent learning model as *proper PAC-learnability.* The work of Pitt and Valiant [42] shows that a prerequisite for proper PAC-learning is the ability to solve the consistent hypothesis problem, which is the problem of finding a concept $f \in \mathcal{C}$ that is consistent with a provided sample. Their result implies that if the consistent hypothesis problem is NP-hard for a given concept class (as they show is the case for the class of k-term DNF formulas) and NP $\neq$ RP, then the learning problem is hard.[4] A more general form of learning in which the goal is to find any polynomial-time algorithm that classifies instances accurately in the PAC sense is commonly called *prediction.* In this less stringent variation of the PAC model, the algorithm need not output a hypothesis from the concept class $\mathcal{C}$ but instead is just required to make its prediction in polynomial time. This idea of prediction in the PAC model originated in the paper of Haussler et al. [30], and is discussed in Pitt and Warmuth [43]. Throughout the remainder of this chapter, when referring to the PAC learning model we allow the learner to output any hypothesis that can be evaluated in polynomial time. That is, given a hypothesis h and an example x, we require that $h(x)$ can be computed in polynomial time. We refer to the model in which the learner is required to output a hypothesis $h \in \mathcal{C}_n$ as the *proper PAC learning model.*

Many variations of the PAC model are known to be equivalent (in terms of what concept classes are efficiently learnable) to the model defined above. We now briefly discuss one of these variations. In the above definition of the PAC model, along with receiving ϵ, δ, and n as input, the learner also receives k, an upperbound on $|f|$. The learner must be given ϵ and δ. Further, by looking at just one example, the value of n is known. Yet giving the learner knowledge of k may appear to make the problem easier. However, if the demand of polynomial-time computation is replaced with expected polynomial-time computation, then the learning algorithm need not be given the parameter k, but could "guess" it instead. We now briefly review the standard *doubling technique* used to convert an algorithm A designed to have k as input to an algorithm B that has no prior knowledge of k. Algorithm B begins with an estimate, say 1, for its upperbound on k and runs algorithm A using this estimate to obtain hypothesis h. Then algorithm B uses a *hypothesis testing* procedure to determine if the error of h is at most ϵ. Since the learner can only gather a random sample of examples, it is not possible to distinguish a hypothesis with error ϵ from one with error just greater than ϵ. However, by drawing a sufficiently large sample and looking at the empirical performance of h on that sample, we can distinguish a hypothesis with error at most $\epsilon/2$ from one with

[4] For background on complexity classes see Chapter 33 (NP defined) and Chapter 35 (RP defined) of this *Handbook.*

error more than ϵ. In particular, given a sample[5] of size $m = \lceil \frac{32}{\epsilon} \ln \frac{2}{\delta} \rceil$, if h misclassifies at most $\frac{3\epsilon}{4} \cdot m$ examples then B accepts h. Otherwise, algorithm B doubles its estimate for k and repeats the process. For the technical details of the above argument as well as for a discussion the other equivalences, see [29].

Sample Complexity Bounds, the VC-Dimension, and Occam's Razor

Although we are concerned with the time complexity of the learning algorithm, a fundamental quantity to first compute is the sample complexity (data) requirements. Blumer et al. [16] identified a combinatorial parameter of a class of functions defined by Vapnik and Chervonenkis [50]. They give strong results relating this parameter, called the **VC-dimension,** to information-theoretic bounds on the sample size needed to have accurate generalization. Note that given a sufficiently large sample there is still the computational problem of finding a "good" hypothesis.

We now define the VC-dimension. We say that a finite set $S \subseteq \mathcal{X}$ is *shattered* by the concept class $\mathcal{C}$ if for each of the $2^{|S|}$ subsets $S' \subseteq S$, there is a concept $f \in \mathcal{C}$ that contains all of S' and none of $S - S'$. In other words, for any of the $2^{|S|}$ possible labelings of S (where each example $s \in S$ is either positive or negative), there is some $f \in \mathcal{C}$ that realizes the desired labeling (see Fig. 30.2). We can now define the VC-dimension of a concept class $\mathcal{C}$. The VC-dimension of $\mathcal{C}$, denoted $\mathrm{VCD}(\mathcal{C})$, is the smallest d for which no set of $d + 1$ examples is shattered by $\mathcal{C}$. Equivalently, $\mathrm{VCD}(\mathcal{C})$ is the cardinality of the largest finite set of points $S \subseteq X$ that is shattered by $\mathcal{C}$.

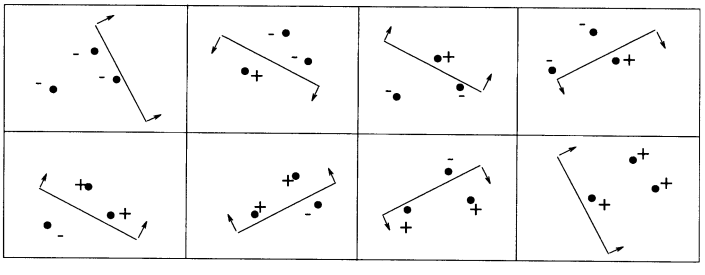


FIGURE 30.2 A demonstration that there are 3 points that are shattered by the class of two-dimensional halfspaces (i.e., $\mathcal{C}_{\text{halfspace}}$ with $n = 2$). Notice that all 8 possible ways that the points can be classified as positive or negative can be realized.

So to prove that $\mathrm{VCD}(\mathcal{C}) \geq d$ it suffices to give d examples that can be shattered. However, to prove $\mathrm{VCD}(\mathcal{C}) \leq d$ one must show that *no* set of $d + 1$ examples can be shattered. Since the VC-dimension is so fundamental in determining the sample complexity required for PAC learning, we now go through several sample computations of the VC-dimension.

Axis-parallel rectangles in $\Re^2$. For this concept class the points lying on or inside the target rectangle are positive, and points lying outside the target rectangle are negative. First, it is easily seen that there is a set of four points (e.g., $\{(0, 1), (0, -1), (1, 0), (-1, 0)\}$) that can be shattered. Thus $\mathrm{VCD}(\mathcal{C}) \geq 4$. We now argue that no set of five points can be shattered. The smallest bounding axis-parallel rectangle defined by the five points is in fact defined by at most four of the points. For p a nondefining point in the set, we see that the set cannot be shattered since it is not possible for p to be classified as negative while also classifying the others as positive. Thus $\mathrm{VCD}(\mathcal{C}) = 4$.

[5] Chernoff bounds are used to compute the sample size so that the hypothesis testing procedure gives the desired output with high probability.

Halfspaces in $\Re^2$. Points lying in or on the halfspace are positive, and the remaining points are negative. It is easily shown that any three noncollinear points (e.g., (0, 1), (0, 0), (1, 0)) are shattered by $\mathcal{C}$ (recall Fig. 30.2). Thus $\mathrm{VCD}(\mathcal{C}) \geq 3$. We now show that no set of size four can be shattered by $\mathcal{C}$. If at least three of the points are collinear then there is no halfspace that contains the two extreme points but does not contain the middle points. Thus the four points cannot be shattered if any three are collinear. Next, suppose that the points form a quadrilateral. There is no halfspace which labels one pair of diagonally opposite points positive and the other pair of diagonally opposite points negative. The final case is that one point p is in the triangle defined by the other three. In this case there is no halfspace which labels p differently from the other three. Thus clearly the four points cannot be shattered. Therefore we have demonstrated that $\mathrm{VCD}(\mathcal{C}) = 3$. Generalizing to halfspaces in $\Re^n$, it can be shown that $\mathrm{VCD}(\mathcal{C}_{\text{halfspace}}) = n + 1$ (see [16]).

Closed sets in $\Re^2$. All points lying in the set or on the boundary of the set are positive, and all points lying outside the set are negative. Any set can be shattered by $\mathcal{C}$, since a closed set can assume any shape in $\Re^2$. Thus, the largest set that can be shattered by $\mathcal{C}$ is infinite, and hence $\mathrm{VCD}(\mathcal{C}) = \infty$.

We now briefly discuss techniques that aid in computing the VC-dimension of more complex concept classes. Suppose we wanted to compute the VC-dimension of $\mathcal{C}^{\cap_s}_{\text{halfspace}}$, the class of intersections of up to s halfspaces over $\Re^d$. We would like to make use of our knowledge of the VC-dimension of $\mathcal{C}_{\text{halfspace}}$, the class of halfspaces over $\Re^d$. Blumer et al. [16] gave the following result: let $\mathcal{C}$ be a concept class with $\mathrm{VCD}(\mathcal{C}) \leq D$. Then the class defined by the intersection of up to s concepts from $\mathcal{C}$ has VC-dimension[6] at most $2Ds\lg(3s)$. In fact, the above result applies when replacing intersection with any Boolean function. Thus the concept class $\mathcal{C}^{\cap_s}_{\text{halfspace}}$ (where each halfspace is defined over $\Re^d$) has VC-dimension at most $2(d+1)s\lg(3s)$.

We now discuss the significance of the VC-dimension to the PAC model. One important property is that for $D = \mathrm{VCD}(\mathcal{C})$, the number of different labelings that can be realized (also referred to as *behaviors*) using $\mathcal{C}$ for a set S is at most $\left(\frac{e|S|}{D}\right)^D$. Thus for a constant D we have polynomial growth in the number of labelings versus the exponential growth of $2^{|S|}$. A key result in the PAC model [16] is an upperbound on the sample complexity needed to PAC learn in terms of ϵ, δ, and the VC-dimension of the **hypothesis class.** They proved that one can design a learning algorithm A for concept class $\mathcal{C}$ using hypothesis space $\mathcal{H}$ in the following manner. Any concept $h \in \mathcal{H}$ consistent with a sample of size $\max\left(\frac{4}{\epsilon}\lg\frac{2}{\delta}, \frac{8\,\mathrm{VCD}(\mathcal{H})}{\epsilon}\lg\frac{13}{\epsilon}\right)$ has error at most ϵ with probability at least $1-\delta$. To obtain a polynomial-time PAC learning algorithm what remains is to solve the algorithmic problem of finding a hypothesis from $\mathcal{H}$ consistent with the labeled sample. As an example application, see Fig. 30.3. Furthermore, Ehrenfencht and Haussler [22] proved an information-theoretic lower bound that learning any concept class $\mathcal{C}$ requires $\Omega\left(\frac{1}{\epsilon}\log\frac{1}{\delta} + \frac{\mathrm{VCD}(\mathcal{C})}{\epsilon}\right)$ examples in the worst case.

A key limitation of this technique to design a PAC learning algorithm is that the hypothesis must be drawn from some *fixed* hypothesis class $\mathcal{H}$. In particular, the complexity of the hypothesis class must be independent of the sample size. However, often the algorithmic problem of finding such a hypothesis from the desired class is NP-hard.

As an example suppose we tried to modify PAC-learn-halfspace from Fig. 30.3 to efficiently PAC learn the class of intersections of at most s halfspaces in $\Re^d$ for d the number of dimensions a constant.

[6]Note that throughout this chapter, lg is used for the base-2 logarithm. When the base of the logarithm is not significant (such as when using asymptotic notation), we use log.

PAC-learn-halfspace(n, ϵ, δ)

Draw a labeled sample S of size $\max\left(\frac{4}{\epsilon}\lg\frac{2}{\delta}, \frac{8(n+1)}{\epsilon}\lg\frac{13}{\epsilon}\right)$

Use a polynomial-time linear programming algorithm to find a halfspace h consistent with S

Output h

FIGURE 30.3 A PAC algorithm to learn a concept from $\mathcal{C}_{\text{halfspace}}$. Recall that $\text{VCD}(\mathcal{C}_{\text{halfspace}}) = n + 1$.

Suppose we are given a sample S of m example points labeled according to some $f \in \mathcal{C}^{\cap_s}_{\text{halfspace}}$. The algorithmic problem of finding a hypothesis from $\mathcal{C}^{\cap_s}_{\text{halfspace}}$ can be formulated as a set covering problem in the following manner. An instance of the set covering problem is a set $\mathcal{U}$ and a family $\mathcal{T}$ of subsets of $\mathcal{U}$ such that $\left(\bigcup_{t\in\mathcal{T}} t\right) = \mathcal{U}$. A solution is a smallest cardinality subset $\mathcal{G}$ of $\mathcal{T}$ such that $\left(\bigcup_{g\in\mathcal{G}} g\right) = \mathcal{U}$. Consider any halfspace g that correctly classifies all positive examples from S. We say that g *covers* all of the negative examples from S that are also classified as negative by g. Thus $\mathcal{U}$ is the set of negative examples from S, and $\mathcal{T}$ is the set of representative halfspaces (one for each behavior with respect to S) that correctly classify all positive examples from S.

So finding a minimum sized hypothesis from $\mathcal{C}^{\cap_s}_{\text{halfspace}}$ consistent with the sample S is exactly the problem of finding the minimum number of halfspaces that are required to cover all negative points in S. Given that the set covering problem is NP-complete, how can we proceed? We can apply the greedy approximation algorithm that has a ratio bound of $\ln|\mathcal{U}| + 1$ to find a hypothesis consistent with S that is the intersection of at most $\ln|S| + 1$ halfspaces. However, since the VC-dimension of the hypothesis grows with the size of the sample, the basic technique described above cannot be applied. In general, when using a set covering approach, the size of the hypothesis often depends on the size of the sample.

Blumer et al. [15, 16] extended this basic technique by showing that finding a hypothesis h consistent with a sample S for which the size of h is sublinear in $|S|$ is sufficient to guarantee PAC learnability. In other words, by obtaining sufficient data compression one obtains good generalization. More formally, let $\mathcal{H}^A_{k,n,m}$ be the hypothesis space used by algorithm A when each example has size n, the target concept has size k, and the sample size is m. We say that algorithm A is an **Occam algorithm** for concept class $\mathcal{C}$ if there exists a polynomial $p(k, n)$ and a constant β, $0 \le \beta < 1$, such that for any sample S with $|S| = m \ge 1$ and any k, n, A outputs a hypothesis[7] h consistent with S such that $|h| \le p(k, n)m^\beta$. Let A be an Occam algorithm for concept class $\mathcal{C}$ that has hypothesis space $\mathcal{H}^A_{k,n,m}$. If $\text{VCD}(\mathcal{H}^A_{k,n,m}) \le p(k, n)(\lg m)^\ell$ (so $|h| \le p(k, n)(\lg m)^\ell$) for some polynomial $p(k, n) \ge 2$ and $\ell \ge 1$, then A is a PAC learning algorithm for $\mathcal{C}$ using sample size

$$m = \max\left(\frac{4}{\epsilon}\lg\frac{2}{\delta}, \frac{2^{\ell+4}p(k,n)}{\epsilon}\left(\lg\frac{8(2\ell+2)^{\ell+1}p(k,n)}{\epsilon}\right)^{\ell+1}\right).$$

Figure 30.4 presents an Occam algorithm to PAC learn a concept from $\mathcal{C}^{\cap_s}_{\text{halfspace}}$ (i.e., the intersection of at most s halfspaces over $\Re^d$). Since $\text{VCD}(\mathcal{C}^{\cap_s}_{\text{halfspace}}) \le 2(d+1)s\lg(3s)$, and the hypothesis consists of the intersection of at most $s(1+\ln m)$ halfspaces, it follows that $\text{VCD}(\mathcal{H}) \le 2(d+1)s(1+\ln m)\lg(3s(1+\ln m)) \le 3ds\ln^2 m$. The size of the sample S then follows by noting that $p(s, d) = 3ds$ and $\ell = 3$. By combining the fact that $\text{VCD}(\mathcal{C}_{\text{halfspace}}) = d+1$ and the general upper bound of $\left(\frac{e|S|}{D}\right)^D$ on the number of behaviors on a sample of size $|S|$ for a class of VC-dimension D, we get that $|\mathcal{T}| \le \left(\frac{em}{d+1}\right)^{d+1} = O(m^{d+1})$.

[7] Recall that we use $|h|$ to denote the number of bits required to represent h.

(For d constant this quantity is polynomial in the size of the sample.) We can construct $\mathcal{T}$ in polynomial time by either using simple geometric arguments or the more general technique of Blumer et al. [16]. Finally, the time complexity of the greedy set covering algorithm is $O(\Sigma_{t\in\mathcal{T}}|t|) = O(m \cdot m^{d+1})$, which is polynomial in s (the number of halfspaces), ϵ, and δ for d (the number of dimensions) constant.

PAC-learn-intersection-of-halfspaces(d, s, ϵ, δ)
Draw a labeled sample S of size $m = \max\left(\frac{4}{\epsilon}\lg\frac{2}{\delta}, \frac{192ds}{\epsilon}\left(6 + 4\lg 3 + \lg\frac{ds}{\epsilon}\right)^3\right)$
Form a set $\mathcal{T}$ of halfspaces that are consistent with the positive examples
Let N be the negative examples from S
Let h be the always true hypothesis
Repeat until $N = \emptyset$
Find a halfspace $t \in \mathcal{T}$ consistent with the max. number of exs. from N
Let $h = h \cap t$
Remove the examples from N that were covered by t
Output h

FIGURE 30.4 An Occam algorithm to PAC learn a concept from $\mathcal{C}^{\cap_s}_{\text{halfspace}}$.

For the problem of learning the intersection of halfspaces the greedy covering technique provided substantial data compression. Namely, the size of our hypothesis only had a logarithmic dependence on the size of the sample. In general, only a sublinear dependence is required as given by the following result of Blumer et al. [15]. Let A be an Occam algorithm for concept class $\mathcal{C}$ that has hypothesis space $\mathcal{H}^A_{k,n,m}$. If $\text{VCD}(\mathcal{H}^A_{k,n,m}) \leq p(k,n)m^\beta$ (so $|h| \leq p(k,n)m^\beta$) for some polynomial $p(k,n) \geq 2$ and $\beta < 1$, then A is a PAC learning algorithm for $\mathcal{C}$ using sample size

$$m = \max\left(\frac{2}{\epsilon}\ln\frac{1}{\delta}, \left(\frac{2\ln 2}{\epsilon}\cdot p(k,n)\right)^{\frac{1}{1-\beta}}\right).$$

Models of Noise

The basic definition of PAC learning assumes that the data received is drawn randomly from $\mathcal{D}$ and properly labeled according to the target concept. Clearly, for learning algorithms to be of practical use they must be robust to noise in the training data. In order to theoretically study an algorithm's tolerance to noise, several formal models of noise have been studied. In the model of **random classification noise** [9], with probability $1-\eta$, the learner receives the uncorrupted example (x, ℓ). However, with probability η, the learner receives the example $(x, \bar{\ell})$. So in this noise model, learner usually gets a correct example, but some small fraction η of the time the learner receives an example in which the label has been inverted. In the model of *malicious classification noise* [47], with probability $1-\eta$, the learner receives the uncorrupted example (x, ℓ). However, with probability η, the learner receives the example (x, ℓ') in which x is unchanged, but the label ℓ' is selected by an adversary who has infinite computing power and has knowledge of the learning algorithm, the target concept, and the distribution $\mathcal{D}$. In the previous two models, only the labels are corrupted. Another noise model is that of *malicious noise* [49]. In this model, with probability $1-\eta$, the learner receives the uncorrupted example (x, ℓ). However, with probability η, the learner receives an example (x', ℓ') about which no assumptions whatsoever may be made. In particular, this example (and its label) may be maliciously selected by an adversary. Thus in this model, the learner usually gets a correct example, but some small fraction η of the time the learner gets noisy examples and the nature of the noise is unknown.

We now define two noise models that are only defined when the instance space is $\{0, 1\}^n$. In the model of *uniform random attribute noise* [47], the example $(b_1 \cdots b_n, \ell)$ is corrupted by a random process that independently flips each bit b_i to $\bar{b}_i$ with probability η for $1 \leq i \leq n$. Note that the label of the "true" example is never altered. In this noise model, the attributes' values are subject to noise, but that noise is as benign as possible. For example, the attributes' values might be sent over a noisy channel, but the label is not. Finally, in the model of *product random attribute noise* [26], the example $(b_1 \cdots b_n, \ell)$ is corrupted by a random process of independently flipping each bit b_i to $\bar{b}_i$ with some fixed probability $\eta_i \leq \eta$ for each $1 \leq i \leq n$. Thus unlike the model of uniform random attribute noise, the noise rate associated with each bit of the example may be different.

Gaining Noise Tolerance in the PAC Model

Some of the first work on designing noise-tolerant PAC algorithms was done by Angluin and Laid [9]. They gave an algorithm for learning Boolean conjunctions that tolerates random **classification noise** of a rate approaching the information-theoretic barrier of 1/2. Furthermore, they proved that the general technique of finding a hypothesis that minimizes disagreements with a sufficiently large sample allows one to handle random classification noise of any rate approaching 1/2. However, they showed that even the very simple problem of minimizing disagreements (when there are no assumptions about the noise) is NP-hard. Until recently, there have been a small number of efficient noise-tolerant PAC algorithms, but no general techniques were available to design such algorithms, and there was little work to characterize which concept classes could be efficiently learned in the presence of noise.

The first (computationally feasible) tool to design noise-tolerant PAC algorithms was provided by the **statistical query model,** first introduced by Kearns [31], and since improved by Aslam and Decatur [12]. In this model, rather than sampling labeled examples, the learner requests the value of various statistics. A *relative-error statistical query* [12] takes the form $\mathbf{SQ}(\chi, \mu, \theta)$ where χ is a predicate over labeled examples, μ is a *relative error bound,* and θ is a *threshold.* As an example, let χ to be "$(h(x) = 1) \wedge (\ell = 0)$" which is true when x is a negative example but the hypothesis classifies x as positive. So the probability that χ is true for a random example is the false positive error of hypothesis h. For target concept f, we define $P_\chi = \Pr_{\mathcal{D}}[\chi(x, f(x))) = 1]$ where $\Pr_{\mathcal{D}}$ is used to denote that x is drawn at random from distribution $\mathcal{D}$. If $P_\chi < \theta$ then $\mathbf{SQ}(\chi, \mu, \theta)$ may return $\perp$. If $\perp$ is not returned, then $\mathbf{SQ}(\chi, \mu, \theta)$ must return an estimate $\hat{P}_\chi$ such that $P_\chi(1 - \mu) \leq \hat{P}_\chi \leq P_\chi(1 + \mu)$. The learner may also request unlabeled examples (since we are only concerned about classification noise).

Let's take our algorithm, PAC-learn-intersection-of-halfspaces and reformulate it as a relative-error statistical query algorithm. First we draw an *unlabeled* sample S_u that we use to generate the set $\mathcal{T}$ of halfspaces to use for our covering. Similar to before, we place a halfspace in $\mathcal{T}$ corresponding to each possible way in which the points of S_u can be divided. Recall that before, we only place a halfspace in $\mathcal{T}$ if it properly labeled the positive examples. Since we have an unlabeled sample we cannot use such an approach. Instead, we use a statistical query (for each potential halfspace) to check if a given halfspace is consistent with most of the positive examples. Finally, when performing the greedy covering step we cannot pick the halfspace that covers the most negative examples, but rather the one that covers the "most" (based on our empirical estimate) negative weight. Figure 30.5 shows our algorithm in more depth.

We now cover the key ideas in proving that SQ-learn-intersection-of-halfspaces is correct. First, we pick S_u so that any hypothesis consistent with S_u (if we knew the labels) would have error at most $\frac{\epsilon}{6r}$ with probability $1 - \frac{\delta}{2}$. Since our hypothesis class is $\mathcal{C}^{\cap_r}_{\text{halfspace}}$, and $\text{VCD}(\mathcal{C}^{\cap_r}_{\text{halfspace}}) \leq 2(d + 1)r\lg(3r)$, we obtain the sample size for S_u.

For each of the s halfspaces that form the target concept, there is some halfspace in $\mathcal{T}$ consistent with that halfspace over S_u, and thus that has error at most $\epsilon/(6r)$ on the positive region. For each such halfspace t our estimate $\hat{P}_\chi$ (for the probability that $t(x) = 0$ and $\ell = 1$) is at most $\frac{\epsilon}{6r} \cdot \frac{3}{2} = \frac{\epsilon}{4r}$. Thus, we place s halfspaces in $\mathcal{T}_{good}$ that produce a hypothesis with false negative error $\leq \epsilon/(6r)$. Let

SQ-learn-intersection-of-halfspaces(d, s, ϵ)

Let $r = 2s\ln(3/\epsilon)$

Draw an unlabeled sample $S_u = \max\left\{\frac{24r}{\epsilon}\lg\frac{4}{\delta}, \frac{96(d+1)r^2\lg(3r)}{\epsilon}\lg\frac{78r}{\epsilon}\right\}$

Form the set $\mathcal{T}$ of halfspaces - one for each linear separation of S_u into two sets

$\mathcal{T}_{good} = \emptyset$

For each $t \in \mathcal{T}$

$\hat{P_\chi} = \mathbf{SQ}\left(t(x) = 0 \wedge \ell = 1, \frac{1}{2}, \frac{\epsilon}{2r}\right)$

If $\hat{P_\chi} \leq \frac{\epsilon}{4r}$ or $\perp$ Then $\mathcal{T}_{good} = \mathcal{T}_{good} \cup t$

Let h be the always true hypothesis

While $\mathbf{SQ}\left(h(x) = 1 \wedge \ell = 0, \frac{1}{9}, \frac{\epsilon}{2}\right) > \frac{4\epsilon}{9}$ and the loop has been executed $< r$ times

$t_{max} = \text{argmax}_{t \in \mathcal{T}_{good}} \mathbf{SQ}\left(h(x) = 1 \wedge \ell = 0 \wedge t(x) = 0, \frac{1}{3}, \frac{7\epsilon}{30s}\right)$

$h = h \cap t_{max}$

Output h

FIGURE 30.5 A relative-error statistical query algorithm to learn a concept from $\mathcal{C}^{\cap_s}_{\text{halfspace}}$ over $\Re^d$.

$e_i = \Pr[h(x) = 1 \wedge \ell = 0]$ (i.e., the false positive error) after i rounds of the while loop. Since e_i is our current error and $\epsilon/(6r)$ is a lower bound on the error we could achieve, by an averaging argument it follows that there is a $t \in \mathcal{T}_{good}$ for which

$$P = \Pr[(h(x) = 1) \wedge (\ell = 0) \wedge (t(x) = 0)] \geq \left(e_i - \frac{\epsilon}{6r}\right) \cdot \frac{1}{s} \geq \left(e_i - \frac{\epsilon}{6}\right) \cdot \frac{1}{s}.$$

Thus for the halfspace t_{max} selected to add to h we know that the statistical query returns an estimate $\hat{P} \geq \frac{2}{3} \cdot P \geq \frac{2}{3s}\left(e_i - \frac{\epsilon}{6}\right)$. Thus for t_{max} we know that $P \geq \frac{3}{4} \cdot \frac{2}{3s}\left(e_i - \frac{\epsilon}{6}\right) = \frac{1}{2s}\left(e_i - \frac{\epsilon}{6}\right)$. Solving the recurrence $e_{i+1} \leq e_i - \frac{1}{2s}\left(e_i - \frac{\epsilon}{6}\right)$ (with $e_i \leq 1$) yields that $e_i \leq \frac{\epsilon}{6} + \left(1 - \frac{1}{2s}\right)^i$. Picking $r = i = 2s\ln\frac{3}{\epsilon}$ suffices to ensure that $e_i \leq \epsilon/2$ as desired.

Once $e_i \leq 2\epsilon/5$, we exit the while loop (since $\frac{2\epsilon}{5} \cdot \frac{10}{9} = \frac{4\epsilon}{9}$). Thus we enter the loop only when $e_i > 2\epsilon/5$, and hence for t_{max}, $P \geq \frac{1}{s}(e_i - \frac{\epsilon}{6})$. So we choose $\theta = \frac{1}{s}(\frac{2\epsilon}{5} - \frac{\epsilon}{6}) = \frac{7\epsilon}{30s}$. Finally, when we exit the loop $\Pr[h(x) = 1 \wedge \ell = 0] \leq \frac{4\epsilon}{9} \cdot \frac{9}{8} = \frac{\epsilon}{2}$. Thus the total error is at most ϵ as desired.

Notice that in the statistical query model there is not a confidence parameter δ. This is because the SQ oracle guarantees that its estimates meet the given requirements. However, when we use random labeled examples to simulate the statistical queries, we can only guarantee that with high probability the estimates meet their requirements. Thus the results on converting an SQ algorithm into a PAC algorithm reintroduce the confidence parameter δ.

By applying uniform convergence results and Chernoff bounds it can be shown that if one draws a sufficiently large sample then a statistical query can be estimated by the fraction of the sample that satisfies the predicate. For example, in the relative-error SQ model with a set $\mathcal{Q}$ of possible queries, Aslam and Decatur [12] show that a sample of size $O\left(\frac{1}{\mu^2\theta}\log\frac{|\mathcal{Q}|}{\delta}\right)$ suffices to appropriately answer $[\chi, \mu, \theta]$ for every $\chi \in \mathcal{Q}$ with probability at least $1-\delta$. (If $\text{VCD}(\mathcal{Q}) = q$ then a sample of size $O\left(\frac{q}{\mu^2\theta}\log\frac{1}{\mu\theta} + \frac{1}{\mu^2\theta}\log\frac{1}{\delta}\right)$ suffices.)

To handle random classification noise of any rate approaching $1/2$ more complex methods are used for answering the statistical queries. Roughly, using knowledge of the noise process and a sufficiently accurate estimate of the noise rate (which must itself be determined by the algorithm), the noise process can be "inverted." The total sample complexity required to simulate an SQ algorithm in the presence of classification noise of rate $\eta \leq \eta_b$ is $\tilde{O}\left(\frac{\log(|\mathcal{Q}|/\delta)}{\mu_*^2\theta_*\rho(1-2\eta_b)^2}\right)$ where μ_* (respectively, θ_*) is the minimum value of μ (respectively, θ) across all queries and $\rho \in [\theta_*, 1]$, The soft-oh notation ($\tilde{O}$) is similar to the standard big-oh notation except log factors are also removed. Alternatively, for the query space $\mathcal{Q}$, the sample

complexity is $O\left(\frac{\mathrm{VCD}(\mathcal{Q})}{\mu_*^2\theta_*\rho_*(1-2\eta_b)^2}\left[\log\left(\frac{1}{\mu_*\theta_*\rho_*(1-2\eta_b)}\right)+\log\frac{1}{\delta}\right]\right)$. Notice that the amount by which η_b is less than $1/2$ is just $\frac{1}{1/2-\eta_b}=\frac{2}{1-2\eta_b}$. Thus the above are polynomial as long as $\frac{1}{2}-\eta_b$ is at least one over a polynomial.

30.4 Exact and Mistake Bounded Learning Models

The PAC learning model is a *batch model*—there is a separation between the *training phase* and the *performance phase.* In the training phase the learner is presented with labeled examples—no predictions are expected. Then at the end of the training phase the learner must output a hypothesis h to classify unseen examples. Also, since the learner never finds out the true classification for the unlabeled instances, all learning occurs in the training phase. In many settings, the learner does not have the luxury of a training phase but rather must learn as it performs. We now study two closely related learning models designed for such a setting.

On-Line Learning Model

To motivate the **on-line learning model** (also known as the *mistake-bounded learning model*), suppose that when arriving at work (in Boston) you may either park in the street or park in a garage. In fact, between your office building and the garage there is a street on which you can always find a spot. On most days, street parking is preferable since you avoid paying the $15 garage fee. Unfortunately, when parking on the street you risk being towed ($75) due to street cleaning, snow emergency, special events, etc. When calling the city to find out when they tow, you are unable to get any reasonable guidance and decide the best thing to do is just learn from experience. There are many pieces of information that you might consider in making your prediction, e.g., the date, the day of the week, the weather. We make the following assumption: after you commit yourself to one choice or the other you learn of the right decision. In this example, the city has rules dictating when they tow; you just don't know them. If you park on the street, at the end of the day you know if your car was towed; otherwise when walking to the garage you see if the street is clear (i.e., you learn if you would have been towed). The on-line model is designed to study algorithms for learning to make accurate predictions in circumstances such as these. Note that unlike the problems addressed by many techniques from reinforcement learning, here there is immediate (versus delayed) feedback.

We now define the on-line learning model for the general setting in which the target function has a real-valued output (without loss of generality, scaled to be between 0 and 1). An on-line learning algorithm for $\mathcal{C}$ is an algorithm that runs under the following scenario. A *learning session* consists of a set of *trials.* In each trial, the learner is given an unlabeled instance $x \in \mathcal{X}$. The learner uses its current hypothesis to predict a value $p(x)$ for the unknown (real-valued) target concept $f \in \mathcal{C}$ and then the learner is told the correct value for $f(x)$. Several *loss functions* have been considered to measure the quality of the learner's predictions. Three commonly used loss functions are the following: the *square loss* defined by $\ell_2(p(x), f(x)) = (f(x)-p(x))^2$, the *log loss* defined by $\ell_{\log}(p(x), f(x)) = -f(x)\log p(x) - (1-f(x))\log(1-p(x))$, and the *absolute loss* defined by $\ell_1(p(x), f(x)) = |f(x)-p(x)|$.

The goal of the learner is to make predictions so that total loss over all predictions is minimized. In this learning model, most often a worst-case model for the environment is assumed. There is some known concept class from which the target concept is selected. An adversary (with unlimited computing power and knowledge of the learner's algorithm) then selects both the target function and the presentation order for the instances. In this model there is no training phase. Instead, the learner receives *unlabeled instances* throughout the entire learning session. However, after each prediction the learner "discovers" the correct value. This feedback can then be used by the learner to improve its hypothesis.

We now discuss the special case when the target function is Boolean and correspondingly, predictions must be either 0 or 1. In this special case the loss function most commonly used is the absolute loss.

Notice that if the prediction is correct then the value of the loss function is 0, and if the prediction is incorrect then the value of the loss function is 1. Thus the total loss of the learner is exactly the number of prediction mistakes. Thus, in the worst-case model we assume that an adversary selects the order in which the instances are presented to the learner and we evaluate the learner by the maximum number of mistakes made during the learning session. Our goal is to minimize the worst-case number of mistakes using an efficient learning algorithm (i.e., each prediction is made in polynomial time). Observe that such mistake bounds are quite strong in that the order in which examples are presented does not matter; however, it is impossible to tell how early the mistakes will occur. Littlestone [37] has shown that in this learning model $\mathrm{VCD}(\mathcal{C})$ is a lower bound on the number of prediction mistakes.

Handling Irrelevant Attributes

Here we consider the common scenario in which there are many attributes the learner could consider, yet the target concept depends on a small number of them. Thus most of the attributes are *irrelevant* to the target concept. We now briefly describe one early algorithm, Winnow of [37], that handles a large number of irrelevant attributes. More specifically, for Winnow, the number of mistakes only grows logarithmically with the number of irrelevant attributes. Winnow (or modifications of it) have many nice features such as noise tolerance and the ability to handle the situation in which the target concept is changing. Also, Winnow can directly be applied to the *agnostic learning model* [35]. In the agnostic learning model no assumptions are made about the target concept. In particular, the learner is unaware of any concept class that contains the target concept. Instead, we compare the performance of an agnostic algorithm (typically in terms of the number of mistakes or based on some other loss function) to the performance of the best hypothesis selected from a comparison or "touchstone" class where the best hypothesis from the touchstone class is the one that incurs the minimum loss over all functions in the touchstone class.

Winnow is similar to the classical perceptron algorithm [45], except that it uses a multiplicative weight-update scheme that permits it to perform much better than classical perceptron training algorithms when many attributes are irrelevant. The basic version of Winnow is designed to learn the concept class of a *linearly separable Boolean function,* which is a map $f : \{0, 1\}^n \to \{0, 1\}$ such that there exists a hyperplane in $\Re^n$ that separates the inverse images $f^{-1}(0)$ and $f^{-1}(1)$ (i.e., the hyperplane separates the points on which the function is 1 from those on which it is 0). An example of a linearly separable function is any monotone disjunction: if $f(x_1,\ldots,x_n) = x_{i_1} \vee \cdots \vee x_{i_k}$, then the hyperplane $x_{i_1} + \cdots + x_{i_k} = 1/2$ is a separating hyperplane. For each attribute x_i there is an associated weight w_i. There are two parameters, θ which determines the threshold for predicting 1 (positive), and α which determines the adjustment made to the weight of an attribute that was partly responsible for a wrong prediction. The pseudo-code for Winnow is shown in Fig. 30.6.

Winnow(θ, α)
For $i = 1, \ldots, n$, initialize $w_i = 1$

To predict the value of $x = (x_1, \ldots, x_n) \in X$:
If $\sum_{i=1}^{n} w_i x_i > \theta$, then predict 1
Else predict 0

Let ρ be the correct prediction
If the prediction was 1 and $\rho = 0$ then
If $x_i = 1$ then let $w_i = w_i/\alpha$
If the prediction was 0 and $\rho = 1$ then
If $x_i = 1$ then let $w_i = w_i \cdot \alpha$

FIGURE 30.6 The algorithm Winnow.

Winnow has been successfully applied to many learning problems. It has the very nice property that one can prove that the number of mistakes only grows logarithmically in the number of variables (and linearly in the number of relevant variables). For example, it can be shown that for the learning of monotone disjunctions of at most k literals, if Winnow is run with $\alpha > 1$ and $\theta \geq 1/\alpha$, then the total number of mistakes is at most $\alpha k(\log_\alpha \theta + 1) + n/\theta$. Observe that Winnow need not have prior knowledge of k although the number of mistakes depends on k. Littlestone [37] showed how to optimally choose θ and α if an upperbound on k is known a priori. Also, while Winnow is designed to learn a linearly separable class, reductions (discussed in "Prediction-Preserving Reductions") can be used to apply Winnow to classes for which the positive and negative points are not linearly separable, e.g., k-DNF.

The Halving Algorithm and Weighted Majority Algorithm

We now discuss some of the key techniques for designing good on-line learning algorithms for the special case of concept learning (i.e., learning Boolean-valued functions). If one momentarily ignores the issue of computation time, then the *halving algorithm* [37] performs very well. It works as follows. Initially all concepts in the concept class $\mathcal{C}$ are candidates for the target concept. To make a prediction for instance x, the learner takes a majority vote based on all remaining candidates (breaking a tie arbitrarily). Then when the feedback is received, all concepts that disagree are removed from the set of candidates. It can be shown that at each step the number of candidates is reduced by a factor of at least 2. Thus, the number of prediction mistakes made by the halving algorithm is at most $\lg|\mathcal{C}|$.

Clearly, the halving algorithm will perform poorly if the data is noisy. We now briefly discuss the *weighted majority algorithm* [38], which is one of several multiplicative weight-update schemes for generalizing the halving algorithm to tolerate noise. Also, the weighted majority algorithm provides a simple and effective method for constructing a learning algorithm A that is provided with a pool of "experts," one of which is known to perform well, but A does not know which one. Associated with each expert is a weight that gives A's confidence in the accuracy of that expert. When asked to make a prediction, A predicts by combining the votes from its experts based on their associated weights. When an expert suggests the wrong prediction, A passes that information to the given expert and reduces its associated weight using a multiplicative weight-updating scheme. Namely, the weight associated with each expert that mispredicts is multiplied by some weight $0 \leq \beta < 1$. By selecting $\beta > 0$ this algorithm is robust against noise in the data. Figure 30.7 shows the weighted majority algorithm in more depth.

Weighted Majority Algorithm (WM)
Let w_i be the weight associated with algorithm A_i
For $1 \leq i \leq n$
Let $w_i = 1$
To make a prediction for an instance $x \in X$:
Let $\mathcal{A}_0$ be the set of algorithms that predicted 0
Let $\mathcal{A}_1$ be the set of algorithms that predicted 1
Let $q_0 = \sum_{A_i \in \mathcal{A}_0} w_i$
Let $q_1 = \sum_{A_i \in \mathcal{A}_1} w_i$
Predict 0 iff $q_0 \geq q_1$
If a mistake is made:
Let γ be the prediction
For $1 \leq i \leq n$
If A_i's prediction for x is γ
Let $w_i = w_i \cdot \beta$ (where $0 \leq \beta < 1$)
Inform A_i of the mistake

FIGURE 30.7 The weighted majority algorithm.

We now briefly discuss some learning problems in which weighted majority is applicable. Suppose one knows that the correct prediction comes from some target concept selected from a known concept class. Then one can apply the weighted majority algorithm where each concept in the class is one of the algorithms in the pool. For such situations, the weighted majority algorithm is a robust generalization of the halving algorithm. (In fact, the halving algorithm corresponds to the special case where $\beta = 0$.) As another example, the weighted majority algorithm can often be applied to help in situations in which the prediction algorithm has a parameter that must be selected and the best choice for the parameter depends on the target. In such cases one can build the pool of algorithms by choosing various values for the parameter.

We now describe some of the results known about the performance of the weighted majority algorithm. If the best algorithm in the pool $\mathcal{A}$ makes at most m mistakes, then the worst case number of mistakes made by the weighted majority algorithm is $O(\log |\mathcal{A}| + m)$ where the constant hidden within the big-oh notation depends on β. Specifically, the number of mistakes made by the weighted majority algorithm is at most $\frac{\log |\mathcal{A}| + m \log \frac{1}{\beta}}{\log \frac{2}{1+\beta}}$ if one algorithm in $\mathcal{A}$ makes at most m mistakes, $\frac{\log \frac{|\mathcal{A}|}{k} + m \log \frac{1}{\beta}}{\log \frac{2}{1+\beta}}$ if there exists a set of k algorithms in $\mathcal{A}$ such that each algorithm makes at most m mistakes, and $\frac{\log \frac{|\mathcal{A}|}{k} + \frac{m}{k} \log \frac{1}{\beta}}{\log \frac{2}{1+\beta}}$ if the total number of mistakes made by a set of k algorithms in $\mathcal{A}$ is m.

When $|\mathcal{A}|$ is not polynomial, the weighted majority algorithm (when directly implemented), is not computationally feasible. Recently, Mass and Warmuth [40] introduced what they call the *virtual weight technique* to implicitly maintain the exponentially large set of weights so that the time to compute a prediction and then update the "virtual" weights is polynomial. More specifically, the basic idea is to simulate Winnow by grouping concepts that "behave alike" on seen examples into blocks. For each block only one weight has to be computed and one constructs the blocks so that the number of concepts combined in each block as well as the weight for the block can be efficiently computed. While the number of blocks increases as new **counterexamples** are received, the total number of blocks is polynomial in the number of mistakes. Thus all predictions and updates can be performed in time polynomial in the number of blocks, which is in turn polynomial in the number of prediction mistakes of Winnow. Many variations of the basic weighted majority algorithm have also been studied. The results of Cesa-Bianchi et al. [19] demonstrate how to tune β as a function of an upper bound on the noise rate.

Query Learning Model

A very well-studied formal learning model is the membership and equivalence query model developed by Angluin [3]. In this model (often called the *exact learning model*) the learner's goal is to learn *exactly* how an unknown (Boolean) target function f, taken from some known concept class $\mathcal{C}$, classifies all instances from the domain. This goal is commonly referred to as *exact identification.* The learner has available two types of queries to find out about f: one is a **membership query,** in which the learner supplies an instance x from the domain and is told $f(x)$. The other query provided is an **equivalence query** in which the learner presents a candidate function h and either is told that $h \equiv f$ (in which case learning is complete), or else is given a **counterexample** x for which $h(x) \neq f(x)$. There is a very close relationship between this learning model and the on-line learning model (supplemented with membership queries) when applied to a classification problem. Using a standard transformation [3, 37], algorithms that use membership and equivalence queries can easily be converted to on-line learning algorithms that use membership queries. Under such a transformation the number of counterexamples provided to the learner in response to the learner's equivalence queries directly corresponds to the number of mistakes made by the on-line algorithm.

In this model a number of interesting polynomial time algorithms are known for learning deterministic finite automata [2], Horn sentences [5], read-once formulas [6], read-twice DNF formulas [1], decision trees [18], and many others. It is easily shown that membership queries alone are not sufficient for efficient

learning of these classes, and Angluin has developed a technique of "approximate fingerprints" to show that equivalence queries alone are also not enough [4]. (In both cases the arguments are information theoretic, and hold even when the computation time is unbounded.) The work of Bshouty et al. [17] extended Angluin's results to establish tight bounds on how many equivalence queries are required for a number of these classes. Maass and Turán studied upper and lower bounds on the number of equivalence queries required for learning (when computation time is unbounded), both with and without membership queries [39].

It is known that any class learnable exactly from equivalence queries can be learned in the PAC setting [3]. At a high level the exact learning algorithm is transformed to a PAC algorithm by having the learner use random examples to "search" for a counterexample to the hypothesis of the exact learner. If a counterexample is found, it is given as a response to the equivalence query. Furthermore, if a sufficiently large sample was drawn and no counterexample was found then the hypothesis has error at most ϵ (with probability at least $1 - \delta$). The converse does not hold [13]. That is, there are concept classes that are efficiently PAC learnable but cannot be efficiently learned in the exact model.

We now describe Angluin's algorithm for learning monotone DNF formulas (see Fig. 30.8). A *prime implicant* t of a formula f is a satisfiable conjunction of literals such that t implies f but no proper subterm of t implies f. For example, $f = (a \wedge c) \vee (b \wedge \bar{c})$ has prime implicants $a \wedge c$, $b \wedge \bar{c}$, and $a \wedge b$. The number of prime implicants of a general DNF formula may be exponentially larger than the number of terms in the formula. However, for monotone DNF the number of prime implicants is no greater than the number of terms in the formula. The key to the analysis is to show that at each iteration, the term t is a new prime implicant of f, the target concept. Since the loop iterates exactly once for each prime implicant there are at most m counterexamples where m is the number of terms in the target formula. Since at most n membership queries are performed during each iteration there are at most nm membership queries overall.

```
Learn-Monotone-DNF
    h ← false
    Do forever
        Make equivalence query with h
        If "yes," output h and halt
        Else let x = b_1 b_2 ··· b_n be the counterexample
            Let t = ⋀_{b_i=1} y_i
            For i = 1, ..., n
                If b_i = 1 perform membership query on x with i^th bit flipped
                If "yes," t = t \ {y_i} and x = b_1 ··· b̄_i ··· b_n
            Let h = h ∨ t
```

FIGURE 30.8 An algorithm that uses membership and equivalence queries to exactly learn an unknown monotone DNF formula over the domain $\{0, 1\}^n$. Let $\{y_1, y_2, \ldots, y_n\}$ be the Boolean variables and let h be the learner's hypothesis.

30.5 Hardness Results

In order to understand what concept classes are learnable, it is essential to develop techniques to prove when a learning problem is hard. Within the learning theory community, there are two basic type of hardness results that apply to all of the models discussed here. There are **representation-dependent hardness results** in which one proves that one cannot efficiently learn $\mathcal{C}$ using a hypothesis class of

$\mathcal{H}$. These hardness results typically rely on some complexity theory assumption[8] such as RP $\neq$ NP. For example, given that RP $\neq$ NP, Pitt and Warmuth [43] showed that k-term-DNF is not learnable using the hypothesis class of k-term-DNF. While such results provide some information, what one would really like to obtain is a hardness result for learning a concept class using any reasonable (i.e., polynomially evaluatable) hypothesis class. (For example, while we have a representation-dependent hardness result for learning k-term-DNF, there is a simple algorithm to PAC learn the class of k-term-DNF formulas using a hypothesis class of k-CNF formulas.) **Representation-independent hardness results** meet this more stringent goal. However, they depend on cryptographic (versus complexity theoretic) assumptions. Kearns and Valiant [32] gave representation-independent hardness results for learning several concept classes such as Boolean formulas, deterministic finite automata, and constant-depth threshold circuits (a simplified form of "neural networks"). These hardness results are based on assumptions regarding the intractability of various cryptographic schemes such as factoring Blum integers and breaking the RSA function.

Prediction-Preserving Reductions

Given that we have some representation-independent hardness result (assuming the security of various cryptographic schemes) one would like a "simple" way to prove that other problems are hard in a similar fashion as one proves a desired algorithm is intractable by reducing a known NP-complete problem to it.[9] Such a complexity theory for predictability has been provided by Pitt and Warmuth [43]. They formally define a *prediction preserving reduction* from concept class $\mathcal{C}$ over domain $\mathcal{X}$ to concept class $\mathcal{C}'$ over domain $\mathcal{X}'$ (denoted by $\mathcal{C} \leq \mathcal{C}'$) that allows an efficient learning algorithm for $\mathcal{C}'$ to be used to obtain any efficient learning algorithm for $\mathcal{C}$. The requirements for such a prediction-preserving reduction are (1) an efficient *instance transformation* g from $\mathcal{X}$ to $\mathcal{X}'$, and (2) the existence of an *image concept.* The instance transformation g must be polynomial time computable. Hence, if $g(x) = x'$ then the size of x' must be polynomially related to the size of x. So for $x \in \mathcal{X}_n$, $g(x) \in \mathcal{X}_{p(n)}$ where $p(n)$ is some polynomial function of n. It is also important that g be independent of the target function. We now define what is meant by the existence of an image concept. For every $f \in \mathcal{C}_n$ there must exist some $f' \in \mathcal{C}'_{p(n)}$ such that for all $x \in \mathcal{X}_n$, $f(x) = f'(g(x))$ and the number of bits to represent f' is polynomially related to the number of bits to represent f.

As an example, let $\mathcal{C}$ be the class of DNF formulas over $\mathcal{X} = \{0,1\}^n$, and $\mathcal{C}'$ be the class of monotone DNF formulas over $\mathcal{X}' = \{0,1\}^{2n}$. We now show that $\mathcal{C} \leq \mathcal{C}'$. Let $y_1, \ldots, y_n$ be the variables for each concept from $\mathcal{C}$. Let $y'_1, \ldots, y'_{2n}$ be the variables for $\mathcal{C}'$. The intuition behind the reduction is that variable y_i for the DNF problem is associated with variable y_{2i-1} for the monotone DNF problem. And variable $\overline{y_i}$ for the DNF problem is associated with variable y_{2i} for the monotone DNF problem. So for example $x = b_1 b_2 \cdots b_n$ where each $b_i \in \{0,1\}$ and $g(x) = b_1(1-b_1)b_2(1-b_2)\cdots b_n(1-b_n)$. Given a target concept $f \in \mathcal{C}$, the image concept f' is obtained by replacing each nonnegated variable y_i from f with y'_{2i-1} and each negated variable $\overline{y_i}$ from f with y'_{2i}. It is easily confirmed that all required conditions are met.

If $\mathcal{C} \leq \mathcal{C}'$, what implications are there with respect to learnability? Observe that if we are given a polynomial prediction algorithm A' for $\mathcal{C}'$, one can use A' to obtain a polynomial prediction algorithm A for $\mathcal{C}$ as follows. If A' requests a labeled example then A can obtain a labeled example $x \in \mathcal{X}$ from its oracle and give $g(x)$ to A'. Finally, when A' outputs hypothesis h', A can make a prediction for $x \in \mathcal{X}$ using $h(g(x))$. Thus if $\mathcal{C}$ is known not to be learnable then $\mathcal{C}'$ also is not learnable. Thus the reduction given above implies that the class of monotone DNF formulas is just as hard to learn in the PAC model as

[8] For background on complexity classes see Chapter 33 (NP defined) and Chapter 35 (RP defined) of this *Handbook.*
[9] See Chapter 34 of this *Handbook* for background on reducibility and completeness.

the class of arbitrary DNF formulas. Equivalently, if there were an efficient PAC algorithm for the class of monotone DNF formulas, then there would also be an efficient PAC algorithm for arbitrary DNF formulas.

Pitt and Warmuth [43] gave a prediction preserving reduction from the class of Boolean formulas to class of DFAs. Thus since Kearns and Valiant [32] proved that Boolean formula are not predictable (under cryptographic assumptions), it immediately follows that DFAs are not predictable. In other words, DFAs cannot be efficiently learned from random examples alone. Recall that any algorithm to exactly learn using only equivalence queries can be converted into an efficient PAC algorithm. Thus if DFAs are not efficiently PAC learnable (under cryptographic assumptions), it immediately follows that DFAs are not efficiently learnable from only equivalence queries (under cryptographic assumptions). Contrasting this negative result, recall that DFAs are exactly learnable from membership and equivalence queries [2], and thus are PAC learnable with membership queries.

Notice that for $\mathcal{C} \leq \mathcal{C}'$, the result that an efficient learning algorithm for $\mathcal{C}'$ also provides an efficient algorithm for $\mathcal{C}$ relies heavily on the fact that membership queries are NOT allowed. The problem created by membership queries (whether in the PAC or exact model) is that algorithm A' for $\mathcal{C}'$ may make a membership query on an example $x' \in \mathcal{X}'$ for which $g^{-1}(x')$ is not in $\mathcal{X}$. For example, the reduction described earlier demonstrates that if there is an efficient algorithm to PAC learn monotone DNF formulas then there is an efficient algorithm to PAC learn DNF formulas. Notice that we already have an algorithm Learn-Monotone-DNF that exactly learns this class with membership and equivalence queries (and thus can PAC learn the class when provided with membership queries). Yet, the question of whether or not there is an algorithm with access to a membership oracle to PAC learn DNF formulas remains one of the biggest open questions within the field. We now describe the problem with using the algorithm Learn-Monotone-DNF to obtain an algorithm for general DNF formulas. Suppose that we have 4 Boolean variables (thus the domain is $\{0, 1\}^4$). The algorithm Learn-Monotone-DNF could perform a membership query on the example 00100111. While this example is in the domain $\{0, 1\}^8$ there is no $x \in \{0, 1\}^4$ for which $g(x) = 00100111$. Thus the provided membership oracle for the DNF problem cannot be used to respond to the membership query posed by Learn-Monotone-DNF.

We now define a more restricted type of reduction $\mathcal{C} \leq_{wmq} \mathcal{C}'$ that yields results even when membership queries are allowed [7]. For these reductions, we just add the following third condition to the two conditions already described: for all $x' \in \mathcal{X}'$, if x' is not in the image of g (i.e., there is no $x \in \mathcal{X}$ such that $g(x) = x'$), then the classification of x' for the image concept must always be positive or always be negative. As an example, we show that $\mathcal{C} \leq_{wmq} \mathcal{C}'$ where $\mathcal{C}$ is the class of DNF formulas and $\mathcal{C}'$ is the class of read-thrice DNF formulas (meaning that each literal can appear at most three times). Thus learning a read-thrice DNF formula (even with membership queries) is as hard as learning general DNF formula (with membership queries). Let s be the number of literals in f. (If s is not known *a priori,* the standard doubling technique can be applied.) The mapping g maps from an $x \in \{0, 1\}^n$ to an $x' \in \{0, 1\}^{sn}$. More specifically, $g(x)$ simply repeats, s times, each bit of x. To see that there is an image concept, note that we have s variables for each concept in $\mathcal{C}'$ associated with each variable for a concept in $\mathcal{C}$. Thus we can rewrite $f \in \mathcal{C}$ as a formula $f' \in \mathcal{C}'$ in which each variable only appears once. At this point we have shown $\mathcal{C} \leq \mathcal{C}'$ but still need to do more to satisfy the new third condition. We want to ensure that the s variables (call them $\ell'_1, \ldots, \ell'_s$) for f' associated with one literal ℓ_i of f all take the same value. We do this by defining our final image concept as: $f' \wedge \Gamma_1 \wedge \cdots \wedge \Gamma_n$ where n is the number of variables and Γ_i is of the form $(\ell'_1 \rightarrow \ell'_2) \wedge \cdots \wedge (\ell'_{s-1} \rightarrow \ell'_s) \wedge (\ell'_s \rightarrow \ell'_1)$. This formula evaluates to true exactly when $\ell'_1, \ldots, \ell'_s$ have the same value. Thus an x' for which no $g(x) = x'$ will not satisfy some Γ_i and thus we can respond "no" to a membership query on x'. Finally, f' is a read-thrice DNF formula. This reduction also proves that Boolean formulas $\leq_{wmq}$ read-thrice Boolean formulas.

30.6 Weak Learning and Hypothesis Boosting

As originally defined, the PAC model requires the learner to produce, with arbitrarily high probability, a hypothesis that is arbitrarily close to the target concept. While for many problems it is easy to find simple

algorithms ("rules-of-thumb") that are often correct, it seems much harder to find a single hypothesis that is highly accurate. Informally, a **weak learning** algorithm is one that outputs a hypothesis that has some advantage over random guessing. (To contrast this sometimes a PAC learning algorithm is called a *strong learning* algorithm.) This motivates the question: are there concept classes for which there is an efficient weak learner, but there is no efficient PAC learner? Somewhat surprisingly the answer is no [46]. The technique used to prove this result is to take a weak learner and transform (boost) it into a PAC learner. The general method of converting a rough rule-of-thumb (weak learner) into a highly accurate prediction rule (PAC learner) is referred to as **hypothesis boosting**.[10]

We now formally define the weak learning model [32, 46]. As in the PAC model, there is the instance space $\mathcal{X}$ and the concept class $\mathcal{C}$. Also the examples are drawn randomly and independently according to a fixed but unknown distribution $\mathcal{D}$ on $\mathcal{X}$. The learner's hypothesis h must be a polynomial time function that given an $x \in \mathcal{X}$ returns a prediction of $f(x)$ for $f \in \mathcal{C}$, the unknown target concept. The accuracy requirements for the hypothesis of a weak learner are as follows. There is a polynomial function $p(n)$ and algorithm A such that, for all $n \geq 1$, $f \in \mathcal{C}_n$, for all distributions $\mathcal{D}$, and for all $0 < \delta \leq 1$, algorithm A, given n, δ, and access to labeled examples from $\mathcal{D}$, outputs a hypothesis h such that, with probability $\geq 1 - \delta$, $\text{error}_D(h)$ is at most $(1/2 - 1/p(n))$. Algorithm A should run in time polynomial in n and $1/\delta$.

If $\mathcal{C}$ is strongly learnable, then it is weakly learnable—just fix $\epsilon = 1/4$ (or any constant less than $1/2$). The converse (weak learnability implying strong learnability) is not at all obvious. In fact, if one restricts the distributions under which the weak learning algorithm runs then weak learnability *does not* imply strong learnability. Namely, Kearns and Valiant [32] have shown that under a uniform distribution, monotone Boolean functions are weakly, but not strongly, learnable. Thus it is important to take advantage of the requirement that the weak learning algorithm must work for all distributions. Using this property, Schapire [46] proved the converse result: if concept class $\mathcal{C}$ is weakly learnable, then it is strongly learnable.

Proving that weak learnability implies strong learnability has also been called the *hypothesis boosting problem,* because a way must be found to boost the accuracy of slightly-better-than-half hypothesis to be arbitrarily close to 1. There have been several boosting algorithms proposed since the original work of Schapire [46]. Figure 30.9 describes AdaBoost [23] that has shown promise in empirical use. The key to forcing the weak learner to output hypothesis that can be combined to create a highly accurate hypothesis is to create different distributions on which the weak learner is trained.

Freund and Schapire [23] have done some experiments showing that by applying AdaBoost to some simple rules of thumb or using C4.5 [44] as the weak learner they can perform better than previous algorithms on some standard benchmarks. Also, Freund and Schapire have presented a more general version for AdaBoost for the situation in which the predictions are real-valued (versus binary). Some of the practical advantages of AdaBoost are that is fast, simple and easy to program, requires no parameters to tune (besides T, the number of rounds), no prior knowledge is needed about the weak learner, it is provably effective, and it is flexible, since you can combine it with any classifier that finds weak hypothesis.

30.7 Research Issues and Summary

In this chapter, we have described the fundamental models and techniques of computational learning theory. There are many interesting research directions besides those discussed here. One general direction of research is in defining new, more realistic models, and models that capture different learning scenarios. Here are a few examples. Often, as in the problem of weather prediction, the target concept is probabilistic in nature. Thus Kearns and Schapire [34] introduced the p-concepts model. Also there has been a lot of

[10] We note that one can easily boost the confidence δ by first designing an algorithm A that works for say $\delta = 1/2$ and then running A several times taking a majority vote. For an arbitrary $\delta > 0$ the number of runs of A needed are polynomial in $\lg 1/\delta$.

AdaBoost

For $i = 1, \ldots, m$

$D_1(i) = 1/m$

For $t = 1, \ldots, T$

Call WeakLearn providing it with the distribution D_t to obtain the hypothesis h_t

Calculate the error of h_t : $\epsilon_t = \sum_{i:h_t(x_i) \neq y_i} D_t(i)$

If $\epsilon_t > 1/2$, set $T = t - 1$ and abort the loop

Let $\beta_t = \epsilon_t/(1 - \epsilon_t)$

Update distribution D_t:

Let Z_t be a normalization constant so D_{t+1} is a valid probability distribution

If $h_t(x_i) = y_i$

Let $D_{t+1} = \frac{D_t(i)}{Z_t} \cdot \beta_t$

Else

Let $D_{t+1} = \frac{D_t(i)}{Z_t}$

Output the final hypothesis:

$$h_{fin}(x) = \arg\max_{y \in Y} \sum_{t:h_t(x)=y} \log \frac{1}{\beta_t}$$

FIGURE 30.9 The procedure AdaBoost to boost the accuracy of a mediocre hypothesis (created by WeakLearn) to a very accurate hypothesis. The input is a sequence $\langle (x_1, y_1), \ldots, (x_m, y_m) \rangle$ of labeled examples where each label comes from the set $Y = \{1, \ldots, k\}$.

work in extending the VC theory to real-valued domains (e.g.,[28]). In both the PAC and on-line models, many algorithms use membership queries. While most work has assumed that the answers provided to the membership queries are reliable, in reality a learner must be able to handle inconclusive and noisy results from the membership queries. See Blum et al. [14] for a summary of several models introduced to address this situation. As one last example, there has been much work recently in exploring models of a "helpful teacher," since teaching is often used to assist human learning (e.g.,[8, 25]).

Finally, there has been work to bridge the computational learning research with the research from other fields such as neural networks, natural language processing, DNA analysis, inductive logic programming, information retrieval, expert systems, and many others.

30.8 Defining Terms

Classification noise: A model of noise in which the label may be reported incorrectly. In the *random classification noise model,* with probability η the label is inverted. In the *malicious classification noise* model, with probability η an adversary can choose the label. In both models with probability $1 - \eta$ the example is not corrupted. The quantity η is referred to as the *noise rate.*

Concept class: A set of rules from which the target function is selected.

Counterexample: A counterexample x to a hypothesis h (where the target concept is f) is either an example for which $f(x) = 1$ and $h(x) = 0$ (a *positive* counterexample) or for which $f(x) = 0$ and $h(x) = 1$ (a *negative* counterexample).

Equivalence query: A query to an oracle in which the learner provides a hypothesis h and is either told that h is logically equivalent to the target or otherwise given a counterexample.

Hypothesis boosting: The process of taking a weak learner that predicts correctly just over half of the time and transforming (boosting) it into a PAC (strong) learner whose predictions are as accurate as desired.

Hypothesis class: The class of functions from which the learner's hypothesis is selected.

Membership query: The learner supplies the membership oracle with an instance x from the domain and is told the value of $f(x)$.

Occam algorithm: An algorithm that draws a sample S and then outputs a hypothesis consistent with the examples in S such that the size of the hypothesis is sublinear in S (i.e., it performs some data compression).

On-line learning model: The learning session consists of a set of trials where in each trial, the learner is given an unlabeled instance $x \in \mathcal{X}$. The learner uses its current hypothesis to predict a value $p(x)$ for the unknown (real-valued) target and is then told the correct value for $f(x)$. The performance of the learner is measured in terms of the sum of the loss over all predictions.

PAC learning: This is a batch model in which first there is a training phase in which the learner sees examples drawn randomly from an unknown distribution $\mathcal{D}$, and labeled according to the unknown target concept f drawn from a known concept class $\mathcal{C}$. The learner's goal is to output a hypothesis that has error at most ϵ with probability at least $1 - \delta$. (The learner receives ϵ and δ as inputs.) In the *proper* PAC learning model the learner must output a hypothesis from $\mathcal{C}$. In the *nonproper* PAC learning model the learner can output any hypothesis h for which $h(x)$ can be computed in polynomial time.

Representation-dependent hardness result: A hardness result in which one proves that one cannot efficiently learn $\mathcal{C}$ using a hypothesis class of $\mathcal{H}$. These hardness results typically rely on some complexity theory assumption such as RP $\neq$ NP.

Representation-independent hardness result: A hardness result in which no assumption is made about the hypothesis class used by the learner (except the hypothesis can be evaluated in polynomial time). These hardness results typically rely on cryptographic assumptions such as the difficulty of breaking RSA.

Statistical query model: A learning model in which the learner does not have access to labeled examples, but rather only can ask queries about statistics about the labeled examples and receive unlabeled examples. Any statistical query algorithm can be converted to a PAC algorithm that can handle random classification noise (as well as some other types of noise).

Vapnik–Chervonenkis (VC) dimension: A finite set $S \subseteq \mathcal{X}$ is *shattered* by $\mathcal{C}$ if for each of the $2^{|S|}$ subsets $S' \subseteq S$, there is a concept $f \in \mathcal{C}$ that contains all of S' and none of $S - S'$. In other words, for any of the $2^{|S|}$ possible labelings of S (where each example $s \in S$ is either positive or negative), there is some $f \in \mathcal{C}$ that realizes the desired labeling (see Fig. 30.2). The **Vapnik–Chervonenkis dimension** of $\mathcal{C}$, denoted as $\mathrm{VCD}(\mathcal{C})$, is the smallest d for which no set of $d + 1$ examples is shattered by $\mathcal{C}$. Equivalently, $\mathrm{VCD}(\mathcal{C})$ is the cardinality of the largest finite set of points $S \subseteq X$ that is shattered by $\mathcal{C}$.

Weak learning model: A variant of the PAC model in which the learner is only required to output a hypothesis that with probability $\geq 1 - \delta$, has error at most $(1/2 - 1/p(n))$. I.e., it does noticeably better than random guessing. Recall in the standard PAC (strong learning) model the learner must output a hypothesis with arbitrarily low error.

References

[1] Aizenstein, H. and Pitt, L., Exact learning of read-twice DNF formulas. In *Proc. 32th Annu. IEEE Sympos. Found. Comput. Sci.,* IEEE Computer Society Press, 170–179, 1991.

[2] Angluin, D., Learning regular sets from queries and counterexamples. *Inform. Comput.,* 75(2), 87–106, Nov. 1987.

[3] Angluin, D., Queries and concept learning. *Machine Learning,* 2(4), 319–342, Apr. 1988.

[4] Angluin, D., Negative results for equivalence queries. *Machine Learning,* 5, 121–150, 1990.

[5] Angluin, D., Frazier, M., and Pitt, L., Learning conjunctions of Horn clauses. *Machine Learning,* 9, 147–164, 1992.
[6] Angluin, D., Hellerstein, L., and Karpinski, M., Learning read-once formulas with queries. *J. ACM.* 40, 185–210, 1993.
[7] Angluin, D. and Kharitonov, When won't membershipt queries help? *J. Comput. Syst. Sci.,* 50(2), 336–355, 1995.
[8] Angluin, D. and Kriķis, M., Teachers, learners and black boxes. In *Proc. 10th Annu. Conf. on Comput. Learning Theory.* ACM Press, New York, 285–297, 1997.
[9] Angluin, D. and Laird, P., Learning from noisy examples. *Machine Learning,* 2(4), 343–370, 1988.
[10] Angluin, D. and Smith, C., Inductive inference. In *Encyclopedia of Artificial Intelligence,* John Wiley & Sons, New York, 409–418, 1987.
[11] Anthony, M. and Biggs, N., *Computational Learning Theory, Cambridge Tracts in Theoretical Computer Science (30).* Cambridge Tracts in Theoretical Computer Science (30). Cambridge University Press, 1992.
[12] Aslam, J.A. and Decatur, S.E., Specification and simulation of statistical query algorithms for efficiency and noise tolerance. *J. Comp. Syst. Sci.,* 1998. To appear.
[13] Blum, A., Separating distribution-free and mistake-bound learning models over the Boolean domain. *SIAM J. Computing,* 23(5), 990–1000, 1994.
[14] Blum, A., Chalasani, P., Goldman, S.A., and Slonim, D.K., Learning with unreliable boundary queries. In *Proc. 8th Annu. Conf. on Comput. Learning Theory.* 1995. ACM Press, New York, 98–107, 1995. To appear in COLT '95 Special Issue of *J. Comput. Syst. Sci.*
[15] Blumer, A., Ehrenfeucht, A., Haussler, D., and Warmuth, M.K., Occam's razor. *Inform. Proc. Lett.,* 24, 377–380, 1987.
[16] Blumer, A., Ehrenfeucht, A., Haussler, D., and Warmuth, M.K., Learnability and the Vapnik-Chervonenkis dimension. *J. ACM.* 36(4), 929–965, 1989.
[17] Bshouty, N., Goldman, S., Hancock, T., and Matar, S., Asking questions to minimize errors. *J. Comp. Syst. Sci.,* 52(2), 268–286, Apr. 1996.
[18] Bshouty, N.H. and Mansour, Y.,. Simple learning algorithms for decision trees and multivariate polynomials. In *Proceedings of the 36th Annual Symposium on Foundations of Computer Science,* 1995. IEEE Computer Society Press, Los Alamitos, CA, 304–311, 1995.
[19] Cesa-Bianchi, N., Freund, Y., Helmbold, D.P., Haussler, D., Schapire, R.E., and Warmuth, M.K., How to use expert advice. *J. ACM,* 44(3), 427–485, 1997.
[20] Devroye, L., Györfi, L., and Lugosi, G., *A Probabalistic Theory of Pattern Recognition, Applications of Mathematics.* Applications of Mathematics. Springer-Verlag, New York, 1996.
[21] Duda, R.O. and Hart, P.E., *Pattern Classification and Scene Analysis.* John Wiley & Sons, 1973.
[22] Ehrenfeucht, A. and Haussler, D., A general lower bound on the number of examples needed for learning. *Inform. Comput.,* 82(3), 247–251, Sep. 1989.
[23] Freund, Y. and Schapire, R., Experiments with a new boosting algorithm. In *Proceedings of the Thirteenth International Conference on Machine Learning,* 1996. Morgan Kaufmann, 148–156, 1996.
[24] Gold, E.M., Language identification in the limit. *Inform. Control,* 10, 447–474, 1967.
[25] Goldman, S. and Mathias, D., *J. Comput. Syst. Sci.,* (52)2, 255–267, Apr. 1996.
[26] Goldman, S. and Sloan, R., Can PAC learning algorithms tolerate random attribute noise? *Algorithmica,* 14(1), 70–84, Jul. 1995.
[27] Goldman, S.A. and Scott, S.D., A theoretical and empirical study of a noise-tolerant algorithm to learn geometric patterns. In *Proceedings of the Thirteenth International Conference on Machine Learning,* 1996. Morgan Kaufmann, 191–199, 1996. To appear in *Machine Learning.*

[28] Haussler, D., Decision theoretic generalizations of the PAC model for neural net and other learning applications. *Inform. Comput.,* 100(1), 78–150, Sep. 1992.
[29] Haussler, D., Kearns, M., Littlestone, N., and Warmuth, M.K., Equivalence of models for polynomial learnability. *Inform. Comput.,* 95(2), 129–161, Dec. 1991.
[30] Haussler, D., Littlestone, N., and Warmuth, M.K., Predicting {0, 1} functions on randomly drawn points. *Inform. Comput.,* 115(2), 284–293, 1994.
[31] Kearns, M., Efficient noise-tolerant learning from statistical queries. In *Proc. 25th Annu. ACM Sympos. Theory Comput.,* ACM Press, New York, 392–401, 1993.
[32] Kearns, M. and Valiant, L.G., Cryptographic limitations on learning Boolean formulae and finite automata. In *Proc. of the 21st Symposium on Theory of Computing,* 1989. ACM Press, New York, 433–444, 1989. To appear in *J. ACM.*
[33] Kearns, M. and Vazirani, U., *An Introduction to Computational Learning Theory.* MIT Press, Cambridge, MA, 1994.
[34] Kearns, M.J. and Schapire, R.E., Efficient distribution-free learning of probabilistic concepts. *J. Comput. Syst. Sci.,* 48(3), 464–497, 1994.
[35] Kearns, M.J., Schapire, R.E., and Sellie, L.M., Toward efficient agnostic learning. *Machine Learning,* 17(2/3), 115–142, 1994.
[36] Kodratoff, Y. and Michalski, R.S., Eds., *Machine Learning: An Artificial Intelligence Approach,* Vol. III. Morgan Kaufmann, Los Altos, CA, 1990.
[37] Littlestone, N., Learning when irrelevant attributes abound: A new linear-threshold algorithm. *Machine Learning,* 2, 285–318, 1988.
[38] Littlestone, N. and Warmuth, M.K., The weighted majority algorithm. *Inform. Comp.,* 108(2), 212–261, 1994.
[39] Maass, W. and Turán, G., Lower bound methods and separation results for on-line learning models. *Machine Learning,* 9, 107–145, 1992.
[40] Maass, W. and Warmuth, M.K., Efficient learning with virtual threshold gates. In *Proc. 12th International Conference on Machine Learning,* 1995. Morgan Kaufmann, 378–386, 1995. To appear in *Inform. Comput.*
[41] Natarajan, B.K., *Machine Learning: A Theoretical Approach.* Morgan Kaufmann, San Mateo, CA, 1991.
[42] Pitt, L. and Valiant, L., Computational limitations on learning from examples. *J. ACM,* 35, 965–984, 1988.
[43] Pitt, L. and Warmuth, M.K., Prediction preserving reducibility. *J. Comput. Syst. Sci.,* 41(3), 430–467, Dec. 1990. Special issue for the *Third Annual Conference of Structure in Complexity Theory* (Washington, DC, Jun. 88).
[44] Quinlan, J.R.,. *C45: Programs for Machine Learning.* Morgan Kaufmann, 1993.
[45] Rosenblatt, F., The perceptron: A probabilistic model for information storage and organization in the brain. *Psych. Rev.,* 65, 386–407, 1958. Reprinted in *Neurocomputing* (MIT Press, 1988).
[46] Schapire, R.E., The strength of weak learnability. *Machine Learning,* 5(2), 197–227, 1990.
[47] Sloan, R., Four types of noise in data for PAC learning. *Inform. Proc. Lett.,* 54, 157–162, 1995.
[48] Valiant, L.G., A theory of the learnable. *Commun. ACM,* 27(11), 1134–1142, Nov. 1984.
[49] Valiant, L.G., Learning disjunctions of conjunctions. In *Proceedings of the 9th International Joint Conference on Artificial Intelligence,* Vol. 1 (Los Angeles, 1985.) International Joint Committee for Artificial Intelligence, 560–566, 1985.
[50] Vapnik, V.N. and Chervonenkis, A.Y., On the uniform convergence of relative frequencies of events to their probabilities. *Theory of Probab. and its Applications,* 16(2), 264–280, 1971.

Further Information

Good introductions to computational learning theory (along with pointers to relevant literature) can be found in such textbooks as [33, 41], and [11]. Many recent results can be found in the proceedings from the following conferences: ACM Conference on Computational Learning Theory (COLT), European Conference on Computational Learning Theory (EuroCOLT), International Workshop on Algorithmic Learning Theory (ALT), International Conference on Machine Learning (ICML), IEEE Symposium on Foundations of Computer Science (FOCS), ACM Symposium on Theoretical Computing (STOC), and Neural Information Processing Conference (NIPS).

Some major journals in which many learning theory papers can be found are: *Machine Learning, Journal of the ACM, SIAM Journal of Computing, Information and Computation,* and *Journal of Computer and System Sciences.* See Kodratoff and Michalski [36] for background information on machine learning, and see Angluin and Smith [10] for a discussion of inductive inference models and research.

31

Linear Programming[1]

Vijay Chandru
Indian Institute of Science

M.R. Rao
Indian Institute of Management
Bangalore

31.1 Abstract

Linear programming has been a fundamental topic in the development of the computational sciences. The subject has its origins in the early work of L.B.J. Fourier on solving systems of linear inequalities, dating back to the 1820s. More recently, a healthy competition between the simplex and interior point methods has led to rapid improvements in the technologies of linear programming. This combined with remarkable advances in computing hardware and software have brought linear programming tools to the desktop, in a variety of application software for decision support. Linear programming has provided a fertile ground for the development of various algorithmic paradigms. Diverse topics such as symbolic computation, numerical analysis, computational complexity, computational geometry, combinatorial optimization, and randomized algorithms all have some linear programming connection. This chapter reviews this universal role played by linear programming in the science of algorithms.

[1] Dedicated to George Dantzig on this the 50^{th} Anniversary of the Simplex Algorithm.

31.2 Introduction

Linear programming has been a fundamental topic in the development of the computational sciences [50]. The subject has its origins in the early work of L.B.J. Fourier [30] on solving systems of linear inequalities, dating back to the 1820s. The revival of interest in the subject in the 1940s was spearheaded by G.B. Dantzig [19] in the U.S. and L.V. Kantorovich [45] in the former U.S.S.R. They were both motivated by the use of linear optimization for optimal resource utilization and economic planning. Linear programming, along with classical methods in the calculus of variations, provided the foundations of the field of mathematical programming, which is largely concerned with the theory and computational methods of mathematical optimization. The 1950s and 1960s marked the period when linear programming fundamentals (**duality, decomposition** theorems, network flow theory, **matrix factorizations**) were worked out in conjunction with the advancing capabilities of computing machinery [20].

The 1970s saw the realization of the commercial benefits of this huge investment of intellectual effort. Many large-scale **linear programs** were formulated and solved on mainframe computers to support applications in industry (for example, oil, airlines) and for the state (for example, energy planning, military logistics). The 1980s were an exciting period for linear programmers. The polynomial time-complexity of linear programming was established. A healthy competition between the simplex and interior point methods ensued that finally led to rapid improvements in the technologies of linear programming. This combined with remarkable advances in computing hardware and software have brought linear programming tools to the desktop, in a variety of application software (including spreadsheets) for decision support.

The fundamental nature of linear programming in the context of algorithmics is borne out by a few examples.

- Linear programming is at the starting point for variable elimination techniques on algebraic constraints [12], which in turn forms the core of algebraic and symbolic computation.
- Numerical linear algebra and particularly sparse matrix technology was largely driven in its early development by the need to solve large-scale linear programs [37, 61].
- The complexity of linear programming played an important role in the 1970s in the early stages of the development of the polynomial hierarchy and particularly in the $\mathcal{NP}$-completeness and $\mathcal{P}$-completeness in the theory of computation [67, 68].
- Linear-time algorithms based on "prune and search" techniques for low-dimensional linear programs have been used extensively in the development of computational geometry [25].
- Linear programming has been the testing ground for very exciting new developments in randomized algorithms [60].
- **Relaxation** strategies based on linear programming have played a unifying role in the construction of approximation algorithms for a wide variety of combinatorial optimization problems [17, 33, 75].

In this chapter we will encounter the basic algorithmic paradigms that have been invoked in the solution of linear programs. An attempt has been made to provide intuition about some fairly deep and technical ideas without getting bogged down in details. However, the details are important and the interested reader is invited to explore further through the references cited. Fortunately, there are many excellent texts, monographs, and expository papers on linear programming [5, 9, 16, 20, 62, 65, 67, 69, 71] that the reader can choose from, to dig deeper into the fascinating world of linear programming.

31.3 Geometry of Linear Inequalities

Two of the many ways in which linear inequalities can be understood are through algorithms or through nonconstructive geometric arguments. Each approach has its own merits (aesthetic and otherwise). Since

the rest of this chapter will emphasize the algorithmic approaches, in this section we have chosen the geometric version. Also, by starting with the geometric version, we hope to hone the reader's intuition about a convex **polyhedron,** the set of solutions to a finite system of linear inequalities[2]. We begin with the study of linear, homogeneous inequalities. This involves the geometry of (convex) **polyhedral cones.**

Polyhedral Cones

A homogeneous linear equation in n variables defines a *hyperplane* of dimension $(n-1)$ which contains the origin and is therefore a linear subspace. A homogeneous linear inequality defines a *halfspace* on one "side" of the hyperplane, defined by converting the inequality into an equation. A system of linear homogeneous inequalities therefore, defines an object which is the intersection of finitely many halfspaces, each of which contains the origin in its boundary. A simple example of such an object is the nonnegative orthant. Clearly the objects in this class resemble cones with the apex defined at the origin and with a prismatic boundary surface. We call them *convex polyhedral cones.*

A convex polyhedral cone is the set of the form

$$\mathcal{K} = \{x | Ax \leq 0\} .$$

Here A is assumed to be an $m \times n$ matrix of real numbers. A set is convex if it contains the line segment connecting any pair of points in the set. A convex set is called a convex cone if it contains all nonnegative scalar multiples of points in it. A convex set is polyhedral if it is represented by a finite system of linear inequalities. As we shall deal exclusively with cones that are both convex and polyhedral, we shall refer to them as cones.

The representation of a cone as the solutions to a finite system of homogeneous linear inequalities is sometimes, referred to as the "constraint" or "implicit" description. It is implicit because it takes an algorithm to generate points in the cone. An "explicit" or "edge" description can also be derived for any cone.

THEOREM 31.1 *Every cone $\mathcal{K} = \{x : Ax \leq 0\}$ has an "edge" representation of the following form. $\mathcal{K} = \{x : x = \Sigma_{j=1}^{L} e^j \mu_j, \quad \mu_j \geq 0 \ \ \forall j\}$ where each distinct edge of $\mathcal{K}$ is represented by a point e^j.*

Thus, for any cone we have two representations:

- CONSTRAINT REPRESENTATION. $\mathcal{K} = \{x : Ax \leq 0\}$.
- EDGE REPRESENTATION. $\mathcal{K} = \{x : x = E\mu, \quad \mu \geq 0\}$.

The matrix E is a representation of the edges of $\mathcal{K}$. Each column $E_{i.}$ of E contains the coordinates of a point on a distinct edge. Since positive scaling of the columns is permitted, we fix the representation by scaling each column so that the last non-zero entry is either 1 or -1. This scaled matrix E is called the Canonical Matrix of Generators of the cone $\mathcal{K}$.

Every point in a cone can be expressed as a positive combination of the columns of E. Since the number of columns of E can be huge, the edge representation does not seem very useful. Fortunately, the following tidy result helps us out.

THEOREM 31.2 **(Caratheodory)** [11] *For any cone $\mathcal{K}$, every $\bar{x} \in \mathcal{K}$ can be expressed as the positive combination of at most d edge points, where d is the dimension of $\mathcal{K}$.*

[2]For the study of infinite systems of linear inequalities see Chapter 33 of this *Handbook*.

Conic Duality

The representation theory for convex polyhedral cones exhibits a remarkable duality relation. This duality relation is a central concept in the theory of linear inequalities and linear programming as we shall see later.

Let $\mathcal{K}$ be an arbitrary cone. The *dual* of $\mathcal{K}$ is given by

$$\mathcal{K}^* = \left\{ u : x^T u \leq 0, \quad \forall x \in \mathcal{K} \right\} .$$

THEOREM 31.3 *The representations of a cone and its dual are related by*
$\mathcal{K} = \{x : Ax \leq 0\} = \{x : x = E\mu, \quad \mu \geq 0\}$ *and*
$\mathcal{K}^* = \{u : E^T u \leq 0\} = \{u : u = A^T \lambda, \quad \lambda \geq 0\}$.

COROLLARY 31.1 $\mathcal{K}^*$ is a convex polyhedral cone and duality is involutory (that is $(\mathcal{K}^*)^*). = \mathcal{K}$.

As we shall see, there is not much to linear inequalities or linear programming, once we have understood convex polyhedral cones.

Convex Polyhedra

The transition from cones to polyhedra may be conceived of, algebraically, as a process of dehomogenization. This is to be expected, of course, since polyhedra are represented by systems of (possibly inhomogeneous) linear inequalities and cones by systems of *homogeneous* linear inequalities. Geometrically, this process of dehomogenization corresponds to realizing that a polyhedron is the Minkowski or set sum of a cone and a **polytope** (bounded polyhedron). But before we establish this identity, we need an algebraic characterization of polytopes. Just as cones in $\Re^n$ are generated by taking *positive* linear combinations of a finite set of points in $\Re^n$, polytopes are generated by taking *convex* linear combinations of a finite set of (generator) points.

DEFINITION 31.1 Given K points $\{x^1, x^2, \cdots, x^K\}$ in $\Re^n$ the *Convex Hull* of these points is given by

$$C.H.\left(\left\{x^i\right\}\right) = \left\{ \bar{x} : \bar{x} = \sum_{i=1}^{K} \alpha_i x^i, \quad \sum_{i=1}^{K} \alpha_i = 1, \quad \alpha \geq 0 \right\} ,$$

i.e., the convex hull of a set of points in $\Re^n$ is the object generated in $\Re^n$ by taking all convex linear combinations of the given set of points. Clearly, the convex hull of a finite list of points, is always bounded.

THEOREM 31.4 [80] *P is a polytope if and only if it is the convex hull of a finite set of points.*

DEFINITION 31.2 An **extreme point** of a convex set S is a point $x \in S$ satisfying

$$x = \alpha\bar{x} + (1-\alpha)\tilde{x}, \quad \bar{x}, \tilde{x} \in S, \quad \alpha \in (0,1) \ \rightarrow \ x = \bar{x} = \tilde{x}$$

Equivalently, an extreme point of a convex set S is one that cannot be expressed as a convex linear combination of some other points in S. When S is a polyhedron, extreme points of S correspond to the geometric notion of corner points. This correspondence is formalized in the corollary below.

COROLLARY 31.2 A polytope P is the convex hull of its extreme points.

Now we go on to discuss the representation of (possibly unbounded) convex polyhedra.

THEOREM 31.5 *Any convex polyhedron P represented by a linear inequality system* $\{y : yA \leq c\}$ *can be also represented as the set addition of a convex cone R and a convex polytope Q.*

$$\begin{aligned} P &= Q + R = \{x : x = \bar{x} + \bar{r},\ \ \bar{x} \in Q,\ \ \bar{r} \in R\} \\ Q &= \left\{ \bar{x} : \bar{x} = \sum_{i=1}^{K} \alpha_i x^i,\ \ \sum_{i=1}^{K} \alpha_i = 1,\ \ \alpha \geq 0 \right\} \\ R &= \left\{ \bar{r} : \bar{r} = \sum_{j=1}^{L} \mu_j r^j,\ \ \mu \geq 0 \right\} . \end{aligned}$$

It follows from the statement of the theorem that P is nonempty if and only if the polytope Q is nonempty. We proceed now to discuss the representations of R and Q, respectively.

The cone *R* associated with the polyhedron P is called the *recession or characteristic cone* of P. A hyperplane representation of R is also readily available. It is easy to show that

$$R = \{r : Ar \leq 0\} .$$

An obvious implication of Theorem 31.5 is that P equals the polytope Q if and only if $R = \{0\}$. In this form, the vectors $\{r^j\}$ are called the *extreme rays* of P.

The polytope *Q* associated with the polyhedron P is the convex hull of a finite collection $\{x^i\}$ of points in P. It is not difficult to see that the minimal set $\{x^i\}$ is precisely the set of extreme points of P. A nonempty pointed polyhedron P, it follows, must have at least one extreme point.

The **affine hull** of P is given by

$$\begin{aligned} A.H.\{P\} &= \left\{ x : x = \sum \alpha_i x^i \right\} \\ x^i &\in P \ \ \forall i, \ \text{ and } \sum \alpha_i = 1 . \end{aligned}$$

Clearly, the x^i can be restricted to the set of extreme points of P in the definition above. Furthermore, $A.H.\{P\}$ is the smallest affine set that contains P. A hyperplane representation of $A.H.\{P\}$ is also possible. First let us define the implicit linear equality system of P to be

$$\{A^{=}x = b^{=}\} = \{A_{i.}x = b_i \ \ \forall x \in P\} .$$

Let the remaining inequalities of P be defined as

$$A^{+}x \leq b^{+} .$$

It follows that P must contain at least one point $\bar{x}$ satisfying

$$A^{=}\bar{x} = b^{=} \text{ and } A^{+}\bar{x} < b^{+} .$$

LEMMA 31.1 $A.H.\{P\} = \{x : A^{=}x = b^{=}\}$.

The **dimension** of a polyhedron P in $\Re^n$ is defined to be the dimension of the affine hull of P, which equals the maximum number of affinely independent points, in $A.H.\{P\}$, minus one. P is said to be full-dimensional if its dimension equals n or, equivalently, if the affine hull of P is all of $\Re^n$.

A **supporting hyperplane** of the polyhedron P is a hyperplane H

$$H = \left\{ x : b^T x = z^* \right\} ,$$

satisfying

$$\begin{aligned} b^T x &\leq z^* \ \forall x \in P \\ b^T \hat{x} &= z^* \ for\ some\ \hat{x} \in P . \end{aligned}$$

A supporting hyperplane H of P is one that touches P such that all of P is contained in a halfspace of H. Note that a supporting plane can touch P at more than one point.

A **face** of a nonempty polyhedron P is a subset of P that is either P itself or is the intersection of P with a supporting hyperplane of P. It follows that a face of P is itself a nonempty polyhedron. A face of dimension, one less than the dimension of P, is called a **facet.** A face of dimension one is called an **edge** (note that extreme rays of P are also edges of P). A face of dimension zero is called a **vertex** of P (the vertices of P are precisely the extreme points of P). Two vertices of P are said to be adjacent if they are both contained in an edge of P. Two facets are said to be adjacent if they both contain a common face of dimension one less than that of a facet. Many interesting aspects of the facial structure of polyhedra can be derived from the following representation lemma.

LEMMA 31.2 F is a face of $P = \{x : Ax \leq b\}$ if and only if F is nonempty and $F = P \cap \{x : \tilde{A}x = \tilde{b}\}$, where $\tilde{A}x \leq \tilde{b}$ is a subsystem of $Ax \leq b$.

As a consequence of the lemma, we have an algebraic characterization of extreme points of polyhedra.

THEOREM 31.6 *Given a polyhedron P, defined by* $\{x : Ax \leq b\}$, x^i *is an extreme point of P if and only if it is a face of P satisfying* $A^i x^i = b^i$ *where* $((A^i), (b^i))$ *is a submatrix of* (A, b) *and the rank of* A^i *equals n.*

Now we come to Farkas Lemma, which says that a linear inequality system has a solution if and only if a related (polynomial size) linear inequality system has no solution. This lemma is representative of a large body of theorems in mathematical programming known as *theorems of the alternative.*

LEMMA 31.3 **(Farkas)** [27] Exactly one of the alternatives

$$I.\ \ \exists\, x\, :\, Ax \leq b \qquad II.\ \ \exists\, y \geq 0\, :\, A^T y = 0,\ b^T y < 0$$

is true for any given real matrices A, b.

Optimization and Dual Linear Programs

The two fundamental problems of linear programming (which are polynomially equivalent) are

- **Solvability:** This is the problem of checking if a system of linear constraints on real (rational) variables is solvable or not. Geometrically, we have to check if a polyhedron, defined by such constraints, is nonempty.
- **Optimization:** This is the problem (LP) of optimizing a linear objective function over a polyhedron described by a system of linear constraints.

Building on polarity in cones and polyhedra, duality in linear programming is a fundamental concept which is related to both the complexity of linear programming and to the design of algorithms for solvability and optimization. We will encounter the solvability version of duality (called Farkas' Lemma) while discussing the Fourier elimination technique below. Here we will state the main duality results for optimization. If we take the *primal* linear program to be

$$(P) \quad \min_{x \in \Re^n} \{cx : Ax \geq b\} ,$$

there is an associated *dual* linear program

$$(D) \quad \max_{y \in \Re^m} \left\{ b^T y : A^T y = c^T ,\ y \geq 0 \right\} ,$$

and the two problems satisfy

1. For any $\hat{x}$ and $\hat{y}$ feasible in (P) and (D) (i.e., they satisfy the respective constraints), we have $c\hat{x} \geq b^T \hat{y}$ (`weak duality`).
2. (P) has a finite optimal solution if and only if (D) does.
3. x^* and y^* are a pair of optimal solutions for (P) and (D), respectively, if and only if x^* and y^* are feasible in (P) and (D) (i.e., they satisfy the respective constraints) **and** $cx^* = b^T y^*$ (`strong duality`).
4. x^* and y^* are a pair of optimal solutions for (P) and (D), respectively, if and only if x^* and y^* are feasible in (P) and (D) (i.e., they satisfy the respective constraints) **and** $(Ax^* - b)^T y^* = 0$ (`complementary slackness`).

The strong duality condition above gives us a good stopping criterion for optimization algorithms. The complementary slackness condition, on the other hand gives us a constructive tool for moving from dual to primal solutions and vice-versa. The weak duality condition gives us a technique for obtaining lower bounds for minimization problems and upper bounds for maximization problems.

Note that the properties above have been stated for linear programs in a particular form. The reader should be able to check that if for example the primal is of the form

$$\left(P'\right) \quad \min_{x \in \Re^n} \{cx : Ax = b,\ x \geq 0\} ,$$

then the corresponding dual will have the form

$$\left(D'\right) \quad \max_{y \in \Re^m} \left\{ b^T y : A^T y \leq c^T \right\} .$$

The tricks needed for seeing this is that any equation can be written as two inequalities, an unrestricted variable can be substituted by the difference of two nonnegatively constrained variables, and an inequality can be treated as an equality by adding a nonnegatively constrained variable to the lesser side. Using these tricks, the reader could also check that dual construction in linear programming is involutory (i.e., the dual of the dual is the primal).

Complexity of Linear Equations and Inequalities

Complexity of Linear Algebra

Let us restate the fundamental problem of linear algebra as a decision problem.

$$\mathcal{CLS} = \left\{ (A, b) :\ \exists\, x \in \mathcal{Q}^n, Ax = b \right\} . \tag{31.1}$$

In order to solve the decision problem on $\mathcal{CLS}$ it is useful to recall *homogeneous* linear equations. A basic result in linear algebra is that any linear subspace of $\mathcal{Q}^n$ has two representations, one from hyperplanes and the other from a vector basis.

$\mathcal{L} = \{x \in \mathcal{Q}^n : Ax = 0\}$

$\mathcal{L} = \{x \in \mathcal{Q}^n : x = Cy, \quad y \in \mathcal{Q}^k\}$.

Corresponding to a linear subspace $\mathcal{L}$ there exists a dual (orthogonal complementary) subspace $\mathcal{L}^*$ with the roles of the hyperplanes and basis vectors of $\mathcal{L}$ exchanged.

$\mathcal{L}^* = \{z : Cz = 0\}$

$\mathcal{L}^* = \{z : z = Ax\}$

$dimension\mathcal{L} + dimension\mathcal{L}^* = n$.

Using these representation results it is quite easy to establish the *Fundamental Theorem of Linear Algebra.*

THEOREM 31.7 *Either $Ax = b$ for some x* or *$yA = 0$, $yb \neq 0$ for some y.*

Along with the basic theoretical constructs outlined above, let us also assume knowledge of the *Gaussian Elimination Method* for solving a system of linear equations. It is easily verified that on a system of size m by n, this method uses $O(m^2n)$ elementary arithmetic operations. However we also need some bound on the size of numbers handled by this method. By the size of a rational number we mean the length of binary string encoding the number. And similarly for a matrix of numbers.

LEMMA 31.4 For any square matrix S of rational numbers, the size of the determinant of S is polynomially related to the size of S itself.

Since all the numbers in a basic solution (i.e., basis-generated) of $Ax = b$ are bounded in size by subdeterminants of the input matrix (A, b) we can conclude that $\mathcal{CLS}$ is a member of $\mathcal{NP}$. The Fundamental Theorem of Linear Algebra further establishes that $\mathcal{CLS}$ is in $\mathcal{NP} \bigcap co\mathcal{NP}$. And finally the polynomiality of Gaussian Elimination establishes that $\mathcal{CLS}$ is in $\mathcal{P}$.

Complexity of Linear Inequalities

From our earlier discussion of polyhedra, we have the following algebraic characterization of extreme points of polyhedra.

THEOREM 31.8 *Given a polyhedron P, defined by $\{x : Ax \leq b\}$, x^i is an extreme point of P if and only if it is a face of P satisfying $A^i x^i = b^i$ where $((A^i), (b^i))$ is a submatrix of (A, b) and the rank of A^i equals n.*

COROLLARY 31.3 The decision problem of verifying the membership of an input string (A, b) in the language $\mathcal{L}_I = \{(A, b) : \exists\, x \; such \; that \; Ax \leq b\}$ belongs to $\mathcal{NP}$.

PROOF It follows from the theorem that every extreme point of the polyhedron $P = \{x : Ax \leq b\}$ is the solution of an $(n \times n)$ linear system whose coefficients come from (A, b). Therefore we can guess a polynomial length string representing an extreme point and check its membership in P in polynomial time.

A consequence of Farkas Lemma is that the decision problem of testing membership of input (A, b) in the language

$$\mathcal{L}_I = \{(A, b) : \exists\, x \;\text{ such that }\; Ax \leq b\}$$

is in $\mathcal{NP} \bigcap co\mathcal{NP}$. That $\mathcal{L}_I$ can be recognized in polynomial time, follows from algorithms for linear programming that we now discuss.

We are now ready for a tour of some algorithms for linear programming. We start with the classical technique of Fourier, which is interesting because of its simple syntactic specification. It leads to simple proofs of the duality principle of linear programming that was alluded to above. We will then review the simplex method of linear programming [19], a method that uses the vertex-edge structure of a convex polyhedron to execute an optimization march. The simplex method has been finely honed over almost five decades now. We will spend some time with the ellipsoid method and in particular with the polynomial equivalence of solvability (optimization) and **separation** problems. This aspect of the ellipsoid method [35] has had a major impact on the identification of many tractable classes of combinatorial optimization problems. We conclude the tour of the basic methods with a description of Karmarkars [47] breakthrough in 1984, which was an important landmark in the brief history of linear programming. A noteworthy role of interior point methods has been to make practical the theoretical demonstrations of tractability of various aspects of linear programming, including solvability and optimization, that were provided via the ellipsoid method.

In later sections we will review the more sophisticated (and naturally esoteric) aspects of linear programming algorithms. This will include **strongly polynomial** algorithms for special cases, randomized algorithms and specialized methods for large-scale linear programming. Some readers may notice that we do not have a special section devoted to the discussion of parallel computation in the context of linear programming. This is partly because we are not aware of a well developed framework for such a discussion. We have instead introduced discussion and remarks about the effects of parallelism in the appropriate sections of this chapter.

31.4 Fourier's Projection Method

Linear programming is at the starting point for variable elimination techniques on algebraic constraints [12], which in turn forms the core of algebraic and symbolic computation. Constraint systems of linear *inequalities* of the form $Ax \leq b$, where A is an $m \times n$ matrix of real numbers are widely used in mathematical models. Testing the solvability of such a system is equivalent to linear programming. We now describe the elegant syntactic variable elimination technique due to Fourier [30].

Suppose we wish to eliminate the first variable x_1 from the system $Ax \leq b$. Let us denote

$$I^+ = \{i : A_{i1} > 0\} \quad I^- = \{i : A_{i1} < 0\} \quad I^0 = \{i : A_{i1} = 0\} \ .$$

Our goal is to create an equivalent system of linear inequalities $\tilde{A}\tilde{x} \leq \tilde{b}$ defined on the variables $\tilde{x} = (x_2, x_3, \cdots, x_n)$.

- If I^+ is empty then we can simply delete all the inequalities with indices in I^-, since they can be trivially satisfied by choosing a large enough value for x_1. Similarly, if I^- is empty we can discard all inequalities in I^+.
- For each $k \in I^+$, $l \in I^-$ we add $-A_{l1}$ times the inequality $A_k x \leq b_k$ to A_{k1} times $A_l x \leq b_l$. In these new inequalities the coefficient of x_1 is wiped out, i.e., x_1 is eliminated. Add these new inequalities to those already in I^0.
- The inequalities $\{\tilde{A}_{i1}\tilde{x} \leq \tilde{b}_i\}$ for all $i \in I^0$ represent the equivalent system on the variables $\tilde{x} = (x_2, x_3, \cdots, x_n)$.

Repeat this construction with $\tilde{A}\tilde{x} \leq \tilde{b}$ to eliminate x_2 and so on until all variables are eliminated. If the resulting $\tilde{b}$ (after eliminating x_n) is nonnegative we declare the original (and intermediate) inequality

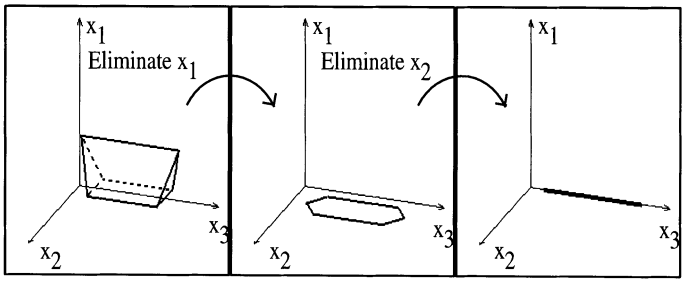


FIGURE 31.1 Variable elimination and projection.

systems as being consistent. Otherwise[3] $\tilde{b} \not\geq 0$ and we declare the system inconsistent.

As an illustration of the power of elimination as a tool for theorem proving, we show now that Farkas Lemma is a simple consequence of the correctness of Fourier elimination. The lemma gives a direct proof that solvability of linear inequalities is in $\mathcal{NP} \bigcap co\mathcal{NP}$.

Farkas Lemma Exactly one of the alternatives

$$I. \quad \exists\, x \in \Re^n \,:\, Ax \leq b \qquad II. \quad \exists\, y \in \Re^m_+ \,:\, y^t A = 0,\ y^t b < 0$$

is true for any given real matrices $A,\ b$.

PROOF Let us analyze the case when Fourier Elimination provides a proof of the inconsistency of a given linear inequality system $Ax \leq b$. The method clearly converts the given system into $RAx \leq Rb$ where RA is zero and Rb has at least one negative component. Therefore, there is some row of R, say r, such that $rA = 0$ and $rb < 0$. Thus $\neg I$ implies II. It is easy to see that I and II cannot both be true for fixed A, b.

In general, the Fourier elimination method is quite inefficient. Let k be any positive integer and n the number of variables be $2^k + k + 2$. If the input inequalities have left-hand sides of the form $\pm x_r \pm x_s \pm x_t$ for all possible $1 \leq r < s < t \leq n$, it is easy to prove by induction that after k variables are eliminated, by Fourier's method, we would have at least $2^{\frac{n}{2}}$ inequalities. The method is therefore exponential in the worst case and the explosion in the number of inequalities has been noted, in practice as well, on a wide variety of problems. We will discuss the central idea of minimal generators of the projection cone that results in a much improved elimination method [41].

First let us identify the set of variables to be eliminated. Let the input system be of the form

$$P = \left\{ (x, u) \in \Re^{n_1+n_2} \mid Ax + Bu \leq b \right\},$$

where u is the set to be eliminated. The projection of P onto x or equivalently the effect of eliminating the u variables is

$$P_x = \left\{ x \in \Re^{n_1} \mid \exists u \in \Re^{n_2} \text{ such that } Ax + Bu \leq b \right\}.$$

Now W, the *projection cone* of P, is given by

$$W = \left\{ w \in \Re^m \mid wB = 0,\ w \geq 0 \right\}.$$

[3]Note that the final $\tilde{b}$ may not be defined if all the inequalities are deleted by the monotone sign condition of the first step of the construction described above. In such a situation we declare the system $Ax \leq b$ *strongly consistent,* since it is consistent for any choice of b in $\Re^m$. In order to avoid making repeated references to this exceptional situation, let us simply assume that it does not occur. The reader is urged to verify that this assumption is indeed benign.

A simple application of Farkas Lemma yields a description of P_x in terms of W.

Projection Lemma Let G be any set of generators (e.g., the set of extreme rays) of the cone W. Then $P_x = \{ x \in \Re^{n_1} \mid (gA)x \leq gb \quad \forall\, g \in G \}$.

The lemma, sometimes attributed to Černikov [10], reduces the computation of P_x to enumerating the extreme rays of the cone W or equivalently the extreme points of the polytope $W \cap \{ w \in \Re^m \mid \Sigma_{i=1}^m w_i = 1 \}$.

31.5 The Simplex Method

Consider a polyhedron $\mathcal{K} = \{x \in \Re^n : \ Ax = b, \ x \geq 0\}$. Now $\mathcal{K}$ cannot contain an infinite (in both directions) line, since it is lying within the nonnegative orthant of $\Re^n$. Such a polyhedron is called a *pointed* polyhedron. Given a pointed polyhedron $\mathcal{K}$ we observe that

- If $\mathcal{K} \neq \emptyset$ then $\mathcal{K}$ has at least one extreme point.
- If $\min\{cx : \ Ax = b, \ x \geq 0\}$ has an optimal solution, then it has an optimal extreme point solution.

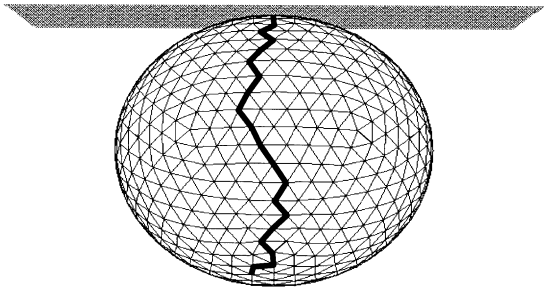


FIGURE 31.2 The simplex path.

These observations together are sometimes called the fundamental theorem of linear programming since they suggest simple finite tests for both solvability and optimization. To generate all extreme points of $\mathcal{K}$, in order to find an optimal solution, is an impractical idea. However, we may try to run a partial search of the space of extreme points for an optimal solution. A simple local improvement search strategy of moving from extreme point to adjacent extreme point until we get to a local optimum is nothing but the simplex method of linear programming [19, 20]. The local optimum also turns out to be a global optimum, because of the convexity of the polyhedron $\mathcal{K}$ and the objective function cx.

Procedure: **Primal Simplex**($\mathcal{K}$,c)

0. **Initialize:**
   - x_0 := an extreme point of $\mathcal{K}$
   - $k := 0$
1. **Iterative Step:**

   do

   If for all edge directions $\mathcal{D}_k$ at x_k, the objective function is

nondecreasing, i.e.,

$$cd \geq 0 \quad \forall\, d \in \mathcal{D}_k$$

then exit and return optimal x_k.

Else pick some d_k in $\mathcal{D}_k$ such that $cd_k < 0$.

If $d_k \geq 0$ **then** declare the linear program unbounded in objective value and exit.

Else $x_{k+1} := x_k + \theta_k * d_k$, where

$$\theta_k = \max\{\theta : x_k + \theta * d_k \geq 0\}$$

$k := k + 1$

od

2. **End**

Remarks:

1. In the initialization step we assumed that an extreme point x_0 of the polyhedron $\mathcal{K}$ is available. This also assumes that the solvability of the constraints defining $\mathcal{K}$ has been established. These assumptions are reasonable since we can formulate the solvability problem as an optimization problem, with a self-evident extreme point, whose optimal solution either establishes unsolvability of $Ax = b,\ x \geq 0$, or provides an extreme point of $\mathcal{K}$. Such an optimization problem is usually called a Phase I model. The point is, of course, that the simplex method as described above can be invoked on the Phase I model and if successful, can be invoked once again to carry out the intended minimization of cx. There are several different formulations of the Phase I model that have been advocated. Here is one.

$$\min\{v_0 : Ax + bv_0 = b,\ x \geq 0,\ v_0 \geq 0\}\ .$$

   The solution $(x, v_0)^T = (0, \cdots, 0, 1)$ is a self-evident extreme point, and $v_0 = 0$ at an optimal solution of this model is a necessary and sufficient condition for the solvability of $Ax = b,\ x \geq 0$.
2. The scheme for generating improving edge directions uses an algebraic representation of the extreme points as certain bases, called feasible bases, of the vector space generated by the columns of the matrix A. It is possible to have linear programs for which an extreme point is geometrically overdetermined (degenerate), i.e., there are more than d facets of $\mathcal{K}$ that contain the extreme point, where d is the dimension of $\mathcal{K}$. In such a situation, there would be several feasible bases corresponding to the same extreme point. When this happens, the linear program is said to be *primal degenerate.*
3. There are two sources of nondeterminism in the primal simplex procedure. The first involves the choice of edge direction d_k made in Step 1. At a typical iteration there may be many edge directions that are improving in the sense that $cd_k < 0$. Dantzig's Rule, Maximum Improvement Rule, and Steepest Descent Rule are some of the many rules that have been used to make the choice of edge direction in the simplex method. There is, unfortunately, no clearly dominant rule, and successful codes exploit the empirical and analytic insights that have been gained over the years to resolve the edge selection nondeterminism in the simplex method.
   The second source of nondeterminism arises from degeneracy. When there are multiple feasible bases corresponding to an extreme point, the simplex method has to pivot from basis to adjacent basis by picking an entering basic variable (a pseudo edge direction) and by dropping

one of the old ones. A wrong choice of the leaving variables may lead to cycling in the sequence of feasible bases generated at this extreme point. Cycling is a serious problem when linear programs are highly degenerate, as in the case of linear relaxations of many combinatorial optimization problems. The Lexicographic Rule (Perturbation Rule) for choice of leaving variables in the simplex method is a provably finite method (i.e., all cycles are broken).
A clever method proposed by Bland (cf. [71]) preorders the rows and columns of the matrix A. In case of nondeterminism in either entering or leaving variable choices, Bland's Rule just picks the lowest index candidate. All cycles are avoided by this rule also.

Implementation Issues: Basis Representations

The enormous success of the simplex method has been primarily due to its ability to solve large size problems that arise in practice. A distinguishing feature of many of the linear problems that are solved routinely in practice, is the sparsity of the constraint matrix. So from a computational point of view, it is desirable to take advantage of the sparseness of the constraint matrix. Another important consideration in the implementation of the simplex method is to control the accumulation of round off errors that arise because the arithmetic operations are performed with only a fixed number of digits and the simplex method is an iterative procedure.
An algebraic representation of the simplex method in matrix notation is as follows:

0: Find an initial feasible extreme point x^0, and the corresponding feasible basis B (of the vector space generated by the columns of the constraint matrix A). If no such x_0 exists, stop, there is no feasible solution. Otherwise, let $t = 0$, and go to Step 1.

1: Partition the matrix A as $A = (B, N)$, the solution vector x as $x = (x_B, x_N)$ and the objective function vector c as $c = (c_B, c_N)$, corresponding to the columns in B.

2: The extreme point x^t is given by $x^t = (x_B, 0)$, where $Bx_B = b$

3: Solve the system $\pi_B\ B = c_B$ and calculate $r = c_N - \pi_B\ N$. If $r \geq 0$, stop, the current solution $x^t = (x_B, 0)$ is optimal. Otherwise, let $r_k = \min_j\{r_j\}$, where r_j is the jth component of r (actually one may pick any $r_j < 0$ as r_k).

4: Let a_k denote the kth column of N corresponding to r_k. Find y_k such that $B\ y_k = a_k$

5: Find $x_B(p)/y_{pk} = \min_i\{x_B(i)/y_{ik} :\ y_{ik} > 0\}$ where $x_B(i)$ and y_{ik} denote the ith component of x_B and y_k, respectively.

6: The new basis $\hat{B}$ is obtained from B by replacing the pth column of B by the kth column of N. Let the new feasible basis $\hat{B}$ be denoted as B. Return to Step 1.

LU Factorization

At each iteration, the simplex method requires the solution of the following systems:
$Bx_B = b$; $\pi_B B = c_B$ and $By_k = a_k$.

After row interchanges, if necessary, any basis B can be factorized as $B = LU$ where L is a lower triangular matrix and U is an upper triangular matrix. So solving $LUx_B = b$ is equivalent to solving the triangular systems $Lv = b$ and $Ux_B = v$. Similarly, for $By_k = a_k$, we solve $Lw = a_k$ and $Uy_k = w$. Finally, for $\pi_B B = c_B$, we solve $\pi_B L = \lambda$ and $\lambda U = c_B$.

Let the current basis B and the updated basis $\hat{B}$ be represented as
$B = (a_1, a_2, \ldots, a_{p-1}, a_p, a_{p+1}, \ldots, a_m)$ and $\hat{B} = (a_1, a_2, \ldots, a_{p-1}, a_{p+1}, a_{p+2}, \ldots, a_m, a_k)$.

An efficient implementation of the simplex method requires the updating of the triangular matrices L and U as triangular matrices $\hat{L}$ and $\hat{U}$ where $B = LU$ and $\hat{B} = \hat{L}\hat{U}$. This is done by first obtaining $H = (u_1, u_2, \ldots, u_{p-1}, u_{p+1}, \ldots, u_m, w)$ where u_i is the ith column of U and $w = L^{-1}a_k$. The matrix H has zeros below the main diagonal in the first $p-1$ columns and zeros below the element immediately under the diagonal in the remaining columns. The matrix H can be reduced to an upper triangular matrix by Gaussian elimination which is equivalent to multiplying H on the left by matrices $M_i,\ i = p, p+1, \ldots, m-1$, where M_j differs from an identity matrix in column j which is given by $(0, \ldots, 0, 1, m_j, 0 \ldots 0)^T$, where m_j is in position $j+1$. Now $\hat{U}$ is given by $\hat{U} = M_{m-1}, M_{m-2}, \ldots, M_p H$ and $\hat{L}$ is given by $\hat{L} = LM_p^{-1}, \ldots, M_{m-1}^{-1}$. Note that M_j^{-1} is M_j with the sign of the off-diagonal term m_j reversed.

The LU factorization preserves the sparsity of the basis B, in that the number of non-zero entries in L and U is typically not much larger than the number of non-zero entries in B. Furthermore, this approach effectively controls the accumulation of round off errors and maintains good numerical accuracy. In practice, the LU factorization is periodically recomputed for the matrix $\hat{B}$ instead of updating the factorization available at the previous iteration. This computation of $\hat{B} = \hat{L}\hat{U}$ is achieved by Gaussian elimination to reduce $\hat{B}$ to an upper triangular matrix (for details, see for instance [37, 61, 62]). There are several variations of the basic idea of factorization of the basis matrix B, as described here, to preserve sparsity and control round off errors.

REMARK 31.1 The simplex method is not easily amenable to parallelization. However, some steps such as identification of the entering variable and periodic refactorization can be efficiently parallelized.

Geometry and Complexity of the Simplex Method

An elegant geometric interpretation of the simplex method can be obtained by using a *column space representation* [20], i.e., $\Re^{m+1}$ coordinatized by the rows of the $(m+1) \times n$ matrix $\begin{pmatrix} c \\ A \end{pmatrix}$. In fact it is this interpretation that explains why it is called the simplex method. The bases of A correspond to an arrangement of simplicial cones in this geometry, and the pivoting operation corresponds to a physical pivot from one cone to an adjacent one in the arrangement. An interesting insight that can be gained from the column space perspective is that Karmarkar's interior point method can be seen as a natural generalization of the simplex method [14, 77].

However, the geometry of linear programming, and of the simplex method, has been largely developed in the space of the x variables, i.e., in $\Re^n$. The simplex method walks along edge paths on the combinatorial graph structure defined by the boundary of convex polyhedra. These graphs are quite dense (Balinski's Theorem [83] states that the graph of d-dimensional polyhedron must be d-connected). A polyhedral graph can also have a huge number of vertices since the Upper Bound Theorem of McMullen, see [83], states that the number of vertices can be as large as $O(k^{\lfloor d/2 \rfloor})$ for a polytope in d dimensions defined by k constraints. Even a polynomial bound on the diameter of polyhedral graphs is not known. The best bound obtained to date is $O(k^{1+\log d})$ of a polytope in d dimensions defined by k constraints. Hence it is no surprise that there is no known variant of the simplex method with a worst-case polynomial guarantee on the number of iterations.

Klee and Minty [49] exploited the sensitivity of the original simplex method of Dantzig, to projective scaling of the data, and constructed exponential examples for it. These examples were simple projective distortions of the hypercube to embed long isotonic (improving objective value) paths in the graph. Scaling is used in the Klee–Minty construction, to trick the choice of entering variable (based on most negative reduced cost) in the simplex method and thus keep it on an exponential path. Later, several variants of the entering variable choice (best improvement, steepest descent, etc.) were all shown to be susceptible to similar constructions of exponential examples (cf. [71]).

Despite its worst-case behavior, the simplex method has been the veritable workhorse of linear programming for five decades now. This is because both empirical [7, 20] and probabilistic [9, 39] analyses indicate that the number of iterations of the simplex method is just slightly more than linear in the dimension of the primal polyhedron.

The ellipsoid method of Shor [76] was devised to overcome poor scaling in convex programming problems and therefore turned out to be the natural choice of an algorithm to first establish polynomial-time solvability of linear programming. Later Karmarkar [47] took care of both projection and scaling simultaneously and arrived at a superior algorithm.

31.6 The Ellipsoid Method

The Ellipsoid Algorithm of Shor [76] gained prominence in the late 1970s when Haćijan (pronounced Khachiyan) [38] showed that this convex programming method specializes to a polynomial-time algorithm for linear programming problems. This theoretical breakthrough naturally led to intense study of this method and its properties. The survey paper by Bland et al. [8] and the monograph by Akgül [2] attest to this fact. The direct theoretical consequences for combinatorial optimization problems was independently documented by Padberg and Rao [66], Karp and Papadimitriou [48], and Grötschel, Lovász and Schrijver [34]. The ability of this method to implicitly handle linear programs with an exponential list of constraints and maintain polynomial-time convergence is a characteristic that is the key to its applications in combinatorial optimization. For an elegant treatment of the many deep theoretical consequences of the ellipsoid algorithm, the reader is directed to the monograph by Lovász [51], and the book by Grötschel, Lovász and Schrijver [35].

Computational experience with the ellipsoid algorithm, however, showed a disappointing gap between the theoretical promise and practical efficiency of this method in the solution of linear programming problems. Dense matrix computations as well as the slow average-case convergence properties are the reasons most often cited for this behavior of the ellipsoid algorithm. On the positive side though, it has been noted (cf. Ecker and Kupferschmid [24]) that the ellipsoid method is competitive with the best known algorithms for (nonlinear) convex programming problems.

Let us consider the problem of testing if a polyhedron $\mathcal{Q} \in \Re^d$, defined by linear inequalities, is nonempty. For technical reasons let us assume that $\mathcal{Q}$ is rational, i.e., all extreme points and rays of $\mathcal{Q}$ are rational vectors or equivalently that all inequalities in some description of $\mathcal{Q}$ involve only rational coefficients. The ellipsoid method does not require the linear inequalities describing $\mathcal{Q}$ to be explicitly specified. It suffices to have an oracle representation of $\mathcal{Q}$. Several different types of oracles can be used in conjunction with the ellipsoid method [35, 48, 66]. We will use the *strong separation oracle* described below.

Oracle: **Strong Separation** $(\mathcal{Q}, y)$
`Given a vector` $y \in \Re^d$`, decide whether` $y \in \mathcal{Q}$`, and if not find a hyperplane that separates` y `from` $\mathcal{Q}$`; more precisely, find a vector` $c \in \Re^d$ `such that` $c^T y < \min\{c^T x : x \in \mathcal{Q}\}$.

The ellipsoid algorithm initially chooses an ellipsoid large enough to contain a part of the polyhedron $\mathcal{Q}$ if it is nonempty. This is easily accomplished because we know that if $\mathcal{Q}$ is nonempty then it has a rational solution whose (binary encoding) length is bounded by a polynomial function of the length of the largest coefficient in the linear program and the dimension of the space.

The center of the ellipsoid is a feasible point if the separation oracle tells us so. In this case, the algorithm terminates with the coordinates of the center as a solution. Otherwise, the separation oracle outputs an inequality that separates the center point of the ellipsoid from the polyhedron $\mathcal{Q}$. We translate the hyperplane defined by this inequality to the center point. The hyperplane slices the ellipsoid into two halves, one of which can be discarded. The algorithm now creates a new ellipsoid that is the minimum

volume ellipsoid containing the remaining half of the old one. The algorithm questions if the new center is feasible and so on. The key is that the new ellipsoid has substantially smaller volume than the previous one. When the volume of the current ellipsoid shrinks to a sufficiently small value, we are able to conclude that $\mathcal{Q}$ is empty. This fact is used to show the polynomial time convergence of the algorithm.

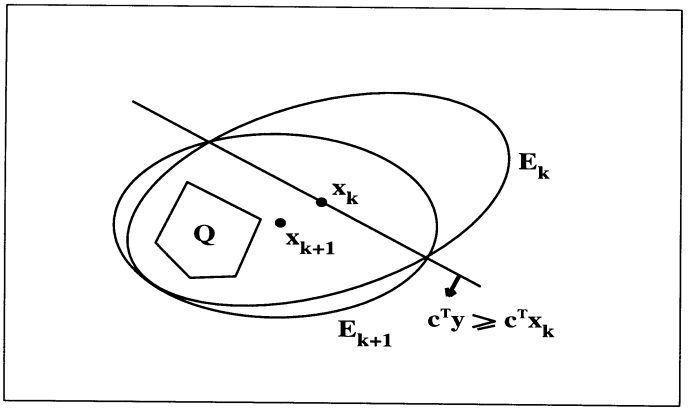


FIGURE 31.3 Shrinking ellipsoids.

Ellipsoids in $\Re^d$ are denoted as $E(A, y)$ where A is a $d \times d$ positive definite matrix and $y \in \Re^d$ is the center of the ellipsoid $E(A, y)$.

$$E(A, y) = \left\{ x \in \Re^d \;\middle|\; (x - y)^T A^{-1} (x - y) \leq 1 \right\}.$$

The ellipsoid algorithm is described on the iterated values, A_k and x^k, which specify the underlying ellipsoids $E_k(A_k, x^k)$.

Procedure: **Ellipsoid** $(\mathcal{Q})$

0. **Initialize:**
   - $N := N(\mathcal{Q})$ `(comment: iteration bound)`
   - $R := R(\mathcal{Q})$ `(comment: radius of the initial ellipsoid/sphere` E_0`)`
   - $A_0 := R^2 I$
   - $x_0 := 0$ `(comment: center of` E_0`)`
   - $k := 0$

1. **Iterative Step:**

   while $k < N$

   call `Strong Separation` $(\mathcal{Q}, x^k)$

   if $x^k \in \mathcal{Q}$ **halt**

   else `hyperplane` $\{x \in \Re^d \mid c^T x = c_0\}$ `separates` x^k `from` $\mathcal{Q}$

   Update

$$b := \frac{1}{\sqrt{c^T A_k c}} A_k c$$

$$x^{k+1} := x^k - \frac{1}{d+1} b$$

$$A_{k+1} := \frac{d^2}{d^2-1}(A_k - \frac{2}{d+1}bb^T)$$
$$k := k+1$$

endwhile

2. **Empty Polyhedron:**
   - **halt** `and` **declare** ``$\mathcal{Q}$ `is empty`''
3. **End**

The crux of the complexity analysis of the algorithm is on the a priori determination of the iteration bound. This in turn depends on three factors. The volume of the initial ellipsoid E_0, the rate of volume shrinkage ($\frac{vol(E_{k+1})}{vol(E_k)} < e^{-\frac{1}{(2d)}}$) and the volume threshold at which we can safely conclude that $\mathcal{Q}$ must be empty. The assumption of $\mathcal{Q}$ being a rational polyhedron is used to argue that $\mathcal{Q}$ can be modified into a full-dimensional polytope without affecting the decision question ("Is $\mathcal{Q}$ nonempty?"). After careful accounting for all these technical details and some others (e.g., compensating for the round-off errors caused by the square root computation in the algorithm), it is possible to establish the following fundamental result.

THEOREM 31.9 *There exists a polynomial $g(d, \phi)$ such that the* `ellipsoid method` *runs in time bounded by $T\ g(d, \phi)$ where ϕ is an upper bound on the size of linear inequalities in some description of $\mathcal{Q}$ and T is the maximum time required by the oracle* `Strong Separation`$(\mathcal{Q}, y)$ *on inputs y of size at most $g(d, \phi)$.*

The size of a linear inequality is just the length of the encoding of all the coefficients needed to describe the inequality. A direct implication of the theorem is that solvability of linear inequalities can be checked in polynomial time if strong separation can be solved in polynomial time. This implies that the standard linear programming solvability question has a polynomial-time algorithm (since separation can be effected by simply checking all the constraints). Happily, this approach provides polynomial-time algorithms for much more than just the standard case of linear programming solvability. The theorem can be extended to show that the optimization of a linear objective function over $\mathcal{Q}$ also reduces to a polynomial number of calls to the strong separation oracle on $\mathcal{Q}$. A converse to this theorem also holds, namely separation can be solved by a polynomial number of calls to a solvability/optimization oracle [35]. Thus, optimization and separation are polynomially equivalent. This provides a very powerful technique for identifying tractable classes of optimization problems. Semidefinite programming and submodular function minimization are two important classes of optimization problems that can be solved in polynomial time using this property of the ellipsoid method.

Semidefinite Programming

The following optimization problem, defined on symmetric $(n \times n)$ real matrices,

$$(SDP) \quad \min_{X \in \Re^{n \times n}} \left\{ \sum_{ij} C \bullet X : A \bullet X = B,\ X \succeq 0 \right\},$$

is called a semidefinite program. Note that $X \succeq 0$ denotes the requirement that X is a positive semidefinite matrix, and $F \bullet G$ for $n \times n$ matrices F and G denotes the product matrix $(F_{ij} * G_{ij})$. From the definition of positive semi-definite matrices, $X \succeq 0$ is equivalent to

$$q^T X q \ \geq\ 0 \quad \text{for every } \ q \in \Re^n\ .$$

Thus (SDP) is really a linear program on $O(n^2)$ variables with an (uncountably) infinite number of linear inequality constraints. Fortunately, the strong separation oracle is easily realized for these constraints. For a given symmetric X we use Cholesky factorization to identify the minimum eigenvalue λ_{min}. If λ_{min} is nonnegative then $X \succeq 0$ and if, on the other hand, λ_{min} is negative we have a separating inequality

$$\gamma_{min}^T X \gamma_{min} \geq 0 ,$$

where γ_{min} is the eigenvector corresponding to λ_{min}. Since the Cholesky factorization can be computed by an $O(n^3)$ algorithm, we have a polynomial-time separation oracle and an efficient algorithm for (SDP) via the Ellipsoid method. Alizadeh [3] has shown that interior point methods can also be adapted to solving (SDP) to within an additive error ϵ in time polynomial in the size of the input and $\log \frac{1}{\epsilon}$.

This result has been used to construct efficient approximation algorithms for maximum stable sets and cuts of graphs [33], Shannon capacity of graphs, minimum colorings of graphs. It has been used to define hierarchies of relaxations for integer linear programs that strictly improve on known exponential-size linear programming relaxations [52].

Minimizing Submodular Set Functions

The minimization of submodular set functions is a generic optimization problem that contains a large class of important optimization problems as special cases [26]. Here we will see why the ellipsoid algorithm provides a polynomial-time solution method for submodular minimization.

DEFINITION 31.3 Let N be a finite set. A real valued set function f defined on the subsets of N is

- submodular if $f(X \cup Y) + f(X \cap Y) \leq f(X) + f(Y)$ for $X, Y \subseteq N$.

EXAMPLE 31.1:

Let $G = (V, E)$ be an undirected graph with V as the node set and E as the edge set. Let $c_{ij} \geq 0$ be the weight or capacity associated with edge $(ij) \in E$. For $S \subseteq V$, define the cut function $c(S) = \Sigma_{i \in S,\, j \in V \setminus S} c_{ij}$. The cut function defined on the subsets of V is submodular since $c(X) + c(Y) - c(X \cup Y) - c(X \cap Y) = \Sigma_{i \in X \setminus Y,\, j \in Y \setminus X} 2c_{ij} \geq 0$.

The optimization problem of interest is

$$\min\{f(X) : X \subseteq N\} .$$

The following remarkable construction that connects submodular function minimization with convex function minimization is due to Lovász (cf. [35]).

DEFINITION 31.4 *The Lovász extension* $\hat{f}(.)$ of a submodular function $f(.)$ satisfies

- $\hat{f} : [0, 1]^N \rightarrow \Re$.
- $\hat{f}(x) = \Sigma_{I \in \mathcal{I}} \lambda_I f(x_I)$ where $x = \Sigma_{I \in \mathcal{I}} \lambda_I x_I$, $x \in [0, 1]^N$, x_I is the incidence vector of I for each $I \in \mathcal{I}$, $\lambda_I > 0$ for each I in $\mathcal{I}$, and $\mathcal{I} = \{I_1, I_2, \cdots, I_k\}$ with $\emptyset \neq I_1 \subset I_2 \subset \cdots \subset I_k \subseteq N\}$. Note that the representation $x = \Sigma_{I \in \mathcal{I}} \lambda_I x_I$ is unique given that the $\lambda_I > 0$ and that the sets in $\mathcal{I}$ are nested.

It is easy to check that $\hat{f}(.)$ is a convex function. Lovász also showed that the minimization of the submodular function $f(.)$ is a special case of convex programming by proving

$$\min\{f(X) : X \subseteq N\} = \min\left\{\hat{f}(x) : x \in [0, 1]^N\right\} .$$

Further, if x^* is an optimal solution to the convex program and

$$x^* = \sum_{I \in \mathcal{I}} \lambda_I x_I ,$$

then for each $\lambda_I > 0$, it can be shown that $I \in \mathcal{I}$ minimizes f. The ellipsoid method can be used to solve this convex program (and hence submodular minimization) using a polynomial number of calls to an oracle for f (this oracle returns the value of $f(X)$ when input X).

31.7 Interior Point Methods

The announcement of the polynomial solvability of linear programming followed by the probabilistic analyses of the simplex method in the early 1980s left researchers in linear programming with a dilemma. We had one method that was good in a theoretical sense but poor in practice and another that was good in practice (and on average) but poor in a theoretical worst-case sense. This left the door wide open for a method that was good in both senses. Narendra Karmarkar closed this gap with a breathtaking new projective scaling algorithm. In retrospect, the new algorithm has been identified with a class of nonlinear programming methods known as logarithmic barrier methods. Implementations of a primal-dual variant of the logarithmic barrier method have proven to be the best approach at present. The recent monogragh by S.J. Wright [81] is dedicated to primal-dual interior point methods. It is a variant of this method that we describe below.

It is well known that moving through the interior of the feasible region of linear program using the negative of the gradient of the objective function, as the movement direction, runs into trouble because of getting "jammed" into corners (in high dimensions, corners make up most of the interior of a polyhedron). This jamming can be overcome if the negative gradient is balanced with a "centering" direction. The centering direction in Karmarkar's algorithm is based on the **analytic center** y_c of a full dimensional polyhedron $\mathcal{D} = \{x : A^T y \leq c\}$ which is the unique optimal solution to

$$\max \left\{ \sum_{j=1}^{n} \ln(z_j) : A^T y + z = c \right\} .$$

Recall the primal and dual forms of a linear program may be taken as

$$\begin{aligned} (P) \quad & \min \{cx : Ax = b,\ x \geq 0\} \\ (D) \quad & \max \left\{ b^T y : A^T y \leq c \right\} . \end{aligned}$$

The logarithmic barrier formulation of the dual (D) is

$$(D_\mu) \quad \max \left\{ b^T y + \mu \sum_{j=1}^{n} \ln (z_j) : A^T y + z = c \right\} .$$

Notice that (D_μ) is equivalent to (D) as $\mu \to 0^+$. The optimality (Karush–Kuhn–Tucker) conditions for (D_μ) are given by

$$\begin{aligned} D_x D_z e &= \mu e \\ Ax &= b \\ A^T y + z &= c , \end{aligned}$$

where D_x and D_z denote $n \times n$ diagonal matrices whose diagonals are x and z, respectively. Notice that if we set μ to 0, the above conditions are precisely the primal-dual optimality conditions; complementary

slackness, primal and dual feasibility of a pair of optimal (P) and (D) solutions. The problem has been reduced to solving the above equations in x, y, z. The classical technique for solving equations is Newton's method which prescribes the directions

$$\begin{aligned} \Delta y &= -\left(AD_x D_z^{-1} A^T\right)^{-1} AD_z^{-1}(\mu e - D_x D_z e) \\ \Delta z &= -A^T \Delta y \\ \Delta x &= D_z^{-1}(\mu e - D_x D_z e) - D_x D_z^{-1} \Delta z\,. \end{aligned} \tag{31.2}$$

The strategy is to take one Newton step, reduce μ, and iterate until the optimization is complete. The criterion for stopping can be determined by checking for feasibility $(x, z \geq 0)$ and if the duality gap $(x^t z)$ is close enough to 0. We are now ready to describe the algorithm.

Procedure: **Primal-Dual Interior**

0. **Initialize:**
   - $x_0 > 0$, $y_0 \in \Re^m$, $z_0 > 0, \mu_0 > 0, \epsilon > 0, \rho > 0$
   - $k := 0$
1. **Iterative Step:**

   do

   Stop `if` $Ax_k = b$, $A^T y_k + z_k = c$ `and` $x_k^T z_k \leq \epsilon$.

   $x_{k+1} \leftarrow x_k + \alpha_k^P \Delta x_k$

   $y_{k+1} \leftarrow y_k + \alpha_k^D \Delta y_k$

   $z_{k+1} \leftarrow z_k + \alpha_k^D \Delta z_k$

   `/*` Δx_k, Δy_k, Δz_k `are the Newton directions from (31.2) */`

   $\mu_{k+1} \leftarrow \rho \mu_k$

   $k := k + 1$

   od
2. **End**

Remarks:

1. The primal-dual algorithm has been used in several large-scale implementations. For appropriate choice of parameters, it can be shown that the number of iterations in the worst-case is $O(\sqrt{n} \log(\epsilon_0/\epsilon))$ to reduce the duality gap from ϵ_0 to ϵ [69, 81]. While this is sufficient to show that the algorithm is polynomial time, it has been observed that the "average" number of iterations is more like $O(\log n \log(\epsilon_0/\epsilon))$. However, unlike the simplex method we do not have a satisfactory theoretical analysis to explain this observed behavior.
2. The stepsizes α_k^P and α_k^D are chosen to keep x_{k+1} and z_{k+1} strictly positive. The ability in the primal-dual scheme to choose separate stepsizes for the primal and dual variables is a major computational advantage that this method has over the pure primal or dual methods. Empirically this advantage translates to a significant reduction in the number of iterations.

3. The stopping condition essentially checks for primal and dual feasibility and near complementary slackness. Exact complementary slackness is not possible with interior solutions. It is possible to maintain primal and dual feasibility through the algorithm, but this would require a Phase I construction via artificial variables. Empirically, this feasible variant has not been found to be worthwhile. In any case, when the algorithm terminates with an interior solution, a post-processing step is usually invoked to obtain optimal extreme point solutions for the primal and dual. This is usually called the purification of solutions and is based on a clever scheme described by Megiddo [56].
4. Instead of using Newton steps to drive the solutions to satisfy the optimality conditions of (D_μ), Mehrotra [59] suggested a predictor-corrector approach based on power series approximations. This approach has the added advantage of providing a rational scheme for reducing the value of μ. It is the predictor-corrector based primal-dual interior method that is considered the current winner in interior point methods. The OB1 code of Lustig, Marsten and Shanno [53] is based on this scheme. CPLEX 4.0 [18], a general purpose linear (and integer) programming solver, also contains implementations of interior point methods.
   Saltzman [70] describes a parallelization of the OB1 method to run on shared-memory vector multiprocessor architectures. Recent computational studies of parallel implementations of simplex and interior point methods on the SGI Power Challenge (SGI R8000) platform indicate that on all but a few small linear programs in the NETLIB linear programming benchmark problem set, interior point methods dominate the simplex method in run times. New advances in handling Cholesky factorizations in parallel are apparently the reason for this exceptional performance of interior point methods.
   As in the case of the simplex method, there are a number of special structures in the matrix A that can be exploited by interior point methods to obtain improved efficiencies. Network flow constraints, generalized upper bounds (GUB), and variable upper bounds (VUB) are structures that often come up in practice and which can be useful in this context [15, 79].
5. Interior point methods, like ellipsoid methods, do not directly exploit the linearity in the problem description. Hence they generalize quite naturally to algorithms for semidefinite and convex programming problems. More details of these generalizations are given in Chapter 33 of this *Handbook.* Kamath and Karmarkar [46] have proposed an interior-point approach for integer programming problems. The main idea is to reformulate an integer program as the minimization of a quadratic energy function over linear constraints on continuous variables. Interior-point methods are applied to this formulation to find local optima.

31.8 Strongly Polynomial Methods

The number of iterations and hence the number of elementary arithmetic operations required for the ellipsoid method as well as the interior point method is bounded by a polynomial function of the number of bits required for the binary representation of the input data. Recall that the size of a rational number a/b is defined as the total number of bits required in the binary representation of a and b. The dimension of the input is the number of data items in the input. An algorithm is said to be *strongly polynomial* if it consists of only elementary arithmetic operations (performed on rationals of size bounded by a polynomial in the size of the input) and the number of such elementary arithmetic operations is bounded by a polynomial in the dimension of the input.

It is an open question as to whether there exists a strongly polynomial algorithm for the general linear programming problem. However, there are some interesting partial results:

- Tardos [78] has devised an algorithm for which the number of elementary arithmetic operations is bounded by a polynomial function of n, m and the number of bits required for the

binary representation of the elements of the constraint matrix A which is $m \times n$. The number of elementary operations does not depend upon the right-hand side or the cost coefficients.

- Megiddo [57] described a strongly polynomial algorithm for checking the solvability of linear constraints with at most two non-zero coefficients per inequality. Later, Hochbaum and Naor [40] showed that Fourier elimination can be specialized to provide a strongly polynomial algorithm for this class.
- Megiddo [58] and Dyer [22] have independently designed strongly polynomial (linear-time) algorithms for linear programming in fixed dimensions. The number of operations for these algorithms is linear in the number of constraints and independent of the coefficients but doubly exponential in the number of variables.

The rest of this section details these three results and some of their consequences.

Combinatorial Linear Programming

Consider the linear program, (LP) $Max\{cx : Ax = b,\ x \geq 0\}$, where A is a $m \times n$ integer matrix. The associated dual linear program is $Min\ \{yb\ :\ y\ A \geq c\}$. Let L be the maximum absolute value in the matrix and let $\Delta = (nL)^n$. We now describe Tardos' algorithm for solving (LP) which permits the number of elementary operations to be free of the magnitudes of the "rim" coefficients b and c.

The algorithm uses Procedure 1 to solve a system of linear inequalities. Procedure 1, in turn, calls Procedure 2 with any polynomial-time linear programming algorithm as the required subroutine. Procedure 2 finds the optimal objective function value of a linear program and a set of variables which are zero in some optimal solution, if the optimum is finite. Note that Procedure 2 only finds the optimal objective value and not an optimal solution. The main algorithm also calls Procedure 2 directly with Subroutine 1 as the required subroutine. For a given linear program, Subroutine 1 finds the optimal objective function value and a dual solution, if one exists. Subroutine 1, in turn, calls Procedure 2 along with any polynomial-time linear programming algorithm as the required subroutine. We omit the detailed descriptions of the Procedures 1 & 2 and Subroutine 1 and instead only give their input/output specifications.

Algorithm: **Tardos**

INPUT: A linear programming problem $\max\{cx : Ax = b,\ x \geq 0\}$

OUTPUT: An optimal solution, if it exists and the optimal objective function value.

1. Call Procedure 1 to test whether $\{Ax = b,\ x \geq 0\}$ is feasible. If the system is not feasible, the optimal objective function value $= -\infty$, stop.
2. Call Procedure 1, to test whether $\{yA \geq c\}$ is feasible. If the system is not feasible, the optimal objective function value $= +\infty$, stop.
3. Call Procedure 2 with the inputs as the linear program $Max\{cx : Ax = b,\ x \geq 0\}$ and Subroutine 1 as the required subroutine. Let $x_i = 0,\ i \in K$ be the set of equalities identified.
4. Call Procedure 1 to find a feasible solution x^* to $\{Ax = b,\ x \geq 0,\ x_i = 0,\ i \in K\}$. The solution x^* is optimal and the optimal objective function value is cx^*
5. **End**

Specification of Procedure 1:

INPUT: A linear system $A\,x \leq b$, where A is a $m \times n$ matrix .

OUTPUT: Either $Ax \leq b$ is infeasible or $\hat{x}$ is a feasible solution.

Specification of Procedure 2:

INPUT: Linear program $Max\{cx : Ax = b,\ x \geq 0\}$, which has a feasible solution and a subroutine which for a given integer vector $\bar{c}$ with $\| \bar{c}\|_\infty \leq n^2\ \Delta$ and a set K of indices, determines $\max\{\bar{c}x : Ax = b,\ x \geq 0,\ x_i = 0,\ i \in K\}$ and if the maximum is finite, finds an optimal dual solution.

OUTPUT: The maximum objective function value z^* of the input linear program $\max\{c\ x\ :\ A\ x = b,\ x \geq 0,\}$ and the set K of indices such $x_i = 0,\ i \in K$ for some optimum solution to the input linear program.

Specification of Subroutine 1:

INPUT: A Linear program $\max\{\bar{c}x : Ax = b,\ x \geq 0,\ x_i = 0,\ i \in K\}$, which is feasible and $\| \bar{c}\|_\infty \leq n^2\ \Delta$.

OUTPUT: The Optimal objective function value z^* and an optimal dual solution y^*, if it exists.

The validity of the algorithm and the analysis of the number of elementary arithmetic operations required are in the paper by Tardos [78]. This result may be viewed as an application of techniques from *diophantine approximation* to linear programming. A scholarly account of these connections is given in the book by Schrijver [71].

REMARK 31.2 Linear programs with $\{0,\ \pm\ 1\}$ elements in the constraint matrix A arise in many applications of polyhedral methods in combinatorial optimization. Network flow problems (shortest path, maximum flow, and transshipment) [1] are examples of problems in this class. Such linear programs, and more generally linear programs with the matrix A made up of integer coefficients of bounded magnitude, are known as *combinatorial linear programs.* The algorithm described shows that combinatorial linear programs can be solved by strongly polynomial methods.

Fourier Elimination and $LI(2)$:

We now describe a special case of the linear programming solvability problem for which Fourier elimination yields a very efficient (strongly polynomial) algorithm. This is the case $LI(2)$ of linear inequalities with at most two variables per inequality. Nelson [63] observed that Fourier elimination is subexponential in this case. He showed that the number of inequalities generated never exceeds $O(mn^{\lceil \log n \rceil} \log n)$. Later Aspvall & Shiloach [4] obtained the first polynomial-time algorithm for solving $LI(2)$ using a graph representation of the inequalities. We give a high-level description of the technique of Hochbaum & Naor [40] that combines Fourier elimination and a graph reasoning technique to obtain the best known sequential complexity bounds for $LI(2)$.

An interesting property of $LI(2)$ systems is that they are closed under Fourier elimination. Therefore, the projection of an $LI(2)$ system on to a subspace whose coordinates are a subset of the variables is also an $LI(2)$ system. Note that $LI(3)$ does not have this closure property. Indeed $LI(3)$ is unlikely to have any special property, since any system of linear inequalities can be reduced to an instance of $LI(3)$ with $0, \pm 1$ coefficients [43].

Given an instance of $LI(2)$ of the form $Ax \leq b$ with each row of A containing at most two nonzero coefficients we construct a graph $G(V,E)$ as follows. The vertices V are $x_0, x_1, \cdots, x_n$ corresponding

to the variables of the constraints (x_0 is an extra dummy variable). The edges E of G are composed of pairs (x_i, x_j) if x_i and x_j are two variables with nonzero coefficients of at least one inequality in the system. There are also edges of the form (x_0, x_k) if x_k is the only variable with a nonzero coefficient in some constraint. Let us also assume that each edge is labelled with all the inequalities associated with its existence.

Aspvall & Shiloach [4] describe a "grapevine algorithm" that takes as input a "rumour" $x_j = \alpha$ and checks its authenticity, i.e., checks if α is too small, too large, or within the range of feasible values of x_j. The idea is simply to start at node x_j and set $x_j = \alpha$. Obviously, each variable x_k that is a neighbor of x_j in G gets an implied lower bound or upper bound (or both), depending on the sign of the coefficient of x_k in the inequality shared with x_j. These bounds get propagated further to neighbors of the x_k and so on. If this propagation is carried out in a breadth-first fashion, it is not hard to argue that the implications of setting $x_j = \alpha$ are completely revealed in $3n - 2$ stages. Proofs of inconsistency can be traced back to delineate if α was either too high or too low a value for x_j.

The grapevine algorithm is similar to Bellman & Ford's classical shortest path algorithm for graphs which also takes $O(mn)$ effort. This subroutine provides the classification test for being able to execute binary search in choosing values for variables. The specialization of Fourier's algorithm for $LI(2)$ can be described now.

Algorithm Fourier $LI(2)$:

For $j = 1, 2, \cdots n$

1. The inequalities of each edge (x_j, x_k) define a convex polygon Q_{jk} in x_j, x_k-space. Compute J_k the sorted collection of x_j coordinates of the corner (extreme) points of Q_{jk}. Let J denote the sorted union (merge) of the J_k (x_k a neighbor of x_j in G).
2. Perform a binary search on the sequence J for a feasible value of x_j. If we succeed in finding a feasible value for x_j among the values in J we fix x_j at that value and contract vertex x_j with x_0. Move to the next variable $j \leftarrow j + 1$ and repeat.
3. Else we know that the sequence is too coarse and that all feasible values lie in the strict interior of some interval $[x_j^1, x_j^2]$ defined by consecutive values in J. In this latter case we prune all but the two essential inequalities, defining the edges of the polygon Q_{jk}, for each of the endpoints x_j^1 and x_j^2.
4. Eliminate x_j using standard Fourier elimination.

End

Notice that at most four new inequalities are created for each variable elimination. Also note that the size of J is always $O(m)$. The complexity is dominated by the search over J. Each search step requires a call to the "grapevine" procedure and there are at most $n \log m$ calls. Therefore the overall time-complexity is $O(mn^2 \log m)$, which is strongly polynomial in that it is polynomial and independent of the size of the input coefficients.

An open problem related to $LI(2)$ is the design of a strongly polynomial algorithm for optimization of an arbitrary linear function over $LI(2)$ constraints. This would imply, via duality, a strongly polynomial algorithm for generalized network flows (flows with gains and losses).

Fixed Dimensional LPs: Prune and Search

Consider the linear program $\max\{cx : Ax \leq b\}$ where A is a $m \times n$ matrix that includes the nonnegativity constraints. Clearly, for fixed dimension n, there is a polynomial-time algorithm because there are at most $\binom{m}{n}$ system of linear equations to be solved, to generate all extreme points of the feasible region. However, Megiddo [56] and Dyer [22] have shown that for the above linear program with fixed n, there is a linear-time algorithm that requires $O(m)$ elementary arithmetic operations on numbers of size bounded by a polynomial in the input data. The algorithm is highly recursive. Before we give an outline of the algorithm, some definitions are required.

DEFINITION 31.5 Given a linear program $\max\{cx : Ax \leq b\}$ and a linear equality $fx = q$,

(i) The inequality $fx < q$ is said to hold for the optimum if either

(a) We know that $Ax \leq b$ is feasible and
$\max\{cx : Ax \leq b, fx \leq q\} > \max\{cx : Ax \leq b, fx = q\}$

or

(b) We know a row vector $y \geq 0$ such that $yA = f$ and $yb < q$.

(ii) The inequality $fx > q$ is said to hold for the optimum if either

(a) We know that $Ax \leq b$ is feasible and
$\max\{cx : Ax \leq b, fx \geq q\} > \max\{cx : Ax \leq b, fx = q\}$

or

(b) We know a vector $y \geq 0$ such that $yA = -f$ and $yb < -q$.

DEFINITION 31.6 For a given linear program $\max\{cx : Ax \leq b\}$ and a given linear equation $fx = q$, the position of the optimum of the linear program relative to the linear equation is said to be known if either we know that $fx < q$ holds for an optimum or $fx > q$ holds for an optimum.

An outline of the algorithm is presented below. The algorithm requires an oracle, denoted as Procedure 1, with inputs as the linear program $\max\{cx : Ax \leq b\}$ where A is a $m \times n$ matrix and a system of p linear equations $Fx = d$ with the rank of F being r. The output of Procedure 1 is either a solution to the linear program (possibly unbounded or infeasible) or a set of $\lceil p/2^{2^{r-1}} \rceil$ equations in $Fx = d$ relative to each of which we know the position of the optimum of the linear program.

`Algorithm Sketch:` **Prune & Search**

`Call Procedure 1 with inputs as the linear program` $\max\{cx : Ax \leq b\}$ `and the system of` m `equations` $Ax = b$. `Procedure 1 either solves the linear program or identifies` $k = \lceil m/2^{2^{n-1}} \rceil$ `equations in` $Ax = b$ `relative to each of which we know the position of the optimum of the linear program. The identified equations are then omitted from the system` $Ax = b$. `The resulting subsystem,` $A_1x = b_1$ `has` $m_1 = m - k$ `equations. Procedure 1 is applied again with the original given linear program and the system of equations` $A_1x = b_1$ `as the inputs. This process is repeated until either the linear program is solved or we know the position of the optimum with respect to each of the equations in` $Ax = b$. `In the latter case the system` $Ax \leq b$ `is infeasible.`

We next describe the input/output specification of Procedure 1. The procedure is highly recursive and splits into a lot of cases. This procedure requires a linear-time algorithm for the identification of the median of a given set of rationals in linear time.

`Specification of Procedure 1:`

> INPUT: `Linear program` $\max\{cx : Ax \leq b\}$ `where` A `is a` $m \times n$ `matrix and a system of` p `equations` $Fx = d$ `where rank of` F `is` r.
>
> OUTPUT: `A solution to the linear program or a set of` $\lceil p/2^{2^{r-1}} \rceil$ `equations in` $Fx = d$ `relative to each of which we know the position of the optimum as in Definition 31.6.`

For fixed n, Procedure 1 requires $O(p + m)$ elementary arithmetic operations on numbers of size bounded by a polynomial in the size of the input data. Since at the outset $p = m$, algorithm `Prune & Search` solves linear programs with fixed n in linear time. Details of the validity of the algorithm as well as analysis of its linear time complexity for fixed n are given by Megiddo [56], Dyer [22], and in the book by Schrijver [71]. As might be expected, the linear-time solvability of linear programs in fixed dimension has important implications in the field of computational geometry which deals largely with two and three dimensional geometry. The book by Edelsbrunner [25] documents these connections.

The linear programming problem is known to be $\mathcal{P}$-complete and therefore we do not expect to find a parallel algorithm that achieves polylog run time. However, for fixed n, there are simple polylog algorithms [23]. In a recent paper, Sen [72] shows that linear programming in fixed dimension n can be solved in $O(\log \log^{n+1} m)$ steps using m processors in a *CRCW PRAM*.

31.9 Randomized Methods for Linear Programming

The advertising slogan for randomized algorithms has been "simplicity and speed" [60]. In the case of fixed-dimensional linear programming there appears to be some truth in the advertisement. In stark contrast with the very technical deterministic algorithm outlined in the last section, we will see that an almost trivial randomized algorithm will achieve comparable performance (but of course at the cost of determinism).

Consider a linear programming problem of the form

$$\min\{cx : Ax \leq b\},$$

with the following properties:

- The feasible region $\{x : Ax \leq b\}$ is nonempty and bounded.
- The objective function c has the form $(1, 0, \ldots, 0)$.
- The minimum to the linear program is unique and occurs at an extreme point of the feasible region.
- Each vertex of $\{x : Ax \leq b\}$ is defined by exactly n constraints where A is $m \times n$.

Note that none of these assumptions compromise the generality of the linear programming problem that we are considering.

Let $\mathcal{S}$ denote the constraints $Ax \leq b$. A feasible $B \subseteq \mathcal{S}$ is called optimal if it defines the uniquely optimal extreme point of the feasible region. The following randomized algorithm due to Sharir and Welzl [74] uses an incremental technique to obtain the optimal basis of the input linear program.

Algorithm: **ShW**

INPUT: The constraint set $\mathcal{S}$ and a feasible basis T.
OUTPUT: The optimal basis for the linear program.

1. If $\mathcal{S}$ equals T, return T;
2. Pick a random constraint $s \in \mathcal{S}$. Now define $\bar{T} = \mathbf{ShW}(\mathcal{S} \setminus \{s\}, T)$;
3. If the point defined by $\bar{T}$ satisfies s, output $\bar{T}$; Else output $\mathbf{ShW}(\mathcal{S}, \mathbf{opt}(\{s\} \cup \bar{T}))$
4. **End**

The subroutine **opt** when given an input of $n+1$ or less constraints $H \subseteq \mathcal{S}$ returns an optimal basis for the linear program with constraints defined by H (and objective function cx). It has been shown [55] that algorithm **ShW** has an expected running time of $O(\min\{mn \exp \sqrt[4]{n \ln (m+1)}, \quad n^4 2^n m\})$. Thus algorithm **ShW** is certainly linear expected time for fixed n but has a lower complexity than `Prune & Search` if n is allowed to vary.

31.10 Large-Scale Linear Programming

Linear programming problems with thousands of rows and columns are routinely solved either by variants of simplex method or by interior point methods. However, for several linear programs that arise in combinatorial optimization, the number of columns (or rows in the dual) are too numerous to be enumerated explicitly. The columns, however, often have a structure which is exploited to generate the columns as and when required in the simplex method. Such an approach, which is referred to as **column generation,** is illustrated next on the *cutting stock problem* (Gilmore and Gomory [32]), which is also known as the *bin packing problem* in the computer science literature.

Cutting Stock Problem

Rolls of sheet metal of standard length L are used to cut required lengths $l_i, i = 1, 2 \ldots, m$. The jth cutting pattern should be such that a_{ij}, the number of sheets length l_i cut from one roll of standard length L must satisfy $\Sigma_{i=1}^{m} a_{ij} l_i \leq L$. Suppose $n_i, i = 1, 2, \ldots, m$ sheets of length l_i are required. The problem is to find cutting patterns so as to minimize the number of rolls of standard length L that are used to meet the requirements. A linear programming formulation of the problem is as follows:

Let $x_j, \quad j = 1, 2, \ldots, n$ denote the number of times the jth cutting pattern is used. In general, $x_j, \quad j = 1, 2, \ldots, n$ should be an integer but in the formulation below the variables are permitted to be fractional.

$$\begin{array}{rll} (P1) & \min \sum_{j=1}^{n} x_j & \\ \text{Subject to} & \sum_{j=1}^{n} a_{ij} x_j \geq n_i & i = 1, 2, \ldots, m \\ & x_j \geq 0 & j = 1, 2, \ldots, n \\ \text{where} & \sum_{i=1}^{m} l_i a_{ij} \leq L & j = 1, 2, \ldots, n\,. \end{array}$$

The formulation can easily be extended to allow for the possibility of p standard lengths, L_k, $k = 1, 2, \ldots, p$ from which the n_i units of length $l_i, i = 1, 2, \ldots, m$ are to be cut.

The cutting stock problem can also be viewed as a bin packing problem. Several bins, each of standard capacity L are to be packed with n_i units of item i, each of which uses up capacity of l_i in a bin. The problem is to minimize the number of bins used.

Column Generation

In general, the number of columns in $(P1)$ is too large to enumerate all the columns explicitly. The simplex method, however, does not require all the columns to be explicitly written down. Given a basic feasible solution and the corresponding simplex multipliers $w_i, i = 1, 2, \ldots, m$, the column to enter the basis is determined by applying dynamic programming to solve the following knapsack problem:

$$
\begin{aligned}
(P2) \quad z = \max \quad & \sum_{i=1}^{m} w_i a_i \\
\text{Subject to} \quad & \sum_{i=1}^{m} l_i a_i \leq L \\
& a_i \geq 0 \text{ and integer } i = 1, 2, \ldots, m \, .
\end{aligned}
$$

Let $a_i^*, i = 1, 2, \ldots, m$ denote an optimal solution to $(P2)$. If $z > 1$, the kth column to enter the basis has coefficients $a_{ik} = a_i^*, i = 1, 2, \ldots, m$.

Using the identified columns, a new improved (in terms of the objective function value) basis is obtained and the column generation procedure is repeated. A major iteration is one in which $(P2)$ is solved to identify, if there is one, a column to enter the basis. Between two major iterations, several minor iterations may be performed to optimize the linear program using only the available (generated) columns.

If $z \leq 1$ the current basic feasible solution is optimal to $(P1)$. From a computational point of view, alternative strategies are possible. For instance, instead of solving $(P2)$ to optimality, a column to enter the basis can be identified as soon as a feasible solution to $(P2)$ with an objective function value greater than 1 has been found. Such an approach would reduce the time required to solve $(P2)$ but may increase the number of iterations required to solve $(P1)$.

A column, once generated may be retained, even if it comes out of the basis at a subsequent iteration, so as to avoid generating the same column again later on. However, at a particular iteration some columns that appear unattractive in terms of their reduced costs may be discarded in order to avoid having to store a large number of columns. Such columns can always be generated again subsequently, if necessary. The rationale for this approach is that such unattractive columns will rarely be required subsequently.

The dual of $(P1)$ has a large number of rows. Hence column generation may be viewed as row generation in the dual. In other words, in the dual we start with only a few constraints explicitly written down. Given an optimal solution w to the current dual problem (i.e., with only a few constraints which have been explicitly written down) find a constraint that is violated by w or conclude that no such constraint exists. The problem to be solved for identifying a violated constraint, if any, is exactly the separation problem that we encountered in Section 31.6.

Decomposition

Large-scale linear programming problems sometimes have a block diagonal structure. Consider for instance, the following linear program:

$$
\begin{aligned}
(P3) \quad & \max \sum_{j=1}^{p} c^j x^j && (31.3) \\
\text{Subject to} \quad & \sum_{j=1}^{p} A^j x^j = b && (31.4) \\
& D^j x^j = d^j \quad j = 2, 3, \ldots, p && (31.5) \\
& x^j \geq 0 \quad j = 1, 2, \ldots, p \, , && (31.6)
\end{aligned}
$$

where A^j is a $m \times n_j$ matrix; D^j is a $m_j \times n_j$ matrix; x_j is a $n_j \times 1$ column vector; c^j is a $1 \times n_j$ row vector; b is a $m \times 1$ column vector; d^j is a $m_j \times 1$ column vector.

The constraints (31.4) are referred to as the linking master constraints. The p sets of constraints (31.5) and (31.6) are referred to as subproblem constraints. Without the constraints (31.4), the problem decomposes into p separate problems which can be handled in parallel. The Dantzig–Wolfe [21] decomposition approach to solving $(P3)$ is described next.

Clearly, any feasible solution to $(P3)$ must satisfy constraints (31.5) and (31.6). Now consider the polyhedron P_j, $j = 2, 3, \ldots, p$ defined by the constraints (31.5) and (31.6). By the representation theorem of polyhedra (see Section 2) a point $x^j \in P_j$ can be written as

$$\begin{aligned} x^j &= \sum_{k=1}^{h_j} x^{jk}\rho_{jk} + \sum_{k=1}^{g_j} y^{jk}\mu_{jk} \\ \sum_{k=1}^{h_j} \rho_{jk} &= 1 \\ \rho_{jk} &\geq 0 \quad k = 1, 2, \ldots, h_j \\ \mu_{jk} &\geq 0 \quad k = 1, 2, \ldots, g_j \ , \end{aligned}$$

where x^{jk} $(k = 1, 2, \ldots, h_j)$ are the extreme points and y^{jk} $(k = 1, 2, \ldots, g_j)$ are the extreme rays of P_j.

Now substituting for x^j, $j = 2, 3, \ldots, p$ in (31.3) and (31.4), $(P3)$ is written as

$$(P4) \qquad \max \left\{ c^1x^1 + \sum_{j=2}^{p} \left[\sum_{k=1}^{h_j} \left(c^j x^{jk} \right) \rho_{jk} + \sum_{k=1}^{g_j} \left(c^j y^{jk} \right) \mu_{jk} \right] \right\}$$

$$\text{Subject to} \qquad A^1x^1 + \sum_{j=2}^{p} \left[\sum_{k=1}^{h_j} \left(A^j x^{jk} \right) \rho_{jk} + \sum_{k=1}^{g_j} \left(A^j y^{jk} \right) \mu_{jk} \right] = b \tag{31.7}$$

$$\sum_{k=1}^{h_j} \rho_{jk} = 1 \quad j = 2, 3, \ldots, p \tag{31.8}$$

$$\rho_{jk} \geq 0 \quad j = 2, 3, \ldots, p; \quad k = 1, 2, \ldots, h_j$$

$$\mu_{jk} \geq 0 \quad j = 2, 3, \ldots, p; \quad k = 1, 2, \ldots, g_j \ .$$

In general, the number of variables in $(P4)$ is an exponential function of the number of variables n_j, $j = 1, 2, \ldots, p$ in $(P3)$. However, if the simplex method is adapted to solve $(P4)$, the extreme points or the extreme rays of P_j, $j = 2, 3, \ldots, p$, and consequently the columns in $(P4)$ can be generated, as and when required, by solving the linear programs associated with the p subproblems. This column generation is described next.

Given a basic feasible solution to $(P4)$, let w and u be row vectors denoting the simplex multipliers associated with constraints (31.7) and (31.8), respectively. For $j = 2, \ldots, p$, solve the following linear programming subproblems:

$$\begin{aligned} (S_j) \qquad z_j &= \min \left(wA^j - c^j \right) x^j \\ D^j x^j &= d^j \\ x^j &\geq 0 \ . \end{aligned}$$

Suppose z_j is finite. An extreme point solution x^{jt} is then identified. If $z_j + u_j < 0$, a candidate column to enter the basis is given by $\begin{pmatrix} A^j x^{jt} \\ 1 \end{pmatrix}$. On the other hand if $z_j + u_j \geq 0$, there is no extreme point of P_j that gives a column to enter the basis at this iteration. Suppose the optimal solution to S_j is unbounded. An extreme ray y^{jt} of P_j is then identified and a candidate column to enter the basis is given by $\begin{pmatrix} A^j y^{jt} \\ 0 \end{pmatrix}$. If the simplex method is used to solve S_j, the extreme point or the extreme ray is identified automatically. If a column to enter the basis is identified from any of the subproblems, a new improved basis is obtained and the column generation procedure is repeated. If none of the subproblems identify a column to enter the basis, the current basic solution to $(P4)$ is optimal.

As in "Cutting Stock Problem," a major iteration is when the subproblems are solved. Instead of solving all the p subproblems at each major iteration, one option is to update the basis as soon as a column to enter the basis has been identified in any of the subproblems. If this option is used at each major iteration, the subproblems that are solved first are typically the ones that were not solved at the previous major iteration.

The decomposition approach is particularly appealing if the subproblems have a special structure that can be exploited. Note that only the objective functions for the subproblems change from one major iteration to another. Given the current state of the art, $(P4)$ can be solved in polynomial time (polynomial in the problem parameters of $(P3)$) by the ellipsoid method but not by the simplex method or interior point methods. However $(P3)$ can be solved in polynomial time by interior point methods.

It is interesting to note that the reverse of decomposition is also possible. In other words, suppose we start with a statement of a problem and an associated linear programming formulation with a large number columns (or rows in the dual). If the column generation (or row generation in the dual) can be accomplished by solving a "compact" linear program, then a "compact" formulation of the original problem can be obtained. Here "compact" refers to the number of rows and columns being bounded by a polynomial function of the parameters (not the number of the columns in the original linear programming formulation) in the statement of the original problem. This result due to Martin [54] enables one to solve the problem in the polynomial time by solving the compact formulation directly using interior point methods.

Compact Representation

Consider the following linear program:

$$\begin{aligned} (P5) \qquad & \min\ cx \\ \text{Subject to} \quad & Ax \geq b \\ & x \geq 0 \\ x \quad & \in \bigcap_{j=1}^{p} P_j\ , \end{aligned}$$

where A is a $m \times n$ matrix; x is a $n \times 1$ vector; c is a $1 \times n$ vector and b is a $m \times 1$ vector; $P_j,\ j = 1, 2, \ldots, p$ is a polyhedron and p is bounded by a polynomial in m and n.

Without loss of generality, it is assumed that $P_j,\ j = 1, 2, \ldots, p$ is nonempty and $(P5)$ is feasible. Given $\bar{x}$ such that $A\bar{x} \geq b$, the constraint identification or separation problem $S_j(\bar{x})$ with respect to P_j is to *either* (a) conclude that $\bar{x} \in P_j$, *or* (b) find a valid inequality $D_i^j x \leq d_i^j$ that is satisfied by $x \in P_j$ but $D_i^j \bar{x} > d_i^j$.

Suppose the separation problem $S_j(\bar{x})$ can be solved by the following linear program

$$\begin{aligned} S_j(\bar{x}) : z_j \quad &= \quad \max\ \left(\bar{x}^T G^j + g^j\right) y \\ F^j y \quad &\leq \quad f^j \\ y \quad &\geq \quad 0\ , \end{aligned}$$

where

G^j is an $n \times k$ matrix; F^j is a $r \times k$ matrix;

g^j is a $1 \times k$ vector; f^j is a $r \times 1$ vector;

y is a $k \times 1$ vector; r and k are bounded by a polynomial in m and n; and

$\bar{x} \in P_j \bigcap \{x : Ax \geq b\}$ if and only if $z_j \leq h^j \bar{x} + k_j$ where h^j is a $1 \times n$ vector and k_j is a scalar.

Now, if w^j denotes the dual variables associated with the $F^j y \leq f^j$ constraints in S_j, it follows from the duality theorem of linear programming that a **compact representation** of $(P5)$ is given by

$$\begin{array}{rcl} \min cx & & \\ \text{Subject to} \quad Ax & \geq & b \\ (F^j)^T w^j - (G^j)^T x & \geq & (g^j)^T \quad j = 1, 2, \ldots p \\ (f^j)^T w^j - h^j x & \leq & k_j \quad j = 1, 2, \ldots p \\ x & \geq & 0 \\ w^j & \geq & 0 \quad j = 1, 2, \ldots, p\,. \end{array}$$

Note that this approach to obtaining a compact formulation is predicated on being able to formulate the separation problem as a compact linear program. This may not always be possible. In fact, Yannakakis [82] shows that for a b-matching problem under a symmetry assumption, no compact formulation is possible. This despite the fact that b-matching can be solved in polynomial time using a polynomial-time separation oracle.

An Application: Neural Net Loading

The decision version of the Hopfield neural net loading problem (cf. [64]) is as follows.

Given y^i $(i = 1, 2, \ldots, p)$ where each y^i is an n-dimensional training vector whose components are $(+1, -1)$, construct a symmetric $n \times n$ synaptic weight matrix W, such that for every n-dimensional vector v whose components are $(+1, -1)$, the following holds:

$$\text{If } d\left(y^i, v\right) \leq k \text{ then } y^i = sgn(Wv) \text{ for } i = 1, 2, \ldots, p\,.$$

A feasible W would represent a Hopfield net with a radius of direct attraction of at least k around each training vector, i.e., a robust network of associative memory. Here k is specified and $d(y^i, v)$ is the Hamming distance between y^i and v. For $t = 1, 2, \ldots, p$, let $v^{tq}, q = 1, 2, \ldots, m_t$ be all the vectors whose components are $(+1, -1)$ and $d(y^t, v^{tq}) \leq k$. The Hopfield neural net loading problem is to find a matrix W such that

$$(P6) \quad \sum_{j=1}^{n} y_i^t w_{ij} v_j^{tq} \geq 1 \quad \text{for } i = 1, 2, \ldots, n;\ t = 1, 2, \ldots, p;\ \text{and } q = 1, 2, \ldots, m_t\,.$$

This is a linear program with the number of inequalities equal to $pn\binom{n}{k}$, which is huge. The synaptic weights have to be symmetric, so in addition to the inequalities given above, $(P6)$ would include the constraints $w_{ij} = w_{ji}$ for $i = 1, 2, \ldots, n$ and $j = 1, 2, \ldots, n$.

Given the weights $\bar{w_{uv}}, u = 1, 2, \ldots, n, v = 1, 2, \ldots, n$, the separation problem, $S_{it}(\bar{w})$ for specified i and t, where $1 \leq i \leq n$ and $1 \leq t \leq p$, can be formulated as follows [13]:

Let

$$u_{ij}^{tq} = \begin{cases} 1 & \text{if } v_j^{tq} = -y_j^t \\ 0 & \text{if } v_j^{tq} = y_j^t\,. \end{cases}$$

Since $d(y^t, v^{tq}) \leq k$ it follows that

$$\sum_{j=1}^{n} u_{ij}^{tq} \leq k \quad q = 1, 2, \ldots, m_t\,.$$

Note that for specified i, t, and q, the inequality in $(P6)$ is equivalent to

$$\sum_{j=1}^{n} y_i^t \, w_{ij} \left[y_j^t \left(-u_{ij}^{tq} \right) + y_j^t \left(1 - u_{ij}^{tq} \right) \right] \geq 1 \, ,$$

which reduces to

$$-\sum_{j=1}^{n} 2 y_i^t \, y_j^t \, w_{ij} \, u_{ij}^{tq} + \sum_{j=1}^{n} y_i^t \, y_j^t \, w_{ij} \geq 1 \, .$$

Consequently, the separation problem after dropping the superscript q is

$$S_{it}\,(\bar{w}) : z_{it} = \max \sum_{j=1}^{n} 2 \, y_i^t \, y_j^t \, \bar{w}_{ij} \, u_{ij}^t$$

$$\text{Subject to} \quad \sum_{j=1}^{n} u_{ij}^t \leq k \tag{31.9}$$

$$0 \leq u_{ij}^t \leq 1 \quad j = 1, 2, \ldots, n \, . \tag{31.10}$$

Note that $S_{it}(\bar{w})$ is trivial to solve and always has a solution such that $u_{ij}^t = 0$ or 1. It follows that for given i and t, $\bar{w}$ satisfies the inequalities in $(P6)$ corresponding to $q = 1, 2, \ldots, m_t$ if and only if

$$z_{it} \leq \sum_{j=1}^{n} y_i^t \, y_j^t \, \bar{w}_{ij} - 1 \, .$$

Let θ_{it} and β_{ij}^t, $j = 1, 2, \ldots, n$ denote the dual variables associated with constraints (31.9) and (31.10), respectively. Now applying the compact representation result stated above, a compact formulation of the neural net loading problem $(P6)$ is as follows:

$$\begin{aligned}
\theta_{it} + \beta i j^t - 2 y_i^t y_j^t w_{ij} &\geq 0 \quad i = 1, 2, \ldots, n \quad t = 1, 2, \ldots, p \quad j = 1, 2, \ldots, n \\
k\,\theta_{it} + \sum_{j=1}^{n} \beta i j^t - \sum_{j=1}^{n} y_i^t y_j^t w_{ij} &\leq -1, \quad i = 1, 2, \ldots, n \quad t = 1, 2, \ldots, p \\
\theta_{it} &\geq 0 \quad i = 1, 2, \ldots, n; \quad t = 1, 2, \ldots, p \\
\beta_{ij}^t &\geq 0 \quad i = 1, 2, \ldots, n; \quad t = 1, 2, \ldots, p; \quad j = 1, 2, \ldots, n \, .
\end{aligned}$$

With the symmetry condition $w_{ij} = w_{ji}$ for $i = 1, 2, \ldots, n$ and $j = 1, 2, \ldots, n$ added in, we now have a linear programming formulation of the Hopfield net loading problem that is compact in the size of the network $n \times n$ and the size of the training set $n \times p$.

31.11 Linear Programming: A User's Perspective

This chapter has been written for readers interested in learning about the algorithmics of linear programming. However, for someone who is primarily a user of linear programming software, there are a few important concerns that we address briefly here.

1. THE EXPRESSION OF LINEAR PROGRAMMING MODELS. The data sources from which the coefficients of a linear programming model are generated may be organized in a format that is incompatible with the linear programming software in use. Tools to facilitate this translation have come to be known as "matrix generators" [28]. Over the years such tools have evolved into more complete modeling languages (for example, AMPL [29]), and GAMS [6].

2. THE VIEWING, REPORTING, AND ANALYSIS OF RESULTS. This issue is similar to that of model expression. The results of a linear programming problem when presented as raw numerical output are often difficult for a user to digest. Report writers and modeling languages like the ones mentioned above usually provide useful features for processing the output into a user-friendly form. Because of the widespread use of linear programming in decision support, user interfaces based on spreadsheets have become popular with software vendors [73].
3. TOOLS FOR ERROR DIAGNOSIS AND MODEL CORRECTION. Many modeling exercises using linear programming involve a large amount of data and are prone to numerical as well as logical errors. Some sophisticated tools [36] are now available for helping users in this regard.
4. SOFTWARE SUPPORT FOR LINEAR PROGRAMMING MODEL MANAGEMENT. Proliferation of linear programming models can occur in practice because of several reasons. The first is that when a model is being developed, it is likely that several versions are iterated on before converging to a suitable model. Scenario analysis is the second type of source for model proliferation. Finally, iterative schemes such as those based on column generation or stochastic linear programming may require the user to develop a large number of models. The software support in optimization systems for helping users in all such situations have come to be known as tools for "model management" [28, 31].
5. THE CHOICE OF A LINEAR PROGRAMMING SOLUTION PROCEDURE. For linear programs with special structure (for example, network flows [1]) it pays to use specialized versions of linear programming algorithms. In the case of general linear optimization software, the user may be provided a choice of a solution method from a suite of algorithms. While the right choice of an algorithm is a difficult decision, we hope that insights gained from reading this chapter would help the user.

31.12 Defining Terms

Analytic center: The interior point of a polytope at which the product of the distances to the facets is uniquely maximized.

Column generation: A scheme for solving linear programs with a huge number of columns.

Compact representation: A polynomial size linear programming representation of an optimization problem.

Decomposition: A strategy of divide and conquer applied to large scale linear programs.

Duality: The relationship between a linear program and its dual.

Extreme point: A corner point of a polyhedron.

Linear program: Optimization of a linear function subject to linear equality and inequality constraints.

Matrix factorization: Representation of a matrix as a product of matrices of simple form.

Polyhedral cone: The set of solutions to a finite system of homogeneous linear inequalities on real-valued variables.

Polyhedron: The set of solutions to a finite system of linear inequalities on real-valued variables. Equivalently, the intersection of a finite number of linear half-spaces in $\Re^n$.

Polytope: A bounded polyhedron.

Relaxation: An enlargement of the feasible region of an optimization problem. Typically, the relaxation is considerably easier to solve than the original optimization problem.

Separation: Test if a given point belongs to a convex set and if it does not, identify a separating hyperplane.

Strongly polynomial: Polynomial algorithms with the number of elementary arithmetic operations bounded by a polynomial function of the dimensions of the linear program.

References

[1] Ahuja, R.K., Magnanti, T.L., and Orlin, J.B., *Network Flows,* Prentice Hall, 1993.

[2] Akgul, M., *Topics in Relaxation and Ellipsoidal Methods,* Research Notes in Mathematics, Pitman Publishing Ltd., 1984.

[3] Alizadeh, F., Interior point methods in semidefinite programming with applications to combinatorial optimization, *SIAM Journal on Optimization,* 5(1), 13–51, Feb. 1995.

[4] Aspvall, B.I. and Shiloach, Y., A polynomial time algorithm for solving systems of linear inequalities with two variables per inequality, *FOCS,* 205–217, 1979.

[5] Bertsimas, D. and Tsitsiklis, J.N., *Introduction to Linear Optimization,* Athena Scientific, MA, 1997.

[6] Bisschop, J. and Meerhaus, A., On the development of a General Algebraic Modeling System (GAMS) in a strategic planning environment, *Mathematical Programming Study,* 20, 1–29, 1982.

[7] Bixby, R.E., Progress in linear programming, *ORSA Journal on Computing,* 6(1), 15–22, 1994.

[8] Bland, R., Goldfarb, D., and Todd, M.J., The ellipsoid method: a survey, *Operations Research,* 29, 1039–1091, 1981.

[9] Borgwardt, K.H., *The Simplex Method: A Probabilistic Analysis,* Springer-Verlag, Berlin, 1987.

[10] Černikov, R.N., The solution of linear programming problems by elimination of unknowns, *Doklady Akademii Nauk,* 139, 1314–1317, 1961. Translation in *Soviet Mathematics Doklady,* 2, 1099–1103, 1961.

[11] Caratheodory, C., Über den Variabiltätsbereich der Fourierschen Konstanten von positiven harmonischen Funktionen, *Rendiconto del Circolo Matematico di Palermo,* 32, 193–217, 1911.

[12] Chandru, V., Variable elimination in linear constraints, *The Computer Journal,* 36(5), 463–472, Aug. 1993.

[13] Chandru, V., Dattasharma, A., Keerthi, S.S., Sancheti, N.K, and Vinay, V., Algorithms for the design of optimal discrete neural networks, In *Proceedings of the Sixth ACM/SIAM Symposium on Discrete Algorithms,* SIAM Press, Jan. 1995.

[14] Chandru. V. and Kochar, B.S., A class of algorithms for linear programming, *Research Memorandum 85–14,* School of Industrial Engineering, Purdue University, Nov. 1985.

[15] Chandru, V. and Kochar, B.S., Exploiting special structures using a variant of Karmarkar's algorithm, *Research Memorandum 86–10,* School of Industrial Engineering, Purdue University, Jun. 1986.

[16] Chvatal, V., *Linear Programming,* Freeman Press, New York, 1983.

[17] Cook, W., Lovász, L., and Seymour, P., Eds., *Combinatorial Optimization: Papers from the DIMACS Special Year,* Series in Discrete Mathematics and Theoretical Computer Science, Volume 20, AMS, 1995.

[18] CPLEX, *Using the CPLEX Callable Library and CPLEX Mixed Integer Library,* CPLEX Optimization, 1993.

[19] Dantzig, G.B., Maximization of a linear function of variables subject to linear inequalities, In *Activity Analysis of Production and Allocation,* C. Koopmans, Ed., John Wiley & Sons, New York, 339–347, 1951.

[20] Dantzig, G.B., *Linear Programming and Extensions,* Princeton University Press, Princeton, NJ, 1963.

[21] Dantzig, G.B. and Wolfe, P., The decomposition algorithm for linear programming, *Econometrica,* 29, 767–778, 1961.
[22] Dyer, M.E., Linear time algorithms for two- and three-variable linear programs, *SIAM Journal on Computing,* 13, 31–45, 1984.
[23] Dyer, M.E., A parallel algorithm for linear programming in fixed dimension, *Proceedings of the 11th Annual ACM Symposium on Computational Geometry,* ACM Press, 345–349, 1995.
[24] Ecker, J.G. and Kupferschmid, M., An ellipsoid algorithm for nonlinear programming, *Mathematical Programming,* 27, 1983.
[25] Edelsbrunner, M., *Algorithms in Combinatorial Geometry,* Springer-Verlag, 1987.
[26] Edmonds, J., Submodular functions, matroids and certain polyhedra, In *Combinatorial Structures and Their Applications,* R. Guy et al., Eds., Gordon Breach, 69–87, 1970.
[27] Farkas, Gy., A Fourier-féle mechanikai elv alkalmazásai (in Hungarian), *Mathematikai és Természettudományi Értesitö,* 12, 457–472, 1894.
[28] Fourer, R., Software for Optimization: A Buyer's Guide (Parts I and II), in *INFORMS Computer Science Technical Section Newsletter,* 17(1/2), 1996.
[29] Fourer, R., Gay, D.M., and Kernighian, B.W., *AMPL: A Modeling Language for Mathematical Programming,* Scientific Press, 1993.
[30] Fourier, L.B.J., reported in Analyse des travaux de l'Academie Royale des Sciences, pendant l'annee 1824, Partie mathematique, *Histoire de l'Academie Royale des Sciences de l'Institut de France 7,* 1827, xlvii-lv. (Partial English Translation in D.A. Kohler, Translation of a Report by Fourier on his Work on Linear Inequalities, *Opsearch,* 10, 38–42, 1973).
[31] Geoffrion, A.M., An introduction to structured modeling, *Management Science,* 33, 547–588, 1987.
[32] Gilmore, P. and Gomory, R.E., A linear programming approach to the cutting stock problem, Part I, *Operations Research,* 9, 849–854; Part II, *Operations Research,* 11, 863–887, 1963.
[33] Goemans, M.X. and Williamson, D.P., 878 approximation algorithms MAX CUT and MAX 2SAT, in *Proceedings of ACM STOC,* 422–431, 1994.
[34] Grötschel, M., Lovasz, L., and Schrijver, A., The ellipsoid method and its consequences in combinatorial optimization, *Combinatorica,* 1, 169–197, 1982.
[35] Grötschel, M., Lovász, L., and Schrijver, A., *Geometric Algorithms and Combinatorial Optimization,* Springer-Verlag, 1988.
[36] Greenberg, H.J., *A Computer-Assisted Analysis System for Mathematical Programming Models and Solutions: A User's Guide for ANALYZE,* Kluwer Academic Publishers, Boston, 1993.
[37] Golub, G.B. and van Loan, C.F., *Matrix Computations,* The Johns Hopkins University Press, Baltimore, 1983.
[38] Haĉijan, L.G., A polynomial algorithm in linear programming, *Soviet Math. Dokl.,* 20, 191–194, 1979.
[39] Haimovich, M., The simplex method is very good! On the expected number of pivot steps and related properties of random linear programs, *Unpublished Manuscript,* 1983.
[40] Hochbaum, D. and Naor, J„ Simple and fast algorithms for linear and integer programs with two variables per inequality, *Proceedings of the Symposium on Discrete Algorithms (SODA),* 1992. SIAM Press (also in the Proceedings of the Second Conference on Integer Programming and Combinatorial Optimization *IPCO,* Pittsburgh, Jun. 1992).
[41] Huynh, T., Lassez, C., and Lassez, J-L., Practical issues on the projection of polyhedral sets, *Annals of Mathematics and Artificial Intelligence,* 6, 295–316, 1992.
[42] IBM, *Optimization Subroutine Library—Guide and Reference (Release 2),* 3rd ed., 1991.
[43] Itai, A., Two-Commodity Flow, *Journal of the ACM,* 25, 596–611, 1978.
[44] Kalai, G. and Kleitman, D.J., A quasi-polynomial bound for the diameter of graphs of polyhedra, *Bulletin of the American Mathematical Society,* 26, 315–316, 1992.

[45] Kantorovich, L.V., Mathematical methods of organizing and planning production (in Russian), Publication House of the Leningrad State University, Leningrad, 1939; English translation in *Management Science,* 6, 366–422, 1959.
[46] Kamath, A. and Karmarkar, N.K., A continuous method for computing bounds in integer quadratic optimization problems, *Journal of Global Optimization,* 2, 229–241, 1992.
[47] Karmarkar, N.K., A new polynomial-time algorithm for linear programming, *Combinatorica,* 4, 373–395, 1984.
[48] Karp, R.M. and Papadimitriou, C.H., On linear characterizations of combinatorial optimization problems, *SIAM Journal on Computing,* 11, 620–632, 1982.
[49] Klee, V. and Minty, G.J., How good is the simplex algorithm? In *Inequalities III,* O. Shisha, Ed., Academic Press, 1972.
[50] Lenstra, J.K., Rinnooy Kan, A.H.G., and Schrijver, A., Eds., *History of Mathematical Programming: A Collection of Personal Reminiscences,* North Holland, 1991.
[51] Lovász, L., *An Algorithmic Theory of Numbers, Graphs and Convexity,* SIAM Press, 1986.
[52] Lovász, L. and Schrijver, A., Cones of matrices and setfunctions, *SIAM Journal on Optimization,* 1, 166–190, 1991.
[53] Lustig, I.J., Marsten, R.E., and Shanno, D.F., Interior point methods for linear programming: Computational state of the art, *ORSA J. on Computing,* 6(1), 1–14, 1994.
[54] Martin, R.K., Using separation algorithms to generate mixed integer model reformulations, *Operations Research Letters,* 10, 119–128, 1991.
[55] Matousek, J., Sharir, M., and Welzl, E., A subexponential bound for linear programming, in *Proceedings of the 8th Annual ACM Symposium on Computational Geometry,* ACM Press, 1–8, 1992.
[56] Megiddo, N., On finding primal- and dual-optimal bases, in *ORSA Journal on Computing,* 3, 63–65.
[57] Megiddo, N., Towards a genuinely polynomial algorithm for linear programming, *SIAM Journal on Computing,* 12, 347–353, 1983.
[58] Megiddo, N., Linear programming in linear time when the dimension is fixed, *JACM,* 31, 114–127, 1984.
[59] Mehrotra, S., On the implementation of a primal-dual interior point method, *SIAM Journal on Optimization,* 2(4), 575–601, 1992.
[60] Motwani, R. and Raghavan, P., *Randomized Algorithms,* Cambridge University Press, 1996.
[61] Murtagh, B.A., *Advanced Linear Programming: Computation and Practice,* McGraw-Hill, New York, 1981.
[62] Murty, K.G., *Linear Programming,* John Wiley & Sons, New York, 1983.
[63] Nelson, C.G., An $O(n^{\log n})$ algorithm for the two-variable-per-constraint linear programming satisfiablity problem, *Technical Report AIM-319,* Dept. of Computer Science, Stanford University, 1978.
[64] Orponen, P., Neural networks and complexity theory, in *Proceedings of the* 17^{th} *International Symposium on Mathematical Foundations of Computer Science,* I.M. Havel and V. Koubek, Eds., Lecture Notes in Computer Science 629, Springer-Verlag, 50–61, 1992.
[65] Padberg, N.W., *Linear Optimization and Extensions,* Springer-Verlag, 1995.
[66] Padberg, M.W. and Rao, M.R., The Russian method for linear inequalities, Part III, Bounded integer programming, Preprint, New York University, 1981.
[67] Papadimitriou, C.H. and Steiglitz, K., *Combinatorial Optimization: Algorithms and Complexity,* Prentice-Hall, Englewood Cliffs, NJ, 1982.
[68] Papadimitriou, C.H. and Yannakakis, M., Optimization, approximation, and complexity classes, In *Journal of Computer and Systems Sciences,* 43, 425–440, 1991.
[69] Saigal, R., *Linear Programming: A Modern Integrated Analysis,* Kluwer Press, 1995.

[70] Saltzman, M.J., Implementation of an interior point LP algorithm on a shared-memory vector multiprocessor, in *Computer Science and Operations Research,* O. Balci, R. Sharda and S.A. Zenios, Eds., Pergamon Press, 87–104, 1992.
[71] Schrijver, A., *Theory of Linear and Integer Programming,* John Wiley & Sons, 1986.
[72] Sen, S., Parallel multidimensional search using approximation algorithms: with applications to linear-programming and related problems, *Proceedings of SPAA,* 1996.
[73] Sharda, R., Linear programming solver software for personal computers: 1995 report, *OR/MS Today,* 22(5), 49–57, 1995.
[74] Sharir, M. and Welzl, E., A combinatorial bound for linear programming and related problems, in *Proceedings of the 9th Symposium on Theoretical Aspects of Computer Science,* LNCS 577, Springer-Verlag, 569–579, 1992.
[75] Shmoys, D.B., Computing near-optimal solutions to combinatorial optimization problems, in [17], cited above, 355–398, 1995.
[76] Shor, N.Z., Convergence rate of the gradient descent method with dilation of the space, *Cybernetics,* 6, 1970.
[77] Stone, R.E. and Tovey, C.A., The simplex and projective scaling as iterated reweighting least squares, *SIAM Review,* 33, 220–237, 1991.
[78] Tardos, E., A strongly polynomial algorithm to solve combinatorial linear programs, *Operations Research,* 34, 250–256, 1986.
[79] Todd, M.J., Exploiting special structure in Karmarkar's algorithm for linear programming, *Technical Report 707,* School of Operations Research and Industrial Engineering, Cornell University, Jul. 1986.
[80] Weyl, H., Elemetere Theorie der konvexen polyerer, *Comm. Math. Helv.,* 1, 3–18, 1935. (English translation in *Annals of Mathematics Studies,* 24, 1950).
[81] Wright, S.J., *Primal-Dual Interior-Point Methods,* SIAM Press, 1997.
[82] Yannakakis, M., Expressing combinatorial optimization problems by linear programs, In *Proceedings of ACM Symposium of Theory of Computing,* 223–228, 1988.
[83] Ziegler, M., *Convex Polytopes,* Springer-Verlag, 1995.

Further Information

Research publications in linear programming are dispersed over a large range of journals. The following is a partial list which emphasize the algorithmic aspects: *Mathematical Programming, Mathematics of Operations Research, Operations Research, INFORMS Journal on Computing, Operations Research Letters, SIAM Journal on Optimization, SIAM Journal on Computing, SIAM Journal on Discrete Mathematics, Algorithmica, Combinatorica.*

Linear programming professionals frequently use the following newsletters:

- *INFORMS Today* (earlier OR/MS Today) published by The Institute for Operations Research and Management Science.
- *INFORMS CSTS Newsletter* published by the INFORMS computer science technical section.
- *Optima* published by the Mathematical Programming Society.

The linear programming FAQ (frequently asked questions) facility is maintained at

http://www.mcs.anl.gov/home/otc/Guide/faq/

To have a linear program solved over the internet check the following locations:

http://www.mcs.anl.gov/home/otc/Server/
http://www.mcs.anl.gov/home/otc/Server/lp/

The universal input standard for linear programs is the MPS format [61].

32

Integer Programming[1]

Vijay Chandru
Indian Institute of Science

M.R. Rao
Indian Institute of Management
Bangalore

32.1 Abstract

Integer programming is an expressive framework for modeling and solving discrete optimization problems that arise in a variety of contexts in the engineering sciences. Integer programming representations work with implicit algebraic constraints (linear equations and inequalities on integer valued variables) to capture the feasible set of alternatives, and linear objective functions (to minimize or maximize over the feasible

[1] Readers unfamiliar with linear programming methodology are strongly encouraged to consult Chapter 31 on linear programming in this *Handbook*.

set) that specify the criterion for defining optimality. This algebraic approach permits certain natural extensions of the powerful methodologies of linear programming to be brought to bear on combinatorial optimization and on fundamental algorithmic questions in the geometry of numbers.

32.2 Introduction

In 1957 the Higgins lecturer of mathematics at Princeton, Ralph Gomory, announced that he would lecture on solving **linear programs** in integers. The immediate reaction he received was "But that's impossible!" This was his first indication that others had thought about the problem [57]. Gomory went on to work on the foundations of the subject of integer programming as a scientist at IBM from 1959 until 1970 (when he took over as Director for Research). From **cutting planes** and the polyhedral combinatorics of corner polyhedra to group knapsack **relaxations,** the approaches that Gomory developed remain striking to researchers even today.

There were other pioneers in integer programming who collectively played a similar role in developing techniques for linear programming in Boolean or 0-1 variables. These efforts were directed at combinatorial optimization problems such as routing, scheduling, layout, and network design. These are generic examples of combinatorial optimization problems that often arise in computer engineering and decision support.

Unfortunately, almost all interesting generic classes of integer programming problems are $\mathcal{NP}$-hard. The scale at which these problems arise in applications and the explosive exponential complexity of the search spaces preclude the use of simplistic enumeration and search techniques. Despite the worst-case intractability of integer programming, in practice we are able to solve many large problems and often enough with off-the-shelf software. Effective software for integer programming is usually problem specific and based on sophisticated algorithms that combine approximation methods with search schemes and that exploit mathematical (and not just syntactic) structure in the problem at hand.

An abstract formulation of combinatorial optimization is

$$(CO) \quad \min\{f(I) : I \in \mathcal{I}\}$$

where $\mathcal{I}$ is a collection of subsets of a finite ground set $E = \{e_1, e_2, \ldots, e_n\}$ and f is a criterion (objective) function that maps 2^E (the power set of E) to the reals. The most general form of an integer linear program is

$$(MILP) \quad \min_{\mathbf{x} \in \Re^n} \left\{ \mathbf{cx} : A\mathbf{x} \geq \mathbf{b},\ x_j \text{ integer } \forall\, j \in J \right\}$$

which seeks to minimize a linear function of the decision vector $\mathbf{x}$ subject to linear inequality constraints and the requirement that a subset of the decision variables are integer valued. This model captures many variants. If $J = \{1, 2, \ldots, n\}$, we say that the integer program is *pure,* and *mixed* otherwise. Linear equations and bounds on the variables can be easily accommodated in the inequality constraints. Notice that by adding in inequalities of the form $0 \leq x_j \leq 1$ for a $j \in J$ we have forced x_j to take value 0 or 1. It is such Boolean variables that help capture combinatorial optimization problems as special cases of (MILP).

The next section contains preliminaries on linear inequalities, polyhedra, linear programming, and an overview of the complexity of integer programming. These are the tools we will need to analyze and solve integer programs. Section 32.4 is the testimony on how integer programs model combinatorial optimization problems. In addition to working a number of examples of such integer programming formulations, we shall also review formal representation theories of (Boolean) **mixed integer linear programs.**

With any mixed integer program we associate a `linear programming relaxation` obtained by simply ignoring the integrality restrictions on the variables. The point being, of course, that we have polynomial-time (and practical) algorithms for solving linear programs (see Chapter 31 of this *Handbook*). Thus the linear programming relaxation of (MILP) is given by

$$(LP) \quad \min_{\mathbf{x} \in \Re^n} \{\mathbf{cx} : A\mathbf{x} \geq \mathbf{b}\}$$

The thesis underlying the integer linear programming approaches is that this linear programming relaxation retains enough of the structure of the combinatorial optimization problem to be a useful weak representation. In Section 32.5 we shall take a closer look at this thesis in that we shall encounter special structures for which this relaxation is "tight." For general integer programs, there are several alternate schemes for generating linear programming relaxations with varying qualities of approximation. A general technique for improving the quality of the linear programming relaxation is through the generation of valid inequalities or cutting planes.

The computational art of integer programming rests on useful interplays between search methodologies and algebraic relaxations. The paradigms of branch & bound and branch & cut are the two enormously effective partial enumeration schemes that have evolved at this interface. These will be discussed in Section 32.6. It may be noted that all general purpose integer programming software available today uses one or both of these paradigms.

A general principle, is that we often need to disaggregate integer formulations to obtain higher quality linear programming relaxations. To solve such huge linear programs we need specialized techniques of large-scale linear programming. These aspects are described in Chapter 31 in this *Handbook.* The reader should note that the focus in this chapter is on solving hard combinatorial optimization problems. We catalog several special structures in integer programs that lead to tight linear programming relaxations (Section 32.7) and hence to polynomial-time algorithms. These include structures such as network flows, matching, and matroid optimization problems. Many hard problems actually have pieces of these nice structures embedded in them. Successful implementations of combinatorial optimization have always used insights from special structures to devise strategies for hard problems.

The inherent complexity of integer linear programming has led to a long-standing research program in approximation methods for these problems. Linear programming relaxation and Lagrangean relaxation (Section 32.7) are two general approximation schemes that have been the real workhorses of computational practice. Primal-dual strategies and semidefinite relaxations (Section 32.8) are two recent entrants that appear to be very promising.

Pure integer programming with variables that take arbitrary integer values is a natural extension of diophantine equations in number theory. Such problems arise in the context of cryptography, dependence analysis in programs, the geometry of numbers and Presburgher arithmetic. Section 32.9 covers this aspect of integer programming.

We conclude the chapter with brief comments on future prospects in combinatorial optimization from the algebraic modeling perspective.

32.3 Preliminaries

Polyhedral Preliminaries

Polyhedral combinatorics is the study of embeddings of combinatorial structures in Euclidean space and their algebraic representations. We will make extensive use of some standard terminology from polyhedral theory. Definitions of terms not given in the brief review below can be found in [95, 124].

A (convex) **polyhedron** in $\Re^n$ can be algebraically defined in two ways. The first and more straightforward definition is the *implicit* representation of a polyhedron in $\Re^n$ as the solution set to a finite system of linear inequalities in n variables. A single linear inequality $\mathbf{ax} \leq a_0\,;\ a \neq 0$ defines a *half-space* of $\Re^n$. Therefore geometrically a polyhedron is the intersection set of a finite number of half-spaces.

A *polytope* is a bounded polyhedron. Every polytope is the convex closure of a finite set of points. Given a set of points whose convex combinations generate a polytope we have an explicit or *parametric* algebraic representation of it. A *polyhedral cone* is the solution set of a system of homogeneous linear inequalities. Every (polyhedral) cone is the conical or positive closure of a finite set of vectors. These generators of the cone provide a parametric representation of the cone. And finally a polyhedron can be alternately defined as the Minkowski sum of a polytope and a cone. Moving from one representation of any of these polyhedral

objects to another defines the essence of the computational burden of polyhedral combinatorics. This is particularly true if we are interested in "minimal" representations.

A set of points $\mathbf{x}^1, \ldots, \mathbf{x}^m$ is *affinely independent* if the unique solution of $\Sigma_{i=1}^m \lambda_i \mathbf{x}^i = \mathbf{0}$, $\Sigma_{i=1}^m \lambda_i = 0$ is $\lambda_i = 0$ for $i = 1, \ldots, m$. Note that the maximum number of affinely independent points in $\Re^n$ is $n+1$. A polyhedron P is of *dimension* k, $\dim P = k$, if the maximum number of affinely independent points in P is $k+1$. A polyhedron $P \subseteq \Re^n$ of dimension n is called *full-dimensional*.

An inequality $\mathbf{ax} \leq a_0$ is called *valid* for a polyhedron P if it is satisfied by all $\mathbf{x}$ in P. It is called *supporting* if in addition there is an $\tilde{\mathbf{x}}$ in P that satisfies $\mathbf{a}\tilde{\mathbf{x}} = a_0$. A *face* of the polyhedron is the set of all $\mathbf{x}$ in P that also satisfy a valid inequality as an equality. In general, many valid inequalities might represent the same face. Faces other than P itself are called *proper*. A *facet* of P is a maximal nonempty and proper face. A facet is then a face of P with a dimension of $\dim P - 1$. A face of dimension zero, i.e., a point $\mathbf{v}$ in P that is a face by itself, is called an **extreme point** of P. The extreme points are the elements of P that cannot be expressed as a strict convex combination of two distinct points in P. For a full-dimensional polyhedron, the valid inequality representing a facet is unique up to multiplication by a positive scalar, and facet-inducing inequalities give a minimal implicit representation of the polyhedron. Extreme points, on the other hand, give rise to minimal parametric representations of polytopes.

The two fundamental problems of linear programming (which are polynomially equivalent) are

- **Solvability.** This is the problem of checking if a system of linear constraints on real (rational) variables is solvable or not. Geometrically, we have to check if a polyhedron, defined by such constraints, is nonempty.
- **Optimization.** This is the problem (LP) of optimizing a linear objective function over a polyhedron described by a system of linear constraints.

Building on polarity in cones and polyhedra, duality in linear programming is a fundamental concept which is related to both the complexity of linear programming and to the design of algorithms for solvability and optimization. Here we will state the main duality results for optimization. If we take the *primal* linear program to be

$$(P) \quad \min_{\mathbf{x} \in \Re^n} \{\mathbf{cx} : A\mathbf{x} \geq \mathbf{b}\}$$

there is an associated *dual* linear program

$$(D) \quad \max_{\mathbf{y} \in \Re^m} \left\{\mathbf{b}^T\mathbf{y} : A^T y = \mathbf{c}^T,\ \mathbf{y} \geq \mathbf{0}\right\}$$

and the two problems satisfy the following:

1. For any $\hat{\mathbf{x}}$ and $\hat{\mathbf{y}}$ feasible in (P) and (D) (i.e., they satisfy the respective constraints), we have $\mathbf{c}\hat{\mathbf{x}} \geq \mathbf{b}^T\hat{\mathbf{y}}$ (**weak duality**). Consequently, (P) has a finite optimal solution if and only if (D) does.
2. $\mathbf{x}^*$ and $\mathbf{y}^*$ are a pair of optimal solutions for (P) and (D), respectively, if and only if $\mathbf{x}^*$ and $\mathbf{y}^*$ are feasible in (P) and (D) (i.e., they satisfy the respective constraints) **and** $\mathbf{cx}^* = \mathbf{b}^T\mathbf{y}^*$ (**strong duality**).
3. $\mathbf{x}^*$ and $\mathbf{y}^*$ are a pair of optimal solutions for (P) and (D), respectively, if and only if $\mathbf{x}^*$ and $\mathbf{y}^*$ are feasible in (P) and (D) (i.e., they satisfy the respective constraints) **and** $(A\mathbf{x}^* - \mathbf{b})^T\mathbf{y}^* = 0$ (**complementary slackness**).

The strong duality condition gives us a good stopping criterion for optimization algorithms. The complementary slackness condition, on the other hand gives us a constructive tool for moving from dual to primal solutions and vice-versa. The weak duality condition gives us a technique for obtaining lower bounds for minimization problems and upper bounds for maximization problems.

Note that the properties above have been stated for linear programs in a particular form. The reader should be able to check, that if for example the primal is of the form

$$(P') \quad \min_{\mathbf{x}\in\Re^n} \{\mathbf{cx} : A\mathbf{x} = \mathbf{b},\ \mathbf{x} \geq 0\}$$

then the corresponding dual will have the form

$$(D') \quad \max_{\mathbf{y}\in\Re^m} \left\{\mathbf{b}^T\mathbf{y} : A^T\mathbf{y} \leq \mathbf{c}^T\right\}$$

The tricks needed for seeing this is that any equation can be written as two inequalities, an unrestricted variable can be substituted by the difference of two nonnegatively constrained variables and an inequality can be treated as an equality by adding a nonnegatively constrained variable to the lesser side. Using these tricks, the reader could also check that duality in linear programming is involutory (i.e., the dual of the dual is the primal).

Linear Diophantine Systems

Let us first examine the simple case of solving a single equation with integer (rational) coefficients and integer variables.

$$a_1x_1 + a_2x_2 + \cdots + a_nx_n \ = \ b \tag{32.1}$$

A classical technique is to use the Euclidean algorithm to eliminate variables. Consider the first iteration in which we compute a_{12}, and the integers δ_1 and δ_2 where

$$a_{12} \ = \ gcd\,(a_1, a_2) \ = \ \delta_1 a_1 \ + \ \delta_2 a_2$$

Now we have a reduced equation that is equivalent to Eq. (32.1).

$$a_{12}x_{12} + a_3x_3 + \cdots + a_nx_n \ = \ b \tag{32.2}$$

It is apparent that integer solutions to Eq. (32.2) are linear projections of integer solutions to Eq. (32.1). However, it is not a simple elimination of a variable as happens in the case of equations over reals. It is a projection to a space whose dimension is one less than the dimension we began with. The solution scheme reduces the equation to a univariate equation and then inverts the projection maps. All of this can be accomplished in polynomial time since the Euclidean algorithm is "good."

Solving a system of linear diophantine equations $A\mathbf{x} = \mathbf{b},\ \mathbf{x} \in Z^n$ now only requires a matrix version of the simple scheme described above. An integer matrix, of full row rank, is said to be in *Hermite normal form* if it has the appearance $[L|0]$ where L is nonsingular and lower triangular with nonnegative entries satisfying the condition that the largest entry of each row is on the main diagonal. A classical result (cf. [113]) is that any integer matrix A with full row rank has a *unique* Hermite normal form HNF$(A) = AK = [L|0]$ where K is a square unimodular matrix (an integer matrix with determinant ± 1).

The matrix K encodes the composite effect of the elementary column operations on the matrix A needed to bring it to normal form. The elementary operations are largely defined by repeated invocation of the Euclidean algorithm in addition to column swaps and subtractions. Polynomial-time computability of Hermite normal forms of integer matrices was first proved by Kannan and Bachem [75] using delicate and complicated analysis of the problems of intermediate swell. Subsequently, a much easier argument based on modulo arithmetic was given by Domich, Kannan and Trotter [36]. As consequences, we have that

- *Linear Diophantine systems can be solved in polynomial time. Assuming A has been preprocessed to have full row rank, to solve $A\mathbf{x} = b, \mathbf{x} \in Z^n$ we first obtain HNF(A) = AK = [L|0]. The*

input system has a solution if and only if $L^{-1}\mathbf{b}$ *is integral and if so, a solution is given by* $\mathbf{x} = K\begin{pmatrix} L^{-1}\mathbf{b} \\ \mathbf{0} \end{pmatrix}$.

- KRONECKER: (cf. [113]) $A\mathbf{x} = \mathbf{b}, \mathbf{x} \in Z^n$ *has no solution if and only if there exists a* $\mathbf{y} \in \Re^m$ *such that* $\mathbf{y}^t A$ *is integral and* $\mathbf{y}^t\mathbf{b}$ *is not. A certificate of unsolvability is always available from the construction described above.*
- *The solutions to a linear Diophantine system are finitely generated. In fact, a set of generators can be found in polynomial time.* $\{\mathbf{x} \in Z^n | A\mathbf{x} = \mathbf{b}\} = \{\mathbf{x}^0 + \Sigma_{i=1}^m \lambda_j \mathbf{x}^j | \lambda_j \in Z\}$, $\mathbf{x}^0 = K\begin{pmatrix} L^{-1}\mathbf{b} \\ \mathbf{0} \end{pmatrix}$ *and* $[\mathbf{x}^1, \mathbf{x}^2, \cdots, \mathbf{x}^m] = K(0, I)^t$.

In summary, linear Diophantine systems are a lot like linear systems over the reals (rationals). The basic theory and complexity results for variable elimination in both constraint domains are similar. This comfortable situation changes when we move on to linear *inequalities* over the integers or equivalently to nonnegative solutions to linear Diophantine systems.

Computational Complexity of Integer Programming

Any algorithm for integer programming is a universal polynomial-time decision procedure on a nondeterministic Turing machine. This statement is credible not only because the solvable systems of linear inequalities (with rational coefficients) over integer-valued variables describe an $\mathcal{NP}$-complete language but also because integer programs are expressive enough to capture most decision problems in $\mathcal{NP}$ via straightforward reductions. This expressiveness derives from our ability to embed sentential inference and combinatorial structures with equal ease in integer programs. We will see ample demonstration of this expressiveness in the next section. This relationship of integer programming with $\mathcal{NP}$ is akin to the relationship of linear programming with $\mathcal{P}$.

Complexity of Linear Inequalities

From our earlier discussion of polyhedra, we have the following algebraic characterization of extreme points of polyhedra.

THEOREM 32.1 *Given a polyhedron P, defined by* $\{\mathbf{x} : A\mathbf{x} \leq \mathbf{b}\}$, *where A is* $m \times n$, $\mathbf{x}^i$ *is an extreme point of P if and only if it is a face of P satisfying* $A^i\mathbf{x}^i = \mathbf{b}^i$ *where* $((A^i), (\mathbf{b}^i))$ *is a submatrix of* $(A, \mathbf{b})$ *and the rank of* A^i *equals n.*

COROLLARY 32.1 The decision problem of verifying the membership of an input string $(A, \mathbf{b})$ in the language $\mathcal{L}_I = \{(A, \mathbf{b}) : \exists\, \mathbf{x}$ *such that* $A\mathbf{x} \leq \mathbf{b}\}$ belongs to $\mathcal{NP}$.

PROOF It follows from the theorem that every extreme point of the polyhedron $Q = \{\mathbf{x} : A\mathbf{x} \leq \mathbf{b}\}$ is the solution of an $(n \times n)$ linear system whose coefficients come from $(A, \mathbf{b})$. Therefore we can guess a polynomial length string representing an extreme point and check its membership in Q in polynomial time.

A consequence of Farkas Lemma [45] (duality in linear programming) is that the decision problem of testing membership of input $(A, \mathbf{b})$ in the language

$$\mathcal{L}_I = \{(A, \mathbf{b}) : \exists\, \mathbf{x} \text{ such that } A\mathbf{x} \leq \mathbf{b}\}$$

is in $\mathcal{NP} \bigcap co\mathcal{NP}$. That $\mathcal{L}_I \in \mathcal{P}$, follows from algorithms for linear programming [77, 81]. This is as

far down the polynomial hierarchy that we can go, since $\mathcal{L}_I$ is known to be $\mathcal{P}$-complete (that is, complete for $\mathcal{P}$ with respect to log-space reductions).

Complexity of Linear Inequalities in Integer Variables

The complexity of integer programming is polynomially equivalent to the complexity of the language

$$\mathcal{L}_{IP} = \{(A, \mathbf{b}) : \exists \mathbf{x} \in Z^n \text{ such that } A\mathbf{x} \leq \mathbf{b}\}$$

We are assuming that the input coefficients in $A, \mathbf{b}$ are integers (rationals). It is very easy to encode Boolean satisfiability as a special case of $\mathcal{L}_{IP}$ with appropriate choice of $A, \mathbf{b}$ as we shall see in the next section. Hence $\mathcal{L}_{IP}$ is $\mathcal{NP}$-hard.

It remains to argue that the decision problem of verifying the membership of an input string $(A, \mathbf{b})$ in the language $\mathcal{L}_{IP}$ belongs to $\mathcal{NP}$. We will have to work a little harder since integer programs may have solutions involving numbers that are large in magnitude. Unlike the case of linear programming there is no extreme point characterization for integer programs. However, since linear diophantine systems are well behaved, we are able to salvage the following technical result that plays a similar role. Let α denote the largest integer in the integer matrices $A, \mathbf{b}$ and let $q = \max\{m, n\}$ where A is $m \times n$.

Finite Precision Lemma [103] If B is an $r \times n$ submatrix of A with rank $B < n$ then $\exists$ a nonzero integral vector $\mathbf{z} = (z_1, z_2, \cdots z_n)$ in the null space of B such that $|z_j| \leq (\alpha q)^{q+1}\ \forall j$.

Repeated use of this lemma shows that if $\{\mathbf{x} \in Z^n : A\mathbf{x} \leq \mathbf{b}\}$ is solvable then there is a polynomial size certificate of solvability and hence that $\mathcal{L}_{IP}$ belongs to $\mathcal{NP}$.

As with any $\mathcal{NP}$-hard problem, researchers have looked for special cases that are polynomial-time solvable. Table 32.1 is a summary of the important complexity classification results in integer programming that have been obtained to date.

32.4 Integer Programming Representations

We will first discuss several examples of combinatorial optimization problems and their formulation as integer programs. Then we will review a general representation theory for integer programs that gives a formal measure of the expressiveness of this algebraic approach. We conclude this section with a representation theorem due to Benders [10] that has been very useful in solving certain large-scale combinatorial optimization problems in practice.

Formulations

Formulating decision problems as integer or mixed integer programs is often considered an art form. However, there are a few basic principles which can be used by a novice to get started. As in all art forms though, principles can be violated to creative effect. We list below a number of example formulations, the first few of which may be viewed as principles for translating logical conditions into models.

1. Discrete Choice:

CONDITION	MODEL
$X \in \{s_1, s_2, \ldots, s_p\}$	$X = \sum_{j=1}^{p} s_j\ \delta_j$ $\sum_{j=1}^{p} \delta_j = 1,\ \delta_j = 0 \text{ or } 1\ \forall j$

TABLE 32.1 Summary of Complexity Results

	Input Data	Generic Problem	Complexity
	`Solvability`	`Does ∃ an x satisfying:`	
1.	$n, m, A, \mathbf{b}$	$A\mathbf{x} = \mathbf{b}; \mathbf{x} \geq \mathbf{0}$, integer	NP-complete [16, 54, 79]
2.	$n, m, A, \mathbf{b}$	$A\mathbf{x} \leq \mathbf{b}; \mathbf{x} \in \{0, 1\}^n$	NP-complete [79, 110]
3.	$n, m, A, \mathbf{b}, d(> 0)$	$A\mathbf{x} \equiv \mathbf{b} \pmod{d}; \mathbf{x} \geq \mathbf{0}$, integer	P [6, 50]
4.	$n, m, A, \mathbf{b}$	$A\mathbf{x} = \mathbf{b}; \mathbf{x}$ integer	P [6, 50]
5.	$n, m, A, \mathbf{b}$	$A\mathbf{x} = \mathbf{b}; \mathbf{x} \geq \mathbf{0}$	P [77, 81]
6.	$n, (m = 1), (A \geq 0), (\mathbf{b} \geq \mathbf{0})$	$A\mathbf{x} = \mathbf{b}; \mathbf{x} \geq \mathbf{0}$, integer	NP-complete [110]
7.	$n, (m = 2), (A \geq 0), (\mathbf{b} \geq \mathbf{0})$	$a^1\mathbf{x} \geq b_1; a^2\mathbf{x} \leq b_2; \mathbf{x} \geq \mathbf{0}$, integer	NP-complete [76]
8.	$(n = k), m, A, \mathbf{b}$	$A\mathbf{x} = \mathbf{b}; \mathbf{x} \geq \mathbf{0}$, integer	P [87]
9.	$n, (m = k), A, \mathbf{b}$	$A\mathbf{x} \leq \mathbf{b}; \mathbf{x}$ integer	P [87]
	`Optimization`	`Find an x that maximizes cx subject to:`	
10.	$n, m, A, \mathbf{b}, \mathbf{c}$	$A\mathbf{x} = \mathbf{b}; \mathbf{x} \geq \mathbf{0}$, integer	NP-hard [54, 79]
11.	$n, m, A, \mathbf{b}, \mathbf{c}$	$A\mathbf{x} \leq \mathbf{b}; \mathbf{x} \in \{0, 1\}^n$	NP-hard [79, 110]
12.	$n, (m = 1), (A \geq 0), (\mathbf{b} \geq \mathbf{0}), \mathbf{c}$	$A\mathbf{x} = \mathbf{b}; \mathbf{x} \geq \mathbf{0}$, integer	NP-hard [110]
13.	$n, (m = 1), (A \geq 0), (\mathbf{b} \geq \mathbf{0}), \mathbf{c}$	$A\mathbf{x} \leq \mathbf{b}; \mathbf{x} \in \{0, 1\}^n$	NP-hard [110]
14.	$n, m, A, \mathbf{b}, \mathbf{c}, (d_i \geq 2 \forall i)$	$A\mathbf{x} \equiv \mathbf{b} \pmod{\mathbf{d}}; \mathbf{x} \geq \mathbf{0}$, integer	NP-hard [20]
15.	$(n = k), m, A, \mathbf{b}, \mathbf{c}$	$A\mathbf{x} = \mathbf{b}; \mathbf{x} \geq \mathbf{0}$, integer	P [87]
16.	$n, (m = k), A, \mathbf{b}, \mathbf{c}$	$A\mathbf{x} \leq \mathbf{b}; \mathbf{x}$ integer	P [87]
17.	$n, m, A, \mathbf{b}, \mathbf{c}$	$A\mathbf{x} = \mathbf{b}; \mathbf{x} \geq \mathbf{0}$	P [77, 81]
18.	$n, m,$ (A is graphic)[a], $\mathbf{b}_1, \mathbf{b}_2, \mathbf{d}_1, \mathbf{d}_2$	$\mathbf{d}_1 \leq \mathbf{x} \leq \mathbf{d}_2, \mathbf{b}_1 \leq A\mathbf{x} \leq \mathbf{b}_2, \mathbf{x} \in Z^n$	P [44, 101]

[a] A is a graphic matrix if it has entries from $\{0, \pm 1, \pm 2\}$ such that the sum of absolute values of the entries in any column is at most 2.

2. DICHOTOMY:

CONDITION	MODEL	
$g(\mathbf{x}) \geq 0$	$g(\mathbf{x}) \geq \delta\, \underline{g}$	$\underline{g}$ and $\underline{h}$ are
or		finite lower
$h(\mathbf{x}) \geq 0$	$h(\mathbf{x}) \geq (1 - \delta)\, \underline{h}$	bounds on g, h,
or both	$\delta = 0$ or 1	respectively.

3. K-FOLD ALTERNATIVES:

condition	model
at least k of	$g_i(\mathbf{x}) \geq \delta_i\, \underline{g}_i \quad i = 1, \ldots, m$
$g_i(\mathbf{x}) \geq 0, i = 1, \ldots, m$	$\sum_{i=1}^{m} \delta_i \leq m - k$
must hold	$\delta_i = 0$ or 1

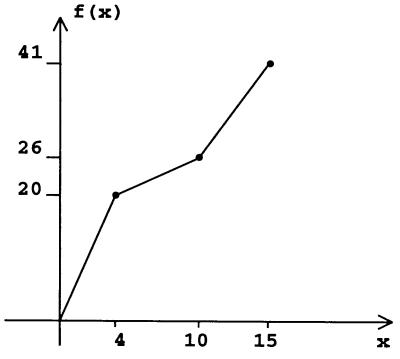
4. CONDITIONAL CONSTRAINTS:

condition	model	
$(f(\mathbf{x}) > 0 \Rightarrow g(\mathbf{x}) \geq 0)$	$g(\mathbf{x}) \geq \delta \underline{g}$	$\bar{f}/\underline{g}$ is an
$\Updownarrow$	$f(\mathbf{x}) \leq (1 - \delta)\, \bar{f}$	upper/lower bound
$(f(\mathbf{x}) \leq 0$ or $g(\mathbf{x}) \geq 0$ or *both* $)$	$\delta = 0$ or 1	on f/g

5. FIXED CHARGE MODELS:

condition	model	
$f(x) = 0$ if $x = 0$	$f(x) = Ky + cx$	
$f(x) = K + cx$ if $x > 0$	$x \le Uy$	U an upper bound
	$x \ge 0$	on x
	$y = 0$ or 1	

6. PIECEWISE LINEAR MODELS:



condition	model	
	$f(x) =$	$5\delta_1 + \delta_2 + 3\delta_3$
$\delta_2 > 0 \Rightarrow \delta_1 = 4$		$4W_1 \le \delta_1 \le 4$
$\delta_3 > 0 \Rightarrow \delta_2 = 6$		$6W_2 \le \delta_2 \le 6W_1$
		$0 \le \delta_3 \le 5W_2$
		$W_1, W_2 = 0$ or 1

FIGURE 32.1 A polyline.

7. CAPACITATED PLANT LOCATION MODEL:

$i =$	$\{1, 2, \ldots, m\}$	possible locations for plants
$j =$	$\{1, 2, \ldots, n\}$	demand sites
$k_i =$	capacity of plant i, if opened	
$f_i =$	fixed cost of opening plant i	
$c_{ij} =$	per unit production cost at i plus transportion cost from i to j	
$d_j =$	demand at location j	

Choose plant locations so as to minimize total cost and meet all demands.
Formulation:

$$\begin{aligned} \min \textstyle\sum_i \sum_j c_{ij}\, x_{ij} \quad &+ \quad \sum_i f_i\, y_i \\ s.t. \ \textstyle\sum_i x_{ij} \quad &\ge \quad d_j \ \forall j \\ \textstyle\sum_j x_{ij} \quad &\le \quad k_i\, y_i \ \forall i \\ x_{ij} \quad &\ge \quad 0 \quad \forall\, i,\ j \\ y_i \quad &= \quad 0 \text{ or } 1 \ \forall i \end{aligned}$$

If the demand d_j is less than the capacity k_i for some ij combination, it is useful to add the constraint $x_{ij} \le d_j y_i$ to improve the quality of the linear programming relaxation.

8. TRAVELING SALESMAN PROBLEM (ALTERNATE FORMULATIONS): A recurring theme in integer programming is that the same decision problem can be formulated in several different ways. Principles for sorting out the better ones have been the subject of some discourse [121]. We now illustrate this with the well-known traveling salesman problem. Given a complete directed graph D(N,A) with distance c_{ij} of arc (i, j), we are to find the minimum length tour beginning at node 1 and visiting each node of D(N,A) exactly once before returning to the start node 1.

Formulation 1:

$$\begin{aligned} \min \sum_{i=1}^{n} \sum_{j=1}^{n} c_{ij}\, x_{ij} & & & \\ s.t.\ \sum_{j=1}^{n} x_{ij} &= 1 \ \forall i \\ \sum_{i=1}^{n} x_{ij} &= 1 \ \forall j \\ \sum_{i\in\phi} \sum_{j\notin\phi} x_{ij} &\geq 1 \ \forall\, \phi \subset N \\ x_{ij} &= 0 \text{ or } 1 \quad \forall\, i\ j \end{aligned}$$

Formulation 2:

$$\begin{aligned} \min \sum_{i=1}^{n} \sum_{j=1}^{n} c_{ij}\, x_{ij} & & & \\ s.t.\ \sum_{j=1}^{n} x_{ij} &= 1 \ \forall i \\ \sum_{i=1}^{n} x_{ij} &= 1 \ \forall j \\ \sum_{j=1}^{n} y_{ji} - \sum_{j=2}^{n} y_{ij} &= 1 \ \forall i \neq 1 \\ y_{ij} &\leq (n-1)\, x_{ij} \quad i = 1, 2, \ldots, n \\ & \qquad\qquad j = 2, \ldots, n \\ x_{ij} &= 0 \text{ or } 1 \quad \forall ij \\ y_{ij} &\geq 0 \quad \forall ij \end{aligned}$$

9. NONLINEAR FORMULATIONS: If we allow arbitrary nonlinear objective and constraint functions the general integer programming problem is closely related to Hilbert's tenth problem and is undecidable [69]. However, when the integer variables are restricted to $0-1$, the problem is of the form

$$(NIP) \quad \min \left\{ f(\mathbf{x}) \mid g_i(\mathbf{x}) \leq 0,\ i = 1, \ldots m,\ \mathbf{x} \in \{0,1\}^n \right\}$$

and we can capture a rich variety of decision problems (modular design, cluster analysis, capital budgeting under uncertainty, and production planning in flexible manufacturing systems, to name a few). Restricting the constraints to linear assignment constraints while allowing quadratic cost functions yields the quadratic assignment problem (QAP)

$$\begin{aligned} \min \sum_{i\neq j} \sum_{k\neq l} c_{ijkl}\, x_{ik}\, x_{jl} & & & \\ s.t.\ \sum_{i} x_{ik} &= 1 \ \forall k \\ \sum_{k} x_{ik} &= 1 \ \forall i \\ x_{ik} &= 0 \text{ or } 1 \ \forall ik \end{aligned}$$

which includes the travelling salesman problem and plant location problems as special cases. Karmarkar [78] has advocated solving integer programs, by first formulating them as minimizing an indefinite quadratic function over a polyhedral region, and then solving the continuous model using interior point methods.
The other side of the coin is that these problems are extremely hard to solve, and the most successful strategies to date for these problems are via linearization techniques and semidefinite relaxations.

10. COVERING AND PACKING PROBLEMS: A wide variety of location and scheduling problems can be formulated as set covering or set packing or set partitioning problems. The three different types of covering and packing problems can be succinctly stated as follows: Given

    (a) A finite set of elements $\mathcal{M} = \{1, 2, \ldots, m\}$, and

    (b) A family F of subsets of $\mathcal{M}$ with each member $F_j,\ j = 1, 2, \ldots, n$ having a profit (or cost) c_j associated with it,

    find a collection, S, of the members of F that maximizes the profit (or minimizes the cost) while ensuring that every element of $\mathcal{M}$ is in

(P1): at most one member of S (set packing problem)

(P2): at least one member of S (set covering problem)

(P3): exactly one member of S (set partitioning problem)

The three problems (P1), (P2), and (P3) can be formulated as integer linear programs as follows:
Let A denote the $m \times n$ matrix where

$$A_{ij} = \begin{cases} 1 & \text{if element } i \in F_j \\ 0 & \text{otherwise} \end{cases}$$

The decision variables are $x_j,\ \ j = 1, 2, \ldots, n$ where

$$x_j = \begin{cases} 1 & \text{if } F_j \text{ is chosen} \\ 0 & \text{otherwise} \end{cases}$$

The set packing problem is

$$\begin{array}{lll} \text{(P1)} & \text{Max} & c\mathbf{x} \\ & \text{Subject to} & A\mathbf{x} \leq \mathbf{e}_m \\ & & x_j = 0 \text{ or } 1, \quad j = 1, 2, \ldots, n \end{array}$$

where $\mathbf{e}_m$ is an m-dimensional column vector of 1s.
The set covering problem (P2) is (P1) with less than or equal to constraints replaced by greater than or equal to constraints and the objective is to minimize rather than maximize. The set partitioning problem (P3) is (P1) with the constraints written as equalities. The set partitioning problem can be converted to a set packing problem or a set covering problem (see [98]) using standard transformations. If the right-hand side vector e_m is replaced by a nonnegative integer vector b, (P1) is referred to as the generalized set packing problem.
The airline crew scheduling problem is a classic example of the set partitioning or the set covering problem. Each element of $\mathcal{M}$ corresponds to a flight segment. Each subset F_j corresponds to an acceptable set of flight segments of a crew. The problem is to cover, at minimum cost, each flight segment exactly once. This is a set partitioning problem. If *dead heading* of crew is permitted, we have the set covering problem.

11. PACKING AND COVERING PROBLEMS IN A GRAPH: Suppose A is the node-edge incidence matrix of a graph. Now, (P1) is a weighted matching problem. If in addition, the right-hand side vector e_m is replaced by a nonnegative integer vector b, (P1) is referred to as a weighted b-matching problem. In this case, each variable x_j that is restricted to be an integer may have a positive upper bound of u_j. Problem (P2) is now referred to as the weighted edge covering problem. Note that by substituting for $x_j = 1 - y_j$, where $y_j = 0 \text{ or } 1$, the weighted edge covering problem is transformed to a weighted b-matching problem in which the variables are restricted to be 0 or 1.
Suppose A is the edge-node incidence matrix of a graph. Now, (P1) is referred to as the weighted vertex packing problem and (P2) is referred to as the weighted vertex covering problem. It is easy to see that the weighted vertex packing problem and the weighted vertex covering problem are equivalent in the sense that the complement of an optimal solution to one problem defines an optimal solution to the other. The *set packing* problem can be transformed to a weighted vertex packing problem in a graph G as follows:

G contains a node for each x_j and an edge between nodes j and k exists if and only if the columns $A_{.j}$ and $A_{.k}$ are not orthogonal. G is called the *intersection graph* of A. Given G, the complement graph $\bar{G}$ has the same node set as G, and there is an edge between nodes j and k in $\bar{G}$ if and only if there is no such corresponding edge in G. A clique in a graph is a subset, k, of nodes of G such that the subgraph induced by k is complete. Clearly, the weighted vertex packing problem in G is equivalent to finding a maximum weighted clique in $\bar{G}$.

12. SATISFIABILITY AND INFERENCE PROBLEMS: In propositional logic, a truth assignment is an assignment of "true" or "false" to each atomic proposition $x_1, x_2, \ldots, x_n$. A literal is an atomic proposition x_j or its negation $\neg x_j$. For propositions in conjunctive normal form, a clause is a disjunction of literals and the proposition is a conjunction of clauses. A clause is obviously satisfied by a given truth assignment if at least one of its literals is true. The satisfiability problem consists of determining whether there exists a truth assignment to atomic propositions such that a set S of clauses is satisfied.
Let T_i denote the set of atomic propositions such that if any one of them is assigned "true," the clause $i \in S$ is satisfied. Similarly let F_i denote the set of atomic propositions such that if any one of them is assigned "false," the clause $i \in S$ is satisfied.
The decision variables are

$$x_j = \begin{cases} 1 & \text{if atomic proposition } j \text{ is assigned true} \\ 0 & \text{if atomic proposition } j \text{ is assigned false} \end{cases}$$

The satisfiability problem is to find a feasible solution to

$$(P4) \quad \sum_{j \in T_i} x_j - \sum_{j \in F_i} x_j \geq 1 - |\, F_i \,| \;\; i \in S$$
$$x_j = 0 \text{ or } 1 \;\text{ for } j = 1, 2, \ldots, n$$

By substituting $x_j = 1 - y_j$, where $y_j = 0$ or 1, for $j \in F_i$, (P4) is equivalent to the set covering problem

$$\begin{aligned} (P5) \qquad \text{Min} \quad & \sum_{j=1}^{n} (x_j + y_j) \\ \text{subject to} \quad & \sum_{j \in T_i} x_j + \sum_{j \in F_i} y_j \geq 1 \;\; i \in S \\ & x_j + y_j \geq 1 \;\; j = 1, 2, \ldots, n \\ & x_j, y_j = 0 \text{ or } 1 \;\; j = 1, 2, \ldots, n \end{aligned}$$

Clearly (P4) is feasible if and only if (P5) has an optimal objective function value equal to n.
Given a set S of clauses and an additional clause $k \notin S$, the logical inference problem is to find out whether every truth assignment that satisfies all the clauses in S also satisfies the clause k. The logical inference problem is

$$
\begin{aligned}
(P6) \qquad \text{Min} \quad & \sum_{j \in T_k} x_j - \sum_{j \in F_k} x_j \\
\text{subject to} \quad & \sum_{j \in T_i} x_j - \sum_{j \in F_i} x_j \geq 1 - \mid F_i \mid \; i \in S \\
& x_j = 0 \text{ or } 1 \quad j = 1, 2, \ldots, n
\end{aligned}
$$

The clause k is implied by the set of clauses S, if and only if (P6) has an optimal objective function value greater than $-\mid F_k \mid$. It is also straightforward to express the MAX-SAT problem (i.e., find a truth assignment that maximizes the number of satisfied clauses in a given set S) as an integer linear program.

13. MULTI-PROCESSOR SCHEDULING: Given n jobs and m processors, the problem is to allocate each job to one and only one of the processors so as to minimize the make span time, i.e., minimize the completion time of all the jobs. The processors may not be identical and hence job j if allocated to processor i requires p_{ij} units of time. The multi-processor scheduling problem is

$$
\begin{aligned}
(P7) \quad \text{Min} \quad & T \\
\text{subject to} \quad & \sum_{i=1}^{m} x_{ij} = 1 \quad j = 1, 2, \ldots, n \\
& \sum_{j=1}^{n} p_{ij} x_{ij} - T \leq 0 \quad i = 1, 2, \ldots, m \\
& x_{ij} = 0 \text{ or } 1 \quad \forall\, i\ j
\end{aligned}
$$

Note that if all p_{ij} are integers, the optimal solution will be such that T is an integer.

Jeroslow's Representability Theorem

R. Jeroslow [70], building on joint work with J.K. Lowe [71], characterized subsets of n-space that can be represented as the feasible region of a mixed integer (Boolean) program. They proved that a set is the feasible region of some mixed integer/linear programming problem (MILP) if and only if it is the union of finitely many polyhedra having the same recession cone (defined below). Although this result is not widely known, it might well be regarded as the fundamental theorem of mixed integer modeling.

The basic idea of Jeroslow's results is that any set that can be represented in a mixed integer model can be represented in a disjunctive programming problem (i.e., a problem with either/or constraints). A *recession direction* for a set S in n-space is a vector $\mathbf{x}$ such that $\mathbf{s} + \alpha\mathbf{x} \in S$ for all $\mathbf{s} \in S$ and all $\alpha \geq 0$. The set of recession directions is denoted $rec(S)$. Consider the general mixed integer constraint set below.

$$
\begin{aligned}
& f(\mathbf{x}, \mathbf{y}, \lambda) \leq \mathbf{b} \\
& \mathbf{x} \in \Re^n, \quad \mathbf{y} \in \Re^p \\
& \lambda = (\lambda_1, \ldots, \lambda_k), \quad \text{with } \lambda_j \in \{0, 1\} \text{ for } j = 1, \ldots, k
\end{aligned}
\tag{32.3}
$$

Here $\mathbf{f}$ is vector-valued function, so that $\mathbf{f}(\mathbf{x}, \mathbf{y}, \lambda) \leq \mathbf{b}$ represents a set of constraints. We say that a set $S \subset \Re^n$ is *represented* by (32.3) if

$$
\mathbf{x} \in S \text{ if and only if } (\mathbf{x}, \mathbf{y}, \lambda) \text{ satisfies (32.3) for some } \mathbf{y}, \lambda\ .
$$

If **f** is a linear transformation, so that (32.3) is a MILP constraint set, we will say that S is *MILP representable.* The main result can now be stated.

THEOREM 32.2 **Jeroslow, Lowe [70, 71]** *A set in n-space is MILP representable if and only if it is the union of finitely many polyhedra having the same set of recession directions.*

Benders Representation

Any mixed integer linear program (MILP) can be reformulated so that there is only one continuous variable. This reformulation, due to Benders [10], will in general have an exponential number of constraints. Benders representation suggests an algorithm for mixed integer programming (known as Benders Decomposition in the literature because of its similarity to Dantzig–Wolfe Decomposition, cf. [95]) that uses dynamic activation of these rows (constraints) as and when required.

Consider the (MILP)

$$\max\{\mathbf{cx} + \mathbf{dy} :\ A\mathbf{x} + G\mathbf{y} \ \leq\ \mathbf{b},\ \mathbf{x} \geq \mathbf{0},\ y \geq \mathbf{0} \text{ and integer}\}$$

Suppose the integer variables **y** are fixed at some values, then the associated linear program is

$$(LP)\quad \max\{\mathbf{cx} : \mathbf{x} \in \mathcal{P} = \{\mathbf{x} : A\mathbf{x} \leq \mathbf{b} - G\mathbf{y},\ \mathbf{x} \geq \mathbf{0}\}\}$$

and its dual is

$$(DLP)\quad \min\{\mathbf{w}(\mathbf{b} - G\mathbf{y}) :\ \mathbf{w} \in Q = \{\mathbf{w} : \mathbf{w}A \geq \mathbf{c},\ \mathbf{w} \geq 0\}\}$$

Let $\{\mathbf{w}^k\}$, $k = 1, 2, \ldots, K$ be the extreme points of Q and $\{\mathbf{u}^j\}$, $j = 1, 2, \ldots, J$ be the extreme rays of the recession cone of Q, $C_Q = \{\mathbf{u} : \mathbf{u}A \geq \mathbf{0},\ \mathbf{u} \geq \mathbf{0}\}$. Note that if Q is nonempty, the $\{\mathbf{u}^j\}$ are all the extreme rays of Q.

From linear programming duality, we know that if Q is empty and $\mathbf{u}^j(\mathbf{b} - G\mathbf{y}) \geq 0$, $j = 1, 2, \ldots, J$ for some $\mathbf{y} \geq \mathbf{0}$ and integer then (LP) and consequently $(MILP)$ has an unbounded solution. If Q is nonempty and $\mathbf{u}^j(\mathbf{b} - G\mathbf{y}) \geq 0$, $j = 1, 2, \ldots, J$ for some $\mathbf{y} \geq \mathbf{0}$ and integer, then (LP) has a finite optimum given by

$$\min_k \left\{\mathbf{w}^k(\mathbf{b} - G\mathbf{y})\right\}$$

Hence, an equivalent formulation of $(MILP)$ is

$$\begin{array}{rcl}
\max\ \alpha & & \\
\alpha & \leq & \mathbf{dy} + \mathbf{w}^k(\mathbf{b} - G\mathbf{y}),\quad k = 1, 2, \cdots, K \\
\mathbf{u}^j(\mathbf{b} - G\mathbf{y}) & \geq & 0,\quad j = 1, 2, \cdots, J \\
\mathbf{y} & \geq & \mathbf{0} \text{ and integer} \\
\alpha & & \text{unrestricted}
\end{array}$$

which has only one continuous variable α as promised.

Aggregation

An integer linear programming problem with only one constraint, other than upper bounds on the variables, is referred to as a **knapsack problem.** An integer linear programming problem with m constraints can be represented as a knapsack problem. In this section, we show this representation for an integer linear program with bounded variables [53, 96]. We show how two constraints can be aggregated into a single constraint. By repeated application of this aggregation, an integer linear program with m constraints can be represented as a knapsack problem.

Consider the feasible set S defined by two constraints with integer coefficients and m nonnegative integer variables with upper bounds, i.e.,

$$S = \left\{ \mathbf{x} : \sum_{j=1}^{n} a_j x_j = d_1; \quad \sum_{j=1}^{n} b_j x_j = d_2; \quad 0 \le x_j \le u_j \text{ and } x_j \text{ integer} \right\}$$

Consider now the problem

$$(P) \quad z = \max \left\{ \left| \sum_{j=1}^{n} a_j x_j - d_1 \right| \; : \; 0 \le x_j \le u_j \text{ and } x_j \text{ integer}, \; j = 1, 2, \ldots, n \right\}$$

This problem is easily solved by considering two solutions, one in which the variables x_j with positive coefficients are set to u_j while the other variables are set to zero and the other in which the variables x_j with negative coefficients are set to u_j while the other variables are set to zero.

Let α be an integer greater than z, the maximum objective value of (P).

It is easy to show that S is equivalent to

$$K = \left\{ \mathbf{x} : \sum_{j=1}^{n} \left(a_j + \alpha b_j\right) x_j = d_1 + \alpha d_2; \; 0 \le x_j \le u_j \text{ and } x_j \text{ integer} \right\}$$

Note that if $\mathbf{x}^* \in S$, then clearly $\mathbf{x}^* \in K$. Suppose $\mathbf{x}^* \in K$. Now we show that $\Sigma_{j=1}^{n} b_j x_j^* = d_2$. Suppose not and that

$$\sum_{j=1}^{n} b_j x_j^* = d_2 + k \tag{32.4}$$

where k is some arbitrary integer, positive or negative. Now multiplying (32.4) by α, and subtracting it from the equality constraint defining K, we have $\Sigma_{j=1}^{n} a_j x_j^* = d_1 - \alpha k$. But since $|\Sigma_{j=1}^{n} a_j x_j^* - d_1| < \alpha$, it follows that $k = 0$ and $\mathbf{x}^* \in S$.

32.5 Polyhedral Combinatorics

One of the main purposes of writing down an algebraic formulation of a combinatorial optimization problem as an integer program is to then examine the linear programming relaxation and understand how well it represents the discrete integer program [107]. There are somewhat special but rich classes of such formulations for which the linear programming relaxation is sharp or tight. These special structures are presented next.

Special Structures and Integral Polyhedra

A natural question of interest is whether the LP associated with an ILP has only integral extreme points. For instance, the linear programs associated with matching and edge covering polytopes in a bipartite graph have only integral vertices. Clearly, in such a situation, the ILP can be solved as LP. A polyhedron or a polytope is referred to as being integral if it is either empty or has only integral vertices.

DEFINITION 32.1 A 0, ± 1 matrix is totally unimodular if the determinant of every square submatrix is 0 or ± 1.

THEOREM 32.3 Hoffman and Kruskal [64] *Let* $A = \begin{pmatrix} A_1 \\ A_2 \\ A_3 \end{pmatrix}$ *be a* $0, \pm 1$ *matrix and* $\mathbf{b} = \begin{pmatrix} \mathbf{b}_1 \\ \mathbf{b}_2 \\ \mathbf{b}_3 \end{pmatrix}$ *be a vector of appropriate dimensions. Then A is totally unimodular if and only if the polyhedron*

$$P(A, \mathbf{b}) = \{\mathbf{x} \,:\, A_1\mathbf{x} \leq \mathbf{b}_1;\, A_2\mathbf{x} \geq \mathbf{b}_2;\, A_3\mathbf{x} = \mathbf{b}_3;\, \mathbf{x} \geq \mathbf{0}\}$$

is integral for all integral vectors **b**.

The constraint matrix associated with a network flow problem (see for instance [1]) is totally unimodular. Note that for a given integral **b**, $P(A, \mathbf{b})$ may be integral even if A is not totally unimodular.

DEFINITION 32.2 A polyhedron defined by a system of linear constraints is totally dual integral (TDI) if for each objective function with integral coefficients, the dual linear program has an integral optimal solution whenever an optimal solution exists.

THEOREM 32.4 Edmonds and Giles [43]
If $P(A) = \{\mathbf{x} \,:\, A\mathbf{x} \leq \mathbf{b}\}$ *is TDI and* **b** *is integral, then* $P(A)$ *is integral.*

Hoffman and Kruskal [64] have in fact shown that the polyhedron $P(A, \mathbf{b})$ defined in Theorem 32.3 is TDI. This follows from Theorem 32.3 and the fact that A is totally unimodular if and only if A^T is totally unimodular.

Balanced matrices, first introduced by Berge [13] have important implications for **packing and covering** problems (see also [14]).

DEFINITION 32.3 A 0, 1 matrix is balanced if it does not contain a square submatrix of odd order with two ones per row and column.

THEOREM 32.5 Berge [13], Fulkerson, Hoffman and Oppenheim [52]
Let A be a balanced 0, 1 *matrix. Then the set packing, set covering, and set partitioning polytopes associated with A are integral, i.e., the polytopes*
$P(A) = \{\mathbf{x} \,:\, \mathbf{x} \geq \mathbf{0}; A\mathbf{x} \leq \mathbf{1}\}$
$Q(A) = \{\mathbf{x} \,:\, \mathbf{0} \leq \mathbf{x} \leq \mathbf{1}; A\mathbf{x} \geq \mathbf{1}\}$ *and*
$R(A) = \{\mathbf{x} \,:\, \mathbf{x} \geq \mathbf{0}; A\mathbf{x} = \mathbf{1}\}$
are integral.

Let $A = \begin{pmatrix} A_1 \\ A_2 \\ A_3 \end{pmatrix}$ be a balanced 0, 1 matrix. Fulkerson, Hoffman and Oppenheim [52] have shown that the polytope $P(A) = \{\mathbf{x} :\, A_1\mathbf{x} \leq \mathbf{1}; A_2\mathbf{x} \geq \mathbf{1}; A_3\mathbf{x} = \mathbf{1}; \mathbf{x} \geq \mathbf{0}\}$ is TDI and by the theorem of Edmonds and Giles [43] it follows that $P(A)$ is integral.

Truemper [119] has extended the definition of balanced matrices to include $0, \pm 1$ matrices.

DEFINITION 32.4 A $0, \pm 1$ matrix is balanced if for every square submatrix with exactly two nonzero entries in each row and each column, the sum of the entries is a multiple of 4.

THEOREM 32.6 Conforti and Cornuejols [26]
Suppose A is a balanced $0, \pm 1$ *matrix. Let* $\mathbf{n}(A)$ *denote the column vector whose ith component is the number of* -1*s in the ith row of A. Then the polytopes*
$P(A) = \{\mathbf{x} \,:\, A\mathbf{x} \leq \mathbf{1} - \mathbf{n}(A); \mathbf{0} \leq \mathbf{x} \leq \mathbf{1}\}$

$Q(A) = \{\mathbf{x} : A\mathbf{x} \geq \mathbf{1} - \mathbf{n}(A); \mathbf{0} \leq \mathbf{x} \leq \mathbf{1}\}$
$R(A) = \{\mathbf{x} : A\mathbf{x} = \mathbf{1} - \mathbf{n}(A); \mathbf{0} \leq \mathbf{x} \leq \mathbf{1}\}$
are integral.

Note that a $0, \pm 1$ matrix A is balanced if and only if A^T is balanced. Moreover A is balanced (totally unimodular) if and only if every submatrix of A is balanced (totally unimodular). Thus if A is balanced (totally unimodular) it follows that Theorem 32.6 (Theorem 32.3) holds for every submatrix of A.

Totally unimodular matrices constitute a subclass of balanced matrices, i.e., a totally unimodular $0, \pm 1$ matrix is always balanced. This follows from a theorem of Camion [17], which states that a $0, \pm 1$ is totally unimodular if and only if for every square submatrix with an even number of nonzeros entries in each row and in each column, the sum of the entries equals a multiple of 4. The 4×4 matrix in Fig. 32.2 illustrates the fact that a balanced matrix is not necessarily totally unimodular. Balanced $0, \pm 1$ matrices have implications for solving the satisfiability problem. If the given set of clauses defines a balanced $0, \pm 1$ matrix, then as shown by Conforti and Cornuejols [26], the satisfiability problem is trivial to solve and the associated MAXSAT problem is solvable in polynomial time by linear programming. A survey of balanced matrices is in Conforti et al. [29].

$$A = \begin{bmatrix} 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 \end{bmatrix} \qquad A = \begin{bmatrix} 1 & 1 & 0 \\ 0 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

FIGURE 32.2 A balanced matrix and a perfect matrix.

DEFINITION 32.5 A $0, 1$ matrix A is perfect if the set packing polytope $P(A) = \{\mathbf{x} : A\mathbf{x} \leq \mathbf{1}; \mathbf{x} \geq \mathbf{0}\}$ is integral.

The chromatic number of a graph is the minimum number of colors required to color the vertices of the graph so that no two vertices with the same color have an edge incident between them. A graph G is perfect if for every node induced subgraph H, the chromatic number of H equals the number of nodes in the maximum clique of H. The connections between the integrality of the set packing polytope and the notion of a perfect graph, as defined by Berge [11, 12], are given in Fulkerson [51], Lovász [89], Padberg [97], and Chvátal [25].

THEOREM 32.7 Fulkerson [51], Lovasz [89], Chvátal [25]

Let A be $0, 1$ matrix whose columns correspond to the nodes of a graph G and whose rows are the incidence vectors of the maximal cliques of G. The graph G is perfect if and only if A is perfect.

Let G_A denote the intersection graph associated with a given $0, 1$ matrix A (see Section "Formulations"). Clearly, a row of A is the incidence vector of a clique in G_A. In order for A to be perfect, every maximal clique of G_A must be represented as a row of A because inequalities defined by maximal cliques are facet defining. Thus by Theorem 32.7, it follows that a $0, 1$ matrix A is perfect if and only if the undominated (a row of A is dominated if its support is contained in the support of another row of A) rows of A form the clique-node incidence matrix of a perfect graph.

Balanced matrices with 0, 1 entries, constitute a subclass of 0, 1 perfect matrices, i.e., if a 0, 1 matrix A is balanced, then A is perfect. The 4×3 matrix in Fig. 32.2 is an example of a matrix that is perfect but not balanced.

DEFINITION 32.6 A 0, 1 matrix A is ideal if the set covering polytope

$$Q(A) = \{\mathbf{x} \ : \ A\mathbf{x} \geq \mathbf{1}; \mathbf{0} \leq \mathbf{x} \leq \mathbf{1}\}$$

is integral.

Properties of ideal matrices are described by Lehman [85], Padberg [99] and Cornuejols and Novick [32]. The notion of a 0, 1 perfect (ideal) matrix has a natural extension to a 0, ± 1 perfect (ideal) matrix. Some results pertaining to 0, ± 1 ideal matrices are contained in Hooker [65], while some results pertaining to 0, ± 1 perfect matrices are given in Conforti, Cornuejols and De Francesco [27].

An interesting combinatorial problem is to check whether a given 0, ± 1 matrix is totally unimodular, balanced, or perfect. Seymour's [116] characterization of totally unimodular matrices provides a polynomial time algorithm to test whether a given matrix 0, 1 matrix is totally unimodular. Conforti, Cornuejols and Rao [30] give a polynomial time algorithm to check whether a 0, 1 matrix is balanced. This has been extended by Conforti et al. [28] to check in polynomial time whether a 0, ± 1 matrix is balanced. An open problem is that of checking in polynomial time whether a 0, 1 matrix is perfect. For linear matrices (a matrix is linear if it does not contain a 2×2 submatrix of all ones), this problem has been solved by Fonlupt and Zemirline [47] and Conforti and Rao [31].

Matroids

Matroids and submodular functions have been studied extensively, especially from the point of view of combinatorial optimization (see for instance Nemhauser & Wolsey [95]). Matroids have nice properties that lead to efficient algorithms for the associated optimization problems. One of the interesting examples of matroid optimization is the problem of finding a maximum or minimum weight spanning tree in a graph. Two different but equivalent definitions of a matroid are given first. A greedy algorithm to solve a linear optimization problem over a matroid is presented. The matroid intersection problem is then discussed briefly.

DEFINITION 32.7 Let $N = \{1, 2, \cdot, n\}$ be a finite set and let $\mathcal{F}$ be a set of subsets of N. Then $I = (N, \mathcal{F})$ is an independence system if $S_1 \in \mathcal{F}$ implies that $S_2 \in \mathcal{F}$ for all $S_2 \subseteq S_1$. Elements of $\mathcal{F}$ are called independent sets. A set $S \in \mathcal{F}$ is a maximal independent set if $S \cup \{j\} \notin \mathcal{F}$ for all $j \in N \backslash S$. A maximal independent set T is a maximum if $|T| \geq |S|$ for all $S \in \mathcal{F}$.

DEFINITION 32.8 The rank $r(Y)$ of a subset $Y \subseteq N$ is the cardinality of the maximum independent subset $X \subseteq Y$. Note that $r(\phi) = 0$, $r(X) \leq |X|$ for $X \subseteq N$ and the rank function is non-decreasing, i.e., $r(X) \leq r(Y)$ for $X \subseteq Y \subseteq N$.

DEFINITION 32.9 A matroid $M = (N, \mathcal{F})$ is an independence system in which every maximal independent set is a maximum.

EXAMPLE 32.1:

Let $G = (V, E)$ be an undirected connected graph with V as the node set and E as the edge set.

(i) Let $I = (E, \mathcal{F})$ where $F \in \mathcal{F}$ if $F \subseteq E$ is such that at most one edge in F is incident to each node of V, i.e., $F \in \mathcal{F}$ if F is a matching in G. Then $I = (E, \mathcal{F})$ is an independence system but not a matroid.

(ii) Let $M = (E, \mathcal{F})$ where $F \in \mathcal{F}$ if $F \subseteq E$ is such that $G_F = (V, F)$ is a forest, i.e., G_F contains no cycles. Then $M = (E, \mathcal{F})$ is a matroid and maximal independent sets of M are spanning trees.

An alternate but equivalent definition of matroids is in terms of submodular functions.

DEFINITION 32.10 Let N be a finite set. A real valued set function f defined on the subsets of N is submodular if $f(X \cup Y) + f(X \cap Y) \leq f(X) + f(Y)$ for $X, Y \subseteq N$.

EXAMPLE 32.2:

Let $G = (V, E)$ be an undirected graph with V as the node set and E as the edge set. Let $c_{ij} \geq 0$ be the weight or capacity associated with edge $(ij) \in E$. For $S \subseteq V$, define the cut function $c(S) = \Sigma_{i \in S,\, j \in V \setminus S} c_{ij}$. The cut function defined on the subsets of V is submodular since $c(X) + c(Y) - c(X \cup Y) - c(X \cap Y) = \Sigma_{i \in X \setminus Y,\, j \in Y \setminus X} 2c_{ij} \geq 0$.

DEFINITION 32.11 A nondecreasing integer valued submodular function r defined on the subsets of N is called a matroid rank function if $r(\phi) = 0$ and $r(\{j\}) \leq 1$ for $j \in N$. The pair (N, r) is called a matroid.

DEFINITION 32.12 A nondecreasing, integer-valued, submodular function f defined on the subsets of N is called a polymatroid function if $f(\phi) = 0$. The pair (N, r) is called a polymatroid.

Matroid Optimization

In order to decide whether an optimization problem over a matroid is polynomially solvable or not, we need to first address the issue of representation of a matroid. If the matroid is given either by listing the independent sets or by its rank function, many of the associated linear optimization problems are trivial to solve. However, matroids associated with graphs are completely described by the graph and the condition for independence. For instance, the matroid in which the maximal independent sets are spanning trees, the graph $G = (V, E)$ and the independence condition of no cycles describes the matroid.

Most of the algorithms for matroid optimization problems require a test to determine whether a specified subset is independent or not. We assume the existence of an oracle or subroutine to do this checking in running time which is a polynomial function of $|N| = n$.

Maximum Weight Independent Set Given a matroid $M = (N, \mathcal{F})$ and weights w_j for $j \in N$, the problem of finding a maximum weight independent set is $\max_{F \in \mathcal{F}} \left\{ \Sigma_{j \in F} w_j \right\}$. The greedy algorithm to solve this problem is as follows:

Procedure: **Greedy**

0. **Initialize:** Order the elements of N so that $w_i \geq w_{i+1}$, $i = 1, 2, \cdots, n-1$. Let $T = \phi$, $i = 1$.
1. **If** $w_i \leq 0$ or $i > n$, **stop** T is optimal, i.e., $x_j = 1$ for $j \in T$ and $x_j = 0$ for $j \notin T$. If $w_i > 0$ and $T \cup \{i\} \in \mathcal{F}$, add element i to T.
2. **Increment** i by 1 and return to Step 1.

Edmonds [40, 41] derived a complete description of the *matroid polytope,* the convex hull of the characteristic vectors of independent sets of a matroid. While this description has a large (exponential) number of constraints, it permits the treatment of linear optimization problems on independent sets of matroids as linear programs. Cunningham [35] describes a polynomial algorithm to solve the *separation problem*[2] for the matroid polytope. The matroid polytope and the associated greedy algorithm have been extended to polymatroids [40, 93].

The separation problem for a polymatroid is equivalent to the problem of minimizing a submodular function defined over the subsets of N, see Nemhauser and Wolsey [95]. A class of submodular functions that have some additional properties can be minimized in polynomial time by solving a maximum flow problem (Rhys [109], Picard and Ratliff [106]). The general submodular function can be minimized in polynomial time by the ellipsoid algorithm (Grötschel, Lovász and Schrijver [60]).

The uncapacitated plant location problem (see "LP Relaxation") can be reduced to maximizing a submodular function. Hence it follows that maximizing a submodular function is NP-hard.

Matroid Intersection

A matroid intersection problem involves finding an independent set contained in two or more matroids defined on the same set of elements.

Let $G = (V_1, V_2, E)$ be a bipartite graph. Let $M_i = (E, \mathcal{F}_i)$, $i = 1, 2$ where $F \in \mathcal{F}_i$ if $F \subseteq E$ is such that no more than one edge of F is incident to each node in V_i. The set of matchings in G constitute the intersection of the two matroids M_i, $i = 1, 2$. The problem of finding a maximum weight independent set in the intersection of two matroids can be solved in polynomial time (Lawler [84], Edmonds [40, 42], Frank [49]). The two (poly) matroid intersection polytope has been studied by Edmonds [42].

The problem of testing whether a graph contains a Hamiltonian path is NP-complete. Since this problem can be reduced to the problem of finding a maximum cardinality independent set in the intersection of three matroids, it follows that the matroid intersection problem involving three or more matroids is NP-hard.

Valid Inequalities, Facets and Cutting Plane Methods

In "Special Structures and Integral Polyhedra," we were concerned with conditions under which the packing and covering polytopes are integral. But in general these polytopes are not integral and additional inequalities are required to have a complete linear description of the convex hull of integer solutions. The existence of finitely many such linear inequalities is guaranteed by Weyl's Theorem [120].

Consider the feasible region of an ILP given by

$$P_I = \{\mathbf{x} \ : \ A\mathbf{x} \leq \mathbf{b}; \mathbf{x} \geq \mathbf{0} \text{ and integer}\} \tag{32.5}$$

Recall that an inequality $\mathbf{fx} \leq f_0$ is referred to as a valid inequality for P_I if $\mathbf{fx}^* \leq f_0$ for all $\mathbf{x}^* \in P_I$. A valid linear inequality for $P_I(A, \mathbf{b})$ is said to be facet defining if it intersects $P_I(A, \mathbf{b})$ in a face of dimension one less than the dimension of $P_I(A, \mathbf{b})$. In the example shown in Fig. 32.3, the inequality $x_2 + x_3 \leq 1$ is a facet defining inequality of the integer hull.

[2]The separation problem for a convex body K is to test if an input point $\mathbf{x}$ belongs to K and if it does not to produce a linear halfspace that separates $\mathbf{x}$ from K. It is known [60, 80, 100] that linear optimization over K is polynomially equivalent to separation from K.

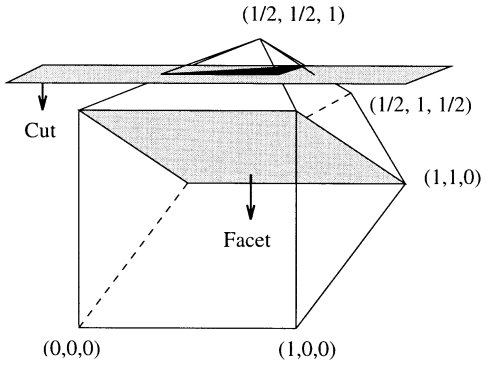


FIGURE 32.3 Relaxation, cuts and facets.

The Relaxation

$$\begin{aligned} x_1 + x_2 + x_3 &\leq 2 \\ x_1 - x_2 - x_3 &\geq -1 \\ 0 \leq x_1, x_2, x_3 &\leq 1 \end{aligned}$$

The Integer Hull

$$\begin{aligned} x_1 + x_2 + x_3 &\leq 2 \\ x_1 - x_2 - x_3 &\geq -1 \\ x_2 + x_3 &\leq 1 \\ 0 \leq x_1, x_2, x_3 &\leq 1 \end{aligned}$$

Let $\mathbf{u} \geq 0$ be a row vector of appropriate size. Clearly $\mathbf{u}A\mathbf{x} \leq \mathbf{ub}$ holds for every $\mathbf{x}$ in P_I. Let $(\mathbf{u}A)_j$ denote the jth component of the row vector $\mathbf{u}A$ and $\lfloor(\mathbf{u}A)_j\rfloor$ denote the largest integer less than or equal to $(\mathbf{u}A)_j$. Now, since $\mathbf{x} \in P_I$ is a vector of nonnegative integers, it follows that $\Sigma_j \lfloor(\mathbf{u}A)_j\rfloor x_j \leq \lfloor\mathbf{ub}\rfloor$ is a valid inequality for P_I. This scheme can be used to generate many valid inequalities by using different $\mathbf{u} \geq 0$. Any set of generated valid inequalities may be added to the constraints in (32.5) and the process of generating them may be repeated with the enhanced set of inequalities. This iterative procedure of generating valid inequalities is called Gomory–Chvátal (GC) rounding. It is remarkable that this simple scheme is complete, i.e., every valid inequality of P_I can be generated by finite application of GC rounding [24, 113].

The number of inequalities needed to describe the convex hull of P_I is usually exponential in the size of A. But to solve an optimization problem on P_I, one is only interested in obtaining a partial description of P_I that facilitates the identification of an integer solution and prove its optimality. This is the underlying basis of any cutting plane approach to combinatorial problems.

The Cutting Plane Method

Consider the optimization problem

$$\max\{\mathbf{cx} : \mathbf{x} \in P_I\{\mathbf{x} : A\mathbf{x} \leq \mathbf{b}; \mathbf{x} \geq \mathbf{0} \text{ and integer}\}\}$$

The generic cutting plane method as applied to this formulation is given below.

Procedure: **Cutting Plane**

1. Initialize $A' \leftarrow A$ and $\mathbf{b}' \leftarrow \mathbf{b}$.
2. Find an optimal solution $\bar{\mathbf{x}}$ to the linear program

$$\max\{\mathbf{cx} : A'\mathbf{x} \leq \mathbf{b}'; \mathbf{x} \geq \mathbf{0}\}$$

   If $\bar{\mathbf{x}} \in P_I$, stop and return $\bar{\mathbf{x}}$.
3. Generate a valid inequality $\mathbf{fx} \leq f_0$ for P_I such that $f\bar{\mathbf{x}} > f_0$ (the inequality ''cuts'' $\bar{\mathbf{x}}$).
4. Add the inequality to the constraint system, update

$$A' \leftarrow \begin{pmatrix} A' \\ \mathbf{f} \end{pmatrix}, \quad \mathbf{b}' \leftarrow \begin{pmatrix} \mathbf{b}' \\ f_0 \end{pmatrix}$$

   Go to Step 2.

In Step 3 of the cutting plane method, we require a suitable application of the GC rounding scheme (or some alternate method of identifying a cutting plane). Notice that while the GC rounding scheme will generate valid inequalities, the identification of one that cuts off the current solution to the linear programming relaxation is all that is needed. Gomory [56] provided just such a specialization of the rounding scheme that generates a cutting plane. While this met the theoretical challenge of designing a sound and complete cutting plane method for integer linear programming, it turned out to be a weak method in practice. Successful cutting plane methods, in use today, use considerable additional insights into the structure of facet-defining cutting planes. Using facet cuts makes a huge difference in the speed of convergence of these methods. Also, the idea of combining cutting plane methods with search methods has been found to have a lot of merit. These branch and cut methods will be discussed in the next section.

The b-Matching Problem

Consider the **b**-matching problem:

$$\max\{\mathbf{cx} : \ A\mathbf{x} \leq \mathbf{b}, \ \mathbf{x} \geq \mathbf{0} \ \text{ and } \ \text{integer}\} \tag{32.6}$$

where A is the node-edge incidence matrix of an undirected graph and $\mathbf{b}$ is a vector of positive integers. Let G be the undirected graph whose node-edge incidence matrix is given by A and let $W \subseteq V$ be any subset of nodes of G (i.e., subset of rows of A) such that $\mathbf{b}(W) = \Sigma_{i \in W} b_i$ is odd. Then the inequality

$$\mathbf{x}(W) = \sum_{e \in E(W)} x_e \leq \frac{1}{2}(\mathbf{b}(W) - 1) \tag{32.7}$$

is a valid inequality for integer solutions to (32.6) where $E(W) \subseteq E$ is the set of edges of G having both ends in W. Edmonds [39] has shown that the inequalities (32.6) and (32.7) define the integral **b**-matching polytope (see also [44]). Note that the number of inequalities (32.7) is exponential in the number of nodes of G. An instance of the successful application of the idea of using only a partial description of P_I is in the blossom algorithm for the matching problem, due to Edmonds [39].

An implication of the ellipsoid method for linear programming is that the linear program over P_I can be solved in polynomial time if and only if the associated separation problem can be solved in polynomial time (see Grotschel, Lovász and Schrijver [60], Karp and Papadimitriou [80], and Padberg and Rao [100]). The separation problem for the **b**-matching problem with or without upper bounds was shown by Padberg and Rao [101], to be solvable in polynomial time. The procedure involves a minor modification of the algorithm of Gomory and Hu [59] for multiterminal networks. However, no polynomial (in the number of nodes of the graph) linear programming formulation of this separation problem is known. Martin [92] has shown that if the separation problem can be expressed as a compact linear program then so can the optimization problem. Hence an unresolved issue is whether there exists a polynomial size (compact) formulation for the **b**-matching problem. Yannakakis [123] has shown that, under a symmetry assumption, such a formulation is impossible.

Other Combinatorial Problems

Besides the matching problem several other combinatorial problems and their associated polytopes have been well studied and some families of facet defining inequalities have been identified. For instance the set packing, graph partitioning, plant location, maximum cut, traveling salesman and Steiner tree problems have been extensively studied from a polyhedral point of view (see for instance Nemhauser and Wolsey [95]).

These combinatorial problems belong to the class of NP-complete problems. In terms of a worst-case analysis, no polynomial time algorithms are known for these problems. Nevertheless, using a cutting plane approach with branch and bound or branch and cut (see Section 32.6), large instances of these problems

have been successfully solved, see Crowder, Johnson and Padberg [34], for general $0-1$ problems, Barahona et al. [7] for the maximum cut problem, Padberg and Rinaldi [102] for the traveling salesman problem and Chopra, Gorres and Rao [23] for the Steiner tree problem.

32.6 Partial Enumeration Methods

In many instances, to find an optimal solution to an integer linear programing problems (ILP), the structure of the problem is exploited together with some sort of partial enumeration. In this section, we review the branch and bound (B & B) and branch and cut (B & C) methods for solving an ILP.

Branch and Bound

The branch and bound (B & B) method is a systematic scheme for implicitly enumerating the finitely many feasible solutions to an ILP. Although, theoretically the size of the enumeration tree is exponential in the problem parameters, in most cases, the method eliminates a large number of feasible solutions. The key features of branch and bound method are

(i) **Selection/Removal** of one or more problems from a candidate list of problems.

(ii) **Relaxation** of the selected problem so as to obtain a lower bound (on a minimization problem) on the optimal objective function value for the selected problem.

(iii) **Fathoming,** if possible, of the selected problem.

(iv) **Branching Strategy:** If the selected problem is not fathomed, branching creates subproblems which are added to the candidate list of problems.

The above four steps are repeated until the candidate list is empty. The B & B method sequentially examines problems that are added and removed from a candidate list of problems.

Initialization Initially the candidate list contains only the original ILP which is denoted as

$$(P) \qquad \min\{\mathbf{cx} : A\mathbf{x} \leq \mathbf{b},\ \mathbf{x} \geq \mathbf{0} \text{ and integer}\}$$

Let $F(P)$ denote the feasible region of (P) and $z(P)$ denote the optimal objective function value of (P). For any $\bar{\mathbf{x}}$ in $F(P)$, let $z_P(\bar{\mathbf{x}}) = \mathbf{c}\bar{\mathbf{x}}$.

Frequently, heuristic procedures are first applied to get a good feasible solution to (P). The best solution known for (P) is referred to as the current incumbent solution. The corresponding objective function value is denoted as z_I. In most instances, the initial heuristic solution is not either optimal or at least not immediately certified to be optimal. So further analysis is required to ensure that an optimal solution to (P) is obtained. If no feasible solution to (P) is known, z_I is set to ∞.

Selection/Removal In each iterative step of B & B, a problem is selected and removed from the candidate list for further analysis. The selected problem is henceforth referred to as the candidate problem (CP). The algorithm terminates if there is no problem to select from the candidate list. Initially there is no issue of selection since the candidate list contains only the problem (P). However, as the algorithm proceeds, there would be many problems on the candidate list and a selection rule is required. Appropriate selection rules, also referred to as branching strategies, are discussed later. Conceptually, several problems may be simultaneously selected and removed from the candidate list. However, most sequential implementations of B & B select only one problem from the candidate list and this is assumed henceforth. Parallel aspects of B & B on $0-1$ integer linear programs are discussed in Cannon and Hoffman [18] and for the case of traveling salesman problems in [4].

The computational time required for the B & B algorithm depends crucially on the order in which the problems in the candidate list are examined. A number of clever heuristic rules may be employed in devising such strategies. Two general purpose selection strategies that are commonly used are

(A) Choose the problem that was added last to the candidate list. This last-in-first-out rule (LIFO) is also called depth first search (DFS), since the selected candidate problem increases the depth of the active enumeration tree.

(B) Choose the problem on the candidate list that has the least lower bound. Ties may be broken by choosing the problem that was added last to the candidate list. This rule would require that a lower bound be obtained for each of the problems on the candidate list. In other words, when a problem is added to the candidate list, an associated lower bound should also be stored. This may be accomplished by using ad-hoc rules or by solving a relaxation of each problem before it is added to the candidate list.

Rule (A) is known to empirically dominate the rule (B) when storage requirements for candidate list and computation time to solve (P) are taken into account. However, some analysis indicates that rule (B) can be shown to be superior if minimizing the number of candidate problems to be solved is the criterion (see Parker and Rardin [105]).

Relaxation In order to analyze the selected candidate problem, (CP), a relaxation (CP_R) of (CP) is solved to obtain a lower bound $z(CP_R) \leq z(CP)$. (CP_R) is a relaxation of (CP) if

(i) $F(CP) \subseteq F(CP_R)$,

(ii) for $\bar{\mathbf{x}} \in F(CP)$, $z_{CP_R}(\bar{\mathbf{x}}) \leq z_{CP}(\bar{\mathbf{x}})$ and

(iii) for $\bar{\mathbf{x}}, \hat{\mathbf{x}} \in F(CP)$, $z_{CP_R}(\bar{\mathbf{x}}) \leq z_{CP_R}(\hat{\mathbf{x}})$ implies that $z_{CP}(\bar{\mathbf{x}}) \leq z_{CP}(\hat{\mathbf{x}})$.

Relaxations are needed because the candidate problems are typically hard to solve. The relaxations used most often are either linear programming or Lagrangean relaxations of (CP); see Section 32.7 for details. Sometimes, instead of solving a relaxation of (CP), a lower bound is obtained by using some ad-hoc rules such as penalty functions.

Fathoming A candidate problem is fathomed if

(FC1) analysis of (CP_R) reveals that (CP) is infeasible. For instance if $F(CP_R) = \phi$, then $F(CP) = \phi$.

(FC2) analysis of (CP_R) reveals that (CP) has no feasible solution better than the current incumbent solution. For instance if $z(CP_R) \geq z_I$, then $z(CP) \geq z(CP_R) \geq z_I$.

(FC3) analysis of (CP_R) reveals an optimal solution of (CP). For instance, if $\mathbf{x}_R$ is optimal for (CP_R) and is feasible in (CP), then $(\mathbf{x}_R)$ is an optimal solution to (CP) and $z(CP) = z_{CP}(\mathbf{x}_R)$.

(FC4) analysis of (CP_R) reveals that (CP) is dominated by some other problem, say CP^*, in the candidate list. For instance if it can shown that $z(CP^*) \leq z(CP)$, then there is no need to analyze (CP) further.

If a candidate problem (CP) is fathomed using any of the criteria above, then further examination of (CP) or its descendants (subproblems) obtained by separation is not required. If (FC3) holds, and $z(CP) < z_I$, the incumbent is updated as $\mathbf{x}_R$ and z_I is updated as $z(CP)$.

Separation/Branching If the candidate problem (CP) is not fathomed, then CP is separated into several problems, say $(CP_1), (CP_2), \ldots, (CP_q)$ where $\bigcup_{t=1}^{q} F(CP_t) = F(CP)$ and typically

$$F(CP_i) \cap F(CP_j) = \phi \quad \forall i \neq j \ .$$

For instance a separation of (CP) into $(CP_i), i = 1, 2, \ldots, q$ is obtained by fixing a single variable, say x_j, to one of the q possible values of x_j in an optimal solution to (CP). The choice of the variable to fix depends upon the separation strategy which is also part of the branching strategy. After separation, the subproblems are added to the candidate list. Each subproblem (CP_t) is a restriction of (CP) since $F(CP_t) \subseteq F(CP)$. Consequently $z(CP) \leq z(CP_t)$ and $z(CP) = \min_t z(CP_t)$.

The various steps in the B & B algorithm are outlined below.

Procedure: **B & B**

0. **Initialize:** Given the problem (P), the incumbent value z_I is obtained by applying some heuristic (if a feasible solution to (P) is not available, set $z_I = +\infty$). Initialize the candidate list $C \leftarrow \{(P)\}$.
1. **Optimality:** If $C = \emptyset$ and $z_I = +\infty$, then (P) is infeasible, stop. Stop also if $C = \emptyset$ and $z_I < +\infty$, the incumbent is an optimal solution to (P).
2. **Selection:** Using some candidate selection rule, select and remove a problem $(CP) \in C$.
3. **Bound:** Obtain a lower bound for (CP) by either solving a relaxation (CP_R) of (CP) or by applying some ad-hoc rules. If (CP_R) is infeasible, return to Step 1. Else, let $\mathbf{x}_R$ be an optimal solution of (CP_R).
4. **Fathom:** If $z(CP_R) \geq z_I$, return to Step 1. Else if $\mathbf{x}_R$ is feasible in (CP) and $z(CP) < z_I$, set $z_I \leftarrow z(CP)$, update the incumbent as $\mathbf{x}_R$ and return to Step 1. Finally, if $\mathbf{x}_R$ is feasible in (CP) but $z(CP) \geq z_I$, return to Step 1.
5. **Separation:** Using some separation or branching rule, separate (CP) into $(CP_i), i = 1, 2, \cdots, q$ and set $C \leftarrow C \cup \{CP_1), (CP_2), \cdots, (CP_q)\}$ and return to Step 1.
6. **End Procedure.**

Although the B & B method is easy to understand, the implementation of this scheme for a particular ILP is a nontrivial task requiring

(A) A relaxation strategy with efficient procedures for solving these relaxations,
(B) Efficient data-structures for handling the rather complicated book-keeping of the candidate list,
(C) Clever strategies for selecting promising candidate problems, and
(D) Separation or branching strategies that could effectively prune the enumeration tree.

A key problem is that of devising a relaxation strategy (A), i.e., to find "good relaxations" which are significantly easier to solve than the original problems and tend to give sharp lower bounds. Since these two are conflicting, one has to find a reasonable trade-off.

Branch and Cut

In the last few years, the branch and cut (B & C) method has become popular for solving combinatorial optimization problems. As the name suggests, the B & C method incorporates the features of both the branch and bound method presented above and the cutting plane method presented in the previous section. The main difference between the B & C method and the general B & B scheme is in the bound step (Step 3).

A distinguishing feature of the B & C method is that the relaxation (CP_R) of the candidate problem (CP) is a linear programming problem and instead of merely solving (CP_R), an attempt is made to solve (CP) by using cutting planes to tighten the relaxation. If (CP_R) contains inequalities that are valid for (CP) but not for the given ILP, then the GC rounding procedure may generate inequalities that are valid for (CP) but not for the ILP. In the B & C method, the inequalities that are generated are always valid for the ILP, and hence can be used globally in the enumeration tree.

Another feature of the B & C method is that often heuristic methods are used to convert some of the fractional solutions, encountered during the cutting plane phase, into feasible solutions of the (CP) or more generally of the given ILP. Such feasible solutions naturally provide upper bounds for the ILP. Some of these upper bounds may be better than the previously identified best upper bound and if so, the current incumbent is updated accordingly.

We thus obtain the B & C method by replacing the bound step (Step 3) of the B & B method by Steps 3(a) and 3(b), and also by replacing the fathom step (Step 4) by Steps 4(a) and 4(b) given below.

3(a) **Bound:** Let (CP_R) be the LP relaxation of (CP). Attempt to solve (CP) by a cutting plane method which generates valid inequalities for (P). Update the constraint System of (P) and the incumbent as appropriate.

Let $F\mathbf{x} \leq \mathbf{f}$ denote all the valid inequalities generated during this phase. Update the constraint system of (P) to include all the generated inequalities, i.e., set $A^T \leftarrow (A^T, F^T)$ and $\mathbf{b}^T \leftarrow (\mathbf{b}^T, \mathbf{f}^T)$. The constraints for all the problems in the candidate list are also to be updated.

During the cutting plane phase, apply heuristic methods to convert some of the identified fractional solutions into feasible solutions to (P). If a feasible solution, $\bar{\mathbf{x}}$, to (P), is obtained such that $\mathbf{c}\bar{\mathbf{x}} < z_I$, update the incumbent to $\bar{\mathbf{x}}$ and z_I to $c\bar{\mathbf{x}}$. Hence the remaining changes to B & B are as follows:

3(b) **If** (CP) is solved go to Step 4(a). **Else,** let $\hat{\mathbf{x}}$ be the solution obtained when the cutting plane phase is terminated (we are unable to identify a valid inequality of (P) that is violated by $\hat{\mathbf{x}}$.) go to Step 4(b).

4(a) **Fathom by Optimality:** Let $\mathbf{x}^*$ be an optimal solution to (CP). If $z(CP) < z_I$, set $\mathbf{x}_I \leftarrow z(CP)$ and update the incumbent as $\mathbf{x}^*$. Return to Step 1.

4(b) **Fathom by Bound:** If $\mathbf{c}\hat{\mathbf{x}} \geq z_I$, return to Step 1. Else go to Step 5.

The incorporation of a cutting plane phase into the B & B scheme involves several technicalities which require careful design and implementation of the B & C algorithm. Details of the state of the art in cutting plane algorithms including the B & C algorithm are reviewed in Jünger, Reinelt and Thienel [72].

32.7 Relaxations

The effectiveness of a partial enumeration strategy such as branch and bound is closely related to the quality of the relaxations used to generate the bounds and incumbents. We describe four general relaxation methods that together cover the most successful computational techniques for bound evaluation in the practice of partial enumeration for integer programming. These are linear programming relaxation, Lagrangean relaxation, group relaxation, and semidefinite relaxation methods.

LP Relaxation

A linear programming relaxation is derived from an integer programming formulation by relaxing the integrality constraints. When there are alternate integer programming formulations, modeling the same decision problem, it becomes necessary to have some criteria for selecting from among the candidate relaxations. We illustrate these ideas on the plant location model, a prototypical integer programming example.

Plant Location Problems

Given a set of customer locations $N = \{1, 2, \ldots, n\}$ and a set of potential sites for plants $M = \{1, 2, \ldots, m\}$, the plant location problem is to identify the sites where the plants are to be located so that the customers are served at a minimum cost. There is a fixed cost f_i of locating the plant at site i and the cost of serving customer j from site i is c_{ij}. The decision variables are

y_i is set to 1 if a plant is located at site i and to 0 otherwise.

x_{ij} is set to 1 if site i serves customer j and to 0 otherwise.

A formulation of the problem is

$$
\begin{aligned}
(P8) \quad \text{Min} \quad & \sum_{i=1}^{m}\sum_{j=1}^{n} c_{ij}x_{ij} \quad + \sum_{i=1}^{m} f_i y_i \\
\text{subject to} \quad & \sum_{i=1}^{m} x_{ij} = 1 \qquad j = 1, 2, \ldots, n \\
& x_{ij} - y_i \leq 0 \qquad i = 1, 2, \ldots, m; \;\; j = 1, 2, \ldots, n \\
& y_i = 0 \text{ or } 1 \qquad i = 1, 2, \ldots, m \\
& x_{ij} = 0 \text{ or } 1 \qquad i = 1, 2, \ldots, m; \;\; j = 1, 2, \ldots, n
\end{aligned}
$$

Note that the constraints $x_{ij} - y_i \leq 0$ are required to ensure that customer j may be served from site i only if a plant is located at site i. Note that the constraints $y_i = 0$ or 1, force an optimal solution in which $x_{ij} = 0$ or 1. Consequently the $x_{ij} = 0$ or 1 constraints may be replaced by non-negativity constraints $x_{ij} \geq 0$.

The linear programming relaxation associated with (P8) is obtained by replacing constraints $y_i = 0$ or 1, and $x_{ij} = 0$ or 1 by nonnegativity constraints on x_{ij} and y_i. The upper bound constraints on y_i are not required provided $f_i \geq 0, \;\; i = 1, 2, \ldots, m$. The upper bound constraints on x_{ij} are not required in view of constraints $\Sigma_{i=1}^{m} x_{ij} = 1$.

Remark: It is frequently possible to formulate the same combinatorial problem as two or more different ILPs. Suppose we have two ILP formulations (F1) and (F2) of the given combinatorial problem with both (F1) and (F2) being minimizing problems. Formulation (F1) is said to be stronger than (F2) if (LP1), the linear programming relaxation of (F1), always has an optimal objective function value which is greater than or equal to the optimal objective function value of (LP2) which is the linear programming relaxation of (F2).

It is possible to reduce the number of constraints in (P8) by replacing the constraints $x_{ij} - y_i \leq 0$ by an aggregate:

$$\sum_{j=1}^{n} x_{ij} - ny_i \leq 0 \;\; i = 1, 2, \ldots, m$$

However, the disaggregated (P8) is a stronger formulation than the formulation obtained by aggregrating the constraints as above. By using standard transformations, (P8) can also be converted into a set packing problem.

Lagrangean Relaxation

This approach has been widely used for about two decades now in many practical applications. Lagrangean relaxation, like linear programming relaxation, provides bounds on the combinatorial optimization problem being relaxed (i.e., lower bounds for minimization problems).

Lagrangean relaxation has been so successful because of a couple of distinctive features. As was noted earlier, in many hard combinatorial optimization problems, we usually have some "nice" tractable embedded subproblems which admit efficient algorithms. Lagrangean relaxation gives us a framework to

"jerry-rig" an approximation scheme that uses these efficient algorithms for the subproblems as subroutines. A second observation is that it has been empirically observed that well-chosen Lagrangean relaxation strategies usually provide very tight bounds on the optimal objective value of integer programs. This is often used to great advantage within partial enumeration schemes to get very effective pruning tests for the search trees.

Practitioners also have found considerable success with designing heuristics for combinatorial optimization by starting with solutions from Lagrangean relaxations and constructing good feasible solutions via so-called dual ascent strategies. This may be thought of as the analogue of rounding strategies for linear programming relaxations (but with no performance guarantees—other than empirical ones).

Consider a representation of our combinatorial optimization problem in the form

$$(P) \quad z = \min \left\{ \mathbf{cx} : A\mathbf{x} \geq \mathbf{b},\ \mathbf{x} \in X \subseteq \Re^n \right\}$$

Implicit in this representation is the assumption that the explicit constraints ($A\mathbf{x} \geq \mathbf{b}$) are *"small"* in number. For convenience let us also assume that X can be replaced by a finite list $\{\mathbf{x}^1, \mathbf{x}^2, \ldots, \mathbf{x}^T\}$.

The following definitions are with respect to (P):

- `Lagrangean` $L(\mathbf{u},\mathbf{x}) = \mathbf{u}(A\mathbf{x} - \mathbf{b}) + \mathbf{cx}$ where $\mathbf{u}$ are the Lagrange multipliers.
- `Lagrangean Subproblem` $\min_{\mathbf{x}\in X}\{L(\mathbf{u},\mathbf{x})\}$
- `Lagrangean-Dual Function` $\mathcal{L}(\mathbf{u}) = \min_{\mathbf{x}\in X}\{L(\mathbf{u},\mathbf{x})\}$
- `Lagrangean-Dual Problem` (D) $d = \max_{\mathbf{u}\geq\mathbf{0}}\{\mathcal{L}(\mathbf{u})\}$

It is easily shown that (D) satisfies a weak duality relationship with respect to (P), i.e., $z \geq d$. The discreteness of X also implies that $\mathcal{L}(\mathbf{u})$ is a piecewise linear and concave function (see Shapiro [115]). In practice, the constraints X are chosen such that the evaluation of the Lagrangean dual function $\mathcal{L}(\mathbf{u})$ is easily made.

An Example: `Traveling Salesman Problem (TSP)`

For an undirected graph G, with costs on each edge, the TSP is to find a minimum cost set H of edges of G such that it forms a Hamiltonian cycle of the graph. H is a Hamiltonian cycle of G if it is a simple cycle that spans all the vertices of G. Alternately H must satisfy

1. Exactly two edges of H are adjacent to each node, and
2. H forms a connected, spanning subgraph of G.

Held and Karp [63] used these observations to formulate a Lagrangean relaxation approach for TSP, that relaxes the degree constraints (condition 1 above) Notice that the resulting subproblems are minimum spanning tree problems that can be easily solved.

The most commonly used general method of finding the optimal multipliers in Lagrangean relaxation is subgradient optimization (cf. Held et al. [62]). Subgradient optimization is the non-differentiable counterpart of steepest descent methods. Given a dual vector $\mathbf{u}^k$, the iterative rule for creating a sequence of solutions is given by

$$\mathbf{u}^{k+1} = \mathbf{u}^k + t_k\, \gamma\left(\mathbf{u}^k\right)$$

where t_k is an appropriately chosen step size, and $\gamma(\mathbf{u}^k)$ is a subgradient of the dual function $\mathcal{L}$ at $\mathbf{u}^k$. Such a subgradient is easily generated by

$$\gamma\left(\mathbf{u}^k\right) = A\mathbf{x}^k - \mathbf{b}$$

where $\mathbf{x}^k$ is an optimal solution of $\min_{\mathbf{x}\in X}\{L(\mathbf{u}^k, \mathbf{x})\}$.

Subgradient optimization has proven effective in practice for a variety of problems. It is possible to choose the step sizes $\{t_k\}$ to guarantee convergence to the optimal solution. Unfortunately, the method is not finite, in that the optimal solution is attained only in the limit. Further, it is not a pure descent method. In practice, the method is heuristically terminated and the best solution in the generated sequence is recorded. In the context of nondifferentiable optimization, the ellipsoid algorithm was devised by Shor [118] to overcome precisely some of these difficulties with the subgradient method.

The ellipsoid algorithm may be viewed as a scaled subgradient method in much the same way as variable metric methods may be viewed as scaled steepest descent methods (cf. [2]). And if we use the ellipsoid method to solve the Lagrangean dual problem, we obtain the following as a consequence of the polynomial-time equivalence of optimization and separation.

THEOREM 32.8 *The Lagrangean dual problem is polynomial-time solvable if and only if the Lagrangean subproblem is. Consequently, the Lagrangean dual problem is $\mathcal{NP}$-hard if and only if the Lagrangean subproblem is.*

The theorem suggests that in practice, if we set up the Lagrangean relaxation so that the subproblem is tractable, then the search for optimal Lagrangean multipliers is also tractable.

Group Relaxations

A relaxation of the integer programming problem is obtained by dropping the nonnegativity restrictions on some variables. Consider the integer linear program

$$(ILP) \quad \max\{\mathbf{cx} \ : \ A\mathbf{x} = \mathbf{b}, \mathbf{x} \geq \mathbf{0} \text{ and integer}\}$$

where the elements of A, b are integral.

Suppose the linear programming relaxation of (ILP) has a finite optimum, then an extreme-point optimal solution $\mathbf{x}^* = \begin{pmatrix} B^{-1}\mathbf{b} \\ \mathbf{0} \end{pmatrix}$ where B is a nonsingular submatrix of A. Let $A = (B, N)$, $\mathbf{x} = \begin{pmatrix} \mathbf{x}_B \\ \mathbf{x}_N \end{pmatrix}$, and $\mathbf{c} = (\mathbf{c}_B, \mathbf{c}_N)$. By dropping the nonnegativity constraint on $\mathbf{x}_B$, substituting $\mathbf{x}_B = B^{-1}(\mathbf{b} - N\mathbf{x}_N)$, ignoring the constant term $\mathbf{c}_B B^{-1} b$ in the objective function, and changing the objective function from maximize to minimize, we obtain the following relaxation of (ILP).

$$(ILP_B) \quad \min \left\{ \left(\mathbf{c}_B B^{-1} N - \mathbf{c}_N \right) \mathbf{x}_N \ : \ \mathbf{x}_B = B^{-1} (\mathbf{b} - N\mathbf{x}_N) , \ \mathbf{x}_B \text{ integer}, \ \mathbf{x}_N \geq \mathbf{0} \text{ and integer} \right\}$$

Now $\mathbf{x}_B = B^{-1}(\mathbf{b} - N\mathbf{x}_N)$ and $\mathbf{x}_B$ integer is equivalent to requiring

$$B^{-1} N \mathbf{x}_N \equiv B^{-1}\mathbf{b} (mod \ \mathbf{1})$$

where the congruence is with respect to each element of the vector $B^{-1}\mathbf{b}$ taken modulo 1.

Hence, (ILP_B) can be written as

$$(ILP_G) \quad \min \left\{ \left(\mathbf{c}_B B^{-1} N - \mathbf{c}_N \right) \mathbf{x}_N \ : \ B^{-1} N \mathbf{x}_N \equiv B^{-1}\mathbf{b} (mod \ \mathbf{1}), \ \mathbf{x}_N \geq \mathbf{0} \text{ and integer} \right\}$$

Since A and $\mathbf{b}$ are integer matrices, the fractional parts of elements of $B^{-1}N$ and $B^{-1}\mathbf{b}$ are of the form $\left(\frac{\text{k}}{\text{detB}}\right)$ where $k \in \{0, 1, \ldots, |det B| - 1\}$. The congruences in (ILP_G) are equivalent to working in a product of cyclic groups. The structure of the product group is revealed by the Hermite normal form of B (see "Linear Diophantine Systems"). Hence, (ILP_G) is referred to as a group (knapsack) problem and is solved by a dynamic programming algorithm [58].

Semidefinite Relaxation

Semidefinite programs are linear optimization problems defined over a cone of positive semidefinite matrices. These are models that generalize linear programs and are specializations of convex programming models. There are theoretical and "practical" algorithms for solving semidefinite programs in polynomial time [3]. Lovász and Schrijver [91] suggest a general relaxation strategy for $0-1$ integer programming problems that obtains semidefinite relaxations. The first step is to consider a homogenized version of a $0-1$ integer program (solvability version).

$$F_I = \left\{ \mathbf{x} \in \Re^{n+1} \,:\, A\mathbf{x} \geq 0,\ x_0 = 1,\ x_i \in \{0,1\} \text{ for } i = 1, 2, \cdots, n \right\}$$

The problem is to check if F_I is nonempty. Note that any $0-1$ integer program can be put in this form by absorbing a general right-hand side b as the negative of x_0 column of A. Now a linear programming relaxation of this integer program is given by:

$$\left\{ \mathbf{x} \in \Re^{n+1} \,:\, A\mathbf{x} \geq 0,\ x_0 = 1,\ 0 \leq x_i \leq x_0 \text{ for } i = 1, 2, \cdots, n \right\}$$

Next, we define two polyhedral cones.

$$\begin{aligned} \mathcal{K} &= \left\{ \mathbf{x} \in \Re^{n+1} \,:\, A\mathbf{x} \geq 0,\ 0 \leq x_i \leq x_0 \text{ for } i = 0, 1, \cdots, n \right\} \\ \mathcal{K}_I &= \text{Cone generated by } 0-1 \text{ vectors in } P_I \end{aligned}$$

Lovász and Schrijver [91] show how we might construct a family of convex cones $\{\mathcal{C}\}$ such that $\mathcal{K}_I \subseteq \mathcal{C} \subseteq \mathcal{K}$ for each $\mathcal{C}$.

(i) Partition the cone constraints of $\mathcal{K}$ into $T_1 = \{A_1\mathbf{x} \geq 0\}$ and $T_2 = \{A_2\mathbf{x} \geq 0\}$, with the constraints $\{0 \leq x_i \leq x_0 \text{ for } i = 0, 1, \cdots, n\}$ repeated in both (the overlap can be larger).

(ii) Multiply each constraint in T_1 with each constraint in T_2 to obtain a quadratic homogeneous constraint.

(iii) Replace each occurrence of a quadratic term $x_i x_j$ by a new variable X_{ij}. The quadratic constraints are now linear homogeneous constraints in X_{ij}.

(iv) Add the requirement that the $(n+1) \times (n+1)$ matrix X is symmetric and positive semidefinite.

(v) Add the constraints $X_{0i} = X_{ii}$ for $i = 1, 2, \ldots, n$.

The system of constraints on the X_{ij} constructed in steps (iii), (iv), and (v) above define a cone $M_+(T_1, T_2)$ parametrized by the partition T_1, T_2. We finally project the cone $M_+(T_1, T_2)$ to $\Re^{n+1}$ as follows.

$$\mathcal{C}_+(T_1, T_2) = \text{Diagonals of matrices in } M_+(T_1, T_2)$$

These resulting cones $\{\mathcal{C}_+(T_1, T_2)\}$ satisfy

$$\mathcal{K}_I \subseteq \mathcal{C}_+(T_1, T_2) \subseteq \mathcal{K}$$

where the X_{ii} in $\mathcal{C}_+(T_1, T_2)$ are interpreted as the original x_i in $\mathcal{K}$ and $\mathcal{K}_I$.

One semidefinite relaxation of the $0-1$ integer program F_I is just $\mathcal{C}_+(T_1, T_2)$ along with the normalizing constraint $X_{00} = x_0 = 1$. For optimization versions of integer programming, we simply carry over the objective function. An amazing result obtained by Lovász and Schrijver [91] is that this relaxation when applied to the vertex packing polytope (see "Aggregation") is at least as good as one obtained by adding "clique, odd hole, odd antihole and odd wheel" valid inequalities (see [61] for the definitions) to the linear programming relaxation of the set packing formulation. This illustrates the power of this approach to reveal structure and yet obtain a tractable relaxation. In particular, it also implies a polynomial-time algorithm

for the vertex packing problem on perfect graphs (cf. [61]). Another remarkable success that semidefinite relaxations have registered is the recent result on finding an approximately maximum weight edge cutset of a graph. This result of Goemans and Williamson [55] will be described in the next section. While the jury is still out on the efficacy of semidefinite relaxation as a general strategy for integer programming, there is little doubt that it provides an exciting new weapon in the arsenal of integer programming methodologies.

32.8 Approximation with Performance Guarantees

The relaxation techniques we encountered in the previous section are designed with the intent of obtaining a "good" lower (upper) bound on the optimum objective value for a minimization (maximization) problem. If in addition, we are able to construct a "good" feasible solution using a heuristic (possibly based on a relaxation technique) we can use the bound to quantify the suboptimality of the incumbent and hence the "quality of the approximation."

In the past few years, there has been significant progress in our understanding of performance guarantees for approximation of $\mathcal{NP}$-hard combinatorial optimization problems (cf. [117]). A ρ-approximate algorithm for an optimization problem is an approximation algorithm that delivers a feasible solution with objective value within a factor of ρ of optimal (think of minimization problems and $\rho \geq 1$). For some combinatorial optimization problems, it is possible to *efficiently* find solutions that are arbitrarily close to optimal even though finding the true optimal is hard. If this were true of most of the problems of interest we would be in good shape. However, the recent results of Arora et al. [5] indicate exactly the opposite conclusion.

A PTAS or polynomial-time approximation scheme for an optimization problem is a family of algorithms A_ρ, such that for each $\rho > 1$, A_ρ is a polynomial-time ρ-approximate algorithm. Despite concentrated effort spanning about two decades, the situation in the early 1990s was that for many combinatorial optimization problems, we had no PTAS and no evidence to suggest the nonexistence of such schemes either. This led Papadimitriou and Yannakakis [104] to define a new complexity class (using reductions that preserve approximate solutions) called MAXSNP and they identified several complete languages in this class. The work of Arora and colleagues completed this agenda by showing that, assuming $\mathcal{P} \neq \mathcal{NP}$, there is no PTAS for a MAXSNP-complete problem.

An implication of these theoretical developments is that for most combinatorial optimization problems, we have to be quite satisfied with performance guarantee factors $\rho \geq 1$ that are of some small fixed value. (There are problems, like the general traveling salesman problem, for which there are no ρ-approximate algorithms for any finite value of ρ—of course assuming $\mathcal{P} \neq \mathcal{NP}$). Thus one avenue of research is to go problem by problem and knock ρ down to its smallest possible value.

A different approach would be to look for other notions of "good approximations" based on probabilistic guarantees or empirical validation. A good example of the benefit to be gained from randomization is the problem of computing the volume of a convex body. Dyer and Frieze [37] have shown that this problem is $\#\mathcal{P}$-hard. Barany and Furedi [8] provide evidence that no polynomial-time deterministic approximation method with relative error less than $(cn)^{\frac{n}{2}}$, where c is a constant, is likely to exist. However, Dyer et al. [38] have designed a fully polynomial randomized approximation scheme (FPRAS) for this problem. The FPRAS of Dyer et al. [38] uses techniques from integer programming and the geometry of numbers (see Section 32.9).

LP Relaxation and Rounding

Consider the well-known problem of finding the *smallest weight vertex cover* in a graph (see "Formulations"). So we are given a graph $G(V, E)$ and a nonnegative weight $w(v)$ for each vertex $v \in V$. We want to find the smallest total weight subset of vertices S such that each edge of G has at least one end in S (this problem is known to be MAXSNP-hard). An integer programming formulation of this problem is given

by

$$\min \left\{ \sum_{v \in V} w(v)x(v) : x(u) + x(v) \geq 1, \ \forall (u, v) \in E, \quad x(v) \in \{0, 1\} \ \forall v \in V \right\}$$

To obtain the linear programming relaxation we substitute the $x(v) \in \{0, 1\}$ constraint with $x(v) \geq 0$ for each $v \in V$. Let $\mathbf{x}^*$ denote an optimal solution to this relaxation. Now let us round the fractional parts of $\mathbf{x}^*$ in the usual way, that is, values of 0.5 and up are rounded to 1 and smaller values to 0. Let $\hat{\mathbf{x}}$ be the 0–1 solution obtained. First note that $\hat{x}(v) \leq 2x^*(v)$ for each $v \in V$. Also, for each $(u, v) \in E$, since $x^*(u) + x^*(v) \geq 1$, at least one of $x(\hat{u})$ and $x(\hat{v})$ must be set to 1. Hence $\hat{\mathbf{x}}$ is the incidence vector of a vertex cover of G whose total weight is within twice the total weight of the linear programming relaxation (which is a lower bound on the weight of the optimal vertex cover). Thus we have a 2-approximate algorithm for this problem which solves a linear programming relaxation and uses rounding to obtain a feasible solution.

The deterministic rounding of the fractional solution worked quite well for the vertex cover problem. One gets a lot more power from this approach by adding in randomization to the rounding step. Raghavan and Thompson [108] proposed the following obvious randomized rounding scheme. Given a $0 - 1$ integer program, solve its linear programming relaxation to obtain an optimal $\mathbf{x}^*$. Treat the $x_j{}^* \in [0, 1]$ as probabilities, i.e., let Probability$\{x_j = 1\} = x_j{}^*$, to randomly round the fractional solution to a $0 - 1$ solution. Using Chernoff bounds on the tails of the binomial distribution, they were able to show, for specific problems, that with high probability, this scheme produces integer solutions which are close to optimal. In certain problems, this rounding method may not always produce a feasible solution. In such cases, the expected values have to be computed as conditioned on feasible solutions produced by rounding. More complex (nonlinear) randomized rounding schemes have been recently studied and have been found to be extremely effective. We will see an example of nonlinear rounding in the context of semidefinite relaxations of the max-cut problem below.

Primal Dual Approximation

The linear programming relaxation of the vertex cover problem, we saw above, is given by

$$(P_{VC}) \quad \min \left\{ \sum_{v \in V} w(v)x(v) : x(u) + x(v) \geq 1, \ \forall (u, v) \in E, \quad x(v) \geq 0 \ \forall v \in V \right\}$$

and its dual is

$$(D_{VC}) \quad \max \left\{ \sum_{(u,v) \in E} y(u, v) : \sum_{u | (u,v) \in E} y(u, v) \leq w(v), \ \forall v \in V, \quad y(u, v) \geq 0 \ \forall (u, v) \in E \right\}$$

The primal-dual approximation approach would first obtain an optimal solution y^* to the dual problem (D_{VC}). Let $\hat{V} \subseteq V$ denote the set of vertices for which the dual constraints are tight, i.e.,

$$\hat{V} = \left\{ v \in V : \sum_{u | (u,v) \in E} y^*(u, v) = w(v) \right\}$$

The approximate vertex cover is taken to be $\hat{V}$. It follows from complementary slackness that $\hat{V}$ is a vertex cover. Using the fact that each edge (u, v) is in the star of at most two vertices (u and v), it also follows that $\hat{V}$ is a 2-approximate solution to the minimum weight vertex cover problem.

In general, the primal-dual approximation strategy is to use a dual solution, to the linear programming relaxation, along with complementary slackness conditions as a heuristic to generate an integer (primal) feasible solution which for many problems turns out to be a good approximation of the optimal solution to the original integer program.

It is a remarkable property of the vertex covering (and packing) problem that all extreme points of the linear programming relaxation have values 0, $\frac{1}{2}$ or 1 [94]. It follows that the deterministic rounding of the linear programming solution to (P_{VC}) constructs exactly the same approximate vertex cover as the primal-dual scheme described above. However, this is not true in general.

Semidefinite Relaxation and Rounding

The idea of using semidefinite programming to approximately solve combinatorial optimization problems appears to have originated in the work of Lovász [90] on the Shannon capacity of graphs. Grötschel, Lovász and Schrijver [61] later used the same technique to compute a maximum stable set of vertices in perfect graphs via the ellipsoid method. As we saw in "Semidefinite Relaxation," Lovasz and Schrijver [91] have devised a general technique of semidefinite relaxations for general $0-1$ integer linear programs. We will present a lovely application of this methodology to approximate the maximum weight cut of a graph (the maximum sum of weights of edges connecting across all strict partitions of the vertex set). This application of semidefinite relaxation for approximating MAXCUT is due to Goemans and Williamson [55].

We begin with a quadratic Boolean formulation of MAXCUT

$$\max \left\{ \frac{1}{2} \sum_{(u,v)\in E} w(u,v)(1 - x(u)x(v)) : \quad x(v) \in \{-1, 1\} \ \forall \ v \in V \right\}$$

where $G(V, E)$ is the graph and $w(u, v)$ is the nonnegative weight on edge (u, v). Any $\{-1, 1\}$ vector of $\mathbf{x}$ values provides a bipartition of the vertex set of G. The expression $(1 - x(u)x(v))$ evaluates to 0 if u and v are on the same side of the bipartition and 2 otherwise. Thus, the optimization problem does indeed represent exactly the MAXCUT problem.

Next we reformulate the problem in the following way.

- We square the number of variables by allowing each $\mathbf{x}(v)$ to denote an n-vector of variables (where n is the number of vertices of the graph).
- The quadratic term $x(u)x(v)$ is replaced by $\mathbf{x}(u) \cdot \mathbf{x}(v)$, which is the inner product of the vectors.
- Instead of the $\{-1, 1\}$ restriction on the $x(v)$, we use the Euclidean normalization $\|\mathbf{x}(v)\| = 1$ on the $\mathbf{x}(v)$.

So we now have a problem

$$\max \left\{ \frac{1}{2} \sum_{(u,v)\in E} w(u,v)(1 - \mathbf{x}(u) \cdot \mathbf{x}(v)) : \quad \|\mathbf{x}(v)\| = 1 \ \forall \ v \in V \right\}$$

which is a relaxation of the MAXCUT problem (note that if we force only the first component of the $\mathbf{x}(v)$'s to have nonzero value, we would just have the old formulation as a special case).

The final step is in noting that this reformulation is nothing but a semidefinite program. To see this we introduce $n \times n$ Gram matrix Y of the unit vectors $\mathbf{x}(v)$. So $Y = X^T X$ where $X = (\mathbf{x}(v) : \ v \in V)$. So the relaxation of MAXCUT can now be stated as a semidefinite program.

$$\max \left\{ \frac{1}{2} \sum_{(u,v)\in E} w(u,v)(1 - Y_{(u,v)}) : \quad Y \succeq 0, \quad Y_{(v,v)} = 1 \ \forall \ v \in V \right\}$$

Note that we are able to solve such semidefinite programs to an additive error ϵ in time polynomial in the input length and $\log \frac{1}{\epsilon}$ using either the ellipsoid method or interior point method (see [3] and Chapters 31 and 34 of this *Handbook*).

Let $\mathbf{x}^*$ denote the near optimal solution to the semidefinite programming relaxation of MAXCUT (convince yourself that $\mathbf{x}^*$ can be reconstructed from an optimal Y^* solution). Now we encounter the final trick of Goemans and Williamson. The approximate maximum weight cut is extracted from $\mathbf{x}^*$ by randomized rounding. We simply pick a random hyperplane H passing through the origin. All the $v \in V$ lying to one side of H get assigned to one side of the cut and the rest to the other. Goemans and Williamson observed the following inequality.

LEMMA 32.1 For $\mathbf{x}_1$ and $\mathbf{x}_2$, two random n-vectors of unit norm, let $x(1)$ and $x(2)$ be ± 1 values with opposing sign if H separates the two vectors and with same signs otherwise. Then $\tilde{E}(1 - \mathbf{x}_1{}^T \mathbf{x}_2) \leq 1.1393 \cdot \tilde{E}(1 - x(1)x(2))$ where $\tilde{E}$ denotes the expected value.

By linearity of expectation, the lemma implies that the expected value of the cut produced by the rounding is at least 0.878 times the expected value of the semidefinite program. Using standard conditional probability techniques for derandomizing, Goemans and Williamson show that a deterministic polynomial-time approximation algorithm with the same margin of approximation can be realized. Hence we have a cut with value at least 0.878 of the maximum value.

32.9 Geometry of Numbers and Integer Programming

Given an integer program with a fixed number of variables (k) we seek a polynomial-time algorithm for solving them. Note that the complexity is allowed to be exponential in k, which is independent of the input length. Clearly if the integer program has all (0, 1) integer variables this is a trivial task, since complete enumeration works. However if we are given an "unbounded" integer program to begin with, the problem is no longer trivial.

Lattices, Short Vectors and Reduced Bases

Euclidean **lattices** are a simple generalization of the regular integer lattice $\mathcal{Z}^n$. A (point) *lattice* is specified by $\{\mathbf{b}_1, \cdots, \mathbf{b}_n\}$ a *basis* (where $\mathbf{b}_i$ are linearly independent n-dimensional rational vectors). The lattice L is given by

$$L = \left\{ \mathbf{x} : \mathbf{x} = \sum_{i=1}^{n} z_i \mathbf{b}_i; \quad z_i \in Z \ \forall i \right\}$$

$$B = \left(\mathbf{b}_1 \vdots \mathbf{b}_2 \vdots \cdots \vdots \mathbf{b}_n \right) \text{ a basis matrix of } L$$

THEOREM 32.9 *$|det\, B|$ is an invariant property of the lattice L (i.e., for every basis matrix B_i of L we have invariant $d(L) = |det\, B_i|$).*

Note that

$$d(L) = \Pi_{i=1}^{n} |\mathbf{b}_i| \text{ where } |\mathbf{b}_i| \text{ denotes the Euclidean length of } \mathbf{b}_i$$

if and only if the basis vectors $\{\mathbf{b}_i\}$ are mutually orthogonal. A "sufficiently orthogonal" basis B (called a **reduced basis**) is one that satisfies a weaker relation

$$d(L) \leq c_n \Pi_{i=1}^{n} \left| \mathbf{b_i} \right| \text{ where } c_n \text{ is a constant that depends only on } n$$

One important use (from our perspective) of a reduced basis is that one of the basis vectors has to be "short." Note that there is substantive evidence that finding the shortest lattice vector is $\mathcal{NP}$-hard [61].

Minkowski proved that in every lattice L there exists a vector of length no larger than $c\sqrt{n}\sqrt[n]{d(L)}$ (with c no larger than 0.32). This follows from the celebrated Minkowski's Convex Body Theorem which forms the centerpiece of the geometry of numbers [19].

THEOREM 32.10 **(Minkowski's Convex Body Theorem)** *If $K \subseteq \Re^n$ is a convex body that is centrally symmetric with respect to the origin, and $L \subseteq \Re^n$ is a lattice such that $vol(K) \geq 2^n d(L)$ then K contains a lattice point different from the origin.*

However, no one has been successful thus far in designing an efficient algorithm (polynomial-time) for constructing the short lattice vector guaranteed by Minkowski. This is where the concept of a reduced basis comes to the rescue. We are able to construct a reduced basis in polynomial time and extract a short vector from it. To illustrate the concepts we will now discuss an algorithm due to Gauss (cf. [6]) that proves the theorem for the special case of planar lattices ($n = 2$). It is called the 60^o algorithm because it produces a reduced basis $\{\mathbf{b_1}, \mathbf{b_2}\}$ such that the acute angle between the two basis vectors is at least 60°.

Procedure: 60^o **Algorithm**

Input: `Basis vectors` $\mathbf{b}_1$ `and` $\mathbf{b}_2$ `with` $|\mathbf{b}_1| \geq |\mathbf{b}_2|$.

Output: `A reduced basis` $\{\mathbf{b}_1, \mathbf{b}_2\}$ `with at least` 60° `angle between the basis vectors.`

0. `repeat until` $|\mathbf{b}_1| < |\mathbf{b}_2|$
1. `swap` $\mathbf{b}_1$ `and` $\mathbf{b}_2$
2. $\mathbf{b}_2 \leftarrow (\mathbf{b}_2 - m\mathbf{b}_1)$ `and` $m = \left[\frac{\mathbf{b}_2^T \mathbf{b}_1}{\mathbf{b}_1^T \mathbf{b}_1}\right] \in Z$.
   `Here` $[\alpha]$ `denotes the integer nearest to` α.
3. `end`

Remarks:

(i) In each iteration the projection of $(\mathbf{b}_2 - m\mathbf{b}_1)$ onto the direction of $\mathbf{b}_1$ is of length atmost $|\mathbf{b}_1|/2$.

(ii) When the algorithm stops $\mathbf{b}_2$ must lie in one of the shaded areas at the top or at the bottom of Fig. 32.4 (since $|\mathbf{b}_2| > |\mathbf{b}_1|$ and since the projection of $\mathbf{b}_2$ must fall within the two vertical lines about $\mathbf{b}_1$).

(iii) The length of $\mathbf{b}_1$ strictly decreases in each iteration. Hence the algorithm is finite. That it is polynomial time takes a little more argument [82].

(iv) The short vector $\mathbf{b}_1$ produced by the algorithm satisfies $|\mathbf{b}_1| \leq (1.075)(d(L))^{\frac{1}{2}}$).

The only known polynomial-time algorithm for constructing a reduced basis in an arbitrary dimensional lattice [86] is not much more complicated than the 60° algorithm. However, the proof of polynomiality is quite technical (see [6] for an exposition).

Lattice Points in a Triangle

Consider a triangle T in the Euclidean plane defined by the inequalities

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 &\leq d_1 \\ a_{21}x_1 + a_{22}x_2 &\leq d_2 \\ a_{31}x_1 + a_{32}x_2 &\leq d_3 \end{aligned}$$

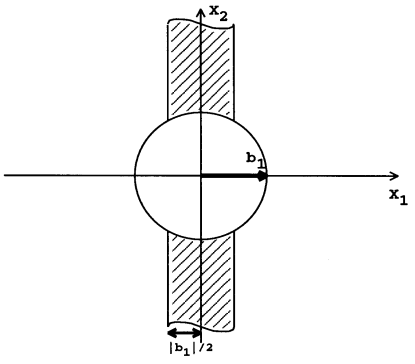


FIGURE 32.4 A reduced basis.

The problem is to check if there is a lattice point $(x_1, x_2) \in Z^2$ satisfying the three inequalities. The reader should verify that checking all possible lattice points within some bounding box of T leads to an exponential algorithm for skinny and long triangles. There are many ways of realizing a polynomial-time search algorithm [46, 73] for two-variable integer programs. Following Lenstra [88] we describe one that is a specialization of his [87] powerful algorithm for integer programming (searching for a lattice point in a polyhedron).

First we use a nonsingular linear transform τ that makes T equilateral (round). The transform sends the integer lattice Z^2 to a generic two-dimensional lattice L. Next we construct a reduced basis $\{\mathbf{b}_1, \mathbf{b}_2\}$ for L using the 60° algorithm. Let β_1 denote the length of the short vector $\mathbf{b}_1$. If l denotes the length of a side of the equilateral triangle $\tau(T)$ we can conclude that T is guaranteed to contain a lattice point if $\frac{l}{\beta_1} \geq \sqrt{6}$. Else $l < \sqrt{6}\beta_1$ and we can argue that just a few (no more than 3) of the lattice lines $\{\text{Affine Hull}\{\mathbf{b_1}\} + \mathbf{kb_2}\}_{\mathbf{k}\in\mathbf{Z}}$ can intersect T. We recurse to the lower dimensional segments (lattice lines intersecting with T) and search each segment for a lattice point. Hence this scheme provides a simple polynomial-time algorithm for searching for a lattice point in a triangle.

Lattice Points in Polyhedra

In 1979, H.W. Lenstra, Jr., announced that he had a polynomial-time algorithm for integer programming problems with a fixed number of variables. The final published version [87] of his algorithm resembles the procedure described above for the case of a triangle and a planar lattice. As we have noted before, integer programming is equivalent to searching a polyhedron specified by a system of linear inequalities for an integer lattice point. Lenstra's algorithm then proceeds as follows.

- First "round" the polyhedron using a linear transformation. This step can be executed in polynomial time (even for varying n) using any polynomial-time algorithm for linear programming.
- Find a reduced basis of the transformed lattice. This again can be done in polynomial time even for varying n.
- Use the reduced basis to conclude that a lattice point must lie inside or recurse to several lower dimensional integer programs by slicing up the polyhedron using a "long" vector of the basis to ensure that the number of slices depends only on n.

A slightly different approach based on Minkowski's convex body theorem was designed by Kannan [74] to obtain an $O(n^{\frac{9n}{2}}L)$ algorithm for integer programming (where L is the length of the input). The following two theorems are straightforward consequences of these polynomial-time algorithms for integer programming with a fixed number of variables.

THEOREM 32.11 **(Fixed Number of Constraints)** *Checking the solvability of*

$$A\mathbf{x} \leq \mathbf{b};\ \mathbf{x} \in Z^n$$

where A is $(m \times n)$ *is solvable in polynomial time if* m *is held fixed.*

THEOREM 32.12 **(Mixed Integer Programming)** *Checking the solvability of*

$$A\mathbf{x} \leq \mathbf{b};\ x_j \in Z \text{ for } j = 1, \cdots, k;\ x_j \in \Re \text{ for } j = k+1, \cdots, n$$

where A is $(m \times n)$ *is solvable in polynomial time if* $\min\{m, k\}$ *is held fixed.*

A related but more difficult question is that of counting the number of feasible solutions to an integer program or equivalently counting the number of lattice points in a polytope. Building on results described above, Barvinok [9] was able to show the following.

THEOREM 32.13 **(Counting Lattice Points)** *Counting the size of the set*

$$\left\{\mathbf{x} \in Z^k\ :\ A\mathbf{x} \leq \mathbf{b}\right\}$$

is solvable in polynomial time if k *is held fixed.*

An Application in Cryptography

In 1982, Adi Shamir [114] pointed out that Lenstra's algorithm could be used to devise a polynomial-time algorithm for cracking the basic Merkle–Hellman Cryptosystem (a knapsack-based public-key cryptosystem). In such a cryptosystem the message to be sent is a (0-1) string $\bar{\mathbf{x}} \in \{0, 1\}^n$. The message is sent as an instance of the (0,1) knapsack problem, which asks for an $\mathbf{x} \in \{0, 1\}^n$ satisfying $\Sigma_{i=1}^n a_i x_i = a_0$. The knapsack problem is an $\mathcal{NP}$-hard optimization problem (we saw in Section "Aggregation" that any integer program can be aggregated to a knapsack). However, if $\{a_i\}$ form a super-increasing sequence, i.e., $a_i > \Sigma_{j=1}^{i-1} a_j\ \forall i$, the knapsack problem can be solved in time $O(n)$ by a simple algorithm:

Procedure: **Best-Fit**

0. $i \leftarrow n$
1. **If** $a_i \leq a_0,\ \overline{x}_i \leftarrow 1,\ a_0 \leftarrow a_0 - a_i$
2. $i \leftarrow (i-1)$
3. `If` $a_0 = 0$ **stop** `and` **return** `the solution` $\mathbf{x}$
4. `If` $i = 1$ **stop** `the knapsack is infeasible.`
5. `repeat 1.`

However, an eavesdropper could solve this problem as well and hence the $\{a_i\}$ have to be encrypted. The disguise is chosen through two "secret" numbers M and U such that $M > \Sigma_{i=1}^n a_i$ and U is relatively prime to M (i.e., $gcd(U, M) = 1$). Instead of $\{a_i\}$, the sequence $\{\tilde{a}_i = Ua_i (\text{mod } M)\}_{i=1,2,\cdots,n}$ is published and $\tilde{a}_0 = Ua_0 (\text{mod } M)$ is transmitted.

Any receiver who "*knows*" U and M can easily reconvert $\{\tilde{a}_i\}$ to $\{a_i\}$ and apply the best fit algorithm to obtain the message $\bar{\mathbf{x}}$. To find a_i given $\tilde{a}_i$, U and M, he runs the Euclidean algorithm on (U, M) to obtain $1 = PU + QM$. Hence, P is the inverse multiplier of U since $PU \equiv 1 (\text{mod } M)$. Using the identity $a_i \equiv P\tilde{a}_i (\text{mod } M)\ \ \forall i = 0, 1, \ldots, n$, the intended receiver can now use the best fit method to

decode the message. An eavesdropper knows only the $\{\tilde{a}_i\}$ and is therefore supposedly unable to decrypt the message.

The objective of Shamir's cryptanalyst (code breaker) is to find a $\hat{P}$ and in $\hat{M}$ (positive integer) such that

$$(*) \qquad \begin{array}{l} \hat{a}_i \equiv \hat{P}\tilde{a}_i \bmod \hat{M} \ \forall i = 0, 1, \cdots, n \\ \{\hat{a}_i\}_{i=1,\cdots,n} \text{ is a super increasing sequence.} \\ \sum_{i=1}^{n} \hat{a}_i < \hat{M} \end{array}$$

It can be shown that for all pairs $(\hat{P}, \hat{M})$ such that $(\hat{P}/\hat{M})$ is "sufficiently"close to (P/M) will satisfy $(*)$. Using standard techniques from diophantine approximation it would be possible to guess P and M from the estimate $(\hat{P}/\hat{M})$. However, the first problem is to get the estimate of (P/M). This is where integer programming helps.

Since $a_i \equiv P\tilde{a}_i \pmod{M}$ for $i = 1, 2, \ldots, n$ we have $a_i P\tilde{a}_i - y_i M$ for some integers $y_1, y_2, \ldots, y_n$. Dividing by $\tilde{a}_i M$ we have $\left(\frac{a_i}{\tilde{a}_i M}\right) = \frac{P}{M} - \frac{y_i}{\tilde{a}_i}$ for $i = 1, 2, \ldots, n$. For small $i(1, 2, 3, \ldots, t)$ the LHS $\approx$ 0 since $\Sigma_{i=1}^{n} a_i < M$ and the $\{a_i\}_{i=1,\cdots,n}$ are super-increasing. Thus $(y_1/\tilde{a}_1), (y_2/\tilde{a}_2), \ldots, (y_t/\tilde{a}_t)$ are "close to" (P/M) and hence to each other. Therefore a natural approach to estimating (P/M) would be to solve the integer program:

$$(IP) \qquad \begin{array}{l} \text{Find } y_1, y_2, \cdots, y_t \in \mathcal{Z} \text{ such that :} \\ \underline{\epsilon_i} \leq \tilde{a}_i y_1 - \tilde{a}_1 y_i \leq \overline{\epsilon_i} \text{ for } i = 2, 3, \cdots, t \\ 0 < y_i < \tilde{a}_i \text{ for } i = 1, 2, \cdots, t \end{array}$$

This t variable integer program provides an estimate of (P/M), whence Diophantine approximation methods can be used to find $\hat{P}$ and $\hat{M}$. If we denote by d a density parameter of the instance where, $d = \frac{\log a_o}{n \log 2}$, the above scheme works correctly (with probability one as $n \to \infty$) if t is chosen to be $(\lfloor d \rfloor + 2)$ [83, 114]. Moreover, the scheme is polynomial-time in n for fixed d. The probabilistic performance is not a handicap, since as Shamir points out, "... a cryptosystem becomes useless when most of its keys can be efficiently cryptanalyzed" [114].

32.10 Prospects in Integer Programming

The current emphasis in software design for integer programming is in the development of shells (for example CPLEX [33], MINTO [111], and OSL [68]) wherein a general purpose solver like Branch & Cut is the driving engine. Problem specific code for generation of cuts and facets can be easily interfaced with the engine. We believe that this trend will eventually lead to the creation of general purpose problem solving languages for combinatorial optimization akin to AMPL [48] for linear and nonlinear programming.

A promising line of research is the development of an empirical science of algorithms for combinatorial optimization [66]. Computational testing has always been an important aspect of research on the efficiency of algorithms for integer programming. However, the standards of test designs and empirical analysis have not been uniformly applied. We believe that there will be important strides in this aspect of integer programming, and more generally of algorithms of all kinds. It may be useful to stop looking at algorithmics as purely a deductive science, and start looking for advances through repeated application of "hypothesize and test" paradigms [67], i.e., through empirical science.

The integration of logic-based methodologies and mathematical programming approaches is evidenced in the recent emergence of constraint logic programming (CLP) systems [15, 112] and logico-mathematical programming [21, 70]. In CLP, we see a structure of Prolog-like programming language in which some of the predicates are constraint predicates whose truth values are determined by the solvability of constraints in a wide range of algebraic and combinatorial settings. The solution scheme is simply a clever orchestration

of constraint solvers in these various domains and the role of conductor is played by SLD-resolution. The clean semantics of logic programming is preserved in CLP. A bonus is that the output language is symbolic and expressive. An orthogonal approach to CLP is to use constraint programming methods to solve inference problems in logic. Imbeddings of logics in mixed integer programming sets were proposed by Williams [122] and Jeroslow [70]. Efficient algorithms have been developed for inference problems in many types and fragments of logic, ranging from Boolean to predicate to belief logics [22].

A persistent theme in the integer programming approach to combinatorial optimization, as we have seen, is that the representation (formulation), of the problem, deeply affects the efficacy of the solution methodology. A proper choice of formulation can therefore make the difference between a successful solution of an optimization problem and the more common perception that the problem is insoluble and one must be satisfied with the best that heuristics can provide. Formulation of integer programs has been treated more as an art form than a science by the mathematical programming community (exceptions are Jeroslow [70] and Williams [121]). We believe that progress in representation theory can have an important influence on future of integer programming as a broad-based problem solving methodology.

32.11 Defining Terms

ρ-Approximation: An approximation method that delivers a feasible solution with objective value within a factor ρ of the optimal value of a combinatorial optimization problem.

Cutting plane: A valid inequality for an **integer polyhedron** that separates the polyhedron from a given point outside it.

Extreme point: A corner point of a polyhedron.

Fathoming: Pruning a search tree.

Integer polyhedron: A polyhedron, all of whose extreme points are integer valued.

Knapsack problem: An integer linear program with a single linear constraint other than the trivial bounds and integrality constraints on the variables.

Lattice: A point lattice generated by taking integer linear combinations of a set of basis vectors.

Linear program: Optimization of a linear function subject to linear equality and inequality constraints.

Mixed integer linear program: A linear program with the added constraint that some of the decision variables are integer valued.

Packing and covering: Given a finite collection of subsets of a finite ground set, to find an optimal subcollection that are pairwise disjoint (packing) or whose union covers the ground set (covering).

Polyhedron: The set of solutions to a finite system of linear inequalities on real-valued variables. Equivalently, the intersection of a finite number of linear half-spaces in $\Re^n$.

Reduced basis: A basis for a lattice that is nearly orthogonal.

Relaxation: An enlargement of the feasible region of an optimization problem. Typically, the relaxation is considerably easier to solve than the original optimization problem.

References

[1] Ahuja, R.K., Magnati, T.L., and Orlin, J.B., *Network Flows: Theory, Algorithms and Applications,* Prentice Hall, 1993.

[2] Akgul, M., *Topics in Relaxation and Ellipsoidal Methods, Research Notes in Mathematics,* Pitman Publishing Ltd., 1984.

[3] Alizadeh, F., Interior point methods in semidefinite programming with applications to combinatorial optimization, *SIAM J. on Optimization,* 5, 13–51, 1995.
[4] Applegate, D., Bixby, R.E., Chvátal, V., and Cook, W., Finding cuts in large TSP's, Technical Report, AT&T Bell Laboratories, Aug. 1994.
[5] Arora, S., Lund, C., Motwani, R., Sudan, M., and Szegedy, M., Proof verification and hardness of approximation problems, in *Proceedings of the* 33^{rd} *IEEE Symposium on Foundations of Computer Science,* 14–23, 1992.
[6] Bachem, A. and Kannan, R., Lattices and the Basis Reduction Algorithm, Technical Report, Computer Science, Carnegie Mellon University, 1984.
[7] Barahona, F., Jünger, M., and Reinelt, G., Experiments in quadratic $0-1$ programming, *Mathematical Programming,* 44, 127–137, 1989.
[8] Barany I. and Furedi, Z., Computing the volume is difficult, *Proceedings of 18th Symposium on Theory of Computing,* ACM Press, 442–447, 1986.
[9] Barvinok, A., Computing the volume, counting integral points in polyhedra when the dimension is fixed, in *Proceedings of the* 34^{th} *IEEE Conference on the Foundations of Computer Science (FOCS),* IEEE Press, 566–572, 1993.
[10] Benders, J.F., Partitioning procedures for solving mixed-variables programming problems, *Numerische Mathematik,* 4, 238–252, 1962.
[11] Berge, C., Farbung von Graphen deren samtliche bzw. deren ungerade Kreise starr sind (Zusammenfassung), Wissenschaftliche Zeitschrift, Martin Luther Universitat Halle-Wittenberg, Mathematisch-Naturwiseenschaftliche Reihe, 114–115, 1961.
[12] Berge, C., Sur certains hypergraphes generalisant les graphes bipartites, *Combinatorial Theory and its Applications I,* Erdos, P., Renyi, A., and Sos, V., Eds., Colloq. Math. Soc. Janos Bolyai, 4, North Holland, Amsterdam, 119–133, 1970.
[13] Berge, C., Balanced matrices, *Mathematical Programming,* 2, 19–31, 1972.
[14] Berge, C. and Las Vergnas, M., Sur un theoreme du type Konig pour hypergraphes, International Conference on Combinatorial Mathematics, *Annals of the New York Academy of Sciences,* 175, 32–40, 1970.
[15] Borning, A., Ed., *Principles and Practice of Constraint Programming,* LNCS Volume 874, Springer-Verlag, 1994.
[16] Borosh, I. and Treybig, L.B., Bounds on positive solutions of linear diophantine equations, *Proc. Amer. Math. Soc.,* 55, 299, 1976.
[17] Camion, P., Characterization of totally unimodular matrices, *Proceedings of the American Mathematical Society,* 16, 1068–1073, 1965.
[18] Cannon, T.L. and Hoffman, K.L., Large-scale zero-one linear programming on distributed workstations, *Annals of Operations Research,* 22, 181–217, 1990.
[19] Cassels, J.W.S., *An Introduction to the Geometry of Numbers,* Springer-Verlag, 1971.
[20] Chandru, V., Complexity of the supergroup approach to integer programming, Ph.D. Thesis, Operations Research Center, MIT, 1982.
[21] Chandru., V. and Hooker, J.N., Extended Horn sets in propositional logic, *JACM,* 38, 205–221, 1991.
[22] Chandru, V. and Hooker, J.N., *Optimization Methods for Logical Inference,* to be published by Wiley Interscience, 1998.
[23] Chopra, S., Gorres, E.R., and Rao, M.R., Solving Steiner tree problems by Branch and Cut, *ORSA Journal of Computing,* 3, 149–156, 1992.
[24] Chvátal, V., Edmonds polytopes and a hierarchy of combinatorial problems, *Discrete Mathematics,* 4, 305–337, 1973.
[25] Chvátal, V., On certain polytopes associated with graphs, *Journal of Combinatorial Theory,* B, 18, 138–154, 1975.

[26] Conforti, M. and Cornuejols, G., A class of logical inference problems solvable by linear programming, FOCS, 33, 670–675, 1992.
[27] Conforti, M., Cornuejols, G., and De Francesco, C., Perfect 0, ±1 matrices, preprint, Carnegie Mellon University, 1993.
[28] Conforti, M., Cornuejols, G., Kapoor, A., and Vuskovic, K., Balanced 0,±1 matrices, Parts I–II, preprints, Carnegie Mellon University, 1994.
[29] Conforti, M., Cornuejols, G., Kapoor, A., Vuskovic, K., and Rao, M.R., Balanced Matrices, in *Mathematical Programming, State of the Art 1994,* Birge, J.R. and Murty, K.G., Eds., University of Michigan, 1994.
[30] Conforti, M., Cornuejols, G., and Rao, M.R., Decomposition of balanced 0,1 matrices, Parts I–VII, preprints, Carneigie Mellon University, 1991.
[31] Conforti, M. and Rao, M.R., Testing balancedness and perfection of linear matrices, *Mathematical Programming,* 61, 1–18, 1993.
[32] Cornuejols, G. and Novick, B., Ideal 0,1 matrices, *Journal of Combinatorial Theory,* 60, 145–157, 1994.
[33] CPLEX, *Using the CPLEX callable Library and CPLEX mixed integer library,* CPLEX Optimization, 1993.
[34] Crowder, H., Johnson, E.L., and Padberg, M.W., Solving large scale 0-1 linear programming problems, *Operations Research,* 31, 803–832, 1983.
[35] Cunningham, W.H., Testing membership in matroid polyhedra, *Journal of Combinatorial Theory,* 36B, 161–188, 1984.
[36] Domich, P.D., Kannan, R., and Trotter, L.E., Hermite normal form computation using modulo determinant arithmetic, *Mathematics of Operations Research,* 12, 50–59, 1987.
[37] Dyer, M. and Frieze, A., On the complexity of computing the volume of a polyhedron, *SIAM J. on Computing,* 17, 967–974, 1988.
[38] Dyer, M., Frieze, A., and Kannan, R., A random polynomial time algorithm for approximating the volume of convex bodies, *Proceedings of 21st Symposium on Theory of Computing,* ACM Press, 375–381, 1989.
[39] Edmonds, J., Maximum matching and a polyhedron with 0-1 vertices, *Journal of Research of the National Bureau of Standards,* 69B, 125–130, 1965.
[40] Edmonds, J., Submodular functions, matroids and certain polyhedra, in *Combinatorial Structures and Their Applications,* Guy, R. et al., Eds., Gordon Breach, 69–87, 1970.
[41] Edmonds, J., Matroids and the greedy algorithm, *Mathematical Programming,* 127–136, 1971.
[42] Edmonds, J., Matroid intersection, *Annals of Discrete Mathematics,* 4, 39–49, 1979.
[43] Edmonds, J. and Giles, R., A min-max relation for submodular functions on graphs, *Annals of Discrete Mathematics,* 1, 185–204, 1977.
[44] Edmonds, J. and Johnson, E.L., Matching well solved class of integer linear programs, in *Combinatorial Structure and Their Applications,* Guy, R. et al., Eds., Gordon Breach, 1970.
[45] Farkas, Gy., A Fourier-féle mechanikai elv alkalmazásai, (in Hungarian), *Mathematikai és Természettudományi Értesitö,* 12, 457–472, 1894.
[46] Feit, S.D., A fast algorithm for the two-variable integer programming problem, *JACM,* 31, 99–113, 1984.
[47] Fonlupt, J. and Zemirline, A., A polynomial recognition algorithm for $K_4 \setminus e$-free perfect graphs, *Research Report,* University of Grenoble, 1981.
[48] Fourer, R., Gay, D.M., and Kernighan, B.W., *AMPL: A Modeling Language for Mathematical Programming,* Scientific Press, 1993.
[49] Frank, A., A weighted matroid intersection theorem, *Journal of Algorithms,* 2, 328–336, 1981.

[50] Frumkin, M.A., Polynomial-time algorithms in the theory of linear diophentine equations, in *Fundamentals of Computation Theory,* Karpinski, M., Ed., *Lecture Notes in Computer Science,* Springer-Verlag, 56, 1977.
[51] Fulkerson, D.R., The perfect graph conjecture and the pluperfect graph theorem, in *Proceedings of the Second Chapel Hill Conference on Combinatorial Mathematics and its Applications,* Bose, R.C. et al., Eds., 171–175, 1970.
[52] Fulkerson, D.R., Hoffman, A., and Oppenheim, R., On balanced matrices, *Mathematical Programming Study,* 1, 120–132, 1974.
[53] Garfinkel, R. and Nemhauser, G.L., *Integer Programming,* John Wiley & Sons, 1972.
[54] Gathen, J.V.Z. and Sieveking, M., Linear integer inequalities are NP-complete, *SIAM J. of Computing,* 1976.
[55] Goemans, M.X. and Williamson, D.P., .878 approximation algorithms MAX CUT and MAX 2SAT, in *Proceedings of ACM STOC,* 422–431, 1994.
[56] Gomory, R.E., Outline of an algorithm for integer solutions to linear programs, *Bulletin of the American Mathematical Society,* 64, 275–278, 1958.
[57] Gomory, R.E., Early integer programming, in *History of Mathematical Programming,* Lenstra, J.K. et al., Eds., North Holland, 1991.
[58] Gomory, R.E., On the relation between integer and noninteger solutions to linear programs, *Proceedings of the National Academy of Sciences of the United States of America,* 53, 260–265, 1965.
[59] Gomory, R.E. and Hu, T.C., Multi-terminal network flows, *SIAM Journal of Applied Mathematics,* 9, 551–556, 1961.
[60] Grötschel, M., Lovász L., and Schrijver, A., The ellipsoid method and its consequences in combinatorial optimization, *Combinatorica,* 1, 169–197, 1982.
[61] Grötschel, M., Lovász, L., and Schrijver, A., *Geometric Algorithms and Combinatorial Optimization,* Springer-Verlag, 1988.
[62] Held, M., Wolfe, P., and Crowder, H.P., Validation of subgradient optimization, *Math. Programming,* 6, 62–88, 1974.
[63] Held, M. and Karp, R.M., The travelling-salesman problem and minimum spanning trees, *Operations Research,* 18, 1138–1162, Part II, 1970. *Math. Programming,* 1, 6–25, 1971.
[64] Hoffman, A.J. and Kruskal, J.K., Integral boundary points of convex polyhedra, *Linear Inequalities and Related Systems,* Kuhn, H.W. and Tucker, A.W., Eds., Princeton University Press, 1, 223–246, 1956.
[65] Hooker, J.N., Resolution and the integrality of satisfiability polytopes, preprint, GSIA, Carnegie Mellon University, 1992.
[66] Hooker, J.N., Towards and empirical science of algorithms, *Operations Research,* 1993.
[67] Hooker, J.N. and Vinay, V., Branching rules for satisfiability, in *Automated Reasoning,* 1995.
[68] IBM, *Optimization Subroutine Library—Guide and Reference (Release 2),* 3rd ed., 1991.
[69] Jeroslow, R.G., There cannot be any algorithm for integer programming with quadratic constraints, *Operations Research,* 21, 221–224, 1973.
[70] Jeroslow, R.G., *Logic-Based Decision Support: Mixed Integer Model Formulation,* Annals of Discrete Mathematics, Vol. 40, North Holland, 1989.
[71] Jeroslow, R.G. and Lowe, J.K., Modeling with integer variables, *Mathematical Programming Studies,* 22, 167–184, 1984.
[72] Jünger, M., Reinelt, G., and Thienel, S., Practical problem solving with cutting plane algorithms, in *Combinatorial Optimization: Papers from the DIMACS Special Year,* Cook, W., Lovász, L., and Seymour, P., Eds., Series in Discrete Mathematics and Theoretical Computer Science, Vol. 20, AMS, 111–152, 1995.
[73] Kannan, R., A polynomial algorithm for the two-variable integer programming problem, *JACM,* 27, 1980.

[74] Kannan, R., Minkowski's convex body theorem and integer programming, *Mathematics of Operations Research,* 12, 415–440, 1987.
[75] Kannan, R. and Bachem, A., Polynomial algorithms for computing the Smith and Hermite normal forms of an integer matrix, *SIAM J. of Computing,* 8, 1979.
[76] Kannan, R. and Monma, C.L., On the computational complexity of integer programming problems, in *Lecture Notes in Economics and Mathematical Systems 157,* Henn, R., Korte, B., and Oettle, W., Eds., Springer-Verlag, 1978.
[77] Karmarkar, N.K., A new polynomial-time algorithm for linear programming, *Combinatorica,* 4, 373–395, 1984.
[78] Karmarkar, N.K., An interior-point approach to NP-complete problems—Part I, in *Contemporary Mathematics,* Vol. 114, 297–308, 1990.
[79] Karp, R., Reducibilities among combinatorial problems, in *Complexity of Computer Computations,* Miller, R.E. and Thatcher, J.W., Eds., Plenum Press, 85–103, 1972.
[80] Karp, R.M. and Papadimitriou, C.H., On linear characterizations of combinatorial optimization problems, *SIAM Journal on Computing,* 11, 620–632, 1982.
[81] Khachiyan, L.G., A polynomial algorithm in linear programming, *Doklady Akademiia Nauk SSSR,* 244(5), 1093–1096, 1979, translated into English in *Soviet Mathematics Doklady,* 20(1), 191–194, 1979.
[82] Lagarias, J.C., Worst-case complexity bounds for algorithms in the theory of integral quadratic forms, *Journal of Algorithms,* 1, 142–186, 1980.
[83] Lagarias, J.C., Knapsack public key cryptosystems and diophantine approximation, *Advances in Cryptology,* Proceedings of CRYPTO 83, Plenum Press, 3–23, 1983.
[84] Lawler, E.L., Matroid intersection algorithms, *Mathematical Programming,* 9, 31–56, 1975.
[85] Lehman, A., On the width-length inequality, mimeographic notes, 1965, *Mathematical Programming,* 17, 403–417, 1979.
[86] Lenstra, A.K., Lenstra, Jr., H.W., and Lovász, L., Factoring Polynomials with Rational Coefficients, Report 82–05, *University of Amsterdam,* 1982.
[87] Lenstra, Jr., H.W., Integer programming with a fixed number of variables, *Mathematics of Operations Research,* 8, 538–548, 1983.
[88] Lenstra, Jr., H.W., Integer programming and cryptography, *The Mathematical Intelligencer,* Vol. 6, 1984.
[89] Lovász, L., Normal hypergraphs and the perfect graph conjecture, *Discrete Mathematics,* 2, 253–267, 1972.
[90] Lovász, L., On the Shannon capacity of a graph, *IEEE Transactions on Information Theory,* 25, 1–7, 1979.
[91] Lovász, L. and Schrijver, A., Cones of matrices and set functions, *SIAM Journal on Optimization,* 1, 166–190, 1991.
[92] Martin, R.K., Using separation algorithms to generate mixed integer model reformulations, *Operations Research Letters,* 10, 119–128, 1991.
[93] McDiarmid, C.J.H., Rado's theorem for polymatroids, *Proceedings of the Cambridge Philosophical Society,* 78, 263–281, 1975.
[94] Nemhauser, G.L. and Trotter, Jr., L.E., Properties of vertex packing and independence system polyhedra, *Mathematical Programming,* 6, 48–61, 1974.
[95] Nemhauser, G.L. and Wolsey, L.A., *Integer and Combinatorial Optimization,* John Wiley & Sons, 1988.
[96] Padberg, M.W., Equivalent knapsack-type formulations of bounded integer linear programs: an alternative approach, *Naval Research Logistics Quarterly,* 19, 699–708, 1972.
[97] Padberg, M.W., Perfect zero-one matrices, *Mathematical Programming,* 6, 180–196, 1974.
[98] Padberg, M.W., Covering, packing and knapsack problems, *Annals of Discrete Mathematics,* 4, 265–287, 1979.

[99] Padberg, M.W., Lehman's forbidden minor characterization of ideal 0,1 matrices, *Discrete Mathematics,* 111, 409–420, 1993.
[100] Padberg, M.W. and Rao, M.R., The Russian method for linear inequalities, Part III, Bounded integer programming, Preprint, New York University, 1981.
[101] Padberg, M.W. and Rao, M.R., Odd minimum cut-sets and *b*-matching, *Mathematics of Operations Research,* 7, 67–80, 1982.
[102] Padberg, M.W. and Rinaldi, G., A branch and cut algorithm for the resolution of large scale symmetric travelling salesman problems, *SIAM Review,* 33, 60–100, 1991.
[103] Papadimitriou, C.H. and Steiglitz, K., *Combinatorial Optimization: Algorithms and Complexity,* Prentice Hall, 1982.
[104] Papadimitriou, C.H. and Yannakakis, M., Optimization, approximation, and complexity classes, in *Journal of Computer and Systems Sciences,* 43, 425–440, 1991.
[105] Parker, G. and Rardin, R.L., *Discrete Optimization,* John Wiley & Sons, 1988.
[106] Picard, J.C. and Ratliff, H.D., Minimum cuts and related problems, *Networks,* 5, 357–370, 1975.
[107] Pulleyblank, W.R., Polyhedral combinatorics, in *Handbooks in Operations Research and Management Science (Volume 1: Optimization),* Nemhauser, G.L, Rinnoy Kan, A.H.G., and Todd, M.J., Eds., North Holland, 371–446, 1989.
[108] Raghavan, P. and Thompson, C.D., Randomized rounding: a technique for provably good algorithms and algorithmic proofs, *Combinatorica,* 7, 365–374.
[109] Rhys, J.M.W., A selection problem of shared fixed costs and network flows, *Management Science,* 17, 200–207, 1970.
[110] Sahni, S., Computationally related problems, *SIAM J. of Computing,* 3, 1974.
[111] Savelsbergh, M.W.P., Sigosmondi, G.S., and Nemhauser, G.L., MINTO, a Mixed INTeger Optimizer, *Operations Research Letters,* 15, 47–58, 1994.
[112] Saraswat, V. and Van Hentenryck, P., Eds., *Principles and Practice of Constraint Programming,* MIT Press, 1995.
[113] Schrijver, A., *Theory of Linear and Integer Programming,* John Wiley & Sons, 1986.
[114] Shamir, A., A polynomial-time algorithm for breaking the basic Merkle-Hellman cryptosystem, *Proceedings of the Symposium on the Foundations of Computer Science,* IEEE Press, 1982.
[115] Shapiro, J.F., A survey of Lagrangean techniques for discrete optimization, *Annals of Discrete Mathematics,* 5, 113–138, 1979.
[116] Seymour, P., Decompositions of regular matroids, *Journal of Combinatorial Theory,* B, 28, 305–359, 1980.
[117] Shmoys, D.B., Computing near-optimal solutions to combinatorial optimization problems, in *Combinatorial Optimization: Papers from the DIMACS Special Year,* Cook, W., Lovász, L., and Seymour, P., Eds., Series in Discrete Mathematics and Theoretical Computer Science, Vol. 20, AMS, 355–398, 1995.
[118] Shor, N.Z., Convergence rate of the gradient descent method with dilation of the space, *Cybernetics,* 6, 1970.
[119] Truemper, K., Alpha-balanced graphs and matrices and GF(3)-representability of matroids, *Journal of Combinatorial Theory,* B, 55, 302–335, 1992.
[120] Weyl, H., Elemetere Theorie der konvexen polyerer, *Comm. Math. Helv.,* 1, 3–18, 1935. (English translation in *Annals of Mathematics Studies,* 24, Princeton, 1950).
[121] Williams, H.P., Experiments in the formulation of integer programming problems, *Mathematical Programming Study,* 2, 1974.
[122] Williams, H.P., Linear and integer programming applied to the propositional calculus, *International Journal of Systems Research and Information Science,* 2, 81–100, 1987.
[123] Yannakakis, M., Expressing Combinatorial optimization problems by linear programs, in *Proceedings of ACM Symposium of Theory of Computing,* 223–228, 1988.

[124] Ziegler, M., *Convex Polytopes,* Springer-Verlag, 1995.

Further Information

Research publications in integer programming are dispersed over a large range of journals. The following is a partial list which emphasize the algorithmic aspects: *Mathematical Programming, Mathematics of Operations Research, Operations Research, Discrete Mathematics, Discrete Applied Mathematics, Journal of Combinatorial Theory (Series B), INFORMS Journal on Computing, Operations Research Letters, SIAM Journal on Computing, SIAM Journal on Discrete Mathematics, Journal of Algorithms, Algorithmica, Combinatorica.*

Integer programming professionals frequently use the following newsletters to communicate with each other:

- *INFORMS Today* (earlier *OR/MS Today*) published by The Institute for Operations Research and Management Science (INFORMS).
- *INFORMS CSTS Newsletter* published by the INFORMS computer science technical section.
- *Optima* published by the Mathematical Programming Society.

The International Symposium on Mathematical Programming (ISMP) is held once every three years and is sponsored by the Mathematical Programming Society. The most recent ISMP was held in August 1997 in Lausanne, Switzerland. A conference on Integer Programming and Combinatorial Optimization (IPCO) is held on years when the symposium is not held. Some important results in integer programming are also announced in the general conferences on algorithms and complexity (for example, SODA (SIAM), STOC (ACM), and FOCS (IEEE)). The annual meeting of the Computer Science Technical Section (CSTS) of the INFORMS held each January (partial proceedings published by Kluwer Press) is an important source for recent results in the computational aspects of integer programming.

33

Convex Optimization

Stephen A. Vavasis
Cornell University

33.1 Introduction

Nonlinear constrained **optimization** refers to the problem of minimizing $f(\mathbf{x})$ subject to $\mathbf{x} \in D$, where D is a subset of $\mathbf{R}^n$ and f is a continuous function from D to $\mathbf{R}$. Thus, an input instance is a specification of D, called the **feasible set,** and f, called the **objective function.** The output from an optimization algorithm is either a point $\mathbf{x}^* \in D$, called an **optimizer** or **global optimizer,** such that $f(\mathbf{x}^*) \leq f(\mathbf{x})$ for all $\mathbf{x} \in D$, or else a statement (preferably accompanied by a certificate) that no such $\mathbf{x}^*$ exists. The terms "**minimizer**" and "global minimizer" are also used.

Stated in this manner, nonlinear optimization encompasses a huge range of scientific and engineering problems. In fact, this statement of the problem is too general: there are undecidable problems that can naturally fit into the framework of the last paragraph. Thus, most optimization algorithms are limited to some subclass of the general case. Furthermore, most algorithms compute something easier to find than a true global minimizer, like a local minimizer.

This chapter focuses on **convex programming,** the subclass of nonlinear optimization in which the set D is convex and the function f is also convex. The term "convex" is defined in the next section. Convex programming is interesting for three reasons: (a) convex problems arise in many applications, some of which are described in subsequent sections, (b) mathematicians have developed a rich body of theory on the topic of convexity, and (c) there are powerful algorithms for efficiently solving convex problems, which is the topic of most of this chapter. The development of "**self-concordant barrier functions**" by Nesterov and Nemirovskii [37] has led to much new research in the field. The purpose of this chapter is to describe

some of the fundamentals of convex optimization and sketch some of the recent developments based on self-concordance. At the end of the chapter, some issues concerning general (nonconvex) nonlinear optimization are raised. This chapter does not cover "low dimensional" convex programming in which the number of unknowns is very small, e.g., less than 10. Such problems arise in computational geometry and are covered in that chapter.

The remainder of this chapter is organized as follows. In the next section we cover general definitions and principles of convex optimization. In Section 33.3 we cover the **ellipsoid method** for convex optimization. In Section 33.4 and Section 33.5 we describe an **interior-point method** for convex programming, based on self-concordant barrier functions. In Section 33.6 the interior-point method is specialized to **semidefinite programming** and in Section 33.7 further specialized to **linear programming.** In Section 33.8 we discuss some Turing-machine complexity issues for convex optimization. Finally, in Section 33.9 we make some brief remarks on applications to nonconvex optimization.

33.2 Underlying Principles

Convexity

The main theme of this chapter is convexity. A set $D \subset \mathbf{R}^n$ is said to be **convex** if, for any $\mathbf{x},\mathbf{y} \in D$, and for any $\lambda \in [0, 1]$, $\lambda\mathbf{x} + (1-\lambda)\mathbf{y} \in D$. In other words, for any $\mathbf{x},\mathbf{y} \in D$, the segment joining $\mathbf{x},\mathbf{y}$ also lies in D. Some useful properties of **convex sets** are as follows:

- A convex set is connected.
- The intersection of two convex sets is convex. In fact, an infinite intersection of convex sets is convex.
- An affine transformation applied to a convex set yields a convex set. An **affine transformation** from $\mathbf{R}^n$ to $\mathbf{R}^m$ is a mapping of the form $\mathbf{x} \mapsto A\mathbf{x} + \mathbf{b}$, where A is an $m \times n$ matrix and $\mathbf{b}$ is an m-vector.

Let D be a convex subset of $\mathbf{R}^n$. A function $f : D \to \mathbf{R}$ is said to be **convex** if for all $\mathbf{x},\mathbf{y} \in D$, and for all $\lambda \in [0, 1]$,

$$f(\lambda\mathbf{x} + (1-\lambda)\mathbf{y}) \leq \lambda f(\mathbf{x}) + (1-\lambda) f(\mathbf{y}) \,. \tag{33.1}$$

This definition may be stated geometrically as follows. If $\mathbf{x},\mathbf{y} \in D$, then the segment in $\mathbf{R}^{n+1}$ joining $(\mathbf{x}, f(\mathbf{x}))$ to $(\mathbf{y}, f(\mathbf{y}))$ lies above the graph of f. Some useful properties of **convex functions** are as follows:

- A convex function composed with an affine transformation is convex.
- The pointwise maximum of two convex functions is convex.
- If f, g are convex functions, so is $\alpha f + \beta g$ for any nonnegative scalars α and β.
- If f is a convex function on D, then the set $\{\mathbf{x} \in D : f(\mathbf{x}) \leq \alpha\}$ is convex for any choice of $\alpha \in \mathbf{R}$.

Some examples of convex functions are as follows:

- A **linear function** $f(\mathbf{x}) = \mathbf{a}^T\mathbf{x} + c$, where $\mathbf{a} \in \mathbf{R}^n$ and $c \in \mathbf{R}$ are given. (A constant function is a special case of a linear function.)
- More generally, a **positive semidefinite** quadratic function. Recall that a quadratic function has the form $f(\mathbf{x}) = \frac{1}{2}\mathbf{x}^T H\mathbf{x} + \mathbf{a}^T\mathbf{x} + c$, where H is a given $n \times n$ **symmetric matrix, a** is a given vector, and c a given scalar. Recall that an $n \times n$ real matrix H is **symmetric** if $H^T = H$, where superscript ^{T} denotes matrix transpose. This function is convex provided that H is a positive semidefinite matrix. Recall that a real symmetric matrix H is said to be **positive semidefinite** if either of the following two equivalent conditions holds:
  1. For every $\mathbf{w} \in \mathbf{R}^n$, $\mathbf{w}^T H\mathbf{w} \geq 0$.

2. Every eigenvalue of H is nonnegative.

If every eigenvalue of H is positive (or, equivalently, if $\mathbf{w}^T H\mathbf{w} > 0$ for every nonzero vector $\mathbf{w}$), we say H is **positive definite.**

- The function $f(x) = -\ln x$ defined on the positive real numbers.

Feasible sets D for optimization problems are typically defined as the set of points $\mathbf{x}$ satisfying a list of **equality constraints,** that is, **constraints** of the form $g(\mathbf{x}) = 0$ where g is a real-valued function, and **inequality constraints,** that is, constraints of the form $h(\mathbf{x}) \leq 0$, where h is a real-valued function. The entries of $\mathbf{x}$ are called the **variables** or **decision variables** or **unknowns** of the optimization problem. Some commonly occurring constraints are as follows.

- A linear equation $\mathbf{a}^T\mathbf{x} = b$, where $\mathbf{a}$ is a fixed vector in $\mathbf{R}^n$ and b is a scalar.
- A linear inequality $\mathbf{a}^T\mathbf{x} \leq b$.
- A p-norm constraint of the form $\|\mathbf{x} - \mathbf{x}_0\|_p \leq r$ where $\mathbf{x}_0 \in \mathbf{R}^n$, $p \geq 1$ and $r \geq 0$ are given.
- A constraint that X must be symmetric and positive semidefinite, where X is a matrix whose entries are variables.
  The constraint that X is symmetric amounts to $n(n-1)/2$ linear equality constraints among off-diagonal elements of X. The **semidefinite constraint** can be written in the form $-\pi(X) \leq 0$ where
  $$\pi(X) = \min_{\|\mathbf{w}\|_2=1} \mathbf{w}^T X\mathbf{w} .$$
  The function $-\pi$ is a convex function of matrices. In the case that X is symmetric, $\pi(X)$ is the minimum eigenvalue of X.

A **convex constraint** is defined to be either a linear equality or inequality constraint, or is a constraint of the form $h(\mathbf{x}) \leq \mathbf{0}$ where h is a convex function. In fact, all of the above constraints are convex constraints. Because the intersection of convex sets is convex, any arbitrary conjunction of constraints of any type chosen from the above list, or affine transformations of these constraints, yields a convex set.

A convex set defined by linear constraints (that is, constraints of the first two types in the above list) is said to be a **polyhedron.**

Finally, we come to the definition of the main topic of this chapter. A **convex optimization** or **convex programming** instance is an optimization instance in which the feasible set D is defined by convex constraints, and the objective function f is also convex.

An alternative (more general) definition is that convex programming means minimizing a convex function $f(\mathbf{x})$ over a convex set D. The distinction (compared to the definition in the last paragraph) is that a convex set D is sometimes represented by means other than convex constraints. For example, the set $D = \{(x, y) \in \mathbf{R}^2 : x \geq 0, xy \geq 1\}$ is convex even though $xy \geq 1$ is not a convex constraint. The interior-point method described below can sometimes be applied in this more general setting.

Some special cases of convex programming problems include the following:

1. **Linear programming** (LP) refers to optimization problems in which $f(\mathbf{x})$ is a linear function of $\mathbf{x}$ and D is a polyhedron, i.e., it is defined by the conjunction of linear constraints.
2. **Quadratic programming** (QP) refers to optimization problems in which $f(\mathbf{x})$ is a quadratic function and D is a polyhedron. Such problems will be convex if the matrix in the quadratic function is positive semidefinite.
3. **Semidefinite programming** (SDP) refers to optimization problems in which $f(\mathbf{x})$ is a linear function and D is defined by linear and semidefinite constraints.

The word "programming" in these contexts is not directly related to "programming" a computer in the usual sense. This terminology arises for historical reasons.

One reason to be interested in convex optimization is that these classes of problems occur in many applications. Linear programming [6] is the problem that launched optimization as a discipline of study and is widely used for scheduling and planning problems. Quadratic programming arises in applications in which the objective function involves distance or energy (which are typically quadratic functions). Quadratic programming also arises as a subproblem of more general optimization algorithms; see, e.g., [35].

Finally, semidefinite programming is more general than both linear and convex programming, and arises in a number of unexpected contexts. The recent survey [55] presents many applications in which, at first glance, the optimization problem seemingly has nothing to do with positive semidefinite matrices. One recently discovered interesting class of applications is approximation algorithms for combinatorial problems. See the chapter on approximation algorithms.

Derivatives

Derivatives play a crucial role in optimization for two related reasons: necessary conditions for optimality usually involve derivatives, and most optimization algorithms involve derivatives. Both of these points are developed in more detail below. For a function $f : \mathbf{R}^n \to \mathbf{R}$, we denote its first derivative (the **gradient**) by ∇f and its second derivative (the **Hessian** matrix) as $\nabla^2 f$. Recall that, under the assumption that f is C^2, the Hessian matrix is always symmetric.

A convex function is not necessarily differentiable: for example, the convex function $f(x) = |x|$ is not differentiable at the origin. Convex functions, however, have the special property that even when they fail to be differentiable, they possess a **subdifferential,** which is a useful generalization of derivative. Let D be a nonempty convex subset of $\mathbf{R}^n$. Let f be a convex function on a convex set D, and let $\mathbf{x}$ be a point in the interior of D. This definition can also be generalized to the case when D has no interior, i.e., its affine hull has dimension less than n. (The **affine hull** of a set X is the minimal affine set containing X. An **affine set** is a subset of $\mathbf{R}^n$ of the form $\{A\mathbf{x} + \mathbf{b} : \mathbf{x} \in \mathbf{R}^p\}$ where A is an $n \times p$ matrix and $\mathbf{b}$ is an n-vector.) The **subdifferential** of f at $\mathbf{x}$ is defined to be the set of vectors $\mathbf{v} \in \mathbf{R}^n$ such that $f(\mathbf{y}) \geq f(\mathbf{x}) + \mathbf{v}^T(\mathbf{y} - \mathbf{x})$ for all $\mathbf{y} \in D$. For example, the subdifferential of the function $f(x) = |x|$ at $x = 0$ is the closed interval $[-1, 1]$. Some useful properties of the subdifferential are

- The subdifferential is a nonempty, closed, convex set.
- When f is differentiable at a point $\mathbf{x}$ in the interior of D, the subdifferential is a singleton set whose unique member is the ordinary derivative $\nabla f(\mathbf{x})$.

An element of the subdifferential is called a **subgradient** of f.

Since a convex function is not necessarily differentiable, *a fortiori* it is not necessarily twice-differentiable. When f is twice-continuously differentiable, there is a simple characterization of convexity. Let f be a C^2 function defined on a convex set D with a nonempty interior. Then f is convex if and only if the second derivative of f, that is, its Hessian matrix $\nabla^2 f(\mathbf{x})$, is positive semidefinite for all $\mathbf{x} \in D$.

Optimality Conditions

Recall that $\mathbf{x}^* \in D$ is the optimizer if $f(\mathbf{x}^*) \leq f(\mathbf{x})$ for all $\mathbf{x} \in D$. Verifying optimality thus apparently requires global knowledge of the function f. Given a feasible point $\mathbf{x}^*$, it is desirable to be able to check whether $\mathbf{x}^*$ is an optimizer using information about D and f only in a small neighborhood of $\mathbf{x}^*$. For general optimization problems, such a local characterization is not possible because general optimization problems can have local minimizers that are not globally optimal. We define this term as follows: $\mathbf{x}^* \in D$ is a **local minimizer** of f if there exists an open set $N \subset \mathbf{R}^n$ containing $\mathbf{x}^*$ such that $f(\mathbf{x}) \geq f(\mathbf{x}^*)$ for all $\mathbf{x} \in N \cap D$.

In the case of convex optimization, local minimizers are always global minimizers; we state this as a theorem.

THEOREM 33.1 *Let f be a convex function defined on a convex domain D. Let $\mathbf{x}^* \in D$ be a local minimizer of f. Then $\mathbf{x}^*$ is a global minimizer of f.*

PROOF Suppose $\mathbf{x}^*$ is a local minimizer, and let $\mathbf{x}$ be any other feasible point. Consider the sequence of points $\mathbf{x}_1, \mathbf{x}_2, \ldots$ converging to $\mathbf{x}^*$ given by $\mathbf{x}_i = (1 - 1/i)\mathbf{x}^* + (1/i)\mathbf{x}$; clearly these points lie in D since D is convex. By the convexity of f, $f(\mathbf{x}_i) \leq (1 - 1/i) f(\mathbf{x}^*) + (1/i) f(\mathbf{x})$. On the other hand, by the local minimality of $\mathbf{x}^*$, there is an i sufficiently large such that $f(\mathbf{x}_i) \geq f(\mathbf{x}^*)$. Combining these inequalities, we have $f(\mathbf{x}^*) \leq (1 - 1/i) f(\mathbf{x}^*) + (1/i) f(\mathbf{x})$, which simplifies to $f(\mathbf{x}^*) \leq f(\mathbf{x})$.

For general optimization problems, **local optimizers** have local characterizations in terms of derivatives. Usually derivative characterizations are either necessary or sufficient for local minimality, but rarely is there a single condition that is both necessary *and* sufficient. In the case of convex optimization, there exist derivative conditions that are both necessary and sufficient for *global* optimality, provided that a certain "**constraint qualification**" holds, and provided that the functions in question are differentiable.

To set the stage, let us first consider the general nonconvex case. Given a point $\mathbf{x}^* \in D$, we will say that a nonzero vector $\mathbf{v}$ is a **feasible direction** at $\mathbf{x}^*$ if there exists a sequence of points $\mathbf{x}_1, \mathbf{x}_2, \ldots \in D$ converging to $\mathbf{x}^*$ and a sequence of positive scalars $\alpha_1, \alpha_2, \ldots$ converging to zero such that $\mathbf{x}_k - \mathbf{x}^* = \alpha_k \mathbf{v} + o(\alpha_k)$. Assuming the objective function f is differentiable at $\mathbf{x}^*$, it is easily seen that a necessary condition for local minimality is that $\nabla f(\mathbf{x}^*)^T \mathbf{v} \geq 0$ for any feasible direction $\mathbf{v}$.

In order to apply the condition in the previous paragraph, one must be able to determine the set of feasible directions at $\mathbf{x}^*$. Assume that D is defined via a sequence of equality constraints $g_1(\mathbf{x}) = \cdots = g_p(\mathbf{x}) = 0$ and inequality constraints $h_1(\mathbf{x}) \leq 0, \ldots, h_q(\mathbf{x}) \leq 0$ with $g_1, \ldots, g_p$ and $h_1, \ldots, h_q$ continuous. We say that an inequality constraint is **active** at $\mathbf{x}^*$ if it is satisfied as an equality. Clearly the inactive constraints can be ignored in determining local optimality: if $h_i(\mathbf{x}^*) < 0$, then by continuity, $h_i(\mathbf{x}) < 0$ for all $\mathbf{x}$ sufficiently close to $\mathbf{x}^*$ and hence this constraint has no effect on the problem locally. Thus, the feasible directions at $\mathbf{x}^*$ are determined by equality and active inequality constraints. Let $A \subset \{1, \ldots, q\}$ index the inequality constraints that are active at $\mathbf{x}^*$, i.e., $h_i(\mathbf{x}^*) = 0$ for all $i \in A$. For uniform terminology, we will regard equality constraints as always being active.

Assume further that all active constraints at $\mathbf{x}^*$ are C^1 in a neighborhood of $\mathbf{x}^*$. Consider replacing the active constraints locally at $\mathbf{x}^*$ by **linearizations.** For an equality constraint, say $g_i(\mathbf{x}) = 0$, we know that $g_i(\mathbf{x}^* + \mathbf{v}) = g_i(\mathbf{x}^*) + \mathbf{v}^T \nabla g_i(\mathbf{x}^*) + o(\|\mathbf{v}\|) = \mathbf{v}^T \nabla g_i(\mathbf{x}^*) + o(\|\mathbf{v}\|)$. This means that a feasible direction $\mathbf{v}$ for this one constraint in isolation must satisfy $\mathbf{v}^T \nabla g_i(\mathbf{x}^*) = 0$. Similarly a feasible direction $\mathbf{v}$ for an inequality constraint $h_i(\mathbf{x}) \leq 0$ active at $\mathbf{x}^*$ must satisfy $\mathbf{v}^T \nabla h_i(\mathbf{x}^*) \leq 0$. This leads to the following definition: a nonzero vector $\mathbf{v}$ is said to be a **linearized feasible direction** at $\mathbf{x}^*$ if $\mathbf{v}^T \nabla g_i(\mathbf{x}^*) = 0$ for all $i = 1, \ldots, p$, and $\mathbf{v}^T \nabla h_i(\mathbf{x}^*) \leq 0$ for all $i \in A$. Observe that the linearized feasible directions at $\mathbf{x}^*$ form a polyhedral cone, and assuming we have reasonably explicit representations of the constraints, membership in this cone is easy to check.

Under the assumption of differentiability, the set of feasible directions is a subset of the linearized feasible directions, but in general, it could be a proper subset. A **constraint qualification** is a condition on the constraints that guarantees that every linearized feasible direction is also a feasible direction. There are many constraint qualifications proposed in the literature. Two simple ones are as follows: (1) All the active constraints are linear, and (2) all the constraints are C^1 and their gradient vectors are linearly independent. If either (1) or (2) holds at a feasible point $\mathbf{x}^*$, then every linearized feasible direction is a feasible direction.

Assuming a constraint qualification holds, we can now state necessary conditions for local minimality, known as the **Karush–Kuhn–Tucker** (KKT) conditions. The conditions state that (1) $\mathbf{x}^*$ must be feasible and (2) for every linearized feasible direction $\mathbf{v}$ at $\mathbf{x}^*$, $\mathbf{v}^T \nabla f(\mathbf{x}^*) \geq 0$.

Using Farkas' lemma (a result that is equivalent to linear programming duality; see, e.g., [38]), we can reformulate condition (2) as the following equivalent statement: The gradient $\nabla f(\mathbf{x}^*)$ can be written in the form

$$\nabla f\left(\mathbf{x}^*\right) = \sum_{i=1}^{p} \lambda_i \nabla g_i\left(\mathbf{x}^*\right) - \sum_{i \in A} \mu_i \nabla h_i\left(\mathbf{x}^*\right) \tag{33.2}$$

where $\mu_i \geq 0$ for each $i \in A$. The variables λ_i, μ_i are called **multipliers.** In the case of equality

constraints only, we have only the first summation present on the right-hand side. In this special case the KKT conditions are called the "Lagrange multiplier" conditions.

The third (and final) way to write the KKT conditions is as follows. We introduce multipliers μ_i for constraints in $\{1, \ldots, q\} - A$ and force them to be zero. This allows us to write the conditions without explicit reference to A. We state this as a theorem.

THEOREM 33.2 *Consider minimizing $f(\mathbf{x})$ over a domain D, where*

$$D = \left\{ \mathbf{x} \in \mathbf{R}^n : g_1(\mathbf{x}) = \cdots = g_p(\mathbf{x}) = 0; h_1(\mathbf{x}) \leq 0, \ldots, h_q(\mathbf{x}) \leq 0 \right\} . \tag{33.3}$$

Let $\mathbf{x}^* \in D$ be a local minimizer of f, and assume that a constraint qualification holds at $\mathbf{x}^*$. Assume that $g_1, \ldots, g_p, h_1, \ldots, h_q$, f are all C^1 in a neighborhood of $\mathbf{x}^*$. Then there exist parameters $\lambda_1, \ldots, \lambda_p$ and $\mu_1, \ldots, \mu_q$ satisfying

$$\begin{aligned} \nabla f\left(\mathbf{x}^*\right) &= \sum_{i=1}^{p} \lambda_i \nabla g_i\left(\mathbf{x}^*\right) - \sum_{i=1}^{q} \mu_i \nabla h_i\left(\mathbf{x}^*\right) , \\ h_i\left(\mathbf{x}^*\right) \mu_i &= 0 \quad \textit{for } i = 1, \ldots, q , \\ \mu_i &\geq 0 \quad \textit{for } i = 1, \ldots, q , \\ g_i(\mathbf{x}^*) &= 0 \quad \textit{for } i = 1, \ldots, p , \\ h_i(\mathbf{x}^*) &\leq 0 \quad \textit{for } i = 1, \ldots, q . \end{aligned} \tag{33.4}$$

The first condition is the same as before, except we have inserted dummy multipliers for inactive inequality constraints. The second condition expresses the fact that μ_i must be zero for every inactive constraint. This condition is known as the **complementarity** condition. The last two relations express feasibility of $\mathbf{x}^*$.

A point that satisfies all of these conditions is said to be a **KKT point** or **stationary point.** In general, the KKT conditions are not sufficient for local minimality. To see this, consider the case of an unconstrained problem, in which case the KKT conditions reduce to the single requirement that $\nabla f(\mathbf{x}^*) = \mathbf{0}$. This condition is satisfied at local maxima as well as local minima! To address this shortcoming, the KKT conditions are often accompanied by second-order conditions about the second derivative of the objective function. We omit second-order conditions from this discussion.

Let us now specialize this discussion to convex programming, where everything becomes simpler. Assuming a constraint qualification, the KKT conditions are necessary for local and hence global minimality. It turns out that they are also *sufficient* for local (and hence global) minimality, provided that the constraints are convex and differentiable. (The differentiability assumption can be dropped because one can use subgradients in place of derivatives in the KKT conditions.) Thus, the second-order optimality conditions are extraneous in the case of convex programming.

Unfortunately, the constraint qualification cannot be discarded for convex programming. For example, the convex optimization problem of minimizing $x + y$ subject to $x^2 + y^2 \leq 1$ and $x \geq 1$ has a single feasible point $(1, 0)$ and hence trivially this point is optimal. However, $(1, 0)$ is not a KKT point; the failure of the KKT conditions at $(1, 0)$ arises from the fact that the linearized feasible directions are not feasible directions. (There are no feasible directions!) In the case of convex programming, there is a specialized constraint qualification called the "**Slater condition,**" which applies to **feasible regions** defined by convex constraints. The condition is: there exists a point $\mathbf{x} \in D$ such that all the nonlinear constraints (i.e., all the h_i that are not linear functions) are inactive at $\mathbf{x}$. Note that the existence of a *single* such point means that the linearized feasible directions are the same as the feasible directions at *every* point in the domain.

33.3 The Ellipsoid Algorithm

There are many general purpose algorithms in the literature for nonlinear optimization. There is nothing to prevent such algorithms from being applied to convex problems. In this case, convergence to a KKT point means convergence to a global minimizer, so convexity already buys us something even with an algorithm that knows nothing about convexity. Rather than covering general nonlinear optimization algorithms, we will cover two specific algorithms for convex programming in this chapter: the ellipsoid method and an interior-point method. The ellipsoid method is easier to understand and is easier to set up than interior-point methods, but it can inefficient in practice. Interior-point methods have come to be the preferred approach for convex programming.

The ellipsoid method is due to Yudin and Nemirovskii [68], although some of the ideas appeared earlier in Shor [46]. The treatment that follows is based on a section of the book by Nemirovskii and Yudin [36]. In that book, the ellipsoid method is called the "modified method of centers of gravity." It is a variant of a method called "method of centers of gravity" also described in [36]. The method of centers of gravity requires at each step the computation of the centroid of a region defined by convex constraints. The method of centers of gravity is optimal in an information-theoretic sense (i.e., compared to other convex optimization methods, it uses the fewest number of function evaluations to compute an approximate optimizer within a prespecified tolerance of optimal) but is computationally intractable because there is no known efficient way to compute the required center of gravity.

For this section we are considering the convex programming problem

$$\begin{array}{ll} \text{minimize} & f(\mathbf{x}) \\ \text{subject to} & h_1(\mathbf{x}) \leq 0 \\ & \vdots \\ & h_q(\mathbf{x}) \leq 0 \end{array} \tag{33.5}$$

where $f, h_1, \ldots, h_q$ are convex. In other words, any linear equality constraints have been removed by solving for some of the variables and reducing to a lower dimension.

Recall that an ellipsoid is a subset of $\mathbf{R}^n$ defined as follows. Let B be a symmetric positive definite $n \times n$ matrix. Let $\mathbf{c}$ be an n-vector. Then the set

$$E = \left\{ \mathbf{x} \in \mathbf{R}^n : (\mathbf{x} - \mathbf{c})^T B^{-1} (\mathbf{x} - \mathbf{c}) \leq 1 \right\} \tag{33.6}$$

is said to be an **ellipsoid.** Point $\mathbf{c}$ is said to be the **center** of the ellipsoid. An ellipsoid is easily seen to be a convex set because it is the image of the unit ball $\{\mathbf{x} \in \mathbf{R}^n : \|\mathbf{x}\| \leq 1\}$ under an affine transformation. From now on, $\|\cdot\|$ denotes the ordinary Euclidean norm unless otherwise noted.

In order to explain the ellipsoid method, we need the following preliminary result.

THEOREM 33.3 *Let D be a convex subset of $\mathbf{R}^n$, and let f be a convex function from $\mathbf{R}^n$ to $\mathbf{R}$. Suppose f is differentiable at a point $\mathbf{x} \in D$. Let $\mathbf{y} \in D$ be another point. If $f(\mathbf{y}) \leq f(\mathbf{x})$, then*

$$\nabla f(\mathbf{x})^T (\mathbf{y} - \mathbf{x}) \leq 0 \,. \tag{33.7}$$

We omit the proof of this result, which follows from the definition of derivative and standard convexity arguments. In the case when f is not differentiable at $\mathbf{x}$, a similar result holds, where in place $\nabla f(\mathbf{x})$ we can use any subgradient of f at $\mathbf{x}$.

The ellipsoid method is initialized with an ellipsoid E_0 containing the optimal solution $\mathbf{x}^*$. There is no general-purpose method for computing E_0; it must come from some additional knowledge about the convex programming problem. In Section 33.8 we discuss the issue of determining E_0 for the special case of linear programming.

The method now constructs a sequence of ellipsoids $E_1, E_2, \ldots$ with the following two properties: each E_i contains the optimal solution $\mathbf{x}^*$, and the volume of the E_i's decreases at a fixed rate. The algorithm terminates when it determines that, for some p, E_p is sufficiently small so that its center is a good approximation to the optimizer.

The procedure for obtaining E_{i+1} from E_i is as follows. We first check whether $\mathbf{c}_i$, the center of E_i, is feasible. Case 1 is that it is infeasible. Then we select a violated constraint, say h_j such that $h_j(\mathbf{c}_i) > 0$. Let H_i be the halfspace defined by the single linear inequality

$$H_i = \left\{ \mathbf{y} : \nabla h_j(\mathbf{c}_i)^T (\mathbf{y} - \mathbf{c}_i) \leq 0 \right\} . \tag{33.8}$$

(When h_j is not differentiable at $\mathbf{c}_i$, we instead use any subgradient.) Note that the boundary of this halfspace passes through $\mathbf{c}_i$. Observe now that the entire feasible region is contained in H_i. The reason is that for any feasible point $\mathbf{x}$, we have $h_j(\mathbf{x}) \leq 0 \leq h_j(\mathbf{c}_i)$, hence $\mathbf{x} \in H_i$ by (33.7). In particular, the optimizer is in H_i.

We must also rule out the degenerate case when $\nabla h_j(\mathbf{c}_i) = \mathbf{0}$, because in this case (33.8) does not specify a halfspace. In this case, $\mathbf{c}_i$ is the global minimizer of h_j by the KKT conditions. Since $h_j(\mathbf{c}_i) > 0$ and $\mathbf{c}_i$ is a global minimizer, there does not exist an $\mathbf{x}$ such that $h_j(\mathbf{x}) \leq 0$. This means that the feasible region is empty and the problem has no solution.

Case 2 is that the center $\mathbf{c}_i$ of E_i is feasible. In this case, the ellipsoid algorithm takes H_i to be the halfspace

$$H_i = \left\{ \mathbf{y} : \nabla f\,(\mathbf{c}_i)^T \left(\mathbf{y} - \mathbf{c}_i\right) \leq 0 \right\} . \tag{33.9}$$

(When f is not differentiable at $\mathbf{c}_i$, we instead use a subgradient. When $\nabla f(\mathbf{c}_i) = \mathbf{0}$, we are done, i.e., the ellipsoid method has found the exact optimizer.) Observe that by (33.7), every point $\mathbf{x}$ with a lower objective function value than $\mathbf{c}_i$ (and in particular, the optimizer) lies in H_i.

Thus, in both cases, we compute a halfspace H_i whose boundary passes through $\mathbf{c}_i$ such that the optimizer is guaranteed to lie in H_i. By induction, the optimizer also lies in E_i. The ellipsoid algorithm now computes a new ellipsoid E_{i+1} that is a superset of $E_i \cap H_i$. By induction, we know that the optimizer must thus lie in E_{i+1}.

The formulas for E_{i+1} are as follows. Let us suppose $E_i = \{\mathbf{x} : (\mathbf{x} - \mathbf{c}_i)^T B_i^{-1} (\mathbf{x} - \mathbf{c}_i) \leq 1\}$ and $H_i = \{\mathbf{a}_i^T (\mathbf{x} - \mathbf{c}_i) \leq 0\}$ where $\mathbf{a}_i$ is a gradient/subgradient of either a constraint or the objective function. We define

$$E_{i+1} = \left\{ \mathbf{x} : \left(\mathbf{x} - \mathbf{c}_{i+1}\right)^T B_{i+1}^{-1} \left(\mathbf{x} - \mathbf{c}_{i+1}\right) \leq 1 \right\} \tag{33.10}$$

where

$$B_{i+1} = \frac{n^2}{n^2 - 1} \cdot \left(B_i - \frac{2 B_i \mathbf{a}_i \mathbf{a}_i^T B_i}{(n+1) \mathbf{a}_i^T B_i \mathbf{a}_i} \right) \tag{33.11}$$

and

$$\mathbf{c}_{i+1} = \mathbf{c}_i - \frac{B_i \mathbf{a}_i}{(n+1)\sqrt{\mathbf{a}_i^T B_i \mathbf{a}_i}} . \tag{33.12}$$

Properties of these formulas are summarized in the following theorem.

THEOREM 33.4 *With B_{i+1} defined by (33.11), $\mathbf{c}_{i+1}$ defined by (33.12), and E_{i+1} defined by (33.10), we have the following properties:*

- *Matrix B_{i+1} is symmetric and positive definite.*
- $E_i \cap H_i \subset E_{i+1}$.
- *If we let* vol *denote volume, then*

$$\frac{vol\left(E_{i+1}\right)}{vol\left(E_i\right)} = \frac{n}{n+1}\left(\frac{n^2}{n^2-1}\right)^{(n-1)/2}. \tag{33.13}$$

Each of these properties is verified by substituting definitions and then algebraically simplifying. See, e.g., [4].

Using the standard inequality $1 + x \leq e^x$, one can show that the volume decrease factor on the right-hand side of (33.13) is bounded above by $\exp(-1/(2n+2))$. Thus, the volumes of the ellipsoids form a decreasing geometric sequence.

The algorithm is terminated at an ellipsoid E_p that is judged to be sufficiently small. Once E_p is reached, the ellipsoid algorithm returns as an approximate optimizer a point $\mathbf{c}^*$, where $\mathbf{c}^*$ is the center of one of $E_0, \ldots, E_p$, and is chosen as follows. Among all of $\{\mathbf{c}_0, \ldots, \mathbf{c}_p\}$, consider the subset of feasible $\mathbf{c}_i$'s, i.e., consider the set $\{\mathbf{c}_0, \ldots, \mathbf{c}_p\} \cap D$. Choose $\mathbf{c}^*$ from this subset to have the smallest objective function value. This is the approximate minimizer. If $\{\mathbf{c}_0, \ldots, \mathbf{c}_p\} \cap D$ is empty, then one of the following is true: either E_0 was chosen incorrectly, p has not been chosen sufficiently large, or $\text{vol}(D) = 0$.

Thus, the ellipsoid method is summarized as follows.

Ellipsoid Algorithm for (33.5)
Choose E_0 centered at $\mathbf{c}_0$ so that $\mathbf{x}^* \in E_0$.
$i := 0$.
while $\text{vol}(E_i)$ too large do
  if $\mathbf{c}_i \notin D$
    choose a j such that $h_j(\mathbf{c}_i) > 0$.
    $\mathbf{a}_i := \nabla h_j(\mathbf{c}_i)$.
  else
    $\mathbf{a}_i := \nabla f(\mathbf{c}_i)$.
  end if
  define $E_{i+1}, \mathbf{c}_{i+1}$ by (33.10)–(33.12).
  $i := i + 1$.
end while
Let $\mathbf{c}^*$ minimize f over $\{\mathbf{c}_0, \ldots \mathbf{c}_i\} \cap D$.

We now analyze the relationship between the number of steps p and the degree to which $\mathbf{c}^*$ approximates the optimizer. This complexity analysis requires some additional assumptions. Let the volume of the feasible region D be v_D. We assume that v_D is a positive real number. There are two cases when this assumption fails. The first case is that $v_D = \infty$. In this case, there is no difficulty with the ellipsoid method itself, but the upcoming complexity analysis no longer holds. The assumption also fails if $v_D = 0$. This happens when the dimension of the affine hull of D is less than n, including the case when $D = \emptyset$.

Since v_D is finite and positive, there is an ellipsoid that contains D. Let us assume that in fact E_0 contains D. There is no way to detect whether E_0 contains D or that $0 < \mathbf{v}_D < \infty$ within the framework of the ellipsoid algorithm itself; instead, this must be discerned from additional information about the problem. There are special classes of convex programming problems, notably linear and quadratic programming, in which it is possible to initialize and terminate the ellipsoid algorithm without prior information (other than the specification of the problem) even when $v_D = \infty$ or $v_D = 0$.

Under the assumption that v_D is finite and positive and that E_0 contains D, let us now analyze the running time of the ellipsoid algorithm, following the analysis in [36]. Let us assume that the last iteration of the algorithm is the pth and generates an ellipsoid E_p whose volume is less than $v_D\gamma^n$, where $\gamma \in (0, 1)$ will be defined below. Define the bijective affine map $\phi : \mathbf{R}^n \to \mathbf{R}^n$ by the formula $\phi(\mathbf{x}) = (1-\gamma)\mathbf{x}^* + \gamma\mathbf{x}$. Note that this mapping shrinks $\mathbf{R}^n$ toward $\mathbf{x}^*$. Let $D' = \phi(D)$. Observe that D' contains $\mathbf{x}^*$, and by convexity, $D' \subset D$. Finally, observe that $\text{vol}(D') = v_D\gamma^n$. Since $\text{vol}(E_p) < v_D\gamma^n$, there exists at least

one point $\mathbf{y}' \in D'$ that is not in E_p. Since $\mathbf{y} \in D' \subset D \subset E_0$ and $\mathbf{y}' \notin E_p$, there must have been an iteration, say m, such that $\mathbf{y}' \notin E_m$ but $\mathbf{y}' \in E_{m-1}$. This means that $\mathbf{y}' \notin H_{m-1}$. Recall that H_{m-1} was chosen according to one of two rules depending on whether $\mathbf{c}_{m-1}$ was feasible or not. For the step under consideration, $\mathbf{c}_{m-1}$ cannot be infeasible as the following argument shows. Recall that when $\mathbf{c}_i$ is infeasible for some i, H_i contains the entire feasible region. Since $\mathbf{y}'$ is feasible and not in H_{m-1}, this means that $\mathbf{c}_{m-1}$ must have been feasible. In particular, this means that the ellipsoid method will be able to return a feasible $\mathbf{c}^*$.

Furthermore, we can use the existence of $\mathbf{c}_{m-1}$ to get bounds on the distance to optimality of $\mathbf{c}^*$. Recall that, because step $m-1$ falls into case 2, halfspace H_{m-1} contains all feasible points whose objective function value is less than $\mathbf{c}_{m-1}$. Thus, $f(\mathbf{y}') \geq f(\mathbf{c}_{m-1})$. Further, $f(\mathbf{c}_{m-1}) \geq f(\mathbf{c}^*)$. Let $\mathbf{y} = \phi^{-1}(\mathbf{y}')$, i.e., $\mathbf{y}' = (1-\gamma)\mathbf{x}^* + \gamma\mathbf{y}$. Observe that $\mathbf{y} \in D$ since $\mathbf{y}' \in D'$. By convexity, $f(\mathbf{y}') \leq (1-\gamma)f(\mathbf{x}^*) + \gamma f(\mathbf{y})$, i.e.,

$$f\left(\mathbf{y}'\right) - f\left(\mathbf{x}^*\right) \leq \gamma\left(f(\mathbf{y}) - f\left(\mathbf{x}^*\right)\right) \tag{33.14}$$

so

$$f\left(\mathbf{c}^*\right) - f\left(\mathbf{x}^*\right) \leq \gamma\left(f(\mathbf{y}) - f\left(\mathbf{x}^*\right)\right) . \tag{33.15}$$

Let us say that a feasible point $\mathbf{x}$ is an ϵ-**approximate** optimizer if

$$\frac{f(\mathbf{x}) - f\left(\mathbf{x}^*\right)}{f\left(\mathbf{x}^{\#}\right) - f\left(\mathbf{x}^*\right)} \leq \epsilon \tag{33.16}$$

where $\mathbf{x}^{\#}$ is the feasible point with the worst objective function value. Thus, assuming f is bounded, a 0-approximate optimizer is the true optimizer, and every feasible point is a 1-approximate optimizer. We see from (33.15) that the ellipsoid method computes a γ-approximate optimizer.

Recall we defined $\gamma \in (0, 1)$ as a free parameter. Thus, the ellipsoid method can return arbitrarily good approximate optima. The price paid for a better approximation is a higher running time. Let the volume of the initial ellipsoid be denoted v_0. Recall that we run the method until $\text{vol}(E_p) < v_D\gamma^n$. Recall also that the ellipsoids decrease by a multiplicative factor less than $\exp(-1/(2n+2))$ per step. Thus, we must pick p sufficiently large so that $\exp(-p/(2n+2))v_0 < v_D\gamma^n$, i.e.,

$$p > (2n+2)\left(n\ln(1/\gamma) + \ln v_0 - \ln v_D\right) . \tag{33.17}$$

Thus, we see that the running time depends logarithmically on the desired degree of accuracy. The running time also grows in the case when E_0 is much larger than D.

This raises the question: is there a case when $\ln v_0$ would have to be much larger than $\ln v_D$? The answer is no: for any convex set D satisfying $0 < \text{vol}(D) < \infty$, there exists an ellipsoid E such that $D \subset E$ and such that $\text{vol}(E) \leq n^n\text{vol}(D)$. Thus, if we had complete information about D, we could ensure that the term $\ln v_0 - \ln v_D$ in (33.17) is never more than $n \ln n$.

33.4 A Primal Interior-Point Method

Interior-point methods for linear programming were introduced by Karmarkar [25]. It was discovered later that Karmarkar's interior-point method is related to log-barrier methods introduced earlier in the literature (see, e.g., [11]) and also to an affine scaling method due to Dikin [7]. Soon after Karmarkar's method was published, several papers, e.g., Kapoor and Vaidya [24] and Ye and Tse [67] extended the method to quadratic programming. A further extension of interior-point methods to some classes of nonlinear constraints came in the work of Jarre [19, 20] and Mehrotra and Sun [32]. An extension to the general case of convex programming came later with the landmark monograph by Nesterov and Nemirovskii [37] and the introduction of self-concordant functions. In this chapter, we reverse the historical development by first presenting the general convex programming case, and then specializing to linear programming. (It

should be pointed out that lecture notes containing the main ideas of self-concordance had circulated for some years previous to publication of the monograph in 1994. In addition, many of the journal citations in this chapter have dates three or four years after the initial circulation of the result via preprints.)

Interior-point methods are generally believed to be the best approach for large-scaled structured linear programming and more general convex programming. As for complexity theory, interior-point methods represent only a slight improvement on the ellipsoid method. The difference is that interior-point methods work extremely well in practice (much better than their known worst-case bounds), whereas the running time of the ellipsoid method tends to be equal to its worst-case bound.

Interior-point methods are generally classified into three categories: path-following, potential reduction, and projective. Karmarkar's original method was a projective method, but projective methods have faded in importance since 1984. Both of the remaining two classes are used in practice, though at the time of this writing, path-following methods seem to be preferred. Because of space limitations, we cover only path-following methods in this section. Potential reduction is briefly described in the context of semidefinite programming in Section 33.6.

A second way of classifying interior-point methods is whether they are primal or primal-dual. The algorithm presented in this section is a "primal" algorithm. When we specialize to semidefinite and linear programming in upcoming sections, we present primal-dual algorithms, which are generally considered to be superior to primal algorithms.

In this section we closely follow the treatment of self-concordance due to Jarre [21, 22] rather than [37]. The foundation of all interior-point methods is Newton's method. Let $g : \mathbf{R}^n \to \mathbf{R}$ be a three-times differentiable function, and suppose we want to minimize g over $\mathbf{R}^n$ (no constraints). Suppose we have a current point $\mathbf{x}_c$, called an **iterate,** and we would like to compute a better point $\mathbf{x}^+$. Newton's method for optimization is based on expanding g as a Taylor series around $\mathbf{x}_c$:

$$\begin{aligned} g(\mathbf{x}) &= g(\mathbf{x}_c) + \nabla g(\mathbf{x}_c)^T (\mathbf{x} - \mathbf{x}_c) + \frac{1}{2} (\mathbf{x} - \mathbf{x}_c)^T \nabla^2 g(\mathbf{x}_c) (\mathbf{x} - \mathbf{x}_c) \\ &\quad + O\left(\|\mathbf{x} - \mathbf{x}_c\|^3 \right) . \end{aligned} \tag{33.18}$$

A reasonable approach to minimizing g might be to minimize $q(\mathbf{x})$, where $q(\mathbf{x})$ is the quadratic function that arises from dropping high-order terms from (33.18):

$$q(\mathbf{x}) = g(\mathbf{x}_c) + \nabla g(\mathbf{x}_c)^T (\mathbf{x} - \mathbf{x}_c) + \frac{1}{2} (\mathbf{x} - \mathbf{x}_c)^T \nabla^2 g(\mathbf{x}_c) (\mathbf{x} - \mathbf{x}_c) . \tag{33.19}$$

This function $q(\mathbf{x})$ is called a "quadratic model." In the case when $\nabla^2 g(\mathbf{x}_c)$ is a positive definite matrix, a standard linear algebra result tells us that the minimizer of $q(\mathbf{x})$ is

$$\mathbf{x}^+ = \mathbf{x}_c - \left(\nabla^2 g(\mathbf{x}_c) \right)^{-1} \nabla g(\mathbf{x}_c) \tag{33.20}$$

where the exponent '-1' denotes matrix inversion. In a pure Newton's method, $\mathbf{x}^+$ would be taken as the next iterate, and a new quadratic model would be formed. In the case when $\nabla^2 g(\mathbf{x}_c)$ is not positive definite, the pure Newton's method has to be modified in some manner.

In the case of convex programming, however, we have already seen that $\nabla^2 g(\mathbf{x}_c)$ is always at least positive semidefinite, so indefiniteness is not as severe a difficulty as in the general case. Suppose in fact that g is C^2 and that $\nabla^2 g(\mathbf{x})$ is positive definite for all $\mathbf{x}$; then the Newton step is always well-defined. This assumption of positive definiteness implies that g is strictly convex. A function $g(\mathbf{x})$ is said to be **strictly convex** if for all points $\mathbf{x}$,$\mathbf{y}$ in the domain such that $\mathbf{x} \neq \mathbf{y}$, and for all $\lambda \in (0, 1)$,

$$g(\lambda \mathbf{x} + (1 - \lambda)\mathbf{y}) < \lambda g(\mathbf{x}) + (1 - \lambda) g(\mathbf{y}) . \tag{33.21}$$

Strict convexity has the following useful consequence. A strictly convex function defined on a convex domain D has at most one point satisfying the KKT conditions, and hence at most one minimizer. (But a strictly convex function need not have a minimizer: consider the function $g(x) = e^x$.)

One interesting feature of Newton's method is its invariance under affine transformation. In particular, for any fixed nonsingular square matrix A and fixed vector $\mathbf{b}$, Newton's method applied to $g(\mathbf{x})$ starting at $\mathbf{x}_c$ is equivalent to Newton's method applied to $h(\mathbf{x}) = g(A\mathbf{x} + \mathbf{b})$ starting at a point $\mathbf{x}'_c$ defined by $A\mathbf{x}'_c + \mathbf{b} = \mathbf{x}_c$. (Note that other classical methods for optimization, such as steepest descent, do not have this desirable property.) Suppose we want to measure progress in Newton's method, i.e., determine whether the current iterate $\mathbf{x}$ is near the minimizer $\mathbf{x}^*$. The obvious distance measure $\|\mathbf{x} - \mathbf{x}^*\|$ is not useful, since $\mathbf{x}^*$ is not known *a priori;* the measure $g(\mathbf{x}) - g(\mathbf{x}^*)$ suffers from the same flaw. The gradient norm $\|\nabla g(\mathbf{x})\|$ seems like a better choice since we know that $\nabla g(\mathbf{x}^*) = \mathbf{0}$ at optimum, except that this measure of nearness is not preserved under affine transformation. Nesterov and Nemirovskii propose the metric $\left(\nabla g(\mathbf{x})(\nabla^2 g(\mathbf{x}))^{-1}\nabla g(\mathbf{x})\right)^{1/2}$ to measure the degree of optimality. This measure has the advantage of invariance under affine transformation.

Newton's method as described so far does not handle constraints. A classical approach to incorporating constraints into Newton's method is a barrier-function approach. Assume from now on that D is a convex, closed set with an interior point.

Let $F(\mathbf{x})$ be a function defined on the interior of D with the following two properties:

- F is strictly convex, and
- Let $\mathbf{x}$ be an arbitrary point on the boundary of D, and let $\mathbf{x}_1, \mathbf{x}_2, \ldots$ be a sequence of interior points converging to $\mathbf{x}$. Then $F(\mathbf{x}_k) \to \infty$ as $k \to \infty$.

In this case, F is said to be a **barrier function** for D. For example, the function $F(x) = -\ln x$ is a barrier function for the nonnegative real numbers $\{x \in \mathbf{R} : x \geq 0\}$. Note that a set D admits many different barrier functions. Barrier functions are a classical technique in optimization—see for instance [11]—although their prominence in the literature had waned during the 1970s and 1980s until the advent of interior-point methods.

For the rest of this section we will make the assumption that the objective function is a linear function $\mathbf{c}^T\mathbf{x}$. Thus, our problem is written:

$$\begin{array}{ll} \text{minimize} & \mathbf{c}^T\mathbf{x} \\ \text{subject to} & h_1(\mathbf{x}) \leq 0\,, \\ & \vdots \\ & h_q(\mathbf{x}) \leq 0\,. \end{array} \tag{33.22}$$

This assumption is without loss of generality for the following reason. Given a general convex programming problem with a nonlinear objective function of the form (33.5), we can introduce a new variable z and a new constraint $f(\mathbf{x}) \leq z$ (which is convex, since $f(\mathbf{x}) - z$ is a convex function), and then we replace the objective function with "minimize z."

The barrier-function method for minimizing $\mathbf{c}^T\mathbf{x}$ on D is as follows. We choose a sequence of positive parameters $\mu_0, \mu_1, \mu_2, \ldots$ strictly decreasing to zero. On the kth iteration of the barrier-function method, we minimize $\mathbf{c}^T\mathbf{x} + \mu_k F(\mathbf{x})$ on the interior of D using Newton's method. The starting point for Newton's method is taken to be the (approximately-computed) minimizer of $\mathbf{c}^T\mathbf{x} + \mu_{k-1}F(\mathbf{x})$ from the previous iteration. The kth iteration continues until a sufficiently good approximate minimizer of $\mathbf{c}^T\mathbf{x} + \mu_k F(\mathbf{x})$ is found, at which point we move to iteration $k+1$. Since $\mu_k \to 0$ as $k \to \infty$, in the limit we are minimizing the original objective function $\mathbf{c}^T\mathbf{x}$.

The reason this method makes progress is as follows. Because the barrier function blows up near the boundaries of D, the minimizer of the sum $\mathbf{c}^T\mathbf{x} + \mu_k F(\mathbf{x})$ will be bounded away from the boundaries of D. As we let μ_k approach zero, the importance of the barrier function is diminished, and the original optimization problem dominates. The intuition behind the barrier function method is as follows. Newton's method works very well in the unconstrained case, but has no notion of inequality constraints. The barrier function has information about the constraints encoded in its Hessian in a way that is useful for Newton's

method. In particular, the barrier term prevents Newton's method from coming close to the boundary too soon.

Consider $\mu > 0$ fixed for a moment. Because of the assumption that F is strictly convex, the objective function $\mathbf{c}^T\mathbf{x} + \mu F(\mathbf{x})$ has at most one minimizer. Let this minimizer be denoted $\mathbf{x}(\mu)$. By the KKT conditions, $\mathbf{c} + \mu\nabla F(\mathbf{x}(\mu)) = \mathbf{0}$. The **central path** is defined to be the collection of points $\{\mathbf{x}(\mu) : 0 < \mu < \infty\}$. The case when $\mathbf{c}^T\mathbf{x} + \mu F(\mathbf{x})$ has no minimizer at all can occur if the domain D is unbounded and $\mathbf{c}^T\mathbf{x}$ has no lower bound over the domain. We do not consider that case here. An example of the central path is presented in Fig. 33.1.

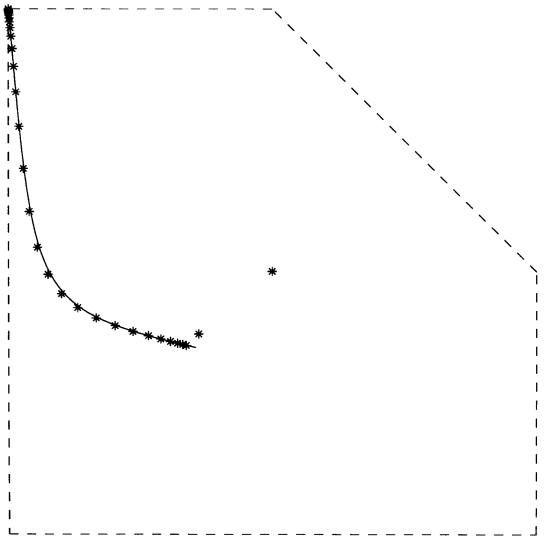


FIGURE 33.1 The central path, which is the solid trajectory, and iterates in a small-step path-following interior-point method, which are asterisks. The dashed segments indicate the boundary of the feasible region. The example problem is linear programming with two variables and five constraints. The iterates lie close to but not exactly on the central path, and their spacing decreases geometrically.

Everything presented so far concerning barrier function methods was known in the 1960s. The important recent progress has been the discovery of the self-concordance conditions, due to Nesterov and Nemirovskii. The following conditions are called the **finite-difference self-concordance conditions.** Let int(D) denote the interior of the feasible region D. Suppose $\mathbf{x} \in \text{int}(D)$ and let E be the open ellipsoid

$$E = \left\{\mathbf{y} : (\mathbf{y} - \mathbf{x})^T \nabla^2 F(\mathbf{x})(\mathbf{y} - \mathbf{x}) < 1\right\} . \tag{33.23}$$

The first finite-difference self-concordance condition is that

$$E \subset \text{int}(D) . \tag{33.24}$$

Next, suppose $\mathbf{y} \in E$ and let

$$r = \left((\mathbf{y} - \mathbf{x})^T \nabla^2 F(\mathbf{x})(\mathbf{y} - \mathbf{x})\right)^{1/2} \tag{33.25}$$

[so that, by assumption, $r < 1$ and $\mathbf{y} \in \text{int}(D)$]. The second condition is that for all $\mathbf{h} \in \mathbf{R}^n$,

$$(1 - r)^2 \mathbf{h}^T \nabla^2 F(\mathbf{x})\mathbf{h} \le \mathbf{h}^T \nabla^2 F(\mathbf{y})\mathbf{h} \le \frac{1}{(1 - r)^2} \mathbf{h}^T \nabla^2 F(\mathbf{x})\mathbf{h} . \tag{33.26}$$

We assume from now on that F satisfies (33.24) and (33.26). The finite-difference self-concordance conditions are the easiest to use in the analysis of interior-point methods, but they are difficult to verify because of multitude of free parameters in (33.26). In the next section we will state the differential self-concordance condition, which implies (33.24) and (33.26) but is easier to verify analytically.

The rationale behind (33.26) is as follows. The condition that $r < 1$ means that $\mathbf{y}$ lies in an ellipsoidal open set around $\mathbf{x}$, where the ellipsoid is defined by the symmetric positive definite matrix $\nabla^2 F(\mathbf{x})$. It is very natural to define a neighborhood in this manner, because other simpler definitions of a neighborhood around $\mathbf{x}$ would not be invariant under affine transformations. An ellipsoid like (33.23) is sometimes known as a **Dikin ellipsoid.** Condition (33.26) says that the Hessian of F does not vary too much over this natural neighborhood. Thus, we see the significance of the term "self-concordant:" variation in the Hessian of F is small with respect to the neighborhood defined by the Hessian itself. Subtracting $\mathbf{h}^T \nabla^2 F(\mathbf{x})\mathbf{h}$ from all three quantities of (33.26) and simplifying yields

$$\left|\mathbf{h}^T \left(\nabla^2 F(\mathbf{y}) - \nabla^2 F(\mathbf{x})\right) \mathbf{h}\right| \leq \left(\frac{1}{(1-r)^2} - 1\right) \mathbf{h}^T \nabla^2 F(\mathbf{x})\mathbf{h} \,. \tag{33.27}$$

The self-concordance condition is not sufficient by itself to yield a good algorithm. The difficulty is that the minimizer of F could be very close to a boundary of F, causing a very steep gradient in a small neighborhood and slowing down convergence. An example of this undesirable behavior, due to Jarre, is the self-concordant barrier function $-(\ln x)/\epsilon - \ln(1-x)$ for the interval $[0, 1]$ where $\epsilon > 0$ is small. This barrier function has its minimizer arbitrarily close to 1, and very large derivatives near the minimizer. To prevent this behavior, we impose a second condition, called the "self-limiting" condition by Jarre, that for all $\mathbf{x} \in \text{int}(D)$,

$$\nabla F(\mathbf{x})^T (\nabla^2 F(\mathbf{x}))^{-1} \nabla F(\mathbf{x}) \leq \theta \tag{33.28}$$

where $\theta \geq 1$ is a parameter that measures the quality of the self-concordant function. This parameter θ is called the **self-concordance parameter.** One way to interpret (33.28) is that it is the simplest possible upper bound on the first derivative of F that is invariant under affine transformations.

The parameter θ plays a crucial role in the analysis of interior-point methods. Thus, the question arises, what are typical values for θ? In subsequent sections we present specific classes of self-concordant barrier functions arising in applications, along with their accompanying values of θ. For this section the reader should imagine that $\theta = O(q)$, where q is the number of inequality constraints.

One kind of path-following chooses the parameter sequence $\mu_0, \mu_1, \mu_2, \ldots$ according to the rule

$$\mu_k = (1-\sigma)\mu_{k-1} \,, \tag{33.29}$$

where σ is a problem-dependent scalar that is constant across all iterations. This is known as "small-step" path-following in the literature, and is the easiest rule to analyze, though not very efficient in practice. Because of the small changes in μ per step, there is a need for only one Newton step at each iteration.

Let us fix the iteration number k and cease to write it as a subscript. Note that $\nabla^2 g = \mu \nabla^2 F$ since the second derivative of the linear term is zero. From iterate $\mathbf{x}$ the next iterate $\mathbf{x}^+$ is computed by applying one step of Newton's method to g, i.e.,

$$\mathbf{x}^+ := \mathbf{x} - \mu^{-1} \left(\nabla^2 F(\mathbf{x})\right)^{-1} (\mathbf{c} + \mu \nabla F(\mathbf{x})) \tag{33.30}$$

where we have used the fact that $\nabla g(\mathbf{x}) = \mathbf{c} + \mu \nabla F(\mathbf{x})$. Now we define $\mu^+ = (1-\sigma)\mu$ by (33.29) and repeat the iteration.

In order to carry out a complexity analysis, we would like to claim that if the current iterate $\mathbf{x}$ is close to the central path for parameter μ, then the next iterate $\mathbf{x}^+$ is also close to the central path for parameter μ^+. This requires a notion of proximity to the central path. Here we use the metric mentioned above. For each $(\mathbf{y}, \mu)$, we let

$$\lambda(\mathbf{y}, \mu) = \mu^{-1} \left[\left(\mathbf{c} + \mu \nabla F(\mathbf{y})\right)^T \left(\nabla^2 F(\mathbf{y})\right)^{-1} (\mathbf{c} + \mu \nabla F(\mathbf{y})) \right]^{1/2} . \tag{33.31}$$

Observe that if $\mathbf{y}$ is the point on the central path for parameter μ, then $\lambda(\mathbf{y},\mu) = 0$.

Returning to our complexity analysis, we claim that if $\lambda(\mathbf{x},\mu) \leq 1/5$, then $\lambda(\mathbf{x}^+, \mu^+) \leq 1/5$. This claim will require the correct choice of σ in (33.29). To prove this claim, we first analyze the intermediate quantity $\lambda(\mathbf{x}^+, \mu)$. In other words, we would like to know how much progress is made during one step of Newton's method applied to g. To preserve generality, let us state this as a theorem.

THEOREM 33.5 *Let $\mu > 0$ and $\mathbf{x} \in int(D)$ be given. Let λ denote $\lambda(\mathbf{x},\mu)$, and suppose $\lambda < 1$. Let $\mathbf{x}^+$ be chosen by (33.30). Then*

$$\lambda\left(\mathbf{x}^+, \mu\right) \leq \frac{\lambda^2}{(1-\lambda)^2} \,. \tag{33.32}$$

PROOF Let $\mathbf{v} = -\mu^{-1}(\nabla^2 F(\mathbf{x}))^{-1}(\mathbf{c} + \mu\nabla F(\mathbf{x}))$, which is the update in (33.30), and let $\mathbf{y} = \mathbf{x} + s\mathbf{v}$ for some $s \in [0, 1]$ to be chosen later. Note that if $s = 1$, then $\mathbf{y} = \mathbf{x}^+$. We see from the definition (33.31) of $\lambda(\cdot, \cdot)$ that r in (33.25) is λs and hence satisfies $r < 1$ by assumption. Therefore, by (33.27), for any $\mathbf{h} \in \mathbf{R}^n$,

$$\left|\mathbf{h}^T \left(\nabla^2 F(\mathbf{x}) - \nabla^2 F(\mathbf{y})\right) \mathbf{h}\right| \leq \left(\frac{1}{(1-s\lambda)^2} - 1\right) \mathbf{h}^T \nabla^2 F(\mathbf{x}) \mathbf{h} \,. \tag{33.33}$$

We now apply the "generalized Cauchy–Schwarz" inequality from Jarre [22], which is the following. Let A, M be two symmetric matrices such that A is positive definite and such that $|\mathbf{z}^T M \mathbf{z}| \leq \mathbf{z}^T A \mathbf{z}$ for all vectors $\mathbf{z}$. Then for all $\mathbf{a}, \mathbf{b}$, $(\mathbf{a}^T M \mathbf{b})^2 \leq (\mathbf{a}^T A \mathbf{a})(\mathbf{b}^T A \mathbf{b})$. We omit the proof here. Observe that if we define $A = \gamma \nabla^2 F(\mathbf{x})$, where γ is the scalar on the right-hand side of (33.33), and $M = \nabla^2 F(\mathbf{x}) - \nabla^2 F(\mathbf{y})$, then the hypothesis of the generalized Cauchy–Schwarz inequality holds by (33.33). Therefore, we can apply the conclusion, with "$\mathbf{a}$" taken to be an indeterminate vector $\mathbf{h}$ and "$\mathbf{b}$" taken to be $\mathbf{v}$ to obtain

$$\begin{aligned} \left|\mathbf{v}^T \left(\nabla^2 F(\mathbf{x}) - \nabla^2 F(\mathbf{y})\right) \mathbf{h}\right| &\leq \left(\frac{1}{(1-s\lambda)^2} - 1\right) \\ &\quad \sqrt{\mathbf{v}^T \nabla^2 F(\mathbf{x}) \mathbf{v}} \sqrt{\mathbf{h}^T \nabla^2 F(\mathbf{x}) \mathbf{h}} \qquad (33.34) \\ &= \left(\frac{1}{(1-s\lambda)^2} - 1\right) \lambda \sqrt{\mathbf{h}^T \nabla^2 F(\mathbf{x}) \mathbf{h}} \,. \qquad (33.35) \end{aligned}$$

Let us now come up with new expressions for the two terms on the left-hand side of (33.34). Recall by choice of $\mathbf{v}$ that the first term $\mathbf{v}^T \nabla^2 F(\mathbf{x}) \mathbf{h}$ evaluates to $-(\mathbf{c}/\mu + \nabla F(\mathbf{x}))^T \mathbf{h}$. For the second term, recall that $\mathbf{y} = \mathbf{x} + s\mathbf{v}$. Thinking of $\mathbf{y}$ as a function of s, observe that $\nabla^2 F(\mathbf{y})\mathbf{v} = \nabla^2 F(\mathbf{x} + s\mathbf{v})\mathbf{v} = d(\nabla F(\mathbf{x} + s\mathbf{v}))/ds = d(\mathbf{c}/\mu + \nabla F(\mathbf{x} + s\mathbf{v}))/ds$. Thus, we have

$$\begin{aligned} \left|-(\mathbf{c}/\mu + \nabla F(\mathbf{x}))^T \mathbf{h} - \frac{d(\mathbf{c}/\mu + \nabla F(\mathbf{x} + s\mathbf{v}))^T}{ds} \mathbf{h}\right| &\leq \left(\frac{1}{(1-s\lambda)^2} - 1\right) \\ &\quad \lambda \sqrt{\mathbf{h}^T \nabla^2 F(\mathbf{x}) \mathbf{h}} \,. \qquad (33.36) \end{aligned}$$

Let us define

$$p(s) = (1-s)(\mathbf{c}/\mu + \nabla F(\mathbf{x}))^T \mathbf{h} - (\mathbf{c}/\mu + \nabla F(\mathbf{x} + s\mathbf{v}))^T \mathbf{h} \,. \tag{33.37}$$

Observe that $p(0) = 0$ because the two terms cancel. Observe also that (33.36) is equivalent to

$$\left|p'(s)\right| \leq \left(\frac{1}{(1-s\lambda)^2} - 1\right) \lambda \sqrt{\mathbf{h}^T \nabla^2 F(\mathbf{x}) \mathbf{h}} \,. \tag{33.38}$$

Therefore, by integrating both sides of (33.38) for s from 0 to 1, we conclude that

$$|p(1)| \leq \frac{\lambda^2}{1-\lambda} \sqrt{\mathbf{h}^T \nabla^2 F(\mathbf{x}) \mathbf{h}} \,. \tag{33.39}$$

On the other hand, directly from (33.37) we have $p(1) = -(\mathbf{c}/\mu + \nabla F(\mathbf{x}+\mathbf{v}))^T \mathbf{h}$. We combine this with (33.39), using the fact that $\mathbf{x}+\mathbf{v} = \mathbf{x}^+$, to obtain

$$\left|\left(\mathbf{c}/\mu + \nabla F\left(\mathbf{x}^+\right)\right)^T \mathbf{h}\right| \leq \frac{\lambda^2}{1-\lambda}\sqrt{\mathbf{h}^T \nabla^2 F(\mathbf{x})\mathbf{h}}\,. \tag{33.40}$$

We can now put an upper bound on the right-hand side of (33.40) using self-concordance to obtain

$$\left|\left(\mathbf{c}/\mu + \nabla F\left(\mathbf{x}^+\right)\right)^T \mathbf{h}\right| \leq \frac{\lambda^2}{(1-\lambda)^2}\sqrt{\mathbf{h}^T \nabla^2 F(\mathbf{x}^+)\mathbf{h}}\,. \tag{33.41}$$

Now finally we will define the hitherto indeterminate $\mathbf{h}$ to be $\mathbf{h} = (\nabla^2 F(\mathbf{x}^+))^{-1}(\mathbf{c}/\mu + \nabla F(\mathbf{x}^+))$. If we substitute this into (33.41) and use the definition of $\lambda(\cdot,\cdot)$, we obtain

$$\lambda\left(\mathbf{x}^+, \mu\right)^2 \leq \frac{\lambda^2}{(1-\lambda)^2}\lambda\left(\mathbf{x}^+, \mu\right)\,. \tag{33.42}$$

Dividing both sides by $\lambda(\mathbf{x}^+, \mu)$ proves (33.32).

In the specific case $\lambda \leq 1/5$, we see that $\lambda(\mathbf{x}^+, \mu) \leq 1/16$. Next, we want to show that $\lambda(\mathbf{x}^+, \mu^+) \leq 1/5$. Observe that the function $\mathbf{y} \mapsto (\mathbf{y}^T \nabla^2 F(\mathbf{x}^+)^{-1}\mathbf{y})^{1/2}$ is a norm on $\mathbf{R}^n$, and hence obeys the triangle inequality. Thus, we can apply this triangle inequality to $\mu^+\lambda(\mathbf{x}^+, \mu^+) - \mu\lambda(\mathbf{x}^+, \mu)$ [plugging in (33.31)] to obtain

$$\begin{aligned}
\mu^+\lambda\left(\mathbf{x}^+, \mu^+\right) - \mu\lambda\left(\mathbf{x}^+, \mu\right) &\leq \left|\mu - \mu^+\right| \\
&\quad \cdot \left[\nabla F\left(\mathbf{x}^+\right)^T \left(\nabla^2 F\left(\mathbf{x}^+\right)\right)^{-1} \nabla F\left(\mathbf{x}^+\right)\right]^{1/2} \qquad (33.43)\\
&\leq \left|\mu - \mu^+\right| \cdot \sqrt{\theta} \qquad (33.44)
\end{aligned}$$

where recall that θ was defined in (33.28). If we substitute (33.29) for μ^+ and simplify, we obtain

$$\lambda\left(\mathbf{x}^+, \mu^+\right) \leq \frac{\sigma\sqrt{\theta} + \lambda\left(\mathbf{x}^+, \mu\right)}{1-\sigma}\,. \tag{33.45}$$

If we choose

$$\sigma = \frac{1}{9\sqrt{\theta}} \tag{33.46}$$

and use the inequality $\lambda(\mathbf{x}^+, \mu) \leq 1/16$, we conclude finally that $\lambda(\mathbf{x}^+, \mu^+) \leq 25/128 < 1/5$.

Observe that (33.46) controls how fast μ can decrease in (33.29). Thus, the rate of convergence of the interior-point method is determined by the value of the self-concordance parameter.

The next step in our analysis is to show that as μ is driven to zero, the interior-point method converges to the true minimizer of (33.22). Let $(\mathbf{x}, \mu)$ be an iterate of the above algorithm, so that $\lambda(\mathbf{x}, \mu) \leq 1/5$. Let $\mathbf{x}^*$ denote the global minimizer for the original problem (33.22). We now show that $\mathbf{c}^T\mathbf{x} - \mathbf{c}^T\mathbf{x}^* \leq O(\mu)$, so the difference of the iterates from optimality tends to zero. This argument is broken up as two separate theorems. First, we show that $c^T\mathbf{x} \leq \mathbf{c}^T\hat{\mathbf{x}} + O(\mu)$, where $\hat{\mathbf{x}}$ is the point on the central path for μ. Second, we show $\mathbf{c}^T\hat{\mathbf{x}} \leq \mathbf{c}^T\mathbf{x}^* + O(\mu)$.

THEOREM 33.6 *Let $\mathbf{x} \in int(D)$ and $\mu > 0$ satisfy $\lambda(\mathbf{x}, \mu) \leq 1/5$. Let $\hat{\mathbf{x}} \in D$ be the central path point for μ, i.e., $\mathbf{c} + \mu\nabla F(\hat{\mathbf{x}}) = \mathbf{0}$. Then*

$$c^T\mathbf{x} \leq \mathbf{c}^T\hat{\mathbf{x}} + \mu\left(0.27\sqrt{\theta} + 0.044\right)\,. \tag{33.47}$$

PROOF Let us consider applying Newton's method for minimizing $\mathbf{c}^T\mathbf{x} + \mu F(\mathbf{x})$ starting from $\mathbf{x}$, keeping μ fixed. Observe that (33.32) says that Newton's method converges quadratically, and it must converge to $\hat{\mathbf{x}}$. Let the Newton iterates be $\mathbf{y}^{(0)}(=\mathbf{x}), \mathbf{y}^{(1)}, \mathbf{y}^{(2)}, \ldots$ and sequence of proximities be denoted $\lambda_0, \lambda_1, \ldots$, so $\lambda_0 \leq 1/5$ by assumption. Let us consider two consecutive iterates in this sequence $\mathbf{y}^{(k)}, \mathbf{y}^{(k+1)}$ that we denote $\mathbf{y}$ and $\mathbf{y}^+$. By the definition of Newton's method, we have

$$\mathbf{y}^+ - \mathbf{y} = -\mu^{-1}\left(\nabla^2 F(\mathbf{y})\right)^{-1}(\mathbf{c} + \mu\nabla F(\mathbf{y})) \tag{33.48}$$

so

$$\begin{aligned}
\mu \cdot \left|\mathbf{c}^T\mathbf{y}^+ - \mathbf{c}^T\mathbf{y}\right| &= \left|\mathbf{c}^T\left(\nabla^2 F(\mathbf{y}))^{-1}(\mathbf{c} + \mu\nabla F(\mathbf{y})\right)\right| && (33.49)\\
&= \Big|(\mathbf{c} + \mu\nabla F(\mathbf{y}))^T\left(\nabla^2 F(\mathbf{y}))^{-1}(\mathbf{c} + \mu\nabla F(\mathbf{y})\right) - \\
&\qquad \mu\nabla F(\mathbf{y})^T\left(\nabla^2 F(\mathbf{y}))^{-1}(\mathbf{c} + \mu\nabla F(\mathbf{y})\right)\Big| && (33.50)\\
&\leq t + \mu t^{1/2} s^{1/2} && (33.51)
\end{aligned}$$

where

$$t = (\mathbf{c} + \mu\nabla F(\mathbf{y}))^T\left(\nabla^2 F(\mathbf{y})\right)^{-1}(\mathbf{c} + \mu\nabla F(\mathbf{y})) \tag{33.52}$$

and

$$s = \nabla F(\mathbf{y})^T\left(\nabla^2 F(\mathbf{y})\right)^{-1}\nabla F(\mathbf{y})\,. \tag{33.53}$$

We obtained (33.51) by applying first the triangle inequality to split up the two terms of (33.50), and then the Cauchy–Schwarz inequality to the second term. Observe that $t = (\lambda_k\mu)^2$ by definition of λ_k. Also, observe that $s \leq \theta$ by (33.28). Substituting these bounds into (33.51) and dividing by μ yields

$$\left|\mathbf{c}^T\mathbf{y}^+ - \mathbf{c}^T\mathbf{y}\right| \leq \mu\left(\lambda_k^2 + \lambda_k\sqrt{\theta}\right)\,. \tag{33.54}$$

Thus,

$$\left|\mathbf{c}^T\hat{\mathbf{x}} - \mathbf{c}^T\mathbf{x}\right| \leq \mu\sum_{k=0}^{\infty}\left(\lambda_k^2 + \lambda_k\sqrt{\theta}\right)\,. \tag{33.55}$$

Now we use the fact that the λ's are a decreasing quadratic series starting from $1/5$ and bounded by (33.32) to find that the sum of the λ_k's is at most 0.27 and the sum of λ_k^2 is at most 0.044. This proves (33.47).

THEOREM 33.7 *Let $(\hat{\mathbf{x}}, \mu)$ be a point on the central path. Let $\mathbf{x}^*$ be the optimal solution. Then*

$$\mathbf{c}^T\hat{\mathbf{x}} \leq \mathbf{c}^T\mathbf{x}^* + \mu\theta\,. \tag{33.56}$$

PROOF Let $\mathbf{h} = \mathbf{x}^* - \hat{\mathbf{x}}$. Then $\mathbf{c}^T\hat{\mathbf{x}} - \mathbf{c}^T\mathbf{x}^* = -\mathbf{c}^T\mathbf{h}$. To prove the theorem, we must show that $-\mathbf{c}^T\mathbf{h} \leq \mu\theta$. If $\mathbf{c}^T\mathbf{h} \geq 0$ then this result is trivial, so suppose $\mathbf{c}^T\mathbf{h} < 0$. Let $\psi(t) = F(\hat{\mathbf{x}} + t\mathbf{h})$. Observe that ψ is a convex function of t satisfying $\psi'(t) = \nabla F(\hat{\mathbf{x}} + t\mathbf{h})^T\mathbf{h}$ and $\psi''(t) = \mathbf{h}^T\nabla^2 F(\hat{\mathbf{x}} + t\mathbf{h})\mathbf{h}$. Observe that

$$\begin{aligned}
\left|\psi'(t)\right| &= \left|\nabla F\left(\hat{\mathbf{x}} + t\mathbf{h}\right)^T\mathbf{h}\right| && (33.57)\\
&= \left|\nabla F\left(\hat{\mathbf{x}} + t\mathbf{h}\right)^T\left(\nabla^2 F\left(\hat{\mathbf{x}} + t\mathbf{h}\right)\right)^{-1/2}\left(\nabla^2 F\left(\hat{\mathbf{x}} + t\mathbf{h}\right)\right)^{1/2}\mathbf{h}\right| && (33.58)\\
&\leq \left(\nabla F\left(\hat{\mathbf{x}} + t\mathbf{h}\right)^T\left(\nabla^2 F\left(\hat{\mathbf{x}} + t\mathbf{h}\right)\right)^{-1}\nabla F\left(\hat{\mathbf{x}} + t\mathbf{h}\right)\right)^{1/2}
\end{aligned}$$

$$\left(\mathbf{h}^T \nabla^2 F\left(\hat{\mathbf{x}} + t\mathbf{h}\right)\mathbf{h}\right)^{1/2} \tag{33.59}$$

$$\leq \quad \theta^{1/2}\psi''(t)^{1/2} \tag{33.60}$$

where (33.59) follows from (33.58) by the Cauchy–Schwarz inequality, and (33.60) follows from (33.28). Thus, $\psi''(t) \geq \psi'(t)^2/\theta$. Observe also that $\psi'(0) = \nabla F(\hat{\mathbf{x}})^T\mathbf{h} = -\mathbf{c}^T\mathbf{h}/\mu$ since $\mathbf{c} + \mu\nabla F(\hat{\mathbf{x}}) = \mathbf{0}$ (by the optimality of $\hat{\mathbf{x}}$).

Now, consider the function χ defined by $\chi(t) = (-\mu/(\mathbf{c}^T\mathbf{h}) - t/\theta)^{-1}$. Observe that $\chi(0) = -\mathbf{c}^T\mathbf{h}/\mu$, so $\chi(0) = \psi'(0)$. Also, note that $\chi'(t) = \theta^{-1}(-\mu/(\mathbf{c}^T\mathbf{h}) - t/\theta)^{-2} = \chi(t)^2/\theta$.

Thus, χ is the solution to the initial value problem $u(0) = -\mathbf{c}^T\mathbf{h}/\mu$, $u'(t) = u(t)^2/\theta$. On the other hand, ψ' is a solution to the differential inequality $u(0) = -\mathbf{c}^T\mathbf{h}/\mu$, $u'(t) \geq u(t)^2/\theta$. Since u^2/θ is an increasing function of u for positive u, a theorem about differential inequalities tells us that ψ' must dominate χ for $t \geq 0$. But notice that χ blows up to ∞ at $t = -\mu\theta/(\mathbf{c}^T\mathbf{h})$ (recall we are assuming $\mathbf{c}^T\mathbf{h} < 0$). Thus, ψ' must blow up at some t_0 satisfying $t_0 \leq -\mu\theta/(\mathbf{c}^T\mathbf{h})$. On the other hand, we already know that ψ does not blow up on $[0, 1)$ because by convexity $\hat{\mathbf{x}} + t\mathbf{h} \in \text{int}(D)$ for $t < 1$. Thus, $-\mu\theta/(\mathbf{c}^T\mathbf{h}) \geq 1$, i.e., $-\mathbf{c}^T\mathbf{h} \leq \mu\theta$. This proves (33.56).

We can now summarize the interior-point method presented in this section.

Interior-point algorithm for (33.22)
Start with $\mu_0 > 0$ and $\mathbf{x}_0 \in \text{int}(D)$ satisfying $\lambda(\mathbf{x}_0, \mu_0) \leq 1/5$.
$i := 0$.
while $\mu_i(\theta + 0.27\sqrt{\theta} + 0.044) > \epsilon$
  $\mathbf{x}_{i+1} := \mathbf{x}_i - \mu_i^{-1}(\nabla^2 F(\mathbf{x}_i))^{-1}(\mathbf{c} + \mu_i \nabla F(\mathbf{x}_i))$.
  $\mu_{i+1} := (1 - 1/(9\sqrt{\theta}))\mu_i$.
  $i := i + 1$.
end while
return $\mathbf{x}_i$.

The preceding algorithm lacks an initialization procedure, i.e., a method to construct μ_0 and $\mathbf{x}_0$. In many situations it is easy to find $\mathbf{x}_0 \in \text{int}(D)$, but not so easy to find an $\mathbf{x}_0$ that is close to the central path (i.e., that satisfies $\lambda(\mathbf{x}_0, \mu_0) \leq 1/5$ for some μ_0). In this case, there is a general-purpose iterative procedure that starts at interior point $\mathbf{x}_0$ and eventually produces a point near the central path. This iterative procedure is based on the following observation. For any $\mathbf{x}_0 \in \text{int}(D)$, $\mathbf{x}_0$ is the minimizer of the artificial objective function $g(\mathbf{x}) = \mathbf{c}_0^T\mathbf{x} + \mu_0 F(\mathbf{x})$, where $\mathbf{c}_0 = -\nabla F(\mathbf{x}_0)$ and $\mu_0 = 1$: this claim is easily verified by checking that $\nabla g(\mathbf{x}_0) = \mathbf{0}$. Therefore, we can use a path-following method on the sequence of objective functions $\mathbf{c}_0^T\mathbf{x} + \mu F(\mathbf{x})$ and try to drive μ toward ∞ instead of 0. Once μ is sufficiently large, the first term of the objective function no longer matters, and can be replaced by $\mathbf{c}^T\mathbf{x}$ where $\mathbf{c}$ is the actual gradient of the linear functional that is under consideration. We omit the details.

The running time of this initialization procedure is determined by how close $\mathbf{x}_0$ is to the boundary of D. If $\mathbf{x}_0$ is close to the boundary, then $\nabla F(\mathbf{x}_0)$ is large, and the initialization procedure requires more steps to make μ sufficiently large. Thus, a good starting point for an interior-point method would be near the **analytic center** of D. The analytic center is the point $\mathbf{x}_a$ that minimizes the barrier function $F(\mathbf{x})$, i.e., the point satisfying $\nabla F(\mathbf{x}_a) = \mathbf{0}$. The analytic center is the limit of the central path as $\mu \to \infty$.

This approach to initializing an interior-point method, in which one works with an artificial objective function first and then the actual objective function, is commonly called "phase I—phase II" in the literature. Similar approaches have also been used for initializing the simplex method; see, e.g., [6].

If no interior feasible point is known, then there is no general-purpose procedure to initialize the interior-point algorithm, and an initial point must be constructed from additional information about the

problem. In the special cases of linear, quadratic and semidefinite programming, there are initialization techniques that do not need any other additional *a priori* information.

We would like to compare the complexity of the ellipsoid method with the interior-point method. A complete comparison is not possible because of the incompatible assumptions made concerning the initialization procedure. Therefore, we compare only the rates of convergence. Recall that the ellipsoid method guarantees error tolerance of γ after $O(n^2 \log(1/\gamma))$ iterations. Each iteration requires a rank-one update to an $n \times n$ matrix given by (33.11), which requires $O(n^2)$ operations.

The convergence rate of the interior-point method is determined as follows. Convergence to tolerance ϵ is achieved after $\mu \leq O(\epsilon/\theta)$ as seen from the algorithm. On each step μ is decreased by a factor of $1 - \text{const} \cdot \theta^{-1/2}$. Thus, the number of steps p to reduce the error tolerance to ϵ must satisfy $(1 - \text{const} \cdot \theta^{-1/2})^p \leq \epsilon/\theta$. By taking logarithms, using the approximation $\ln(1+\delta) \approx \delta$, and dropping the low-order term, we conclude that the number of iterations is $O(\sqrt{\theta}\log(1/\epsilon))$. Each iteration requires the solution of a system of $n \times n$ linear equations given by (33.30), which takes $O(n^3)$ operations.

Let us assume for now that $\theta = O(n)$, where n is the dimension of the problem. Actual values for barrier parameters are provided in the next section. In this case, we conclude that the interior-point method improves on the number of iterations over the ellipsoid method by a factor of $n^{1.5}$. On the other hand, an interior-point iteration is a factor of $O(n)$ more expensive than an ellipsoid iteration, so the total savings comes out to a factor of $O(\sqrt{n})$. Karmarkar [25] showed how to save another factor of $\sqrt{n}$ in an interior-point algorithm for LP by approximately solving the linear systems using information from the previous iteration. The amortized cost of this technique is $O(n^{2.5})$ per iteration rather than $O(n^3)$, but this technique is not often used in practice.

In fact, as mentioned in the introduction, interior-point methods in practice are far more efficient than the ellipsoid method. The reason for the improvement in practice is not so much the theoretical factor of $O(\sqrt{n})$ or $O(n)$ mentioned in the last paragraph. Rather, interior-point methods are efficient in practice because the number of iterations is usually much less than $O(\sqrt{\theta}\log(1/\epsilon))$: in particular, usually it is possible to decrease μ at a much faster rate than (33.29) and still maintain approximate centrality. In such an algorithm, the decrease in μ is chosen adaptively rather than according to a fixed formula like (33.29). Nonetheless, even for these algorithms, there is no known upper bound better than $O(\sqrt{\theta}\log(1/\epsilon))$ iterations. The reason for the mismatch between the upper bound and practical behavior is not completely understood. Todd and Ye [53] showed a lower bound of $\Omega(n^{1/3}\log(1/\epsilon))$ for linear programming (where $n = \theta$) that holds for a large variety of interior-point methods.

A second reason why interior-point methods outperform the ellipsoid method is that the linear system given by (33.30) in practice is often sparse, meaning that most entries of $\nabla^2 F(\mathbf{x})$ are zeros. Many special methods have been developed for solving sparse systems of linear equations [15]; the running time for solving sparse equations can be much better than the worst-case estimate of $O(n^3)$. Sparsity considerations are the primary reason that the amortized $O(n^{2.5})$ linear system solver mentioned above is not used in practice. The ellipsoid method as presented in Section 33.3 is not able to take advantage of sparsity because (33.11) implies that B_{i+1} in general will be a completely dense $n \times n$ matrix. On the other hand, Todd [50] and Burrell and Todd [5] show that a different way of representing B_{i+1} leads to better preservation of sparsity in the ellipsoid method, in which case the sparsity properties of the two algorithms may be comparable.

33.5 Additional Remarks on Self-Concordance

In this section we give some additional theoretical background on self-concordance and an example.

The first question is, for what convex sets do self-concordant barrier functions exist? It turns out that *every* closed, convex subset D of $\mathbf{R}^n$ with a nonempty interior has a self-concordant barrier function with

parameter $\theta = O(n)$. Given such a set D, Nesterov and Nemirovskii prove that the function

$$F(\mathbf{x}) = \text{const} \cdot \ln \text{vol}\left(\left\{\mathbf{a} \in \mathbf{R}^n : \mathbf{a}^T(\mathbf{y} - \mathbf{x}) \leq 1 \quad \forall \mathbf{y} \in D\right\}\right) \tag{33.61}$$

is self-concordant with parameter $O(n)$. This function is called the **universal** barrier. This barrier is not useful in practice because there is evidently no algorithm to evaluate (33.61), let alone its derivatives, for general convex sets. In practice, one constructs self-concordant barrier functions for certain special cases of commonly occurring forms of constraints.

Another question is, what is the best possible value of θ? In the last paragraph it was claimed that it is theoretically possible to always choose $\theta \leq \text{const} \cdot n$. On the other hand, it can be proved that $\theta \geq 1$ in all cases. This lower bound of 1 is tight. The barrier function

$$F(\mathbf{x}) = \ln\left(1 - x_1^2 - \cdots - x_n^2\right) \tag{33.62}$$

when D is the unit ball in $\mathbf{R}^n$ has self-concordance parameter 1. This result generalizes to any ellipsoidal constraint, since self-concordance is preserved by affine transformations.

How does one verify that (33.62) is indeed self-concordant? The two conditions (33.24) and (33.26) in the last section appear nontrivial to check, even for a simple function written down in closed form like (33.62).

It turns out that there is a simpler definition of self-concordance, which is as follows. A function F is **self-concordant** if for all $\mathbf{x} \in \text{int}(D)$ and for all $\mathbf{h} \in \mathbf{R}^n$

$$\left|\nabla^3 F(\mathbf{x})[\mathbf{h}, \mathbf{h}, \mathbf{h}]\right| \leq 2\left(\mathbf{h}^T \nabla^2 F(\mathbf{x})\mathbf{h}\right)^{3/2} . \tag{33.63}$$

The notation on the left-hand side means the application of the trilinear form $\nabla^3 F(\mathbf{x})$ to the three vectors $\mathbf{h}, \mathbf{h}, \mathbf{h}$. (We could have used analogous notation for the right-hand side: $2(\nabla^2 F(\mathbf{x})[\mathbf{h}, \mathbf{h}])^{3/2}$.) It should be apparent at an intuitive level why (33.63) and (33.26) are related: a bound on the third derivative in terms of the second means that the second derivative cannot vary too much over a neighborhood defined in terms of the second derivative.

The proof that (33.63) implies (33.26) is based on such an argument. The trickiest part of the proof is an inequality involving trilinear forms and related to the generalized Cauchy–Schwarz inequality above. The trilinear inequality is proved in [37] and has been simplified by Jarre [22]. Furthermore, under an assumption of sufficient differentiability, the other direction holds: (33.26) implies (33.63) as noted by Todd [51].

We now present one particularly simple barrier function, easily analyzed by (33.63). We consider the following barrier function for the positive orthant $O_n = \{\mathbf{x} \in \mathbf{R}^n : x_1 \geq 0, \ldots, x_n \geq 0\}$:

$$F(\mathbf{x}) = -\sum_{i=1}^{n} \ln x_i . \tag{33.64}$$

This barrier is the key to linear programming. The gradient $\nabla F(\mathbf{x})$ is seen to be $(-1/x_1, \ldots, -1/x_n)$ and the Hessian $\text{diag}(1/x_1^2, \ldots, 1/x_n^2)$, where $\text{diag}(\cdot)$ denotes a diagonal matrix with the specified entries. This matrix is positive definite so we see that (33.64) is indeed a strictly convex function that tends to ∞ at the boundaries of the orthant. The third derivative is $\text{diag}(-2/x_1^3, \ldots, -2/x_n^3)$, where "diag" denotes a diagonal tensor. Thus,

$$\nabla^3 F(\mathbf{x})[\mathbf{h}, \mathbf{h}, \mathbf{h}] = -2h_1^3/x_1^3 - \cdots - 2h_n^3/x_n^3 \tag{33.65}$$

compared to

$$\mathbf{h}^T \nabla^2 F(\mathbf{x})\mathbf{h} = h_1^2/x_1^2 + \cdots + h_n^2/x_n^2 . \tag{33.66}$$

It is now obvious that (33.63) holds. Furthermore, (33.28) is also easily checked; one finds that $\theta = n$ for this barrier. Nesterov and Nemirovskii show that n is also a lower bound for the barrier parameter on the positive orthant, and hence this barrier is optimal.

33.6 Semidefinite Programming and Primal-Dual Methods

In this section we specialize the interior-point framework to semidefinite programming. Interior-point methods for semidefinite programming were developed independently by Nesterov and Nemirovskii [37] and Alizadeh [1]. In this section we follow Alizadeh's treatment.

The following optimization problem is said to be primal standard form **semidefinite programming** (SDP):

$$\begin{array}{ll} \text{minimize} & C \bullet X \\ \text{subject to} & A_1 \bullet X = b_1 \ , \\ & \vdots \\ & A_m \bullet X = b_m \ , \\ & X \succeq 0 \ . \end{array} \tag{33.67}$$

The notation here is as follows. All of $C, A_1, \ldots, A_m, X$ are symmetric $n \times n$ matrices. Matrices $A_1, \ldots, A_m$ and C are given as data, as are scalars $b_1, \ldots, b_m$. Matrix X is the unknown. The notation $Y \bullet Z$ means elementwise inner product, i.e., $Y \bullet Z = \Sigma_{i,j} y_{ij} z_{ij}$. This is equal to trace XY if X, Y are symmetric. The constraint $X \succeq 0$ means that X is positive semidefinite. More generally, the notation $X \succeq Y$ means $X - Y$ is positive semidefinite. This order is known as the **Löwner partial order.**

The set S of symmetric positive semidefinite matrices is convex as noted in Section 33.2. Furthermore, the function $F(X) = -\ln \det(X)$ is a self-concordant barrier function on S whose parameter of self-concordance is n. We omit the proof; see [37]. This barrier is optimal for S in the sense that there is no self-concordant barrier on S with parameter less than n. (A set like the feasible region of (33.67), which is defined as the conjunction of a semidefinite constraint and equality constraints, may admit a better barrier. See remarks on this issue in the next section.)

Given this specification of the barrier, it is now straightforward to set up the interior-point method of Section 33.4 and solve (33.67). The only difficulty is the inclusion of linear equality constraints. This is handled by regarding the barrier function F as restricted to the affine set given by $\{X : A_i \bullet X = b_i \text{ for each } i\}$. Restricting $F(X)$ to this affine set means that the Newton step (33.30) now must be modified with linear projections on the Hessian and gradient. An example of a projected Newton step for LP is presented in the next section.

In this section we introduce **primal-dual** interior-point methods. In practice, primal-dual methods are preferred for SDP and LP over the primal method of Section 33.4. The framework of self-concordant barrier functions can be extended to primal-dual interior-point methods at a fairly general level [37], but we treat only the two special cases of primal-dual SDP and LP.

The dual problem for a convex optimization can be obtained from the KKT conditions (33.4): the multipliers in the KKT conditions turn out to be the solution to another convex optimization problem called the dual. An alternative way to derive the dual is by considering the KKT conditions of the barrier problem rather than the original problem. The SDP barrier problem is to minimize $C \bullet X - \mu \ln \det(X)$ subject to $X \succ 0$, $A_i \bullet X = b_i$ for $i = 1, \ldots, m$. The notation $X \succ 0$ means X is positive definite. Consider the KKT conditions for this barrier problem. Observe that the relation $X \succ 0$ drops out of the KKT conditions because it is inactive at the optimizer (because the barrier function blows up at the boundary of the feasible region, since $\det X = 0$ if X is semidefinite but not positive definite). Note also that the gradient of $\ln \det(X)$ is the function $X \mapsto X^{-1}$. Therefore, the KKT condition for the barrier problem is

$$C - \mu X^{-1} = y_1 A_1 + \cdots + y_m A_m \tag{33.68}$$

plus feasibility:

$$A_1 \bullet X = b_1, \ldots, A_m \bullet X = b_m \ . \tag{33.69}$$

Here, $y_1, \ldots, y_m$, are unconstrained Lagrange multipliers.

We claim that these are also the KKT conditions for another SDP barrier problem. Consider the SDP

$$\begin{array}{ll} \text{maximize} & b_1 y_1 + \cdots + b_m y_m \\ \text{subject to} & C - S = y_1 A_1 + \cdots + y_m A_m \ , \\ & S \succeq 0 \ , \end{array} \tag{33.70}$$

where $S, y_1, \ldots, y_m$ are the variables. Attaching a barrier changes the objective to maximizing $b_1 y_1 + \cdots + b_m y_m + \mu \ln \det(S)$. The KKT optimality condition is that $b_1 = A_1 \bullet X, \ldots, b_m = A_m \bullet X$, and $X = \mu S^{-1}$. Here X is a multiplier for the equality constraint of (33.70). But notice that the identification $X = \mu S^{-1}$ means that (33.68) and (33.69) are satisfied! Thus, the two barrier problems have the same KKT conditions. We say that (33.70) is the **dual** to (33.67). One can check that the dual of the dual, after some simplification, is again the primal.

Let us assume that a constraint qualification such as the Slater condition holds. Then the KKT conditions are necessary and sufficient for both primal and dual optimality. Furthermore, the primal and dual barrier objective function optimal values satisfy the following relation:

$$\begin{aligned} C \bullet X &= (S + y_1 A_1 + \cdots + y_m A_m) \bullet X && (33.71) \\ &= S \bullet X + y_1 A_1 \bullet X + \cdots + y_m A_m \bullet X && (33.72) \\ &= S \bullet X + b_1 y_1 + \cdots + b_m y_m && (33.73) \\ &= n\mu + b_1 y_1 + \cdots + b_m y_m \ . && (33.74) \end{aligned}$$

The last line was obtained by noting that since $SX = \mu I$, $S \bullet X = n\mu$. This follows because $S \bullet X = \text{trace } SX = \text{trace } \mu I$.

Thus, we see that for the barrier optimizers, the original primal function minimum value is exactly $n\mu$ greater than the dual objective function maximum value. Since we drive μ to zero in an interior-point method, we conclude that the two SDP's (33.67) and (33.70) have the same optimum value. Because of relationship (33.74), the parameter μ in an interior-point method is sometimes called the **duality gap.**

A **primal-dual interior-point method** for (33.67) and (33.70) is a method that simultaneously updates the primal and dual iterates, attempting to shrink the duality gap to zero. Alizadeh proposes a potential-reduction primal-dual algorithm for SDP, which is a generalization of Ye's [63] potential-reduction algorithm for linear programming. We define $q = n + \sqrt{n}$ and define

$$\psi(X, S) = q \ln(X \bullet S) - \ln \det(XS) \ . \tag{33.75}$$

The motivation for this definition is as follows. The first term decreases to $-\infty$ as the duality gap tends to 0. On the other hand, the second term is a penalty for loss of centrality. Unlike the method of Section 33.4, a potential-reduction method does not explicitly maintain proximity to the central path. Each iteration reduces the potential by a fixed constant amount, thus driving the potential to $-\infty$. The step taken is related to Newton's method for decreasing the potential function. To gain further insight, consider the case when X, S are on the central path, so that $XS = \mu I$: then the value of the potential function is $q \ln n + \sqrt{n} \ln \mu$, which tends to $-\infty$ as $\mu \to 0$. This algorithm has the same theoretical running time as the algorithm in Section 33.4, namely $O(\sqrt{n} \log(1/\epsilon))$ iterations to reduce the duality gap to $O(\epsilon)$.

We conclude this section with some applications of SDP, drawn from Vandenberghe and Boyd [55]. The most direct application is to eigenvalue optimization problems. For instance suppose we want to minimize the maximum eigenvalue of an unknown symmetric matrix A that is parameterized by n free variables $x_1, \ldots, x_n$ according to the formula $A = A_0 + x_1 A_1 + \cdots + x_n A_n$, where $A_0, \ldots, A_n$ are given symmetric matrices. This can be stated as the following semidefinite program: find the minimum t such that $tI - A_0 - x_1 A_1 - \cdots - x_n A_n \succeq 0$. Some applications and further references for eigenvalue optimization are in [55]. Another application is quadratically constrained quadratic programming, which is minimizing a convex quadratic function subject to ellipsoidal and linear constraints. An ellipsoidal

constraint $(\mathbf{x}-\mathbf{c})^T A(\mathbf{x}-\mathbf{c}) \leq 1$ can be transformed into an SDP constraint of the form:

$$\begin{bmatrix} I & R(\mathbf{x}-\mathbf{c}) \\ (\mathbf{x}-\mathbf{c})^T R^T & 1 \end{bmatrix} \succeq 0 \tag{33.76}$$

where R is a matrix such that $R^T R = A$ (e.g., the Cholesky factor of A). Note that a collection of SDP constraints can be concatenated along the diagonal to make a single large SDP constraint in order to put the problem in standard form. (These transformations do not necessarily lead to the most efficient algorithm for ellipsoidal constraints.) Another geometric problem that can be transformed to SDP is finding the smallest ellipsoid that contains the union of a set of given ellipsoids. See [55]. There are also applications of SDP to combinatorial optimization and control theory described in [1] and [55]. It should be noted that some of the combinatorial implications of semidefinite programming were observed before the advent of interior-point methods because the ellipsoid method can also solve semidefinite problems. See Grötschel, Lovász, and Schrijver [17, 18].

33.7 Linear Programming

A special case of SDP is linear programming. If we suppose that each matrix A_i is diagonal and C is diagonal, then one can show that the solution X is also diagonal without loss of generality. In this case, the inequality $X \succeq 0$ means that each diagonal entry of X is nonnegative. Thus, if we let $\mathbf{x}$ be the vector of diagonal entries of X, and similarly for S, A_i's and C, then we obtain the primal and dual forms of linear programming:

$$\begin{array}{ll} \text{minimize} & \mathbf{c}^T\mathbf{x} \\ \text{subject to} & A\mathbf{x} = \mathbf{b}\,, \\ & \mathbf{x} \geq \mathbf{0} \end{array} \tag{33.77}$$

and

$$\begin{array}{ll} \text{maximize} & \mathbf{b}^T\mathbf{y} \\ \text{subject to} & A^T\mathbf{y} + \mathbf{s} = \mathbf{c}\,, \\ & \mathbf{s} \geq \mathbf{0}\,. \end{array} \tag{33.78}$$

In these equations, A, $\mathbf{b}$,$\mathbf{c}$ are given: A is an $m \times n$ matrix assumed to have rank m (and hence $m \leq n$), $\mathbf{b}$ is an m-vector, and $\mathbf{c}$ is an n-vector. The variables are $\mathbf{x}$ for the primal and $(\mathbf{y},\mathbf{s})$ for the dual.

The barrier function for linear programming is the specialization of the SDP barrier to the case of diagonal matrices, which turns out to be the barrier for the positive orthant (33.64). Recall that this barrier has parameter n. Although this barrier is optimal for the orthant, it is not necessarily optimal for the orthant in conjunction with the linear constraints. The combination of the two types of constraints means that the feasible region is actually a polytope of dimension $n-m$ in the primal case, and m in the dual case. Since the noncomputable universal barrier has parameter $O(n-m)$ in the one case and $O(m)$ in the other case, the question arises whether there is a computable barrier with a better parameter than n for linear programming. Partial progress on this was made by Vaidya and is described in [37]: Vaidya's barrier has parameter $O(\sqrt{mn})$ for the dual problem.

Using the standard barrier (33.64), we can write down LP with the barrier functions and derive the KKT conditions. This yields the following system of equations that describe the central path for both the primal and dual. These equations are a special case of (33.68)–(33.69) for SDP derived in the previous section.

$$A\mathbf{x} = \mathbf{b}\,, \tag{33.79}$$

$$A^T\mathbf{y} + \mathbf{s} = \mathbf{c}\,, \tag{33.80}$$

$$x_i s_i = \mu \quad \text{for } i = 1, \ldots, n, \tag{33.81}$$

$$x_i, s_i > 0 \quad \text{for } i = 1, \ldots, n\,. \tag{33.82}$$

We have introduced the LP central path as a special case of SDP, which is in turn a special case of general convex programming, but in fact, the LP central path was discovered and analyzed before the others. This discovery is usually credited to [2, 29, 31, 47], and several of these authors considered algorithms that follow this path. The first interior-point method for LP was Karmarkar's [25] projective method, which requires $O(n \log(1/\epsilon))$ iterations. Gill et al. [16] spotted the connection between Newton's method and Karmarkar's interior-point method. Renegar [41] also related interior-point methods to Newton's method and used this analysis to reduce the running time to $O(\sqrt{n} \log(1/\epsilon))$ iterations. Note that the $\sqrt{n}$ factor is the specialization of the $\sqrt{\theta}$ factor from the general convex case since $\theta = n$ for (33.64).

Kojima, Mizuno and Yoshise [28] and Monteiro and Adler [33] introduced a primal-dual path-following method for (33.79)–(33.82). The idea is similar to what we have seen before: an iterate is a solution to (33.79)–(33.82), except (33.81) is satisfied only approximately. We then decrease μ and take a step of Newton's method to regain centrality. The iteration bound is also $O(\sqrt{n} \log(1/\epsilon))$. Because of the simple form, we can write down Newton's method explicitly. First, we need a measure of proximity to the central path. It turns out that the standard proximity measure $\lambda(\cdot, \cdot)$ for the primal or dual alone, after some manipulation and substitution of primal-dual approximations, has the form $\|XS\mathbf{e}/\mu - \mathbf{e}\|$, where $X = \text{diag}(\mathbf{x})$, $S = \text{diag}(\mathbf{s})$, and $\mathbf{e}$ is the vector of all 1's.

Assuming we have a primal-dual point $(\mathbf{x}, \mathbf{y}, \mathbf{s})$ satisfying (33.79), (33.80) and (33.82), and a parameter μ such that $\|XS\mathbf{e}/\mu - \mathbf{e}\| \leq \lambda_0$ (where $\lambda_0 = 1/5$ for example), we can take a step to solve (33.79)–(33.82) for a smaller value of μ, say $\bar{\mu}$. The Newton step $(\Delta\mathbf{x}, \Delta\mathbf{y}, \Delta\mathbf{s})$ linearizes (33.81) and is defined by

$$A\Delta\mathbf{x} = \mathbf{0}, \tag{33.83}$$

$$A^T\Delta\mathbf{y} + \Delta\mathbf{s} = \mathbf{0}, \tag{33.84}$$

$$X\Delta\mathbf{s} + S\Delta\mathbf{x} = -XS\mathbf{e} + \bar{\mu}\mathbf{e}. \tag{33.85}$$

Many of the best current interior-point methods are based on (33.83)–(33.85). Observe that (33.83)–(33.85) define a square system of linear equations with special structure; currently there is significant research on how to solve these equations efficiently and accurately. See [62].

It should be pointed out that Newton's method for solving (33.79)–(33.82) as a system of nonlinear equations, which is the way (33.83)–(33.85) were obtained, is not the same as Newton's method for minimizing the barrier in either the primal or dual. Newton's method for minimizing the barrier in the primal is equivalent to Newton's method for solving a system of nonlinear equations similar to (33.79)–(33.82) except with the third equation replaced by $s_i = \mu/x_i$. The equation $s_i = \mu/x_i$ is of course equivalent to $x_i s_i = \mu$, but these two equations induce different Newton steps. See El-Bakry et al. [10] for more information on this matter.

33.8 Complexity of Convex Programming

In this section we consider complexity issues for linear, semidefinite and convex programming. "Complexity" means establishing bounds on the asymptotic running time for large problems. An immediate issue we must confront is the problem of modeling the numerical computations in the ellipsoid and interior-point method. Actual computers have floating-point arithmetic, which provides a fixed number of significant digits (about 16 decimal digits in IEEE standard double precision) for arithmetic. Restricting arithmetic to a fixed number of digits precludes the solution of poorly conditioned problems, and hence is not desirable for a complexity theory.

One way of modeling the notion that the precision of arithmetic should not have an *a priori* limit is the Turing machine model of computation, covered in detail in another chapter. For the purpose of numerical algorithms, the Turing machine model can be regarded as a computational device that carries out operations on arbitrary-precision rational numbers, and the running time of a rational operation is polynomially dependent on the number of bits in the numerators and denominators. The Turing machine model is also called the "bit complexity" model in the literature.

In this model, the data for a linear programming instance is presented as a triple $(A, \mathbf{b}, \mathbf{c})$ where the entries of these items must be rational numbers. In fact, without loss of generality, let us assume the data is integral since we can clear common denominators in a rational representation, causing at most a polynomial increase in the size of the problem.

Let L be the total number of bits to write this data. We assume that L is at least $mn + m + n$, i.e., each coordinate entry of $(A, \mathbf{b}, \mathbf{c})$ requires at least one bit.

Khachiyan [26] showed that if an LP instance has a feasible solution, then it must have a feasible solution lying in a ball of radius $O(2^{cL})$ of the origin, where c is a universal constant. This ball can thus be used as the initial ellipsoid for the ellipsoid method. Furthermore, when the solution is known to accuracy $O(2^{-dL})$, then a rounding procedure can determine the exact answer.

A word is needed here concerning "the exact answer." For a linear programming problem, it can be shown that the exact answer is the solution to a square system of linear equations whose coefficients are all rational and are derived from the initial data $(A, \mathbf{b}, \mathbf{c})$. This is because the optimal solution in a linear programming problem is always attained at a vertex (provided the feasible region has a vertex). Vertices of a polytope are determined by which constraints are active at that vertex. Solving a system of equations over the rationals can be done exactly with only a polynomial increase in bit-length, a result due to Edmonds [9].

Thus, in the case of linear programming with integer data, it is possible to deduce from the problem data itself valid initialization and termination procedures without additional knowledge. These procedures are also guaranteed to correctly diagnose problems with no feasible points, feasible sets with volume 0, and problems with an unbounded objective function. Khachiyan's analysis shows that the number of arithmetic operations for the ellipsoid method is bounded by $O(n^4 L)$. (The actual Turing machine running time is higher by a factor of L^c to account for the time to do each arithmetic operation.) This bound is a polynomial in the length of the input, which is L. Thus, Khachiyan's linear programming algorithm is polynomial time. This result generalizes to quadratic programming.

A similar analysis can be carried out for interior-point methods; see, e.g.,[57] for the details. One reaches the conclusion that Karmarkar's method requires $O(nL)$ iterations to find the optimal solution, whereas Renegar's requires $O(\sqrt{n}L)$ iterations, where each iteration involves solving a system of equations. In the Turing-machine polynomial-time analysis of both the ellipsoid and interior-point methods, it is necessary to truncate the numerical data after each major iteration to $O(L)$ digits (else the number of digits grows exponentially). This truncation requires additional analysis to make sure that the invariants of the two algorithms are not violated.

Until 1979, the question of polynomiality of LP was a well-known open question. The reason is that the dominant algorithm in practice for solving LP problems until recently has been the simplex method of Dantzig [6]. The simplex method has many variants because many rules are possible for the selection of the "pivot column." Klee and Minty [27] showed that some common variants of the simplex method are exponential time in the worst case, even though they are observed to run in polynomial time in practice. "Exponential" means that the running time could be as large as $\Omega(2^n)$ operations. Kalai [23] proposed a variant of simplex that has worst-case running time lower than exponential but still far above polynomial. There is no known variant of simplex that is polynomial-time, but on the other hand, there is also no proof that such a variant does not exist.

The ellipsoid algorithm, while settling the question of polynomiality, is not completely satisfactory for the following reason. Observe that the number of arithmetic operations of the simplex method depends on the dimensions of the relevant matrices and vectors but not on the data in the matrix. This is common for other numerical computations that have finite algorithms, such as solving systems of equations, finding maximum flows, and so on. An algorithm whose number of arithmetic operations is polynomial in the dimension of the problem, and whose intermediate numerical data require a polynomially-bounded number of bits, is said to be **strongly polynomial time.** Notice that the ellipsoid method is not strongly polynomial time because the number of arithmetic operations depends on the number of bits L in the data rather than on n.

The question of whether there is a strongly polynomial time algorithm for LP remains open. Some partial progress was made by Megiddo [30], who showed that linear programming in a low, fixed dimension is strongly polynomial. Low dimensional linear programming arises in computational geometry. Partial progress in another direction was made by Tardos [49], who proposed an LP algorithm based on the ellipsoid method such that the number of arithmetic operations depends only on the number of bits in A, and is independent of the number of bits in **b** and **c**. This algorithm uses "rounding" to the nearest integer as a main operation. Tardos's result is important because many network optimization problems can be posed as LP instances in which the entries of A have all small integer data, but **b** and **c** have complicated numerical data. Tardos's result was generalized by Vavasis and Ye as described below.

We should add a word on how interior-point methods are initialized and terminated in practice. Three common techniques for initialization are phase I–phase II methods, big-M methods, and infeasible methods. Phase I–phase II methods were described at the end of Section 33.4. A big-M method appends a dummy variable to the LP instance, whose entry in the objective function vector is some very large number M. Because of the new variable, a feasible interior point is easily found. Because of the large weight on the new variable, it is driven to zero at optimum. A final class of initialization methods are "infeasible" interior-point methods. In these methods, an iterate satisfying (33.82) and approximately satisfying (33.81) is maintained, but the iterate does not necessarily satisfy (33.79) or (33.80). A trajectory is followed that simultaneously achieves feasibility and optimality in the limit.

All three initialization methods are guaranteed to work correctly and in polynomial time if the parameters are chosen correctly. For instance, M in the big-M method should be at least 2^{cL} for a theoretical guarantee. Infeasible interior-point methods are currently the best in practice but are the most difficult to analyze. See Wright [62].

As for termination, many practical interior-point methods simply stop when μ is sufficiently small and output the iterate at that step as the optimizer. Khachiyan's solution for termination at an exact optimizer is generally not used in practiced. Ye [65] proposed a termination procedure based on projection that also gives the exact solution, but possibly at a much earlier stage than the theoretical worst-case $\sqrt{n}L$ iterations.

Finally, we mention a recent LP interior-point algorithm by Vavasis and Ye [59] that is based on decomposing the system of Eqs. (33.83)–(33.85) into "layers." This algorithm has the property that its running time bound to find the exact optimizer is polynomial in a function of A that does not depend on **b** and **c**. If A contains integer data, then the bound by Vavasis and Ye depends polynomially on the number of bits in A. Thus, the layered algorithm gives a new proof of Tardos's result, but it is more general because no integrality assumption is made. In the case of generic real data, the running time of the algorithm depends on a parameter $\bar{\chi}_A$ that was discovered independently by a number of authors including Stewart [48] and Todd [54]. The earliest discoverer of the parameter is apparently Dikin [8].

Vavasis and Ye use a big-M initialization, but they show that the running time is independent of how big M is (i.e., the running time bound does not degrade if M is chosen too large). Furthermore, finite termination to an exact optimizer is built into the layered algorithm. Another consequence of the algorithm is a characterization of the LP central path as being composed of at most $O(n^2)$ alternating straight and curved segments such that the straight segments can be followed with a line search in a single step. There are still some obstacles preventing this algorithm from being completely practical; in particular, it is not known how to efficiently estimate the parameter $\bar{\chi}_A$ in a practical setting. Also, it is not known whether the approach generalizes beyond linear programming since there is no longer a direct relation between the layered step of [59] and Newton's method.

The complexity results in this section are not known to carry over to SDP. Consider the following SDP feasibility question: Given $A_1, \ldots, A_m$, each a symmetric $n \times n$ matrix with integer entries, and given m integer scalars $b_1, \ldots, b_m$, does there exist a real symmetric $n \times n$ matrix X satisfying $A_1 \bullet X = b_1, \ldots, A_m \bullet X = b_m$, $X \succeq 0$? This problem is not known to be solvable in polynomial time; in fact, it is not known even to lie in NP. The best Turing machine complexity bound, due to Porkolab and Khachiyan [40], is $O(mL2^{p(n)})$ operations, where $p(n)$ is a polynomial function of n, and L is the number of bits to write $A_1, \ldots, A_m, b_1, \ldots, b_m$. The technique uses a decision procedure for problems over the reals due to

Renegar [42]. Note that the Turing machine is not required to compute such an X, but merely produce a yes/no answer. Such a problem is called a "decision problem" or "language recognition problem."

There are two hurdles to generalizing Khachiyan's LP analysis to this SDP decision problem. The first is that the exact solution to an SDP is irrational even if the original problem data is all integral. This hurdle by itself is apparently not such a significant difficulty. For instance, in [60] it is shown that a certain optimization problem (the trust-region problem) with an irrational optimizer nonetheless has an associated decision problem solvable in polynomial time. The trust-region problem is much simpler than SDP, but the arguments of [60] might generalize to SDP.

The second hurdle, apparently more daunting, is that points in the feasible region of an SDP problem may be double-exponentially large in the size of the data (as opposed to LP, where they are at most single-exponentially large). For instance, consider example of minimizing x_n subject to $x_0 \geq 2$, and $x_1 \geq x_0^2$, $x_2 \geq x_1^2, \ldots, x_n \geq x_{n-1}^2$. (Recall that, as shown at the end of Section 33.6, convex quadratic constraints can be expressed in the SDP framework.) Clearly any feasible point satisfies $x_n \geq 2^{2^n}$. Note that even writing down 2^{2^n} as a binary number requires an exponential number of digits. This example is from Alizadeh [1], who attributes it to M. Ramana.

Since problems beyond linear and quadratic programming do not have rational solutions even when the data is rational, many researchers have abandoned the Turing machine model of complexity and adopted a real-number model of computation, in which the algorithm manipulates real numbers with unit cost per arithmetic operation. The preceding complexity bounds for the simplex method and layered-step method are both valid in the real number model. Other LP algorithms, as well as interior-point and ellipsoid algorithms for more general convex problems, need some new notion to replace the factor L. Several recent papers, e.g., [13, 43, 64], have defined a "condition number" for convex programming problems and have bounded the complexity in terms of this condition number.

33.9 Nonconvex Optimization

The powerful techniques of convex programming can sometimes carry over to nonconvex problems, although generally none of the accompanying theory carries over to the nonconvex setting. For instance, the barrier function method was originally invented for general inequality-constrained nonconvex optimization—see, e.g., Wright [61]—although it is does not necessarily converge to a global or even local optimizer in that setting. More recently, El-Bakry et al. [10] have looked at following the primal-dual central path for nonconvex optimization (which, as noted above, is not the same as following the barrier trajectory). In this section we will briefly summarize some known complexity results and approaches to nonconvex problems.

Perhaps the simplest class of nonconvex continuous optimization problems is nonconvex quadratic programming, that is, minimizing $\mathbf{x}^T H\mathbf{x}/2 + \mathbf{c}^T\mathbf{x}$ subject to linear constraints, where H is a symmetric but not positive semidefinite matrix. This problem is NP-hard [45], and NP-complete when posed as a decision problem [56]. Even when H has only one negative eigenvalue, the problem is still NP-hard [39]. Testing whether a point is a *local* minimizer is NP-hard [34]. If a polynomial time algorithm existed for computing an approximate answer guaranteed to be within a constant factor of optimal, this would imply that $\tilde{P} = \tilde{NP}$, where $\tilde{P}$ and $\tilde{NP}$ are complexity classes believed to be unequal [3].

In spite of all these negative results, some positive results are possible for nonconvex optimization. First, there are classes of nonconvex problems that can be transformed to SDP by nonlinear changes of variables; see [55] for examples. Sometimes the transformation from nonconvex to convex is not an exact transformation. For instance, if the feasible region is strictly enlarged, then the convex problem is said to be a **relaxation** of the nonconvex problem. The relaxed problem, while not equivalent to the original nonconvex problem, nonetheless yields a lower bound on the optimum value which may be useful for other purposes. See [55] for examples of semidefinite relaxations of nonconvex problems.

Another approach is to use convex optimization as a "subroutine" for nonconvex optimization. There are many examples of this in the nonlinear optimization literature; see [35]. Sometimes this can be done in a way that gives provable complexity bounds for nonconvex problems, such as the result by Vavasis [58] for nonconvex quadratic programming.

Finally, sometimes interior-point techniques, while not yielding global minima or approximations to global minima, can still deliver interesting results in polynomial time. For instance, Ye [66] shows that a primal-dual potential reduction algorithm applied to nonconvex quadratic programming can approximate a KKT point in polynomial time.

33.10 Research Issues and Summary

We have shown that convex optimization can be solved via two efficient algorithms, the ellipsoid method and interior-point methods. These methods can be derived in the general setting of convex programming and can also be specialized to semidefinite and linear programming. Perhaps the most outstanding research issue in the field is a better understanding of the complexity of semidefinite programming. Is this problem solvable in polynomial time? Is there a guaranteed efficient method for finding a starting point, and for determining whether a feasible region is empty? Another topic of great interest recently is to understand primal-dual path-following methods for SDP. The difficulty is that there are several different, inequivalent ways to generalize the Eqs. (33.83)–(33.85) (which define the primal-dual path-following step for linear programming) to semidefinite programming, and it is not clear which generalization to prefer. See Todd [52] for more information. Another issue not completely understood for SDP is the correct handling of sparsity, since the matrices $A_1, \ldots, A_m$ are often sparse [14].

Linear programming is known to be polynomial time but not known to be strongly polynomial time. This is another notorious open problem.

33.11 Defining Terms

Barrier function: Let D be a convex feasible region whose interior is nonempty. A barrier function $F : \text{int}(D) \to \mathbf{R}$ is a strictly convex function that tends to infinity as the boundary of D is approached.

Central path: Given a convex programming problem, the central path point for some $\mu > 0$ is the minimizer of $f(\mathbf{x}) + \mu F(\mathbf{x})$, where $f(\mathbf{x})$ is the objective function and $F(\mathbf{x})$ is a barrier function for D. The central path is the trajectory of all such points as μ varies from 0 to ∞. This curve joins the analytic center ($\mu = \infty$) of the region to the optimizer ($\mu = 0$). Many interior-point methods select iterates that are close to the central path according to a proximity measure.

Constraint: A feasible region $D \subset \mathbf{R}^n$ in an optimization problem is often represented as the set of points $\mathbf{x}$ satisfying a sequence of constraints involving $\mathbf{x}$. There are two main kinds of constraints: equality constraints, which have the form $g(\mathbf{x}) = 0$, and inequality constraints, which have the form $h(\mathbf{x}) \leq 0$.

Some specific types of constraints are as follows: A linear equality constraint has the form $\mathbf{a}^T\mathbf{x} = b$, where $\mathbf{a} \in \mathbf{R}^n$, $b \in \mathbf{R}$ are specified. A linear inequality constraint has the form $\mathbf{a}^T\mathbf{x} \leq b$. An ellipsoidal constraint has the form $(\mathbf{x} - \mathbf{c})^T A(\mathbf{x} - \mathbf{c}) \leq 1$, where $\mathbf{c}$ is a vector and A is a symmetric $n \times n$ positive definite matrix. A semidefinite constraint has the form

$$X \text{ is symmetric positive semidefinite}$$

where X is a matrix composed of variables.

For a point $\mathbf{x}_1 \in D$, an inequality constraint $h(\mathbf{x}) \leq 0$ is active at $\mathbf{x}_1$ if $h(\mathbf{x}_1) = 0$, else the constraint is inactive. Equality constraints by default are always active.

A convex constraint is either a linear equality constraint defined above, or is an inequality constraint $h(\mathbf{x}) \leq 0$, where h is a convex function. The feasible region defined by one or more convex constraints is convex.

Constraint qualification: A constraint qualification is a condition imposed on the constraints at some feasible point that ensures that every linearized feasible direction is also a feasible direction.

Convex function: Let f be a real-valued function defined on a convex set D. This function is convex if for all $\mathbf{x},\mathbf{y} \in D$ and for all $\lambda \in [0, 1]$, $f(\lambda\mathbf{x} + (1 - \lambda)\mathbf{y}) \geq \lambda f(\mathbf{x}) + (1 - \lambda) f(\mathbf{y})$. This function is strictly convex if the preceding inequality is strict whenever $\mathbf{x} \neq \mathbf{y}$ and $\lambda \in (0, 1)$.

Convex programming: An optimization problem in which the objective function is convex and the feasible region is specified via a sequence of convex constraints is called convex programming.

Convex set: A set $D \subset \mathbf{R}^n$ is convex if for any $\mathbf{x},\mathbf{y} \in D$ and for any $\lambda \in [0, 1]$, $\lambda\mathbf{x} + (1 - \lambda)\mathbf{y} \in D$.

Ellipsoid: Let A be an $n \times n$ symmetric positive definite matrix and let $\mathbf{c}$ be an n-vector. The set $E = \{\mathbf{x} \in \mathbf{R}^n : (\mathbf{x} - \mathbf{c})^T A(\mathbf{x} - \mathbf{c}) \leq 1\}$ is an ellipsoid.

Ellipsoid method: The ellipsoid method, due to Yudin and Nemirovskii [68], is a general-purpose algorithm for solving convex programming problems. It constructs a sequence of ellipsoids with shrinking volume each of which contains the optimizer. See Section 33.3.

Feasible direction: Let $D \subset \mathbf{R}^n$ be a feasible region and say $\mathbf{x} \in D$. A nonzero vector $\mathbf{v}$ is a feasible direction at $\mathbf{x}$ if there exists a sequence of points $\mathbf{x}_1, \mathbf{x}_2, \ldots$ all in D converging to $\mathbf{x}$ and a sequence of positive scalars $\alpha_1, \alpha_2, \ldots$ converging to zero such that $\mathbf{x}_k - \mathbf{x} = \alpha_k\mathbf{v} + o(\alpha_k)$.

Feasible region or feasible set: Defined under *optimization.*

Global optimizer: See *optimizer.*

Gradient: The gradient of a function $f : \mathbf{R}^n \to \mathbf{R}$ is its first derivative and is denoted ∇f. Note that $\nabla f(\mathbf{x})$ is an n-vector.

Hessian: The Hessian of a function $f : \mathbf{R}^n \to \mathbf{R}$ is its second derivative and is denoted $\nabla^2 f$. The Hessian is an $n \times n$ matrix which, under the assumption that f is C^2, is symmetric.

Interior-point method: An interior-point method for convex programming is an algorithm that constructs iterates interior to the feasible region and approaching the optimizer along some trajectory (usually the central path). See Section 33.4 to Section 33.7.

Karush–Kuhn–Tucker (KKT) conditions: Consider the optimization problem $\min\{f(\mathbf{x}) : \mathbf{x} \in D\}$ in which D is specified by equality and inequality constraints and f is differentiable. A point $\mathbf{x}$ satisfies the KKT conditions if $\mathbf{x}$ is feasible and if for every linearized feasible direction $\mathbf{v}$ at $\mathbf{x}$, $\mathbf{v}^T \nabla f(\mathbf{x}) \geq 0$. If $\mathbf{x}$ is a local optimizer and a constraint qualification holds at $\mathbf{x}$, then $\mathbf{x}$ must satisfy the KKT conditions. The standard form of the KKT conditions is given by (33.4) above.

Linear function: The function $\mathbf{x} \mapsto \mathbf{a}^T\mathbf{x} + b$, where $\mathbf{a},b$ are given, is linear.

Linear programming: Linear programming is an optimization problem in which $f(\mathbf{x})$ is a linear function and D is a polyhedron. The standard primal-dual format for linear programming is (33.77) and (33.78).

Linearized feasible direction: Let $D \subset \mathbf{R}^n$ be a feasible region specified as sequence of p equality constraints $g_1(\mathbf{x}) = \cdots = g_p(\mathbf{x}) = 0$ and q inequality constraints $h_1(\mathbf{x}) \leq 0, \ldots, h_q(\mathbf{x}) \leq 0$. Say $\mathbf{x} \in D$, and let $A \subset \{1, \ldots, q\}$ be the constraints active at D. Assume all these constraints are C^1 in a neighborhood of $\mathbf{x}$. A nonzero vector $\mathbf{v}$ is a linearized feasible direction at $\mathbf{x}$ if $\mathbf{v}^T \nabla g_i(\mathbf{x}) = 0$ for all $i = 1, \ldots, p$ and $\mathbf{v}^T \nabla h_i(\mathbf{x}) \leq 0$ for all $i \in A$.

Local optimizer: For the optimization problem $\min\{f(\mathbf{x}) : \mathbf{x} \in D\}$, $D \subset \mathbf{R}^n$, a local optimizer or local minimizer is a point $\mathbf{x}^* \in D$ such that there exists an open set $N \subset \mathbf{R}^n$ containing $\mathbf{x}^*$

such that $f(\mathbf{x}) \geq f(\mathbf{x}^*)$ for all $\mathbf{x} \in D \cap N$. In the case of convex programming problems, a local optimizer is also a global optimizer.

Minimizer: See *optimizer.*

Objective function: Defined under *optimization.*

Optimization: Optimization refers to solving problems of the form: $\min\{f(\mathbf{x}) : \mathbf{x} \in D\}$ where D (the feasible region) is a subset of $\mathbf{R}^n$ and f (the objective function) is a real-valued function. The vector $\mathbf{x}$ is called the vector of variables or decision variables or unknowns. Set D is often described via constraints.

Optimizer: For the optimization problem of minimizing $f(\mathbf{x})$ subject to $\mathbf{x} \in D$, the optimizer or minimizer is the point $\mathbf{x}^* \in D$ such that $f(\mathbf{x}^*) \leq f(\mathbf{x})$ for all $\mathbf{x} \in D$. Also called "global optimizer." Some authors use the term "optimizer" to mean "local optimizer."

Polyhedron: A polyhedron is the set defined by a sequence of linear constraints.

Positive definite: A square matrix A is positive definite if $\mathbf{x}^T A\mathbf{x} > 0$ for all nonzero vectors $\mathbf{x}$. This term is applied mainly to symmetric matrices.

Positive semidefinite: A square matrix A is positive semidefinite if $\mathbf{x}^T A\mathbf{x} \geq 0$ for all vectors $\mathbf{x}$. This term is applied mainly to symmetric matrices.

Quadratic programming: Quadratic programming is the optimization problem of minimizing a quadratic function $f(\mathbf{x}) = \mathbf{x}^T H\mathbf{x}/2 + \mathbf{c}^T\mathbf{x}$ subject to $\mathbf{x} \in D$, where D is a polyhedron.

Self-concordant barrier function: A barrier function F is said to be self-concordant if it satisfies (33.63) for all $\mathbf{x} \in \text{int}(D)$ and all $\mathbf{h} \in \mathbf{R}^n$. The parameter of self-concordance is the smallest θ such that F satisfies (33.28) for all $\mathbf{x} \in \text{int}(D)$.

Semidefinite programming: Semidefinite programming is the problem of minimizing a linear function subject to linear and semidefinite constraints. The standard primal-dual form for SDP is given by (33.67) and (33.70).

Slater condition: The Slater condition is a constraint qualification for convex programming. Let D be a convex feasible region specified by convex constraints. The Slater condition is that there exists a point $\mathbf{x}_0 \in D$ such that all the nonlinear constraints are inactive at $\mathbf{x}_0$. This condition serves as a constraint qualification for all of D, i.e., it implies that for every point $\mathbf{x} \in D$, the linearized feasible directions at $\mathbf{x}$ coincide with the feasible directions at $\mathbf{x}$.

Stationary point: For an optimization problem, a stationary point is a point satisfying the KKT conditions.

Subdifferential: For a convex function f defined on a convex set D, the subdifferential of f at $\mathbf{x}$ is defined to be the set of vectors $\mathbf{v} \in \mathbf{R}^n$ such that $f(\mathbf{y}) \geq f(\mathbf{x}) + \mathbf{v}^T(\mathbf{y} - \mathbf{x})$ for all $\mathbf{y} \in D$. The subdifferential is a nonempty closed convex set that coincides with the ordinary derivative in the case that f is differentiable at $\mathbf{x}$ and $\mathbf{x}$ is in the interior of D.

Subgradient: A subgradient of f at $\mathbf{x}$ is an element of the subdifferential.

Symmetric matrix: A square matrix A is symmetric if $A = A^T$, where the superscript ^{T} indicates matrix transpose.

Acknowledgments

The author received helpful comments on this chapter from Florian Jarre, Michael Todd, and the anonymous referee. This work has been supported in part by NSF grant CCR-9619489. Research supported in part by NSF through grant DMS-9505155 and ONR through grant N00014-96-1-0050. Support was also received from the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Computational and Technology Research, U.S. Dept. of Energy, under Contract W-31-109-Eng-38 through Argonne National Laboratory. Support was also received from the J.S. Guggenheim Foundation.

References

[1] Alizadeh, F., Interior point methods in semidefinite programming with applications to combinatorial optimization. *SIAM Journal on Optimization,* 5, 13–51, 1995.

[2] Bayer, D.A. and Lagarias, J.C., The nonlinear geometry of linear programming. I. Affine and projective scaling trajectories. *Transactions of the AMS,* 314, 499–526, 1989.

[3] Bellare, M. and Rogaway, P., The complexity of approximating a nonlinear program. In *Complexity in Numerical Optimization,* Pardalos, P.M., Ed., World Scientific, 1993.

[4] Bland, R., Goldfarb, D., and Todd, M., The ellipsoid method: a survey. *Operations Research,* 29, 1039–1091, 1981.

[5] Burrell, B. and Todd, M., The ellipsoid method generates dual variables. *Mathematics of Operations Research,* 10, 688–700, 1985.

[6] Dantzig, G., *Linear Programming and Extensions.* Princeton University Press, Princeton, NJ, 1963.

[7] Dikin, I.I., Iterative solution of problems of linear and quadratic programming. *Soviet Math Doklady,* 8, 674–675, 1967.

[8] Dikin, I.I., On the speed of an iterative process. *Upravlyaemye Sistemi,* 12, 54–60, 1974.

[9] Edmonds, J., Systems of distinct representatives and linear algebra. *J. Res. Nat. Bur. Standards,* 71B, 241–245, 1967.

[10] El-Bakry, A.S., Tapia, R.A., Zhang, Y., and Tsuchiya, T., On the formulation and theory of the Newton interior point method for nonlinear programming. Technical Report TR92-40, Department of Computational and Applied Mathematics, Rice University, 1992. To appear in Journal of Optimization Theory and Applications.

[11] Fiacco, A.V. and McCormick, G.P., *Nonlinear Programming: Sequential Unconstrained Minimization Techniques.* John Wiley & Sons, Chichester, U.K., 1968.

[12] Fletcher, R., *Practical Methods of Optimization,* 2nd ed., John Wiley & Sons, Chichester, U.K., 1987.

[13] Freund, R. and Vera, J., Some characterizations and properties of the "distance to ill-posedness" and the condition measure of a conic linear system. Technical Report 3862-95-MSA, MIT Sloan School of Management, 1995.

[14] Fujisawa, K., Kojima, M., and Nakata, K., Exploiting sparsity in primal-dual interior-point methods for semidefinite programming. Technical Report B-324, Mathematical and Computing Sciences, Tokyo Institute of Technology, 1997.

[15] George, A. and Liu, J.W.H., *Computer Solution of Large Sparse Positive Definite Systems.* Prentice–Hall, Englewood Cliffs, NJ, 1981.

[16] Gill, P., Murray, W., Saunders, M., Tomlin, J., and Wright, M., On projected Newton barrier methods for linear programming and an equivalence to Karmarkar's projective method. *Mathematical Programming,* 36, 183–209, 1986.

[17] Grötschel, M., Lovász, L., and Schrijver, A., The ellipsoid method and its consequences in combinatorial optimization. *Combinatorica,* 1, 169–197, 1981.

[18] Grötschel, M., Lovász, L., and Schrijver, A., *Geometric Algorithms and Combinatorial Optimization.* Springer-Verlag, New York, 1988.

[19] Jarre, F., On the method of analytic centers for solving smooth convex programs. In *Optimization,* Dolecki, S., Ed., volume 1405 of *Lecture Notes in Mathematics,* 69–85. Springer-Verlag, 1989.

[20] Jarre, F., On the convergence of the method of analytic centers when applied to convex quadratic programs. *Mathematical Programming,* 49, 341–358, 1991.

[21] Jarre, F., Interior-point methods for convex programming. *Applied Mathematics and Optimization,* 26, 287–311, 1992.

[22] Jarre, F., Interior-point methods via self-concordance or relative Lipschitz condition. Habilitationsschrift, Bayerische Julius-Maximilians-Universität Würzburg, 1994.
[23] Kalai, G., A subexponential randomized simplex algorithm. In *Proceedings of the 24th ACM Symposium on the Theory of Computing,* 475–482, 1992.
[24] Kapoor, S. and Vaidya, P.M., Fast algorithms for convex quadratic programming and multicommodity flows. In *Proc. 18th Annual ACM Symposium on Theory of Computing,* 147–159, 1986.
[25] Karmarkar, N., A new polynomial-time algorithm for linear programming. *Combinatorica,* 4, 373–395, 1984.
[26] Khachiyan, L.G., A polynomial algorithm in linear programming. *Dokl. Akad. Nauk SSSR,* 244, 1093–1086, 1979. Translated in *Soviet Math. Dokl.* 20, 191–194, 1979.
[27] Klee, V. and Minty, G.J., How good is the simplex algorithm? In *Inequalities III,* Shisha, O., Ed., Academic Press, New York, 1972.
[28] Kojima, M., Mizuno, S., and Yoshise, A., A polynomial-time algorithm for a class of linear complementarity problems. *Mathematical Programming,* 44, 1–26, 1989.
[29] McLinden, L., An analogue of Moreau's proximation theorem, with applications to the nonlinear complementarity problem. *Pacific Journal of Mathematics,* 88, 101–161, 1980.
[30] Megiddo, N., Linear programming in linear time when the dimension is fixed. *Journal of the ACM,* 31, 114–127, 1984.
[31] Megiddo, N., Pathways to the optimal set in linear programming. In *Progress in Mathematical Programming: Interior Point and Related Method,* Megiddo, N., Ed., 131–158. Springer-Verlag, New York, 1989.
[32] Mehrotra, S. and Sun, J., A method of analytic centers for quadratically constrained convex quadratic programs. *SIAM J. Numerical Analysis,* 28, 529–544, 1991.
[33] Monteiro, R.C. and Adler, I., Interior path following primal-dual algorithm. Part I: linear programming. *Mathematical Programming,* 44, 27–41, 1989.
[34] Murty, K.G. and Santosh, N.K., Some NP-complete problems in quadratic and nonlinear programming. *Mathematical Programming,* 39, 117–129, 1987.
[35] Nash, S. and Sofer, A., *Linear and Nonlinear Programming.* McGraw-Hill, New York, 1996.
[36] Nemirovsky, A.S. and Yudin, D.B., *Problem Complexity and Method Efficiency in Optimization.* John Wiley & Sons, Chichester, U.K., 1983. Translated by E. R. Dawson from *Slozhnost' Zadach i Effektivnost' Metodov Optimizatsii,* 1979, Glavnaya redaktsiya fiziko-matematicheskoi literatury, Izdatelstva "Nauka."
[37] Nesterov, Y. and Nemirovskii, A., *Interior Point Polynomial Methods in Convex Programming: Theory and Algorithms,* volume 13 of *SIAM Studies in Applied Mathematics.* SIAM Press, Philadelphia, 1994.
[38] Papadimitriou, C.H. and Steiglitz, K., *Combinatorial Optimization: Algorithms and Complexity.* Prentice Hall, Englewood Cliffs, NJ, 1982.
[39] Pardalos, P.M. and Vavasis, S.A., Quadratic programming with one negative eigenvalue is NP-hard. *J. Global Optimiz.,* 1, 15–22, 1991.
[40] Porkolab, L. and Khachiyan, L., On the complexity of semidefinite programs. Technical Report RRR 50-95, Rutgers Center for Operations Research (RUTCOR), Rutgers University, 1995.
[41] Renegar, J., A polynomial-time algorithm based on Newton's method for linear programming. *Mathematical Programming,* 40, 59–94, 1988.
[42] Renegar, J., On the computational complexity and geometry of the first-order theory of the reals. Part I: introduction, preliminaries; the geometry of semi-algebraic sets; the decision problem for the existential theory of the reals. *J. Symbolic Computing,* 13, 255–299, 1992.
[43] Renegar, J., Linear programming, complexity theory, and elementary functional analysis. Unpublished manuscript, Department of Operations Research and Industrial Engineering, Cornell University, 1993.

[44] Rockafellar, R.T., *Convex Analysis.* Princeton University Press, Princeton, NJ, 1970.
[45] Sahni, S., Computationally related problems. *SIAM J. Comp.,* 3, 262–279, 1974.
[46] Shor, N.Z., Convergence rate of the gradient descent method with dilatation of the space. *Kibernetika,* 6(2), 80–85, 1970. Translated in *Cybernetics,* 6(2), 102–108.
[47] Sonnevend, G., An "analytic center" for polyhedrons and new classes of global algorithms for linear (smooth, convex) programming. In *System Modelling and Optimization: Proceedings of the 12th IFIP–Conference held in Budapest, Hungary, Sep. 1985,* Prekopa, A., Szelezsan, J., and Strazicky, B., Eds., volume 84 of *Lecture Notes in Control and Information Sciences,* 866–876. Springer-Verlag, Berlin, Germany, 1986.
[48] Stewart, G.W., On scaled projections and pseudoinverses. *Linear Algebra and its Applications,* 112, 189–193, 1989.
[49] Tardos, É., A strongly polynomial algorithm to solve combinatorial linear programs. *Operations Research,* 34, 250–256, 1986.
[50] Todd, M., On minimum volume ellipsoids containing part of a given ellipsoid. *Mathematics of Operations Research,* 7, 253–261, 1982.
[51] Todd, M., Unpublished lecture notes for OR635: Interior point methods. 1995.
[52] Todd, M., On search directions in interior-point methods for semidefinite programming. Technical Report TR1205, School of Operations Research and Industrial Engineering, Cornell University, 1997.
[53] Todd, M. and Ye, Y., A lower bound on the number of iterations of long-step and polynomial interior-point linear programming algorithms. Technical Report 1082, School of Operations Research and Industrial Engineering, Cornell University, 1994. To appear in *Annals of Operations Research.*
[54] Todd, M.J., A Dantzig-Wolfe-like variant of Karmarkar's interior-point linear programming algorithm. *Operations Research,* 38, 1006–1018, 1990.
[55] Vandenberghe, L. and Boyd, S., Semidefinite programming. *SIAM Review,* 38, 49–95, 1996.
[56] Vavasis, S.A., Quadratic programming is in NP. *Info. Proc. Lett,* 36, 73–77, 1990.
[57] Vavasis, S.A., *Nonlinear Optimization: Complexity Issues.* Oxford University Press, New York, 1991.
[58] Vavasis, S.A., Approximation algorithms for indefinite quadratic programming. *Mathematical Programming,* 57, 279–311, 1992.
[59] Vavasis, S.A. and Ye, Y., A primal-dual interior point method whose running time depends only on the constraint matrix. *Mathematical Programming,* 74, 79–120, 1996.
[60] Vavasis, S.A. and Zippel, R., Proving polynomial-time for sphere-constrained quadratic programming. Technical Report 90-1182, Department of Computer Science, Cornell University, Ithaca, New York, 1990.
[61] Wright, M.H., *Numerical Methods for Nonlinearly Constrained Optimization.* Ph.D. Thesis, Stanford University, 1976.
[62] Wright, S.J., *Primal-Dual Interior-Point Methods.* SIAM Press, Philadelphia, 1997.
[63] Ye, Y., An $O(n^3L)$ potential reduction algorithm for linear programming. *Mathematical Programming,* 50, 239–258, 1991.
[64] Ye, Y., Toward probabilistic analysis of interior-point algorithms for linear programming. *Mathematics of Operations Research,* 1993, to appear, 1991.
[65] Ye, Y., On the finite convergence of interior-point algorithms for linear programming. *Mathematical Programming,* 57, 325–336, 1992.
[66] Ye, Y., On the complexity of approximating a KKT point of quadratic programming. Preprint, 1996.
[67] Ye, Y. and Tse, E., An extension of Karmarkar's projective algorithm for convex quadratic programming. *Math. Progr.,* 44, 157–179, 1989.

[68] Yudin, D.B. and Nemirovsky, A.S., Informational complexity and efficient methods for solving complex extremal problems. *Ekonomika i Matematicheskie Metody,* 12, 357–369, 1976. Translated in *Matekon: Translations of Russian and East European Math. Economics,* 13, 25–45, Spring 1977.

Further Information

Interior-point methods for linear programming are covered in-depth by Wright [62]. Interior-point methods for semidefinite programming are the topic of the review article by Vandenberghe and Boyd [55]. Interior-point methods for general convex programming were introduced by Nesterov and Nemirovskii [37]. General theory of convexity is covered by Rockafellar [44]. General theory and algorithms for nonlinear optimization is covered by Fletcher [12] and Nash and Sofer [35]. Vavasis's book [57] covers complexity issues in optimization. The web page http://www.mcs.anl.gov/otc/ surveys optimization algorithms and software and includes the archive of interior-point preprints.

34

Approximation Algorithms for NP-Hard Optimization Problems

Philip N. Klein
Brown University

Neal E. Young
Dartmouth College

34.1 Introduction

In this chapter, we discuss **approximation algorithms** for **optimization problems.** An *optimization problem* consists in finding the best (cheapest, heaviest, etc.) element in a large set $\mathcal{P}$, called the **feasible region** and usually specified implicitly, where the quality of elements of the set are evaluated using a function $f(x)$, the **objective function,** usually something fairly simple. The element that minimizes (or maximizes) this function is said to be an **optimal solution** and the value of the objective function at this element is the **optimal value.**

$$\text{optimal value} = \min\{f(x) \mid x \in \mathcal{P}\} \tag{34.1}$$

An example of an optimization problem familiar to computer scientists is that of finding a minimum-cost spanning tree of a graph with edge costs. For this problem, the feasible region $\mathcal{P}$, the set over which

we optimize, consists of *spanning trees;* recall that a spanning tree is a set of edges that connect all the vertices but forms no cycles. The value $f(T)$ of the objective function applied to a spanning tree T is the sum of the costs of the edges in the spanning tree.

The minimum-cost spanning tree problem is familiar to computer scientists because there are several good algorithms for solving it — procedures that, for a given graph, quickly determine the minimum-cost spanning tree. No matter what graph is provided as input, the time required for each of these algorithms is guaranteed to be no more than a slowly growing function of the number of vertices n and edges m (e.g., $O(m \log n)$).

For most optimization problems, in contrast to the minimum-cost spanning tree problem, there is no known algorithm that solves all instances quickly in this sense. Furthermore, there is not likely to be such an algorithm ever discovered, for many of these problems are NP-hard, and such an algorithm would imply that every problem in NP could be solved quickly (i.e., P = NP), which is considered unlikely.[1] One option in such a case is to seek an *approximation algorithm*—an algorithm that is guaranteed to run quickly (in time polynomial in the input size) and to produce a solution for which the value of the objective function is quantifiably close to the optimal value.

Considerable progress has been made towards understanding which combinatorial-optimization problems can be approximately solved, and to what accuracy. The theory of NP-completeness can provide evidence not only that a problem is hard to solve precisely but also that it is hard to approximate to within a particular accuracy. Furthermore, for many natural NP-hard optimization problems, approximation algorithms have been developed whose accuracy nearly matches the best achievable according to the theory of NP-completeness. Thus optimization problems can be categorized according to the best accuracy achievable by a polynomial-time approximation algorithm for each problem.

This chapter, which focuses on discrete (rather than continuous) NP-hard optimization problems, is organized according to these categories; for each category, we describe a representative problem, an algorithm for the problem, and the analysis of the algorithm. Along the way we demonstrate some of the ideas and methods common to many approximation algorithms. Also, to illustrate the diversity of the problems that have been studied, we briefly mention a few additional problems as we go. We provide a sampling, rather than a compendium, of the field—many important results, and even areas, are not presented. In "Further Information," we mention some of the areas that we do not cover, and we direct the interested reader to more comprehensive and technically detailed sources, such as the excellent recent book [7]. Because of limits on space for references, we do not cite the original sources for algorithms covered in [7].

34.2 Underlying Principles

Our focus is on *combinatorial* optimization problems, problems where the feasible region $\mathcal{P}$ is finite (though typically huge). Furthermore, we focus primarily on optimization problems that are NP-hard. As our main organizing principle, we restrict our attention to algorithms that are provably good in the following sense: for *any* input, the algorithm runs in time polynomial in the length of the input and returns a solution (i.e., a member of the feasible region) whose value (i.e., objective function value) is guaranteed to be near-optimal in some well-defined sense.[2] Such a guarantee is called the **performance guarantee.** Performance guarantees may be *absolute,* meaning that the additive difference between the optimal value and the value found by the algorithm is bounded. More commonly, performance guarantees are *relative,* meaning that the value found by the algorithm is within a multiplicative factor of the optimal value.

[1] For those unfamiliar with the theory of NP-completeness, see Chapters 27 and 28 or [5].

[2] An alternative to this *worst-case* analysis is *average-case* analysis. See Chapter 14.

When an algorithm with a performance guarantee returns a solution, it has implicitly discovered a bound on the exact optimal value for the problem. Obtaining such bounds is perhaps the most basic challenge in designing approximation algorithms. If one can't compute the optimal value, how can one expect to prove that the output of an algorithm is near it? Three common techniques are what which we shall call **witnesses, relaxation,** and **coarsening.**

Intuitively, a *witness* encodes a short, easily verified proof that the optimal value is at least, or at most, a certain value. Witnesses provide a dual role to **feasible solutions** to a problem. For example, for a maximization problem, where any feasible solution provides a *lower* bound to the optimal value, a witness would provide an *upper* bound on the optimal value. Typically, an approximation algorithm will produce not only a feasible solution, but also a witness. The performance guarantee is typically proven with respect to the two bounds—the upper bound provided by the witness and the lower bound provided by the feasible solution. Since the optimal value is between the two bounds, the performance guarantee also holds with respect to the optimal value.

Relaxation is another way to obtain a lower bound on the minimum value (or an upper bound in the case of a maximization problem). One formulates a new optimization problem, called a relaxation of the original problem, using the same objective function but a larger feasible region $\mathcal{P}'$ that includes $\mathcal{P}$ as a subset. Because $\mathcal{P}'$ contains $\mathcal{P}$, any $x \in \mathcal{P}$ (including the optimal element x) belongs to $\mathcal{P}'$ as well. Hence the optimal value of the relaxation, $\min\{f(x) \mid x \in \mathcal{P}'\}$, is less than or equal to the optimal value of the original optimization problem. The intent is that the optimal value of the relaxation should be easy to calculate and should be reasonably close to the optimal value of the original problem.

Linear programming can provide both witnesses and relaxations, and is therefore an important technique in the design and analysis of approximation algorithms. **Randomized rounding** is a general approach, based on the probabilistic method, for converting a solution to a relaxed problem into an approximate solution to the original problem.

To *coarsen* a problem instance is to alter it, typically restricting to a less complex feasible region or objective function, so that the result problem can be efficiently solved, typically by dynamic programming. For coarsening to be useful, the coarsened problem must approximate the original problem, in that there is a rough correspondence between *feasible solutions* of the two problems, a correspondence that approximately preserves cost. We use the term *coarsening* rather loosely to describe a wide variety of algorithms that work in this spirit.

34.3 Approximation Algorithms with Small Additive Error

Minimum-Degree Spanning Tree

For our first example, consider a slight variant on the minimum-cost spanning tree problem, the *minimum-degree spanning tree problem.* As before, the feasible region $\mathcal{P}$ consists of spanning trees of the input graph, but this time the objective is to find a spanning tree whose *degree* is minimum. The degree of a vertex of a spanning tree (or, indeed, of any graph), is the number of edges incident to that vertex, and the degree of the spanning tree is the maximum of the degrees of its vertices. Thus minimizing the degree of a spanning tree amounts to finding a smallest integer k for which there exists a spanning tree in which each vertex has at most k incident edges.

Any procedure for finding a minimum-degree spanning tree in a graph could be used to find a Hamiltonian path in any graph that has one, for a Hamiltonian path is a degree-two spanning tree. (A Hamiltonian path of a graph is a path through that graph that visits each vertex of the graph exactly once.) Since it is NP-hard even to determine whether a graph has a Hamiltonian path, even determining whether the minimum-degree spanning tree has degree two is presumed to be computationally difficult.

An Approximation Algorithm for Minimum-Degree Spanning Tree

Nonetheless, the minimum-degree spanning-tree problem has a remarkably good approximation algorithm [7, Ch. 7]. For an input graph with m edges and n vertices, the algorithm requires time slightly more than the product of m and n. The output is a spanning tree whose degree is guaranteed to be at most *one more* than the minimum degree. For example, if the graph has a Hamiltonian path, the output is either such a path or a spanning tree of degree three.

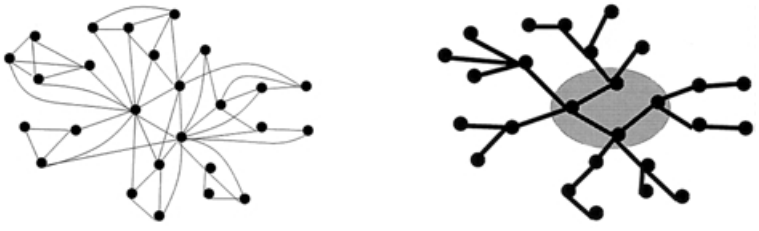


FIGURE 34.1 On the left is an example input graph G. On the right is a spanning tree T that might be found by the approximation algorithm. The shaded circle indicates the nodes in the witness set S.

Given a graph G, the algorithm naturally finds the desired spanning tree T of G. The algorithm also finds a witness—in this case, a set S of vertices proving that T's degree is nearly optimal. Namely, let k denote the degree of T, and let $T_1, T_2, \ldots, T_r$ be the subtrees that would result from T if the vertices of S were deleted. The following two properties are enough to show that T's degree is nearly optimal.

1. There are no edges of the graph G between distinct trees T_i, and
2. The number r of trees T_i is at least $|S|(k-1) - 2(|S|-1)$.

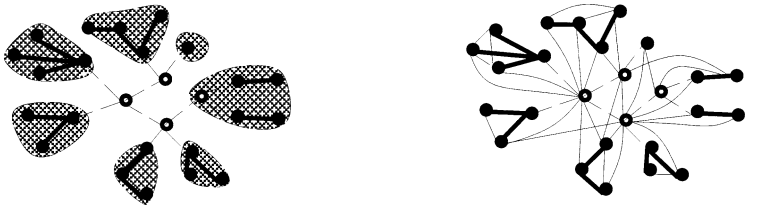


FIGURE 34.2 The figure on the left shows the r trees $T_1, \ldots, T_r$ obtained from T by deleting the nodes of S. Each tree is indicated by a shaded region. The figure on the right shows that no edges of the input graph G connect different trees T_i.

To show that T's degree is nearly optimal, let V_i denote the set of vertices comprising subtree T_i ($i = 1, \ldots, r$). Any spanning tree T^* at all must connect up the sets $V_1, V_2, \ldots, V_r$ and the vertices $y_1, y_2, \ldots, y_{|S|} \in S$, and must use at least $r + |S| - 1$ edges to do so. Furthermore, since no edges go between distinct sets V_i, all these edges must be incident to the vertices of S.

Hence we obtain

$$\begin{aligned} \sum \left\{ \deg_{T^*}(y) \;\middle|\; y \in S \right\} &\geq r + |S| - 1 \\ &\geq |S|(k-1) - 2(|S|-1) + |S| - 1 \\ &= |S|(k-1) - (|S|-1) \end{aligned} \tag{34.2}$$

where $\deg_{T^*}(y)$ denotes the degree of y in the tree T^*. Thus the average of the degrees of vertices in S is at least $\frac{|S|(k-1)-(|S|-1)}{|S|}$, which is strictly greater than $k-2$. Since the average of the degrees of vertices in S is greater than $k-2$, it follows that at least one vertex has degree at least $k-1$.

FIGURE 34.3 The figure on the left shows an arbitrary spanning tree T^* for the same input graph G. The figure on the right has r shaded regions, one for each subset V_i of nodes corresponding to a tree T_i in Fig.34.2. The proof of the algorithm's performance guarantee is based on the observation that at least $r + |S| - 1$ edges are needed to connect up the V_i's and the nodes in S.

We have shown that for every spanning tree T^*, there is at least one vertex with degree at least $k-1$. Hence the minimum degree is at least $k-1$.

We have not explained how the algorithm obtains both the spanning tree T and the set S of vertices, only how the set S shows that the spanning tree is nearly optimal. The basic idea is as follows. Start with any spanning tree T, and let d denote its degree. Let S be the set of vertices having degree d or $d-1$ in the current spanning tree. Let $T_1, \ldots, T_r$ be the subtrees comprising $T - S$. If there are no edges between these subtrees, the set S satisfies property 1 and one can show it also satisfies property 2; in this case the algorithm terminates. If on the other hand there is an edge between two distinct subtrees T_i and T_j, inserting this edge in T and removing another edge from T results in a spanning tree with fewer vertices having degree at least $d-1$. Repeat this process on the new spanning tree; in subsequent iterations the improvement steps are somewhat more complicated but follow the same lines. One can prove that the number of iterations is $O(n \log n)$.

We summarize our brief sketch of the algorithm as follows: either the current set S is a witness to the near-optimality of the current spanning tree T, or there is a slight modification to the set and the spanning tree that improve them. The algorithm terminates after a relatively small number of improvements.

This algorithm is remarkable not only for its simplicity and elegance but also for the quality of the approximation achieved. As we shall see, for most NP-hard optimization problems, we must settle for approximation algorithms that have much weaker guarantees.

Other Problems Having Small-Additive-Error Algorithms

There are a few other natural combinatorial-optimization problems for which approximation algorithms with similar performance guarantees are known. Here are two examples:

Edge Coloring

Given a graph, color its edges with a minimum number of colors so that, for each vertex, the edges incident to that vertex are all different colors. For this problem, it is easy to find a witness. For any graph G, let v be the vertex of highest degree in G. Clearly one needs to assign at least $\deg_G(v)$ colors to the edges of G, for otherwise there would be two edges with the same color incident to v. For any graph G, there is an edge coloring using a number of colors equal to one plus the degree of G. The proof of this fact

translates into a polynomial-time algorithm that approximates the minimum edge-coloring to within an additive error of 1.

Bin Packing

The input consists of a set of positive numbers less than 1. A solution is a partition of the numbers into sets summing to no more than 1. The goal is to minimize the number of sets in the partition. There are approximation algorithms for bin packing that have very good performance guarantees. For example, the performance guarantee for one such algorithm is as follows: for any input set I of item weights, it finds a packing that uses at most $\mathrm{OPT}(I) + O(\log^2 \mathrm{OPT}(I))$ bins, where $\mathrm{OPT}(I)$ is the number of bins used by the best packing, i.e., the optimal value.

34.4 Randomized Rounding and Linear Programming

A *linear programming* problem is any optimization problem in which the feasible region corresponds to assignments of values to variables meeting a set of linear inequalities and in which the objective function is a linear function. An instance is determined by specifying the set of variables, the objective function, and the set of inequalities. **Linear programs** are capable of representing a large variety of problems and have been studied for decades in combinatorial optimization and have a tremendous literature (see e.g., Chapters 31 and 32 of this book). Any linear program can be solved—that is, a point in the feasible region maximizing or minimizing the objective function can be found—in time bounded by a polynomial in the size of the input.

A *(mixed) integer linear programming problem* is a linear programming problem augmented with additional constraints specifying that (some of) the variables must take on integer values. Such constraints make integer linear programming even more general than linear programming—in general, solving **integer linear programs** is NP-hard.

For example, consider the following *balanced matching problem:* The input is a bipartite graph $G = (V, W, E)$. The goal is to choose an edge incident to each vertex in V ($|V|$ edges in total), while minimizing the maximum *load* of (number of chosen edges adjacent to) any vertex in W. The vertices in V might represent tasks, the vertices in W might represent people, while the presence of edge $\{v, w\}$ indicates that person w is competent to perform task v. The problem is then to assign each task to a person competent to perform it, while minimizing the maximum number of tasks assigned to any person.[3]

This balanced matching problem can be formulated as the following integer linear program:

$$\text{minimize } \Delta$$
$$\text{subject to } \begin{cases} \sum_{u \in N(v)} x(u, v) & = & 1 & \forall v \in V \\ \sum_{v \in N(u)} x(u, v) & \leq & \Delta & \forall w \in W \\ x(u, v) & \in & \{0, 1\} & \forall (u, v) \in E\,. \end{cases}$$

Here $N(x)$ denotes the set of neighbors of vertex x in the graph. For each edge (u, v) the variable $x(u, v)$ determines whether the edge (u, v) is chosen. The variable Δ measures the maximum load.

[3] Typically, randomized rounding is applied to NP-hard problems, whereas the balanced matching problem here is actually solvable in polynomial time. We use it as an example for simplicity—the analysis captures the essential spirit of a similar analysis for the well-studied integer multicommodity flow problem. (A simple version of that problem is, "Given a network and a set of commodities (each a pair of vertices), choose a path for each commodity minimizing the maximum congestion on any edge.")

Relaxing the integrality constraints (i.e., replacing them as well as we can by linear inequalities) yields the linear program:

$$\text{minimize } \Delta$$
$$\text{subject to } \begin{cases} \sum_{u \in N(v)} x(u,v) & = & 1 & \forall v \in V \\ \sum_{v \in N(u)} x(u,v) & \leq & \Delta & \forall w \in W \\ x(u,v) & \geq & 0 & \forall (u,v) \in E\ . \end{cases}$$

Rounding a fractional solution to a true solution. This relaxed problem can be solved in polynomial time simply because it is a linear program. Suppose we have an optimal solution x^*, where each $x^*(e)$ is a fraction between 0 and 1. How can we convert such an optimal *fractional* solution into an *approximately optimal* integer solution? *Randomized rounding* is a *general* approach for doing just this [12, Ch. 5].

Consider the following polynomial-time randomized algorithm to find an integer solution $\hat{x}$ from the optimal solution x^* to the linear program:

1. Solve the linear program to obtain a **fractional solution** x^* of load Δ^*.
2. For each vertex $v \in V$:
   (a) Choose a single edge incident to v *at random,* so that the probability that a given edge (u, v) is chosen is $x^*(u, v)$. (Note that $\Sigma_{u \in N(v)} x^*(u, v) = 1$.)
   (b) Let $\hat{x}(u, v) \leftarrow 1$.
   (c) For all other edges (u', v) incident to v, let $\hat{x}(u', v) \leftarrow 0$.

The algorithm will always choose one edge adjacent to each vertex in V. Thus, $\hat{x}$ is a feasible solution to the original integer program. What can we say about the load? For any particular vertex $w \in W$, the load on w is $\Sigma_{u \in N(v)} \hat{x}(u, v)$. For any particular edge $(u, v) \in E$, the probability that $\hat{x}(u, v) = 1$ is $x^*(u, v)$. Thus the *expected* value of the load on a vertex $u \in U$ is $\Sigma_{v \in N(u)} x^*(u, v)$, which is at most Δ^*. This is a good start. Of course, the *maximum* load over all $u \in U$ is likely to be larger. How much larger?

To answer this, we need to know more about the distribution of the load on v than just the expected value. The key fact that we need to observe is that the load on any $v \in V$ is a sum of *independent* $\{0, 1\}$-*random variables.* This means it is not likely to deviate much from its expected value. Precise estimates come from standard bounds, called "Chernoff" or "Hoeffding" bounds, such as the following:

THEOREM 34.1 *Let X be the sum of independent $\{0, 1\}$ random variables. Let $\mu > 0$ be the expected value of X. Then for any $\epsilon > 0$,*

$$\Pr[X \geq (1 + \epsilon)\mu] < \exp\left(-\mu \min\left\{\epsilon, \epsilon^2\right\}/3\right)\ .$$

(See, e.g.,[12, Ch. 4.1].) This is enough to analyze the performance guarantee of the algorithm. It is slightly complicated, but not too bad:

Claim *With probability at least 1/2, the maximum load induced by $\hat{x}$ exceeds the optimal by at most an additive error of*

$$\max\left\{3\ln(2m), \sqrt{3\ln(2m)\Delta^*}\right\}\ ,$$

where $m = |W|$.

Proof sketch: As observed previously, for any particular v, the load on v is a sum (of independent random $\{0, 1\}$-variables) with expectation bounded by Δ^*. Let ϵ be just large enough so that $\exp(-\Delta^* \min\{\epsilon, \epsilon^2\}/3) = 1/(2m)$. By the Chernoff-type bound above, the probability that the load

exceeds $(1+\epsilon)\Delta^*$ is then less than $1/(2m)$. Thus, by the naive union bound,[4] the probability that the *maximum* load on any $v \in V$ is more than $\Delta^*(1+\epsilon) = \Delta^* + \epsilon\Delta^*$ is less then $1/2$. We leave it to the reader to verify that the choice of ϵ makes $\epsilon\Delta^*$ equal the expression in the statement of the claim.

Summary. This is the general randomized-rounding recipe:

1. Formulate the original NP-hard problem as an integer linear programming problem (IP).
2. Relax the program IP to obtain a linear program (LP).
3. Solve the linear program, obtaining a fractional solution.
4. Randomly round the fractional solution to obtain an approximately optimal integer solution.

34.5 Performance Ratios and ρ-Approximation

Relative (multiplicative) performance guarantees are more common than absolute (additive) performance guarantees. One reason is that many NP-hard optimization problems are **rescalable:** given an instance of the problem, one can construct a new, equivalent instance by scaling the objective function. For instance, the traveling salesman problem is rescalable—given an instance, multiplying the edge weights by any $\lambda > 0$ yields an equivalent problem with the objective function scaled by λ. For rescalable problems, the best one can hope for is a relative performance guarantee [15].

A ρ-**approximation algorithm** is an algorithm that returns a feasible solution whose objective function value is at most ρ times the minimum (or, in the case of a maximization problem, the objective function value is at least ρ times the maximum). We say that the **performance ratio** of the algorithm is ρ.[5]

34.6 Polynomial Approximation Schemes

The *knapsack problem* is an example of a rescalable NP-hard problem. An instance consists of a set of pairs of numbers $(\text{weight}_i, \text{profit}_i)$, and the goal is to select a subset of pairs for which the sum of weights is at most 1 so as to maximize the sum of profits. (Which items should one put in a knapsack of capacity 1 so as to maximize profit?)

Since the knapsack problem is rescalable and NP-hard, we assume that there is no approximation algorithm achieving, say, a fixed absolute error. One is therefore led to ask, what is the best performance ratio achievable by a polynomial-time approximation algorithm? In fact (assuming P $\neq$ NP), there is no such *best* performance ratio: for any given $\epsilon > 0$, there is a polynomial approximation algorithm whose performance ratio is $1+\epsilon$. The smaller the value of ϵ, however, the greater the running time of the corresponding approximation algorithm. Such a collection of approximation algorithms, one for each $\epsilon > 0$, is called a *(polynomial) approximation scheme.*

Think of an approximation scheme as an algorithm that takes an additional parameter, the value of ϵ, in addition to the input specifying the instance of some optimization problem. The running time of this algorithm is bounded in terms of the size of the input and in terms of ϵ. For example, there is an approximation scheme for the knapsack problem that requires time $O(n\log(1/\epsilon) + 1/\epsilon^4)$ for instances with n items. Below we sketch a much simplified version of this algorithm that requires time $O(n^3/\epsilon)$. The algorithm works by coarsening.

[4] The probability that any of several events happens is at most the sum of the probabilities of the individual events.

[5] This terminology is the most frequently used, but one also finds alternative terminology in the literature. Confusingly, some authors have used the term $1/\rho$-approximation algorithm or $(1-\rho)$-approximation algorithm to refer to what we call a ρ-approximation algorithm.

The algorithm is given the pairs $(\text{weight}_1, \text{profit}_1), \ldots, (\text{weight}_n, \text{profit}_n)$, and the parameter ϵ. We assume without loss of generality that each weight is less than or equal to 1. Let $\text{profit}_{\max} = \max_i \text{profit}_i$. Let OPT denote the (unknown) optimal value. Since the item of greatest profit itself constitutes a solution, albeit not usually a very good one, we have $\text{profit}_{\max} \leq \text{OPT}$. In order to achieve a relative error of at most ϵ, therefore, it suffices to achieve an absolute error of at most $\epsilon\ \text{profit}_{\max}$.

We transform the given instance into a coarsened instance by rounding each profit down to a multiple of $K = \epsilon\ \text{profit}_{\max}/n$. In so doing, we reduce each profit by less than $\epsilon\ \text{profit}_{\max}/n$. Consequently, since the optimal solution consists of no more than n items, the profit of this optimal solution is reduced by less than $\epsilon\ \text{profit}_{\max}$ in total. Thus, the optimal value for the coarsened instance is at least $\text{OPT} - \epsilon\ \text{profit}_{\max}$, which is in turn at least $(1-\epsilon)\text{OPT}$. The corresponding solution, when measured according to the original profits, has value at least this much. Thus we need only solve the coarsened instance optimally in order to get a performance guarantee of $1 - \epsilon$.

Before addressing the solution of the coarsened instance, note that the optimal value is the sum of at most n profits, each at most $\text{profit}_{\max}$. Thus $\text{OPT} \leq n^2 K/\epsilon$. The optimal value for the coarsened instance is therefore also at most $n^2 K/\epsilon$.

To solve the coarsened instance optimally, we use dynamic programming. Note that for the coarsened instance, each achievable total profit can be written as $i \cdot K$ for some integer $i \leq n^2/\epsilon$. The dynamic-programming algorithm constructs an $\lceil n^2/\epsilon \rceil \times (n+1)$ table $T[i, j]$ whose i, j entry is the minimum weight required to achieve profit $i \cdot K$ using a subset of the items 1 through j. The entry is infinity if there is no such way to achieve that profit.

To fill in the table, the algorithm initializes the entries $T[i, 0]$ to infinity, then executes the following step for $j = 1, 2, \ldots, n$:

$$\text{For each } i\text{, set } T[i, j] := \min\left\{T[i, j-1], \text{weight}_j + T\left[i - \left(\widehat{\text{profit}}_j/K\right), j-1\right]\right\}$$

where $\widehat{\text{profit}}_j$ is the profit of item j in the rounded-down instance. A simple induction on j shows that the calculated values are correct. The optimal value for the coarsened instance is

$$\widehat{\text{OPT}} = \max\{iK \mid T[i, n] \leq 1\}\ .$$

The above calculates the optimal *value* for the coarsened instance; as usual in dynamic programming, a corresponding feasible solution can easily be computed if desired.

Other Problems Having Polynomial Approximation Schemes

The running time of the knapsack approximation scheme depends polynomially on $1/\epsilon$. Such a scheme is called a *fully polynomial* approximation scheme. Most natural NP-complete optimization problems are **strongly NP-hard,** meaning essentially that the problems are NP-hard even when the numbers appearing in the input are restricted to be no larger in magnitude than the size of the input. For such a problem, we cannot expect a **fully polynomial approximation scheme** to exist [5, Section 4.2]. On the other hand, a variety of NP-hard problems in fixed-dimensional Euclidean space have approximation schemes. For instance, given a set of points in the plane:

Covering with Disks: Find a minimum set of area-1 disks (or squares, etc.) covering all the points [7, Section 9.3.3].

Euclidean Traveling Salesman: Find a closed loop passing through each of the points and having minimum total arc length [1].

Euclidean Steiner Tree: Find a minimum-length set of segments connecting up all the points [1].

Similarly, many problems in planar graphs or graphs of fixed genus can have **polynomial approximation schemes** [7, Section 9.3.3], For instance, **given a planar graph with weights assigned to its vertices:**

Maximum-Weight Independent Set: Find a maximum-weight set of vertices, no two of which are adjacent.

Minimum-Weight Vertex Cover: Find a minimum-weight set of vertices such that every edge is incident to at least one of the vertices in the set.

The above algorithms use relatively more sophisticated and varied coarsening techniques.

34.7 Constant-Factor Performance Guarantees

We have seen that, assuming P $\neq$ NP, rescalable NP-hard problems do not have polynomial-time approximation algorithms with small absolute errors but may have fully polynomial approximation schemes, while strongly NP-hard problems do not have fully polynomial approximation schemes but may have polynomial approximation schemes. Further, there is a class of problems that do not have approximation schemes: for each such problem there is a constant c such that any polynomial-time approximation algorithm for the problem has relative error at least c (assuming P $\neq$ NP). For such a problem, the best one can hope for is an approximation algorithm with constant performance ratio.

Our example of such a problem is the *vertex cover* problem: given a graph G, find a minimum-size set C (a *vertex cover*) of vertices such that every edge in the graph is incident to some vertex in C. Here the feasible region $\mathcal{P}$ consists of the vertex covers in G, while the objective function is the size of the cover. Here is a simple approximation algorithm [7]:

1. Find a maximal independent set S of edges in G.
2. Let C be the vertices incident to edges in S.

(A set S of edges is *independent* if no two edges in S share an endpoint. The set S is *maximal* if no larger independent set contains S.) The reader may wish to verify that the set S can be found in linear time, and that because S is maximal, C is necessarily a cover.

What performance guarantee can we show? Since the edges in S are independent, *any* cover must have at least one vertex for each edge in S. Thus S is a witness proving that any cover has at least $|S|$ vertices. On the other hand, the cover C has $2|S|$ vertices. Thus the cover returned by the algorithm is at most *twice* the size of the optimal vertex cover.

The Weighted Vertex Cover Problem. The *weighted* vertex cover problem is a generalization of the vertex cover problem. An instance is specified by giving a graph $G = (V, E)$ and, for each vertex v in the graph, a number $\mathrm{wt}(v)$ called its *weight.* The goal is to find a vertex cover minimizing the total weight of the vertices in the cover. Here is one way to represent the problem as an integer linear program:

$$\begin{aligned} \text{minimize} \quad & \sum_{v \in V} \mathrm{wt}(v) x(v) \\ \text{subject to} \quad & \begin{cases} x(u) + x(v) \geq 1 & \forall \{u, v\} \in E \\ x(v) \in \{0, 1\} & \forall v \in V\,. \end{cases} \end{aligned}$$

There is one $\{0, 1\}$-variable $x(v)$ for each vertex v representing whether v is in the cover or not, and there are constraints for the edges that model the covering requirement. The feasible region of this program corresponds to the set of vertex covers. The objective function corresponds to the total weight of the vertices in the cover. Relaxing the integrality constraints yields

$$\begin{aligned} \text{minimize} \quad & \sum_{v \in V} \mathrm{wt}(v) x(v) \\ \text{subject to} \quad & \begin{cases} x(u) + x(v) \geq 1 & \forall \{u, v\} \in E \\ x(v) \geq 0 & \forall v \in V\,. \end{cases} \end{aligned}$$

This relaxed problem is called the *fractional* weighted vertex cover problem; feasible solutions to it are called *fractional* vertex covers.[6]

Rounding a Fractional Solution to a True Solution. By solving this linear program, an optimal fractional cover can be found in polynomial time. For this problem, it is possible to convert a fractional cover into an approximately optimal true cover by rounding the fractional cover in a simple way:

1. Solve the linear program to obtain an optimal fractional cover x^*.
2. Let $C = \left\{v \in V : x^*(v) \geq \frac{1}{2}\right\}$.

The set C is a cover because for any edge, at least one of the endpoints must have fractional weight at least $1/2$. The reader can verify that the total weight of vertices in C is at most twice the total weight of the fractional cover x^*. Since the fractional solution was an optimal solution to a relaxation of the original problem, this is a 2-approximation algorithm [7].

For most problems, this simple kind of rounding is not sufficient. The previously discussed technique called *randomized rounding* is more generally useful.

Primal-Dual Algorithms—Witnesses Via Duality. For the purposes of approximation, solving a linear program *exactly* is often unnecessary. One can often design a faster algorithm based on the witness technique, using the fact that *every linear program has a well-defined notion of "witness."* The witnesses for a linear program P are the feasible solutions to another related linear program called the *dual* of P.

Suppose our original problem is a minimization problem. Then for each point y in the feasible region of the dual problem, the value of the objective function at y is a lower bound on the value of the optimal value of the original linear program. That is, any feasible solution to the dual problem is a possible witness—both for the original integer linear program and its relaxation. For the weighted vertex cover problem, the dual is the following:

$$\text{maximize} \sum_{e \in E} y(e)$$

$$\text{subject to} \begin{cases} \sum_{e \ni v} y(e) \leq \text{wt}(v) & \forall v \in V \\ y(e) \geq 0 & \forall e \in E\,. \end{cases}$$

A feasible solution to this linear program is called an *edge packing.* The constraints for the vertices are called *packing constraints.*

Recall the original approximation algorithm for the unweighted vertex cover problem: find a maximal independent set of edges S; let C be the vertices incident to edges in S. In the analysis, the set S was the witness.

Edge packings generalize independent sets of edges. This observation allows us to generalize the algorithm for the unweighted problem. Say an edge packing is *maximal* if, for every edge, one of the edge's vertices has its packing constraint met. Here is the algorithm:

1. Find a maximal edge packing y.
2. Let C be the vertices whose packing constraints are tight for y.

The reader may wish to verify that a maximal edge packing can easily be found in linear time and that the set C is a cover because y is maximal.

---

[6]The reader may wonder whether additional constraints of the form $x(v) \leq 1$ are necessary. In fact, assuming the vertex weights are nonnegative, there is no incentive to make any $x(v)$ larger than 1, so such constraints would be redundant.

What about the performance guarantee? Since only vertices whose packing constraints are tight are in C, and each edge has only two vertices, we have

$$\sum_{v \in C} \mathrm{wt}(v) = \sum_{v \in C} \sum_{e \ni v} y(e) \leq 2 \sum_{e \in E} y(e) .$$

Since y is a solution to the dual, $\Sigma_e y(e)$ is a lower bound on the weight of any vertex cover, fractional or otherwise. Thus, the algorithm is a 2-approximation algorithm.

Summary. This is the general primal-dual recipe:

1. Formulate the original NP-hard problem as an integer linear programming problem (IP).
2. Relax the program IP to obtain a linear program (LP).
3. Use the dual (DLP) of LP as a source of witnesses.

Beyond these general guidelines, the algorithm designer is still left with the task of figuring out *how* to find a good solution and witness. See [7, Ch. 4] for an approach that works for a wide class of problems.

Other Optimization Problems with Constant-Factor Approximations

Constant-factor approximation algorithms are known for problems from many areas. In this section, we describe a sampling of these problems. For each of the problems described here, there is no polynomial approximation scheme (unless P = NP); thus constant-factor approximation algorithms are the best we can hope for. For a typical problem, there will be a simple algorithm achieving a small constant factor while there may be more involved algorithms achieving better factors. The factors known to be achievable typically come close to, but do not meet, the best lower bounds known (assuming P $\neq$ NP).

For the problems below, we omit discussion of the techniques used; many of the problems are solved using a relaxation of some form, and (possibly implicitly) the primal-dual recipe. Many of these problems have polynomial approximation schemes if restricted to graphs induced by points in the plane or constant-dimensional Euclidean space (see "Other Problems Having Polynomial Approximation Schemes").

MAX-SAT: Given a propositional formula in conjunctive normal form (an "and" of "or"'s of possibly negated Boolean variables), find a truth assignment to the variables that maximizes the number of clauses (groups of "or"'ed variables in the formula) that are true under the assignment. A variant called MAX-3SAT restricts to the formula to have three variables per clause. MAX-3SAT is a canonical example of a problem in the complexity class **MAX-SNP** [7, Section 10.3].

MAX-CUT: Given a graph, partition the vertices of the input graph into two sets so as to maximize the number of edges with endpoints in distinct sets. For MAX-CUT and MAX-SAT problems, the best approximation algorithms currently known rely on randomized rounding and a generalization of linear programming called **semidefinite programming** [7, Section 11.3].

Shortest Superstring: Given a set of strings $\sigma_1, \ldots, \sigma_k$, find a minimum-length string containing all σ_i's. This problem has applications in computational biology [3, 10].

K-Cluster: Given a graph with weighted edges and given a parameter k, partition the vertices into k clusters so as to minimize the maximum distance between any two vertices in the same cluster. For this and related problems see [7, Section 9.4].

Traveling Salesman: Given a complete graph with edge weights satisfying the **triangle inequality,** find a minimum-length path that visits every vertex of the graph [7, Ch. 8].

Edge and Vertex Connectivity: Given a weighted graph $G = (V, E)$ and an integer k, find a minimum-weight edge set $E' \subseteq E$ such that between any pair of vertices, there are k edge-disjoint paths in the graph $G' = (V, E')$. Similar algorithms handle the goal of k *vertex-*

disjoint paths and the goal of *augmenting* a given graph to achieve a given connectivity [7, Ch. 6]

Steiner Tree: Given an undirected graph with positive edge-weights and a subset of the vertices called *terminals,* find a minimum-weight set of edges through which all the terminals (and possibly other vertices) are connected [7, Ch. 8]. The Euclidean version of the problem is "Given a set of points in $\mathbb{R}^n$, find a minimum-total-length union of line segments (with arbitrary endpoints) that is connected and contains all the given points."

Steiner Forest: Given a weighted graph and a collection of *groups* of terminals, find a minimum-weight set of edges through which every pair of terminals within each group are connected [7, Ch. 4]. The algorithm for this problem is based on a primal-dual framework that has been adapted to a wide variety of network design problems. See Section "Other Problems Having Poly-Logarithmic Performance Guarantees."

34.8 Logarithmic Performance Guarantees

When a constant-ratio performance guarantee is not possible, a slowly-growing ratio is the next best thing. The canonical example of this is the *set cover* problem: Given a family of sets $\mathcal{F}$ over a universe $\mathcal{U}$, find a minimum-cardinality *set cover* C—a collection of the sets that collectively contain all elements in $\mathcal{U}$. In the weighted version of the problem, each set also has a weight and the goal is to find a set cover of minimum total weight. This problem is important due to its generality. For instance, it generalizes the vertex cover problem.

Here is a simple greedy algorithm:

1. Let $C \leftarrow \emptyset$.
2. Repeat until all elements are covered: add a set S to C maximizing

$$\frac{\text{the number of elements in } S \text{ not in any set in } C}{\text{wt}(S)} .$$

3. Return C.

The algorithm has the following performance guarantee [7, Section 3.2]:

THEOREM 34.2 *The greedy algorithm for the weighted set cover problem is an H_s-approximation algorithm, where s is the maximum size of any set in $\mathcal{F}$.*

By definition $H_s = 1 + \frac{1}{2} + \frac{1}{3} + \cdot + \frac{1}{s}$; also, $H_s \leq 1 + \ln s$.

We will give a direct argument for the performance guarantee and then relate it to the general primal-dual recipe. Imagine that as the algorithm proceeds, it assigns *charges* to the elements as they are covered. Specifically, when a set S is added to the cover C, if there are k elements in S not previously covered, assign each such element a charge of $\text{wt}(S)/k$. Note that *the total charge assigned over the course of the algorithm equals the weight of the final cover C.*

Next we argue that *the total charge assigned over the course of the algorithm is a lower bound on H_s times the weight of the optimal vertex cover.* These two facts together prove the theorem.

Suppose we could prove that for any set T in the optimal cover C^*, the elements in T are assigned a total charge of at most $\text{wt}(T)H_s$. Then we would be done, because every element is in at least one set in the optimal cover:

$$\sum_{i \in U} charge(i) \leq \sum_{T \in C^*} \sum_{i \in T} charge(i) \leq \sum_{T \in C^*} \text{wt}(T) H_s .$$

So, consider, for example, a set $T = \{a, b, c, d, e, f\}$ with $\text{wt}(T) = 3$. For convenience, assume that the greedy algorithm covers elements in T in alphabetical order. What can we say about the charge assigned to a? Consider the iteration when a was first covered and assigned a charge. At the beginning of that iteration, T was not yet chosen and none of the 6 elements in T were yet covered. Since the greedy algorithm had the option of choosing T, whatever set it did choose resulted in a charge to a of at most $\text{wt}(T)/|T| = 3/6$.

What about the element b? When b was first covered, T was not yet chosen, and at least 5 elements in T remained uncovered. Consequently, the charge assigned to b was at most $3/5$. Reasoning similarly, the elements c, d, e, and f were assigned charges of at most $3/4$, $3/3$, $3/2$, and $3/1$, respectively. The total charge to elements in T is at most

$$3 \times (1/6 + 1/5 + 1/4 + 1/3 + 1/2 + 1/1) = \text{wt}(T)H_{|T|} \leq \text{wt}(T)H_s \ .$$

This line of reasoning easily generalizes to show that for any set T, the elements in T are assigned a total charge of at most $\text{wt}(T)H_s$.

Underlying Duality. What role does duality and the primal-dual recipe play in the above analysis? A natural integer linear program for the weighted set cover problem is

$$\begin{aligned} &\text{minimize} \sum_{S \in \mathcal{F}} \text{wt}(S)x(S) \\ &\text{subject to} \begin{cases} \sum_{S \ni i} x(S) & \geq & 1 & \forall i \in \mathcal{U} \\ x(S) & \in & \{0, 1\} & \forall S \in \mathcal{F} \ . \end{cases} \end{aligned}$$

Relaxing this integer linear program yields the linear program

$$\begin{aligned} &\text{minimize} \sum_{S \in \mathcal{F}} \text{wt}(S)x(S) \\ &\text{subject to} \begin{cases} \sum_{S \ni i} x(S) & \geq & 1 & \forall i \in \mathcal{U} \\ x(S) & \geq & 0 & \forall S \in \mathcal{F} \ . \end{cases} \end{aligned}$$

A solution to this linear program is called a *fractional* set cover. The dual is

$$\begin{aligned} &\text{minimize} \sum_{i \in \mathcal{U}} y(i) \\ &\text{subject to} \begin{cases} \sum_{i \in S} y(i) & \leq & \text{wt}(S) & \forall S \in \mathcal{F} \\ y(i) & \geq & 0 & \forall i \in \mathcal{U} \ . \end{cases} \end{aligned}$$

The inequalities for the sets are called *packing constraints.* A solution to this **dual linear program** is called an *element packing.* In fact, the "charging" scheme in the analysis is just an element packing y, where $y(i)$ is the charge assigned to i divided by H_s. In this light, the previous analysis is simply constructing a dual solution and using it as a witness to show the performance guarantee.

Other Problems Having Poly-Logarithmic Performance Guarantees

Minimizing a Linear Function Subject to a Submodular Constraint. This is a natural generalization of the weighted set cover problem. Rather than state the general problem, we give the following special case as an example: Given a family $\mathcal{F}$ of sets of n-vectors, with each set in $\mathcal{F}$ having a cost, find a subfamily of sets of minimum total cost whose union has rank n. A natural generalization of the greedy set cover algorithm gives a logarithmic performance guarantee [13].

Vertex-Weighted Network Steiner Tree. Like the network Steiner tree problem described in "Other Optimization Problems With Constant-Factor Approximations," an instance consists of a graph and a set of terminals; in this case, however, the graph can have vertex weights in addition to edge weights. An adaptation of the greedy algorithm achieves a logarithmic performance ratio.

Network Design Problems. This is a large class of problems generalizing the Steiner forest problem (see "Other Optimization Problems With Constant-Factor Approximations"). An example of a problem in this class is *survivable network design:* given a weighted graph $G = (V, E)$ and a nonnegative integer r_{uv} for each pair of vertices, find a minimum-cost set of edges $E' \subseteq E$ such that for every pair of vertices u and v, there are at least r_{uv} edge-disjoint paths connecting u and v in the graph $G = (V, E')$. A primal-dual approach, generalized from an algorithm for the Steiner forest problem, yields good performance guarantees for problems in this class. The performance guarantee depends on the particular problem; in some cases it is known to be bounded only logarithmically [7, Ch. 4]. For a commercial application of this work see [11].

Graph Bisection. Given a graph, partition the nodes into two sets of equal size so as to minimize the number of edges with endpoints in different sets. An algorithm to find an approximately minimum-weight bisector would be remarkably useful, since it would provide the basis for a divide-and-conquer approach to many other graph optimization problems. In fact, a solution to a related but easier problem suffices.

Define a $\frac{1}{3}$-*balanced cut* to be a partition of the vertices of a graph into two sets each containing at least one-third of the vertices; its weight is the total weight of edges connecting the two sets. There is an algorithm to find a $\frac{1}{3}$-balanced cut whose weight is $O(\log n)$ times the minimum weight of a bisector. Note that this algorithm is not, strictly speaking, an approximation algorithm for any one optimization problem: the output of the algorithm is a solution to one problem while the quality of the output is measured against the optimal value for another. (We call this kind of performance guarantee a "bait-and-switch" guarantee.) Nevertheless, the algorithm is nearly as useful as a true approximation algorithm would be because in many divide-and-conquer algorithms the precise balance is not critical. One can make use of the balanced-cut algorithm to obtain approximation algorithms for many problems, including the following.

Optimal Linear Arrangement. Assign vertices of a graph to distinct integral points on the real number line so as to minimize the total length of edges.

Minimizing Time and Space for Sparse Gaussian Elimination. Given a sparse, positive-semidefinite linear system, the order in which variables are eliminated affects the time and storage space required for solving the system; choose an ordering to simultaneously minimize both time and storage space required.

Crossing Number. Embed a graph in the plane so as to minimize the number of edge-crossings.

The approximation algorithms for the above three problems have performance guarantees that depend on the performance guarantee of the balanced-separator algorithm. It is not known whether the latter performance guarantee can be improved: there might be an algorithm for balanced separators that has a constant performance ratio.

There are several other graph-separation problems for which approximation algorithms are known, e.g., problems involving directed graphs. All these approximation algorithms for cut problems make use of linear-programming relaxation. See [7, Ch. 5].

34.9 Multi-Criteria Problems

In many applications, there are two or more objective functions to be considered. There have been some approximation algorithms developed for such *multi-criteria* optimization problems (though much work remains to be done). Several problems in previous sections, such as the k-cluster problem described in "Other Optimization Problems With Constant-Factor Approximations," can be viewed as a bi-criteria problem: there is a budget imposed on one resource (the number of clusters), and the algorithm is required to approximately optimize use of another resource (cluster diameter) subject to that budget constraint. Another example is scheduling unrelated parallel machines with costs: for a given budget on cost, jobs are assigned to machines in such a way that the cost of the assignment is under budget and the makespan of the schedule is nearly minimum.

Other approximation algorithms for bi-criteria problems use the bait-and-switch idea mentioned in "Other Problems Having Poly-Logarithmic Performance Guarantees." For example, there is a polynomial approximation scheme for variant of the minimum-spanning-tree problem in which there are two unrelated costs per edge, say weight and length: given a budget L on length, the algorithm finds a spanning tree whose length is at most $(1 + \epsilon)L$ and whose weight is no more than the minimum weight of a spanning tree having length at most L [14].

34.10 Hard-to-Approximate Problems

For some optimization problems, worst-case performance guarantees are unlikely to be possible: it is NP-hard to approximate these problems even if one is willing to accept very poor performance guarantees. Following are some examples [7, Sections 10.5, 10.6].

Maximum Clique. Given a graph, find a largest set of vertices that are pairwise adjacent (see also [6]).

Minimum Vertex Coloring. Given a graph, color the vertices with a minimum number of colors so that adjacent vertices receive distinct colors.

Longest Path. Given a graph, find a longest simple path.

Max Linear Satisfy. Given a set of linear equations, find a largest possible subset that are simultaneously satisfiable.

Nearest Codeword. Given a linear error-correcting code specified by a matrix, and given a vector, find the codeword closest in Hamming distance to the vector.

Nearest Lattice Vector. Given a set of vectors $v_1, \ldots, v_n$ and a vector v, find an integer linear combination of the v_i that is nearest in Euclidean distance to v.

34.11 Research Issues and Summary

We have given examples for the techniques most frequently used to obtain approximation algorithms with provable performance guarantees, the use of witnesses, relaxation, and coarsening. We have categorized NP-hard optimization problems according to the performance guarantees achievable in polynomial time:

1. A small additive error,
2. A relative error of ϵ for any fixed positive ϵ,
3. A constant-factor performance guarantee,
4. A logarithmic- or polylogarithmic-factor performance guarantee,
5. No significant performance guarantee.

The ability to categorize problems in this way has been greatly aided by recent research developments in complexity theory. Novel techniques have been developed for proving the hardness of approximation of optimization problems. For many fundamental problems, we can state with considerable precision how good a performance guarantee can be achieved in polynomial time: known lower and upper bounds match or nearly match. Research toward proving matching bounds continues. In particular, for several problems for which there are logarithmic-factor performance guarantees (e.g., balanced cuts in graphs), researchers have so far not ruled out the existence of constant-factor performance guarantees.

Another challenge in research is methodological in nature. This chapter has presented methods of *worst-case analysis:* ways of universally bounding the error (relative or absolute) of an approximation algorithm. This theory has led to the development of many interesting and useful algorithms, and has proved useful in making distinctions between algorithms and between optimization problems. However, worst-case bounds are clearly not the whole story. Another approach is to develop algorithms tuned for a particular probability distribution of inputs, e.g., the uniform distribution. This approach is of limited usefulness because the distribution of inputs arising in a particular application rarely matches that for which the algorithm was tuned. Perhaps the most promising approach would address a hybrid of the worst-case and probabilistic models. The performance of an approximation algorithm would be defined as the probabilistic performance on a probability distribution selected by an adversary from among a large class of distributions. Blum [2] has presented an analysis of this kind in the context of graph coloring, and others (see [7, Section 13.7]) have addressed similar issues in the context of on-line algorithms.

34.12 Defining Terms

ρ-Approximation algorithm: An approximation algorithm that is guaranteed to find a solution whose value is at most (or at least, as appropriate) ρ times the optimum. The ratio ρ is the *performance ratio* of the algorithm.

Absolute performance guarantee: An approximation algorithm with an **absolute performance guarantee** is guaranteed to return a feasible solution whose value differs additively from the optimal value by a bounded amount.

Approximation algorithm: For solving an optimization problem. An algorithm that runs in time polynomial in the length of the input and outputs a feasible solution that is guaranteed to be nearly optimal in some well-defined sense called the *performance guarantee.*

Coarsening: To *coarsen* a problem instance is to alter it, typically restricting to a less complex feasible region or objective function, so that the resulting problem can be efficiently solved, typically by dynamic programming. This is not standard terminology.

Dual linear program: Every linear program has a corresponding linear program called the dual. For the linear program under **linear program,** the dual is $\max_y \{b \cdot y : A^T y \leq c \text{ and } y \geq \overline{0}\}$. For any solution x to the original linear program and any solution y to the dual, we have $c \cdot x \geq (A^T y)^T x = y^T (Ax) \geq y \cdot b$. For optimal x and y, equality holds. For a problem formulated as an integer linear program, feasible solutions to the dual of a relaxation of the program can serve as witnesses.

Feasible region: See **optimization problem.**

Feasible solution: Any element of the feasible region of an optimization problem.

Fractional solution: Typically, a solution to a relaxation of a problem.

Fully polynomial approximation scheme: An approximation scheme in which the running time of A_ϵ is bounded by a polynomial in the length of the input and $1/\epsilon$.

Integer linear program: A linear program augmented with additional constraints specifying that the variables must take on integer values. Solving such problems is NP-hard.

Linear program: A problem expressible in the following form. Given an $n \times m$ real matrix A, m-vector b and n-vector c, determine $\min_x \left\{c \cdot x : Ax \geq b \text{ and } x \geq \overline{0}\right\}$ where x ranges over all n-vectors and the inequalities are interpreted component-wise (i.e., $x \geq \overline{0}$ means that the entries of x are nonnegative).

MAX-SNP: A complexity class consisting of problems that have constant-factor approximation algorithms, but no approximation schemes unless P = NP.

Mixed integer linear program: A linear program augmented with additional constraints specifying that some of the variables must take on integer values. Solving such problems is NP-hard.

Objective function: See **optimization problem.**

Optimal solution: To an optimization problem. A feasible solution minimizing (or possibly maximizing) the value of the objective function.

Optimal value: The minimum (or possibly maximum) value taken on by the objective function over the feasible region of an optimization problem.

Optimization problem: An optimization problem consists of a set $\mathcal{P}$, called the *feasible region* and usually specified implicitly, and a function $f : \mathcal{P} \to \mathbb{R}$, the *objective function.*

Performance guarantee: See **approximation algorithm.**

Performance ratio: See **ρ-approximation algorithm.**

Polynomial approximation scheme: A collection of algorithms $\{A_\epsilon : \epsilon > 0\}$, where each A_ϵ is a $(1 + \epsilon)$-approximation algorithm running in time polynomial in the length of the input. There is no restriction on the dependence of the running time on ϵ.

Randomized rounding: A technique that uses the probabilistic method to convert a solution to a relaxed problem into an approximate solution to the original problem.

Relative performance guarantee: An approximation algorithm with a **relative performance guarantee** is guaranteed to return a feasible solution whose value is bounded by a multiplicative factor times the optimal value.

Relaxation: A relaxation of an optimization problem with feasible region $\mathcal{P}$ is another optimization problem with feasible region $\mathcal{P}' \supset \mathcal{P}$ and whose objective function is an extension of the original problem's objective function. The relaxed problem is typically easier to solve. Its value provides a bound on the value of the original problem.

Rescalable: An optimization problem is rescalable if, given any instance of the problem and integer $\lambda > 0$, there is an easily computed second instance that is the same except that the objective function for the second instance is (element-wise) λ times the objective function of the first instance. For such problems, the best one can hope for is a multiplicative performance guarantee, not an absolute one.

Semidefinite programming: A generalization of linear programming in which any subset of the variables may be constrained to form a semidefinite matrix. Used in recent results obtaining better approximation algorithms for cut, satisfiability, and coloring problems.

Strongly NP-hard: A problem is strongly NP-hard if it is NP-hard even when any numbers appearing in the input are bounded by some polynomial in the length of the input.

Triangle inequality: A complete weighted graph satisfies the triangle inequality if $\text{wt}(u, v) \leq \text{wt}(u, w) + \text{wt}(w, v)$ for all vertices u, v, and w. This will hold for any graph representing points in a metric space. Many problems involving edge-weighted graphs have better approximation algorithms if the problem is restricted to weights satisfying the triangle inequality.

Witness: A structure providing an easily verified bound on the optimal value of an optimization problem. Typically used in the analysis of an approximation algorithm to prove the performance guarantee.

References

[1] Arora, S., Polynomial time approximation scheme for Euclidean TSP and other geometric problems. In [8], 2–11, 1996.

[2] Blum, A., *Algorithms for Approximate Graph Coloring.* Ph.D. Thesis, Massachusetts Institute of Technology. MIT Laboratory for Computer Science Technical Report MIT/LCS/TR-506, Jun. 1991.

[3] Blum, A., Jiang, T., Li, M., Tromp, J., and Yannakakis, M., Linear approximation of shortest superstrings. *Journal of the ACM,* 41(4), 630–647, 1994.

[4] Crescenzi, P. and Kann, V., A compendium of NP optimization problems, 1995. http://www.nada.kth.se/nada/theory/problemlist.html.

[5] Garey, M.R. and Johnson, D.S., *Computers and Intractibility: A Guide to the Theory of NP-Completeness.* W.H. Freeman, New York, 1979.

[6] Håstad, J., Clique is hard to approximate within $n^{1-\epsilon}$. In [8], 627–636, 1996.

[7] Hochbaum, D.S., Ed., *Approximation Algorithms for NP-Hard Problems.* PWS Publishing, 1997.

[8] IEEE, *37th Annual Symposium on Foundations of Computer Science,* Burlington, VT, 1996.

[9] Johnson, D.S., Approximation algorithms for combinatorial problems. *Journal of Computer and System Sciences,* 9, 256–278, 1974.

[10] Li, M., Towards a DNA sequencing theory (learning a string) (preliminary version). In *31st Annual Symposium on Foundations of Computer Science,* volume I, 125–134, St. Louis, MO, IEEE, 1990.

[11] Mihail, M., Shallcross, D., Dean, N., and Mostrel, M., A commercial application of survivable network design: ITP/INPLANS CCS network topology analyzer. In *Proceedings of the Seventh Annual ACM-SIAM Symposium on Discrete Algorithms,* 279–287, Atlanta, GA, 1996.

[12] Motwani, R. and Raghavan, P., *Randomized Algorithms.* Cambridge University Press, 1995.

[13] Nemhauser, G.L. and Wolsey, L.A., *Integer and Combinatorial Optimization.* John Wiley & Sons, New York, 1988.

[14] Ravi, R. and Goemans, M.X., The constrained minimum spanning tree problem. In *Proc. 5th Scand. Worksh. Algorithm Theory,* number 1097 in Lecture Notes in Computer Science, 66–75. Springer-Verlag, 1996.

[15] Shmoys, D.B., Computing near-optimal solutions to combinatorial optimization problems. In Cook, W., Lovasz, L., and Seymour, P., Eds., *Combinatorial Optimization,* volume 20 of *DIMACS Series in Discrete Mathematics and Computer Science,* 355–397. AMS, 1995.

Further Information

For an excellent *survey* of the field of approximation algorithms, focusing on recent results and research issues, see the survey by David Shmoys [15]. Further details on almost all of the topics in this chapter, including algorithms and hardness results, can be found in the definitive book edited by Dorit Hochbaum [7]. NP-completeness is the subject of the classic book by Michael Garey and David Johnson [5]. An article by Johnson anticipated many of the issues and methods subsequently developed [9]. Randomized rounding and other probabilistic techniques used in algorithms are the subject of an excellent text by [12]. As of this writing, a searchable compendium of approximation algorithms and hardness results, by Crescenzi and Kann, is available on-line [4].

35

Scheduling Algorithms

David Karger
Massachusetts Institute of Technology

Cliff Stein
Dartmouth College

Joel Wein
Polytechnic University

35.1 Introduction

Scheduling theory is concerned with the *optimal allocation of scarce resources to activities over time.* The *practice* of this field dates to the first time two humans contended for a shared resource and developed a plan to share it without bloodshed. The *theory* of the design of algorithms for scheduling is younger, but still has a significant history—the earliest papers in the field were published more than forty years ago.

Scheduling problems arise in a variety of settings, as is illustrated by the following examples:

EXAMPLE 35.1:

Consider the central processing unit of a computer that must process a sequence of jobs that arrive over time. In what order should the jobs be processed in order to minimize, on average, the time that a job is in the system from arrival to completion?

EXAMPLE 35.2:

Consider a team of five astronauts preparing for the reentry of their space shuttle into the atmosphere. There is a set of tasks that must be accomplished by the team before reentry. Each task must be carried out by exactly one astronaut, and certain tasks can not be started until other tasks are completed. Which tasks should be performed by which astronaut, and in which order, to ensure that the entire set of tasks is accomplished as quickly as possible?

EXAMPLE 35.3:

Consider a factory that produces different sorts of widgets. Each widget must first be processed by machine 1, then machine 2, and then machine 3, but different widgets require different amounts of processing time on different machines. The factory has orders for batches of widgets; each order has a date by which it must be completed. In what order should the machines work on different widgets in order to insure that the factory completes as many orders as possible on time?

More generally, scheduling problems involve *jobs* that must scheduled on *machines* subject to certain *constraints* to optimize some *objective function.* The goal is to specify a *schedule* that specifies when and on which machine each job is to be executed.

Researchers have studied literally thousands of scheduling problems, and it would be impossible even to enumerate all known variants in the space of this chapter. Our goal is more modest. We wish to make the reader familiar with an assortment of algorithmic techniques that have proved useful for solving a large variety of scheduling problems. We will demonstrate these techniques by drawing from a collection of "basic problems" that model important issues arising in many scheduling problems, while at the same time remaining simple enough to permit elegant and useful analysis. These basic problems have received much attention, and their centrality was reinforced by two influential surveys [13, 29]. All three examples above fit into the basic problem framework.

In this survey we focus exclusively on algorithms that provably run, in the worst case, in time polynomial in the size of the input. If the algorithm always gives an optimum solution, we call it an *exact* algorithm. Many of the problems that we consider, however, are $\mathcal{NP}$-hard, and it thus seems unlikely that polynomial-time algorithms exist to solve them. In these cases we will be interested in *approximation algorithms;* we define a *ρ-approximation algorithm* to be an algorithm that runs in polynomial time and delivers a solution of value at most ρ times the optimum.

The rest of this chapter is organized as follows. We complete this introduction by laying out a standard framework covering the basic scheduling problems and a notation for describing them. We then explore various techniques that can be used to solve them. In Section 35.2 we present a collection of heuristics that use some simple rule to assign a priority to each job and then schedule the jobs in priority order. These heuristics are useful both for solving certain problems optimally in polynomial time, and for giving simple but high-quality approximations for certain $\mathcal{NP}$-hard scheduling problems. Many scheduling problems require a more complex approach than a simple priority rule; in Section 35.3 we study algorithms that are more sophisticated in their greedy choices. In Section 35.4 we discuss the application of some basic tools of combinatorial optimization, such as network optimization and linear programming, to the design of scheduling algorithms. We then turn exclusively to $\mathcal{NP}$-hard problems. In Section 35.5 we introduce the

notion of a *relaxation* of a problem, and show how to use relaxations to design approximation algorithms. Finally, in Section 35.6 we discuss enumeration and scaling techniques by which certain other $\mathcal{NP}$-hard scheduling problems can be approximated arbitrarily closely in polynomial time.

The Framework of Basic Problems

A scheduling problem is defined by three separate elements: the **machine environment,** the *optimality criterion,* and a set of *side constraints and characteristics.* We first discuss the simplest machine environment, and use that to introduce a variety of **optimality criteria** and side constraints. We then introduce and discuss more complex machine environments.

The One-Machine Environment

In all of our scheduling problems we begin with a set $\mathcal{J}$ of n jobs, numbered $1, \ldots, n$. In the *one-machine* environment we have one machine that can process at most one job at a time. Each job j has a processing requirement p_j; namely, it requires processing for a total of p_j units of time on the machine. If each job must be processed in an uninterrupted fashion, we have a *nonpreemptive* scheduling environment, whereas if a job may be processed for a period of time, interrupted and continued at a later point in time, we have a *preemptive* environment. A schedule S for the set $\mathcal{J}$ specifies, for each job j, which p_j units of time the machine uses to process job j. Given a schedule S, we denote the *completion time* of job j in schedule S by C_j^S.

The goal of a scheduling algorithm is to produce a "good" schedule, but the definition of "good" will vary depending on the application. In Example 35.2 above, the goal is to process the entire batch of jobs as quickly as possible, or, in other words, to minimize the completion time of the last job finished in the schedule. In Example 35.1 we care less about the completion time of the last job in the batch as long as, on average, the jobs receive good service. Therefore, given a set of jobs and a machine environment, we must specify an *optimality criterion;* the goal of a scheduling algorithm will be to construct a schedule that optimizes this criterion. The two optimality criteria discussed in our examples are among the most basic optimality criteria: the *average completion time* of a schedule and its *makespan.* We define the makespan $C_{\max}^S = \max_j C_j^S$ of a schedule S to be the maximum completion time of any job in S, and the average completion of schedule S to be $\frac{1}{n}\Sigma_{j=1}^n C_j^S$. Note that optimizing the average completion time is equivalent to optimizing the *sum* of completion times $\Sigma_{j=1}^n C_j^S$.

We next turn to *side constraints and characteristics* that modify the one-machine environment. A number of side constraints and characteristics are possible; for example, we must specify whether or not preemption is allowed. Two other possible constraints model the arrival of jobs over time or the possibility of logical dependence between jobs. In a scheduling environment with *release date* constraints, we associate with each job j a release date r_j; job j is only available for processing at time r_j or later. In a scheduling environment with *precedence constraints* we are given a partial order $\prec$ on the set $\mathcal{J}$ of jobs; if $j' \prec j$ then we may not begin processing job j until job j' is completed.

Although we are early in our discussion of scheduling models, we already have enough information to define a number of problems. We refer to various scheduling problems in the now-standard notation defined by [13]. A problem is denoted by $\alpha|\beta|\gamma$, where (i) α denotes the machine environment, (ii) β denotes various side constraints and characteristics, and (iii) γ denotes an optimality criterion.

For the one-machine environment α is 1. For the optimality criteria we have introduced so far, γ is either ΣC_j or $C_{\max}$. At this point in our discussion, β is a subset of r_j, *prec*, and *pmtn*, where these denote respectively the presence of (nontrivial) release date constraints, precedence constraints, and the ability to schedule preemptively. Any of the side constraints not explicitly listed are assumed *not* to be present—e.g., we default to a nonpreemptive model unless *pmtn* is given in the side constraints. As an illustration, $1||\Sigma C_j$ denotes the problem of nonpreemptively scheduling independent jobs on one machine so as to minimize their average completion time, while $1|r_j|\Sigma C_j$ denotes the variant of the problem in which jobs have release

dates. As another example, $1|r_j, pmtn, prec|C_{\max}$ denotes the problem of preemptively scheduling jobs with release dates and precedence constraints on one machine so as to minimize their makespan. Note that Example 35.1, given above, can be modeled by $1|r_j|\Sigma C_j$, or, if preemption is allowed, by $1|r_j, pmtn|\Sigma C_j$.

Two other possible elements of a scheduling application might lead to different objective functions in the one-machine environment. It is possible that not all jobs are of equal importance, and thus, when measuring average service provided to a job, one might wish to weight the average so as to give more importance to certain jobs. We model this by assigning a weight $w_j > 0$ to each job j, and generalize the ΣC_j criterion to the *average weighted completion time* of a schedule, $\frac{1}{n}\Sigma_{j=1}^{n} w_j C_j$. In the scheduling notation this optimality criterion is denoted by $\Sigma w_j C_j$.

It is also possible that each job j may have an associated *due date* d_j by which it should be completed. This gives rise to two different optimality criteria. Given a schedule S, we define $L_j = C_j^S - d_j$ to be the *lateness* of job j, and we will be interested in constructing a schedule that minimizes $L_{\max} = \max_{j=1}^{n} L_j$, the *maximum lateness* of any job in the schedule. Alternatively, we concern ourselves with constructing a schedule that maximizes the number of jobs that complete by their due dates. To capture this, given a schedule S we define $U_j = 0$ if $C_j^S \leq d_j$ and $U_j = 1$ otherwise; we can thus describe our optimality criterion as the minimization of ΣU_j, or more generally, $\Sigma w_j U_j$. As illustrations, $1|r_j|L_{\max}$ denotes the problem of nonpreemptively scheduling, on one machine, jobs with release dates and due dates so as to minimize the maximum lateness of any job, and $1|prec|\Sigma w_j U_j$ denotes the problem of nonpreemptively scheduling precedence-constrained jobs on one machine so as to minimize the total (summed) weight of the late jobs. Deadlines are not listed in the side constraints since they are implicit in the objective function.

Finally, we will consider one scheduling problem that deals with a more general optimality criterion. For each job j, we let $f_j(t)$ be any function that is nondecreasing with the completion time of the job, and, with respect to a schedule S, define $f_{\max} = \max_{j=1}^{n} f_j(C_j^S)$. The specific problem that we will consider (in Section 35.3) is $1|prec|f_{\max}$—the scheduling of precedence-constrained jobs on one machine so as to minimize the maximum value of $f_j(C_j)$ over all $j \in \mathcal{J}$.

More Complex Machine Environments: Parallel Machines and the Shop

Having introduced all of the optimality criteria, side characteristics and conditions that we will use in this survey, we now discuss more complex machine environments.

We first discuss *parallel machine* environments. In these environments we are given m machines. A job j with processing requirement p_j can be processed on any one of the machines, or, if preemption is allowed, started on one machine, and when preempted potentially continued on another machine. A machine can process at most one job at a time and a job can be processed by at most one machine at a time.

In the *identical parallel machine* environment the machines are identical, and job j requires p_j units of processing time when processed on any machine. In the *uniformly related machines* environment each machine i has a speed $s_i > 0$, and thus job j, if processed entirely on machine i, would take a total of p_j/s_i time to process. In the *unrelated parallel machines* environment we model machines that have different capabilities and thus their relative performance on a job is unrelated. In other words, the speed of machine i on job j, s_{ij}, depends on both the machine and the job; job j requires p_j/s_{ij} processing time on machine i. We define $p_{ij} = p_j/s_{ij}$.

In the *shop environment,* which primarily models various sorts of production environments, we again have m machines. In this setting a job j is made up of *operations,* with each operation requiring processing on a specific one of the m machines. Different operations may take different amounts of time (possibly 0). In the *open shop* environment, the operations of a job can be processed in any order, as long as no two operations are processed on different machines simultaneously. In the *job shop environment,* there is a total order on the operations of a job, and one operation can not be started until its predecessor in the total order is completed. A special case of the job shop is the *flow shop,* in which the order of the operations

is the same—each job requires processing on the same machines and in the same order, but different jobs may require different amounts of processing on the same machine. Typically in the flow shop and open shop environment, each job is processed exactly once on each machine.

In the scheduling notation, the identical, uniformly related and unrelated machine environments are denoted respectively by P, Q, and R. The open, flow and job shop environments are denoted by O, F, and J. When the environment has a fixed number of machines the number is included in the environment specification; so, for example, P2 denotes the environment with two identical parallel machines. Note that Example 35.2 can be modeled by P5$|prec|C_{\max}$, and Example 35.3 can be modeled by F3$|r_j|\Sigma U_j$.

35.2 Priority Rules

The most obvious approach to solving a scheduling problem is a greedy one: whenever a machine becomes available, assign some job to it. A more sophisticated variant of this approach is to give each job a *priority* derived from the particular optimality criterion, and then, whenever a machine becomes available, assign the available job of highest priority to it. In this section we discuss such scheduling strategies for one-machine, parallel-machine and shop problems. In all of our algorithms, the priority of a job can be determined without reference to other jobs. This typically gives a simple scheduling algorithm that runs in $O(n \log n)$ time—the bottleneck being the time needed to sort the jobs by priority. We also discuss the limitations of these approaches, giving examples where they do not perform well.

One Machine

We first focus on algorithms for single-machine problems in which we give each job a priority, sort by priorities, and schedule in this order. To establish the correctness of such algorithms, it is often possible to apply an *interchange argument.* Suppose that there is an optimal schedule with jobs processed in nonpriority order. It follows that some adjacent pair of jobs in the schedule has inverted priorities. We show that if we swap these two jobs, the scheduling objective function is improved, thus contradicting the claim that the original schedule was optimal.

Average Weighted Completion Time: $1||\Sigma w_j C_j$

In perhaps the simplest scheduling problem, our objective is to minimize the sum of completion times ΣC_j. Intuitively, it makes sense to schedule the largest job at the end of the schedule to ensure that it does not contribute to the delay on any other job. We formalize this by defining the *shortest processing time* (SPT) algorithm: order the jobs by nondecreasing processing time (breaking ties arbitrarily) and schedule in that order.

THEOREM 35.1 SPT *is an exact algorithm for* $1||\Sigma C_j$.

PROOF To establish the optimality of the schedule constructed by SPT we use an interchange argument. Suppose for the purpose of contradiction that the jobs in the optimal schedule are *not* scheduled in nondecreasing order of completion time. Then there is some pair of jobs j and k such that j immediately precedes k in the schedule but $p_j > p_k$.

Suppose we exchange jobs j and k. All jobs other than j and k still start, and thus complete, at the same time as they did before the swap. All that changes is the completion times of jobs j and k. Suppose that originally job j started at time t and ended at time $t + p_j$, so that job k started at time $t + p_j$ and finished at time $t + p_j + p_k$. It follows that the original contribution of these two jobs to the sum of completion times, namely $(t + p_j) + (t + p_j + p_k) = 2t + 2p_j + p_k$, is replaced by their new contribution

of $2t + 2p_k + p_j$. This gives a net decrease of $p_j - p_k$ in ΣC_j, which is positive if $p_j > p_k$, implying that our original ordering was not optimal—a contradiction.

This algorithm and its proof of optimality generalize to the optimization of average *weighted* completion time, $1||\Sigma w_j C_j$. Intuitively, we would like to schedule as much weight as possible with each unit of processing time. This suggests scheduling jobs in nonincreasing order of w_j/p_j; the optimality of this rule can be established by a simple generalization of the previous interchange argument.

THEOREM 35.2 **[39]** *Scheduling jobs in nonincreasing order of w_j/p_j gives an optimal schedule for $1||\Sigma w_j C_j$.*

Maximum Lateness: $1||L_{\max}$

Maximum Lateness: $1||\mathrm{L}_{\max}$

A simple greedy algorithm also solves $1||L_{\max}$, in which we seek to minimize the maximum job lateness. A natural strategy is to schedule the job that is closest to being late, which suggests the `EDD` algorithm: order the jobs by nondecreasing due dates (breaking ties arbitrarily) and schedule in that order.

THEOREM 35.3 **[23]** `EDD` *is an exact algorithm for* $1||L_{\max}$.

PROOF We again use an interchange argument to prove that the schedule constructed by `EDD` is optimal. Assume without loss of generality that all due dates are distinct, and number the jobs so that $d_1 < d_2 < \cdots < d_n$. Among all optimal schedules, we consider the one with the fewest *inversions,* where an inversion is a pair of jobs j, k such that $j < k$ but k is scheduled before j. Suppose the given optimal schedule S is not the `EDD` schedule. Then there is a pair of jobs j and k such that $d_j < d_k$ but k immediately precedes j in the schedule.

Suppose we exchange jobs j and k. This does not change the completion time or lateness of any job other than j and k. We claim that we can only decrease $\max(L_j, L_k)$, so we do not increase the maximum lateness. Furthermore, since $j < k$, swapping jobs j and k decreases the number of inversions in the schedule. It follows that the new schedule has the same or better lateness than the original one but fewer inversions, a contradiction.

To prove the claim, note that in schedule S $C_j^S > C_k^S$ but $d_j < d_k$. It follows that $\max(L_j^S, L_k^S) = C_j^S - d_j$. Under the exchange, job j's completion time, and thus lateness, decreases. Job k's completion time rises to C_j^S, but this gives it a lateness of $C_j^S - d_k < C_j^S - d_j$. Thus, the maximum of the two latenesses has decreased.

Preemption and Release Dates

We now consider the more complex one-machine environment in which jobs may arrive over time, as modeled by the introduction of release dates. The greedy heuristics of the previous sections are not immediately applicable, since jobs of high priority might be released relatively late and thus not be available for processing before jobs of lower priority. The most natural idea to cope with this complication is to always process the available (released) job of highest priority. In a preemptive setting, this would mean, upon the release of a job of higher priority, preempting the currently running job and switching to the "better" job. We will show that this idea in fact yields optimal scheduling algorithms.

We thus define the *Shortest Remaining Processing Time Algorithm* `SRPT` : at each point in time, schedule the job with shortest remaining processing time, preempting when jobs of shorter processing time are released. We also generalize `EDD` : upon the release of jobs with earlier dues dates than the job currently being processed, preempt the current job and process the job with the earliest due date.

THEOREM 35.4 **[2, 22]** SRPT *is an exact algorithm for* $1|r_j, pmtn|\Sigma C_j$, *and* EDD *is an exact algorithm for* $1|r_j, pmtn|L_{\max}$

PROOF As before, we argue by contradiction, using a similar greedy exchange argument. However, instead of exchanging entire jobs, we exchange pieces of jobs, which is now allowed in our preemptive environment.

We focus on $1|r_j, pmtn|\Sigma C_j$. Consider a schedule in which available job j with the shortest remaining processing time is not being processed at time t, and instead available job k is being processed. Let p'_j and p'_k denote the remaining processing times for jobs j and k after time t, so $p'_j < p'_k$. In total, $p'_j + p'_k$ time is spent on jobs j and k after time t. We now perform an exchange. Take the first p'_j units of time that were devoted to either of jobs j and k after time t, and use them instead to process job j to completion. Then, take the remaining p'_k units of time that were spent processing jobs j and k, and use them to schedule job j. This exchange preserves feasibility since both jobs were released by time t.

In the new schedule, all jobs other than j and k have the same completion times as before. Job k finishes when job j originally finished. But job j, which needed $p'_j < p'_k$ additional work, finishes *before* job k originally finished. Thus we have reduced $C_j + C_k$ without increasing any other completion time, meaning we have reduced ΣC_j, a contradiction.

The argument that EDD solved $1|r_j, pmtn|L_{\max}$ goes much the same way. If at time t, job j with the earliest remaining due date is not being processed and job k with a later due date is, we reallocate the time spent processing job k to job j. This makes job j finish earlier, and makes job k finish when job j did originally. This cannot increase objective function value.

By considering how SRPT and EDD function if all jobs are available at time 0, we conclude that on one machine, in the absence of release dates, the ability to preempt jobs does not yield schedules with improved ΣC_j or $L_{\max}$ optimality criteria. This is not the case when jobs have release dates; intuitively, a problem such as $1|r_j|\Sigma C_j$ seems more difficult, as one can not simply preempt the current job for a newly-arrived better one, but rather must decide whether to start a worse job or wait for the better one. This intuition about the additional difficulty of this setting is justified—$1|r_j|\Sigma C_j$ and $1|r_j|L_{\max}$ are in fact $\mathcal{NP}$-complete problems. We discuss approximation algorithms for these problems in later sections.

We also note that these ideas have their limitations, and do not generalize to the $\Sigma w_j C_j$ criterion – $1|r_j, pmtn|\Sigma w_j C_j$ is $\mathcal{NP}$-hard. Finally, we note that SRPT and EDD are *on-line* algorithms – their decisions about which job to schedule currently do not require any information about which jobs are to be released in the future. See [38] for a comprehensive survey of on-line scheduling.

The Two-Machine Flow Shop

We now consider a more complex machine environment in which we want to minimize the makespan in a flow shop. In general, this problem is $\mathcal{NP}$-hard, even in the case of three machines. However, in the special case of the two-machine flow shop $F2||C_{\max}$, a priority-based ordering approach due to Johnson [24] yields an exact algorithm. We denote the operations of job j on the first and second machines as a pair (a_j, b_j). Intuitively, we want to get jobs done on the first machine as quickly as possible so as to minimize idleness on the second machine due to waiting for jobs from the first machine. This suggests using an SPT rule on the first machine. On the other hand, it would be useful to process the jobs with large b_j as early as possible on the second machine, while machine 1 is still running, so they will not create a large tail of processing on machine 2 after machine 1 is finished. This suggests some kind of *longest processing time first* (LPT) rule for machine 2.

We now formalize this intuition. We partition our jobs into two sets. A is the set of jobs j for which $a_j \leq b_j$, while B is the set for which $a_j > b_j$. We construct a schedule by first ordering all the jobs in A

by nondecreasing a_j value, and then all the jobs in B by nonincreasing b_j values. We process jobs in this order on both machines. This is called *Johnson's rule.*

It may be surprising that we do not reorder jobs to process them on the second machine. It turns out that for two-machine flow shops, such reordering is not necessary. A schedule in which all jobs are processed in the same order is called a *permutation schedule.*

LEMMA 35.1 An instance of F2$||C_{\max}$ always has an optimal schedule that is a permutation schedule.

Note that for three or more machines there is not necessarily an optimal permutation schedule.

PROOF Consider any optimal schedule, and number the jobs according to the time at which they complete on machine 1. Suppose that job k immediately precedes job j in the order in which jobs are completed on machine 2, but $j < k$. Let t be the time at which job k is started on machine 2. It follows that job k has completed on machine 1 by time t. Numbering $j < k$ means that j is processed earlier than k on machine 1, so it follows that job j also has completed on machine 1 by time t. Therefore, we can swap the order of jobs j and k on machine 2, and still have a legal schedule (since no other job's start time changes) with the same makespan. We can continue performing such swaps until there are none left to be done, implying that jobs on machine 2 are processed in the same order as those on machine 1.

Having limited our search for optimal schedules to permutation schedules, we present a clever argument given by Lawler et al. [29] to establish the optimality of the permutation schedule specified by Johnson's rule.

Renumber the jobs according to the ordering given by Johnson's rule. Notice that in a permutation schedule for F2$||C_{\max}$, there must be a job k that is started on machine 2 immediately after its completion on machine 1; for example, the job that starts immediately after the last idle time on machine 2. The makespan of the schedule is thus determined by the processing times of k jobs on machine 1 and $n-k+1$ jobs on machine 2, which is just a sum of $n+1$ processing times. If we reduce *all* the a_i and b_i by the same value p, then *every* sum of $n+1$ processing times decreases by $(n+1)p$, so the makespan of every permutation schedule is reduced by $(n+1)p$.

Now note that if a job has $a_i = 0$ it is scheduled first in some optimal permutation schedule, since it delays no jobs on machine 1 and only "buys time" for jobs that are processed later than it on machine 2. Similarly, if a job has $b_i = 0$, it is scheduled last in some optimal schedule.

Therefore, we can construct an optimal permutation schedule by repeatedly finding the minimum operation size amongst all the a_j and b_j values of the unscheduled jobs, subtracting that value from all of the operation sizes, and then scheduling the job with the new zero processing time according to the above rules. Now observe that the schedule constructed is exactly the schedule that orders the jobs by Johnson's rule. We have therefore proved the following.

THEOREM 35.5 **[24]** *Johnson's rule yields an optimal schedule for* F2$||C_{\max}$.

Parallel Machines

We now turn to the case of parallel machines. In the move to parallel machines, many problems that are easily solvable on one machine become $\mathcal{NP}$-hard; the focus therefore tends to be on approximation algorithms. In some cases, the simple priority-based rules we used for one machine generalize well. That is, we assign a priority to every job, and, whenever a machine becomes available, it starts processing the job that has the highest remaining priority. The schedules created by such algorithms, which immediately give work to any machine that becomes idle, will be referred to as *busy schedules.*

In this section, we also introduce a new method of analysis. Instead of arguing correctness based on interchange arguments, we give lower bounds on the quality of the optimal schedule. We then show that our algorithm produces a schedule whose quality is within some factor of the lower bound, thus demonstrating *a fortiori* that it is within this factor of the optimal schedule. This is a general technique for approximation, and it has the pleasing feature that we are able to guarantee that we are within a certain factor of the optimal value, *without knowing what that optimal value is.* Sometimes we can show that our greedy algorithm *achieves* the lower bound, thus demonstrating that the algorithm is actually optimal.

In this section, we devote most of our attention to the problem of minimizing the makespan (schedule length) on m parallel machines, and study the behavior of the greedy algorithm for the problem. We remark that for the average-completion-time problem $P||\Sigma C_j$, the greedy SPT algorithm also turns out to yield an optimal schedule. We discuss this further in Section "Applications of Matching."

As was mentioned in Section "One Machine," $P||C_{\max}$ is trivial when $m = 1$, as any schedule with no idle time will be optimal. Once we have more than one machine, things become more complicated. With preemption, it is possible to greedily construct an optimal schedule in polynomial time. In the nonpreemptive setting, however, it is unlikely that there is a polynomial time exact algorithm, since the problem is $\mathcal{NP}$-complete via a simple reduction from the $\mathcal{NP}$-complete partition problem [7]. We will thus focus on finding an approximately optimal solution. First, we will show that any busy schedule gives a 2-approximation. We will then see how this can be improved with a slightly smarter algorithm, the *Longest Processing Time* (LPT) algorithm, which is a 4/3-approximation algorithm. In Section 35.6 we will show that a more complicated algorithm can guarantee an even better quality of approximation.

Our analyses of these algorithms are all based on comparing their performance to certain lower bounds on the quality of the optimal schedule; their performance compared to the optimum can only be better. Our algorithms will make use of two simple lower bounds on the makespan $C^*_{\max}$ of the optimal schedule:

$$C^*_{\max} \geq \sum_{j=1}^{n} p_j/m \tag{35.1}$$

$$C^*_{\max} \geq p_j \text{ for all jobs } j\ . \tag{35.2}$$

The first lower bound says that the schedule is at least as long as the average machine load, and the second says that the schedule is as least as long as the size of any job. To demonstrate the power of these lower bounds, we begin with the preemptive problem, $P|pmtn|C_{\max}$. In this case, we show how to find a schedule that matches the maximum of the two lower bounds given above. We then use the lower bounds to establish performance guarantees for approximation algorithms for the nonpreemptive case.

Minimizing $C_{\max}$ with Preemptions

We give a simple algorithm, called McNaughton's wrap-around rule [32], that creates an optimal schedule for $P|pmtn|C_{\max}$ with at most $m - 1$ preemptions. This algorithm is different from many scheduling algorithms in that it creates the schedule machine by machine, rather than over time.

Observing that the lower bounds (35.1) and (35.2) still apply to preemptive schedules, we will give a schedule of length $D = \max\{\Sigma_j p_j/m, \max_j p_j\}$. We order the jobs arbitrarily. Then we begin placing jobs on the machines, in order, filling machine i up until time D before starting machine $i + 1$. Thus, a job of length p_j may be split, assigned to the last t units of time of machine i and the first $p_j - t$ units of time on machine $i + 1$, for some t. It is now easy to verify that since there are no more than mD units to be processed, every job is scheduled, and because $D - t \geq p_j - t$ for any t, a job is scheduled on at most one machine at any time. Thus we have created an optimal preemptive schedule.

THEOREM 35.6 **[32]** *McNaughton's wrap-around rule gives an optimal schedule for* $P|pmtn|C_{\max}$.

List Scheduling for $P||C_{max}$

In contrast to $P|pmtn|C_{max}$, $P||C_{max}$ is $\mathcal{NP}$-hard. We consider the performance of the list scheduling (LS) algorithm, which is a generic greedy algorithm: whenever a machine becomes available, process any unprocessed job.

THEOREM 35.7 **[11]** LS *is a* 2*-approximation algorithm for* $P||C_{max}$.

PROOF Let j' be the last job to finish in the schedule constructed by LS and let $s_{j'}$ be the time that j' begins processing. C_{max} is therefore $s_{j'} + p_{j'}$. All machines must be busy up to time $s_{j'}$, since otherwise job j' could have been started earlier. The maximum amount of time that all machines can be busy is $\Sigma_{j=1}^{n} p_j/m$, and so we obtain that

$$\begin{aligned} C_{max} &\leq s_{j'} + p_{j'} \\ &\leq \sum_{j=1}^{n} p_j + p_{j'} \\ &\leq C^*_{max} + C^*_{max} = 2C^*_{max} . \end{aligned}$$

The last inequality comes from lower bounds (35.1) and (35.2) above.

This algorithm can easily be implemented in $O(n + m)$ time. By a similar analysis, the algorithm guarantees an approximation of the same quality even if the jobs have release dates [14].

Longest Processing Time First for $P||C_{max}$

It is useful to think of the analysis of LS in the following manner. Every job starts being processed before time $\Sigma_{j=1}^{n} p_j$, and hence the schedule length is no more than $\Sigma_{j=1}^{n} p_j$ plus the length of the longest job that is running at time $\Sigma_{j=1}^{n} p_j$.

This motivates the natural idea that it is good to run the longer jobs early in the schedule and the shorter jobs later. This is formalized in the *Longest Processing Time* (LPT) rule: sort jobs in nonincreasing order of processing time and list schedule in that order.

THEOREM 35.8 **[12]** LPT *is a* 4/3*-approximation algorithm for* $P||C_{max}$.

PROOF We start by simplifying the problem. Suppose that j', the last job to finish in our schedule, is not the last job to start. Remove all jobs that start after time $s_{j'}$. This does not affect the makespan of our schedule, since these jobs must have run on other machines. Furthermore, it can only decrease the optimal makespan for the modified instance. Thus, if we prove an approximation bound for this new instance, it applies *a fortiori* to our original instance.

We can therefore assume that the last job to finish is the last to start, namely the smallest job. In this case, by the analysis of Theorem 35.7 above, LPT returns a schedule of length no more than $C^*_{max} + p_{min}$. We now consider two cases:

Case 1: $p_{min} \leq C^*_{max}/3$. In this case $C^*_{max} + p_{min} \leq C^*_{max} + (1/3)C^*_{max} \leq (4/3)C^*_{max}$.

Case 2: $p_{min} > C^*_{max}/3$. In this case, all jobs have $p_j > C^*_{max}/3$, and hence in the optimal schedule there are at most 2 jobs per machine. Number the jobs in order of nonincreasing p_j. If $n \leq m$, then the optimal schedule trivially puts one job on each machine. We thus consider the remaining case with $m < n \leq 2m$. In this case, we claim that, for each $j = 1, \ldots, m$ the optimal schedule pairs job j with job $2m + 1 - j$ if $2m + 1 - j \leq n$ and places job j by itself otherwise. This can be shown to be optimal via a simple

interchange argument. We finish the proof by observing that this is exactly the schedule that LPT would construct.

This algorithm needs to sort the jobs, and can be implemented in $O(m + n \log n)$ time. If we are willing to spend substantially more time, we can obtain a $(1+\epsilon)$-approximation algorithm for any fixed $\epsilon > 0$; see Section 35.6.

List Scheduling for $P|prec|C_{max}$

Even when our input contains precedence constraints, list scheduling is still a 2-approximation algorithm. Given a precedence relation *prec*, we say that a job is *available* at time t if all its predecessors have completed processing by time t. Recall that in list scheduling, whenever a machine becomes idle, any available job is scheduled. Before giving the algorithm, we give one additional lower bound that is relevant to scheduling with precedence constraints. Let $j_{i_1}, j_{i_2}, \ldots, j_{i_k}$ be any set of jobs such that $j_{i_1} \prec j_{i_2} \prec \cdots \prec j_{i_k}$, then

$$C^*_{max} \geq \sum_{\ell=1}^{k} p_{i_\ell} \ . \tag{35.3}$$

In other words, the total processing time of any chain of jobs is a lower bound on the makespan.

THEOREM 35.9 **[11]** LS *is a* 2*-approximation algorithm for* $P|prec|C_{max}$.

PROOF Let j_1 be the last job to finish. Define j_2 to be the latest-finishing predecessor of j_1, and inductively define $j_{\ell+1}$ to be the latest-finishing predecessor of j_ℓ, continuing until reaching j_k, a job with no predecessors. Let $\mathcal{C} = \{j_1, \ldots, j_k\}$. We partition time into two sets, A, the points in time when a job in $\mathcal{C}$ is running, and B, the remaining time. Observe that during all times in B, all machines must be busy, for if they were not, there would be a job from $\mathcal{C}$ that had all its predecessors completed and would be ready to run. Hence, $C_{max} \leq |A| + |B| \leq \Sigma_{j \in \mathcal{C}} p_j + \Sigma_{j=1}^{n} p_j \leq 2C^*_{max}$, where the last inequality follows by applying lower bounds (35.3) and (35.1). Note that $|A|$ is the total length of intervals in A.

For the case when all processing times are exactly one, $P|prec|C_{max}$ is solvable in polynomial time if there are only two machines [27], and is $\mathcal{NP}$-complete if there are an arbitrary number of machines [40]. The complexity of the problem in the case when there are a fixed constant number of machines is one of the more famous open problems in scheduling.

List Scheduling for $O||C_{max}$

List scheduling can also be applied to $O||C_{max}$. Recall that in this problem, each job must be processed for disjoint intervals of time on several different machines. By an analysis similar to that used for $P||C_{max}$, we will show that any algorithm that constructs a busy schedule for $O||C_{max}$ is a 2-approximation algorithm. Let P_{max} be the maximum total processing time, summed over all machines, for any one job, and let Π_{max} be the maximum total processing time, summed over all jobs, of any one machine. Clearly, both P_{max} and Π_{max} are lower bounds on the makespan of the optimal schedule. We show that any busy schedule has makespan at most $P_{max} + \Pi_{max}$.

To see this, consider the machine M' that finishes processing last, and consider the last job j' to finish on machine M'. At any time during the schedule, either M' is processing a job or job j' is being processed (if neither of these is true, then list scheduling would require that j' be running on M', a contradiction). However, the total length of time during which j' undergoes processing is at most P_{max}. During all the remaining time in the schedule, machine M' must be busy. But machine M' is busy for at most

$\Pi_{\max}$ time units. Thus the total length of the schedule is at most $P_{\max} + \Pi_{\max}$, as claimed. Since $P_{\max} + \Pi_{\max} \leq C^*_{\max} + C^*_{\max} = 2C^*_{\max}$, we obtain

THEOREM 35.10 **(Racsmány, see [3])** *List scheduling is a 2-approximation algorithm for* $O||C_{\max}$.

Limitations of Priority Rules

For many problems, simple scheduling rules do not yield good schedules, and thus given a scheduling problem, the algorithm designer should be careful about applying one of these rules without justification. In particular, for many problems, particularly those with precedence constraints and release dates, the optimal schedule has *unforced idle time.* That is, if one is constructing the schedule over time, there may be a time t when there is an idle machine m and an available job j, but scheduling job j on machine m at time t will yield a suboptimal schedule.

Consider the problem $Q||C_{\max}$ and recall that for $P||C_{\max}$, list scheduling is a 2-approximation algorithm. Consider a two-job two-machine instance in which $s_1 = 1$, $s_2 = x$, $p_1 = 1$, $p_2 = 1$, and $x > 2$. Then `LS` , `SPT`, and `LPT` all schedule one job on machine 1 and one on machine 2, and the makespan is thus 1. However, the schedule that places both jobs on machine 2 has makespan $2/x < 1$. By making x arbitrarily large, we see that none of these simple algorithms, which all have approximation ratio at least $x/2$, have bounded approximation ratios.

For this problem there is actually a simple heuristic that comes within a factor of 2 of optimal, but for some problems, such as $Q|prec|C_{\max}$ and $R||C_{\max}$, there is no simple algorithm known that comes anywhere close to optimal. We also note that even though list scheduling is a 2-approximation for $O||C_{\max}$, for $F||C_{\max}$ busy schedules can be of makespan $\Omega(m)$ times optimal [10].

35.3 Sophisticated Greedy Approaches

As we have just argued, for many problems, the priority algorithms that consider jobs in isolation, as in Section 35.2, are not sufficient. In this section, we consider algorithms that do more than sort jobs by some priority measure — they take other jobs into account when making a decision about where to schedule a job. The algorithms we study here are "incremental" in nature: they start with an empty solution and grow it, one job at a time, until the optimal solution is revealed. At each step the decision about which job to add to the growing solution is made greedily, but is based on the current context of jobs which have already been scheduled. We present two examples which are classic examples of the *dynamic programming* paradigm, and several others that are more specialized.

All the algorithms share an analysis based on the idea of *optimal substructure.* Namely, if we consider the optimal solution to a problem, we can often argue that its "subparts" (e.g., prefixes of the optimal schedule) are optimal solutions to "subproblems" (e.g., the problem of scheduling the set of jobs in that prefix). This lets us argue that as our algorithms build their solution incrementally, they are building optimal solutions to bigger and bigger subproblems of the original problem, until they reach an optimal solution to the entire problem.

An Incremental Greedy Algorithm for $1||f_{\max}$

The first problem we consider is $1||f_{\max}$, which was defined in Section 35.1. In this problem, each job has some nondecreasing *penalty function* on its completion time C_j, and the goal is to find a schedule minimizing the maximum $f_j(C_j)$. As one example, $1||L_{\max}$ is captured by setting $f_j(t) = t - d_j$.

A greedy strategy still applies, when suitably modified. It is convenient, instead of talking about scheduling the "most penalizing" (e.g., earliest due date) job first, to talk about scheduling the "least penalizing" (e.g., latest due date) job last. Let $p(\mathcal{J}) = \Sigma_{j \in \mathcal{J}} p_j$ be the total processing time of the entire set of

jobs. Note that some job must complete at time $p(\mathcal{J})$. We find the job j that minimizes $f_j(p(\mathcal{J}))$, and schedule this job last. We then (recursively) schedule all the remaining jobs before j so as to minimize their maximum penalty. We call this algorithm `Least-Cost-Last`.

Observe the difference between this and our previous scheduling rules. In this new scheme, we cannot determine the best job to schedule second-to-last until we know which job is scheduled last (we need to know the processing time of the last job in order to know the processing time of the recursive subproblem). Thus, instead of a simple $O(n \log n)$-time sorting algorithm based on absolute priorities, we are faced with an algorithm that inspects k jobs in order to identify the job to be scheduled kth, giving a total running time of $O(n + (n-1) + \cdots + 1) = O(n^2)$.

This change in algorithm is matched by a change in analysis. Since the notion of which job is worst can change as the schedule is constructed, there is no obvious fixed priority to which we can apply a local exchange argument. Instead, as with $P|pmtn|C_{\max}$ in Section "Minimizing $C_{\max}$ with Preemptions," we show that our algorithm's greedy decisions are in agreement with a provable lower bound on the quality of the optimal schedule. Our algorithm produces a schedule that matches the lower bound and must therefore be optimal.

Let $f^*_{\max}(S)$ denote the optimal value of the objective function if we are only scheduling the jobs in S. Consider the following two facts about $f^*_{\max}$:

$$\begin{aligned} f^*_{\max}(\mathcal{J}) &\geq \min_{j \in N} f_j(p(\mathcal{J})) \\ f^*_{\max}(\mathcal{J}) &\geq f^*_{\max}(\mathcal{J} - \{j\}) \end{aligned}$$

The first of these statements follows from the fact that some job must be scheduled last. The second follows from the fact that if we have an optimal schedule for $\mathcal{J}$ and remove a job from the schedule, then we do not increase the completion time of any job. Therefore, since the f_j are increasing functions, we do not increase any penalty.

We use these inequalities to prove by induction that our schedule is optimal. According to our scheduling rule, we schedule last the job j minimizing $f_j(p(\mathcal{J}))$. By induction, this gives us a schedule with objective $\max\{f_j(p(\mathcal{J})), f^*_{\max}(\mathcal{J} - \{j\})\}$. But since each of these quantities is (by the equations above) a lower bound on the optimal $f^*_{\max}(\mathcal{J})$, we see that in fact we obtain a schedule whose value is a lower bound on $f^*_{\max}(\mathcal{J})$, and thus must in fact equal $f^*_{\max}(\mathcal{J})$.

Extension to $1|prec|f_{\max}$

Our argument from the previous section continues to apply even if we introduce *precedence constraints*. In the $1|prec|f_{\max}$ problem, a partial order on jobs is given, and we must build a schedule that does not start a job until all jobs preceding it in the partial order have completed. Our above algorithm applies essentially unchanged to this case. Note that the last job in the schedule must be a job with no successors. We therefore build an optimal schedule by scheduling last the job j that, among jobs with no successors, minimizes $f_j(P(\mathcal{J}))$. We then recursively schedule all other jobs before it. The proof of optimality goes exactly as before, using the fact that if L is the set of all jobs without successors, then

$$f^*_{\max}(\mathcal{J}) \geq \min_{j \in L} f_j(P(\mathcal{J}))$$

This is the same as the first equation above, except that the minimum is taken only over jobs without successors. The remainder of the proof proceeds unchanged.

THEOREM 35.11 **[26]** `Least-Cost-Last` *is an exact algorithm for* $1|prec|f_{\max}$.

It should also be noted that, once again, the fact that our algorithm is greedy makes preemption a moot point. One job needs to finish last, and it immediately follows that we can do no better than executing all of that job last. Thus, our greedy algorithm continues to be optimal.

An Alternative Approach

Moore [33] gave a different approach to $1||f_{\max}$ that may be faster in some cases. His scheme is based on a reduction to the maximum lateness problem and its solution by the EDD rule. To see how an algorithm for $L_{\max}$ can be applied to $1||f_{\max}$, suppose we want to know whether there is a schedule with $f_{\max} \leq B$. We can decide this as follows. Give each job j a deadline d_j equal to the maximum t for which $f_j(t) \leq B$. It is easy to see that a schedule has $f_{\max} \leq B$ precisely when every job finishes by its specified deadline, i.e., $L_{\max} \leq 0$. Thus, we have converted the feasibility problem for $f_{\max}$ into an instance of the lateness problem. The optimization problem may therefore be solved by a binary search for the correct value of B.

Dynamic Programming for $1||\Sigma w_j U_j$

We now consider $1||\Sigma w_j U_j$ problem, in which the goal is to minimize the total weight of late jobs. This problem is *weakly $\mathcal{NP}$-complete.* That is, although it is $\mathcal{NP}$-complete, for integral weights it is possible to solve the problem exactly in $O(n\Sigma w_j)$ time, which is polynomial if the w_j are bounded by a polynomial. The necessary algorithm is a classical dynamic program that builds the solution out of solutions to smaller problems (a detailed introduction to dynamic programming can be found in many algorithms textbooks, see, for example [6]). This $O(n\Sigma w_j)$ dynamic programming algorithm has several consequences. First, it immediately yields an $O(n^2)$-time algorithm for $1||\Sigma U_j$ problem–just take all weights to be 1. Furthermore, we will show in Section 35.6 that this algorithm can be used to derive a *fully polynomial approximation scheme* for the general problem that finds a schedule with $\Sigma w_j U_j$ within $(1+\epsilon)$ of the optimum in time polynomial in $1/\epsilon$ and n.

The first observation to make is that under this objective, a schedule partitions the jobs into two types: those completed by their due dates, and those not completed. Clearly, we might as well process all the jobs that meet their due date before processing any that do not. Furthermore, the processing order of these jobs might as well be determined using the Earliest Due Date (EDD) rule from Section "Maximum Lateness:$1||L_{\max}$:" when all jobs can be completed by their due date (implying nonpositive maximum lateness), EDD, which minimizes maximum lateness, will clearly find a schedule that does so.

It is therefore convenient to discuss *feasible subsets* of jobs that can all be scheduled together to complete by their due dates. The question of finding a minimum weight set of late jobs can then be equivalently restated as finding a maximum weight feasible subset of the jobs.

To solve this problem, we aim to solve a harder one: namely, to identify the fastest-completing maximum weight feasible subset. We do so via dynamic programming. Order the jobs according to increasing due date. Let T_{wj} denote the minimum completion time of a weight w-or-greater feasible subset of $1, \ldots, j$, or ∞ if there is no such subset. Note that $T_{0j} = 0$, while $T_{w0} = \infty$ for all $w > 0$. We now give a dynamic program to compute the other values T_{wj}. Consider the fastest completing weight w-or-greater feasible subset S of $\{1, \ldots, j+1\}$. Either $j+1 \in S$ or it is not. If $j+1 \notin S$, then $S \subseteq \{1, \ldots, j\}$ and is then clearly the fastest completing weight w-or-greater subset of $\{1, \ldots, j\}$, so S completes in time T_{wj}. If $j+1 \in S$, then since we can schedule feasible subsets using EDD , $j+1$ can be scheduled last. The jobs preceding it have weight at least $w - w_{j+1}$, and clearly form the minimum-completion-time subset of $1, \ldots, j$ with this weight. Thus, the completion time of this feasible set is $T_{w-w_{j+1},j} + p_{j+1}$. It follows that

$$T_{w,j+1} = \begin{cases} \min\left(T_{w,j}, T_{w-w_{j+1}} + p_{j+1}\right) & \text{if } T_{w-w_{j+1},j} + p_j \leq d_{j+1} \\ T_{wj} & \text{otherwise} \end{cases}$$

Now observe that there is clearly no feasible subset of weight exceeding Σw_j, so we can stop our dynamic program once we reach this value of w. This takes $O(n\Sigma w_j)$ time. Once we have all the values T_{wj}, we can find the maximum weight feasible subset by identifying the largest value of w for which some T_{wj} (and thus T_{wn}) is finite.

This gives a standard $O(n\Sigma_j w_j)$ time dynamic program for computing T_{wn} for every relevant value w; the maximum w for which T_{wn} is finite is the maximum total weight of jobs that can be completed by their due date.

THEOREM 35.12 **[30]** *Dynamic programming yields an $O(n\Sigma w_j)$-time algorithm for exactly solving $1||\Sigma w_j U_j$.*

We remark that a similar dynamic program can be used to solve the problem in time $O(n\Sigma p_j)$, which is effective when the processing times are polynomially bounded integers. We also note that a quite simple greedy algorithm due to Moore [33] can solve the unweighted $1||\Sigma U_j$ problem in $O(n \log n)$ time.

Dynamic Programming for $P||C_{max}$

For a second example of the applicability of dynamic programming, we return to the $\mathcal{NP}$-hard problem $P||C_{max}$, and focus on a special case that is solvable in polynomial time—the case in which the number of different job processing times is bounded by a constant. While this special case might appear to be somewhat contrived, in Section 35.6 we will show that it can form the core of a polynomial approximation scheme for $P||C_{max}$.

LEMMA 35.2 [18] Given an instance of $P||C_{max}$ in which the p_j take on at most s distinct values, there exists an algorithm which finds an optimal solution in time $n^{O(s)}$.

PROOF Assume for now that we are given a target schedule length T. We again use dynamic programming. Let the different processing times be $z_1, \ldots, z_s$. The key observation is that the set of jobs on a machine can be described by an s-dimensional vector $V = (v_1, \ldots, v_s)$, where v_k is the number of jobs of length z_k. There are at most n^s such vectors since each entry has value at most n. Let $\mathcal{V}$ be the set of all such vectors whose total processing time (that is, $\Sigma v_i z_i$) is less than T. In the optimal schedule, every machine is assigned a set of jobs corresponding to a vector from this set. We now define $M(x_1, \ldots, x_s)$ to be the minimum number of machines needed to schedule a job set consisting of x_i jobs of size z_i, for $i = 1, \ldots, s$. We observe that

$$M(x_1, \ldots, x_s) = 1 + \min_{V \in \mathcal{V}} M(x_1 - v_1, \ldots, x_s - v_s) \ .$$

The minimization is over all possible vectors that could be processed by the "first" machine counted by the quantity 1, and the recursive expression denotes the best way to process the remaining work. Thus we need to compute a table with n^s entries, where each entry depends on $O(n^s)$ other entries, and therefore the computation takes time $O(n^{2s})$.

It remains to handle the assumption that we know T. The easiest way to do this is to perform a binary search on all possible values of T. A slightly more sophisticated approach is to search only over the $O(n^s)$ makespans of vectors describing sets of jobs, as one of these clearly determines the makespan of the solution.

35.4 Matching and Linear Programming

Networks and linear programs are central themes in combinatorial optimization, and are useful tools in the solution of many problems. Therefore, it is not surprising that these techniques can be applied profitably to scheduling problems as well. In this section, we discuss applications of *bipartite matching* and *linear*

programming to the exact solution of certain scheduling problems; in Section 35.5 we will revisit both techniques in the design of approximation algorithms for $\mathcal{NP}$-hard problems.

Applications of Matching

Given a bipartite graph on two sets of vertices A and B and an edge set $E \subseteq A \times B$, a *matching* M is a subset of the edges, such that each vertex A and B is an endpoint of at most one edge of M. A natural matching that is useful in scheduling problems is one that matches jobs to machines; the matching constraints force each job to be scheduled on at most one machine, and each machine to be processing at most one job. If A has no more vertices than B, we call a matching *perfect* if every vertex of A is in some matching edge. It is also possible to assign weights to the edges, and define the weight of a matching to be the sum of the weights of the matching edges. The key fact that we use in this section is that minimum weight perfect matchings can be computed in polynomial time (see, e.g.,[1]).

Matching to Schedule Positions for $R||\Sigma C_j$

In this section we give a polynomial-time algorithm for $R||\Sigma C_j$ that matches jobs to *positions* in the schedule on each machine. For any schedule, let κ_{ik} be the kth-from-last job to run on machine i, and let ℓ_i be the number of jobs that run on machine i. By observing that the completion time of a job is equal to the sum of the processing times of the jobs that run before it, we have that

$$\sum_j C_j = \sum_{i=1}^{m} \sum_{k=1}^{\ell_i} C_{\kappa_{ik}} = \sum_{i=1}^{m} \sum_{k=1}^{\ell_i} \sum_{x=k}^{\ell_i} p_{i,\kappa_{xi}} = \sum_{i=1}^{m} \sum_{k=1}^{\ell_i} k p_{i,\kappa_{ki}} \ . \tag{35.4}$$

From this, we see that the kth from last job to run on a machine contributes exactly k times its processing time to the sum of completion times. Based on this observation, Horn [21] and Bruno, Coffman and Sethi [4] proposed formulating $R||\Sigma C_j$ problem as a minimum-weight bipartite matching problem. We define a bipartite graph $G = (V, E)$ with $V = A \cup B$ as follows. A will contain n vertices v_j, one for each of the n jobs $j = 1, \ldots, n$. B will contain nm nodes w_{ik}, where vertex w_{ik} represents the kth-from-last position on machine i, for $i = 1, \ldots, m$ and $k = 1, \ldots, n$. We include in E an edge (v_j, w_{ik}) between every node in A and every node in B. Using (35.4) we define the weights on the edges from A to B as follows: edge (v_j, w_{ik}) is assigned weight kp_{ij}.

We now argue that a minimum-weight perfect matching in this graph corresponds to an optimal schedule. First, note that for each valid schedule there is a perfect matching in G. Not every perfect matching in G corresponds to a schedule, since a job might be assigned to the kth from last position while less than k jobs are assigned to that machine; however, such a perfect matching is clearly not of minimal weight—a better matching can be obtained by pushing the $k' < k$ jobs assigned to that machine into the k' from last slots. Therefore, a schedule of minimum total completion time corresponds to a minimum-weight perfect matching in the bipartite graph.

THEOREM 35.13 **[4, 21]** *There is a polynomial-time algorithm for* $R||\Sigma C_j$.

In the special case of parallel identical machines, it remains true that the kth-from-last job to run on a machine contributes exactly k times its processing time to the sum of completion times. Since in this case the processing time of each job is the same on any machine, the algorithm is clear: schedule the m largest jobs last on each machine, schedule the next m largest jobs next to last, etc. The schedule constructed is exactly that constructed by the `SPT` algorithm.

COROLLARY 35.1 **[5]** `SPT` is an exact algorithm for $P||\Sigma C_j$.

Matching Jobs to Machines: $O|pmtn|C_{max}$

For our second example of the utility of matching, we give an algorithm for $O|pmtn|C_{max}$ due to Gonzales and Sahni [9]. This algorithm will not find just one matching, but rather a sequence of matchings, each of which will correspond to a partial schedule, and then concatenate all of these partial schedules together. Recall from our discussion of $O||C_{max}$ in Section 35.2 that two lower bounds on the makespan of a nonpreemptive schedule are the *maximum machine load* Π_{max} and the *maximum job size* P_{max}. Both of these remain lower bounds when preemption is allowed. In the nonpreemptive setting, a simple greedy algorithm gives a schedule with makespan bounded by $P_{max} + \Pi_{max}$. We now show that when preemption is allowed, matching can be used to achieve a makespan equal to $\max(P_{max}, \Pi_{max})$.

The intuition behind the algorithm is the following. Consider the schedule at any point in time. At this time, each machine is processing at most one job. In other words, the schedule at each point in time defines a matching between jobs and machines. We aim to find a matching that forms a part of the optimal schedule, and process jobs according to it for some time. Our goal is that processing the matched jobs on their matched machines for some amount of time t, and adjusting P_{max} and Π_{max} to reflect the decreased remaining processing requirements, should reduce $\max(P_{max}, \Pi_{max})$ by t. It follows that if we repeat this process for a total amount of time equal to $\max(P_{max}, \Pi_{max})$, we will reduce $\max(P_{max}, \Pi_{max})$ to 0, implying that there is no work remaining in the system.

What properties should our matching of jobs to machines have? Recall that our goal is to reduce our lower bound. Call a job *tight* if its total processing cost is P_{max}. Call a machine *tight* if its total load is Π_{max}. Clearly, it is necessary that every tight job undergo processing in our matching, since otherwise we will fail to subtract t from P_{max}. Similarly, it is necessary that every tight machine be in the matching in order to ensure that we reduce Π_{max} by t. Lastly, we can only execute the matching for t time if every job–machine pair in the matching actually requires t units of processing. In other words, we are seeking a matching in which every tight machine and job is matched, and each matching edge requires positive processing time. Such a matching is referred to as a *decrementing set.* That it always exists is a nontrivial fact (about stochastic matrices) whose proof is beyond the scope of this survey; we refer the reader to Lawler and Labetoulle's presentation of this algorithm [28].

To find a decrementing set, we construct a (bipartite) graph with a node representing each job and machine, and include an edge between machine node i and job node j if job j requires a nonzero amount of processing on machine i. In this graph we require a matching that matches each tight machine or job node; this can easily be found with a variant of traditional matching algorithms. Note that we must include the nontight nodes in the matching problem, since tight nodes can be matched to them.

Once we have found our decrementing set via matching, we have machines execute the jobs matched to them until one of the matched jobs completes its work on its machine, or until a new job or machine becomes tight (this can happen because some jobs and machines are not being processed in the matching). Whenever this happens, we find a new decrementing set. For simplicity, we assume that $P_{max} = \Pi_{max}$; this can easily be arranged by adding dummy operations, which can only make our task harder. Since our decrementing set includes every tight job and machine, it follows that executing for time t will reduce both P_{max} and Π_{max} by t. It follows that after $P_{max} = \Pi_{max}$ time, both quantities will be reduced to 0. Clearly this means that we are done in time equal to the lower bound.

One might worry that the number of decrementing set calculations we must perform could be non-polynomially bounded, making our approximation algorithm too slow. But this turns out not to happen. We only compute a new decrementing set when a job or machine finishes or when a new job or machine becomes tight. Each job–processor pair can finish only once, meaning that this occurs only nm times during our schedule. Also, each job or machine stays tight forever after it becomes tight; thus, new tight jobs and machines only occur $n + m$ times. Thus, constructing our schedule of optimal length requires only $mn + m + n$ matching computations.

THEOREM 35.14 **[9]** *There is a polynomial time algorithm for* $O|pmtn|C_{\max}$*, that finds an (optimal) schedule of makespan* $\max(P_{\max}, \Pi_{\max})$.

Linear Programming

We now discuss the application of *linear programming* to the design of scheduling algorithms. A *linear program* is given by a vector of variables $x = (x_1, \cdots, x_n)$, a set of m linear constraints of the form $a_{i1}x_1 + a_{i2}x_2 + \cdots + a_{in}x_n \leq b_i, 1 \leq i \leq m$, and a *cost vector* $c = (c_1, \ldots, c_n)$; the goal is to find an x that satisfies these constraints and that minimizes $cx = c_1x_1 + c_2x_2 + \cdots + c_nx_n$. Alternatively but equivalently, some of the inequality constraints might be given as equalities, and/or we may have no objective function and desire simply to find a feasible solution to the set of constraints. Many optimization problems can be formulated as linear programs, and thus solved efficiently, since a linear program can be solved in polynomial time [25].

In this section we consider $R|pmtn|C_{\max}$. To model this problem as a linear program, we use nm variables $x_{ij}, 1 \leq i \leq m, 1 \leq j \leq n$. Variable x_{ij} denotes the fraction of job j that is processed on machine i; for example, we would interpret a linear-programming solution with $x_{1j} = x_{2j} = x_{3j} = \frac{1}{3}$ as assigning $\frac{1}{3}$ of job j to machine 1, $\frac{1}{3}$ to machine 2 and $\frac{1}{3}$ to machine 3.

We now consider what sorts of linear constraints on the x_{ij} are necessary to ensure that they describe a valid solution to an instance of $R|pmtn|C_{\max}$. Clearly the fraction of a job assigned to any machine must be nonnegative, so we will create nm constraints

$$x_{ij} \geq 0 .$$

In any schedule, we must fully process each job. We capture this requirement with the n constraints:

$$\sum_{i=1}^{m} x_{ij} = 1, \quad 1 \leq j \leq n .$$

Note that, along with the previous constraints, these constraints imply that $x_{ij} \leq 1 \; \forall \, i, j$.

Our objective, of course, is to minimize the makespan D of the schedule. Recall that the amount of processing that job j would require, if run entirely on machine i, is p_{ij}. Therefore, for a set of fractional assignments x_{ij}, we can determine the amount of time machine i will work: it is just $\Sigma x_{ij} p_{ij}$, which must be at most D. We model this with the m constraints

$$\sum_{j=1}^{n} p_{ij}x_{ij} \leq D \text{ for } i = 1, \ldots, m .$$

Finally, we must ensure that no job is processed for more than D time; we model this with the n constraints

$$\sum_{i=1}^{m} x_{ij}\, p_{ij} \leq D, 1 \leq j \leq n .$$

To summarize, we formulate the problem as the following linear program:

$$\min \quad D \tag{35.5}$$

$$\sum_{i=1}^{m} x_{ij} \quad = \quad 1, \qquad \text{for } j = 1, \ldots, n , \tag{35.6}$$

$$\sum_{j=1}^{n} p_{ij}x_{ij} \quad \leq \quad D, \qquad \text{for } i = 1, \ldots, m , \tag{35.7}$$

$$\sum_{i=1}^{n} p_{ij} x_{ij} \leq D, \quad \text{for } j = 1, \ldots, n, \tag{35.8}$$

$$x_{ij} \geq 0 \quad \text{for } i = 1, \ldots, m, \ j = 1, \ldots, n. \tag{35.9}$$

It is clear that any feasible schedule for our problem yields an assignment of values to the x_{ij} that satisfies the constraints of our above linear program. However, it is not completely clear that solving the linear program yields a solution to the scheduling problem; this linear program does not specify the ordering of the jobs on a specific machine, but simply assigns the jobs to machines while constraining the maximum load on any machine. It thus fails to explicitly require that a job not be processed simultaneously on more than one machine.

Interestingly enough, we can resolve this difficulty with an application of open-shop scheduling. We define an open shop problem by creating an operation o_{ij} for each positive variable x_{ij}, and define the size of o_{ij} to be $x_{ij} p_{ij}$. We then find an optimal preemptive schedule for this instance, using the matching-based algorithm discussed in Section 35.4 We know that both the maximum machine load and maximum job size of this open shop instance are bounded above by D, and therefore the makespan of the resulting open shop schedule is at most D. If we now reinterpret the operations of each job in the open-shop schedule as fragments of the original job in the unrelated machines instance, we see that we have given a preemptive schedule of length D in which no two fragments of a job are scheduled simultaneously.

We thus have established the following.

THEOREM 35.15 [**28**] *There is an exact algorithm for* $\mathrm{R}|pmtn|C_{\max}$.

We will see further applications of linear programming to the development of approximation algorithms for $\mathcal{NP}$-hard scheduling problems in the next section.

35.5 Using Relaxations to Design Approximation Algorithms

We now turn exclusively to the design of approximation algorithms for $\mathcal{NP}$-hard scheduling problems. Recall that a *ρ-approximation algorithm* is one that is guaranteed to find a solution with value within a multiplicative factor of ρ of the optimum. Many of the approximation algorithms in this area are based on a *relaxation* of the $\mathcal{NP}$-hard problem. A relaxation of a problem is a version of the problem with some of the requirements or constraints removed ("relaxed"). For example, we might consider $1|r_j, pmtn|\Sigma C_j$ to be a relaxation of $1|r_j|\Sigma C_j$ in which the "no preemption" constraint has been relaxed. A second example of a relaxation might be a version of the problem in which we relax the constraint that a machine processes at most one job at a time; a solution to this relaxation may have several jobs scheduled at one time on the same machine.

A solution to the original problem is a solution to the relaxation, but a solution to the relaxation is not necessarily a solution to the original problem. This is clearly illustrated by our nonpreemptive/preemptive example—a nonpreemptive schedule is a legal solution to a preemptive problem, although perhaps not an optimal one, but the converse is not true. It follows that in the case of a minimization problem, the value of the optimal solution to the relaxation is a not-necessarily-tight lower bound on the optimal solution to the original problem.

An idea that has proven quite useful is to define first a relaxation of the problem which can be solved in polynomial time, and then to give an algorithm to convert the relaxation's solution into a valid solution to the original problem, with some degradation in the quality of solution. The key to making this work well

is to find a relaxation that preserves enough of the structure of the original problem to make the optimal relaxed solution "similar" to the original optimum, so that the relaxed solution does not degrade too much when converted to a valid solution.

In this section we discuss two sorts of relaxations of scheduling problems and their use in the design of approximation algorithms, namely the preemptive version of a nonpreemptive problem and a linear-programming relaxation of a problem.

There are generally two different ways to infer a valid schedule from the relaxed solution: one is to infer an *assignment* of jobs to machines while the other is to infer a *job ordering.* We give examples of both methods.

Before going any further, we introduce the notion of a relaxed decision procedure, which we will use both in Section "Rounding a Fractional Assignment to Machines:R$||C_{\max}$" and later in Section 35.6. A *ρ-relaxed decision procedure* (RDP) for a minimization problem accepts as input a target value T, and returns either **no,** asserting that no solution of value $\leq T$ exists, or returns a solution of value at most ρT. A polynomial-time ρ-relaxed decision procedure can easily be converted into a ρ-approximation algorithm for the problem via binary search for the optimum T; see [18, 19] for more details. This simple idea is quite useful, as it essentially lets us assume that we know the value T of an optimal solution to a problem. (Note that this is a different use of the word relax than the term "relaxation.")

Rounding a Fractional Assignment to Machines: R$||C_{\max}$

In this section we give a 2-relaxed decision procedure for R$||C_{\max}$. Recall the linear program that we used in giving an algorithm for R$|pmtn|C_{\max}$. If, instead of the constraints $x_{ij} \geq 0$, we could constrain the x_{ij} to be 0 or 1, the solution would constitute a valid nonpreemptive schedule. Furthermore, note that these integer constraints combined with the constraints (35.7) make the constraints (35.8) unnecessary (if a job is assigned integrally to a machine, constraint (35.7) ensures that is a fast enough machine, thus satisfying constraint (35.8) for that job). In other words, the following formulation has a feasible solution if and only if there is a nonpreemptive schedule of makespan D.

$$\sum_{i=1}^{m} x_{ij} = 1, \qquad \text{for } j = 1, \ldots, n\,, \tag{35.10}$$

$$\sum_{j=1}^{n} p_{ij} x_{ij} \leq D, \qquad \text{for } i = 1, \ldots, m\,, \tag{35.11}$$

$$x_{ij} \in \{0, 1\}, \quad \text{for } i = 1, \ldots, m,\ j = 1, \ldots, n\,. \tag{35.12}$$

This is an example of an *integer linear program,* in which the variables are constrained to be integers. Unfortunately, in contrast to linear programming, finding a solution to an integer linear program is $\mathcal{NP}$-complete. However, this integer programming formulation will still be useful. A very common method for obtaining a relaxation of an optimization problem is to formulate it as an integer linear program, and then to relax the integrality constraints. One obtains a fractional solution and then *rounds* the fractions to integers in a fashion that (hopefully) does not degrade the solution too dramatically.

In our setting, we relax the constraints (35.12) to $x_{ij} \geq 0$. We will also add an additional set of constraints that will ensure that the fractional solutions to this linear program have enough structure to be useful for approximation. Specifically, we disallow any part of a job j being processed on a machine i on which it could not complete in D time in a nonpreemptive schedule. Specifically, we include the following constraints:

$$x_{ij} = 0, \qquad \text{if } p_{ij} \geq D\,. \tag{35.13}$$

(In fact, instead of adding constraints, we can simply remove such variables from the linear program.) As argued above, this constraint is actually implicit in the integer program given by the constraints (35.10)

through (35.12), but was no longer guaranteed when we relaxed the integer constraints. Our new constraints can be seen as a replacement for the constraints (35.8) that we did not need in the integer formulation. Note also that these new constraints are only linear constraints when D is *fixed.* This is why we use an RDP instead of taking the more obvious approach of writing a linear program to minimize D.

To recap, constraints (35.10), (35.11), and (35.13) along with $x_{ij} \geq 0$ constitute a *linear-programming relaxation* of $R||C_{\max}$. Our relaxed decision procedure attempts solve this relaxation, obtaining a solution $\bar{x}_{ij}, 1 \leq i \leq m, 1 \leq j \leq n$. If there is no feasible solution, our RDP can output **no**—nonpreemptive schedule has makespan D or less. If the linear program is feasible, we will give a way to derive an integral assignment of jobs to machines from the fractional solution. Our job is made much easier by the fact, which we cite from the theory of linear programming, that we can find a so-called *basic* solution of this linear program that has at most $n + m$ positive variables. Since these $n + m$ positive variables must be distributed amongst n jobs, there are at most m jobs that are assigned in a fractional fashion to more than one machine.

We may now state our rounding procedure. For each (machine, job) pair (i, j) such that $\bar{x}_{ij} = 1$, we assign job j to machine i. We call the schedule of these jobs S_1. For the remaining at most m jobs, we simply construct a matching of the jobs to machines such that each job is matched to a machine it is already partially assigned to. We schedule each job on the machine to which it is matched, and call the schedule of these jobs S_2.

We defer momentarily the question of whether such a matching exists, and analyze the makespan of the resulting schedule, which is at most the sum of the makespans of S_1 and S_2. Since the x_{ij} form a feasible solution to the relaxed linear program, the makespan of S_1 is at most D. Since S_2 schedules at most one job per machine, and assigns j to i only if $x_{ij} > 0$, meaning $p_{ij} < D$, the makespan of S_2 is at most D (this argument is the reason we had to add constraint (35.13) to our linear program). Thus the overall schedule has length at most $2D$.

The argument that a matching always exists is somewhat complex and can only be sketched here. We create a graph G in which there is one node for each machine and one for each job, and an edge between each machine node i and job node j if $x_{ij} > 0$. We are again helped by the theory of linear programming, as the linear program we solved is a *generalized assignment problem.* As a result, for any basic solution, the structure of G is a forest of trees and 1-*trees*, which are trees with one edge added; for further details see [1]. We need not consider jobs that are already integrally assigned, so for every pair (i, j) such that $x_{ij} = 1$, we remove from G the nodes representing machine i, job j and their mutual edge (note that the constraints imply that this machine and job is not connected to any other machine or job). In the forest that remains, the only leaves are machine nodes, since every remaining job node represents a job that is fractionally assigned by the linear program and thus has an edge to at least two machines.

It is now straightforward to find a matching in G. We first consider the 1-trees, and in particular consider the unique cycle in each 1-tree. The nodes in these cycles alternate between machine nodes and job nodes, with an equal number of each. We arbitrarily choose an orientation of the cycle and assign each job to the machine that follows it in the oriented cycle. We then remove all of the matched nodes from G. What remains is a forest of trees; furthermore, it is possible that for each of these trees we have created at most one new leaf that is a job node. We then root each of the trees in the forest, either at its leaf job node, or, if it does not have one, at an arbitrary vertex. Finally, we assign each job node to one of its children machine nodes in the rooted tree. Each machine node has at most one parent, and thus is assigned at most one job. We have thus successfully matched all job nodes to machine nodes, as we required.

Thus, there exists a 2-relaxed decision procedure for $R||C_{\max}$, and we have the following theorem.

THEOREM 35.16 **[31]** *There is a* 2*-approximation algorithm for* $R||C_{\max}$.

Inferring an Ordering from a Preemptive Schedule for $1|r_j|\Sigma C_j$

In this section and the next we discuss techniques for inferring an ordering of jobs from a relaxation. In this section we consider the problem $1|r_j|\Sigma C_j$. Recall, as mentioned in Section "One Machine," that this problem is $\mathcal{NP}$-hard. However, we can find a good relaxation by the simple expedient of allowing preemption. Specifically, we use $1|r_j, pmtn|\Sigma C_j$ as a relaxation of $1|r_j|\Sigma C_j$. We have seen that $1|r_j, pmtn|\Sigma C_j$ can be solved without linear programming, simply by using the `SRPT` rule. We will make use of this relaxation by extracting from it the order of completion of the jobs in the optimal preemptive schedule, and create a nonpreemptive schedule with the same order of completion.

Our algorithm, which we call `Convert-Preempt-Schedule`, is as follows. We first obtain an optimal preemptive schedule P for the instance in question. We then order the jobs in their order of completion in P; assume by renumbering that $C_1^P \leq \cdots \leq C_n^P$. We schedule the jobs nonpreemptively in the same order. If at some point the next job in the order has not been released, we wait idly until its release date and then schedule it.

THEOREM 35.17 [35] `Convert-Preempt-Schedule` *is a* 2*-approximation algorithm for* $1|r_j|\Sigma C_j$.

PROOF The nonpreemptive schedule N constructed by `Convert-Preempt-Schedule` can be understood as follows. For each job j, consider the point of completion of the last piece of j scheduled in P, insert p_j extra units of time into the schedule at the completion point of j in P (delaying by an additional p_j time the part of the schedule after C_j^P) and then schedule j nonpreemptively in the newly inserted block of length p_j. Then, remove from the schedule all of the time originally allocated to processing job j. Finally, cut out any idle time in the resulting schedule that can be removed without changing the scheduled order of the jobs or violating a release date constraint. The result is exactly the schedule computed by `Convert-Preempt-Schedule`.

Note that the completion of job j is only delayed by insertion of blocks for jobs that finish earlier in P and hence

$$C_j^N \leq C_j^P + \sum_{k \leq j} p_k \, .$$

However, $\Sigma_{k \leq j} p_k \leq C_j^P$, since all of these jobs completed before j in P, and therefore

$$\sum_{j=1}^{n} C_j^N \leq 2 \sum_{j=1}^{n} C_j^P \, .$$

The theorem now follows from the fact that the total completion time of the optimal preemptive schedule is a lower bound on the total completion time of the optimal nonpreemptive schedule.

An Ordering from a Linear Programming Relaxation for $1|r_j, prec|\Sigma w_j C_j$

In this section we generalize the techniques of the previous section, applying them not to a preemptive schedule but instead to a linear programming relaxation of $1|r_j, prec|\Sigma w_j C_j$.

The Relaxation

We begin by describing the linear programming relaxation of our problem. Unlike our previous relaxation, this one does not arise from relaxing the integrality constraints of an integer linear program. Rather, we give several classes of inequalities that would be satisfied by feasible solutions to $1|r_j, prec|\Sigma w_j C_j$. These constraints are necessary but not sufficient to describe a valid solution to the problem.

The linear-programming formulation that we considered for $R||C_{\max}$ assigned jobs to machines but captured no information about the ordering of jobs on a machine. For $1|r_j, prec|\Sigma w_j C_j$ the ordering of jobs on a machine is a critical element of a high-quality solution, so we seek a formulation that can model this. We do this by making time explicit in the formulation: we will have n variables C_j, one for each of the n jobs; C_j will represent the completion time of job j in a schedule.

Consider the following formulation in these C_j variables, solutions to which correspond to optimal solutions of $1|r_j, prec|\Sigma w_j C_j$.

$$\text{minimize} \sum_{j=1}^{n} w_j C_j \tag{35.14}$$

subject to

$$C_j \geq r_j + p_j, \qquad j = 1, \ldots, n\,, \tag{35.15}$$

$$C_k \geq C_j + p_k, \qquad \text{for each pair } j, k \text{ such that } j \prec k\,, \tag{35.16}$$

$$C_k \geq C_j + p_k \quad \text{or} \quad C_j \geq C_k + p_j, \quad \text{for each pair } j, k\,. \tag{35.17}$$

Unfortunately, the last set of constraints are not linear constraints. Instead, we use a class of valid inequalities, introduced by Wolsey [41] and Queyranne [37]. Recall that we denote the entire set of jobs $\{1, \ldots, n\}$ as $\mathcal{J}$, and, for any subset $S \subseteq \mathcal{J}$, we define $p(S) = \Sigma_{j \in S} p_j$ and $p^2(S) = \Sigma_{j \in S} p_j^2$. We claim that for any feasible one-machine schedule (independent of constraints and objective)

$$\sum_{j \in S} p_j C_j \geq \frac{1}{2}\left(p^2(S) + p(S)^2\right), \quad \text{for each } S \subseteq \mathcal{J}\,. \tag{35.18}$$

We show that these inequalities are satisfied by the completion times of any valid schedule for one machine and thus in particular by the completion times of a valid schedule for $1|r_j, prec|\Sigma w_j C_j$.

LEMMA 35.3 [37, 41] Let $C_1, \ldots, C_n$ be the completion times of jobs in any feasible schedule on one machine. Then the C_j must satisfy the inequalities

$$\sum_{j \in S} p_j C_j \geq \frac{1}{2}\left(p(S)^2 + p^2(S)\right) \quad \text{for each } S \subseteq \mathcal{J}\,. \tag{35.19}$$

PROOF We assume that the jobs are indexed so that $C_1 \leq \cdots \leq C_n$. Consider first the case $S = \{1, \ldots, n\}$. Clearly for any job j, $C_j \geq \Sigma_{k=1}^{j} p_k$. Multiplying by p_j and summing over all j, we obtain

$$\sum_{j=1}^{n} p_j C_j \geq \sum_{j=1}^{n} p_j \sum_{k=1}^{j} p_k = \frac{1}{2}\left(p^2(S) + p(S)^2\right)\,.$$

Thus (35.19) holds for $S = \{1, \ldots, n\}$. The general case follows from the fact that for any other set of jobs S, the jobs in S are feasibly scheduled by the schedule of $\{1, \ldots, n\}$—just ignore the other jobs. So we may view S as our entire set of jobs and apply the previous argument.

In the special case of $1||\Sigma w_j C_j$ the constraints (35.19) give an exact characterization of the problem [37, 41]; specifically, any set of C_j that satisfy these constraints must describe the completion times of a feasible schedule, and thus these linear constraints effectively replace the disjunctive constraints (35.17). When we extend the formulation to include constraints (35.15) and (35.16), we no longer have an exact formulation, but rather a linear-programming relaxation of $1|r_j, prec|\Sigma w_j C_j$.

We note that this formulation has an exponential number of constraints; however, it can be solved in polynomial time by the use of the ellipsoid algorithm for linear programming [37, 41]. We also note that in the special case in which we just have release dates, a slightly strengthened version can (remarkably) be solved optimally in $O(n \log n)$ time [8].

Constructing a Schedule from a Solution to the Relaxation

We now show that a solution to this relaxation can be converted efficiently to an approximately optimal schedule. For simplicity, we ignore release dates and consider only $1|prec|\Sigma w_j C_j$. Our approximation algorithm, which we call `Schedule-by-`$\bar{C}_j$, is simple to state. We first solve the linear programming relaxation given by (35.14), (35.16), and (35.18) and call the solution $\bar{C}_1, \ldots, \bar{C}_n$; we renumber the jobs so that $\bar{C}_1 \leq \bar{C}_2 \leq \cdots \leq \bar{C}_n$. We then schedule the jobs in the order $1, \ldots, n$. Since there are no release dates there is no idle time. Note that this ordering of the jobs respects the precedence constraints, because if the $\bar{C}_j$ satisfy (35.14) then $j \prec k$ implies that $\bar{C}_j < \bar{C}_k$.

To analyze `Schedule-by-`$\bar{C}_j$, we begin by understanding why it is not an optimal algorithm. Unfortunately, $\bar{C}_1 \leq \cdots \leq \bar{C}_n$ being a feasible solution to (35.18) does not guarantee that, in the schedule in which job j is designated to complete at time $\bar{C}_j$ (thus defining its start time), at most one job is scheduled at any point in time. More formally, the intervals $(\bar{C}_j - p_j, \bar{C}_j]$, $j = 1, \ldots, n$, are not constrained to be disjoint. If $\bar{C}_1 \leq \cdots \leq \bar{C}_n$ actually corresponded to a valid schedule, then $\bar{C}_j$ would be no less than $\Sigma_{k=1}^{j} p_k$ for all j. We will see that, although the formulation does not guarantee this property, it does yield a relaxation of it, which is sufficient for the purposes of approximation.

THEOREM 35.18 **[17]** `Schedule-by-`$\bar{C}_j$ *is a 2-approximation algorithm for* $1|prec|\Sigma w_j C_j$.

PROOF Since $\bar{C}_j$ optimized a relaxation, we know that $\Sigma w_j \bar{C}_j$ is a lower bound on the true optimum. It therefore suffices to show that our algorithm gets within a factor of 2 of this lower bound. So we let $\tilde{C}_1, \ldots, \tilde{C}_n$ denote the completion times in the schedule found by `Schedule-by-`$\bar{C}_j$, and show that $\Sigma w_j \tilde{C}_j \leq 2\Sigma w_j \bar{C}_j$.

Since the jobs have been renumbered so that $\bar{C}_1 \leq \cdots \leq \bar{C}_n$, taking $S = \{1, \ldots, j\}$ gives

$$\tilde{C}_j = p(S) .$$

We now show that $\bar{C}_j \geq \frac{1}{2} p(S)$. (Again, if the $\bar{C}_j$ were feasible completion times in an actual schedule, we would have $\bar{C}_j \geq p(S)$. This relaxed version of the property is the key to the approximation.)

We use inequality (35.18) for $S = \{1, 2, \ldots, j\}$.

$$\sum_{k=1}^{j} p_k \bar{C}_k \geq \frac{1}{2} \left(p^2(S) + p(S)^2 \right) \geq \frac{1}{2} p(S)^2 . \tag{35.20}$$

Since $\bar{C}_k \leq \bar{C}_j$, for each $k = 1, \ldots, j$, we have

$$\bar{C}_j p(S) = \bar{C}_j \sum_{k=1}^{j} p_k \geq \sum_{k=1}^{j} \bar{C}_k p_k \geq \frac{1}{2} p(S)^2 .$$

Dividing by $p(S)$, we obtain that $\bar{C}_j$ is at least $\frac{1}{2} p(S)$. We therefore see that $\tilde{C}_j \leq 2\bar{C}_j$ and the result follows.

35.6 Polynomial Approximation Schemes Using Enumeration and Rounding

For certain $\mathcal{NP}$-hard scheduling problems there is a limit to our ability to approximate them in polynomial time; for example, Lenstra, Shmoys and Tardos proved that there is no ρ-approximation algorithm, with $\rho < 3/2$, for $R||C_{\max}$ unless $\mathcal{P} = \mathcal{NP}$ [31]. For certain problems, however, we can approximate their optimal solutions arbitrarily closely in polynomial time. In this section we present three *polynomial time approximation schemes (PTAS);* that is, polynomial time algorithms that, for any constant $\rho > 1$, deliver a solution whose objective value is at most ρ times optimal. The running time will depend on ρ—the smaller ρ is, the slower the algorithm will be.

We will present two approaches to the design of such algorithms. The first approach is based on rounding processing times or weights to small integers so that we can apply pseudopolynomial-time algorithms such as that for $1||\Sigma w_j U_j$. A second approach is based on identifying the "important" jobs—those that have the greatest impact on the solution—and processing them separately. In one version, illustrated for $P||C_{\max}$, we round the large jobs so that there are only a constant number of large job sizes, schedule them using dynamic programming, and then schedule the small jobs arbitrarily. In a second version, illustrated for $1|r_j|L_{\max}$, we enumerate all possible schedules for the large jobs, and then fill in the small jobs around them.

From Pseudopolynomial to PTAS: $1||\Sigma w_j U_j$

In Section 35.3, we gave an $O(n\Sigma w_j)$ time algorithm for $1||\Sigma w_j U_j$. Since this gives an algorithm that runs in polynomial time when the weights are polynomial in n, a natural idea is to try to reduce any instance to such a special case. We will scale the weights so that the optimal solution is bounded by a polynomial in n; this will allow us to apply our dynamic programming algorithm to weights of polynomial size.

Assume for now that we know W^*, the value of $\Sigma w_j U_j$ in the optimal schedule. Multiply every weight by $n/(\epsilon W^*)$; now the optimal $\Sigma w_j U_j$ becomes n/ϵ. Clearly, a schedule with $\Sigma w_j U_j$ within a multiplicative $(1+\epsilon)$-factor of optimum under these weights is also within a multiplicative $(1+\epsilon)$-factor of optimum under the original weights. Thus, it suffices to find a schedule with $\Sigma w_j U_j$ at most $(1+\epsilon)n/\epsilon = n/\epsilon + n$ under the new weights.

To do so, increase the weight of every job to the next larger integer. This increases the weight of each job by at most 1 and thus, for any schedule, increases $\Sigma w_j U_j$ by at most n. Under these new weights, $\Sigma w_j U_j$ for the original optimal schedule is now at most $n/\epsilon + n$, so the optimal schedule under these integral weights has $\Sigma w_j U_j \leq n/\epsilon + n$. Since all weights are integers, we can apply the dynamic programming algorithm of Section 35.3 to find an optimal schedule for the rounded instance. Since we only rounded up, the same schedule under the original weights can only have a smaller $\Sigma w_j U_j$. Thus, we find a schedule with weight at most $n/\epsilon + n$ in the (scaled) original weights, i.e., a $(1+\epsilon)$ times optimum schedule.

The running time of our dynamic program is proportional to n times the sum of the (new) weights. This might be a problem, since the weights can be arbitrarily large. However, any job with new weight exceeding $n/\epsilon + n$ *must* be scheduled before its deadline. We therefore identify all such jobs, and modify our dynamic program: T_{wj} becomes the minimum time needed to complete all these jobs that must complete by their deadlines plus other jobs from $1, \ldots, j$ of total weight w. The dynamic programming argument goes through unchanged, but now we consider only jobs of weight $O(n/\epsilon)$. It follows that the largest value of w that we consider is $O(n^2/\epsilon)$, which means that the total running time is $O(n^3/\epsilon)$.

It remains to deal with our assumption that we know W^*. One approach is to use the RDP scheme that performs a binary search on W^*. Of course, we do not expect to arrive at W^* exactly, but note that an estimate will suffice. If we test a value W' with $W^*/\alpha \leq W' \leq W^*$, the analysis above will go through with the running time increased by a factor of α. So we can wait for the RDP binary search to bring us within (say) a constant factor of W^* and then solve the problem.

Of course, if the weights w are extremely large, our binary search could go through many iterations before finding a good value of W'. An elegant trick lets us avoid this problem. We will solve the following problem: find a schedule that minimizes the weight of the maximum-weight late job. The value of this schedule, W', is clearly a lower bound on W^*, as all schedules that minimize $\Sigma w_j U_j$ must have a late job of weight at least W'. Further, W^* is at most nW', since the schedule returned must have at most n late jobs each of which has weight at most W'. Hence our value W' is within a factor of n of optimal. Thus $O(\log n)$ binary search steps suffice to bring us within a constant factor of W^*.

To compute W', we formulate a $1||f_{\max}$ problem. For each job j,

$$f_j(C_j) = \begin{cases} w_j & \text{if } C_j > d_j \\ 0 & \text{if } C_j \leq d_j \end{cases}.$$

This will compute the schedule that minimizes the weight of the maximum weight late job. By the results in the section on "One Machine," we know we can compute this exactly in polynomial time.

THEOREM 35.19 *There exists a $O(n^3(\log n)/\epsilon)$-time $(1+\epsilon)$-approximation algorithm for $1||\Sigma w_j U_j$*

Rounding and Dynamic Programming for $P||C_{\max}$

We now return to the problem of $P||C_{\max}$. Recall that in Lemma 35.2 we gave a polynomial-time algorithm for a special case in which there are a constant number of different job sizes. For the general case, we will focus mainly on the big jobs. We will round and scale these jobs so that there is at most a constant number of sizes of big jobs, and apply the dynamic programming algorithm of Section 35.3 to these rounded jobs. We then finish up by scheduling the small jobs greedily. By the definition of big and small, the overall contribution of the small jobs to the makespan will be negligible.

We will give a $(1+\epsilon)$-RDP for this problem that can be transformed as before into a $(1+\epsilon)$-approximation algorithm. We therefore assume that we have a target optimum schedule length T, We also assume for the rest of this section that ϵT, $\epsilon^2 T$, ϵ^{-1} and ϵ^{-2} are integers. The proofs can easily be modified to handle the case of arbitrary rational numbers.

We first show how to handle the large jobs.

LEMMA 35.4 [18] Let I be an instance of $P||C_{\max}$, let T be a target schedule length, and $\epsilon > 0$. Assume that all $p_j \geq \epsilon T$. Then, for this case, there is a $(1+\epsilon)$-RDP for $P||C_{\max}$.

PROOF We assume $T \geq \max_j p_j$, since otherwise we immediately know that the problem is infeasible. Form instance I' from I with processing times p'_j by rounding each p_j down to an integer multiple of $\epsilon^2 T$. This creates an instance in which

1. $0 \leq p_j - p'_j \leq \epsilon^2 T$
2. There are at most $\frac{T}{\epsilon^2 T} = \frac{1}{\epsilon^2}$ different job sizes,
3. In any feasible schedule, each machine has at most $\frac{T}{\epsilon T} = \frac{1}{\epsilon}$ jobs.

Thus, we can apply Lemma 35.2 to instance I' and obtain an optimal solution to this scheduling problem; let its makespan be D. If $D > T$, then we know that there is no schedule of length $\leq T$ for I, since job sizes in I' are no greater than those in I. In this case we can answer "no schedule of length $\leq T$ exists." If $D \leq T$, then we will answer "there exists a schedule of length $\leq (1+\epsilon)T$." We now show that this answer will be correct. We simply take our schedule for I' and replace the rounded jobs with the original jobs from I. By 1 and 3 in the above list, we add at most $\epsilon^2 T$ to the processing time of each job, and since

there are at most $\frac{1}{\epsilon}$ jobs per machine, we add at most ϵT to the processing time per machine. Thus we can create a schedule with makespan at most $T + \epsilon T = (1 + \epsilon)T$.

We now give the complete algorithm. The idea will be to remove the "small" jobs, use Lemma 35.4 to schedule the remaining jobs, and then add the small jobs back greedily. Given input I_0, target schedule length T, and $\rho = 1 + \epsilon > 1$, we execute the following algorithm.

```
Let R be the set of jobs with p_j ≤ εT. Let I = I_0 − R
Apply Lemma 35.4 to I, T, and ρ.
If this algorithm returns no,
(†)   then output "no schedule of length ≤ T exists."
      else
          for each job j in R
                if there is a machine i with load ≤ T,
                      then add job j to machine i
(*)                   else return "no schedule of length ≤ T exists"
          return "yes, a schedule of length ≤ ρT exists"
```

THEOREM 35.20 *[18] The algorithm above is a ρ-relaxed decision procedure for* $P||C_{\max}$.

PROOF If the algorithm outputs "yes, a schedule of length $\leq \rho T$ exists," then it has constructed such a schedule, and is clearly correct. If the algorithm outputs "no schedule of length $\leq T$ exists" on line (†), then it is because no schedule of length T exists for instance I. But instance I is a subset of the original jobs and so if no schedule exists for I, then no schedule exists for I_0, and the output is correct. If the algorithm outputs "no schedule of length $\leq T$ exists" on line (*), then at this point in the algorithm, every machine must have more than T units of processing on it. Thus, we have that $\Sigma_j p_j > mT$, which means that no schedule of length $\leq T$ exists.

The running time is dominated by the dynamic programming in Lemma 35.2. It is polynomial in n, but the exponent is a polynomial in $1/\epsilon$. While for ρ very close to 1, the running time is prohibitively large, for larger, fixed values of ρ, a modified algorithm yields good schedules with near-linear running times; see [18] for details.

Exhaustive Enumeration for $1|r_j|L_{\max}$

We now turn to the problem of minimizing the maximum lateness in the presence of release dates. Recall from Section 35.2 that without release dates `EDD` is an exact algorithm for this problem. Once we add release dates the problem becomes $\mathcal{NP}$-hard. As we think about approximation algorithms, we come upon an immediate obstacle, namely that the objective function can be 0 or even negative, and hence a solution of value at most $\rho C^*_{\max}$ is clearly impossible. In order to get around this, we must guarantee that the value of the objective function is positive. One simple way to do so is to decrease all the d_j's uniformly by some value δ. This decreases the objective value by exactly δ and does not change the structure of the optimal solution. In particular, if we pick δ large enough so that all the d_j's are negative, we are guaranteed that $L_{\max}$ is positive.

Forcing d_j to be negative is somewhat artificial and so we do not concentrate on this interpretation (note that by taking δ arbitrarily large, we can make any algorithm into an arbitrarily good approximation algorithm). We instead use an equivalent but natural *delivery time* formulation which, in addition to

modeling a number of applications, is a key subroutine in computational approaches to shop scheduling problems [29]. In this formulation, each job, in addition to having a release date r_j and a processing time p_j, has a *delivery time* q_j. A delivery time is an amount of time that must elapse between the completion time of a job on a machine and when it is truly considered finished. Our objective is now to minimize $\max_j\{C_j + q_j\}$. To see the connection to our original problem, note that by setting $q_j = -d_j$ (recall that we made all d_j negative, so all q_j are positive), the delivery-time problem is equivalent to minimizing maximum lateness, and in fact we will overload L_j and define it as $C_j + q_j$.

Jackson's Rule is a 2-Approximation Algorithm

In the delivery-time model, `EDD` translates to Longest Delivery Time First. This is often referred to as *Jackson's rule.* [23]. Let $L^*_{\max}$ be the optimum maximum lateness. The following two lower bounds for this problem are the easily derived analogs of (35.1) and (35.2):

$$L^*_{\max} \geq \sum_j p_j \ , \tag{35.21}$$

$$L^*_{\max} \geq r_j + p_j + q_j \text{ for all } j \ . \tag{35.22}$$

LEMMA 35.5 Jackson's Rule is a 2-approximation algorithm for the delivery time version of $1|r_j|L_{\max}$.

PROOF Let j' be a job for which $L_{j'} = L_{\max}$. Since Jackson's rule creates a schedule with no unforced idle time, we know that there is no idle time between time $r_{j'}$ and $C_{j'}$. Let J' be the set of jobs that run between $r_{j'}$ and $C_{j'}$. Then

$$\begin{aligned} L_{j'} &= C_{j'} + q_{j'} && (35.23)\\ &= r_{j'} + \sum_{j\in J'} p_j + q_{j'} && (35.24)\\ &\leq \left(r_{j'} + q_{j'}\right) + \sum_j p_j && (35.25)\\ &= 2L^*_{\max} \ , && (35.26) \end{aligned}$$

where the last line follows by applying the two lower bounds (35.21) and (35.22).

A PTAS Using Enumeration

The presentation of this section follows that of Hall [16]. The original approximation scheme for this problem is due to Hall and Shmoys [15].

To obtain an algorithm with an improved performance guarantee, we need to look more carefully at when Jackson's rule can go wrong. Let s_j be the starting time of job j, let $r_{\min}(S) = \min_{j\in S} r_j$, let $q_{\min}(S) = \min_{j\in S} q_j$, and recall that $p(S) = \Sigma_{j\in S} p_j$. Then clearly, for any $S \subseteq J$

$$L^*_{\max} \geq r_{\min}(S) + p(S) + q_{\min}(S) \ . \tag{35.27}$$

Now consider a job j' for which $L_{j'} = L_{\max}$. Let t_i be the latest time before $s_{j'}$ at which the machine is idle, and let a be the job that runs immediately after this idle time. Let S be the set of jobs that run between s_a and $C_{j'}$. We call S a *critical section.* Because of the idle time immediately before s_a, we know that for all $j \in S$, $r_j \geq r_a$. In other words we have a set of jobs, all of which were released after time r_a, and which end with the job that achieves $L_{\max}$. Now if for all $j \in S$ $q_j \geq q_{j'}$, then we claim that $Ł_{j'} = L^*_{\max}$. This follows from the fact that

$$L_{\max} = L_{j'} = r_a + p(S) + q_{j'} = r_{\min}(S) + p(S) + q_{\min}(S) \ ,$$

and that the right-hand side, by (35.27), is also a lower bound on $L^*_{\max}$. So, as long as, in a critical section, the job with the shortest delivery time is last, we have an optimal schedule. Thus, if Jackson's rule is not optimal, there must be a job b in the critical section which has $q_b < q_{j'}$. We call the latest-scheduled job in the critical section with $q_b < q_{j'}$ an *interference job.* The following lemma shows the relationship between the interference job and its effect on $L_{\max}$.

LEMMA 35.6 Let b be an interference job in a schedule created by Jackson's rule. Then $L_{\max} < L^*_{\max} + p_b$.

Thus, if interference jobs have small processing times, Jackson's rule does very well. To make sure that this is the case, we will handle the large jobs separately, to ensure that they are not interference jobs, and then use Jackson's rule on the remaining jobs.

Let us assume for now that we know the optimal schedule for instance I. Let s^*_j be the starting time of job j in the optimal schedule, and let $\delta > 0$ be a parameter to be chosen later. Partition the jobs into small jobs $S = \{j : p_j < \delta\}$ and big jobs $B = \{j : p_j \geq \delta\}$. We create instance $\tilde{I}$ as follows: if $j \in S$, then $\tilde{r}_j = r_j$, $\tilde{p}_j = p_j$, and $\tilde{q}_j = q_j$, otherwise, $\tilde{r}_j = s^*_j$, $\tilde{p}_j = p_j$, and $\tilde{q}_j = L^*_{\max}(I) - p_j - s^*_j$. Instance $\tilde{I}$ is no easier than instance I, since we have not decreased any release dates or delivery times. Yet, the optimal schedule for I remains an optimal schedule for $\tilde{I}$, by construction. In $\tilde{I}$ we have given the large jobs a release date equal to their optimal starting time, and a delivery time that is equal to the schedule length minus their completion time, and hence have constrained the large jobs to run exactly when they would run in the optimal schedule for instance I. Thus, in an optimal schedule for $\tilde{I}$, the big jobs run at exactly time $\tilde{r}_j$ and have $L_j = \tilde{r}_j + \tilde{p}_j + \tilde{q}_j = L^*_{\max}$.

Now we claim that if we run Jackson's rule on $\tilde{I}$, the big jobs will not be interference jobs.

LEMMA 35.7 If we run Jackson's rule on $\tilde{I}$, no job $b \in B$ will be an interference job.

PROOF Assume that some job $b \in B$ is an interference job. As above, define the critical section, and jobs a and j'. Since b is an interference job, we know that $\tilde{q}_{j'} > \tilde{q}_b$ and $\tilde{r}_{j'} > \tilde{r}_b$. We also know that $\tilde{r}_b = s^*_b$, and so j' must run after b in the optimal schedule for I. Applying (35.27) to the set consisting of jobs b and j', we get that

$$L^*_{\max} \geq \tilde{r}_b + \tilde{p}_b + \tilde{p}_{j'} + \tilde{q}_{j'} \geq r_b + p_b + p_{j'} + \tilde{q}_b = L^*_{\max} + p_{j'} ,$$

which is a contradiction.

So if we run Jackson's rule on $\tilde{I}$, we get a schedule whose length is at most $L^*_{\max}(I) + \delta$. Choosing $\delta = \epsilon \Sigma_j p_j$, and recalling that $L^*_{\max} \geq \Sigma p_j$, we get a schedule of length at most $(1+\epsilon)L^*_{\max}$. Further, there can be at most $\Sigma_j p_j / (\epsilon \Sigma_j p_j) = 1/\epsilon$ big jobs. The only problem is that we don't know $\tilde{I}$.

We now argue that it is not necessary to know $\tilde{I}$. First, observe that the set of big jobs is purely a function of the input, and ϵ. Now, if we knew the starting times of the big jobs in the optimal schedule for I, we would know $\tilde{I}$, and could run Jackson's rule on the job in S, inserting the big jobs at the appropriate time. This implies a numbering of the big jobs, i.e., each big job j_i is, for some k, the kth job in the schedule for $\tilde{I}$. Thus, we really only need to know k, and not the starting time for job j_i. Thus we just enumerate all possible numberings for the big jobs. There are $n^{1/\epsilon}$ such numberings. Given a numbering, we can run Jackson's rule on the small jobs, and insert the big jobs at the appropriate places in $O(n \log n)$ time, and thus we get an algorithm that in $O(n^{1+1/\epsilon} \log n)$ time finds a schedule with $L_{\max} \leq (1+\epsilon)L^*_{\max}$.

35.7 Research Issues and Summary

In this chapter we have surveyed some of the basic techniques for deterministic scheduling. Scheduling is an old and therefore mature field, but important opportunities for research contributions remain. In addition to some of the outstanding open questions (see the survey by Lawler et al. [29]) it is our feeling that the most meaningful research contributions will be either new and innovative techniques for attacking old problems or new problem definitions that model more realistic applications.

There are other schools of approach to the design of algorithms for scheduling, such as those relying on techniques from artificial intelligence or from computational optimization. It will be quite valuable to forge stronger connections between these different approaches to solving scheduling problems.

35.8 Defining Terms

n**:** Number of jobs.

m**:** Number of machines.

p_j**:** Processing time of job j.

C_j^S**:** Completion time of job j in schedule S.

w_j**:** Weight of job j.

r_j**:** Release date of job j; job j is unavailable for processing before time r_j.

d_j**:** Due date of job j.

$L_j := C_j - d_j$ the lateness of job j.

U_j**:** 1 is job j is scheduled by d_j and 0 otherwise.

$\alpha|\beta|\gamma$**:** Denotes scheduling problem with machine environment α, optimality criterion γ, and side characteristics and constraints denoted by β.

Machine environments:

- 1: One machine.
- P: Parallel identical machines.
- Q: Parallel machines of different speeds.
- R: Parallel unrelated machines.
- O: Open shop.
- F: Flow shop.
- J: Job shop.

Possible characteristics and constraints:

- $pmtn$: Job preemption allowed.
- r_j: Jobs have nontrivial release dates.
- $prec$: Jobs are precedence-constrained.

Optimality criteria:

- ΣC_j: Average (sum) of completion times.
- $\Sigma w_j C_j$: Weighted average (sum) of completion times.
- $C_{\max}$: Makespan (schedule length).
- $L_{\max}$: Maximum lateness over all jobs.
- ΣU_j: Number of on-time jobs.
- $\Sigma w_j U_j$: Weighted number of on-time jobs.

Acknowledgments

We are grateful to Jan Karel Lenstra and David Shmoys for helpful comments.

References

[1] Ahuja, R.K., Magnanti, T.L., and Orlin, J.B., *Network Flows: Theory, Algorithms, and Applications.* Prentice Hall, Englewood Cliffs, NJ, 1993.

[2] Baker, K.R., *Introduction to Sequencing and Scheduling.* John Wiley & Sons, 1974.

[3] Bárány, I. and Fiala, T., Többgépes ütemezési problémák közel optimális megoldása. *Szigma–Mat.–Közgazdasági Folyóirat,* 15, 177–191, 1982.

[4] Bruno, J.L., Coffman, E.G., and Sethi, R., Scheduling independent tasks to reduce mean finishing time. *Communications of the ACM,* 17, 382–387, 1974.

[5] Conway, R.W., Maxwell, W.L., and Miller, L.W., *Theory of Scheduling.* Addison-Wesley, 1967.

[6] Cormen, T.H., Leiserson, C.E., and Rivest, R.L., *Introduction to Algorithms.* MIT Press/McGraw-Hill, 1990.

[7] Garey, M.R. and Johnson, D.S., *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W.H. Freeman and Company, New York, 1979.

[8] Goemans, M., A supermodular relaxation for scheduling with release dates. In *Proceedings of the 5th Conference on Integer Programming and Combinatorial Optimization,* 288–300, Jun. 1996. Published as Lecture Notes in Computer Science 1084, Springer-Verlag.

[9] Gonzalez, T. and Sahni, S., Open shop scheduling to minimize finish time. *Journal of the ACM,* 23, 665–679, 1976.

[10] Gonzalez, T. and Sahni, S., Flowshop and jobshop schedules: complexity and approximation. *Operations Research,* 26, 36–52, 1978.

[11] Graham, R.L., Bounds for certain multiprocessor anomalies. *Bell System Technical Journal,* 45, 1563–1581, 1966.

[12] Graham, R.L., Bounds on multiprocessing anomalies. *SIAM Journal of Applied Mathematics,* 17, 263–269, 1969.

[13] Graham, R.L., Lawler, E.L., Lenstra, J.K., and Rinnooy Kan, A.H.G., Optimization and approximation in deterministic sequencing and scheduling: a survey. *Annals of Discrete Mathematics,* 5, 287–326, 1979.

[14] Gusfield, D., Bounds for naive multiple machine scheduling with release times and deadlines. *Journal of Algorithms,* 5, 1–6, 1984.

[15] Hall, L. and Shmoys, D.B., Approximation schemes for constrained scheduling problems. In *Proceedings of the 30th Annual Symposium on Foundations of Computer Science,* 134–141. IEEE, Oct. 1989.

[16] Hall, L.A., *Approximation Algorithms for NP-Hard Problems,* chapter 1. Hochbaum, D., Ed., PWS Publishing, 1997.

[17] Hall, L.A., Schulz, A.S., Shmoys, D.B., and Wein, J., Scheduling to minimize average completion time: Off-line and on-line approximation algorithms. *Mathematics of Operations Research,* 22, 513–544, Aug. 1997.

[18] Hochbaum, D.S. and Shmoys, D.B., Using dual approximation algorithms for scheduling problems: Theoretical and practical results. *Journal of the ACM,* 34, 144–162, 1987.

[19] Hochbaum, D., Ed., *Approximation Algorithms.* PWS, 1997.

[20] Hoogeveen, J.A., Lenstra, J.K., and van de Velde, S.L., Sequencing and scheduling. In *Annotated Bibliographies in Combinatorial Optimization.* Dell'Amico, M., Maffioli, F., and Martello, S., Eds., John Wiley & Sons, Chichester, U.K., 1997. To appear.

[21] Horn, W., Minimizing average flow time with parallel machines. *Operations Research,* 21, 846–847, 1973.

[22] Horn, W., Some simple scheduling algorithms. *Naval Research Logistics Quarterly,* 21, 177–185, 1974.
[23] Jackson, J.R., Scheduling a production line to minimize maximum tardiness. Management Science Research Project Research Report 43, University of California, Los Angeles, 1955.
[24] Johnson, S.M., Optimal two- and three-stage production schedules with setup times included. *Naval Research Logistics Quarterly,* 61–68, 1954.
[25] Khachiyan, L.G., A polynomial algorithm in linear programming (in Russian). *Doklady Adademiia Nauk SSSR,* 224, 1093–1096, 1979.
[26] Lawler, E.L., Optimal sequencing of a single machine subject to precedence constraints. *Management Science,* 19, 544–546, 1973.
[27] Lawler, E.L., *Combinatorial Optimization: Networks and Matroids.* Holt, Rinehart and Winston, 1976.
[28] Lawler, E.L., and Labetoulle, J., On preemptive scheduling of unrelated parallel processors by linear programming. *Journal of the ACM,* 25, 612–619, 1978.
[29] Lawler, E.L., Lenstra, J.K., Rinnooy Kan, A.H.G., and Shmoys, D.B., Sequencing and scheduling: Algorithms and complexity. In *Handbooks in Operations Research and Management Science,* Graves, S.C., Rinnooy Kan, A.H.G., and Zipkin, P.H., Eds., Vol 4., *Logistics of Production and Inventory,* 445–522. North-Holland, 1993.
[30] Lawler, E.L. and Moore, J.M., A functional equation and its application to resource allocation and sequencing problems. *Management Science,* 77–84, 1969.
[31] Lenstra, J.K., Shmoys, D.B., and Tardos, É., Approximation algorithms for scheduling unrelated parallel machines. *Mathematical Programming,* 46, 259–271, 1990.
[32] McNaughton, R., Scheduling with deadlines and loss functions. *Management Science,* 6, 1–12, 1959.
[33] Moore, J.M., An n-job, one machine sequencing algorithm for minimizing the number of late jobs. *Management Science,* 15, 102–109, 1968.
[34] Pinedo, M., *Scheduling: Theory, Algorithms and Systems.* Prentice Hall, 1995.
[35] Phillips, C., Stein, C., and Wein, J., Scheduling jobs that arrive over time. In *Algorithms and Data Structures,* Selim G. Akl, Ed., number 955 in Lecture Notes in Computer Science, 86–97, Berlin, 1995. Springer-Verlag. Journal version to appear in *Mathematical Programming B.*
[36] Queyranne, M. and Schulz, A.S., Polyhedral approaches to machine scheduling. Technical Report 474/1995, Technical University of Berlin, 1994.
[37] Queyranne, M., Structure of a simple scheduling polyhedron. *Mathematical Programming,* 58, 263–285, 1993.
[38] Sgall, J., On-line scheduling—a survey. In *On-Line Algorithms, Lecture Notes in Computer Science.* Fiat, A. and Woeginger, G., Eds., Springer-Verlag, Berlin, 1997. To appear.
[39] Smith, W.E., Various optimizers for single-stage production. *Naval Research Logistics Quarterly,* 3, 59–66, 1956.
[40] Ullman, J.D., NP-complete scheduling problems. *Journal of Computer and System Sciences,* 10, 384–393, 1975.
[41] Wolsey, L.A., Mixed integer programming formulations for production planning and scheduling problems. Invited talk at the 12th International Symposium on Mathematical Programming, MIT Press, Cambridge, 1985.

Further Information

We conclude by reminding the reader what this chapter is not. In no way is this chapter a comprehensive survey of even the most basic and classical results in scheduling theory, and it is certainly not an up-to-date survey on the field. It also essentially entirely ignores "nontraditional" models, and does not touch on

stochastic scheduling or on any of the other approaches to scheduling and resource allocation. The reader interested in a comprehensive survey of the field should consult the textbook by Pinedo [34] and the survey by Lawler et al. [29]. These sources provide pointers to a number of other references. In addition, we also recommend an annotated bibliography by Hoogeveen et al. that contains information on recent results in scheduling theory [20], the surveys by Queyranne and Schulz on polyhedral formulations [36], by Hall on approximation algorithms [16], and by Sgall on online scheduling [38]. Research on deterministic scheduling theory is published in many journals; for example see *Mathematics of Operations Research, Operations Research, SIAM Journal on Computing,* and *Journal of the ACM.*

36

Artificial Intelligence Search Algorithms[1]

Richard E. Korf
University of California, Los Angeles

36.1 Introduction

Search is a universal problem-solving mechanism in artificial intelligence (AI). In AI problems, the sequence of steps required for solution of a problem are not known *a priori,* but often must be determined by a systematic trial-and-error exploration of alternatives. The problems that have been addressed by AI

[1]This work was supported in part by NSF Grant IRI-9619447, and a grant from Rockwell International.

search algorithms fall into three general classes: **single-agent pathfinding problems, two-player games,** and **constraint-satisfaction problems.**

Classic examples in the AI literature of pathfinding problems are the sliding-tile puzzles, including the 3×3 Eight Puzzle (see Fig. 36.1) and its larger relatives the 4×4 Fifteen Puzzle, and 5×5 Twenty-Four Puzzle. The Eight Puzzle consists of a 3×3 square frame containing eight numbered square tiles, and an empty position called the blank. The legal operators are to slide any tile that is horizontally or vertically adjacent to the blank into the blank position. The problem is to rearrange the tiles from some random initial configuration into a particular desired goal configuration. The sliding-tile puzzles are common testbeds for research in AI search algorithms because they are very simple to represent and manipulate, yet finding optimal solutions to the $N \times N$ generalization of the sliding-tile puzzles is NP-complete [43]. Other well-known examples of single-agent pathfinding problems include Rubik's Cube and theorem proving. Real-world examples include the Travelling Salesman Problem, vehicle navigation, and the wiring of VLSI circuits. In each case, the task is to find a sequence of operations that map an initial state to a goal state.

A second class of search problems include two-player perfect-information games, such as chess, checkers, and Othello. The third category is constraint-satisfaction problems, such as the Eight Queens Problem. The task is to place eight queens on an 8×8 chessboard, such that no two queens are on the same row, column or diagonal. Real-world examples of constraint-satisfaction problems are ubiquitous, including planning and scheduling applications.

We begin by describing the problem-space model on which search algorithms are based. Brute-force searches are then considered including breadth-first, uniform-cost, depth-first, depth-first iterative-deepening, and bidirectional search. Next, various heuristic searches are examined including pure heuristic search, the A* algorithm, iterative-deepening-A*, depth-first branch-and-bound, the heuristic path algorithm, and recursive best-first search. We then consider single-agent algorithms that interleave search and execution, including minimin lookahead search, real-time-A*, and learning-real-time-A*. Next, we consider two-player game searches, including minimax and alpha-beta pruning. Finally, we examine constraint-satisfaction algorithms, such as backtracking, constraint recording, and heuristic repair. The efficiency of these algorithms, in terms of the costs of the solutions they generate, the amount of time the algorithms take to execute, and the amount of computer memory they require are of central concern throughout. Since search is a universal problem-solving method, what limits its applicability is the efficiency with which it can be performed.

36.2 Problem Space Model

A *problem space* is the environment in which a search takes place [34]. A problem space consists of a set of *states* of the problem, and a set of *operators* that change the state. For example, in the Eight Puzzle, the states are the different possible permutations of the tiles, and the operators slide a tile into the blank position. A *problem instance* is a problem space together with an initial state and a goal state. In the case of the Eight Puzzle, the initial state would be whatever initial permutation the puzzle starts out in, and the goal state is a particular desired permutation. The problem-solving task is to find a sequence of operators that map the initial state to a goal state. In the Eight Puzzle the goal state is given explicitly. In other problems, such as the Eight Queens Problem, the goal state is not given explicitly, but rather implicitly specified by certain properties that must be satisfied by a goal state.

A **problem-space graph** is often used to represent a problem space. The states of the space are represented by **nodes** of the graph, and the operators by **edges** between nodes. Edges may be undirected or directed, depending on whether their corresponding operators are reversible or not. The task in a single-agent path-finding problem is to find a path in the graph from the initial node to a goal node. Figure 36.1 shows a small part of the Eight Puzzle problem-space graph.

Although most problem spaces correspond to graphs with more than one path between a pair of nodes, for simplicity they are often represented as trees, where the initial state is the root of the tree. The cost

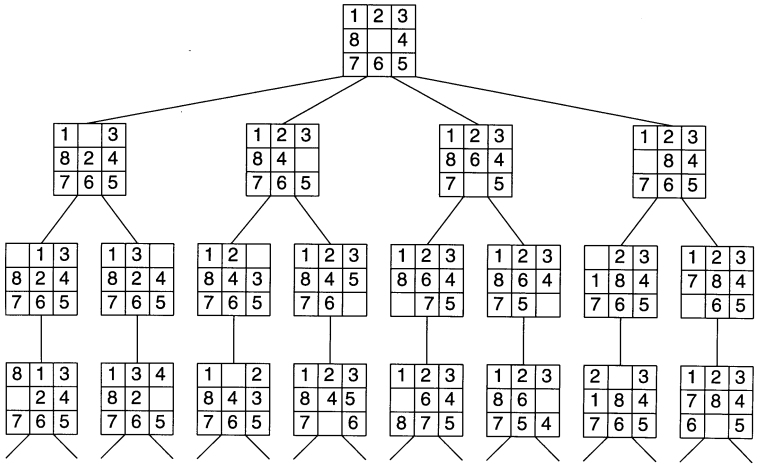


FIGURE 36.1 Eight puzzle search tree fragment.

of this simplification is that any state that can be reached by two different paths will be represented by duplicate nodes, increasing the size of the tree. The benefit of a tree is that the absence of cycles greatly simplifies many search algorithms. In this survey, we will restrict our attention to trees, but there exist graph versions of most of the algorithms we describe as well.

One feature that distinguishes AI search algorithms from other graph-searching algorithms is the size of the graphs involved. For example, the entire chess graph is estimated to contain over 10^{40} nodes. Even a simple problem like the Twenty-Four Puzzle contains almost 10^{25} nodes. As a result, the problem-space graphs of AI problems are never represented explicitly by listing each state, but rather are implicitly represented by specifying an initial state and a set of operators to generate new states from existing states. Furthermore, the size of an AI problem is rarely expressed as the number of nodes in its problem-space graph. Rather, the two parameters of a search tree that determine the efficiency of various search algorithms are its *branching factor* and its *solution depth.* The branching factor is the average number of children of a given node. For example, in the Eight Puzzle the average branching factor is $\sqrt{3}$, or about 1.732. The solution depth of a problem instance is the length of a shortest path from the initial state to a goal state, or the length of a shortest sequence of operators that solves the problem. If the goal were in the bottom row of Fig. 36.1, the depth of the problem instance represented by the initial state at the root would be three moves.

36.3 Brute-Force Search

The most general search algorithms are *brute-force* searches, since they do not require any domain-specific knowledge. All that is required for a brute-force search is a state description, a set of legal operators, an initial state, and a description of the goal state. The most important brute-force techniques are breadth-first, uniform-cost, depth-first, depth-first iterative-deepening, and bidirectional search. In the descriptions of the algorithms below, to *generate* a node means to create the data structure corresponding to that node, whereas to *expand* a node means to generate all the children of that node.

Breadth-First Search

Breadth-first search expands nodes in order of their distance from the root, generating one level of the tree at a time until a solution is found (see Fig. 36.2). It is most easily implemented by maintaining a queue of nodes, initially containing just the root, and always removing the node at the head of the queue, expanding it, and adding its children to the tail of the queue.

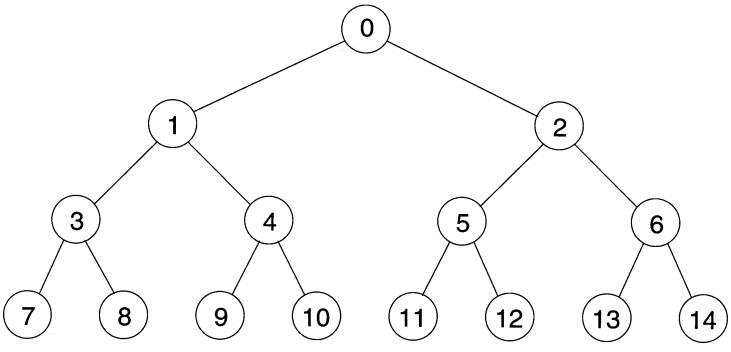


FIGURE 36.2 Order of node generation for breadth-first search.

Since it never generates a node in the tree until all the nodes at shallower levels have been generated, breadth-first search always finds a shortest path to a goal. Since each node can be generated in constant time, the amount of time used by breadth-first search is proportional to the number of nodes generated, which is a function of the branching factor b and the solution depth d. Since the number of nodes at level d is b^d, the total number of nodes generated in the worst case is $b + b^2 + b^3 + \cdots + b^d$, which is $O(b^d)$, the asymptotic time complexity of breadth-first search.

The main drawback of breadth-first search is its memory requirement. Since each level of the tree must be saved in order to generate the next level, and the amount of memory is proportional to the number of nodes stored, the space complexity of breadth-first search is also $O(b^d)$. As a result, breadth-first search is severely space-bound in practice, and will exhaust the memory available on typical computers in a matter of minutes.

Uniform-Cost Search

If all edges do not have the same cost, then breadth-first search generalizes to *uniform-cost search.* Instead of expanding nodes in order of their depth from the root, uniform-cost search expands nodes in order of their cost from the root. At each step, the next node n to be expanded is one whose cost $g(n)$ is lowest, where $g(n)$ is the sum of the edge costs from the root to node n. The nodes are stored in a priority queue. This algorithm is also known as Dijkstra's single-source shortest-path algorithm [6].

Whenever a node is chosen for expansion by uniform-cost search, a lowest-cost path to that node has been found. The worst-case time complexity of uniform-cost search is $O(b^{c/m})$, where c is the cost of an optimal solution, and m is the minimum edge cost. Unfortunately, it also suffers the same memory limitation as breadth-first search.

Depth-First Search

Depth-first search remedies the space limitation of breadth-first search by always generating next a child of the deepest unexpanded node (see Fig. 36.3). Both algorithms can be implemented using a list of unexpanded nodes, with the difference that breadth-first search manages the list as a first-in first-out queue, whereas depth-first search treats the list as a last-in first-out stack. More commonly, depth-first search is implemented recursively, with the recursion stack taking the place of an explicit node stack.

The advantage of depth-first search is that its space requirement is only linear with respect to the search depth, as opposed to exponential for breadth-first search. The reason is that the algorithm only needs to store a stack of nodes on the path from the root to the current node. The time complexity of a depth-first search to depth d is $O(b^d)$, since it generates the same set of nodes as breadth-first search, but simply in a different order. Thus, as a practical matter, depth-first search is time-limited rather than space-limited.

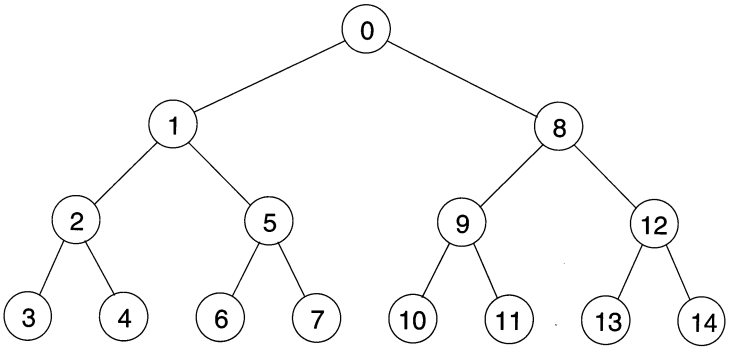


FIGURE 36.3 Order of node generation for depth-first search.

The disadvantage of depth-first search is that it may not terminate on an infinite tree, but simply go down the left-most path forever. Even a finite graph can generate an infinite tree. The usual solution to this problem is to impose a cutoff depth on the search. Although the ideal cutoff is the solution depth d, this value is rarely known in advance of actually solving the problem. If the chosen cutoff depth is less than d, the algorithm will fail to find a solution, whereas if the cutoff depth is greater than d, a large price is paid in execution time, and the first solution found may not be an optimal one.

Depth-First Iterative-Deepening

Depth-first iterative-deepening (DFID) combines the best features of breadth-first and depth-first search [17, 48]. DFID first performs a depth-first search to depth one, then starts over, executing a complete depth-first search to depth two, and continues to run depth-first searches to successively greater depths, until a solution is found (see Fig. 36.4).

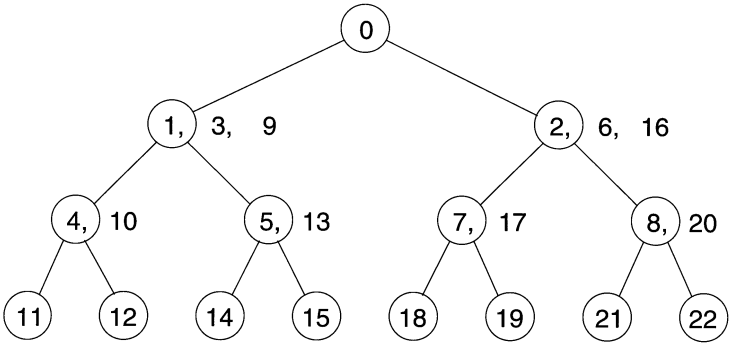


FIGURE 36.4 Order of node generation for depth-first iterative-deepening search.

Since it never generates a node until all shallower nodes have been generated, the first solution found by DFID is guaranteed to be along a shortest path. Furthermore, since at any given point it is executing a depth-first search, saving only a stack of nodes, and the algorithm terminates when it finds a solution at depth d, the space complexity of DFID is only $O(d)$.

Although it appears that DFID wastes a great deal of time in the iterations prior to the one that finds a solution, this extra work is usually insignificant. To see this, note that the number of nodes at depth d is b^d, and each of these nodes are generated once, during the final iteration. The number of nodes at depth $d-1$ is b^{d-1}, and each of these are generated twice, once during the final iteration, and once during the

penultimate iteration. In general, the number of nodes generated by DFID is $b^d + 2b^{d-1} + 3b^{d-2} + \cdots + db$. This is asymptotically $O(b^d)$ if b is greater than one, since for large values of d the lower order terms become insignificant. In other words, most of the work goes into the final iteration, and the cost of the previous iterations is relatively small. The ratio of the number of nodes generated by DFID to those generated by breadth-first search on a tree is approximately $b/(b-1)$. In fact, DFID is asymptotically optimal in terms of time and space among all brute-force shortest-path algorithms on a tree [17].

If the edge costs differ from one another, then one can run an iterative deepening version of uniform-cost search, where the depth cutoff is replaced by a cutoff on the $g(n)$ cost of a node. At the end of each iteration, the threshold for the next iteration is set to the minimum cost of all nodes generated on the previous iteration whose cost exceeded the previous threshold.

On a graph with cycles, however, breadth-first search may be much more efficient than any depth-first search. The reason is that a breadth-first search can check for duplicate nodes whereas a depth-first search cannot. Thus, the complexity of breadth-first search grows only as the number of nodes at a given depth, while the complexity of depth-first search depends on the number of paths of a given length. For example, in a square grid, the number of nodes within a radius r of the origin is $O(r^2)$, whereas the number of paths of length r is $O(3^r)$, since there are three children of every node, not counting its parent. Thus, in a graph with a large number of very short cycles, breadth-first search is preferable to depth-first search, if sufficient memory is available. For two approaches to the problem of pruning duplicate nodes in depth-first search, see [49] and [7].

Bidirectional Search

Bidirectional search is a brute-force algorithm that requires an explicit goal state instead of simply a test for a goal condition [40]. The main idea is to simultaneously search forward from the initial state, and backward from the goal state, until the two search frontiers meet. The path from the initial state is then concatenated with the inverse of the path from the goal state to form the complete solution path.

Bidirectional search still guarantees optimal solutions. Assuming that the comparisons for identifying a common state between the two frontiers can be done in constant time per node, by hashing for example, the time complexity of bidirectional search is $O(b^{d/2})$ since each search need only proceed to half the solution depth. Since at least one of the searches must be breadth-first in order to find a common state, the space complexity of bidirectional search is also $O(b^{d/2})$. As a result, bidirectional search is space bound in practice.

Combinatorial Explosion

The problem with all brute-force search algorithms is that their time complexities grow exponentially with problem size. This is called *combinatorial explosion,* and as a result, the size of problems that can be solved with these techniques is quite limited. For example, while the Eight Puzzle, with about 10^5 states, is easily solved by brute-force search, the Fifteen Puzzle contains over 10^{13} states, and hence cannot be solved with brute-force techniques on current machines. Faster machines will not have a significant impact on this problem, since the 5×5 Twenty-Four Puzzle contains almost 10^{25} states, for example.

36.4 Heuristic Search

In order to solve larger problems, domain-specific knowledge must be added to improve search efficiency. In AI, *heuristic search* has a general meaning, and a more specialized technical meaning. In a general sense, the term *heuristic* is used for any advice that is often effective, but is not guaranteed to work in every case. Within the heuristic search literature, however, the term heuristic usually refers to the special case of a *heuristic evaluation function.*

Heuristic Evaluation Functions

In a single-agent path-finding problem, a heuristic evaluation function estimates the cost of an optimal path between a pair of states. For example, Euclidean or airline distance is an estimate of the highway distance between a pair of locations. A common heuristic function for the sliding-tile puzzles is called Manhattan distance. It is computed by counting the number of moves along the grid that each tile is displaced from its goal position, and summing these values over all tiles. For a fixed goal state, a heuristic evaluation is a function of a node, $h(n)$, that estimates the distance from node n to the given goal state.

The key properties of a heuristic evaluation function are that it estimate actual cost, and that it be inexpensive to compute. For example, the Euclidean distance between a pair of points can be computed in constant time. The Manhattan distance between a pair of states can be computed in time proportional to the number of tiles. In addition, most heuristic functions are derived from relaxations of the original problem, and hence are lower bounds on actual cost, a property referred to as *admissibility.* For example, airline distance is a lower bound on road distance between two points, since the shortest path between a pair of points is a straight line. Similarly, Manhattan distance is a lower bound on the actual number of moves necessary to solve an instance of a sliding-tile puzzle, since every tile must move at least as many times as its distance in grid units from its goal position.

A number of algorithms make use of heuristic functions, including pure heuristic search, the A* algorithm, iterative-deepening-A*, depth-first branch-and-bound, and the heuristic path algorithm. In addition, heuristic information can be employed in bidirectional search as well.

Pure Heuristic Search

The simplest of these algorithms, pure heuristic search, expands nodes in order of their heuristic values $h(n)$ [9]. It maintains a *Closed list* of those nodes that have already been expanded, and an *Open list* of those nodes that have been generated but not yet expanded. The algorithm begins with just the initial state on the Open list. At each cycle, a node on the Open list with the minimum $h(n)$ value is expanded, generating all of its children, and is placed on the Closed list. The heuristic function is applied to the children, and they are placed on the Open list in order of their heuristic values. The algorithm continues until a goal state is chosen for expansion.

In a graph with cycles, multiple paths will be found to the same node, and the first path found may not be the shortest. When a shorter path is found to an Open node, the shorter path is saved and the longer one discarded. When a shorter path to a Closed node is found, the node is moved to Open, and the shorter path is associated with it. The main drawback of pure heuristic search is that since it ignores the cost of the path so far to node n, it does not find optimal solutions.

Breadth-first search, uniform-cost search, and pure heuristic search are all special cases of a more general algorithm called *best-first search.* In each cycle of a best-first search, the node that is best according to some cost function is chosen for expansion. These best-first algorithms differ only in their cost functions: the depth of node n for breadth-first search, $g(n)$ for uniform-cost search, and $h(n)$ for pure heuristic search.

A* Algorithm

The *A* algorithm* [13] combines features of uniform-cost search and pure heuristic search to efficiently compute optimal solutions. A* is a best-first search in which the cost associated with a node is $f(n) = g(n) + h(n)$, where $g(n)$ is the cost of the path from the initial state to node n, and $h(n)$ is the heuristic estimate of the cost of a path from node n to a goal. Thus, $f(n)$ estimates the lowest total cost of any solution path going through node n. At each point a node with lowest f value is chosen for expansion. Ties among nodes of equal f value should be broken in favor of nodes with lower h values. The algorithm terminates when a goal node is chosen for expansion.

A* finds an optimal path to a goal if the heuristic function $h(n)$ is admissible, meaning it never overestimates actual cost [13]. For example, since airline distance never overestimates actual highway distance, and Manhattan distance never overestimates actual moves in the sliding-tile puzzles, A* using these evaluation functions will find optimal solutions to these problems. In addition, A* makes the most efficient use of a given heuristic function in the following sense: among all shortest-path algorithms using a given heuristic function $h(n)$, A* expands the fewest number of nodes [4].

The main drawback of A*, and indeed of any best-first search, is its memory requirement. Since at least the entire Open list must be saved, A* is severely space-limited in practice, and is no more practical than breadth-first search on current machines. For example, while it can be run successfully on the Eight Puzzle, it exhausts available memory in a matter of minutes on the Fifteen Puzzle.

Iterative-Deepening-A*

Just as depth-first iterative-deepening solved the space problem of breadth-first search, *iterative-deepening-A** (IDA*) eliminates the memory constraint of A*, without sacrificing solution optimality [17]. Each iteration of the algorithm is a depth-first search that keeps track of the cost, $f(n) = g(n) + h(n)$, of each node generated. As soon as a node is generated whose cost exceeds a threshold for that iteration, its path is cut off, and the search backtracks before continuing. The cost threshold is initialized to the heuristic estimate of the initial state, and in each successive iteration is increased to the total cost of the lowest-cost node that was pruned during the previous iteration. The algorithm terminates when a goal state is reached whose total cost does not exceed the current threshold.

Since IDA* performs a series of depth-first searches, its memory requirement is linear with respect to the maximum search depth. In addition, if the heuristic function is admissible, IDA* finds an optimal solution. Finally, by an argument similar to that presented for DFID, IDA* expands the same number of nodes, asymptotically, as A* on a tree, provided that the number of nodes grows exponentially with solution cost. These facts, together with the optimality of A*, imply that IDA* is asymptotically optimal in time and space over all heuristic search algorithms that find optimal solutions on a tree. Additional benefits of IDA* are that it is much easier to implement, and often runs faster than A*, since it does not incur the overhead of managing the Open and Closed lists.

Depth-First Branch-and-Bound

For many problems, the maximum search depth is known in advance, or the search tree is finite. For example, consider the Traveling Salesman Problem (TSP) of visiting each of a given set of cities and returning to the starting city in a tour of shortest total distance. The most natural problem space for this problem consists of a tree where the root node represents the starting city, the nodes at level one represent all the cities that could be visited first, the nodes at level two represent all the cites that could be visited second, etc. In this tree, the maximum depth is the number of cities, and all candidate solutions occur at this depth. In such a space, a simple depth-first search guarantees finding an optimal solution using space that is only linear with respect to the number of cities.

The idea of depth-first branch-and-bound (DFBnB) is to make this search more efficient by keeping track of the lowest-cost solution found so far. Since the cost of a partial tour is the sum of the costs of the edges traveled so far, whenever a partial tour is found whose cost equals or exceeds the cost of the best complete tour found so far, the branch representing the partial tour can be pruned, since all its descendents must have equal or greater cost. Whenever a lower-cost complete tour is found, the cost of the best tour is updated to this lower cost. In addition, an admissible heuristic function, such as the cost of the minimum spanning tree of the remaining unvisited cities, can be added to the cost so far of a partial tour to increase the amount of pruning. Finally, by carefully ordering the children of a given node from smallest to largest estimated total cost, a lower-cost solution can be found more quickly, further improving the pruning efficiency.

Interestingly, IDA* and DFBnB exhibit complementary behavior. Both are guaranteed to return an optimal solution using only linear space, assuming that their cost functions are admissible. In IDA*, the cost threshold is always a lower bound on the optimal solution cost, and increases in each iteration until it reaches the optimal cost. In DFBnB, the cost of the best solution found so far is always an upper bound on the optimal solution cost, and decreases until it reaches the optimal cost. While IDA* never expands any nodes whose cost exceeds the optimal cost, its overhead consists of expanding some nodes more than once. While DFBnB never expands any node more than once, its overhead consists of expanding some nodes whose cost exceeds the optimal cost. For problems whose search trees are of bounded depth, or for which it is easy to construct a good solution, such as the TSP, DFBnB is usually the algorithm of choice for finding an optimal solution. For problems with infinite search trees or for which it is difficult to construct a low-cost solution, such as the sliding-tile puzzles or Rubik's Cube, IDA* is usually the best choice.

Complexity of Finding Optimal Solutions

The time complexity of a heuristic search algorithm depends on the accuracy of the heuristic function. For example, if the heuristic evaluation function is an exact estimator, then A* runs in linear time, expanding only those nodes on an optimal solution path. Conversely, with a heuristic that returns zero everywhere, A* becomes uniform-cost search, which has exponential complexity.

In general, the time complexity of A* and IDA* is an exponential function of the error in the heuristic function [36]. For example, if the heuristic has constant absolute error, meaning that it never underestimates by more than a constant amount regardless of the magnitude of the estimate, then the running time of A* is linear with respect to the solution cost [11]. A more realistic assumption is constant relative error, which means that the error is a fixed percentage of the quantity being estimated. In that case, the running times of A* and IDA* are exponential [38]. The base of the exponent, however, is smaller than the brute-force branching factor, reducing the asymptotic complexity and allowing larger problems to be solved. For example, using appropriate admissible heuristic functions, IDA* can optimally solve random instances of the Twenty-Four Puzzle [26] and Rubik's Cube [27].

Heuristic Path Algorithm

Since the complexity of finding optimal solutions to these problems is generally exponential in practice, in order to solve significantly larger problems, the optimality requirement must be relaxed. An early approach to this problem was the *heuristic path algorithm (HPA)* [39]. HPA is a best-first search algorithm, where the figure of merit of a node n is $f(n) = (1 - w) * g(n) + w * h(n)$. Varying w produces a range of algorithms from uniform-cost search ($w = 0$), through A* ($w = 1/2$), to pure heuristic search ($w = 1$). Increasing w beyond $1/2$ generally decreases the amount of computation, while increasing the cost of the solution generated. This tradeoff is often quite favorable, with small increases in solution cost yielding huge savings in computation [22]. Furthermore, it can be shown that the solutions found by this algorithm are guaranteed to be no more than a factor of $w/(1-w)$ greater than optimal [3], but often are significantly better.

Recursive Best-First Search

The memory limitation of the heuristic path algorithm can be overcome simply by replacing the best-first search with IDA* using the same weighted evaluation function. However, with $w \geq 1/2$, IDA* is no longer a best-first search, since the total cost of a child can be less than that of its parent, and thus nodes are not necessarily expanded in best-first order. An alternative algorithm is recursive best-first search (RBFS) [22]. RBFS is a best-first search that runs in space that is linear with respect to the maximum search depth, regardless of the cost function used. Even with an admissible cost function, RBFS generates fewer nodes than IDA*, and is generally superior to IDA*, except for a small increase in the cost per node generation.

It works by maintaining on the recursion stack the complete path to the current node being expanded, as well as all immediate siblings of nodes on that path, along with the cost of the best node in the subtree explored below each sibling. Whenever the cost of the current node exceeds that of some other node in the previously expanded portion of the tree, the algorithm backs up to their deepest common ancestor, and continues the search down the new path. In effect, the algorithm maintains a separate threshold for each subtree diverging from the current search path. See [22] for full details on RBFS.

36.5 Interleaving Search and Execution

In the discussion above, it is assumed that a complete solution can be computed, before even the first step of the solution need be executed. This is in contrast to the situation in two-player games, discussed below, where because of computational limits and uncertainty due to the opponent's moves, search and execution are interleaved, with each search determining only the next move to be made. This paradigm is also applicable to single-agent problems. In the case of autonomous vehicle navigation, for example, information is limited by the horizon of the vehicle's sensors, and it must physically move to acquire more information. Thus, one move must be computed at a time, and that move executed before computing the next. Below we consider algorithms designed for this scenario.

Minimin Search

Minimin search determines individual single-agent moves in constant time per move [19]. The algorithm searches forward from the current state to a fixed depth determined by the informational or computational resources available. At the search horizon, the A* evaluation function $f(n) = g(n) + h(n)$ is applied to the frontier nodes. Since all decisions are made by a single agent, the value of an interior node is the minimum of the frontier values in the subtree below the node. A single move is then made to the neighbor of the current state with the minimum value.

Most heuristic functions obey the triangle inequality characteristic of distance measures. As a result, $f(n) = g(n) + h(n)$ is guaranteed to be monotonically nondecreasing along a path. Furthermore, since minimin search has a fixed depth limit, we can apply depth-first branch-and-bound to prune the search tree. The performance improvement due to branch-and-bound is quite dramatic, in some cases extending the achievable search horizon by a factor of five relative to brute-force minimin search on sliding-tile puzzles [19].

Minimin search with branch-and-bound is an algorithm for evaluating the immediate neighbors of the current node. As such, it is run until the best child is identified, at which point the chosen move is executed in the real world. We can view the static evaluation function combined with lookahead search as simply a more accurate, but computationally more expensive, heuristic function. In fact, it provides an entire spectrum of heuristic functions trading off accuracy for cost, depending on the search horizon.

Real-Time-A*

Simply repeating minimin search for each move ignores information from previous searches and results in infinite loops. In addition, since actions are committed based on limited information, often the best move may be to undo the previous move. The principle of rationality is that backtracking should occur when the estimated cost of continuing the current path exceeds the cost of going back to a previous state, plus the estimated cost of reaching the goal from that state. *Real-time-A** (RTA*) implements this policy in constant time per move on a tree [19].

For each move, the $f(n) = g(n) + h(n)$ value of each neighbor of the current state is computed, where $g(n)$ is now the cost of the edge from the current state to the neighbor, instead of from the initial state. The problem solver moves to the neighbor with the minimum $f(n)$ value, and stores with the previous state the

best $f(n)$ value among the remaining neighbors. This represents the $h(n)$ value of the previous state from the perspective of the new current state. This is repeated until a goal is reached. To determine the $h(n)$ value of a previously visited state, the stored value is used, while for a new state the heuristic evaluator is called. Note that the heuristic evaluator may employ minimin lookahead search with branch-and-bound as well.

In a finite problem space in which there exists a path to a goal from every state, RTA* is guaranteed to find a solution, regardless of the heuristic evaluation function [19]. Furthermore, on a tree, RTA* makes locally optimal decisions given the information it has seen so far.

Learning-Real-Time-A*

If a problem is to be solved repeatedly with the same goal state but different initial states, one would like an algorithm that improves its performance over time. Learning-real-time-A* (LRTA*) is such an algorithm. It behaves almost identically to RTA*, except that instead of storing the second-best f value of a node as its new heuristic value, it stores the best value instead. Once one problem instance is solved, the stored heuristic values are saved and become the initial values for the next problem instance. While LRTA* is less efficient than RTA* for solving a single problem instance, if it starts with admissible initial heuristic values, over repeated trials its heuristic values eventually converge to their exact values, at which point the algorithm returns optimal solutions.

36.6 Two-Player Games

The second major application of heuristic search algorithms in AI is two-player games. One of the original challenges of AI, which in fact predates the term "artificial intelligence," was to build a program that could play chess at the level of the best human players [50], a goal recently achieved.

Minimax Search

The standard algorithm for two-player perfect-information games, such as chess, checkers, or Othello, is minimax search with heuristic static evaluation [46]. The algorithm searches forward to a fixed depth in the game tree, limited by the amount of time available per move. At this *search horizon,* a heuristic function is applied to the frontier nodes. In this case, a heuristic evaluation is a function that takes a board position and returns a number that indicates how favorable that position is for one player relative to the other. For example, a very simple heuristic evaluator for chess would count the total number of pieces on the board for one player, appropriately weighted by their relative strength, and subtract the weighted sum of the opponent's pieces. Thus, large positive values would correspond to strong positions for one player, called MAX, whereas large negative values would represent advantageous situations for the opponent, called MIN.

Given the heuristic evaluations of the frontier nodes, values for the interior nodes in the tree are recursively computed according to the minimax rule. The value of a node where it is MAX's turn to move is the maximum of the values of its children, while the value of a node where MIN is to move is the minimum of the values of its children. Thus, at alternate levels of the tree, the minimum or the maximum values of the children are backed up. This continues until the values of the immediate children of the current position are computed, at which point one move to the child with the maximum or minimum value is made, depending on whose turn it is to move.

Alpha-Beta Pruning

One of the most elegant of all AI search algorithms is alpha-beta pruning. While it was in use in the late 1950s, a thorough treatment of the algorithm can be found in [28]. The idea, similar to branch-and-bound, is that the minimax value of the root of a game tree can be determined without examining all the nodes at the search frontier.

Figure 36.5 shows an example of alpha-beta pruning. Only the labeled nodes are generated by the algorithm, with the heavy black lines indicating pruning. At the square nodes MAX is to move, while at the circular nodes it is MIN's turn. The search proceeds depth-first to minimize the memory required, and only evaluates a node when necessary. First, nodes e and f are statically evaluated at 4 and 5, respectively, and their minimum value, 4, is backed up to their parent node d. Node h is then evaluated at 3, and hence the value of its parent node g must be less than or equal to 3, since it is the minimum of 3 and the unknown value of its right child. Thus, we label node g as $<= 3$. The value of node c must be 4 then, because it is the maximum of 4 and a value that is less than or equal to 3. Since we have determined the minimax value of node c, we do not need to evaluate or even generate the brother of node h.

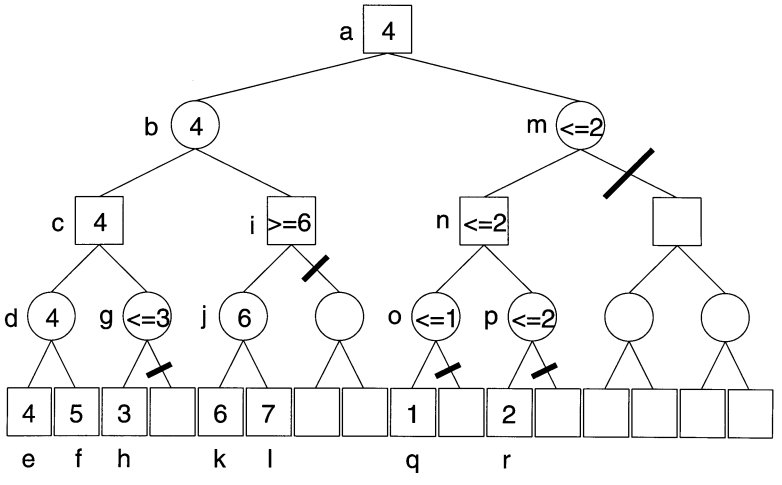


FIGURE 36.5 Alpha-beta pruning example.

Similarly, after statically evaluating nodes k and l at 6 and 7, respectively, the backed up value of their parent node j is 6, the minimum of these values. This tells us that the minimax value of node i must be greater than or equal to 6, since it is the maximum of 6 and the unknown value of its right child. Since the value of node b is the minimum of 4 and a value that is greater than or equal to 6, it must be 4, and hence we achieve another cutoff.

The right half of the tree shows an example of *deep pruning*. After evaluating the left half of the tree, we know that the value of the root node a is greater than or equal to 4, the minimax value of node b. Once node q is evaluated at 1, the value of its parent node o must be less than or equal to 1. Since the value of the root is greater than or equal to 4, the value of node o cannot propagate to the root, and hence we need not generate the brother of node q. A similar situation exists after the evaluation of node r at 2. At that point, the value of node o is less than or equal to 1, and the value of node p is less than or equal to 2, hence the value of node n, which is the maximum of the values of nodes o and p, must be less than or equal to 2. Furthermore, since the value of node m is the minimum of the value of node n and its brother, and node n has a value less than or equal to 2, the value of node m must also be less than or equal to 2. This causes

the brother of node n to be pruned, since the value of the root node a is greater than or equal to 4. Thus, we computed the minimax value of the root of the tree to be 4, by generating only seven of sixteen leaf nodes in this case.

Since alpha-beta pruning performs a minimax search while pruning much of the tree, its effect is to allow a deeper search with the same amount of computation. This raises the question of how much alpha-beta improves performance. The best way to characterize the efficiency of a pruning algorithm is in terms of its *effective branching factor.* The effective branching factor is the d^{th} root of the number of frontier nodes that must be evaluated in a search to depth d, in the limit of large d.

The efficiency of alpha-beta pruning depends upon the order in which nodes are encountered at the search frontier. For any set of frontier node values, there exists some ordering of the values such that alpha-beta will not perform any cutoffs at all. In that case, all frontier nodes must be evaluated and the effective branching factor is b, the brute-force branching factor.

On the other hand, there is an optimal or perfect ordering in which every possible cutoff is realized. In that case, the effective branching factor is reduced from b to $b^{1/2}$, the square root of the brute-force branching factor. Another way of viewing the perfect ordering case is that for the same amount of computation, one can search twice as deep with alpha-beta pruning as without. Since the search tree grows exponentially with depth, doubling the search horizon is a dramatic improvement.

In between worst-possible ordering and perfect ordering is random ordering, which is the average case. Under random ordering of the frontier nodes, alpha-beta pruning reduces the effective branching factor to approximately $b^{3/4}$ [35]. This means that one can search 4/3 as deep with alpha-beta, yielding a 33% improvement in search depth.

In practice, however, the effective branching factor of alpha-beta is closer to the best case of $b^{1/2}$ due to *node ordering.* The idea of node ordering is that instead of generating the tree left to right, we can reorder the tree based on static evaluations of the interior nodes. In other words, the children of MAX nodes are expanded in decreasing order of their static values, while the children of MIN nodes are expanded in increasing order of their static values.

Quiescence, Iterative-Deepening, and Transposition Tables

Two other important ideas are quiescence and iterative-deepening. The idea of quiescence is that the static evaluator should not be applied to positions whose values are unstable, such as those occurring in the middle of a piece trade. In those positions, a small secondary search is conducted until the static evaluation becomes more stable. In games such as chess or checkers, this can be achieved by always exploring any capture moves one level deeper.

Iterative-deepening is used to solve the problem of where to set the search horizon [47], and in fact predated its use as a memory-saving device in single-agent search. In a tournament game, there is a limited amount of time allowed for moves. Unfortunately, it is very difficult to accurately predict how long it will take to perform an alpha-beta search to a given depth. The solution is to perform a series of searches to successively greater depths. When time runs out, the move recommended by the last completed search is made.

The search graphs of most games, such as chess, contain multiple paths to the same node, often reached by making the same moves in a different order, referred to as a transposition of the moves. Since alpha-beta is a depth-first search, it is important to detect when a node has already been searched, in order to avoid researching it. A *transposition table* is a table of previously encountered game states, together with their backed-up minimax values. Whenever a new state is generated, if it is stored in the transposition table, its stored value is used instead of searching the tree below the node.

Almost all two-player game programs use full-width, fixed-depth, alpha-beta minimax search with node ordering, quiescence, iterative-deepening, and transposition tables, among other techniques.

Special-Purpose Hardware

While the basic algorithms are described above, much of the performance advances in computer chess have come from faster hardware. The faster the machine, the deeper it can search in the time available, and the better it plays. Despite the rapidly advancing speed of general-purpose computers, the best machines today are based on special-purpose hardware designed and built only to play chess. For example, DeepBlue is a chess machine that can evaluate about 200 million chess positions per second [33]. In May 1997, it defeated Gary Kasparov, the world champion, in a six-game tournament.

Multiplayer Games, Imperfect and Hidden Information

Minimax search with static evaluation and alpha-beta pruning is most appropriate for two-player games with perfect information, and alternating moves among the players. This paradigm extends in a straightforward way to more than two players, but alpha-beta becomes less effective [20]. Games with chance elements, such as the roll of the dice in backgammon for example, tend to foil search algorithms because of the need to search over all possible chance outcomes. In addition to chance, card games have information that is available to some players but hidden from others, such as the cards in the different hands in bridge. Perhaps poker is one of the ultimate challenges in this area, combining all of the above complexities as well as active deception and the need to model the opponents.

36.7 Constraint-Satisfaction Problems

In addition to single-agent path-finding problems and two-player games, the third major application of heuristic search is constraint-satisfaction problems. The Eight Queens Problem mentioned previously is a classic example. Other examples include graph coloring, Boolean satisfiability, and scheduling problems.

Constraint-satisfaction problems are modeled as follows: There is a set of variables, a set of values for each variable, and a set of constraints on the values that the variables can be assigned. A unary constraint on a variable specifies a subset of all possible values that can be assigned to that variable. A binary constraint between two variables specifies which possible combinations of assignments to the pair of variables satisfy the constraint. For example, in a map or graph coloring problem, the variables would represent regions or nodes, and the values would represent colors. The constraints are binary constraints on each pair of adjacent regions or nodes that prohibit them from being assigned the same color.

Chronological Backtracking

The brute-force approach to constraint satisfaction is called *chronological backtracking*. One selects an order for the variables, and an order for the values, and starts assigning values to the variables one at a time. Each assignment is made so that all constraints involving any of the variables that have already been assigned are satisfied. The reason for this is that once a constraint is violated, no assignment to the remaining variables can possibly resatisfy that constraint. Once a variable is reached which has no remaining legal assignments, then the last variable that was assigned is reassigned to its next legal value. The algorithm continues until either a complete, consistent assignment is found, resulting in success, or all possible assignments are shown to violate some constraint, resulting in failure. Figure 36.6 shows the tree generated by brute-force backtracking to find all solutions to the Four Queens problem. The tree is searched depth-first to minimize memory requirements.

Limited Discrepancy Search

Limited discrepancy search (LDS) [14, 25] is a completely general tree-search algorithm, but is most useful in the context of constraint-satisfaction problems in which the entire tree is too large to search exhaustively.

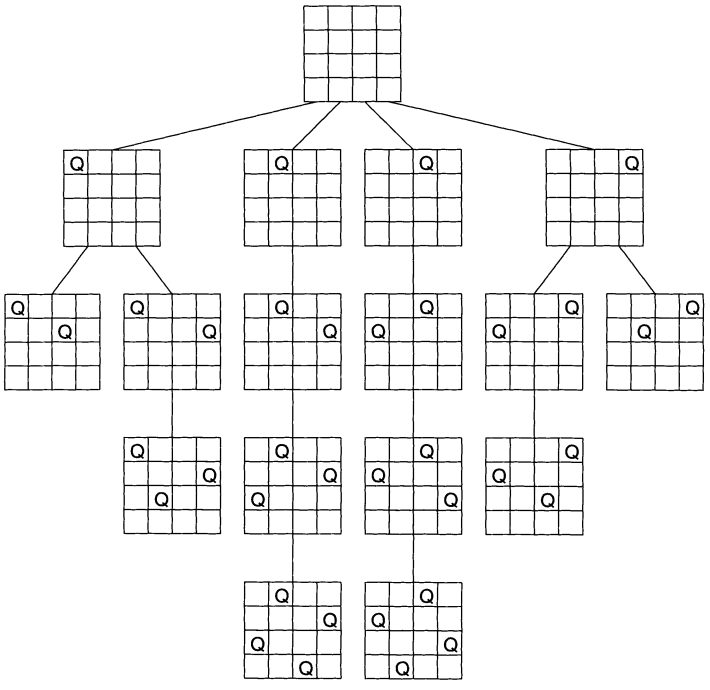


FIGURE 36.6 Tree generated to solve Four Queens Problem.

In that case, we would like to search that subset of the tree that is most likely to yield a solution in the time available. Assume that we can heuristically order a binary tree so that at any node, the left branch is more likely to lead to a solution than the right branch. LDS then proceeds in a series of depth-first iterations. The first iteration explores just the left-most path in the tree. The second iteration explores those root-to-leaf paths with exactly one right branch, or discrepancy, in them. In general, each iteration explores those paths with exactly k discrepancies, with k ranging from zero to the depth of the tree. The last iteration explores just the rightmost branch. Under certain assumptions, one can show that LDS is likely to find a solution sooner than a strict left-to-right depth-first search.

Intelligent Backtracking

One can improve the performance of brute-force backtracking using a number of techniques, such as variable ordering, value ordering, backjumping, and forward checking.

The order in which variables are instantiated can have a large effect on the size of the search tree. The idea of variable ordering is to order the variables from most constrained to least constrained [10, 42]. For example, if a variable has only a single value remaining that is consistent with the previously instantiated variables, it should be assigned that value immediately. In general, the variables should be instantiated in increasing order of the size of their remaining domains. This can either be done statically at the beginning of the search, or dynamically, reordering the remaining variables each time a variable is assigned a new value.

The order in which the values of a given variable are chosen determines the order in which the tree is searched. Since it doesn't effect the size of the tree, it makes no difference if all solutions are to be found. If only a single solution is required, however, value ordering can decrease the time required to find a solution. In general, one should order the values from least constraining to most constraining, in order to minimize the time required to find a first solution [5, 12].

An important idea, originally called backjumping, is that when an impasse is reached, instead of simply

undoing the last decision made, the decision that actually caused the failure should be modified [11]. For example, consider a three-variable problem where the variables are instantiated in the order x, y, z. Assume that values have been chosen for both x and y, but that all possible values for z conflict with the value chosen for x. In chronological backtracking, the value chosen for y would be changed, and then all the possible values for z would be tested again, to no avail. A better strategy in this case is to go back to the source of the failure, and change the value of x, before trying different values for y.

When a variable is assigned a value, the idea of forward checking is to check each remaining uninstantiated variable to make sure that there is at least one assignment for each of them that is consistent with the previous assignments. If not, the original variable is assigned its next value.

Constraint Recording

In a constraint-satisfaction problem, some constraints are explicitly specified, and others are implied by the explicit constraints. Implicit constraints may be discovered either during a backtracking search, or in advance in a preprocessing phase. The idea of constraint recording is that once these implicit constraints are discovered, they should be saved explicitly so that they don't have to be rediscovered.

A simple example of constraint recording in a preprocessing phase is called arc consistency [10, 29, 32]. For each pair of variables x and y that are related by a binary constraint, we remove from the domain of x any values that do not have at least one corresponding consistent assignment to y, and vice versa. In general, several iterations may be required to achieve complete arc consistency. Path consistency is a generalization of arc consistency where instead of considering pairs of variables, we examine triples of constrained variables. The effect of performing arc or path consistency before backtracking is that the resulting search space can be dramatically reduced. In some cases, this preprocessing of the constraints can eliminate the need for search entirely.

Heuristic Repair

Backtracking searches a space of consistent partial assignments to variables, in the sense that all constraints among instantiated variables are satisfied, looking for a complete consistent assignment to the variables, or in other words a solution. An alternative approach is to search a space of inconsistent but complete assignments to the variables, until a consistent complete assignment is found. This approach is known as heuristic repair [31]. For example, in the Eight Queens problem, this amounts to placing all eight queens on the board at the same time, and moving the queens one at a time until a solution is found. The natural heuristic, called min-conflicts, is to move a queen that is in conflict with the most other queens, and move it to a position where it conflicts with the fewest other queens.

What is surprising about this simple strategy is how well it performs, relative to backtracking. While backtracking techniques can solve on the order of hundred-queen problems, heuristic repair can solve million-queen problems, often with only about 50 individual queen moves! This strategy has been extensively explored in the context of Boolean satisfiability, where it is referred to as GSAT [45]. GSAT can satisfy difficult formulas with several thousand variables, whereas the best backtracking-based approach, the Davis–Putnam algorithm [2] with unit propagation, can only satisfy difficult formulas with several hundred variables.

The main drawback of this approach is that it is not complete, in that it is not guaranteed to find a solution in a finite amount of time, even if one exists. If there is no solution, these algorithms will run forever, whereas backtracking will eventually discover that a problem is not solvable.

While constraint-satisfaction problems appear somewhat different from single-agent path-finding problems and two-player games, the is a strong similarity among the algorithms employed. For example, backtracking can be viewed as a form of branch-and-bound, where a node is pruned when a constraint is violated. Similarly, heuristic repair can be viewed as a heuristic search where the evaluation function is the total number of constraints that are violated, and the goal is to find a state with zero constraint violations.

36.8 Research Issues and Summary

Research Issues

The primary research problem in this area is the development of faster algorithms. All the above algorithms are limited by efficiency either in the size of problems that they can solve optimally, or in the quality of the decisions they can make or solutions they can find within practical computational limits. Thus, there is a continual demand for faster algorithms.

A related research area is the development of space-efficient algorithms [23]. While the exponential-space algorithms are clearly impractical, the linear-space algorithms use very little of the memory available on current machines. The primary issue here is given a fixed amount of memory, how to make the best use of it to speed up a search as much as possible. In a two-player game search, the extra memory is used primarily in the transposition table. In single-agent path-finding problems, one of the most effective uses of additional memory is a form of bidirectional search known as perimeter search [8, 15, 30]. The idea is to search breadth-first backward from the goal state until memory is nearly exhausted. Then, the forward search proceeds until it encounters a state on the perimeter of the backward search. The main advantage to this approach is that the nodes on the perimeter can be used to refine the heuristic estimates in the forward search.

Another research area is the development of parallel search algorithms. Most search algorithms have a tremendous amount of potential parallelism, since the basic step of node generation and evaluation is often performed billions of times. As a result, many such algorithms are readily parallelized with nearly linear speedups. The algorithms that are difficult to parallelize are branch-and-bound algorithms, such as alpha-beta pruning, because the results of searching one part of the tree determine whether another part of the tree needs to be examined at all.

Since the performance of a search algorithm depends critically on the quality of the heuristic evaluation function, another important research area is the automatic generation of such functions. This was pioneered in the area of two-player games by Arthur Samuel's landmark checkers program that learned to improve its evaluation function through repeated play [44]. In the area of single-agent problems, a dominant theory is that the exact cost of a solution to a simplified version of a problem can be used as an admissible heuristic evaluation function for the original problem [36]. For example, in the sliding-tile puzzles, if we remove the constraint that a tile can only be slid into the blank position, then any tile can be moved to any adjacent position at any time. The optimal number of moves required to solve this simplified version of the problem is the Manhattan distance, which is an admissible heuristic for the original problem. Automating this approach, however, is still a research problem [41].

Another important research area is the development of selective search algorithms for two-player games. While Deep Blue defeated Gary Kasparov, it did it by evaluating 200 million chess positions per second. Obviously, humans are much more selective in their choices of what positions to examine. The development of alternatives to full-width, fixed-depth minimax search is an active area of research. See [24] for one example of a selective search algorithm, along with pointers to other work in this area.

Summary

We have described search algorithms for three different classes of problems. In the first, single-agent path-finding problems, the task is to find a sequence of operators that map an initial state to a desired goal state. Much of the work in this area has focused on finding optimal solutions to such problems, often making use of admissible heuristic functions to speed up the search without sacrificing optimality. In the second area, two-player games, finding optimal solutions is infeasible, and research has focused on algorithms for making the best move decisions possible given a limited amount of computing time. This approach has also been applied to single-agent problems as well. In the third class of problems, constraint-satisfaction problems, the task is to find a state that satisfies a set of constraints. While all three

of these types of problems are different, the same set of ideas, such as brute-force searches and heuristic evaluation functions, can be applied to all three.

36.9 Defining Terms

Admissible: A heuristic is said to be admissible if it never overestimates actual distance from a given state to a goal. An algorithm is said to be admissible if it always finds an optimal solution to a problem if one exists.

Branching factor: The average number of children of a node in a problem-space graph.

Constraint-satisfaction problem: A problem where the task is to identify a state that satisfies a set of constraints.

Depth: The length of a shortest path from the initial state to a goal state.

Heuristic evaluation function: A function from a state to a number. In a single-agent problem, it estimates the distance from the state to a goal. In a two-player game, it estimates the merit of the position with respect to one player.

Node expansion: Generating all the children of a given state.

Node generation: Creating the data structure that corresponds to a problem state.

Operator: An action that maps one state into another state, such as a twist of Rubik's Cube.

Problem instance: A problem space together with an initial state of the problem and a desired set of goal states.

Problem space: A theoretical construct in which a search takes place, consisting of a set of states and a set of operators.

Problem-space graph: A graphical representation of a problem space, where states are represented by nodes, and operators are represented by edges.

Search: A trial-and-error exploration of alternative solutions to a problem, often systematic.

Search tree: A problem-space graph with no cycles.

Single-agent path-finding problem: A problem where the task is to find a sequence of operators that map an initial state to a goal state.

State: A configuration of a problem, such as the arrangement of the parts of a Rubik's Cube at a given point in time.

References

[1] Bolc, L., and Cytowski, J., *Search Methods for Artificial Intelligence,* Academic Press, London, 1992.

[2] Davis, M., and Putnam, H., A computing procedure for quantification theory, *Journal of the Association for Computing Machinery,* 7, 201–215, 1960.

[3] Davis, H.W, Bramanti-Gregor, A., and Wang, J., The advantages of using depth and breadth components in heuristic search, in *Methodologies for Intelligent Systems 3,* Ras, Z.W. and Saitta, L., Eds., North-Holland, Amsterdam, 19–28, 1989.

[4] Dechter, R., and Pearl, J., Generalized best-first search strategies and the optimality of A*, *Journal of the Association for Computing Machinery,* 32(3), 505–536, Jul. 1985.

[5] Dechter, R., Pearl, J., 1988. Network-Based Heuristics for Constraint-Satisfaction Problems, *Artificial Intelligence,* 34(1), 1–38, 1987.

[6] Dijkstra, E.W., A note on two problems in connexion with graphs, *Numerische Mathematik,* 1, 269–271, 1959.

[7] Dillenburg, J.F., and Nelson, P.C., Improving the efficiency of depth-first search by cycle elimination, *Information Processing Letters,* 45(1), 5–10, 1993.
[8] Dillenburg, J.F., and Nelson, P.C., Perimeter search, *Artificial Intelligence,* 65(1), 165–178, Jan. 1994.
[9] Doran, J.E., and Michie, D., Experiments with the Graph Traverser program, *Proceedings of the Royal Society A,* 294, 235–259, 1966.
[10] Freuder, E.C., A sufficient condition for backtrack-free search. *J. Assoc. Comput. Mach.,* 29(1), 24–32, 1982.
[11] Gaschnig, J., *Performance measurement and analysis of certain search algorithms,* Ph.D. thesis. Department of Computer Science, Carnegie-Mellon Univ., Pittsburgh, PA, 1979.
[12] Haralick, R.M., and Elliott, G.L., Increasing tree search efficiency for constraint satisfaction problems, *Artificial Intelligence,* 14, 263–313, 1980.
[13] Hart, P.E., Nilsson, N.J., and Raphael, B., A formal basis for the heuristic determination of minimum cost paths, *IEEE Transactions on Systems Science and Cybernetics,* 4(2), 100–107, 1968.
[14] Harvey, W.D., and Ginsberg, M.L., Limited discrepancy search, *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI-95),* Montreal, Canada, 607–613, Aug. 1995.
[15] Kaindl, H., Kainz, G., Leeb, A., and Smetana, H., How to use limited memory in heuristic search, *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence (IJCAI-95),* Montreal, Canada, 236–242, Aug. 1995.
[16] Kanal, L. and Kumar, V., Eds., *Search in Artificial Intelligence,* Springer-Verlag, New York, 1988.
[17] Korf, R.E., Depth-first iterative-deepening: An optimal admissible tree search, *Artificial Intelligence,* 27(1), 97–109, 1985.
[18] Korf, R.E., Search in AI: A survey of recent results, in *Exploring Artificial Intelligence,* Shrobe, H.E., Ed., Morgan-Kaufmann, Los Altos, CA, 1988.
[19] Korf, R.E., Real-time heuristic search, *Artificial Intelligence,* 42(2-3), 189–211, Mar. 1990.
[20] Korf, R.E., Multi-player alpha-beta pruning, *Artificial Intelligence,* 48(1), 99–111, Feb. 1991.
[21] Korf, R.E., Search, in the *Encyclopedia of Artificial Intelligence,* 2nd ed., John Wiley, New York, 1460–1467, 1992.
[22] Korf, R.E., Linear-space best-first search, *Artificial Intelligence,* 62(1), 41–78, Jul. 1993.
[23] Korf, R.E., Space-efficient search algorithms, *Computing Surveys,* 27(3), 337–339, Sept. 1995.
[24] Korf, R.E., and Chickering, D.M., Best-first minimax search, *Artificial Intelligence,* 84(1–2), 299–337, July 1996.
[25] Korf, R.E., Improved limited discrepancy search, *Proceedings of the Thirteenth National Conference on Artificial Intelligence (AAAI-96),* Portland, OR, Aug. 1996, 286–291.
[26] Korf, R.E. and Taylor, L.A., Finding optimal solutions to the twenty-four puzzle, *Proceedings of the Thirteenth National Conference on Artificial Intelligence (AAAI-96),* Portland, OR, 1202–1207, Aug. 1996.
[27] Korf, R.E., Finding optimal solutions to Rubik's Cube using pattern databases, *Proceedings of the Fourteenth National Conference on Artificial Intelligence (AAAI-97),* Providence, RI, 700–705, Jul. 1997.
[28] Knuth, D.E., and Moore, R.E., An analysis of alpha-beta pruning, *Artificial Intelligence,* 6(4), 293–326, 1975.
[29] Mackworth, A.K., Consistency in networks of relations. *Artificial Intelligence* 8(1), 99–118, 1977.
[30] Manzini, G., BIDA*: An improved perimeter search algorithm, *Artificial Intelligence,* 75(2), 347–360, June 1995.
[31] Minton, S., Johnston, M.D., Philips, A.B., and Laird, P., Minimizing conflicts: A heuristic repair method for constraint satisfaction and scheduling problems, *Artificial Intelligence,* 58(1-3), 161–205, Dec. 1992.

[32] Montanari, U., Networks of constraints: Fundamental properties and applications to picture processing, *Information Science,* 7, 95–132, 1974.
[33] Newborn, M., *Kasparov vs. Deep Blue: Computer Chess Comes of Age,* Springer-Verlag, 1996.
[34] Newell, A., and Simon, H.A., *Human Problem Solving,* Prentice-Hall, Englewood Cliffs, NJ, 1972.
[35] Pearl, J., The solution for the branching factor of the Alpha-Beta pruning algorithm and its optimality, *Communications of the Association of Computing Machinery,* 25(8), 559–564, 1982.
[36] Pearl, J., *Heuristics,* Addison-Wesley, Reading, MA, 1984.
[37] Pearl, J., and Korf, R.E., Search techniques, in *Annual Review of Computer Science,* Vol. 2, Annual Reviews, Palo Alto, CA, 1987, 451–467.
[38] Pohl, I., First results on the effect of error in heuristic search, in *Machine Intelligence 5,* Meltzer, B. and Michie, D., Eds., American Elsevier, New York, 219–236, 1970.
[39] Pohl, I., Heuristic search viewed as path finding in a graph, *Artificial Intelligence,* 1, 193–204, 1970.
[40] Pohl, I., Bi-directional search, in *Machine Intelligence 6,* Meltzer, B. and Michie, D., Eeds., American Elsevier, New York, 127–140, 1971.
[41] Prieditis, A.E., Machine discovery of effective admissible heuristics, *Machine Learning,* 12, 117–141, 1993.
[42] Purdom, P.W., Search rearrangement backtracking and polynomial average time. *Artificial Intelligence,* 21(1,2), 117–133, 1983.
[43] Ratner, D. and Warmuth, M., Finding a shortest solution for the NxN extension of the 15-Puzzle is intractable, *Proceedings of the Fifth National Conference on Artificial Intelligence (AAAI-86),* Philadelphia, PA, 1986, 168–172.
[44] Samuel, A.L., Some studies in machine learning using the game of checkers, in *Computers and Thought,* Feigenbaum, E. and Feldman, J., Eds., McGraw-Hill, New York, 71–105, 1963.
[45] Selman, B., Levesque, H., and Mitchell, D., A new method for solving hard satisfiability problems, *Proceedings of the Tenth National Conference on Artificial Intelligence (AAAI-92),* San Jose, CA, 440–446, Jul. 1992.
[46] Shannon, C.E., Programming a computer for playing chess, *Philosophical Magazine,* 41, 256–275, 1950.
[47] Slate, D.J., and Atkin, L.R., CHESS 4.5 - The Northwestern University chess program, in *Chess Skill in Man and Machine,* Frey, P.W., Ed., Springer-Verlag, New York, 82–118, 1977.
[48] Stickel, M.E., and Tyson, W.M., An analysis of consecutively bounded depth-first search with applications in automated deduction, in *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI-85),* Los Angeles, CA, Aug. 1985, 1073–1075.
[49] Taylor, L., and Korf, R.E., Pruning duplicate nodes in depth-first search, *Proceedings of the National Conference on Artificial Intelligence (AAAI-93),* Washington D.C., 756–761, Jul. 1993.
[50] Turing, A.M., Computing machinery and intelligence, *Mind,* 59, 433–460, Oct. 1950. Also in *Computers and Thought,* Feigenbaum, E. and Feldman, J., Eds., McGraw-Hill, New York, 1963.

Further Information

The classic reference in this area is [36]. More recent survey articles include [18, 37], and [21]. Much of the material in this article was derived from these sources. A number of papers have been collected in an edited volume devoted to search [16]. The most recent book-length treatment of this area is [1]. Most new research in this area initially appears in the Proceedings of the National Conference on Artificial Intelligence (AAAI) or the Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI). Prominent journals in this area include *Artificial Intelligence* (AIJ), the *Journal of Artificial Intelligence Research* (JAIR), and the *IEEE Transactions on Pattern Analysis and Machine Intelligence* (IEEE TPAMI).

37

Simulated Annealing Techniques[1]

Albert Y. Zomaya
The University of Western Australia

Rick Kazman
Carnegie Mellon University

37.1 Introduction

This chapter will present the essential components of the *simulated annealing (SA)* algorithm and review its origins and potential for solving a wide range of optimization problems, in a manner that is accessible to the widest possible audience. Some historical perspective and description of recent research results will also be provided. During the course of this review bibliographical references will be provided to guide the interested reader to sources that contain additional theoretical results and complete details of individual applications.

Many problems in a variety of disciplines can be formulated as optimization problems; and most of these can be solved by adopting one of two "popular" approaches: *divide-and-conquer* or *hill-climbing* techniques (other approaches can be adopted, see for example, [31], and see also Chapters 31–33 in this volume). In the first approach, the solution is problem-dependent, and typically detailed information about the problem is required in order to develop a solution strategy. Also, not many problems can be subdivided into smaller parts that can be solved separately and then recombined. In the second approach, most hill-climbing algorithms are based on gradient descent methods. These methods suffer from a major

[1] Albert Y. Zomaya — Acknowledges the support of the Australian Research Council grant no. 04/15/412/194.

Parts of this chapter were written while the first author was a visiting professor with the Department of Computer Science, University of Waterloo, Waterloo, Ontario, Canada.
Parallel Computing Laboratory, Department of Electrical & Electronic Engineering, The University of Western Australia, Perth, Western Australia 6907, zomaya@ee.uwa.edu.au.
Rick Kazman — Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA 15213-3890, USA, kazman@sei.cmu.edu.

drawback of getting trapped in a local minimum. That is, the algorithm may get "tarpped" in a valley, from which all paths lead to locally worse solutions, and will never get to an optimal solution that lies outside the valley. The SA algorithm avoids local minima by introducing an element of randomness into the search process [36].

Over the last few years the SA algorithm and its many extensions and refinements have been extensively employed to solve a wide range of application domains, especially in combinatorial optimization problems [2, 5, 8, 13, 15, 47, 54, 57, 58]. An important characteristic of the SA algorithm is that it does not require specialist knowledge about how to solve a particular problem. This makes the algorithm generic in the sense that it can be used in a variety of optimization problems without the need to change the basic structure of the computations.

The versatility of SA has attracted many researchers over a number years, and has recently spawned a number of variations to the original algorithm, including parallel versions to speed up the rate of computations [18, 19, 25, 42, 51, 62].

37.2 The Basic Idea

The SA is a stochastic optimization method modelled on the behavior of condensed matter at low temperatures. It borrows techniques from *statistical mechanics* to find global optima of systems with large numbers of degrees of freedom. The method is analogous to the way that liquids freeze and crystallize or metals cool and anneal. The core of the process is slow cooling, allowing enough time for the redistribution of atoms as they lose mobility until a minimum energy state is reached [57].

During the physical annealing process a solid is placed in a heat bath and the temperature is continually raised until the solid has melted and the particles of the solid are physically disarranged or positioned in random order. The orientation of the particles are referred to as the *spins.* From such a high energy level, the heat bath is cooled slowly by lowering the temperature to allow the particles to align themselves in a regular crystalline lattice structure. This final structure corresponds to a stable low energy state.

The temperature must be lowered slowly to allow the solid to reach equilibrium after each temperature drop. Otherwise, irregular alignments may occur, resulting in defects that get frozen into the solid. Of course, this can result in a metastable (highly unstable) structure rather than the required stable low energy structure.

The idea of annealing was combined with the well-known *Monte Carlo* algorithm [40], which was originally used to perform numerical averaging over large systems in statistical mechanics. The SA algorithm maintains the speed and reliability of gradient descent algorithms while at the same time avoiding local minima [36].

37.3 Global Optimization Problems

There are two classes of algorithms that can be used to solve *global optimization* problems [48]. The first are random or Monte Carlo type of algorithms that make use of pseudo random variables. The second are deterministic algorithms that do not take advantage of randomization; the earliest global optimization methods belong to this class. Global optimization techniques aim at solving the following general class of problems [31]:

$$\begin{aligned} &\text{maximize} && f(x) \\ &\text{subject to:} && h_i(x) \leq 0 \quad i = 1, 2, ..., n \\ & && x \in X \end{aligned} \tag{37.1}$$

where f and h_i (for $i = 1, 2, \ldots, n$) are real-valued functions defined on a domain containing $X \subseteq \Re^m$, where m is the number of variables. If $n = 0$ and $X = [a_1, b_1] \times [a_2, b_2] \times \ldots \times [a_m, b_m]$ defines a hyperrectangle of $\Re^m$, the problem is called *unconstrained,* and if $m = 1$, the problem is *univariate.*

An assumption that is usually made to enable the solution of (37.1) is that the functions f and $h_i (i = 1, 2, ..., n)$ can be evaluated at all points of X. This means that there are methods by which given the values of variables, the corresponding evaluations of functions can be provided. This is true in a number of practical situations in which measurements of functions can be made through empirical means, but no analytical (or closed-form) expressions are available [41, 43, 46].

If no other assumptions are made on f and $h_i (i = 1, 2, ..., n)$, the problem is intractable. It is important to note that no matter how many function evaluations are performed, there is no guarantee that the minimum can be obtained [29]. To help in finding a solution, another simple assumption that is usually made is that the slopes of the functions f and $h_i (i = 1, 2, \ldots, n)$ are bounded. In such case, these function are said to be *Lipschitz.* A real-valued function defined on a compact set $X \subseteq \mathcal{R}^m$ said to be Lipschitz if it satisfies the condition

$$\forall x \in X \quad \forall y \in X \quad | \alpha(x) - \alpha(y) | \leq L \parallel x - y \parallel \tag{37.2}$$

where L is a constant (Lipschitz constant) and $\parallel \bullet \parallel$ is the Euclidean norm (other norms can also be used). In general, deterministic algorithms evaluate a given cost function at points on a grid. A major problem with such deterministic algorithms is that they require knowledge about the problem, or in other words, the cost function (such as the Lipchitz constant L) that needs to be evaluated.

Most global optimization algorithms are of the random type and are related to the so-called *multistart* algorithm [7]. In this case, a local optimization algorithm is executed from different initial or starting points that are chosen at random, usually from a uniform distribution on the domain of the cost function. However, a multistart algorithm is still inefficient because it will inevitably find each local extremum (maximum or minimum) several times. In addition, local search are the most time consuming part of any procedure (the multistart algorithm in this case). A typical multistart algorithm that can take advantage of a local search procedure Ω is shown below.

PROCEDURE MULTISTART
1 k $\leftarrow$ 1
2 $\prod(0) = -\infty$
3 **Generate** a point x from the uniform
distribution over X
4 **Apply** Ω to x to get $\tilde{x}$
5 **If** $f(\tilde{x}) > \prod(k-1)$ **then**
6 $\prod(k) \leftarrow f(\tilde{x})$
7 $x_k = \tilde{x}$
8 **Else**
9 $\prod(k) \leftarrow \prod(k-1)$
10 $x_k = x_{k-1}$
11 **Increment** k; **Return** to 3

From a computational point of view, a local search procedure should not be called more than once in every region of attraction. In this case, the region of attraction of the local maximum $\bar{x}_k$ is defined as the set of points in X from which the local search procedure (Ω) will converge to $\bar{x}_k$.

The SA algorithm can also be classified as a random search technique, but the algorithm avoids getting trapped in local minima by accepting, in addition to movements corresponding to improvement in cost function value, also movements corresponding to a deterioration in cost function value. However, the deterioration movements are accepted with a finite probability. These two different types of movements allow the algorithm to move away from local minima and traverse more states in the region X (Fig. 37.1).

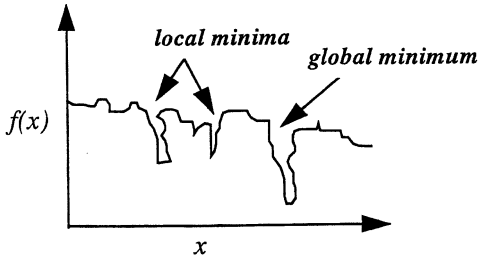


FIGURE 37.1 Local and global minima.

The Metropolis Algorithm

The SA algorithm is based on a procedure introduced by [40] to simulate the equilibrium states of a multibody system at a given finite temperature. The Metropolis procedure consisted of a number of simple principles. At each step, a small perturbation of the configuration (of the given system) is chosen at random and the resulting change in the energy of the system, Δ, is calculated. The new configuration is accepted with probability 1 if $\Delta \leq 0$, and with probability $e^{-\Delta/k_BT}$ if $\Delta > 0$. To temperature (T) can be viewed as a control parameter. The procedure is given below.

PROCEDURE METROPOLIS
1 **Generate** some random initial configuration s
2 **Repeat**
3 $s' \leftarrow$ Some random neighboring configuration of s
4 $\Delta \leftarrow E(s') - E(s)$
5 $\rho_T(\Delta) \leftarrow \min(1, e^{-\Delta/k_BT})$
6 **If** $random(0.1) \leq \rho_T(\Delta)$ **then** $s \leftarrow s'$
7 **Until false**
8 **End**

In the above algorithm, $E(s)$ stands for the energy associated with state s and k_B is the Boltzmann constant ($k_B = 1.38 \times 10^{-16} ergs/K(Kelvin)$). The energy function $E(\bullet)$ is usually replaced by a cost function; then the above procedure can be used to simulate the behavior of the optimization problem with the given cost function.

The essence of the Metropolis procedure can be summarized as follows. If Φ_s is the set of states $\varphi \in S$ reachable in exactly one move (perturbation) from s. Each move must be *reversible* ($\varphi \in \Phi_s \Rightarrow s \in \Phi_\varphi$). The number of possible moves (i.e., $\omega = |\Phi_s|$) must be the same from any state s. It also must be possible to reach any state in Φ from any other in a finite number of moves.

The function $\rho_T(\Delta) = \min(1, e^{-\Delta/k_BT})$ is used to choose a random move, and each move has a probability of $1/\omega$ of being accepted. Following the sequence of accept and reject moves given in the above algorithm, leads to a Markov process with transition function [7, 12]:

$$\Phi_T(s'|s) = \begin{cases} \dfrac{1}{\omega}\rho_T(E(s') - E(s)) & s' \in \Phi_s \\ 1 - \displaystyle\sum_{\varphi \in \Phi_s} \frac{1}{\omega}\rho_T(E(\varphi) - E(s)) & s' = s \\ 0 & \text{otherwise} \end{cases}$$

If T is positive, the above process is *irreducible,* which means that for any two states $s, \varphi \in \Phi$, there is a nonzero probability that state φ will be reached from s. Defining π_T as the stationary (equilibrium)

distribution of this process, it can be found that

$$\pi_T(s) = \frac{e^{(-E(s)/T)}}{\sum_{\varphi \in \Phi} e^{(-E(\varphi)/T)}} \quad \forall s \in \Phi$$

The above follows from the principle of detailed reversibility [7, 12],

$$\pi_T(s)\Omega_T(s'|s) = \pi_T(s')\Omega_T(s|s') \quad \forall s, s' \in \Phi$$

where π_T is the Boltzman distribution. Simply stated, detailed reversibility means that the likelihood of any transition equals that of the opposite transition (i.e., in the opposite direction).

37.4 Simulated Annealing

As mentioned earlier, the SA algorithm is an optimization technique based on the behavior of condensed matter at low temperatures. The procedure employs methods that originated from statistical mechanics to find global minima of systems with very large degrees of freedom. The correspondence between combinatorial optimization problems and the way natural systems search for the ground state (lowest energy state) was first realized by [10, 36]. Based on the analogy given in Fig. 37.2, Kirkpatrick et al. [36] applied Monte Carlo methods, which are usually found in statistical mechanics, to the solution of global optimization problems, and it was shown that better solutions can be obtained by simulating the annealing process that takes place in natural systems.

Further, Kirkpatrick et al. [36] generalized the basic concept that was introduced in [40] by using a *multi-temperature* approach in which the temperature is lowered slowly in stages. At each stage the system is simulated by the Metropolis procedure until the system reaches equilibrium.

At the outset, the system starts with a high T, then a cooling (or annealing) scheme is applied by slowly decreasing T according to some given procedure. At each T a series of random new states are generated. States that improve the cost function are accepted. Now, rather than always rejecting states that do not improve the cost function, these states can be accepted with some finite probability depending on the amount of increase and T. This process randomizes the iterative improvement phase and also allows occasional uphill moves (i.e., moves that do not improve the solution) in an attempt to reduce the probability of falling into a local minimum.

As T decreases, configurations that increase the cost function are more likely to be rejected and the process eventually terminates with a *configuration* (solution) that has the lowest cost. This whole procedure has been proved to lead to a solution that is arbitrary close to the global minimum [1, 38, 57].

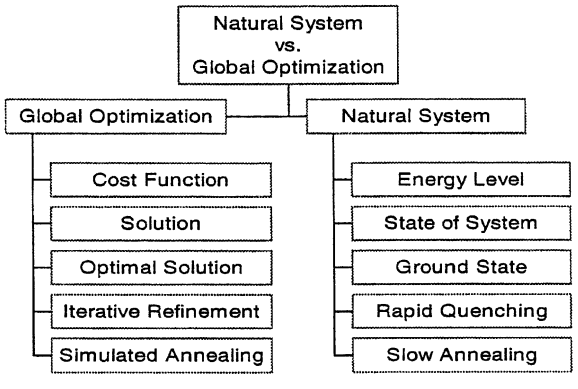


FIGURE 37.2 A one-to-one analogy between a natural process and global optimization.

In addition to having a well-formulated cost function, the design of an efficient annealing algorithm requires three other ingredients [1, 43]:

- The problem to be optimized must be represented by a set of data structures that provide a concise description of the nature of that problem. These data structures must allow for the fast generation and assessment of the different iterations as well as the efficient computation of the overall cost function.
- A large number of random rearrangements (i.e., perturbations, moves, or iterations) must be generated to adequately explore the search space.
- An annealing schedule should be devised to control the temperature during the annealing process. This procedure should specify the initial and final temperatures and the rate at which the temperature is lowered.

A general description of the SA algorithm is given below. After the cost function E is selected the algorithm can be described as follows:

PROCEDURE SIMULATED ANNEALING
Set $S \leftarrow S_0$ (random initial state)
Set $T \leftarrow T_0$ (initial temperature)
While (stopping criterion is not satisfied) **do**
  While (required number of states is not generated) **do**
    Generate a new state (S') by perturbing S.
    Evaluate E
    Compute $\Delta E = E(S') - E(S)$
    If ($\Delta E \leq 0$) **then**
      $S \leftarrow S'$
      Else
        Generate a random variable α, $0 \leq \alpha \leq 1$
        If $\alpha \leq e^{(-\Delta E)/T}$ **then** $S \leftarrow S'$
    End
  End
  Update T (decrement)
End

In the above procedure the Boltzmann constant (k_B) that appears in the Metropolis procedure is combined with the T, and the whole term is called temperature.

From a theoretical standpoint, the SA algorithm works in the following way. Assume a *random walk* on S that converges to a uniform distribution on S. Also denote the transition probability distribution by $\Lambda(s, \bullet)$ (the Markov chain is in state $s \in S$). In every iteration, given state s_i , a new state s' is generated from $\Lambda(s_i, \bullet)$. Then this new generated state is accepted with probability

$$\min\left\{1, e^{(f(s_{i+1})-f(s_i))/T}\right\}$$

which is the Metropolis criterion. In other words, with this probability we can set $s_{i+1} = s'$, otherwise, $s_{i+1} = s_i$. Now, if the Markov chain given by Λ is filtered using the procedure, the sequence of states generated will converge to the Boltzman distribution π_T (where T is the temperature). Therefore, a sequence of states can be generated $\{S_i(T)\}_{i=0}^{\infty}$ with the property that for every $\varepsilon > 0$,

$$\lim_{i\to\infty} \Pr\left(S_i(T) \in S_\varepsilon\right) = \pi_T\left(S_\varepsilon\right) \tag{37.3}$$

where S_ε is any level set ($S_\varepsilon \in S$). It is agreed upon that any *adaptive search* algorithm (e.g., SA) is based on the same property of the family of Boltzman distributions, which can be stated as [20, 27]:

$$\lim_{T \to 0} \pi_T (S_\varepsilon) = 1 \quad \forall \varepsilon > 0 \tag{37.4}$$

By using Eqs. (37.3) and (37.4), it can be concluded that

$$\lim_{T \to 0} \lim_{i \to \infty} \Pr (S_i(T) \in S_\varepsilon) = 1 \quad \forall \varepsilon > 0 \tag{37.5}$$

Basically, Eq. (37.5) governs the behavior of the SA algorithm. A number of states are generated $S_0, S_1, \ldots$ using the filtered random walk described above, except that now the temperature (T) will be decreased to zero as we iterate, according to some annealing schedule.

It can be noticed that the SA algorithm is characterized by its simple and elegant structure. However, a number of factors need to be considered to have an efficient implementation of the algorithm. These factors are, for example, the choice of cost function, the annealing schedule, and the algorithm termination condition.

Cost Function

The cost function is an application-dependent factor that measures the value of one solution relative to another [1, 57]. In some applications the cost function can be of an analytical nature, which means that the structure of the function can be determined *a priori* [46]. In other cases, the cost function is nonanalytical and need to be determined in some indirect manner from observing the process that is being optimized [43].

Nevertheless, the value of the difference in the cost function (i.e., ΔE) is crucial for the success of the iterative process. The value of $e^{-\Delta E/T}$ suggests that for a state to be accepted with, say, probability of 0.85 at the initial T, the T must be at least six times higher than ΔE (i.e., $e^{-1/6} = 0.846$). As the ratio of T and ΔE decreases, so does the probability of accepting that state, which means that when we get to ratios of 1:7, 1:8 or greater, there is extremely low probability of accepting poor solutions [47].

Annealing Schedule

The annealing schedule determines the process by which T should be decreased, which influences the performance of the overall algorithm to a great extent. In this case, two issues need to be considered: the first is how T should be decremented over time, and the second is how many iterations should be computed at any given T.

In some complex problems, the annealing schedule needs to be designed specifically to suit the application [21, 32, 59]. A worst-case scheduling scheme that is quite slow was developed to guarantee the convergence of the SA algorithm [21]. Other research showed that the SA algorithm can converge faster than the worst-case situation [30]. Also, an adaptive cooling schedule was developed that is based on the characteristics of the cost distribution and the annealing curve [32].

Staying at the same T for a long period of time will guarantee finding the best solution since the SA algorithm is *asymptotically* optimal. This means that the longer the algorithm runs the better is the quality of the solution obtained [49]. However, this is not acceptable from a practical point of view.

The number of iterations that need to be computed at any given T can be determined in two different ways [47, 55]. The number of iterations for any T can be given by

$$Y(T) = e^{(E_{\max} - E_{\min})/T} \tag{37.6}$$

where $E_{\max}$ and $E_{\min}$ are the highest and lowest values of the cost function obtained so far for the current T [32]. However, $Y(T)$ gets too large when T decreases, thus, requiring the introduction of an

arbitrary upper bound. Another method, which is based on experimental data, allows a certain number of acceptances or rejections at any given T before allowing T to decrease. A ratio of 1:10 of accepts and rejects, respectively, was proposed in [17].

The next step is to decrease T. It is recommended that T is decreased by a factor which is less than one. In this case, the rate of decrease of T becomes exponential. The advantage of this kind of rate is that it becomes slow when the search process is near completion, which gives the system a chance to find the global minimum. Also, if one uses the formula $T_i = a^i T_{i-1}$, where $a < 1$ (ideally set between 0.5–0.99) this will have the same effect. The a in this case will control the rate of annealing.

Another important factor in designing the annealing schedule is the choice of the initial value of T. A high initial T is usually selected to ensure that most of the moves attempted are accepted because they lead to a lower cost. This allows a wider search of the solution space at the beginning. A method suggested in [36] to find the initial T can be outlined as follows:

I. Begin with a random T.

II. Try a number of iterations at T, while keeping track of the percentage of the accepted moves (A) and rejected (R) ones.

III. If $A/(A + R) < 0.8$, then update T to $2T$ and then goto (II) and repeat the process until the system is "warm" enough.

However, one problem with this method is that the T might become too warm, which will consume much more time than what might be necessary.

Algorithm Termination

A number of schemes were proposed for estimating the termination (or freezing) temperature [47]. It was found that the value of T at which no further improvements can be made is [60]:

$$T_f = \frac{E_{m'} - E_m}{\ln \nu} \tag{37.7}$$

where E_m is the absolute minimum value of the cost function, which could be set to some predetermined value that depends on the application. $E_{m'}$ is the next largest value of the cost function (i.e., compared to E_m), and ν is the number of moves that takes to get from $E_{m'}$ to E_m. The freezing temperature determined by using Eq. (37.7) represents the worst-case scenario, because by using T_f the annealing process will not stop too soon, that is it will not stop before the minimum has been found.

Another approach which is employed by a commercial SA package named TimberWolf^R [53] stops the annealing process when no new solution have been accepted after four consecutive decreases in T. However, it has been shown that this approach might stop the annealing process prematurely [47, 55].

37.5 Convergence Conditions

The SA algorithm has a formal proof of convergence which depends on the annealing schedule. As seen from previous discussion, by manipulating the annealing schedule one can control the behavior of the algorithm. For any given T, a sufficient number of iterations always leads to equilibrium, at which point the temporal distribution of accepted states is stationary (the stationary distribution is Boltzmann). We also need to note, that at high T, almost any change is accepted. This means that the algorithm tries to span a very large neighborhood of the current state. At lower T, transitions to higher energy states becomes less frequent and the solution stabilizes.

The convergence of the SA algorithm to a global optimum can be proven by using Markov chain theory [12, 33, 38]. The sequence of perturbations (or moves) which are accepted by the algorithm form a Markov chain because the next state depends only on the current state and it is not influenced by past

states—no record is kept for the past states. So, given that the current state is j, the probability that the next state is k is the product of two probabilities: the probability that state k is generated by one move from state j and the probability of accepting state k.

A Markov chain is irreducible if for every pair of states (j, k), there is a sequence of moves which allows k to be reached from j. If the Markov chain ensures irreducibility, then the sequence of states accepted at a given T forms a Markov chain with stationary distribution [1, 49, 57]. If T takes on a sequence of values approaching zero, where there are a sufficient number of iterations at each value, then the probability of being at a global optimum at the termination of the execution of the algorithm is one.

Over the last few years a number of studies dealt with the convergence problem since it a major issue in optimization problem [1, 20, 21, 27, 28, 38, 57]. Actually, if one is to compare the SA algorithm with other stochastic methods such as *genetic algorithms* and *neural networks* one finds in the SA literature more solid studies as far as issues of convergence are concerned [39, 41, 45]. Also, it has been shown that even if several iterations of the SA algorithm result in deteriorations of the cost function, that the algorithm will eventually recover and move towards the global maximum (or its neighborhoods) [6].

However, there are a number of underlying principles that govern the convergence of the SA algorithm [23, 24, 38]:

- The existence of a unique asymptotic probability distribution (stationary distribution) for the stationary Markov chain corresponding to each strictly positive value of an algorithm control parameter (i.e., T).
- The existence of stationary distribution limits as $T \to 0$.
- The desired behavior of the stationary distribution limit (probability distribution with probability one) [see Eq. (37.5)].
- Sufficient conditions on the annealing schedule to ensure that the nonstationary algorithm asymptotically achieves the limiting distribution.

The work in [21] proposes a version of the SA algorithm which is called the *Gibbs sampler.* It is shown that for temperature schedules of the form

$$T_n = \frac{c}{\log n} \quad (n\text{:large})$$

that if c is sufficiently large, then convergence will be guaranteed. Along the same lines, Gidas [23] considers the convergence of the SA algorithm and similar algorithms that are based on Markov chain sampling methods that are related to the Metropolis algorithm.

For more theoretical details on convergence of the SA algorithm, the reader is referred to [1, 20, 21, 23, 27, 28, 38, 57] to name a few.

Before we conclude this section, it is important to note that the SA algorithm is closely related to another algorithm that has been used for global optimization and generated a lot of interest, which is called the *Langevin algorithm* [22]. This algorithm is based on the Langevin stochastic differential equation (proposed by Langevin in 1908 to describe the motion of a particle in a viscous fluid) given by

$$dx(t) = -\nabla g(x(t))dt + \gamma(t)dw(t) \tag{37.8}$$

where ∇g is the gradient of g and $w(t)$ is a standard r-dimensional *Wiener process* [44]. When $\gamma(t) \equiv \gamma_0(\gamma(t)$ is constant), then the probability density function of the solution process $x^{\gamma_0}(t)$ of Eq. (37.8) approaches

$$e^{(2g(x)/\gamma_0^2)} \tag{37.9}$$

as $t \to \infty$. Equation (37.8) could also have a normalization constant. This distribution is exactly the Boltzmann distribution at temperature $\gamma_0^2/2$. This observation has generated many interesting studies [11, 22].

However, only a few practical problems have been proposed along with some promising results. In [3], the authors use a modified Langevin algorithm which employs an interactive temperature schedule. They ran their tests on $g(\cdot)$ defined on $\Re^b$ where $b = 1, 2, \ldots, 14$. Other numerical results were reported by [23] that used a $g(\cdot)$ defined on $\Re$ with 400 local minima. Further, it was suggested that the Langevin algorithm be used with other multistart methods [31].

The comparison of different optimization algorithms is a very difficult task [20]. Some analytical results have been proposed to assist in such comparisons [52]. Also, a standard set of test functions have been developed in [16] to compare different optimization techniques. However, the applicability of these tests to the Langevin algorithm is rather questionable. This is due to the fact that such tests are more suited to compare algorithms in low dimensional spaces, but not the case of the Langevin algorithm which supposed to be used for functions in large dimensions. The structure and characteristics of such functions cannot be determined *a priori.*

37.6 Parallel Simulated Annealing Algorithms

One of the main drawbacks of the SA algorithm is the amount of time it takes to converge to a solution. As seen earlier, a number of approaches have been introduced to improve the computational efficiency of the SA algorithm [8, 21, 25, 32, 35, 53]. However, most of these techniques used heuristics to simplify the search process by either limiting the cost function or educe the chances of generating future moves that are going to be rejected.

A more effective way to improve the speed of computations is to run SA algorithms by using parallel processor platforms. For a parallel SA algorithm to have the same desirable convergence properties as the sequential algorithm, one must either maintain the serial decision sequence, or employ a different decision sequence such that the generated Markov chains have the same probability distribution as the sequential algorithm [42, 43].

Since, during the annealing process, a new state is generated by perturbing the previous state, it is natural to think that the SA algorithm is inherently sequential and cannot be parallelized. However, a number of ways have been used to remove this dependence between subsequent moves [4, 5, 37, 39, 42, 51, 62]. A number of other issues need to be considered when parallelizing the SA algorithm, such as the division of the search space among processors, and most importantly the amount of speedup obtained with the parallel algorithm.

With the large multitude of parallel SA algorithms proposed in the literature over the last few years, one could observe that most of these algorithms don't have the same convergence characteristics as the sequential algorithm, which to some extent compromises the range of their applicability. Successful parallel algorithms tend to be application-dependent [9, 14, 63].

Two problem-independent algorithms were proposed in [37]. These two algorithms were called move-decomposition and parallel-moves. The former divides the move-evaluate-decide task into several subtasks, and maps these subtasks onto several processors, while the latter generates many moves in parallel but only chooses a serializable subset of these moves to be performed. This subset consists of a number of moves which, when applied to the current state in any order, always produces the same final state. In addition, both methods were used in a hybrid scheme that consists of applying move-decompose at higher temperatures and parallel-moves at lower temperatures. The hybrid algorithm was mapped onto parallel processor system. Overall, limited levels of parallelism were achieved by this work: the speedup approached saturation with four processors.

Another method was developed in [51]. In this work, the authors developed a parallel scheme in which the states have the same probability distribution as the sequential algorithm. The algorithm applies two modes of operations depending on the value of the moves' acceptance rate $\lambda(T)$ which is more formally defined as the ratios of accepted moves to the number of attempted moves for a given T. For m processors, the algorithm has the following two modes:

- High temperature mode: if $\lambda(T) \geq \frac{1}{m}$ each processor evaluates only one move and one of the accepted moves is chosen randomly. The processors' memories are updated with the new solution and the next step takes place. For the Markov chain to have the same acceptance rate as its sequential counterpart, the number of moves attempted is computed as follows: when m evaluations are performed in parallel and at least one move has been accepted, the algorithm assumes that $\frac{m+1}{m-l+1}$ moves have been attempted (l: number of rejected moves). On the other hand, if no moves have been accepted, the algorithm assumes that m moves have been attempted.
- Low temperature mode: in case of $\lambda(T) < \frac{1}{m}$, the different processors perform moves, in parallel, until a move is accepted. Then, the processors are synchronized and their memories updated with the new solution. Now, the next evaluation step takes place.

The high temperature mode is inefficient since many moves are not counted. In addition, since few moves are rejected at high temperatures, the value $\frac{m+1}{m-l+1}$, which represents speedup of the algorithm, approaches one. The results produced by the work showed that the computing time in this mode is higher than the sequential mode. It is important to note, that the time spent in this mode (i.e., high temperature) increases as the number of processor grows. In the low temperature mode, the algorithm is biased towards moves that can be generated rapidly.

The above problem was also encountered in the work by [37]. In their work, the authors evaluate the cost function for VLSI circuit placement in parallel, while simultaneously an additional processor is selecting the next state. However, the maximum speedup that was reported is bounded by $1 + 2\alpha + \eta$, where α is the average number of cells affected per move, and η is the average number of wires affected per move. Most cost function based computations exploit fine-grain parallelism, which lead to communication and synchronization dominating this type of algorithms [26], and so speedups due to parallelism are minimal.

Another approach to developing parallel SA is based on exploiting parallelism in making accept-reject decisions [62]. If a processor is assigned to each node in the tree shown below (Fig. 37.3), then cost function evaluation for each suggested move can proceed in parallel.

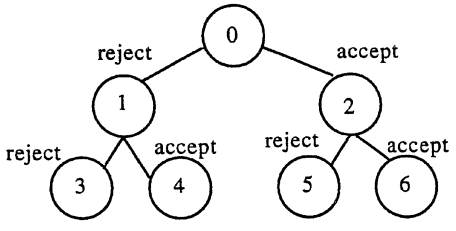


FIGURE 37.3 Decision tree.

The above algorithm maintains the serial decision sequence. The algorithm is based on the concurrency technique of *speculative computation* in which work is performed before it is known whether or not it is needed. As noted earlier, each move in the SA algorithm results in either an accept or a reject decision. Therefore, while one processor evaluates the current move, two other moves can be speculatively computed.

For each temperature, the algorithm forms an unbalanced binary tree of processors which identifies the future work which is likely to be needed. Computation in the tree begins at the root and continues until a processor makes a decision but does not have a corresponding slave processor. The current solution is communicated to the root.

The main problem with the work proposed by [62] is the accompanied heavy communication cost. The root sends the solution to the reject node, generates a move, and then sends a new solution to the accept processor. Then, each node transfers the solutions to its slaves. After making a decision, the node at the end of the correct path sends its solution to the root processor. In the case of large networks of processors,

the solution has to travel through several nodes to reach the root processor. In at least one experiment, the reported computation time of the sequential algorithm was lower than that of the parallel algorithm [61].

A more efficient variation of the above algorithm was developed in [42, 43], which attempts to minimize the communication overhead. It is based on the observation that the difference between the solution at a node and any of its slaves and master is a maximum of one move. The gist of the work is to communicate only the new moves.

Initially, the root processor broadcasts a solution to all the processors in the network. Moreover, the algorithm ensures that all the nodes have the current solution at the beginning of each computation phase. The root generates the first move and then each other processor receives only the number of moves required for its operation.

Figure 37.4 shows an unbalanced tree which consists of ten processors. Each node has a maximum of two slaves: the one on the left is the reject processor, while the one on the right is the accept processor. Node (0) is the root processor and the dotted arrows show the communication direction (i.e., they point to the destination processor). Note that the number attached to the arrow is the number of moves required to be transferred.

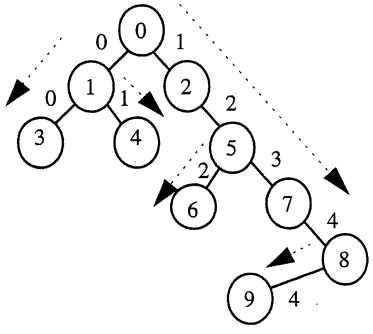


FIGURE 37.4 Moves transfer in a 10-processor unbalanced tree.

In Fig. 37.4, node (1) is the root's reject processor—it does not require any moves. The same applies to node (3). The move generated by node (1) is sent to node (4). Node (2) requires the move generated by the root processor. It receives the move, generate its new move, and then sends both moves to node (5). The technique is applied to all the nodes in the network. A node evaluates its solution immediately after communicating with its slaves.

Once the correct path is identified, the node at the end of this path transfers the number of moves required to update its neighbors which, in turn, send the number of moves to update their neighbors. The maneuver is repeated until all the nodes are updated. The new iteration begins when the root processor completes communicating with its slaves.

Figure 37.5 illustrates the updating process in the 10-processor network described above. It is assumed that the correct path ends at node (8), hence the move generated by node (8) is accepted. Nodes (7) and (9) need to know only this move to update their solution. Node (5) requires the two moves generated by nodes (7) and (8). Simultaneously, node (5) updates the solution of its master and its reject processor. The same procedure is applied to all other nodes in the network.

The algorithm exhibits several salient advantages over the work of [61, 62], for example,

- For problems of large size, the communication overhead is significantly reduced.
- The communication time depends only on the shape of the tree and the number of its nodes. Thus, in comparison to [61, 62], as the size of the problem increases, the performance of the method improves as compared with a sequential solution.

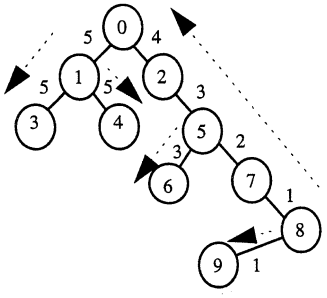


FIGURE 37.5 Updating the nodes of the 10-processor unbalanced tree.

- At low temperatures, less moves are required to be transferred because there are more rejects and, accordingly, the efficiency of the algorithm increases.

The two methods (i.e., [62] and [42]) were compared using the Traveling Salesman Problem as a benchmark [46]. Figures 37.6, 37.7, and 37.8 below show the results of both algorithms in the case of 20, 50, and 100 cities. The results show that the performance of the [42] algorithm improves as the number of cities increases.

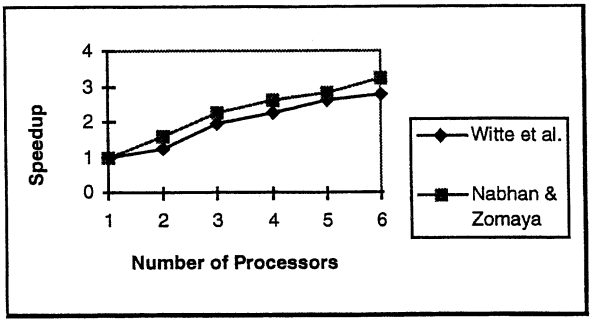


FIGURE 37.6 Speedup comparisons for 20 cities.

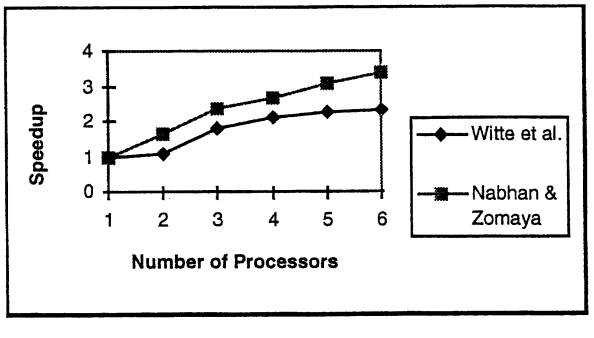


FIGURE 37.7 Speedup comparisons for 50 cities.

The annealing schedule for the above experiments was $T_k = \eta^k T_{k-1}$, where $\eta = 0.99$ (to ensure a slow cooling rate). Normally, an initial high T is chosen to allow a good search of the solution space

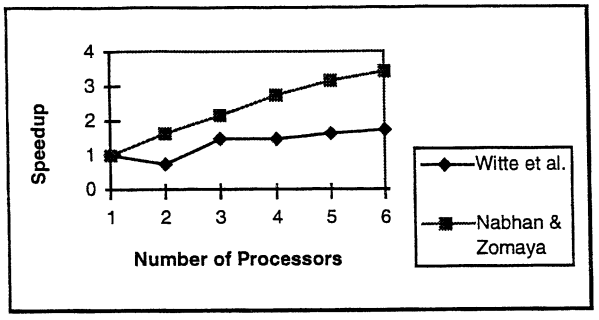


FIGURE 37.8 Speedup comparisons for 100 cities.

at the outset. The maximum number of moves (μ) attempted at each T was set equal to 100 times the number of cities. At each T a series of moves are generated until μ moves are computed, or the number of accepted moves equals to 0.1μ. The initial T (i.e., T_0) was set to 50, and the final (freezing) T set to 0.2. It is important to note, that in all the experiments, the solutions in the case of the parallel algorithm were identical to that of the sequential algorithm. For more details the reader is referred to [42].

The parallel SA algorithm developed in [43] was applied with success to wider range of problems such as solving the scheduling problem in parallel processor systems. In general, SA implementations based on speculative computation seem to produce high quality solutions in reasonable amounts of time [56].

Other parallel implementations of the SA algorithm have been proposed in the literature. These are based on the idea that the SA algorithm can tolerate, to some extent, increases or errors in the cost function without jeopardizing convergence to the correct global minimum. This type of parallel algorithms is known as *asynchronous* [26], since different processors can operate on old data, under certain conditions, and still converge to a reasonable solution [5, 14, 34, 50]. In this case, the processors do not need to update their states in a synchronous fashion and can minimize the amount of information that they need to transmit. This helps in reducing the synchronization barrier quite considerably.

However, Greening [26] argues that one could construct a problem that can converge to a good solution in the sequential SA algorithm, but which converges to a local minimum in an asynchronous case. A number of other approaches can be found in the literature that employ both synchronous and asynchronous techniques [26]. Overall, synchronous methods seem to provide higher quality solutions.

In general, an efficient parallel SA algorithm should be *scalable*. Scalability is a general requirement for any well-designed parallel algorithm [64]. Scalability requires that a given algorithm be reasonably efficient when the size of the problem grows and the number of processors increases.

37.7 Research Issues and Summary

This chapter reviewed the SA algorithm and provided an insight into its origins and its more recent developments. The algorithm is a very powerful optimization technique that was motivated by the behavior of condensed matter at low temperatures. The analogy arises from the way that liquids freeze or metals cool and anneal.

The basic idea is to allow enough time for the states (atoms) of the system to rearrange themselves so that they achieve the lowest energy state (cost function value). Of course, in the case of an optimization problem, the sequence of states is generated through some Monte Carlo probability selection method.

The main strength of the SA algorithm is that of accepting states that may not improve immediately the value of the cost function. Accepting such states is limited by some probabilistic acceptance criterion. This enables the algorithm to escape from local minima and to find an optimal (globally minimal) solution. An important factor discussed in this chapter that contributes to the success of the algorithm in finding a good solution is the annealing schedule.

Parallel versions of the SA algorithm were also discussed in this chapter. These algorithms aim to speedup the rate of convergence of the sequential algorithm. One important issue that a parallel algorithm needs to maintain is the continuity of the Markov chain. If the Markov chain is broken, then there is no guarantee that the parallel algorithm will eventually converge, and most probably it will get trapped into local minima.

The SA algorithm (sequential or parallel) has been applied to a wide range of problems as can be seen from the list of references at the end of this chapter. However, there is still great potential in using the SA algorithm to solve more formidable problems that can be formulated as optimization problems. There is however, an even greater opportunity: using the SA algorithm in combination with other stochastic techniques such as neural networks and genetic algorithms to produce more powerful problem solving tools.

Another aspect that needs more research is the production of more efficient parallel SA algorithms. At this stage, the algorithms that provide high quality solutions within reasonable amounts of time are of the synchronous type. This means that their speedup is hindered by the need to synchronize and also to communicate massive amounts of data. These performance limitations don't occur in the case of asynchronous SA algorithms, however, asynchronous algorithms don't produce high quality solutions and might fail to converge. Therefore, there is a need to develop more efficient parallel SA algorithms that provide high quality solutions along with large speedup ratios.

37.8 Defining Terms

Asymptotic optimality: A given algorithm is called asymptotically optimal when the quality of the solution that can be obtained improves the more the algorithm is executed.

Detailed reversibility: The probability of a transition from state i to state j is equal to the probability in the reverse direction, from state j to state i.

Global optimization: Finding the lowest minimum of a nonlinear function $f(x)$ in some closed subregion U of $\Re^n$, in cases, where $f(x)$ may have multiple local minima in this subregion.

Irreducible Markov chain: A Markov chain is irreducible if for every two states i and j, there is a sequence of iterations or moves which enables j to be reached from i.

Local minimum: This is any point that has the lowest cost function value among all points in some open n-dimensional region around itself.

Markov chain: A sequence of trials where the outcome of any trial corresponds to the state of the system. A main characteristic of the Markov chain is that the new state depends only on the previous state and not on any earlier state.

Scalability: A parallel algorithm is scalable if it is capable of delivering an increase in performance proportional to the increase in the number of processors utilized.

References

[1] Aarts, E. and Korst, J., *Simulated Annealing and Boltzmann Machines,* Wiley, Chichester, U.K., 1989.

[2] Abramson, D., A Very High Speed Architecture for Simulated Annealing. *IEEE Computer,* 25(5), 27–36, 1992.

[3] Aluffi-Pentini, F., Parisi, V., and Zirilli, F., Global Optimization and Stochastic Differential Equations, *Journal of Optimization Theory and Applications,* 47, 1–16, 1985.

[4] Azencott, R., Ed., *Simulated Annealing: Parallelization Techniques,* 1st ed., Wiley, New York, 1992.

[5] Banerjee, P., Jones, M.H., and Sargent, J.S., Parallel Simulated Annealing Algorithms for Cell Placement on Hypercube Multiprocessors. *IEEE Transactions on Parallel and Distributed Systems,* 1(1), 91–106, 1990.

[6] Belisle, C.J.P., Convergence Theorems for a Class of Simulated Annealing Algorithms on $\Re^d$. *Journal of Applied Probability,* 29(4), 885–895, 1992.

[7] Boender, C.G.E., and Romeijn, H.E., Stochastic Methods. In *Handbook of Global Optimization,* Horst, R. and Pardalos, P.M., Eds., 829–869, Kluwer Academic Publishers, The Netherlands, 1995.

[8] Bohachevsky, I.O., Johnson, M.E., and Stein, M.L., Generalized Simulated Annealing for Function Optimization, *Technometrics,* 28, 209–217, 1986.

[9] Casotto, A., Romeo, F., and Sangiovanni-Vincentelli, A., A Parallel Simulated Annealing Algorithm for the Placement of Macro-Cells, *IEEE Transactions on Computer-Aided Design,* 6, 838–847, 1987.

[10] Cerny, V., A Thermodynamic Approach to the Traveling Salesman Problem, *Journal of Optimization Theory and Applications,* 45, 41–51, 1985.

[11] Chiang, T.S., Hwang, C.R., and Sheu, S.J., Diffusion for Global Optimization in $\Re^n$, *SIAM Journal on Control and Optimization,* 25, 737–753, 1987.

[12] Cinlar, E., *Introduction to Stochastic Processes,* Prentice-Hall, Englewood Cliffs, NJ, 1975.

[13] Corana, A., Marchesi, M., Martini, C., and Ridella, S., Minimizing Multimodal Functions for Continuous Variables with the "Simulated Annealing" Algorithm, *ACM Transactions on Mathematical Software,* 13, 262–280, 1987.

[14] Darema, F., Kirkpatrick, S., and Norton, A.V., Parallel Algorithms for Chip Placement by Simulated Annealing, *IBM Journal of Research and Development,* 31, 259–260, 1987.

[15] Dekkers, A., and Aarts, E., Global Optimization and Simulated Annealing, *Mathematical Programming,* 50, 367–393, 1991.

[16] Dixon, L.C.W., and Szego, G.P., *Towards Global Optimization,* North-Holland, The Netherlands, 1978.

[17] Donnett, J.G., Simulated Annealing and Code Partitioning for Distributed Multimicroprocessors. Department of Computer and Information Science, Queen's University, Kingston, Canada, 1987.

[18] Dueck, G., New Optimization Heuristics: The Great Deluge Algorithm and the Record-to-Record Travel, *Journal of Computational Physics,* 104, 86–92, 1993.

[19] Dueck, G. and Scheuer, T., Threshold Accepting: A General Purpose Optimization Algorithm Appearing Superior to Simulated Annealing, *Journal of Computational Physics,* 90, 161–175, 1990.

[20] Gelfand, S.B. and Mitter, S.K., Simulated Annealing. In *Advanced School on Stochastics in Combinatorial Optimization,* Andreatta, G., Mason, F., and Serfami, P., Eds., 1–51. World Scientific Publishing, Singapore, 1987.

[21] Geman, S., and Geman, D., Stochastic Relaxation, Gibbs Distributions, and the Bayesian Restoration of Images, *IEEE Transactions on Pattern Analysis and Machine Intelligence,* 6(6), 721–736, 1984.

[22] Geman, S., and Hwang, C.R., Diffusions for Global Optimization, *SIAM Journal of Control and Optimization,* 24, 1031–1043, 1986.

[23] Gidas, B., Nonstationary Markov Chains and Convergence of the Annealing Algorithm, *Journal of Statistical Physics,* 39, 73–131, 1985.

[24] Golden, B.L., and Skiscim, C.C., Using Simulated Annealing to Solve Routing and Location Problems, *Naval Research and Logistics Quarterly,* 33, 261–279, 1986.

[25] Greene, J.W. and Supowit, K.J., Simulated Annealing Without Rejected Moves, *IEEE Transactions on Computer-Aided Design,* 5(1), 221–228, 1986.

[26] Greening, D.R., Parallel Simulated Annealing Techniques. *Physica D,* 42, 293–306, 1990.
[27] Guus, C., Boender, E., and Romeijn, H.E., Stochastic Methods. In *Handbook of Global Optimization,* Horst, R. and Pardalos, P.M., Eds., 829–869. Kluwer Academic Publishers, The Netherlands, 1995.
[28] Hajek, B., Cooling Schedules for Optimal Annealing, *Mathematics of Operations Research,* 13, 311–329, 1988.
[29] Hansen, P. and Jaumard, B., Lipschitz Optimization. In *Handbook of Global Optimization,* Horst, R. and Pardalos, P.M., Eds., 407–493, Kluwer Academic Publishers, The Netherlands, 1995.
[30] Hastings, H.M., Convergence of Simulated Annealing, *ACM SIGACT,* 17(2), 52–63, 1985.
[31] Horst, R. and Pardalos, P.M., Eds., *Handbook of Global Optimization,* 1st ed., Kluwer Academic Publishers, The Netherlands, 1995.
[32] Huang, M.D., Romeo, F., and Sangiovanni-Vincentelli, A., An Efficient General Cooling Schedule for Simulated Annealing, University of California, Berkeley, 1984.
[33] Isaacson, D.L. and Madsen, R.W., *Markov Chains Theory and Applications,* Wiley, New York, 1976.
[34] Jayaraman, R. and Dareme, F., Error Tolerance in Parallel Simulated Annealing Techniques, *Proceedings of the International Conference on Computer Design,* 545–548, 1988.
[35] Jones, M. and Banerjee, P., An Improved Simulated Annealing Algorithm for Standard Cell Placement, *Proceedings of the International Conference on Computer Design,* 83–86, 1987.
[36] Kirkpatrick, S., Gelatt, Jr., C.D., and Vecchi, M.P., Optimization by Simulated Annealing, *Science,* 220(4598), 671–680, 1983.
[37] Kravitz, S.A. and Rutenbar, R., Placement by Simulated Annealing on a Multiprocessor, *IEEE Transactions on Computer-Aided Design,* 6, 534–549, 1987.
[38] Lundy, M. and Mees, A., Convergence of an Annealing Algorithm, *Mathematical Programming,* 34, 111–124, 1986.
[39] Mahfoud, S.W. and Goldberg, D.E., Parallel Recombinative Simulated Annealing: A Genetic Algorithm, *Parallel Computing,* 21(1), 1–28, 1995.
[40] Metropolis, N., Rosenbluth, A.W., Rosenbluth, M.N., Teller, A.H., and Teller, E., Equations of State Calculation by Fast Computing Machines, *Journal of Chemical Physics,* 21(6), 1087–1092, 1953.
[41] Mills, P.M., Zomaya, A.Y., and Tade, M., *Neuro-Adaptive Process Control: A Practical Approach,* Wiley, New York, 1996.
[42] Nabhan, T.M. and Zomaya, A.Y., A Parallel Simulated Annealing Algorithm with Low Communication Overhead, *IEEE Transactions on Parallel and Distributed Systems,* 6(12), 1226–1233, 1995.
[43] Nabhan, T.M. and Zomaya, A.Y., A Parallel Computing Engine for a Class of Time Critical Processes, *IEEE Transactions on Systems, Man and Cybernetics,* Part B, 27(4), 27(5), 774–786, 1997.
[44] Narayan Bhat, U., *Elements of Applied Stochastic Processes,* 2nd ed., Wiley, New York, 1984.
[45] Patterson, D.W., *Artificial Neural Networks,* 1st ed. Prentice-Hall, Singapore, 1996.
[46] Press, W.H., Flanner, B.P., Teuklosky, S.A., and Vetterling, W.T., *Numerical Recipes in C: The Art of Scientific Computing,* Cambridge University Press, Cambridge, 1988.
[47] Quadrel, R.W., Woodbury, R.F., Fenves, S.J., and Talukdar, S.N., Controlling Asynchronous Team Design Environments by Simulated Annealing, *Research in Engineering Design,* 5, 88–104, 1993.
[48] Ratschek, H. and Rokne, J., *New Computer Methods for Global Optimizations,* Ellis-Horwood, Chichester, U.K., 1988.
[49] Romeo, F., Sechen, C., and Sangiovanni-Vincentelli, A., Simulated Annealing Research at Berkley, *Proceedings of the International Conference on Computer Design,* 652–657, 1984.

[50] Rose, J.S., Snelgrove, W.M., and Vranesic, Z.G., Parallel Standard Cell Placement Algorithms with Quality Equivalent to Simulated Annealing, *IEEE Transactions on Computer-Aided Design,* 7, 387–396, 1988.
[51] Roussel-Ragot, P. and Dreyfus, G., A Problem Independent Parallel Implementation of Simulated Annealing: Models and Experiments, *IEEE Transactions on Computer-Aided Design,* 9(8), 827–835, 1990.
[52] Rubenstein, R., *Simulation and the Monte Carlo Method,* Wiley, New York, 1981.
[53] Sechen, C. and Sangiovanni-Vincentelli, A., The TimberWolf Placement and Routing Package, *Proceedings of the Custom Integrated Circuits Conference,* 522–527, 1984.
[54] Siarry, P., Bergonzi, L., and Dreyfus, G., Thermodynamics Optimization of Block Placement, *IEEE Transactions on Computer-Aided Design,* 6(2), 211–221, 1987.
[55] Slagle, J., Bose, A., Busalacchi, P., Park, B., and Wee, C., *Enhanced Simulated Annealing for Automatic Reconfiguration of Multiprocessors in Space,* Department of Computer Science, University of Minnesota, Minneapolis, 1989.
[56] Sohn, A., Parallel N-ary Speculative Computation of Simulated Annealing, *IEEE Transactions on Parallel and Distributed Systems,* 6, 997–1005, 1995.
[57] van Laarhoven, P.J.M. and Aarts, E.H.L., *Simulated Annealing: Theory and Applications,* Kluwer Academic Publishers, Boston, 1987.
[58] Vanderbilt, D. and Louie, S.G., A Monte Carlo Simulated Annealing Approach to Optimization Over Continuous Variables, *Journal of Computational Physics,* 56, 259–271, 1984.
[59] Vecchi, M.P. and Kirkpatrick, S., Global Wiring by Simulated Annealing, *IEEE Transactions on Computer-Aided Design,* 2(4), 215–222, 1983.
[60] White, S., Concepts of Scale in Simulated Annealing, *Proceedings of the International Conference on Computer Design,* 646–651, 1984.
[61] Witte, E.E., Parallel Simulated Annealing Using Speculative Computation. M.S. thesis, Washington University, 1990.
[62] Witte, E.E., Chamberlain, R.D., and Franklin, M.A., Parallel Simulated Annealing Using Speculative Computation, *IEEE Transactions on Parallel and Distributed Systems,* 2(4), 483–494, 1991.
[63] Wong, D.F., Leong, H.W., and Liu, C.L., *Simulated Annealing for VLSI Design,* 1st ed., Kluwer Academic Publishers, The Netherlands, 1988.
[64] Zomaya, A.Y., Ed., *Parallel and Distributed Computing Handbook,* 1st ed., McGraw-Hill, New York, 1996.

Further Information

The list of references given in this chapter is quite extensive, and it shows that the use of the SA algorithm is quite ubiquitous. Actually, at the time of the writing of this chapter, one HotBot search on the Web returned 14,797 hits. This gives the reader some indication of the popularity of the SA algorithm.

There are no specific journals or conference proceedings that exclusively publish material related to the SA algorithm. However, it could be noticed from the list of references that the *IEEE Transactions on Computer-Aided Design* and the *IEEE Transactions on Parallel and Distributed Systems* often publish material related to the SA algorithm. Also, proceedings of the IEEE/ACM Conference on Computer-Aided Design and the International Conference on Computer Design are good sources of material. The range of applications is quite diverse: parallel processing, graph drawing (Chapter 9 in this volume), VLSI design (Chapter 23 in this volume), scheduling (Chapter 35 in this volume), to name a few.

A good starting point is the paper by Kirkpatrick et al. [36] and the books given in references [1, 57], after which one could proceed to more advanced topics, such as convergence issues and other theoretical themes [3, 6, 11, 20, 21, 23, 30, 38] and parallelizing techniques [4, 26]. One could also try to learn

more bout the type of applications that the SA algorithm can be applied to solve by reading some of the references cited in this chapter. Furthermore, the reader might want to download some of the readily available software packages from Web. These could provide a valuable starting point to experiment with SA code.

38

Cryptographic Foundations

Yvo Desmedt
University of Wisconsin – Milwaukee

38.1 Introduction

Cryptography studies methods to protect several aspects of data, in particular *privacy* and *authenticity*, against a malicious adversary who tries to break the security. In contrast with steganography, where the data and its existence is physically hidden, cryptography transforms the data mathematically, usually using a key. **Cryptanalysis** is the study of methods to break cryptosystems.

Cryptography has been studied for centuries, although initially it focused only on protecting privacy. Originally it was used in the context of military and diplomatic communication. Most of these historical cryptoschemes have no practical value nowadays since they have been cryptanalyzed, i.e., broken. However, it should be noted that it has taken **cryptanalysts** (those researchers or technicians trying to break cryptosystems) more than 300 years to find a general method to solve polyalphabetic ciphers with repeating keywords (see "Other Historical Ciphers"). This contrasts with popular modern cryptosystems, such as DES and RSA (see Sections 38.5 and "RSA" of Chapter 39), that have only been around for a few decades, which brings us now to modern cryptography.

Modern cryptography differs from historical cryptography in many respects. First of all, mathematics plays a more important role than ever before. By means of probability theory, Shannon was able to prove that Vernam's one-time pad (see Section 38.4) is secure. Second, the rather new area of computational complexity has been used as a foundation for cryptography. Indeed, the concept of **public key**, which facilitates the use of cryptography (see "Conventional Cryptography Versus Public Key") finds its origin there. Third, the widespread use of communication implies that cryptography is no longer a uniquely

military topic. High-speed networks and computers are responsible for a world in which postal mail is being replaced by electronic communication in such applications as bank transactions, access to worldwide databases as in the World Wide Web, e-mail, etc. This also implies a whole new range of security needs that need to be addressed, for example, anonymity (see Chapter 44), authenticity (see Chapter 40), commitment and identification, law enforcement, nonrepudiation (see Chapter 40), revocation, secure distributed computation, timestamping, traceability, witnessing, etc.

To illustrate the concept, we will first describe some historical cryptosystems in Section 38.2, explain how these can be broken and, in Section 38.3, define cryptosystems.

38.2 Historical Cryptosystems

We will now discuss some historical cryptosystems to lay the foundation for describing how they are broken. For a more complete survey of historical cryptosystems, the reader may refer to the literature.

The Caesar Cipher and Exhaustive Key Search

One of the oldest cryptosystems is the *Caesar cipher*, often incorrectly cited as the first cryptosystem. Caesar replaced each symbol in the original text, now called **plaintext** or **cleartext**, by one that was three positions further in the alphabet, counted cyclically. The word "plaintext," for example, would become "sodlqwhaw" in this system. The result is called **ciphertext**. The problem with this scheme is that anyone who knows how the text is encoded can break it. To prevent this, a *key* is used.

To describe a more modern variant of the Caesar cipher, let n be the cardinality of the alphabet being used, which is 26 for the English alphabet, or 27 when the space symbol is included in the alphabet. (In many old cryptoschemes, the space symbol was dropped since it would facilitate breaking the code.) The first symbol of the plaintext is mapped into the number 0, the second into 1, etc. To **encrypt** with the Caesar cipher, one adds modulo n the key k to the symbol m, represented as an integer between 0 and $n-1$. (Two integers a and b are equivalent modulo n, denoted as $a \equiv b \bmod n$, when a and b have the same nonnegative remainder when divided by n). The corresponding symbol, then, in the ciphertext is $c = m + k \bmod n$, where the equality indicates that $0 \leq c < n$. If a long enough message contains redundancy, as plain English does, then an **exhaustive search** of all possible keys will reveal the correct plaintext. **Decryptions** (the process that permits the person who knows the secret key to compute the plaintext from the ciphertext) with the wrong key will (likely) not produce an understandable text; therefore, the keyspace in the Caesar cipher is too small.

Substitution Cipher and Ciphertext-Only Attack

We will now consider the substitution cipher. In plaintext, each symbol m is replaced by the symbol $E_k(m)$, specified by the key k. To allow unique decryption the function E_k must be one-to-one. Moreover, if the same symbols are used in the ciphertext as in the plaintext, it must be a bijection. If the key can specify any such bijection, the cipher is called a *simple substitution cipher*. Obviously, for the English alphabet there are $26! = 403291461126605635584000000$, roughly $4 * 10^{26}$, different keys. We will now discuss the security of the scheme, assuming that the cryptanalyst only knows the ciphertext and the fact that a substitution cipher was used. Such an attack is called a *ciphertext-only attack*. Note that an exhaustive key search would take too long on a modern computer. Indeed, a modern parallel computer can perform 10^9 operations per second. For simplicity, assume that such a computer could perform 10^9 symbol decryptions per second. One wonders then how long the ciphertext needs to be before one can be certain that the cryptanalyst has found a sufficiently correct key. This measure is called the *unicity distance.* Shannon's theory of secrecy [25] tells us that this is 28 symbols for an English text. An exhaustive key search would roughly take $3.6 * 10^{11}$ years before finding a sufficiently correct key. However, a much faster method for breaking a substitution cipher exists, which we will now describe.

In English the letter "e" is the most frequently used. Furthermore, no other letter has a frequency of occurrence that comes close to that of "e." A cryptanalyst starts the procedure by counting how many times each letter appears in the ciphertext. When the ciphertext is long enough, the most frequent letter in the ciphertext corresponds to the letter "e" in the plaintext. The frequencies of the letters "T,O,A,N,I,R,S,H" are too similar to decide by which letter they have been substituted. Therefore the cryptanalyst will use the frequency distribution of two or three consecutive letters, called a *digram* and a *trigram*. When the space symbols have been discounted, the most frequent digrams are: "th;" "e" as the first letter, decreasing in order as follows: "er,ed,es,en,ea;" and "e" as the second letter: "he,re." The digram "he" is also quite common. This permits the identification of the letter "h," and then the letter "t." The next step is to distinguish the vowels from the consonants. With the exception of the digrams "ea,io" two vowels rarely follow one another. This allows one to identify the letter "n," since 4 out of 5 letters following "n" are vowels. Using similar properties of other digrams and trigrams, the full key is found. If mistakes are made, they are easily spotted and one can recover using backtracking.

Ideal Ciphers and Known-Plaintext Attack

Redundancy in a language permits breaking a substitution cipher; however, one may question the security of a text if it is compressed first. Shannon proved that if *all* redundancy is removed by the source coder, a cryptanalyst using a ciphertext-only attack cannot find a unique plaintext solution. In fact there are 26! meaningful plaintexts! Shannon [25] called such systems *ideal*. However, one cannot conclude that such a system is secure. Indeed, if a cryptanalyst knows just one (not too short) plaintext and its corresponding ciphertext, finding the key and breaking all future ciphertexts encrypted with the same key is a straightforward procedure. Such an attack is known as a *known-plaintext attack*.

Other types of attacks are the *chosen text attacks,* which comprise *chosen-plaintext* and *chosen-ciphertext attacks*. In a chosen-plaintext attack, the cryptanalyst succeeds in having a plaintext of his choice being encrypted. In a chosen-ciphertext attack, it is a ciphertext of his choice that is decrypted. The cryptanalyst can, in this context, break the simple substitution cipher having the string of all the different symbols in the alphabet encrypted or decrypted. Chosen-plaintext attacks in which the text is not excessively long are quite realistic in a commercial environment. Indeed, company A could send an (encrypted) message about a potential collaboration to a local branch of company B. This company, after having decrypted the message, will most likely forward it to its headquarters, encrypting it with a key that the cryptanalyst in company A wants to break. In order to break it, it is sufficient to eavesdrop on the corresponding ciphertext. Note that a chosen-ciphertext attack is a little harder. It requires access to the output of the decryption device, for example, when the corresponding, and likely unreadable, text is discarded.

Although ideal ciphers are a nice information theoretical concept, their applicability is limited in an industrial context where standard letters, facilitating a known-plaintext attack, are often sent. Information theory is unable to deal with known-plaintext attack. Indeed, finding out whether a known-plaintext is difficult or not is a computational complexity issue. A similar note applies to the unicity distance. Many modern ciphers, such as DES (see Section 38.5), have unicity distances shorter than that of the substitution cipher, but no method is known to break them in a very efficient way. Therefore, we will not discuss in further detail the results of ideal ciphers and unicity distance.

Other Historical Ciphers

Before finishing the discussion on historic cryptosystems, we will briefly mention the transposition cipher and the polyalphabetic ciphers with repeating keywords. In a transposition cipher, also known as a permutation cipher, the text is split into blocks of equal length and the order of the letters in each block is mixed according to the key. Note that in a transposition cipher the frequency of individual letters is not affected by encrypting the data, but the frequency of digrams is. It can be cryptanalyzed by trying to restore the distribution of digrams and trigrams. In most polyalphabetic ciphers, known as periodic

substitution ciphers, the plaintext is also split into blocks of equal length, called the period d. One uses d substitution ciphers by encrypting the i^{th} symbol ($1 \leq i \leq d$) in a block using the i^{th} substitution cipher. The cryptanalysis is similar to the simple substitution cipher once the period d has been found. The Kasiski method [14] analyzes repetition in the ciphertext to find the exact period. Friedman [10] index of coincidence to find the period is beyond the scope of this introduction. Other types of polyalphabetic ciphers are running key ciphers, the Vernam's one-time pad (see Section 38.4), and rotor machines, such as Enigma.

Many modern cryptosystems are **polygram substitution ciphers,** which are a substitution of many symbols at once. To make them practical, only a subset of keys is used. For example DES (see Section 38.5) used in Electronic Code Book mode substitutes 64 bits at a time using a 56 bit key instead of a $\log_2(2^{64}!)$ bit key (which is longer than 2^{64} bits) if any polygram substitution were allowed. Substitution and transposition ciphers are examples of **block ciphers,** in which the plaintext and ciphertext are divided into strings of equal length, called blocks, and each block is encrypted one at a time.

38.3 Definitions

As mentioned in Section 38.1, modern cryptography covers more than simply the protection of privacy, but to give all these definitions is beyond the scope of this chapter. We will focus on privacy and authenticity.

Privacy

DEFINITION 38.1 A cryptosystem used to protect privacy, also called an *encryption scheme* or system, consists of an *encryption* algorithm E and a *decryption* algorithm D. The input to E is a plaintext message $m \in M$ and a key k in the key space K. The algorithm might use randomness $r \in R$ as an extra input. The output of the encryption is called the ciphertext $c \in C$ and $c = E_k(m) = f_E(k, m, r)$.

The decryption algorithm (which may use randomness) has input a key $k' \in K'$ and a ciphertext $c \in C$ and outputs the plaintext m, so, $m = D_{k'}(E_k(m))$. To guarantee unique decryption, the following must be satisfied:

$$\text{for all } k \in K, \text{ for all } m, \text{ for all } m' \neq m, \text{ for all } r \text{ and } r' : f_E(k, m, r) \neq f_E(k, m', r') \ .$$

Security

Clearly, in order to prevent any unauthorized person from decrypting the ciphertext, the decryption key k' must be secret. Indeed, revealing parts of it may help the cryptanalyst.

The types of attacks that a cryptanalyst can use have been informally described in Section 38.2. The most powerful attack is the adaptive chosen text attack, in which the cryptanalyst employs several chosen texts. In each run, the cryptanalyst observes the output and adapts the next chosen text based on the previous ones and the output of previous attacks.

An encryption scheme is *secure* if, given public parameters and old plaintext-ciphertext pairs obtained using known-plaintexts and/or chosen text attacks, the new ciphertext is indistinguishable from a random string of the same length uniformly chosen. We will discuss this further in Sections "Levels of Security" and "Security."

Authenticity

While the terminology is rather standard for cryptosystem's protecting privacy, that for authenticity is not. Research and a better understanding of the topic have made it clear that using concepts as encryption and decryption makes no sense. This was done in the early stages when the concept was introduced.

DEFINITION 38.2 A cryptosystem used to protect authenticity, also called an *authentication scheme* or system, consists of an *authenticator generation* algorithm G and a *verification* algorithm V. The input to G is a message $m \in M$ and a key k' in the key space K'. The algorithm might use randomness $r \in R$ as an extra input. The output of the generation algorithm is the authenticated message (m, a) where m is the message m and a is the authenticator. In other words $(m, a) = G_{k'}(m) = (m, f_G(k', m, r))$.

The inputs of the verification algorithm V (which may use randomness) are a key $k \in K$ and a string (m', a'). The message is accepted as authentic if $V(m', a', k)$ returns a one (if the Turing machine accepts), else it is rejected. To guarantee that authentic messages are accepted one needs that $V_k(G_{k'}(m))$ is (almost always) one.

Security

It is clear that to prevent any unauthorized person from authenticating fraudulent messages the key k' must be secret. Indeed, revealing parts of it may help the cryptanalyst.

The types of attacks that a cryptanalyst can use are similar to those that have been informally described in Section 38.2. The goal of the cryptanalyst has changed. It is to construct a new, not yet authenticated message. The most powerful attack is the adaptive chosen text attack, in which the cryptanalyst employs several chosen messages which are given as input to G. The cryptanalyst observes the output of G in order to adapt the next chosen message based on previous ones and the output of previous attacks.

An authentication scheme is *secure* if, given public parameters and old message-authenticator pairs obtained using known message and/or chosen message attacks, the probability that any cryptanalyst can construct a *new* pair (m', a') which the verification algorithm V will accept as authentic, is negligible. Chapter 40 discusses this in more detail.

Levels of Security

Modern cryptography uses different models to define security. One distinguishes between: heuristic security, as secure as, proven secure, and unconditionally secure.

A cryptographic system or protocol is *heuristically secure* as long as no attack has been found. Many practical cryptosystems fall within this category.

One says that a cryptosystem or protocol is *as secure as* another if it can be proven that a new attack against one implies a new attack against the other and vice versa. A much stronger statement is that a system is proven secure.

To speak about proven security, one must first formally model what security is, which is not always obvious. A system or protocol is said *proven secure relative to an assumption* if one can prove that if the assumption is true, this implies that the formal security definition is satisfied for that system or protocol.

In all aforementioned cryptosystems one usually assumes that the opponent, e.g., the eavesdropper, has a bounded computer power. In the modern theoretical computer science model, this is usually expressed as having the running time of the opponent be bounded above by a polynomial in function of the security parameter, which often is the length of the secret key. Infeasible corresponds with a minimum running time bounded below by a superpolynomial in the length of the security parameter. Note that there is no need to use a polynomial versus superpolynomial model. Indeed, having a huge constant as a lower bound for a cryptanalytic effort would be perfectly satisfactory. For example, according to quantum physics, time is discrete. A huge constant could be the estimated number of time units that have elapsed since the (alleged) big bang.

A cryptosystem is unconditionally secure when the computer power of the opponent is unbounded and it satisfies a formal definition of security. Although these systems are not based on mathematical or computational assumptions, usually these systems can only exist in the real world when true randomness can be extracted from the universe. In Section 38.4 we will discuss an unconditionally secure encryption scheme.

A special class of cryptosystems assumes the correctness of the laws of quantum physics. These cryptosystems are known as quantum cryptography.

Conventional Cryptography Versus Public Key

We will now discuss whether k must remain secret and the relationship between k' and k. If it is easy to compute k' from k, it is obvious that k must also remain secret. The key is unique to a sender–receiver pair. In this case, the cryptosystem is called a **conventional** or **symmetric cryptosystem**.

If, on the other hand, given k it is hard to compute k' and hard to compute a k'', which allows partial cryptanalysis, then the key k can be made public. The system, the concept of which was invented by Diffie and Hellman [7] and independently by Merkle [19], is called a public key or **asymmetric cryptosystem**. This means that for privacy protection each receiver R will publish a personal k_R, and for authentication, the sender S makes k_S public. In the latter case the obtained authenticator is called a **digital signature**, since anyone who knows the correct public key k_S can verify the correctness. The scheme is then called a signature scheme.

The public key k is considered a given input in the discussion on the security of encryption and authentication schemes (see "Privacy – Security" and "Authenticity – Security").

Security

It was stated in the literature that digital signature schemes have the property that the sender cannot deny having sent the message. However, the sender can claim that the secret key was physically stolen. That would allow him to deny ever having sent a message. Such situations must be dealt with by an authority. Protocols have been presented in which the message is being deposited to a notary public or arbiter. Schemes have been developed in which the arbiter does not need to know the message that was authenticated. Another solution is digital timestamping based on cryptography (the signer needs to alert an authority that his public key must have been stolen).

The original description of public key systems did not explain the importance of the authenticity of the public key. Indeed, if it is not authentic, the one who created the fake public key can decrypt messages intended for the legitimate receiver, or can sign claiming to be the sender. So the security is then lost. In practice, this problem is solved by using a *certificate* that is itself a digital signature(s) of a message. It is provided by a known trusted entity(ies) who guarantee(s) that the public key of S is k_S.

We will now explain the need to use randomness in public key encryption systems. If no random input is used, the public key system is vulnerable to a partially known-plaintext attack. In particular, if the sender S, e.g., a supervisor, uses a standard letter to send almost the same message to different receivers, e.g., to inform them about their salary increase, the system does not guarantee privacy. Indeed, any of the receivers of these letters can exhaustively fill in the nonstandard part and encrypt the resulting letter using the public key of the receiver until the obtained ciphertext corresponds with the eavesdropped one! So, if no redundancy is used the eavesdropper can always verify whether a particular message has been encrypted. It is easy to see that if no randomness is used the resulting ciphertext will have a probability distribution which is 0 for all values, except for the deterministic encryption of the plaintext. Although these attacks are well known, very few practical public key systems take precautions against it. Goldwasser and Micali [11] called schemes avoiding this weakness **probabilistic encryption schemes** and presented a first solution (see Section 39.7). It is secure against known-plaintext attack under a computational number theoretic assumption.

Practical Concerns

To be practical the encryption, decryption, authentication and verification algorithms must be efficient. In the modern theoretical computer science model, this is usually expressed as having a running time

bounded by a polynomial in function of the length of the key and by stating that the length of the message is bounded by a polynomial in function of the length of the key.

It is clear that to be useful $M = \{0, 1\}^*$ (or have a polynomial length). However, this is often not the case. A mode or protocol is then needed to specify how the encryption and decryption algorithms (or the authentication and verification algorithms) are used on a longer text. For an example, see Section "The Modes".

38.4 The One-Time Pad

The *one-time pad* (a conventional cryptosystem) and Shannon's [25] analysis of its security is one of the most important discoveries in modern cryptography. We will first discuss the scheme, then give a formal definition of the security, prove it to be secure, and briefly discuss some of the applications of the one-time pad.

The Scheme

In Vernam's one-time pad the key is (at least) as long as the message. Let the message be a string of symbols belonging to the alphabet (a finite set) S, for example $\{0, 1\}$, on which a binary operation "$*$" is defined, for example the exor (exclusive or). We assume that $S(*)$ forms a group.

Before encrypting the message the sender and receiver have obtained a secret key, a string, of which the symbols have been chosen uniformly random in the set S and independent. Let m_i, k_i, and c_i be the i^{th} symbols of, respectively, the message, the key, and the ciphertext, each belonging to S. The encryption algorithm produces $c_i = m_i * k_i$ in S. To decrypt, the receiver computes $m_i = c_i * k_i^{-1}$. *The key is used only once.* This implies that if a new message needs to be encrypted a new key is chosen, which explains the terminology: one-time pad.

It is trivial to verify that this is an encryption scheme. In the case $S(*) = Z_2(+) = \{0, 1\}(+)$, the integers modulo 2, the encryption algorithm and the decryption algorithm are identical and the operation corresponds with an exor (exclusive or). We will now define what privacy means.

Security

DEFINITION 38.3 Shannon defined an encryption system to be perfect when, for a cryptanalyst not knowing the secret key, the message m is independent of the ciphertext c, formally:

$$\text{prob}\,(\mathbf{m} = m \mid \mathbf{c} = E_k(m)) = \text{prob}(\mathbf{m} = m)\ . \tag{38.1}$$

THEOREM 38.1 *The one-time pad is perfect.*

PROOF Let the length of the message be l (expressed as the number of symbols). Then the message, key, and ciphertext belong to $S^l = S \times S \cdots \times S$. Since S^l is a group it is sufficient to discuss the proof for the case $l = 1$. Let $c = E_k(m) = m * k$ in S. Now, if $k' = m^{-1} * c = k$, then $\text{prob}(\mathbf{c} = c \mid \mathbf{m} = m, \mathbf{k} = k') = 1$, else it is zero. Using this fact, we obtain

$$\begin{aligned}
\text{prob}(\mathbf{m} = m, \mathbf{c} = c) &= \sum_{k' \in S} \text{prob}\left(\mathbf{c} = c \mid \mathbf{m} = m, \mathbf{k} = k'\right) \cdot \text{prob}\left(\mathbf{m} = m, \mathbf{k} = k'\right) \\
&= \text{prob}(\mathbf{m} = m, \mathbf{k} = k) \\
&= \text{prob}(\mathbf{m} = m) \cdot \text{prob}(\mathbf{k} = k) \qquad (\mathbf{k} \text{ is independent of } \mathbf{m})
\end{aligned}$$

$$= \quad \text{prob}(\mathbf{m} = m) \cdot \frac{1}{|S|} \qquad (\mathbf{k} \text{ is uniform.})$$

Also, if $c = m' * k'$ then $\text{prob}(\mathbf{c} = E_k(m) \mid \mathbf{m} = m', \mathbf{k} = k') = 1$, else it is 0. This gives

$$\begin{aligned}
\text{prob}(\mathbf{c} = c) &= \sum_{k',m'} \text{prob}\left(\mathbf{c} = c \mid \mathbf{m} = m', \mathbf{k} = k'\right) \cdot \text{prob}\left(\mathbf{m} = m', \mathbf{k} = k'\right) \\
&= \sum_{\substack{k',m' \\ c=m'*k'}} \text{prob}\left(\mathbf{m} = m', \mathbf{k} = k'\right) \\
&= \sum_{m'} \text{prob}\left(\mathbf{m} = m'\right) \cdot \text{prob}\left(\mathbf{k} = \left(m'\right)^{-1} * c\right) \qquad (\mathbf{k} \text{ is independent of } \mathbf{m}) \\
&= \frac{1}{|S|} \cdot \sum_{m' \in S} \text{prob}\left(\mathbf{m} = m'\right) \qquad (\mathbf{k} \text{ is uniform}) \\
&= \frac{1}{|S|},
\end{aligned}$$

implying that $\text{prob}(\mathbf{m} = m, \mathbf{c} = c) = \text{prob}(\mathbf{m} = m) \cdot \text{prob}(\mathbf{c} = c)$.

COROLLARY 38.1 In the one-time pad, the ciphertext has a uniform distribution.

This corollary corresponds with the more modern view on the definition of privacy. Note that it is easy to make variants of the one-time pad that are also perfect.

Shannon proved that the length of the key must be at least, what he called, the entropy (see Chapter 40) of the message. Recent work has demonstrated that the length of the key must be at least the length of the message.

Its Use

The use of the one-time pad to protect private communication is rather limited, since a new secret key is needed for each message. However, many zero-knowledge interactive proofs use the principle that the product in a group of a uniformly chosen element with any element of the group, whatever distribution, gives a uniform element.

Also, the idea of *stream cipher,* in which the truly random tape is replaced by a pseudo-random tape, finds its origin in the one-time pad.

38.5 DES and Block Ciphers

As already observed in "Other Historical Ciphers," many modern encryption schemes, in particular conventional encryption schemes, are (based on) polygram substitution ciphers. To obtain a practical scheme, the number of possible keys needs to be reduced from the maximal possible ones. To solve this problem Shannon [25] proposed the use of *mixing transformation,* in which the plaintext is iteratively transformed using substitution and transposition ciphers. Feistel adapted this idea. The data encryption standard (DES) is a typical example of such an encryption scheme, which we will now describe. The messages belong to $\{0, 1\}^{64}$ and the key has 56 bits. To encrypt a message longer than 64 bits, a *mode* is used. The official modes used for DES are discussed in "The Modes." We will now describe the actual algorithm.

The Algorithm

First note that the detailed design criteria of the DES algorithm are classified, while the algorithm itself is public. The DES algorithm, as described by NBS (now called NIST), consists of three fundamental parts:

- The enciphering computation which follows a typical Feistel approach,
- The calculation of $f(R, K)$, and
- The key schedule calculation.

They are represented, respectively, in Figs. 38.1, 38.2, and 38.4 are briefly described below.

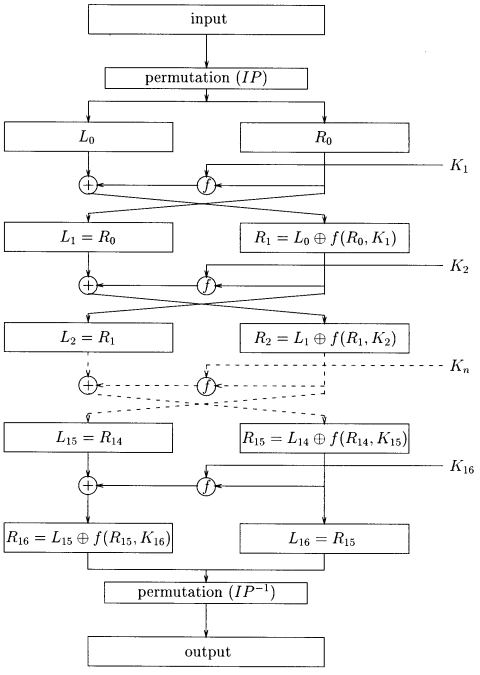


FIGURE 38.1 DES block diagram of the enciphering computation.

First observe that several boxes are used in the DES algorithm. It would be too long an explanation to give the details of all these boxes; this can be found in the literature. The kind of boxes (e.g., permutation) will be mentioned. Remark that the input numbering starts from 0 for some boxes and from 1 for some other ones.

In the *enciphering computation,* the input is first permuted by a fixed permutation IP from 64 bits into 64 bits. The result is split up into the 32 left bits and the 32 right bits, respectively. In Fig. 38.1 this corresponds to L and R. Then a bitwise modulo 2 sum of the left part L_i and of $f(R_i, K_i)$ is carried out. The function f is described in Fig. 38.2. After this transformation, the left and right 32 bit blocks are interchanged. From Fig. 38.1, one can observe that the encryption operation continues iteratively for 16 steps or rounds. In the last round, no interchange of the final obtained left and right parts is performed. The output is obtained by applying the inverse of the initial permutation IP to the result of the 16^{th} round.

In the *calculation of* $f(R_i, K_i)$ the 32 right bits are first expanded to 48 bits in the box E, (see Figs. 38.2 and 38.3) by taking some input bits twice, others only once. Then a bitwise modulo 2 sum of the expanded right bits and of 48 bits of the key, called the *subkey* is performed. This subkey is obtained from the *key schedule calculation,* which will be explained later on. The results of the modulo 2 sum go to eight S-boxes.

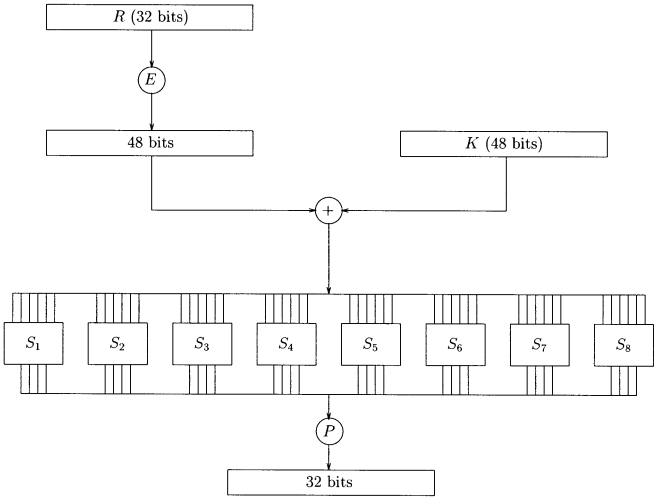


FIGURE 38.2 DES block diagram of $f(R, K)$.

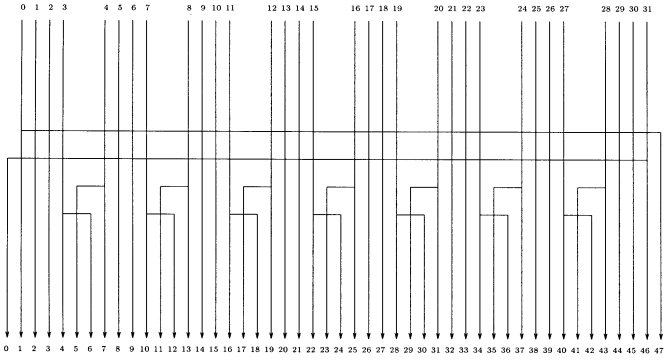


FIGURE 38.3 The box E.

Each S-box has six inputs and four outputs. The S-boxes are nonlinear functions. The 32 output bits of the S-boxes are permuted in the box P.

Let us finally describe the *key schedule calculation* (see Fig. 38.4). The key consists of 64 bits, of which only 56 bits are used. The other 8 bits are not used in the algorithm. The selection of the 56 bits is performed in box PC_1, together with a permutation. The result is split into two 28 bit words C and D. To obtain the 48 bit subkey, the words C and D are first rotated to the left once or twice (as indicated in a table). PC_2, which consists of a selection and a permutation, is then applied to the result. The output of PC_2 is the 48 bit subkey K_i which is used in $f(R_i, K_i)$.

Due to the symmetry, the decryption algorithm is identical to the encryption operation, except that one uses K_{16} as first subkey, K_{15} as second, etc. Since the total number of rotations in the key scheduling is 28, in a 28 bit register, $C_{16} = C_0$ and $D_{16} = D_0$. So the subkeys in the decryption operation can be obtained by rotating the registers C and D to the right instead of to the left and reading the table, specifying the number of rotations from the bottom up.

The Modes

The data encryption standard can be used in four standard modes: the ECB mode (electronic code book), sometimes called the block mode, the CBC mode (cipher block chaining), the CFB mode (cipher feedBack)

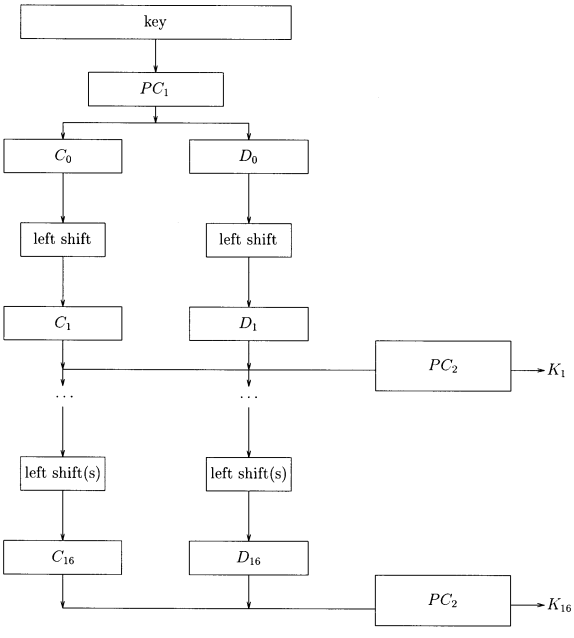


FIGURE 38.4 DES block diagram of key scheduling.

and the OFB mode (output feedBack). Nonstandard modes have been developed and are, for example, used in Kerberos. It is beyond the scope of this chapter to give all the details and properties for these modes; the reader can find them in several publications. Let us quickly survey these modes when used to encrypt data. How these work for decryption is left as an exercise. When we refer to the DES encryption algorithm, we silently assume that each time a 56 bit key is given as input.

The *ECB mode,* in encryption, works similarly to a substitution cipher. The plaintext is divided into blocks, m_i of 64 bits. The i^{th} input to DES is the plaintext block m_i. The output corresponds to the ciphertext block c_i. This mode is not recommended. As we discussed in Section 38.2, most texts contain redundancy. If the same key is used for too long a time, most parts of the plaintext can be recovered.

In the *CBC mode* in encryption, the ciphertext block c_i is the 64 bit output of the DES encryption algorithm. The input to the DES is the bitwise modulo 2 sum (Exor) of the 64 bit plaintext m_i and of the previous 64 bit ciphertext c_{i-1}.

In the *CFB and OFB mode* the ciphertext block c_i and the plaintext block m_i are n bits, where $1 \leq n \leq 64$. In both modes, the ciphertext block $c_i = m_i \oplus \text{Select}_n(\text{DES}_k(d_i))$, where d_i is the input to DES, $\oplus$ is the bitwise modulo 2 sum, and Select_n selects the n most significant bits.

The input d_i to the DES in the CFB encryption mode is constructed from the $(64 - n)$ least significant bits of d_{i-1} (the previous input), shifted to the left by n positions, and concatenated with c_{i-1}, the n bits of the ciphertext.

In the *OFB mode* in encryption, DES is used as a stream cipher (see "Its Use"). The output of DES is used as a pseudo-random generator. The input d_i is the concatenation of the $(64 - n)$ least significant bits of d_{i-1} (the previous input), shifted to the left by n positions, and the n most significant bits of $\text{DES}_k(d_{i-1})$.

Authentication

The CBC and CFB modes of DES can also be used to achieve authentication. The main difference with previous modes is that there is no ciphertext output. When the CBC mode is used, the final output of the DES encryption algorithm is used as authenticator, called the message authentication code (MAC). When the CFB mode is used, the DES encryption algorithm is used one more time.

The maximum length of an authenticator that can be obtained this way is 64 bits. Although this is definitely too short, no standard has fixed this problem yet.

Variants

Since the introduction of the data encryption standard, several variants were proposed. The popularity of these variants is different from country to country. Some examples are FEAL, IDEA, and GOST. The security of these schemes varies. Note that the security of DES is coming near its end. Therefore, the U.S. Federal Government is considering the development of a new, fast alternative to DES. To avoid this weakness, double and triple encryption is used. Both use a 112 bit key. Double encryption DES is obtained by running $\mathrm{DES}_{k_1} \circ \mathrm{DES}_{k_2}$. The triple encryption uses DES as encryption and as decryption, denoted as DES^{-1} giving: $\mathrm{DES}_{k_1} \circ \mathrm{DES}_{k_2}^{-1} \circ \mathrm{DES}_{k_1}$.

38.6 Research Issues and Summary

It is important to remark that cryptography does not solve all modern security problems. Cryptography, although rarely stated explicitly, assumes the existence of secure and reliable hard or software. Some research on secure distributed computation has allowed this assumption to be relaxed slightly. Also in the communication context, modern cryptography only addresses part of a bigger problem. Spread spectrum techniques prevent jamming attacks, and reliable fault tolerant networks reduce the impact of the destruction of communication equipment.

Finally it should be pointed out that cryptography is not sufficient to protect data. For example, it only eliminates the threat of eavesdropping while transmitted remotely. However, data can often be gathered at the source or at the destination using, for example, physical methods. These include theft (physical or virtual), the caption of electromagnetic radiation when data is displayed on a normal screen, etc.

This chapter introduced elementary principles of modern cryptography, including the concepts of conventional cryptosystems and public key systems, several different types of attacks, the different levels of security schemes can have, the one-time pad, and DES as an example of a block cipher.

38.7 Defining Terms

Asymmetric cryptosystem: A cryptosystem in which given the key of one party (sender or receiver, depending from context), it is computationally difficult or, when using information theory, impossible to obtain the other party's secret key.

Block cipher: A family of cryptosystems in which the plaintext and ciphertext are divided into strings of equal length, called blocks, and each block is encrypted one at a time.

Ciphertext: The result of an encryption operation.

Cleartext: The unencrypted, usually readable text.

Conventional cryptosystem: A cryptosystem in which the keys of all parties must remain secret.

Cryptanalysis: The study of methods to break cryptosystems.

Cryptanalyst: A person who (wants to) breaks cryptosystems.

Decryption: The operation that transforms ciphertext into plaintext using a key.

Digital signature: The digital equivalent of a handwritten signature. A digital signature of a message is strongly message-dependent and is generated by the sender using his/her secret key and a suitable public key cryptosystem.

Encrypt: The operation that transforms plaintext into ciphertext.

Exhaustive search: A method to break a cryptosystem by trying all possible inputs, in particular all possible keys.

Plaintext: A synonym for cleartext, i.e., the unencrypted text.

Polygram substitution cipher: A substitution cipher of many symbols at once.

Probabilistic encryption schemes: A public key system in which randomness is used, such that two encryptions of the same ciphertext give, very likely, different ciphertexts.

Public key: A key that is public, or a family of cryptosystems in which a key of one of the parties can be made public.

Symmetric cryptosystem: A system in which it is easy to find one party's key from the other party's key.

References

[1] Bennett, C.H. and Brassard, G., An update on quantum cryptography. In *Advances in Cryptology. Proc. of Crypto 84, Santa Barbara, CA, Aug. 1984, (Lecture Notes in Computer Science 196)*, 475–480. Springer-Verlag, New York, 1985.

[2] Blundo, C., De Santis, A., and Vaccaro, V., On secret sharing schemes. Technical report, Universita di Salerno, 1995.

[3] Brassard, G., *Modern Cryptology*, (Lecture Notes in Computer Science, Vol. 325,) Springer-Verlag, New York, 1988.

[4] Denning, D.E.R., *Cryptography and Data Security,* Addison-Wesley, Reading, MA, 1982.

[5] Desmedt, Y. and Seberry, J., Practical proven secure authentication with arbitration. In *Advances in Cryptology—Auscrypt '92, Proceedings, Gold Coast, Queensland, Australia, Dec. 1992, (Lecture Notes in Computer Science 718),* Seberry, J. and Zheng, Y., Eds., 27–32. Springer-Verlag, 1993.

[6] Desmedt, Y. and Yung, M., Arbitrated unconditionally secure authentication can be unconditionally protected against arbiter's attacks. In *Advances in Cryptology—Crypto '90, Proceedings Santa Barbara, CA, Aug. 1990, (Lecture Notes in Computer Science 537)*, Menezes, A.J. and Vanstone, S.A., Eds., 177–188. Springer-Verlag, 1991.

[7] Diffie, W. and Hellman, M.E., New directions in cryptography. *IEEE Trans. Inform. Theory*, IT–22(6), 644–654, Nov. 1976.

[8] Diffie, W. and Hellman, M.E., Privacy and authentication: An introduction to cryptography. *Proc. IEEE*, 67, 397–427, Mar. 1979.

[9] Eberle, H., A high-speed DES implementation for network applications. In *Advances in Cryptology — Crypto '92, Proceedings Santa Barbara, CA, Aug. 1992. (Lecture Notes in Computer Science 740),* Brickell, E.F., Ed., 521–539. Springer-Verlag, 1993.

[10] Friedman, W.F., The index of coincidence and its applications in cryptography. Riverbank publication no. 22, Riverbank Labs, Geneva, IL, 1920.

[11] Goldwasser, S. and Micali, S., Probabilistic encryption. *Journal of Computer and System Sciences*, 28(2), 270–299, Apr. 1984.

[12] Haber, S. and Stornetta, W.S., How to time-stamp a digital document. *Journal of Cryptology*, 3(2), 99–111, 1991.

[13] Kahn, D., *The Codebreakers,* MacMillan, New York, 1967.

[14] Kasiski, F.W., *Die Geheimschriften und die Dechiffir-kunst,* Mitler & Son, 1863.

[15] Kohl, J. and Newmann, B.C., The kerberos network authentication service, MIT Project, Athena, Version 5.

[16] Konheim, A., *Cryptography: A Primer,* John Wiley, Toronto, 1981.

[17] Massey, J.L., Contemporary cryptology: an introduction. In Simmons, G.J., Ed., *Contemporary Cryptology*, 3–64. IEEE Press, New York, 1992.

[18] Menezes, A., van Oorschot, P., and Vanstone, S., *Applied Cryptography,* CRC Press, Boca Raton, FL, 1996.
[19] Merkle, R.C., Secure communications over insecure channels. *Comm. ACM,* 21, 294–299, 1978.
[20] Meyer, C.H. and Matyas, S.M., *Cryptography: A New Dimension in Computer Data Security,* J. Wiley, New York, 1982.
[21] National Bureau of Standards, *DES Modes of Operation,* Federal Information Processing Standard, publication 81, U.S. Department of Commerce, Washington, DC, 1980.
[22] Popek, G.J. and Kline, C.S., Encryption and secure computer networks. *ACM Computing Surveys,* 11(4), 335–356, Dec. 1979.
[23] Sacco, L., *Manuale di crittografia,* Rome, Italy, 2nd ed., 1936. Translated in English, *Manual of Cryptography,* Aegean Park Press, Laguna Hills, CA, 1977.
[24] Saltzer, J., On digital signatures. *ACM Operating Syst. Rev.,* 12(2), 12–14, Apr. 1978.
[25] Shannon, C.E., Communication theory of secrecy systems, *Bell System Techn. Jour.,* 28, 656–715, Oct. 1949.
[26] Simmons, G.J., Ed., *Contemporary Cryptology,* IEEE Press, New York, 1992.
[27] Stinson, D.R., *Cryptography: Theory and Practice,* CRC Press, Boca Raton, FL, 1995.
[28] U.S. Department of Commerce, National Bureau of Standards. *Data Encryption Standard,* Jan. 1977. *FIPS* PUB 46 (NBS Federal Information Processing Standards Publ.).

Further Information

Since the introduction of public key, the research on cryptography has boomed. Readers interested in applied oriented cryptography should consult [18]. This book discusses block ciphers in great length. Those who prefer a textbook can consult [27].

The most known annual conferences on the topic of cryptography are Eurocrypt and Crypto, running since the early 1980s, of which the proceedings are published in *Springer's Lecture Notes in Computer Science.* More recent conferences include Asiacrypt (which absorbed Auscrypt) and the Workshop on Fast Software Encryption. Some conferences focus on computer security issues as well as cryptography, as for example the Workshop on Cryptographic Protocols and the ACM Conference on Computer and Communications Security. Several local and regional conferences are also organized, many with proceedings published in *Springer's Lecture Notes in Computer Science,* for example the IMA Conference on Cryptography and Coding in Britain. Some results on the topic have appeared in less specialized conferences such as FOCS, STOC, etc.

Articles on the topic appear in a wide variety of journals, but unfortunately several years after the results have been presented at conferences. The *Journal of Cryptology* is dedicated to research on cryptography. Some other specialized journals have a different focus, e.g., *Cryptologia* is primarily on historic aspects of cryptography.

39

Encryption Schemes

Yvo Desmedt
University of Wisconsin–Milwaukee

39.1 Introduction

Several conventional encryption schemes were discussed in Chapter 38. The concept of public key was introduced in the section on "Conventional Cryptography Versus Public Key" of Chapter 38. In this chapter we will discuss some public key encryption systems based on number theory. We first give the minimal number theory and algebraic background needed to understand these schemes from a mathematical viewpoint (see Section 39.2). Then, we present in Section 39.3 the most popular schemes based on number theory. We explain in Section 39.4 the computational number theory required to understand why these schemes run in (expected) polynomial time. To avoid overburdening the reader with number theory, we will postpone the number theory needed to understand the computational aspect until Section 39.4.

39.2 Minimal Background

Algebra

For the reader not familiar with elementary algebra, we review the definition of a semigroup, monoid, group, etc.

DEFINITION 39.1 A set M with an operator "$*$", denoted as $M(*)$, is a *semigroup* if the following conditions are satisfied:

1. The operation is *closed* in M, *i.e.,* $\forall a, b \in M : (a * b) \in M$,
2. The operation is *associative* in M, *i.e.,* $\forall a, b, c \in M : (a * b) * c = a * (b * c)$,

When additionally

3. The operation has an *identity element* in M, i.e., $\exists e : \forall a \in M : a * e = e * a = a$,

we call $M(*)$ a *monoid.* When $M(*)$ is a monoid, we call the cardinality of M the *order* of M.

When using multiplicative notation $(*)$ we will usually denote the identity element as 1 and when using additive notation $(+)$ as 0.

DEFINITION 39.2 In a monoid $M(*)$ the element $a \in M$ has an *inverse* if an element denoted as a^{-1} exists such that

$$a * a^{-1} = a^{-1} * a = e \ ,$$

where e is the identity element in $M(*)$. Two elements $a, b \in M$ *commute* if $a * b = b * a$.

An element having an inverse is often called a *unit.* When working in a monoid, we define $a^{-n} = (a^{-1})^n$ and $a^0 = 1$.

DEFINITION 39.3 A monoid $G(*)$ is a *group* if each element in G has an inverse. It is an *Abelian group* if, additionally, any two elements of G commute.

If H is a monoid (group) and $H \subseteq G$, where G is a monoid (group), then H is called a submonoid (subgroup).

DEFINITION 39.4 Let $M(*)$ be a monoid and $a \in M$. One defines

$$\langle a \rangle = \left\{ a^k \mid k \in N \right\}$$

where N are the natural numbers, i.e., $\{0, 1, \ldots\}$. If $\langle a \rangle$ is a group, we call it a **cyclic group** and call the order of $\langle a \rangle$ the **order** of a, or $\mathrm{ord}(a)$.

Note that in modern cryptography, finite sets are more important than infinite sets. In this respect the following result is interesting.

THEOREM 39.1 *Let $M(*)$ be finite monoid. If $a \in M$ has an inverse in $M(*)$, then $\langle a \rangle$ is an Abelian group. Also,* $\mathrm{ord}(a)$ *is the smallest positive integer m for which $a^m = 1$.*

PROOF Since the set M is finite, positive integers k_1 and k_2 must exist, where $k_1 < k_2$, such that $a^{k_1} = a^{k_2}$. Now, since a is invertible, this means $a^{k_1 - k_2} = 1$. This implies that $\langle a \rangle$ is an Abelian group. So, then $\langle a \rangle = \{a^k \mid 0 \leq k < \mathrm{ord}(a)\}$.

We will now define what a ring is. Unfortunately, there are two different definitions for it in the literature, but we will use the one that is most relevant to modern cryptography.

DEFINITION 39.5 A set R with two operations $+$ and $*$ and two distinct elements 0 and 1 is a ring if

1. $R(+)$ is an Abelian group with identity element 0.
2. $R(*)$ is a monoid with identity element 1.
3. R is distributive, *i.e.,* $\forall a, b, c \in R$:

$$\begin{aligned} a * (b + c) &= (a * b) + (a * c) \\ (a + b) * c &= (a * c) + (b * c) \, . \end{aligned}$$

A ring R is commutative if it is commutative for the multiplication. A ring R in which $R^0 = R \setminus \{0\}$ is an Abelian group for the multiplication is called a field.

A subring is defined similarly as a subgroup.

Number Theory

Integers

We denote the set of integers as Z, and the positive integers as Z^+, i.e., $\{1, 2, \ldots\}$.

DEFINITION 39.6 If α is a real number we call $\lfloor \alpha \rfloor$ the integer such that

$$\lfloor \alpha \rfloor \leq \alpha < \lfloor \alpha \rfloor + 1 \, . \tag{39.1}$$

THEOREM 39.2 $\forall a \in Z \; \forall b \in Z^+ \; \exists q, r \in Z : a = q \cdot b + r$ *where* $0 \leq r < b$.

PROOF From (39.1) it follows that

$$0 \leq \frac{a}{b} - \left\lfloor \frac{a}{b} \right\rfloor < 1 \, .$$

Since $b > 0$ this gives

$$0 \leq a - b \cdot \left\lfloor \frac{a}{b} \right\rfloor < b \, .$$

By taking $q = \lfloor a/b \rfloor$ and $r = a - b \cdot \lfloor a/b \rfloor$ we obtain the result.

The reader has probably recognized q and r as the quotient and nonnegative remainder of the division of a by b. If the remainder is zero then a is a *multiple* of b, or b *divides* a, written as $b \mid a$. We also say that b is a factor of a. If b is different from 1 and a, then b is a nontrivial factor, or a proper divisor of a.

One can categorize the positive integers based on the number of distinct positive divisors. The element 1 has one positive divisor. **Primes** have exactly two, and the other elements, called **composites**, have more than two. In cryptography, a prime number is usually denoted as p or q. Theorem 39.3 is easy to prove using induction and the following lemma.

LEMMA 39.1 When n is composite, the least positive nontrivial factor of n is a prime.

PROOF Let $a, b, c \in Z^+$. It is easy to prove that if $b \mid a$ and $c \mid b$, then $c \mid a$. Also, if b is a nontrivial factor of a, then $1 < b < a$. Using contradiction, this implies the lemma.

THEOREM 39.3 *If $n \geq 2$, then n is the product of primes.*

Greatest Common Divisor

Using a definition other than the standard one and proving these to be equivalent will allow us to introduce several important results. First we will define what an integral modulus is.

DEFINITION 39.7 A modulus is a subset of the integers which is closed under addition and subtraction. A modulus that contains only 0 is called the zero modulus.

THEOREM 39.4 *For each nonzero modulus a positive integer d exists such that all the elements of the modulus are multiplies of d.*

PROOF If such an element exists, it must clearly be the least positive integer. We will now prove its existence by contradiction. Suppose that a is not a multiple of d. Then, using Theorem 39.2, we have $a = qd + r$, where q and r are integers such that $1 \leq r < d$. Clearly, r is then an element of the modulus and is smaller than d. We obtain a contradiction.

It is obvious that if a and b are integers, then $am + bn$, where m and n are any integers, form a modulus. (For those familiar with ring theory, this implies that this modulus is an ideal.) We are now ready to define the greatest common divisor.

DEFINITION 39.8 When a and b are integers, not both zero, then the least positive integer d in the modulus $am + bn$ is the greatest common divisor of a and b, denoted as $\gcd(a, b)$ or (a, b). If $(a, b) = 1$, a and b are called co-prime.

COROLLARY 39.1 Theorem 39.4 implies

$$\exists x, y \in Z : ax + by = \gcd(a, b) \tag{39.2}$$

$$\forall x, y \in Z : \gcd(a, b) \mid ax + by \tag{39.3}$$

$$\text{If } c \mid a \text{ and } c \mid b, \text{ then } c \mid \gcd(a, b)\,. \tag{39.4}$$

PROOF (39.2) and (39.3) follow from Theorem 39.4 and (39.2) implies (39.4).

Due to (39.4), the definition of the greatest common divisor is equivalent to the traditional one.

Congruences

In the section on "The Caesar Cipher and Exhaustive Key Search" of Chapter 38 we defined what it means for two numbers to be equivalent modulo n. It is easy to see that this satisfies the definition of equivalence relation. The corresponding equivalence classes are called *residue classes.* When a is an integer, we let $\widehat{a} = \{b \mid b \equiv a \bmod n\}$. When we work modulo n, we call $Z_n = \{\widehat{0}, \widehat{1}, \ldots, \widehat{n-1}\}$ the set of all residue classes. Other notations for Z_n, which we do not explain, are $Z/(n)$ and Z/nZ. The following theorem is trivial to prove.

THEOREM 39.5 *If $a \equiv b \pmod n$ and $c \equiv d \pmod n$, then $a + c \equiv b + d \pmod n$ and $a * c \equiv b * d \pmod n$.*

Due to this theorem we can say that if $a \equiv b \bmod n$, that a and b are **congruent** modulo n. Now let A and B be two residue classes and we define $A + B = \{a + b \mid a \in A \text{ and } b \in B\}$ and similarly $A * B$. Theorem 39.5 tells us that $A + B$ and $A * B$ are residue classes. This implies the following:

COROLLARY 39.2 Z_m is a commutative ring.

PROOF Given that Z is a commutative ring, this follows easily from Theorem 39.5.

If one selects one representative out of each residue class, we call the resulting set of integers a *complete residue system.* It is easy to see that $\{0, 1, \ldots, m-1\}$ is a complete residue system. Adding (multiplying) two integers a and b in this complete residue system is easy by adding (multiplying) them as integers and taking the non-negative remainder after division by m. If c is the result we write $c = a + b \bmod n$ ($c = a * b \bmod n$), as was done in Chapter 38 in the section on "The Caesar Cipher and Exhaustive Key Search". Using this addition (multiplication) one can view the ring Z_n as corresponding to the set $\{0, 1, \ldots, n-1\}$. We do not discuss this formally.

THEOREM 39.6 *Let $n \geq 2$. The element $a \in Z_n$ has an inverse modulo n if and only if $gcd(a, n) = 1$.*

PROOF If $\gcd(a, n) = 1$ then (39.2) implies that integers x and y exist such that $xa + yn = 1$, or $xa = 1 \bmod n$. So $x \equiv a^{-1}$.

Now if an inverse $a^{-1} \in Z_n$ exists, then $a \cdot a^{-1} \equiv 1 \bmod n$. Using the definition of congruence this means that $n \mid (a \cdot a^{-1} - 1)$, or that $a \cdot a^{-1} - 1 = yn$, where y is an integer. From (39.3) this implies that $\gcd(a, n) \mid 1$.

COROLLARY 39.3 Z_n^*, the set of elements in Z_n relatively prime to n is an Abelian group for the multiplication.

COROLLARY 39.4 When p is a prime, Z_p is a finite field.

COROLLARY 39.5 If $\gcd(a, n) = 1$ then the equation $ax \equiv b \bmod n$ has exactly one solution modulo n.

PROOF Assume that we had different solutions modulo n. Let us say x_1 and x_2. Then $ax_1 \equiv ax_2 \bmod n$, and since a has an inverse, we obtain that $x_1 \equiv x_2 \bmod n$. This is a contradiction.

Euler–Fermat Theorem

The Euler–Fermat theorem is probably the most important theorem for understanding the RSA cryptosystem (see "RSA"). We first give the following definition.

DEFINITION 39.9 The order of Z_n^* is denoted as $\phi(n)$. The function ϕ is called the Euler-totient function, or Euler function. If one selects one representative out of each residue class co-prime to n, we call the resulting set of integers a *reduced residue system.*

LEMMA 39.2 If $\{a_1, a_2, \ldots, a_{\phi(n)}\}$ is a reduced residue system, and $\gcd(k, n) = 1$, then $\{ka_1, ka_2, \ldots, ka_{\phi(n)}\}$ is a reduced residue system.

PROOF First $\gcd(ka_i, n) = 1$. Secondly, if $i \neq j$, then $ka_i \not\equiv ka_j \mod n$, by contradiction.

This lemma implies the Euler–Fermat Theorem.

THEOREM 39.7 $\forall b \in Z_n^* : b^{\phi(n)} \equiv 1 \mod n$.

PROOF Let $\{a_1, a_2, \ldots, a_{\phi(n)}\}$ be a reduced residue system. Lemma 39.2 implies that

$$\prod_{h=1}^{\phi(n)} (ba_h) \equiv \prod_{h=1}^{\phi(n)} a_h \mod n . \tag{39.5}$$

Since $\gcd(a_h, n) = 1$, a_h^{-1} exists, so

$$\begin{aligned} \prod_{h=1}^{\phi(n)} \left(a_h^{-1} a_h b\right) &\equiv \prod_{h=1}^{\phi(n)} a_h^{-1} a_h \mod n, \qquad \text{implying} \\ b^{\phi(n)} &\equiv 1 \mod n . \end{aligned}$$

COROLLARY 39.6 If $m \in Z_n^*$, then $m^{\phi(n)+a} \equiv m^a \mod n$.

It is easy to see that when p is a prime, $\phi(p) = p - 1$. The next corollary is known as Fermat's little theorem.

COROLLARY 39.7 Let p be a prime. $\forall b \in Z_p : b^p \equiv b \mod p$.

39.3 Encryption Schemes

We will explain some encryption schemes from a mathematical viewpoint without addressing the algorithms needed. These are explained in Section 39.4.

Discrete Log

Given an element $b \in \langle a \rangle$, a cyclic group, we know that a k exists such that $b = a^k$ in this cyclic group. This k is called the **discrete logarithm** of b in the base a and often denoted as $k = \log_a(b)$. To have any value to cryptography it must be hard to find this k. So a proper group needs to be chosen, which we discuss later.

One of the first schemes that was based on discrete logarithm is a key distribution scheme (see Chapter 40). Here we discuss the El Gamal encryption scheme [12].

Generating a Public Key

We assume that a finite group $\langle g \rangle(\cdot)$ has been chosen of a large enough order, and that q, a multiple of the order of the $\text{ord}(g)$, is given (it is sufficient that not too large an upperbound on q is known). Note

that q is not necessarily a prime. For simplicity we assume that q is public. This information could be part of a person's public key. We also assume that the group operation $(\cdot)$ and the inverse of an element can be computed in (expected) polynomial time.

When Alice wants to generate her public key, she chooses a uniform random $a \in Z_q$ and computes $y_A = g^a$ in this group and makes y_A public.

El Gamal Encryption

To encrypt a message $m \in \langle g \rangle$ (otherwise a mode[1] is used), the sender finds the public key y_A of the receiver. The sender chooses[2] a uniformly random $k \in Z_q$ and sends as ciphertext $c = (c_1, c_2) = (g^k, m \cdot y_A^k)$ computed in $\langle g \rangle$.

El Gamal Decryption

To decrypt the ciphertext, the legitimate receiver knowing the secret key a computes $m' = c_2 \cdot (c_1^a)^{-1}$. It is easy to verify that $m' = m \cdot (g^a)^k \cdot (g^k)^{-a} = m$.

Suitable Group

As we already mentioned, to have any cryptographic value the discrete logarithm must be hard. Unfortunately, there is no proof that the discrete logarithm is a hard problem. One can only state that so far no one has found an algorithm running in polynomial time for the general problem. For some groups the problem is a little easier than in the general case. For some the discrete logarithm is even easy. Indeed, for example, for the cyclic group $Z_n(+)$ the discrete logarithm corresponds with finding x such that $ax = b \bmod n$, which is easy as we will discuss in "Gcd and Modulo Inverses" and "Linear Equations."

Groups that are used in practice are the multiplicative group of a finite field (see also "Primitive Elements"), or a subgroup of it and elliptic curve groups.

RSA

RSA, which is a heuristic cryptosystem, is an acronym for the inventors of the scheme, Rivest, Shamir, and Adleman [25]. It is basically an application of the Euler–Fermat Theorem.

Generating a Public Key

To select her public key, Alice chooses two random primes p and q, large enough, and multiplies these to obtain $n = p \cdot q$. She chooses a random element $e \in Z_{\phi(n)}^*$ uniformly and computes $d = e^{-1} \bmod \phi(n)$. She publishes (n, e) as public key and keeps d as a secret key. Note that p, q, and $\phi(n)$ need to remain secret.

Encryption

To encrypt a message the sender finds the public key of the receiver, which we call (n, e). When $m \in Z_n$ (otherwise a mode[1] is used) the resulting ciphertext $c = m^e \bmod n$.

---

[1] If one uses modes discussed for DES, the CFB and OFB modes cannot be used, since the encryption mode as well as the decryption mode use the encryption algorithm.

[2] It is possible that q is secret. For example, when the group $\langle g \rangle$ was selected by the one who constructed the public key. We then assume that a not too large upperbound on q is public and is used instead of q.

Decryption

To decrypt a ciphertext $c \in Z_n$, the legitimate receiver, let us say Alice, knowing her secret key d computes $m' = c^d \bmod n$.

We will now explain why the decryption works for the case $m \in Z_n^*$. This can easily be generalized for all $m \in Z_n$, using the Chinese Remainder Theorem (see "Chinese Remainder Theorem"). Since $e \cdot d = 1 \bmod \phi(n)$ we have $e \cdot d = 1 + k\phi(n)$ for some integer k. So, due to Euler–Fermat Theorem (see Corollary 39.6), when $m \in Z_n^*$ we have that $c^d = m^{ed} = m^{k \cdot \phi(n)+1} = m \bmod n$. So $m' = m$.

Notes

Since RSA is rather slow compared to DES, RSA is often used to send a uniformly random key to the receiver. The message itself is then encrypted using a conventional cryptosystem. RSA is also very popular as a signature scheme (see Chapter 40).

To speed up encryption, it has been suggested to choose $e = 3$, or a small e, or an e such that $w(e)$, the Hamming[3] weight of e, is small. Numerative attacks have been presented against such solutions. To avoid these it seems best to choose e as described.

When m is chosen as a uniformly random element in Z_n, and p and q are large enough, no attack is known for finding m. It has been argued, without proof (see "Euler-Totient Function" for more details), that this is as hard as **factoring** n.

39.4 Computational Number Theory: Part 1

When describing the RSA and the ElGamal public key systems, we silently assumed many efficient algorithms. Indeed for the ElGamal system we need efficient algorithms to

- Select an element of large enough order (when choosing the public key).
- Select a uniformly random element in Z_q (when generating a public key and when encrypting).
- Raise an element in a group to a power (when constructing the public key, when encrypting and decrypting).
- Multiply two elements in the group (when encrypting and decrypting).
- Compute an inverse, for example in Z_p^* (when decrypting).
- Guarantee that $m \in \langle g \rangle$ (when encrypting).

For RSA we need algorithms to

- Select a random prime (when generating a public key).
- Multiply two large integers (when constructing a public key).
- Compute $\phi(n)$, or a multiple of it (when constructing a public key).
- Randomly select an element in $Z_{\phi(n)}^*$ (when constructing a public key).
- Compute an inverse (when constructing a public key).
- Raise an element in a group to a power (when encrypting and decrypting).

We now discuss the necessary algorithms. To avoid repetition we will proceed in a different order than listed above. If necessary we will discuss more number theory before giving the algorithm.

[3] The Hamming weight of a binary string is the number of ones in the string.

Multiplication and Modulo Multiplication

Discussing fast integer multiplication and modulo multiplication is a chapter in itself, and therefore beyond the scope of this chapter. Although algorithms based on FFT (see Chapter 17) are order-wise very fast, the numbers used in modern cryptography are too small to compensate for the rather large constants in FFT based algorithms.

The algorithms used are the trivial ones learned in elementary school. However, usually the base is a power of 2, instead of 10.

Fast Exponentiation

Assume that we have a semigroup $M(*)$ in which an efficient algorithm exists for multiplying two elements. Since squaring an element might be faster [17] than multiplying, we allow for a separate algorithm **square** to square an element.

Input declaration: an element $a \in M$ and b a positive integer.
Output declaration: a^b in $M(*)$

function fastexpo(a, b)
begin
case
$b = 1$ then fastexpo:= a
$b > 1$ and odd then fastexpo:= $a*$**fastexpo**$(a, b-1)$
b is even then fastexpo:=**square**(**fastexpo**$(a,b/2)$)
end

The above function uses, at maximum, $2|b|$ multiplications, where $|b|$ is the binary length of b.

Gcd and Modulo Inverses

Let $a \geq b \geq 0$. The algorithm to compute the greatest common divisor goes back to Euclid, and is therefore called the Euclidean algorithm. It is based on the observation that $\gcd(a, b) = \gcd(a - b, b)$. This trick can be repeated until $0 \leq r = a - mb < b$, which exists (see Theorem 39.2). This gives the modern version of the Euclidean algorithm:

Input declaration: non-negative integers a, b where $a \geq b$
Output declaration: $\gcd(a, b)$

function gcd(a, b)
begin
if $b = 0$ then gcd := a
else gcd :=**gcd**$(b, a \bmod b)$
end

Note: We assume that $a \bmod b$ returns the least nonnegative remainder of a when divided by b. When $a = b = 0$ the function returns 0.

THEOREM 39.8 *The number of divisions required to compute the* $\gcd(a, b)$ *when* $0 < b < a$*, is, at maximum,* $1 + \lfloor \log_R a \rfloor$*, where* $R = (1 + \sqrt{5})/2$.

PROOF We denote $r_{-1} = a, r_0 = b$ and r_i the remainder obtained in the i^{th} division. Note that the $\gcd(a, b)$ will be such a remainder. We define n such that $r_{n-1} = \gcd(a, b)$, implying that $r_n = 0$. We call the i^{th} quotient d_i. So, we have

$$\begin{aligned} d_i &= \left\lfloor \frac{r_{i-2}}{r_{i-1}} \right\rfloor \qquad (1 \leq i \leq n) && (39.6) \\ r_{i-2} &= d_i \cdot r_{i-1} + r_i \qquad (1 \leq i \leq n) && (39.7) \\ r_{n-2} &= d_n \cdot r_{n-1} \, . \end{aligned}$$

Note that $r_i < r_{i-1}$ $(0 \leq i \leq n)$, $r_{n-1} \geq 1 = f_2$ and $r_{n-2} \geq 2 = f_3$, where f_n is the n^{th} Fibonacci number. Using this and the fact that $f_{n+1-i} = f_{n-i} + f_{n-i-1}$ and (39.7), it is easy to prove with inverse induction, starting from $i = n - 1$, that $r_i \geq f_{n+1-i}$. So, $a \geq f_{n+2}$ and $b \geq f_{n+1}$. Since $r_{n-2} \geq 2$, this implies the following. When the Euclidean algorithm takes exactly n divisions to compute $\gcd(a, b)$ we have $a \geq f_{n+2}$ and $b \geq f_{n+1}$. So, if $b' < a' < f_{n+2}$ then there are, at maximum, $n - 1$ divisions when computing the greatest common divisor of a' and b' using the Euclidean algorithm.

Now since $R^2 = R + 1$ we have $R^{n-1} = R^{n-2} + R^{n-3}$. This implies, using induction, that $\forall n \geq 1 : R^{n-2} \leq f_n \leq R^{n-1}$. So, if $0 < b' < a'$ and $R^{n-2} < a' < R^n$ there are, at maximum, $n - 1$ divisions. This corresponds to saying that if $n - 2 < \log_R(a') < n$, or $n - 2 \leq \lfloor \log_R(a') \rfloor < n$ there are, at maximum, $n - 1$ divisions.

An extended version of the Euclidean algorithm allows computing the inverse of b modulo a, where $0 < b < a$.

Algorithm 1

Input: *positive integers* a, b *where* $a > b > 0$

Output: $g = \gcd(a, b)$, x *and* y, *integers such that* $ax + by = \gcd(a, b)$ *and* $c = b^{-1}$ mod$a$ *if it exists (otherwise* $c = 0$*)*

begin
$r_{-1} = a; r_0 = b; x_{-1} := 1; y_{-1} := 0; x_0 := 0; y_0 := 1; i := 0;$
while $r_i \neq 0$ do
    begin
        $i := i + 1;$
        $d := \left\lfloor \frac{r_{i-2}}{r_{i-1}} \right\rfloor;$
        $r_i := r_{i-2} - d * r_{i-1};$
        $x_i := x_{i-2} - d * x_{i-1};$
        $y_i := y_{i-2} - d * y_{i-1};$
    end
$g := r_{i-1}; x := x_{i-1}; y := y_{i-1};$
if $g = 1$ then
        if $i \bmod 2 = 1$ then $c := y$
                else $c := y + a$
    else $c := 0;$
end

Observe that only $d, r_i, r_{i-1}, x_i, x_{i-1}, y_i$, and y_{i-1} are needed. So, a more careful implementation allows saving memory.

LEMMA 39.3 For $i \geq -1$ we have in Algorithm 1 that

$$ax_i + by_i = r_i \ . \tag{39.8}$$

PROOF We use the definition of d_i in (39.6). It is easy to verify that (39.8) holds for $i = -1$ and $i = 0$. For larger values, we use induction and assume that (39.8) has been proven for $i - 1$ and $i - 2$. Observe that $x_i = x_{i-2} - d_i x_{i-1}$ and $y_i = y_{i-2} - d_i y_{i-1}$. So

$$\begin{aligned} ax_i + by_i &= a\left(x_{i-2} - d_i x_{i-1}\right) + b\left(y_{i-2} - d_i y_{i-1}\right) \\ &= \left(ax_{i-2} + by_{i-2}\right) - d_i\left(ax_{i-1} + by_{i-1}\right) \\ &= r_{i-2} - d_i r_{i-1} \end{aligned}$$

using the induction hypothesis. Since $r_i = r_{i-2} - d_i * r_{i-1}$ we have proven the claim.

THEOREM 39.9 *In Algorithm 1* $c = b^{-1} \bmod a$ *if* $b^{-1} \bmod a$ *exists and* $0 \leq c < a$.

PROOF We assume that $\gcd(a, b) = 1$. From the proof of Theorem 39.8 and Lemma 39.3 we have for the x and y returned by Algorithm 1 that $ax + by = \gcd(a, b)$. This implies that $c \equiv b^{-1} \bmod a$. So, we only need to prove that $0 \leq c < a$.

We let n be as in the proof of Theorem 39.8 and d_i be as in (39.6). We first claim that

$$(-1)^i y_i \geq 0 \tag{39.9}$$

when $-1 \leq i \leq n$. This is easy to verify for $i = -1$ and $i = 0$. We now assume that (39.9) is true for $i = k - 1$ and $i = k - 2$. Since $d_i > 0$ when $0 < b < a$ and using the recursive definition of y_i we have $(-1)^k y_k = (-1)^k (y_{k-2} - d_k y_{k-1}) = (-1)^{k-2} y_{k-2} + (-1)^{k-1} d_k y_{k-1} \geq 0$ by the induction hypothesis.

Due to (39.9) we have $|y_i| = |y_{i-2}| + d_i * |y_{i-1}|$. Since $y_0 = 1$ and $d_i \geq 1$, we have

$$|y_1| \geq |y_0| \qquad \text{and} \qquad |y_i| > \left|y_{i-1}\right| \quad \text{for} \quad 2 \leq i \leq n \ , \tag{39.10}$$

as is easy to verify using induction.

Finally we claim that for all i $(0 \leq i \leq n)$:

$$y_{i-1} r_i - y_i r_{i-1} = (-1)^{i+1} a \tag{39.11}$$

$$x_{i-1} r_i - x_i r_{i-1} = (-1)^i b \tag{39.12}$$

For $i = 0$, (39.11) is trivially satisfied. Assume that the equations are satisfied for $i = k - 1$. Noticing that $r_k = r_{k-2} - d_k r_{k-1}$ and $y_k = y_{k-2} - d_k y_{k-1}$, we obtain for $i = k$ that

$$\begin{aligned} y_{k-1} r_k - y_k r_{k-1} &= y_{k-1}\left(r_{k-2} - d_k r_{k-1}\right) - \left(y_{k-2} - d_k y_{k-1}\right) r_{k-1} \\ &= y_{k-1} r_{k-2} - y_{k-2} r_{k-1} \\ &= -\left(y_{k-2} r_{k-1} - y_{k-1} r_{k-2}\right) \ , \end{aligned}$$

which, using the induction hypothesis, proves (39.11). Similarly one can prove (39.12).

Since $r_n = 0$ and $r_{n-1} = \gcd(a, b) = 1$, (39.11) and (39.12) imply that $y_n = (-1)^n a$ and, respectively, $x_n = (-1)^{n+1} b$. So if $n \geq 2$ then, using (39.10), $|y_{n-1}| < a$ and $|y_{n-1}| \neq 0$. If $n = 1$, then by the definition $x_1 = x_{-1} - d_1 * x_0 = 1$ and by (39.12) $x_1 = b$, so $b = 1$ and $y_{n-1} = y_0 = 1 < a$. Thus, $0 < |y_{n-1}| < a$. Using this and (39.9), we obtain that if n is odd $c = y_{n-1} < a$ and $c > 0$, else $-a < y_{n-1} < 0$, so $0 < c = a + y_{n-1} < a$.

Random Selection

We assume that the user has a binary random generator that outputs a string of independent bits with uniform distribution. Using this generator to output one bit is called a coin flip.

We will now describe how to select a natural number a with uniform distribution such that $0 \leq a \leq b$, where $2^{k-1} \leq b < 2^k$. One lets the generator output k bits. We view these bits as a binary representation of an integer x. If $x \leq b$, one outputs $a = x$, else one flips k new bits until $x \leq b$. The expected number of coin flips is bounded above by $2k$.

The case one requires that $b_1 \leq a \leq b_2$ is easy to reduce to the previous one. Indeed, choose a' uniformly such that $0 \leq a' \leq (b_2 - b_1)$ and add b_1 to the result.

Selecting an element in Z_n^* can be done by selecting an integer a such that $0 \leq a \leq n-1$ and by repeating the procedure until $\gcd(a, n) = 1$. The expected number of coin flips is $O(\log\log(n) \cdot \log(n))$, which we do not prove.

Before we discuss how to select a prime we will discuss more number theory. This number theory will also be useful to explain the first probabilistic encryption scheme (see Section 39.7).

39.5 More Algebra and Number Theory

Primes and Factorization

LEMMA 39.4 If p is a prime and $p \mid a \cdot b$, then p divides a or b.

PROOF If $p \not\mid a$, then $\gcd(a, p) = 1$. So, due to Corollary 39.1, integers x, y exists such that $xa + yp = 1$, or $xab + ybp = b$. Since $p \mid ab$, this implies that $p \mid b$.

THEOREM 39.10 *Prime factorization of any integer n is unique, i.e., the primes $p_1, \ldots, p_k$ and the integers $a_i \geq 1$ such that*

$$n = \prod_{i=1}^{k} p_i^{a_i}$$

are unique.

PROOF From Theorem 39.3 we know that n is a product of primes. Assume that

$$\prod_{i=1}^{k} p_i^{a_i} = \prod_{j=1}^{m} q_j^{b_j}\ , \tag{39.13}$$

where q_j are primes and $b_j \geq 1$. First, due to Lemma 39.4 and the fact that p_i and q_j are primes, a p_i is equal to some q_j and vice versa (by induction). So, $k = m$ and when renaming $p_i = q_i$. If $a_i > b_i$ then dividing both sides of (39.13) by $p_i^{b_i}$ we obtain a contradiction due to Lemma 39.4.

COROLLARY 39.8 The least common multiple of a and b, denoted as $\text{lcm}(a, b) = a \cdot b / \gcd(a, b)$.

PROOF The proof is left as an exercise.

Euler-Totient Function

The algorithm used to compute $\phi(n)$ when constructing an RSA public key is trivially based on the following theorem. First, we define what, in number theory, is called a multiplicative function and we also introduce the Möbius function.

DEFINITION 39.10 A real or a complex valued function defined on the positive integers is called an *arithmetical* or a *number-theoretic function.* An arithmetical function f is *multiplicative* if it is not the zero function and if for all positive integers m and n where $\gcd(m, n) = 1$ we have $f(m \cdot n) = f(m) \cdot f(n)$. If the gcd restriction is removed, it is *completely multiplicative*.

DEFINITION 39.11 A square is an integer to the power 2. The *Möbius function* is defined as being

$$\mu(n) \quad = \quad \begin{cases} 0 & \text{if } n \text{ is divisible by a square different from 1,} \\ (-1)^k & \text{if } n = p_1 \cdot p_2 \cdots p_k, \text{ where } p_i \text{ are distinct primes.} \end{cases}$$

A number is squarefree if $\mu(n) \neq 0$.

LEMMA 39.5 If $n \geq 1$ we have

$$\sum_{d|n} \mu(d) = \left\lfloor \frac{1}{n} \right\rfloor \quad = \quad \begin{cases} 1 & \text{if } n = 1, \\ 0 & \text{if } n > 1. \end{cases} \tag{39.14}$$

PROOF When $n = 1$, $\mu(1) = 1$ since it is the product of 0 different primes. We now discuss the case that $n > 1$. Then accordingly to Theorem 39.10 $n = \prod_{i=1}^{k} p_i^{a_i}$, where p_i are distinct primes. In (39.14) only when d is squarefree is $\mu(d) \neq 0$, so

$$\begin{aligned} \sum_{d|n} \mu(d) \quad &= \quad \mu(1) + \mu(p_1) + \cdots + \mu(p_k) + \mu(p_1 \cdot p_2) + \cdots + \mu(p_{k-1} \cdot p_k) \\ &\quad + \cdots + \mu(p_1 \cdot p_2 \cdots p_k) \\ &= \quad 1 + \binom{k}{1} \cdot (-1) + \binom{k}{2} \cdot (-1)^2 + \cdots + \binom{k}{k} (-1)^k = (1-1)^k = 0 \end{aligned}$$

LEMMA 39.6 If $n \geq 1$ then

$$\phi(n) = n \cdot \sum_{d|n} \frac{\mu(d)}{d} .$$

PROOF From the definition of the Euler-totient function, we immediately have

$$\phi(n) = \sum_{k=1}^{n} \left\lfloor \frac{1}{\gcd(k, n)} \right\rfloor .$$

This can be rewritten, using Lemma 39.5, as

$$\phi(n) = \sum_{k=1}^{n} \sum_{d|\gcd(k,n)} \mu(d) = \sum_{k=1}^{n} \sum_{\substack{d|k \\ d|n}} \mu(d) .$$

In the second sum, all d divide n. For such a fixed d, k must be a multiple of d, i.e., $k = md$. Now, $1 \le k \le n$, which is equivalent to the requirement that $1 \le m \le n/d$. Thus,

$$\phi(n) = \sum_{d|n} \sum_{m=1}^{\frac{n}{d}} \mu(d) = \sum_{d|n} \mu(d) \sum_{m=1}^{\frac{n}{d}} 1 = \sum_{d|n} \mu(d) \cdot \frac{n}{d} .$$

THEOREM 39.11 $\phi(n) = n \cdot \prod_{p|n} \left(1 - \frac{1}{p}\right)$, *where p are distinct primes.*

PROOF When $n = 1$, no primes divide n, so the product is equal to 1. We will now consider the case where $n > 1$, so $n = \prod_{i=1}^{m} p_i^{a_i}$, where p_i are distinct primes and $a_i \ge 1$. Obviously

$$\prod_{i=1}^{m} \left(1 - \frac{1}{p_i}\right) = 1 - \sum_{i=1}^{m} \frac{1}{p_i} + \sum_{\substack{i,j \\ i \neq j}} \frac{1}{p_i p_j} - \sum_{\substack{i,j,k \\ i \neq j \\ i \neq k \\ j \neq k}} \frac{1}{p_i p_j p_k} + \cdots + \frac{(-1)^m}{p_1 p_2 \cdots p_m} . \tag{39.15}$$

Each term on the right-hand side corresponds to a $\pm 1/d$, where d is a squarefree divisor of n. For those, the numerator is $\mu(d)$. For the other divisors of n, $\mu(d) = 0$, so the sum in (39.15) is $\sum_{d|n}(\mu(d)/d)$. Using Lemma 39.6, we have proven the theorem.

COROLLARY 39.9 $\phi(n)$ is multiplicative.

This corollary implies that, given the factorization of n, it is easy to compute $\phi(n)$. Assuming the correctness of the Extended Riemann Hypothesis and given $\phi(n)$, one can factor n in polynomial time, which we will not prove.

Linear Equations

THEOREM 39.12 *The equation*

$$ax = b \bmod n \tag{39.16}$$

has no solution if $\gcd(a, n) \not| b$, *else it has* $\gcd(a, n)$ *solutions.*

PROOF If $ax = b \bmod n$, an integer y exists such that $ax + ny = b$. Statement (39.3) in Corollary 39.1 tells us that $\gcd(a, n)$ must divide b. From now on, we assume it does. We will prove that there are then $\gcd(a, n)$ solutions.

Corollary 39.5 tells us that if $\gcd(a, n) = 1$, the solution is unique. Suppose now that $d = \gcd(a, n)$. If $ax = b \bmod n$, then $d \mid n \mid (ax - b)$, which implies that

$$\frac{a}{d}x = \frac{b}{d} \bmod \frac{n}{d} .$$

This has a unique solution x_0 modulo n/d, since $\gcd(a/d, n/d) = 1$. Using the definition of congruence, this implies that x_0 is a solution of (39.16). Another implication is that all other solutions modulo n must have the form

$$x = x_0 + m \cdot \frac{n}{d} \qquad \text{where } 0 \le m \le d - 1 .$$

Polynomials

The following theorem plays an important role in secret sharing (see Chapter 41).

THEOREM 39.13 *Let p be a prime and f a polynomial. The number of solutions (including repeating ones) of $f(x) = 0 \bmod p$ is, at maximum, the degree of f.*

PROOF The theorem is trivially satisfied when the number of solutions is zero. If a_1 is a solution then $f(x) = (x - a_1)q_1(x) + r_1(x)$, where $r_1(x)$ is the remainder, i.e., has degree zero. Since $p \mid f(a_1)$ and $p \mid (a_1 - a_1)q_1(a_1), r_1(x) = 0 \bmod p$. If a_1 is a solution of $q_1(x)$ obtain, $f(x) = (x - a_1)^2 q_1'(x) \bmod p$. In general, an $f_1(x)$ exists such that $f(x) = (x - a_1)^{h_1} f_1(x) \bmod p$ and $f_1(a_1) \not\equiv 0 \bmod p$. Clearly the degree of $f_1(x)$ is $m - h_1$. Suppose that a_2 is another solution modulo p. Then, $f(a_2) = (a_2 - a_1)^{h_1} f_1(a_2) = 0 \bmod p$. Since $a_2 \not\equiv a_1 \bmod p$, Lemma 39.4 implies $p \mid f_1(a_2)$, or, in other words, that $f_1(a_2) \bmod p$. So, using a similar argument as previously, we obtain that $f(x) = (x - a_1)^{h_1}(x - a_2)^{h_2} f_2(x)$ where a_1 nor a_2 are solutions of $f_2(x) = 0$. This gives (using induction) the following:

$$f(x) = \left(\prod_{i=1}^{l} (x - a_i)^{h_i} \right) f_l(x) \,,$$

where $f_l(x) = 0 \bmod p$ has no solutions. This immediately implies the theorem.

When p is composite, the theorem does not extend, as we will briefly discuss in Section "Chinese Remainder Theorem."

Quadratic Residues

DEFINITION 39.12 The set of **quadratic residue modulo** n, denoted as $\mathrm{QR}_n = \{x \mid \exists y \in Z_n^* : x \equiv y^2 \bmod n\}$. The set of **quadratic nonresidues modulo** n is $\mathrm{QNR}_n = Z_n^* \setminus \mathrm{QR}_n$.

THEOREM 39.14 *If p is an odd prime, then $|\mathrm{QR}_p| = |\mathrm{QNR}_p| = (p - 1)/2$.*

PROOF First $x^2 = a \bmod p$ has, at maximum, 2 solutions due to Theorem 39.13. Now, squaring all elements of Z_p^* and noticing that $x^2 \equiv (-x)^2 \bmod p$, we obtain the result.

DEFINITION 39.13 Let p be an odd prime. The *Legendre symbol*

$$\left(\frac{a}{p}\right) = (a \mid p) \quad = \quad \begin{cases} 1 & \text{if } a \in \mathrm{QR}_p, \\ -1 & \text{if } a \in \mathrm{QNR}_p, \\ 0 & \text{if } a \equiv 0 \bmod p \,. \end{cases}$$

We will now discuss Euler's criterion.

THEOREM 39.15 *Let p be an odd prime. $(a \mid p) = a^{(p-1)/2} \bmod p$.*

PROOF The case $a \equiv 0 \bmod p$ is trivial, which we will now exclude. If $(a \mid p) = 1$ then an integer x exists such that $x^2 = a \bmod p$, so $a^{(p-1)/2} = x^{p-1} = 1 \bmod p$ due to Fermat's little theorem

(Corollary 39.7). Since $|\mathrm{QR}_p| = (p-1)/2$ the polynomial equation

$$a^{\frac{p-1}{2}} = 1 \bmod p \tag{39.17}$$

in a has at least $(p-1)/2$ solutions (see Theorem 39.14), and using Theorem 39.13, this implies exactly $(p-1)/2$ solutions. So, if $a^{(p-1)/2} = 1 \bmod p$, then $(a \mid p) = 1$.

Similarly, using Fermat's little theorem, the equation

$$a^{p-1} - 1 = 0 \bmod p \tag{39.18}$$

in a has exactly $p-1$ solutions. Now $a^{p-1} - 1 = (a^{(p-1)/2} - 1)(a^{(p-1)/2} + 1)$. So, the $(p-1)/2$ solutions of (39.18) that are not solutions of (39.17) must be the solutions of the equation $(a^{(p-1)/2} + 1) = 0 \bmod p$. Since $|\mathrm{QNR}_p| = (p-1)/2$, and all solutions of (39.17) are quadratic residues, we have for all $a \in \mathrm{QNR}_p$ that $(a^{(p-1)/2} + 1) = 0 \bmod p$.

COROLLARY 39.10 If p is an odd prime, then $(a \mid p) \cdot (b \mid p) = (a \cdot b \mid p)$.

COROLLARY 39.11 If p is an odd prime, then $(-1 \mid p) = (-1)^{\frac{p-1}{2}}$.

We now discuss Gauss' lemma.

THEOREM 39.16 *Let p be an odd prime, n an integer such that $p \nmid n$, and m the cardinality of the set*

$$A = \left\{ a_i \mid a_i = k \cdot n \bmod p \text{ for } 1 \le k \le (p-1)/2 \text{ and } p/2 < a_i < p \right\} .$$

Then $(n \mid p) = (-1)^m$.

PROOF Let $B = \{b_k \mid b_k = k \cdot n \bmod p \text{ for } 1 \le k \le (p-1)/2\}$. We define $C = \{c_j\} = B \setminus A$, and let $|C| = l$. Observe that for all c_j and a_i we have that $p \neq a_i + c_j$, by contradiction. Indeed, otherwise $p \mid (a_i + c_j) = (k_i + k_j)n$, which is not possible. Therefore,

$$\left(\prod_{j=1}^{l} c_j \right) \cdot \left(\prod_{i=1}^{m} (p - a_i) \right) = \left(\frac{p-1}{2} \right)! \tag{39.19}$$

Now, trivially,

$$\left(\prod_{j=1}^{l} c_j \right) \cdot \left(\prod_{i=1}^{m} a_i \right) \equiv \prod_{k=1}^{\frac{p-1}{2}} (k \cdot n) = \left(\frac{p-1}{2} \right)! \cdot \left(n^{\frac{p-1}{2}} \right) \pmod p . \tag{39.20}$$

$$\equiv (-1)^m \cdot \left(\frac{p-1}{2} \right)! \pmod p \tag{39.21}$$

using (39.19) to obtain the last congruence. So, combining (39.20) and (39.21), we have $n^{(p-1)/2} \equiv (-1)^m \bmod p$, which gives the result using Euler's criterion.

COROLLARY 39.12 If p is an odd prime, then

$$\left(\frac{2}{p} \right) = \begin{cases} 1 & \text{if } p \equiv 1 \bmod 8 \text{ or } p \equiv 7 \bmod 8, \\ -1 & \text{if } p \equiv 3 \bmod 8 \text{ or } p \equiv 5 \bmod 8 . \end{cases}$$

PROOF It is easy to verify that when $n = 2$, the set $A = \{\lfloor (p+3)/4 \rfloor \cdot 2, \ldots, ((p-1)/2) \cdot 2\}$. So, $m = (p-1)/2 - \lfloor p/4 \rfloor$. Let $p = 8a + r$, where $r = 1, 3, 5$, or 7. Then m modulo 2 is, respectively, 0, 1, 1, and 0.

Using Gauss' lemma one can prove the law of quadratic reciprocity. Since the proof is rather long (but not complicated) we refer the reader to the literature.

THEOREM 39.17 *If p and q are odd distinct primes, we have*

$$\left(\frac{p}{q}\right) \cdot \left(\frac{q}{p}\right) = (-1)^{\frac{(p-1)(q-1)}{4}} .$$

Jacobi Symbol

DEFINITION 39.14 Let $n = \prod_{i=1}^{h} p_i$ where p_i are (not necessarily distinct) primes and a an integer. The Jacobi symbol $(a \mid n) = \prod_{i=1}^{h} (a \mid p_i)$. The set $Z_n^{+1} = \{a \in Z_n \mid (a \mid n) = 1\}$.

Since $1 = p^0$, we have that $(a \mid 1) = 1$. Also, if n is a prime, it is obvious that the Jacobi symbol is the same as the Legendre symbol. We will discuss why there is no need to factor n to compute the Jacobi symbol. The following theorems are useful in this respect.

THEOREM 39.18 *The Jacobi symbol has the following properties:*

1. *If $a \equiv b \bmod n$, then $(a \mid n) = (b \mid n)$.*
2. $(a \mid n) \cdot (a \mid m) = (a \mid n \cdot m)$.
3. $(a \mid n) \cdot (b \mid n) = (a \cdot b \mid n)$.

PROOF These follow immediately from the definition and Corollary 39.10.

THEOREM 39.19 *When n is an odd positive integer we have $(-1 \mid n) = (-1)^{(n-1)/2}$.*

PROOF When a and b are odd, then trivially

$$(a-1)/2 + (b-1)/2 \equiv (ab-1)/2 \bmod 2 . \tag{39.22}$$

This allows one to prove by induction that

$$\sum_{i=1}^{h} \frac{p_i - 1}{2} \equiv \frac{\left(\prod_{i=1}^{h} p_i\right) - 1}{2} \bmod 2 . \tag{39.23}$$

The rest follows from Corollary 39.11.

THEOREM 39.20 *If n is an odd positive integer, then*

$$\left(\frac{2}{n}\right) = \begin{cases} 1 & \text{if } n \equiv 1 \bmod 8 \text{ or } n \equiv 7 \bmod 8, \\ -1 & \text{if } n \equiv 3 \bmod 8 \text{ or } n \equiv 5 \bmod 8 . \end{cases}$$

PROOF It is easy to verify that Corollary 39.12 can be rewritten as $(2 \mid p) = (-1)^{(p^2-1)/8}$. The rest of the proof is similar to that of the preceding theorem replacing (39.22) by

$$\left(a^2-1\right)/8 + \left(b^2-1\right)/8 \equiv \left(a^2b^2-1\right)/8 \bmod 2\,,$$

where a and b are odd integers. Note that $k(k+1)$ is always even.

THEOREM 39.21 *If m and n are odd positive integers and* $\gcd(m,n) = 1$, *then*

$$\left(\frac{m}{n}\right)\cdot\left(\frac{n}{m}\right) = (-1)^{\frac{(m-1)(n-1)}{4}}\,.$$

PROOF Let $m = \prod_{i=1}^{k} p_i$ and $n = \prod_{j=1}^{l} q_j$, where p_i and q_j are primes. From Theorem 39.18, we have

$$\begin{aligned}\left(\frac{m}{n}\right)\cdot\left(\frac{n}{m}\right) &= \left(\prod_{i=1}^{k}\prod_{j=1}^{l}\left(\frac{p_i}{q_j}\right)\right)\cdot\left(\prod_{i=1}^{k}\prod_{j=1}^{l}\left(\frac{q_j}{p_i}\right)\right) = \left(\prod_{i=1}^{k}\prod_{j=1}^{l}\left(\frac{p_i}{q_j}\right)\cdot\left(\frac{q_j}{p_i}\right)\right)\\ &= \prod_{i=1}^{k}\prod_{j=1}^{l}(-1)^{\frac{(p_i-1)(q_j-1)}{4}} = (-1)^{\frac{(m-1)(n-1)}{4}}\end{aligned}$$

using Theorem 39.17 to obtain the second to last equation and (39.23) to obtain the last.

One could wonder whether $(a \mid n) = 1$ implies that $a \in \mathrm{QR}_n$. Before disproving this, we discuss the Chinese Remainder Theorem.

Chinese Remainder Theorem

THEOREM 39.22 *If* $\gcd(n_1, n_2) = 1$, *then the system of equations*

$$x \equiv a_1 \bmod n_1 \tag{39.24}$$

$$x \equiv a_2 \bmod n_2 \tag{39.25}$$

has exactly one solution modulo $n_1 \cdot n_2$.

PROOF Due to (39.24) and the definition of modulo computation, x must have the form $x = a_1 + n_1 \cdot y$ for some y. Using (39.25) $a_1 + n_1 \cdot y \equiv a_2 \bmod n_2$, which has exactly one solution in y modulo n_2. This follows from Corollary 39.5 since $\gcd(n_1, n_2) = 1$. So, $n_1 y$ has only one solution modulo $n_1 \cdot n_2$, or $x = a_1 + n_1 \cdot y$ is unique modulo $n_1 \cdot n_2$.

As an application we consider the equation $x^2 = a \bmod n$, where n is the product of two different primes, p and q. Since $p \mid n \mid (x^2 - a)$, a solution modulo n is also a solution modulo p (and q respectively). The Chinese Remainder Theorem tells us that it is sufficient to consider the solutions of $x^2 = a \bmod p$ and $x^2 = a \bmod q$, which we now discuss. If $(a \mid p) = 1$ then there are two solutions modulo p, when $(a \mid p) = -1$ there are no solutions modulo p, and finally when $(a \mid p) = 0$, there is one solution modulo p. So $a \in \mathrm{QR}_n$ only if $(a \mid p) = 1$ and $(a \mid q) = 1$, implying that $(a \mid n) = 1$. However, the converse is obviously not true, so $(a \mid n) = 1$ does not necessarily imply that $a \in QR_n$.

Order of an Element

We describe some general properties of an order of an element.

LEMMA 39.7 Let $\langle\alpha\rangle(\cdot)$ be a cyclic group of order l, an integer. If $\alpha^m = 1$, then $l \mid m$.

PROOF From Theorem 39.1 we have that l is the smallest positive integer for which $\alpha^l = 1$. Assume that $l \nmid m$. Then $m = ql + r$, where $0 < r < l$. So, $1 = \alpha^m = (\alpha^l)^q \cdot \alpha^r = 1 \cdot \alpha^r$. So, r is a smaller positive integer for which $\alpha^r = 1$. So, we have a contradiction.

LEMMA 39.8 Let $K(\cdot)$ be an Abelian group and $\alpha, \beta \in K$ with $k = \text{ord}(\alpha)$ and $l = \text{ord}(\beta)$. If $\gcd(k, l) = 1$, then $\text{ord}(\alpha \cdot \beta) = \text{ord}(\alpha) \cdot \text{ord}(\beta)$.

PROOF Let us call m the order of $\alpha \cdot \beta$. Since $(\alpha\beta)^m = 1$, we have $\alpha^m = \beta^{-m}$. This implies that $\alpha^{lm} = \beta^{-lm} = (\beta^l)^{-m} = 1$ and $\beta^{-km} = (\alpha^k)^m = 1$. Using Lemma 39.7, we obtain that $k \mid lm$ and respectively $l \mid km$. Since $\gcd(k, l) = 1$, this implies that $k \mid m$ and $l \mid m$. So, $kl \mid m$. Now, trivially, $(\alpha\beta)^{kl} = 1$. Using Lemma 39.7, this implies that kl must be the order of $\alpha\beta$.

THEOREM 39.23 *Let $K(\cdot)$ be an Abelian group and let*

$$m = \max_{\alpha \in K}(\text{ord}(\alpha)) .$$

We have that $\forall \beta \in K : \text{ord}(\beta) \mid m$.

PROOF We prove this by contradiction. When $\beta = 1$, the identity, the result is trivial. We now assume that $\beta \neq 1$. Let $d = \text{ord}(\beta)$ and suppose that $d \nmid m$. Then, a prime p and an integer $e \geq 1$ exit such that $p^e \mid d$, $p^e \nmid m$, but $p^{e-1} \mid m$. It is easy to verify that when $\text{ord}(\alpha) = m$ we have that $\text{ord}(\alpha^{p^{e-1}}) = m/(p^{e-1})$ and that $\text{ord}(\beta^{d/p^e}) = p^e$. Since, $\gcd(m/(p^{e-1}), p^e) = 1$, Lemma 39.8 implies that $\text{ord}(\alpha^{p^{e-1}} \cdot \beta^{d/p^e}) = m \cdot p$, which is larger than m. So, we obtain a contradiction.

DEFINITION 39.15 The *exponent,* $\exp(K)$, of a finite group K is the smallest positive integer such that $\forall \beta \in K : \beta^{\exp(K)} = 1$.

COROLLARY 39.13 When K is a finite Abelian group then $\exp(K) = \max_{\alpha \in K}(\text{ord}(\alpha))$.

The following theorem is used to prove that Z_p^*, where p is prime, is a cyclic group.

THEOREM 39.24 *Let K be a finite Abelian group. K is cyclic if and only if* $\exp(K) = |K|$.

PROOF First, if $K = \langle\alpha\rangle$, then $|K| = \text{ord}(\alpha)$, so $\exp(K) = |K|$. Secondly, let K be a finite Abelian group such that $\exp(K) = |K|$. By Corollary 39.13 an element α exists such that $\exp(K) = \text{ord}(\alpha)$. Since $\exp(K) = |K|$, we have that $|K| = \text{ord}(\alpha) = |\langle\alpha\rangle|$, implying that K is cyclic.

Primitive Elements

Before proving that Z_p^* (p is prime) is a cyclic group, we will define a **primitive element.**

DEFINITION 39.16 If the order of α in the group Z_n^* is $\phi(n)$, then we say that α is a primitive element of Z_n^*, or a primitive root mod$n$, or a generator of Z_n^*.

So, when Z_n^* has a primitive element, the group is cyclic.

THEOREM 39.25 *If p is prime, Z_p^* is cyclic.*

PROOF Due to Fermat's little theorem (Corollary 39.7) we have that $\forall a \in Z_p^* : a^{|Z_p^*|} = 1$. So $\exp(Z_p^*) \leq |Z_p^*|$. By the definition of the exponent of a group, all elements of Z_p^* satisfy the equation $x^{\exp(Z_p^*)} - 1 = 0$. Now by Theorem 39.13, this equation has at most $\exp(Z_p^*)$ solutions in Z_p, therefore at most $\exp(Z_p^*)$ solutions in Z_p^*. So, $|Z_p^*| \leq \exp(Z_p^*)$. Since these results imply $|Z_p^*| = \exp(Z_p^*)$ and using Theorem 39.24 the theorem is proven.

To prove that Z_{p^e} is cyclic when p is an odd prime, we use the following lemma.

LEMMA 39.9 When p is an odd prime, a primitive root $g \mod p$ exists such that for all integers $e > 1$:

$$g^{\phi\left(p^{e-1}\right)} \not\equiv 1 \mod p^e \,. \tag{39.26}$$

PROOF We will start with the case where $e = 2$. If g is a primitive root mod p which satisfies (39.26), then there is nothing to prove. Else, when $g^{\phi(p^{e-1})} = 1 \mod p^2$, we choose $g_0 = g + p$ (modulo p) as a primitive element. Now, using Theorem 39.11,

$$(g_0)^{\phi\left(p^{e-1}\right)} \equiv (g+p)^{p-1} \equiv \sum_{i=0}^{p-1} \binom{p-1}{i} p^i g^{p-1-i} \equiv 1 + (p-1)pg^{p-2} + 0 \not\equiv 1 \pmod{p^2} \,,$$

satisfying (39.26). So we assume from now on that g satisfies (39.26) when $e = 2$.

The case $e > 2$ is proven by induction. We will assume that (39.26), up to $e \geq 2$, has been proven. Due to the Euler–Fermat Theorem $g^{\phi(p^{e-1})} \equiv 1 \mod p^{e-1}$, or

$$g^{\phi\left(p^{e-1}\right)} = 1 + lp^{e-1} \,,$$

for an integer l. Due to our assumption, $p \not| \, l$. Since $e \geq 2$, $\phi(p^e) = p\phi(p^{e-1})$. So,

$$\begin{aligned} g^{\phi\left(p^{(e-1)+1}\right)} &= (1 + lp^{e-1})^p = \sum_{i=0}^{p} \binom{p}{i} l^i p^{i(e-1)} \\ &= 1 + plp^{e-1} + \frac{p(p-1)}{2} l^2 p^{2(e-1)} + rp^{3(e-1)} \,, \end{aligned} \tag{39.27}$$

for some integer r. Since p is odd, we have that $2 \mid (p-1)$. This implies $p^{e+1} \mid p(p-1)p^{2(e-1)}/2$, since $e + 1 \leq 2e - 1$ when $e \geq 2$. Also, $e + 1 \leq 3e - 3$ when $e \geq 2$. So, modulo p^{e+1} the equation (39.27) becomes

$$g^{\phi(p^e)} \equiv 1 + lp^e \not\equiv 1 \mod p^{e+1} \,.$$

THEOREM 39.26 *When p is an odd prime and $e \geq 1$ is an integer, Z_{p^e} is cyclic.*

PROOF The case $e = 1$ was proven in Theorem 39.25. When $e \geq 2$, we consider a g satisfying the conditions in Lemma 39.9 and call k the order of g modulo p^e. Since g is a primitive root modulo p, $p - 1 \mid k$, so $k = m(p-1)$. From Euler–Fermat Theorem $k \mid \phi(p^e)$, implying that $m \mid p^{e-1}$, or $m = p^s$. So, $k = (p-1)p^s$. Now, $s = e - 1$, otherwise we have a contradiction with Lemma 39.9.

DEFINITION 39.17 The Carmichael function $\lambda(n) = \exp(Z_n^*)$.

THEOREM 39.27 *We have that*

$$\lambda(2^k) \quad = \quad \begin{cases} 2^{k-1} & \text{if } k < 3 \\ 2^{k-2} & \text{if } k \geq 3\,. \end{cases}$$

PROOF It is easy to verify that 1 and 3 are primitive roots modulo 2 and 4, respectively. When $k \geq 3$, we prove by induction that

$$\forall a \in Z_{2^k}^* : \quad a^{2^{k-2}} = 1 \bmod 2^k\,. \tag{39.28}$$

First, if $k = 3$, then $2^{k-2} = 2$. Now all a are odd, and $a^2 = (2l+1)^2 = 4l(l+1) + 1$, for some l. Since $l(l+1)$ is even, $a^2 = 1 \bmod 8$. We will now assume that (39.28) is valid for k. This implies that for all odd integers $a^{2^{k-2}} = 1 + q2^k$ and squaring both sides gives $a^{2^{k-1}} = 1 + q2^{k+1} + q^2 2^{2k} \equiv 1 \bmod 2^{k+1}$. So when $k \geq 3$ and $a \in Z_{2^k}$ then $\text{ord}(a) \mid 2^{k-2}$. We now need to prove that an $a \in Z_{2^k}^*$ exists for which $\text{ord}(a) = 2^{k-2}$. We take $a = 3$ and need to prove that $3^{2^{k-3}} \not\equiv 1 \bmod 2^k$. Using (39.28) instead of the Euler–Fermat Theorem, the last part of the proof of Lemma 39.9 can easily be adapted to prove the claim, and this is left as an exercise.

The Chinese Remainder Theorem implies the following corollary.

COROLLARY 39.14 If $n = 2^{a_0} p_1^{a_1} \cdots p_k^{a_k}$, where p_i are different odd primes, then

$$\lambda(n) = \text{lcm}\left(\lambda\left(2^{a_0}\right), \phi\left(p_1^{a_1}\right), \ldots, \phi\left(p_k^{a_k}\right)\right)\,.$$

COROLLARY 39.15 When n is the product of two different odd primes, as in RSA, Z_n^* is *not* cyclic.

Lagrange Theorem

The following theorem is well known in elementary algebra.

THEOREM 39.28 *The order of a subgroup H of a finite group G is a factor of the order of G.*

PROOF Define the left coset of $x \in G$ relative to the subgroup H as $Hx = \{hx \mid h \in H\}$. First, any two cosets, let us say Hx and Hy, have the same cardinality. Indeed, the map mapping $a \in Hx$ to $ax^{-1}y$ is a bijection. Secondly, these cosets partition G. Obviously, a coset is not empty and any element $a \in G$ belongs to the coset Ha. Suppose now that b belongs to two different cosets Hx and Hy. Then, $b = h_1 x$, $h_1 \in H$ and $b = h_2 y$, where $h_2 \in H$. But then $y = h_2^{-1} h_1 x$, so $y \in Hx$. Then any element $z \in Hy$ will also belong to Hx, since $z = hy$ for some $h \in H$. So, we have a contradiction. Since each coset has the same cardinality and they form a partition, the result follows immediately.

39.6 Computational Number Theory: Part 2

Computing the Jacobi Symbol

Theorems 39.18 through 39.21 can easily be used to adapt the Euclidean algorithm to compute the Jacobi symbol.

Input declaration: integers a, n, where $0 \leq a < n$
Output declaration: $(a \mid n)$

function Jacobi(a, n)
begin

if $n = 1$
  then Jacobi
  else if $a = 0$
    then Jacobi:= 0
    else if $a = 1$
      then Jacobi:= 1
      else if a is even
        then if ($n \equiv 3 \bmod 8$) or ($n \equiv 5 \bmod 8$)
          then Jacobi:= $-$**Jacobi**$(a/2 \bmod n, n)$
          else Jacobi:= **Jacobi**$(a/2 \bmod n, n)$
        else if n is even
          then Jacobi:=**Jacobi**$(a \bmod 2, 2)\cdot$ **Jacobi**$(a \bmod n/2, n/2)$
          else if ($a \equiv 3 \bmod 4$) and ($n \equiv 3 \bmod 4$)
            then Jacobi:= $-$**Jacobi**$(n \bmod a, a)$
            else Jacobi:=**Jacobi**$(n \bmod a, a)$
end

Note that if $a \geq n$ that $(a \mid n) = (a \bmod n \mid n)$. The proof of correctness, the analysis of the algorithm, and a nonrecursive version are left as exercises.

Selecting a Prime

The methods used to select a (uniform random) prime of a certain length consist in choosing a (uniformly random) positive integer of a certain length and then testing whether the number is a prime. There are several methods to test whether a number is a prime. Since the research on the topic is so extensive, it is worth a book in itself. We will therefore limit our discussion.

Primality testing belongs to $\mathbf{NP} \cap \mathbf{co} - \mathbf{NP}$ as proven by Pratt [23]. It is not known whether the problem is in **P**. A random (Las Vegas) polynomial time algorithm has been presented [1], which is unfortunately not very practical. In most applications in cryptography, it is sufficient to know that a number is "likely" a prime. We distinguish two types of primality tests, depending on who chooses the prime. We will first discuss the case where the user of the prime chooses it.

Fermat Pseudoprimes

We will start by discussing primality tests where the user chooses a uniformly random number of a certain length and tests for primality. In this case a Fermat **pseudoprime** test is sufficient. The contrapositive of Fermat's little theorem (see Corollary 39.7) tells us that if a number is composite a

witness $a \not\equiv 0 \bmod n$ exists such that $a^{n-1} \not\equiv 1 \bmod n$, then n is composite. A number, n is called a Fermat pseudoprime to the base a if $a^{n-1} \not\equiv 1 \bmod n$. Although all primes are Fermat pseudoprimes, unfortunately not all pseudoprimes are necessarily primes. The Carmichael numbers are composite numbers but have the property that for $\underline{\text{all}}\ a \in Z_n^* : a^{n-1} = 1 \bmod n$. In other words, that $\lambda(n) \mid n-1$. It is easy to verify that $n = 3 * 11 * 17$ satisfies this. Alford, Granville and Pomerance [3] recently showed that there are infinitely many Carmichael numbers. We will use the following result.

THEOREM 39.29 *If n is a Carmichael number, then n is odd and squarefree.*

PROOF 2 is not a Carmichael number and if $n > 2$, then $2 \mid \lambda(n)$. Since n is a Carmichael number, $\lambda(n) \mid n-1$, so n is odd. Suppose now that $p^2 \mid n$, where p is a prime. Since $p \neq 2$, using Corollary 39.14, we have that $p \mid \lambda(n)$. This implies that $p \mid n-1$, since $\lambda(n) \mid n-1$. Now, $p \mid n$ and $p \mid n-1$, implying that $p \mid 1$, which is a contradiction.

In the case where an odd number n is chosen uniformly random among those of a given length and one uses Fermat's primality test with $a = 2$, the probability that the obtained number is not a prime is sufficiently small for cryptographic applications. However, if the number n is not chosen by the user and is given by an outsider, Fermat's test makes no sense due to the existence of Carmichael numbers, which the outsider could choose. We will now discuss how to check whether a number given by an outsider is "likely" to be a prime.

Probabilistic Primality Tests

We will now describe the Solovay–Strassen algorithm [26] in which the probability of receiving an incorrect statement that a composite n is prime can be made sufficiently small for each number n. The algorithm was also independently discovered by D. H. Lehmer.

We will assume that using a function call to **Random**, with input n, outputs a natural number a with a uniform distribution such that $1 \leq a < n$.

Input declaration: an odd integer $n > 2$.
Output declaration: element of {likely-prime, composite}

function Solovay-Strassen(n)
begin
a :=**Random**(n); (so, $1 \leq a \leq n-1$)
if gcd$(a, n) \neq 1$ then Solovay-Strassen:=composite
else if $(a \mid n) \not\equiv a^{\frac{n-1}{2}} \bmod n$ then Solovay-Strassen:=composite
else Solovay-Strassen:=likely-prime
end

To discuss how good this algorithm is, we first introduce the set

$$E(n) = \{a \in Z_n^* \mid (a \mid n) \equiv a^{\frac{n-1}{2}} \bmod n\}\ .$$

LEMMA 39.10 Let $n \geq 3$ be an odd integer. We have that n is prime if and only if $E(n) = Z_n^*$.

PROOF If n is an odd prime, the claim follows directly from Euler's criterion (Theorem 39.15).

We prove the converse using a contradiction. So, we assume that n is composite and $E(n) = Z_n^*$, which implies (by squaring) that $\forall a \in Z_n^* : a^{n-1} \equiv 1 \bmod n$. Thus n is a Carmichael number, implying that n is squarefree (see Theorem 39.29). So, $n = pr$, where p is a prime and $\gcd(p, r) = 1$. Let b be a quadratic nonresidue modulo p and $a \equiv b \bmod p$ and $a \equiv 1 \bmod r$. Now, from Theorem 39.18 $(a \mid n) = (a \mid p)(a \mid r) = (b \mid p) = -1$. Thus, due to our assumption $a^{(n-1)/2} = -1 \bmod n$, implying that $a^{(n-1)/2} = -1 \bmod r$, since $r \mid n \mid (a^{(n-1)/2} + 1)$. This contradicts with the choice of $a \equiv 1 \bmod r$.

THEOREM 39.30 *If n is an odd prime, then Solovay–Strassen(n) returns likely-prime. If n is an odd composite, then Solovay–Strassen(n) returns likely-prime with a probability less or equal to* $1/2$.

PROOF The first part follows from Corollary 39.4 and Theorem 39.15. To prove the second part, we use Lemma 39.10. It implies that $E(n) \neq Z_n^*$. So, since $E(n)$ is a subgroup of Z_n^*, as is easy to verify, $E(n)$ it is a proper subgroup. Applying Lagrange's theorem (see Theorem 39.28) implies that $|E(n)| \leq |Z_n^*|/2 = \phi(n)/2 \leq (n-1)/2$, which implies the theorem.

It is easy to verify that the protocol runs in expected polynomial time. If, when running the Solovay–Strassen algorithm k times for a given integer n, it returns each time likely-prime, then the probability that n is composite is, at maximum, 2^{-k}.

It should be observed that the Miller–Rabin primality test algorithm has an even smaller probability of making a mistake. Recent research has produced algorithms that are better in this respect.

Selecting an Element with a Larger Order

We will discuss how to select an element of large order in Z_p^*, p a prime. Based on the Chinese Remainder Theorem, Pohlig and Hellman [21] proved that if $p-1$ has only small prime factors, it is easy to compute the discrete logarithm in Z_p^*. Even if $p-1$ has one large prime factor, uniformly picking a random element will not necessarily imply that it has (with high probability) a high order. Therefore, we will discuss how one can generate a primitive element when the prime factorization of $p-1$ is given. The algorithm follows immediately from the following theorem.

THEOREM 39.31 *If p is a prime then an integer a exists such that*

$$\text{for all primes } q \mid p-1 \text{ we have } a^{\frac{p-1}{q}} \not\equiv 1 \bmod p\,. \tag{39.29}$$

Such an element a is a primitive root modulo p.

PROOF A primitive element must satisfy (39.29) and exists due to Theorem 39.25. We now prove, using a contradiction, that an element satisfying (39.29) must be a primitive element. Let $l = \text{ord}(a)$ modulo p. First Lemma 39.7 implies that $l \mid p-1$. So, when $p-1 = \prod_{i=1}^{m} q_i^{b_i}$, where q_i are different primes and $b_i \geq 1$, then $l = \prod_{i=1}^{m} q_i^{c_i}$, where $c_i \leq b_i$. Suppose $c_j < b_j$, then $a^{(p-1)/q_j} \equiv 1 \bmod p$, which is a contradiction.

As in the Solovay–Strassen algorithm, we will use a random generator.

Input declaration: a prime p and the prime factorization of $p-1 = \prod_{i=1}^{m} q_i^{b_i}$
Output declaration: a primitive root modulo p

function generator(p)
begin
repeat
    generator:=**Random**(p); (so, $1 \leq$ generator $\leq p-1$)
until for all q_i (generator)$^{(p-1)/q_i} \neq 1 \bmod p$
end

The expected running time follows from the following theorem.

THEOREM 39.32 *Z_p^*, p a prime has $\phi(p-1)$ different generators.*

PROOF Theorem 39.25 says that there is at least one primitive root. Assume that g is a primitive root modulo p and let $a \in Z_{p-1}^*$. Then an element $b = a^{-1} \bmod p-1$ exists, so $ab = k(p-1)+1$. We claim that $h = g^a \bmod p$ is a primitive root. Indeed, $h^b = g^{ab} = g^{k(p-1)+1} = g \bmod p$. Therefore any element $g^i = h^{bi} \bmod p$. So, we have proven that there are at least $\phi(p-1)$ primitive roots. Now if $d = \gcd(a, n) \neq 1$, then $h = g^a \bmod p$ cannot be a generator. Indeed, if h is generator, then $h^x = 1 \bmod p$ implies that $x \equiv 0 \bmod p-1$. Now, $h^x = g^{ax} = 1 \bmod p$, or $ax = 0 \bmod p-1$, which has only $\gcd(a, p-1)$ incongruent solutions modulo $p-1$ (see Theorem 39.12).

This algorithm can easily be adapted to construct an element of a given order. This is important in the DSS signature scheme. A variant of this algorithm allows one to prove that p is prime.

Other Algorithms

We have only discussed computational algorithms needed to explain the encryption schemes explained in this chapter. However, so much research has been done in the area of computational number theory, that a series of books are necessary to give a decent, up-to-date description of the area. For example, we did not discuss an algorithm to compute square roots, nor did we discuss algorithms used to cryptanalyze cryptosystems, such as algorithms to factor integers and to compute discrete logarithms. It is the progress on these algorithms and their implementations that dictates the size of the numbers one needs to choose to obtain a decent security.

39.7 Probabilistic Encryption

Although the Goldwasser–Micali probabilistic encryption is not very practical, its historical contribution is too significant to be ignored.

Generating a Public Key

When Alice wants to generate her public key, she first selects two primes p and q. She computes $n = p \cdot q$. Then she repeats choosing a random element y until $(y \mid n) = 1$ and $y \in \text{QNR}_n$. Knowing the prime factorization of n, the generation of such a y will take expected polynomial time. She then publishes (n, y) as her public key.

Note, that if $p \equiv q \equiv 3 \bmod 4$, $y = -1$ is such a number since $(-1 \mid p) = (-1 \mid q) = -1$, which is due to Corollary 39.11.

Encryption

To encrypt a bit b the sender uniformly chooses an $r \in Z_n^*$ and sends $c = y^b r^2 \bmod n$ as ciphertext.

Decryption

To decrypt, Alice sets $b = 0$ if $(c \mid p) = 1$. If, however, $(c \mid p) = -1$, then $b = 1$. Due to Corollary 39.10, the decryption is correct.

Let n be a composite, for example, the product of two primes of equal binary length. One can prove that the scheme is secure against a ciphertext-only and known-plaintext attack if it is hard to decide whether a number $a \in Z_n^{+1}$ is a quadratic residue or not.

39.8 Research Issues and Summary

This chapter surveyed some modern public key encryption schemes and the number theory and computational number theory required to understand these schemes. It has been argued, without a proof, that these encryption schemes are based on the discrete logarithm problem and the integer factorization problem. Worse, the computational difficulty of these problems is only assumed hard and it remains an open problem to prove or disprove this.

The problem of making new public key encryption schemes that are secure and significantly faster than these surveyed is an important issue. Many researchers have addressed this and have usually been unsuccessful.

39.9 Defining Terms

Composite: An integer with at least three different positive divisors.

Congruent: Two numbers a and b are congruent modulo c if they have the same nonnegative remainder after division by c.

Cyclic group: A group that can be generated by one element, i.e., all elements are powers of that one element.

Discrete logarithm problem: Given α and β in a group, the discrete logarithm decision problem is to decide whether β is a power of α in that group. The discrete logarithm search problem is to find what this power is (if it exists).

Integer factorization problem: To find a nontrivial divisor of a given integer.

Order of an element: The smallest positive integer such that the element raised to this integer is the identity element.

Prime: An integer with exactly two different positive divisors, namely 1 and itself.

Primitive element: An element that generates all the elements of the group, in particular of the group of integers modulo a given prime.

Pseudoprime: An integer, not necessarily prime, that passes some test.

Quadratic nonresidue modulo n: An integer relatively prime to n that is *not* congruent to a square modulo n.

Quadratic residue modulo n: An integer relatively prime to n that is congruent to a square modulo n.

References

[1] Adleman, L.M. and Huang, M.-D.A., Recognizing primes in random polynomial time. In *Proceedings of the Nineteenth Annual ACM Symp. Theory of Computing, STOC,* 462–469, May 25–27, 1987.

[2] Aho, A.V., Hopcroft, J.E., and Ullman, J.D., *The Design and Analysis of Computer Algorithms,* Addison-Wesley, Reading, MA, 1974.

[3] Alford, W.R., Granville, A., and Pomerance, C., There are infinitely many Carmichael numbers, *Annals of Mathematics,* 140, 703–722, 1994.

[4] Apostol, T.M., *Introduction to Analytic Number Theory,* Springer, New York, 1976.

[5] Bach, E., How to generate factored random numbers, *SIAM J. Comput.,* 17(2), 179–193, Apr. 1988.

[6] Bach, E. and Shallit, J., *Algorithmic number theory,* Vol. 1, Efficient Algorithms of *Foundation of Computing Series,* MIT Press, New York, 1996.

[7] Beauchemin, P., Brassard, G., Crépeau, C., Goutier, C., and Pomerance, C., The generation of random numbers which are probably prime, *Journal of Cryptology,* 1(1), 53–64, 1988.

[8] Berlekamp, E.R., *Algebraic Coding Theory,* McGraw-Hill, 1968.

[9] Berlekamp, E.R., *Algebraic Coding Theory,* Aegen Park Press, 1984.

[10] Brassard, G. and Bratley, P., *Algorithmics—Theory & Practice,* Prentice Hall, 1988.

[11] Cassels, J.W.S., *An Introduction to the Geometry of Numbers,* Springer-Verlag, New York, 1971.

[12] ElGamal, T., A public key cryptosystem and a signature scheme based on discrete logarithms, *IEEE Trans. Inform. Theory,* 31, 469–472, 1985.

[13] Hardy, G. and Wright, E., *An Introduction to the Theory of Numbers,* 5th ed., Oxford Science Publications, London, 1985.

[14] Hua. *Introduction to Number Theory,* Springer, New York, 1982.

[15] Hungerford, T., *Algebra,* 5th ed., Springer-Verlag, New York, 1989.

[16] Jacobson, N., *Basic Algebra I,* W. H. Freeman, New York, 1985.

[17] Knuth, D.E., *The Art of Computer Programming,* Vol. 2, *Seminumerical Algorithms,* Addison-Wesley, Reading, MA, 1981.

[18] LeVeque, W., *Fundamentals of Number Theory,* Addison-Wesley, New York, 1977.

[19] Miller, G.L., Riemann's hypothesis and tests for primality, *J. Computer Systems Sci.,* 13, 300–317, 1976.

[20] Peralta, R., A simple and fast probabilistic algorithm for computing square roots modulo a prime number, *IEEE Transaction on Information Theory,* IT-32(6), 846–847, 1986.

[21] Pohlig, S.C. and Hellman, M.E., An improved algorithm for computing logarithms over $GF(p)$ and its cryptographic significance, *IEEE Trans. Inform. Theory,* IT-24(1), 106–110, Jan. 1978.

[22] Pomerance, C., The quadratic sieve factoring algorithm. In *Advances in Cryptology. Proc. of Eurocrypt 84 Paris, France, April 1984, (Lecture Notes in Computer Science 209),* Beth, T., Cot, N., and Ingemarsson, I., Eds., 169–182. Springer-Verlag, Berlin, 1985.

[23] Pratt, V.R., Every prime has a succinct certificate. *SIAM Journal on Computing,* 4(3), 214–220, 1975.

[24] Rabin, M., Probabilistic algorithm for primality testing, *Journal of Number Theory,* 12, 128–138, 1980.

[25] Rivest, R.L., Shamir, A., and Adleman, L., A method for obtaining digital signatures and public key cryptosystems, *Commun. ACM,* 21, 294–299, Apr. 1978.

[26] Solovay, R. and Strassen, V., A fast Monte-Carlo test for primality, *SIAM Journal on Computing,* 6(1), 84–85, erratum (1978), ibid., 7,118, 1977.

Further Information

The *Further Information* section in the previous chapter discusses further information on public key encryption schemes.

The number theory required to understand modern public key systems can be found in many books on number theory such as [4, 13, 14]. These books discuss detailed proofs of the law of quadratic reciprocity (Theorem 39.17).

More information on computational number theory, also called algorithmic number theory, can be found in the recent book by Bach and Shallit [6]. In the first volume the authors have discussed number theoretical problems that can be solved in (expected) polynomial time. In the forthcoming second volume, they will discuss the number theoretical problems that are considered intractable today. Although Knuth's book on seminumerical algorithms [17] is rather old, it is still a worthy reference book. ANTS (Algorithmic Number Theory Symposium) is a rather new conference that takes place every two years. The proceedings of the conferences are published in the *Lecture Notes in Computer Science* (Springer-Verlag). The first two conferences (1994 and 1996) have volume numbers 877 and 1122. Results on the topic have appeared in a wide range of conferences and journals, such as FOCS and STOC.

40

Crypto Topics and Applications I

Jennifer Seberry,
Chris Charnes,
Josef Pieprzyk, and
Rei Safavi-Naini
University of Wollongong

40.1 Introduction

In this chapter we discuss four related areas of cryptology, namely; authentication, hashing, message authentication codes (MACs), and digital signatures. These topics represent currently active and growing research topics in cryptology. Due to space limitations, we concentrate only on the essential aspects of each topic. The bibliography is intended to supplement our survey. We have included sufficiently many items to provide the interested reader with an overall view of the current state of knowledge in the above areas.

Authentication deals with the problem of providing assurance to a receiver that a communicated message originates from a particular transmitter, and that the received message has the same content as the transmitted message. A typical authentication scenario occurs in computer networks, where the identity of two communicating entities is established by means of authentication.

Hashing is concerned with the problem of providing a relatively short *fingerprint* of a much longer message or electronic document. A *hashing function* must satisfy (at least) the critical requirement that

the fingerprints of two distinct messages are distinct. Hashing functions have numerous applications in cryptology. They are often taken as primitives in constructing other cryptographic functions.

Message authentication codes (MACs) are symmetric key primitives that provide message integrity against active spoofing by appending a cryptographic checksum to a message that is verifiable only by the intended recipient of the message. Message authentication is one of the most important ways of ensuring the integrity of information that is transferred by electronic means.

Digital signatures provide electronic equivalents of handwritten signatures. They preserve the essential features of handwritten signatures, and can be used to sign electronic documents. Digital signatures can potentially be used in legal contexts.

40.2 Authentication

One of the main goals of a cryptographic system is to provide authentication, which simply means providing assurance about the content and origin of communicated messages.

Historically, cryptography began with secret writing, and this remained the main area of development until very recently. With the rapid progress in data communication, the need for providing message integrity and authenticity has escalated to the extent that currently authentication is seen as the more urgent goal of cryptographic systems.

Traditionally, it was assumed that a secrecy system provides authentication by the virtue of the secret key being only known by the intended communicants; this would prevent an enemy from constructing a fraudulent message. Simmons [66] argued that the two goals of cryptography are independent. He shows that a system that provides perfect secrecy might not provide any protection against authentication threats. Similarly, a system can provide perfect authentication without concealing messages.

In the rest of this chapter, we use the term *communication system* to encompass message transmission as well as storage. The system consists of a *transmitter* who wants to send a message, a *receiver* who is the intended recipient of the message, and an *enemy* who attempts to construct a fraudulent message with the aim of getting it accepted by the receiver unwittingly. In some cases, there is a fourth party, called the *arbiter,* whose basic role is to provide protection against cheating by the transmitter and/or the receiver. The communication is assumed to take place over a public channel, and hence the communicated messages can be seen by all the principals. An authentication threat is an attempt by an enemy in the system to modify a communicated message or inject a fraudulent message into the channel. In a secrecy system the attacker is passive, while in an authentication system the enemy is active and not only observes the communicated messages and gathers information such as plaintext and ciphertext, but also actively interacts with the system to achieve its goal. This view of the system clearly explains Simmons' motivation for basing authentication systems on game theory.

The most important criteria that can be used to classify authentication systems are

- The relation between authenticity and secrecy;
- The framework for the security analysis.

The first criterion divides authentication systems into those that provide *authentication with* and *without secrecy.* The second criterion divides systems into systems with *unconditional security,* systems with *computational security,* and systems with *provable security*. *Unconditional security* implies that the enemy has unlimited computational power, while in *computational security* the enemy's resources are limited and the security relies on the required computation exceeding the enemy's computational power. A system with *provable security* is in fact a subclass of computationally secure systems, and its compromise is equivalent to a solution of a known difficult problem.

These two classifications are orthogonal and produce four subclasses. Below we review the basic concepts of authentication theory, some known bounds and constructions in unconditional security, and then consider computational security.

Unconditional Security

A basic model introduced by Simmons [66] has remained the mainstay of most of the theoretical research on authentication systems. The model has the same structure as described in the previous section but excludes the arbiter. To provide protection against an enemy, the transmitter and receiver use an *authentication code (A-code)*. An A-code is a collection $\mathcal{E}$ of mappings called *encoding rules* (also called *keys*) from a set $\mathcal{S}$ of *source states* (also called *transmitter states*) into the set $\mathcal{M}$ of cryptogram (also called *codewords*). For A-codes without secrecy, also called *Cartesian A-codes*, a codeword uniquely determines a source state. That is, the set of codewords is partitioned into subsets each corresponding to a distinct source state. In a *systematic Cartesian A-code*, $\mathcal{M} = \mathcal{S} \times \mathcal{T}$ where $\mathcal{T}$ is a set of *authentication tags* and each codeword is of the form $s.t$, $s \in \mathcal{S}, t \in \mathcal{T}$ where '.' denotes concatenation. Let the cardinality of the set $\mathcal{S}$ of source states be denoted as k; that is, $|\mathcal{S}| = k$. Let $E = |\mathcal{E}|$ and $M = |\mathcal{M}|$. The encoding process adds key dependent redundancy to the message, so $k < M$. A key (or encoding rule) e determines a subset $\mathcal{M}_e \subset \mathcal{M}$ of codewords that are authentic under e.

The *incidence matrix* A of an A-code is a binary matrix of size $E \times M$ whose rows are labeled by encoding rules and columns by codewords, such that $A(e, m) = 1$ if m is a valid codeword under e, and $A(e, m) = 0$, otherwise.

An *authentication matrix* B of a Cartesian A-code is a matrix of size $E \times k$ whose rows are labeled by the encoding rules and columns by the source states, and $B(e, s) = t$ if t is the tag for the source state s under the encoding rule e.

To use the system, the transmitter and receiver must secretly share an encoding rule. The enemy does not know the encoding rule and uses an *impersonation* attack, in which it only uses its knowledge of the system, or a *substitution* attack in which it waits to see a transmitted codeword, and then constructs a fraudulent codeword. The security of the system is measured in terms of the enemy's chance of success with the chosen attack. The enemy's chance of success in an impersonation attack is denoted by P_0, and in a substitution attack by P_1. The best chance an enemy has in succeeding in either of the above attacks is called the *probability of deception*, P_d.

An attack is said to be *spoofing of order* ℓ if the enemy has seen ℓ communicated codewords under a single key. The enemy's chance of success in this case is denoted by P_ℓ.

The chance of success can be defined using two different approaches. The first approach corresponds to an average case analysis of the system and can be described as the enemy's payoff in the *game theory model*. It has been used by a number of authors, including MacWilliams, Gilbert and Sloane [32], Fak [28], and Simmons [66]. The second is to consider the worst-case scenario. This approach is based on the relation between A-codes and error correcting codes (also called E-codes).

Using the game theory model, P_j is the value of a zero-sum game between communicants and the enemy. For impersonation

$$P_0 = \max_{m \in \mathcal{M}} (\text{payoff}(m)) \ ,$$

and for substitution

$$P_1 = \sum_{j=1}^{E} \sum_{m \in \mathcal{M}} P(m) \max_{m'} \text{payoff}(m, m') \ ,$$

where $P(m)$ is the probability of a codeword m occurring in the channel, and payoff(m, m') is the enemy's payoff (best chance of success) when it substitutes an intercepted codeword m with a fraudulent one, m'.

Bounds on the Performance of the A-Code

The first types of bounds relate the main parameters of an A-code, that is, E, M, k, and hence are usually called *combinatorial bounds*. The most important combinatorial bound for A-codes with secrecy is

$$P_i \geq \frac{k - i}{M - i}, \ \ i = 1, 2, \ldots$$

and for A-codes without secrecy is

$$P_i \geq k/M, \ \ i = 1, 2, \ldots \ .$$

An A-code that satisfies these bounds with equality, that is, with $P_i = \dfrac{k-i}{M-i}$ for A-codes with secrecy and $P_i = k/M$ for Cartesian A-codes, is said to *provide perfect protection for spoofing of order* i. The enemy's best strategy in spoofing of order i for such an A-code is to randomly select one of the remaining codewords.

A-codes that provide perfect protection for all orders of spoofing up to r are said to be *r-fold secure*. These codes can be characterized using combinatorial structures such as orthogonal arrays and t-designs.

An orthogonal array $OA_\lambda(t, k, v)$ is an array with λv^t rows, each row of size k, from the elements of set X of v symbols, such that in any t columns of the array every t-tuple of elements of X occurs in exactly λ rows. Usually t is referred to as *the strength* of the OA.

EXAMPLE 40.1:

The following table gives a $OA_2(2, 5, 2)$ on the set {0, 1}:

0	0	0	0	0
1	1	0	0	0
0	0	0	1	1
1	1	0	1	1
1	0	1	0	1
0	1	1	0	1
1	0	1	1	0
0	1	1	1	0

A t-(v, k, λ) *design* is a collection of b subsets, each of size k, of a set, X, of size v where every distinct subset of size t occurs exactly λ times.

The *incidence matrix* of a t-(v, k, λ) is a binary matrix, $A = (a_{ij})$, of size $b \times v$ such that $a_{ij} = 1$ if element j is in block i and 0 otherwise.

EXAMPLE 40.2:

The following table gives a 3-(8, 4, 1) design on the set {0, 1, 2, 3, 4, 5, 6, 7}:

7	0	1	3
7	1	2	4
7	2	3	5
7	3	4	6
7	4	5	0
7	5	6	1
7	6	0	2
2	4	5	6
3	5	6	0
4	6	0	1
5	0	1	2
6	1	2	3
0	2	3	4
1	3	4	5

with incidence matrix:

$$\begin{array}{cccccccc}
1 & 1 & 0 & 1 & 0 & 0 & 0 & 1 \\
0 & 1 & 1 & 0 & 1 & 0 & 0 & 1 \\
0 & 0 & 1 & 1 & 0 & 1 & 0 & 1 \\
0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 \\
1 & 0 & 0 & 0 & 1 & 1 & 0 & 1 \\
0 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\
1 & 0 & 1 & 0 & 0 & 0 & 1 & 1 \\
0 & 0 & 1 & 0 & 1 & 1 & 1 & 0 \\
1 & 0 & 0 & 1 & 0 & 1 & 1 & 0 \\
1 & 1 & 0 & 0 & 1 & 0 & 1 & 0 \\
1 & 1 & 1 & 0 & 0 & 1 & 0 & 0 \\
0 & 1 & 1 & 1 & 0 & 0 & 1 & 0 \\
1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\
0 & 1 & 0 & 1 & 1 & 1 & 0 & 0
\end{array}$$

The main theorems relating A-codes with r-fold security and combinatorial structures are due to a number of authors, including Stinson [69], and Tombak and Safavi-Naini [77]. The following are the most general forms of these theorems.

THEOREM 40.1 *[77] Let the source be r-fold uniform. Then an A-code provides r-fold security against spoofing if and only if the incidence matrix of the code is the incidence matrix of a $(r+1)$- (M, k, λ) design.*

In the above theorem, an r-fold uniform source is a source for which every string of r distinct outputs from the source has probability $\dfrac{1}{k(k-1)\cdots(k-r+1)}$.

THEOREM 40.2 *Let $P_0 = P_1 = P_2 = \cdots = P_r = k/M$. Then the authentication matrix is a $OA(r+1, k, \ell)$ where $\ell = M/k$.*

The so-called *information theoretic bounds* characterize the enemy's chance of success using uncertainty measures. The first such bound for Cartesian A-codes, derived by MacWilliams, Gilbert and Sloane [32], is

$$P_1 \geq 2^{-\frac{H(M)}{2}}$$

where $H(M)$ is the entropy of the codeword space. The first general bound on P_0, due to Simmons [66], is

$$P_0 \geq 2^{-(H(E)-H(E|M))}$$

where $H(E)$ is the entropy of the key space and $H(E|M)$ is the conditional entropy of the key when a codeword is intercepted. Write $I(E; M)$ for the mutual information of E and M. Then,

$$P_0 \geq 2^{-I(E;M)} .$$

The above bound relates the enemy's best chance of success to the mutual information between the cryptogram space and the key space. A general form of this bound, proved independently by Rosenbaum [63] and Pei [47], is

$$P_\ell \geq 2^{-I(E;M^\ell)} ,$$

where $I(E; M^\ell)$ is the mutual information between a string of ℓ codewords, and where m^ℓ is the key.

Similar bounds for A-codes without secrecy are proved by MacWilliams et al. [32] and by Walker [79]. A general bound on the probability of deception, P_{d_r}, derived by Rosenbaum [63], is

$$P_{d_r} \geq 2^{-\frac{H(E)}{r+1}} .$$

Other Types of Attack

Tombak and Safavi-Naini [76] consider other types of attacks, similar to those for secrecy systems. In a *plaintext attack* against A-codes with secrecy, the enemy not only knows the codeword but also knows the corresponding plaintext. In *chosen content attack* the enemy wants to succeed with a codeword that has a prescribed plaintext. It is shown that by applying some transformation on the A-code it is possible to provides immunity against the above attacks.

A-codes with secrecy are generally more difficult to analyze than Cartesian A-codes. Moreover, the verification process for the former is not as efficient. In the case of Cartesian A-codes, verification of a received codeword, $s.t$, amounts to recalculating the tag using the secret key and the source state s to obtain t' and comparing it with the received tag t. For an authentic codeword we have $t = t'$. In the case of A-codes with secrecy, when a codeword m is received, the receiver must try all authentic codewords using his secret key, otherwise there must be an inverse algorithm that allows the receiver to find and verify the source state. The former process is costly and the latter does not exist for a general A-code. For practical reasons, the majority of research has concentrated on Cartesian A-codes.

Efficiency

Authentication systems require secure transmission of key information prior to the communication and hence, similar to secrecy systems, it is desirable to have a small number of key bits.

Rees and Stinson [60] prove that for any (M, k, E) A-code that is onefold secure, $E \geq M$. For A-codes with secrecy that provide onefold security, Stinson [69] shows that

$$E \geq \frac{M^2 - M}{k^2 - k} .$$

Under similar conditions for Cartesian A-codes, Stinson [69] shows that

$$E \geq k(\ell - 1) + 1$$

where $\ell = k/M$.

For A-codes with r-fold security, Stinson [71] shows that

$$E \geq \frac{M(M-1)\cdots(M-r)}{k(k-1)\cdots(k-r)} .$$

An A-code *provides perfect authenticity of order r* if $P_{d_r} = k/M$. In such codes the probability of success of the spoofer does not improve with the interception of extra codewords.

The following bound is established by Tombak and Safavi-Naini [75] for codes of the above type:

$$E \geq \frac{M^{r+1}}{k^{r+1}} .$$

Their bound shows that the provision of perfect authenticity requires $\log M - \log k$ extra key bits on average for every added order of spoofing. Hence using the same key for authentication of more than one message is expensive.

A second measure of efficiency, often used for systematic Cartesian A-codes, is the size of the tag space for a fixed size of source space and probability of success in substitution. Stinson [72] shows that, for perfect

protection against substitution, the size of key space grows linearly with the size of the source. Johansson, Kabatianskii and Smeets [37] show that if $P_1 > P_0$, A-codes with an exponential (in E) number of source states can be obtained.

A-Codes and E-Codes

An error correcting code provides protection against random channel error. The study of error-correcting codes was motivated by Shannon's Channel Capacity Theorem and has been a very active research area since the early 1950s. Error-correcting codes add redundancy to a message in such a way that a codeword corrupted by the channel noise can be detected and/or corrected. The main difference between an A-code and an E-code is that in the former redundancy depends on a secret key while in the latter it only depends on the message being coded. There exists a duality between authentication codes and error correcting codes. In the words of Simmons [66], " ... one (coding theory) is concerned with clustering the most likely alterations as closely about the original code as possible and the other (authentication theory) with spreading the optimal (to the opponent) alterations as uniformly as possible over $\mathcal{M}$."

The relation between E-codes and A-codes is explored in the work of Johansson et al. [37], who show that it is possible to construct E-codes from A-codes and vice-versa. Their work uses a worst case analysis approach in analyzing the security of A-codes. That is, in the case of substitution attack, they consider the *best* chance of success an enemy has when it intercepts all possible codewords. This contrasts with the information theoretic (or game theory) approach in which the average success probability of the enemy over all possible intercepted codewords is calculated.

The work of Johansson et al. is especially useful as it allows the well-developed body of bounds and asymptotic results from the theory of error correcting codes to be employed in the context of authentication codes, to derive upper and lower bounds on the size of the source for A-codes with given E, T, and P_1.

Authentication with Arbiter

In the basic model of authentication discussed above, the enemy is an outsider and we assume that the transmitter and the receiver are trustworthy. Moreover, because the key is shared by the transmitter and the receiver, the two principals are cryptographically indistinguishable. In an attempt to model authentication systems in which the transmitter and the receiver are distinguished and to remove assumptions about the trustworthiness of the two, Simmons [66] introduced a fourth principal called the *arbiter*. The transmitter and receiver have different keys and the arbiter has access to all or part of the key information. The system has a *key distribution phase* during which keys satisfying certain conditions are chosen. After that there is a *transmission phase* during which the transmitter uses its key to produce a codeword and finally a *dispute phase* during which disputes are resolved with the aid of the arbiter. The arbiter in Simmons' model is active during the transmission phase and is assumed to be trustworthy. Yung and Desmedt [83] remove this assumption and consider a model in which the arbiter is only trusted to resolve disputes. Johansson [35] and Kurosawa [39] derive lower bounds on the probability of deception in such codes. Johansson [36] and Taylor [73] propose constructions.

Shared Generation of Authenticators

Many applications require the power to generate an authentic message and/or to verify the authenticity of a message to be distributed among a number of principals. An example of such a situation is multiple signatures in a bank account or court room. Desmedt and Frankel [25] introduced systems with *shared generation of authenticators (SGA-systems)*, which have been studied in recent papers by Safavi-Naini [64], Gehrmann, van Dijk and Smeets, [29], and Safavi-Naini and Martin [65]. In such systems there is a group P of transmitters created with an structure Γ that determines authorized subsets of P. Each principal has a secret key which is used to generate a partial tag. The system has two phases. In the key distribution

phase, a *trusted authority* generates keys for the transmitters and the receivers and securely delivers the keys to them. In the communication phase, the trusted authority is not active. When an authorized group of transmitters wants to construct an authentic codeword, using its key, each group member generates a partial tag for the source state s which needs to be authenticated, and sends it to a *combiner*. The combiner is a fixed algorithm with no secret input. It combines its inputs to produce a tag, t, to be appended to s. The receiver is able to authenticate this codeword using its secret key. Safavi-Naini and Martin [65] give a general construction for SGA-systems by combining A-codes and secret sharing schemes. Gehrmann et al. [29] propose an efficient construction for SGA-systems, based on maximum rank distance separable codes.

Multiple Authentication

As noted before, in the theory of A-codes the possible attacks by the enemy are limited to impersonation and substitution. This means that the security of the system is only for one message communication; after that the key must be changed. To extend protection over more than one message transmission, there exist a number of alternatives. The most obvious ones use A-codes that provide perfect protection against spoofing of order ℓ. However, little is known about the construction of such codes and it is preferable to use A-codes that provide protection against substitution for more that one message transmission. Vanroose, Smeets and Wan [82] suggest *key strategies* in which the communicants change their key after each transmitted codeword, using some prespecified strategy. In this case the key information shared by the communicants is the *sequence of keys* to be used for consecutive transmission slots. The resulting bounds on the probability of deception generalize the bounds given by the following authors: Pei [47], Rosenbaum [63], and Walker [79].

Another successful approach proposed by Wegman and Carter [80] uses a special class of hash functions together with a one time pad of random numbers. This construction is discussed in more detail in Section 40.5.

40.3 Computationally Secure Systems

The study of computationally secure A-systems is relatively informal, cf. Simmons [67]. The basic framework is similar to unconditionally secure systems. A simple computationally secure A-code can be obtained by considering $\mathcal{S} = GF(2^{40})$ and $\mathcal{M} = GF(2^{64})$. We use $\mathcal{E}$ to be the collection of DES [51] encryption functions and so $E = 2^{56}$. To construct the codeword corresponding to a source state s, using the key k, we append 24 zeros to s and then use DES with key k to encrypt $s.\mathbf{0}_{24}$, where $\mathbf{0}_{24}$ is the string of 24 zeros.

It is easy to see that the above scheme is an A-code with secrecy. It allows the receiver to easily verify the authenticity of a received codeword by decrypting a received codeword and checking the existence of the string of zeros. If an enemy wants to impersonate the transmitter its chance of success is 2^{-56}, which is the probability of guessing the correct key. For a substitution attack, the enemy waits to see a transmitted codeword. Then it uses all the keys to decrypt the codeword and once a decryption of the right form (ending in 24 zeros) is obtained, a possible key is found. In general there is more than one key with this property. On the average there are $2^{56} \times 2^{40}/2^{64} = 2^{32}$ pairs (s, k) that produce the same cryptogram and hence the chance of guessing correctly is 2^{-32}. A better strategy for the enemy is to randomly choose a cryptogram and send it to the receiver. In this case his chance of success is 2^{-24}, which is better than the previous case.

The security of computationally secure A-systems weakens very quickly as the enemy intercepts more cryptograms. Trying all possible keys on ℓ received cryptograms enables the enemy to uniquely identify the key, in which case the enemy's chance of success is one.

Computationally secure A-systems without secrecy are obtained by appending an *authenticator* to the message which is verifiable by the intended receiver. The authenticator can be produced by a *symmetric*

key algorithm or an *asymmetric key algorithm*. The former is the subject of the section on message authentication codes (MAC), while the latter is discussed in the section on digital signatures.

40.4 Hashing

In many cryptographic applications, it is necessary to produce a relatively short *fingerprint* of a much longer message or electronic document. The fingerprint is also called a digest of the message. Ideally, a hash function should produce a unique digest of a fixed length for a message of an arbitrary length. Obviously, this is impossible as any hash function is, in fact, a many-to-one mapping. The properties required for secure hashing can be summarized as follows:

- Hashing should be a many-to-one function producing a digest that is a complex function of all bits of the message;
- A hash function should behave as a random function that creates a digest for a given message by randomly choosing an element from the whole digest space;
- For any pair of messages, it should be computationally difficult to find a collision; i.e., distinct messages with the same digest; and
- A hash function should be one-way; i.e., it should be easy to compute the digest of a given message but difficult to determine the message corresponding to a given digest.

The main requirement of secure hashing is that it should be *collision-free* in the sense that finding two colliding messages is computationally intractable. This requirement must hold for long as well as short messages.

Strong and Weak Hash Functions

Hash functions can be broadly classified into two classes: *strong one-way hash functions* (also called *collision-free hash functions*) and *weak one-way hash functions* (also known as *universal one-way hash functions*). A strong one-way hash function is a function h satisfying the following conditions:

1. h can be applied to any message or document M of any size;
2. h produces a fixed size digest;
3. Given h and M, it is easy to compute the digest $h(M)$; and
4. Given h, it is computationally infeasible to find two distinct messages M_1, M_2 that collide, i.e., $h(M_1) = h(M_2)$.

On the other hand, a weak one-way hash function is a function that satisfies conditions 1, 2, 3 and the following:

4'. Given h and a randomly chosen message M, it is computationally intractable to find another message M' that collides with M, i.e., $h(M) = h(M')$.

Strong one-way hash functions are easier to use since there is no precondition on the selection of the messages. On the other hand, for weak one-way hash functions, there is no guarantee that a nonrandom selection of two messages is collision-free. This means that the space of easily found colliding messages must be small. Otherwise, a random selection of two messages would produce a collision with a nonnegligible probability.

Theoretic Constructions

Naor and Yung [44] introduced the concept of a universal one-way hash function (UOWHF) and suggested a construction based on a one-way permutation. In their construction, they employ the notion of a *universal family of functions with collision accessibility property* [80]. The above functions are defined as follows.

DEFINITION 40.1 Suppose $G = \{g \mid A \rightarrow B\}$ is a set of functions. G is strongly *universal*$_r$ if given any r distinct elements $a_1, \ldots, a_r \in A$, and any r elements $b_1, \ldots, b_r \in B$, there are $|G|/|B|^2$ functions which take a_1 to b_1, a_2 to b_2 etc. ($|G|$ and $|B|$ denote the cardinality of sets G and B, respectively.)

DEFINITION 40.2 A strongly *universal*$_r$ family of functions G has the collision accessibility property if it is possible to generate in polynomial time a function $g \in G$ that satisfies the equations

$$g(a_i) = b_i, \ 1 \leq i \leq r \ .$$

Naor and Yung construct a family of UOWHFs by concatenating any one-way permutation with a family of strongly universal$_2$ hash functions having the collision accessibility property. In this construction, the one-way permutation provides the one-wayness of the UOWHF, and the strongly universal$_2$ family of hash functions provides the mapping to the small length output. When a function is chosen randomly and uniformly from the family, the output is distributed randomly and uniformly over the output space.

Zheng, Matsumoto and Imai [84] define a scheme based on the composition of a pairwise independent uniformizer and a strongly universal hash function with a quasi-injection one-way function.

De Santis and Yung [24] construct hash functions from one-way functions with an almost-known preimage size. In other words, if an element of the domain is given, then with a polynomial uncertainty an estimate of the size of the preimage set is easily computable. A *regular* function is an example of such a function. (In a regular function, each image of an n-bit input has the same number of preimages of length n.)

Rompel [62] constructs a UOWHF from any one-way function. His construction is rather elaborate. Briefly, his idea is to transform any one-way function into a UOWHF through a sequence of complicated procedures. First, the one-way function is transformed into another one-way function such that for most elements of the domain except for a fraction, it is easy to find a collision. From this, another one-way function is constructed such that for most of the elements it is hard to find a collision. Subsequently, a length increasing one-way function is constructed for which it is hard to find collisions almost everywhere. Finally, this is turned into a UOWHF which compresses the input in a way that makes it difficult to find a collision (cf. Pieprzyk and Sadeghiyan [49]).

Hashing Based on Block Ciphers

To minimize the design effort for cryptographically secure hash functions, the designers of hash functions tend to base their schemes on existing encryption algorithms. For example, sequential hashing can be obtained by dividing a given message into blocks and applying an encryption algorithm on the message blocks. The message block length must be the same as the block length of the encryption algorithm. If the message length is not a multiple of the block length, then the last block is usually padded with some redundant bits. To provide a randomizing element, an initial public vector is normally used. The proof of the security of such schemes relies on the collision freeness of the underlying encryption algorithm.

In the following, let E denote an arbitrary encryption algorithm. Let $E(K, M)$ denote the encryption of message M with key K using E; let IV denote the initial vector.

Rabin [55] shows that any private-key cryptosystem can be used for hashing. Rabin's scheme is the following. First the message is divided into blocks $M_1, M_2, \ldots$ of the same size as the block length of the

encryption algorithm. In the case of DES, the message is divided into 64-bit blocks. To hash a message $M = (M_1, M_2, \ldots, M_t)$, the following computations are performed:

$$\begin{aligned} H_0 &= IV \\ H_i &= E(M_i, H_{i-1}) \quad i = 1, 2, \ldots, t \\ H(M) &= H_t \end{aligned}$$

where M_i is a message block, H_i are intermediate results of hashing, and $H(M)$ is the digest. Although Rabin's scheme is simple and elegant, it is susceptible to the so-called *birthday attack* when the size of the hash value is 64 bits.

Winternitz [81] proposes a scheme for the construction of a one-way hash function from any block cipher. In any good block cipher, given an input and an output, it should be difficult to determine the key, but from the key and the output it should be easy to determine the input. The scheme uses an operation E^* defined by:

$$E^*(K \parallel M) = E(K, M) \oplus M \, .$$

Based on the above scheme, Davies [23] proposed the following hashing algorithm:

$$\begin{aligned} H_0 &= IV \\ H_i &= E(M_i, H_{i-1}) \oplus H_{i-1} \quad i = 1, 2, \ldots, t \\ H(M) &= H_t . \end{aligned}$$

If $E(K, M)$ is DES, then it may be vulnerable to attacks based on weak keys or a key-collision search. The meet-in-the-middle attack is thwarted because $E(K, M)$ is a one-way function.

Merkle [43, 42] proposed hashing schemes based on Winternitz's construction. These schemes use DES to produce digests of size $\approx$ 128 bits.

Their construction follows a general method for constructing hash algorithms, called the *meta method*. This is the same as the serial method of Damgård [22]. The description of the method is as follows. The message is first divided into blocks of 106 bits. Each 106-bit block M_i of data is concatenated with the 128-bit block H_{i-1}. The concatenation $X_i = M_i \parallel H_{i-1}$ contains 234 bits. Each block X_i is further divided into halves, X_{i1} and X_{i2}.

$$\begin{aligned} H_0 &= IV \\ X_i &= H_{i-1} \parallel M_i \\ H_i &= E^*(00 \parallel \text{first 59 bits of}\{E^*(100 \parallel X_{1i})\} \parallel \\ &\quad \text{first 59 bits of}\{E^*(101 \parallel X_{2i})\}) \parallel \\ &\quad E^*(01 \parallel \text{first 59 bits of}\{E^*(110 \parallel X_{1i})\} \parallel \\ &\quad \text{first 59 bits of}\{E^*(111 \parallel X_{2i})\}) \\ H(M) &= H_t \, . \end{aligned}$$

In this scheme E^* is defined as in Winternitz's construction. The strings 00, 01, 100, 101, 110, and 111 above are used to prevent the manipulation of weak keys.

Hashing Functions Based on Intractable Problems

Hashing functions can also be based on one-way functions such as exponentiation, squaring, knapsack (cf. Pieprzyk and Sadeghiyan [49]), and discrete logarithm. More recently, a group-theoretic construction using the SL_2 groups has been proposed by Tillich and Zémor [74].

A scheme based on RSA exponentiation as the underlying one-way function is defined by

$$\begin{aligned} H_0 &= IV \\ H_i &= (H_{i-1} \oplus M_i)^e \bmod N \quad i = 1, 2, \ldots, t \\ H(M) &= H_t \end{aligned}$$

where the modulus N and the exponent e are public. A correcting block attack can be used to compromise the scheme by appending or inserting a carefully selected last block message to achieve a desired hash value. To immunize the scheme against this attack, it is necessary to add redundancy to the message so that the last message block cannot be manipulated (cf. Davies and Price [23]). To ensure the security of RSA, N should be at least 512 bits in length, making the implementation of the above scheme very slow.

To improve the performance of the above scheme, the public exponent can be made small. For example, squaring can be used:

$$H_i = \left(H_{i-1} \oplus M_i\right)^2 \bmod N \, .$$

It is suggested that 64 bits of every message block be set to 0, to avoid a correcting block attack.

An algorithm for hashing based on squaring is proposed in Appendix D of the X.509 recommendations of the CCITT standards on secure message handling. The proposal stipulates that 256 bits of redundancy be distributed over every 256-bit message block by interleaving every four bits of the message with 1111, so that the total number of bits in each block becomes 512. The exponentiation algorithm, with exponent two, is then run on the modified message in CBC mode (cf. Pieprzyk and Sadeghiyan [49]). In this scheme, the four most significant bits of every byte in each block are set to 1. Coppersmith [20] shows how to construct colliding messages in this scheme.

Damgård [22] describes a scheme based on squaring, which maps a block of n bits into a block of m bits. The scheme is defined by

$$\begin{aligned} H_0 &= IV \\ H_i &= \text{extract}(00111111 \parallel H_{i-1} \parallel M_i)^2 \bmod N \\ H(M) &= H_t . \end{aligned}$$

In the above scheme, the role of *extract* is to extract m bits from the result of the squaring function. To obtain a secure scheme, m should be sufficiently large so as to thwart the birthday attack; this attack will be explained later. Moreover, *extract* should select bits for which finding colliding inputs is difficult. One way to do this is to extract m bits uniformly. However, for practical reasons, it is better to bind them together in bytes. Another possibility is to extract every fourth byte. Daemen, Govaerts and Vanderwalle [21] show that this scheme can be broken.

Impagliazzo and Naor [34] propose a hashing function based on the knapsack problem. The description of the scheme is as follows. Choose at random numbers $a_1, \ldots, a_n$ in the interval $0, \ldots, N$, where n indicates the length of the message in bits, and $N = 2^\ell - 1$ where $\ell < n$. A binary message $M = M_1, M_2, \ldots, M_n$ is hashed as

$$H(M) = \sum_{i=1}^{n} a_i M_i \bmod 2^\ell \, .$$

Impagliazzo and Naor do not give any concrete parameters for the above scheme, but they have shown that it is theoretically sound.

Gibson [31] constructs hash functions whose security is conditional upon the difficulty of factoring certain numbers. The hash function is defined by

$$f(x) = a^x (\bmod n) \, ,$$

where $n = pq$, p and q are large primes, and a is a primitive element of the ring Z_n. In Gibson's hash function n has to sufficiently large to ensure the difficulty of factoring. This constraint makes the hash function considerably slower than the MD4 algorithm.

Tillich and Zémor [74] proposed a hashing scheme where the message digests are given by two-dimensional matrices with entries in the binary Galois fields $GF(2^n)$ for $130 \leq n \leq 170$. The hashing functions are parameterized by the irreducible polynomials of degree n, $P_n(X)$, over $GF(2)$; their choice is left to the user. Their scheme has several provably secure properties: detection of local modification of text; and resistance to the birthday attack as well as a few other attacks. Hashing is fast as digests are produced by matrix multiplication in $GF(2^n)$, which can be parallelized.

Messages (encoded as a binary strings) $x_1x_2\ldots$ of arbitrary length are mapped to products of a selected pair of *generators* $\{A, B\}$ of the group $SL(2, 2^n)$, as follows:

$$x_i = \begin{cases} A & \text{if } x_i = 0 \\ B & \text{if } x_i = 1. \end{cases}$$

The resulting product belongs to the (infinite) group $SL(2, GF(2)[X])$, where $GF(2)[X]$ is the ring of all polynomials over $GF(2)$. The product is then reduced modulo an irreducible polynomial of degree n (Euclidean algorithm), mapping it to an element of $SL(2, 2^n)$. The four n-bit entries of the reduced matrix give the $(3n + 1)$-bit message digest of $x_1x_2\ldots$.

Charnes and Pieprzyk [15] showed that irreducible polynomials which produce collisions for the SL_2 hash functions can be found. Other weaknesses of this scheme are explored in a later paper available from them [17]. In that paper, the weak parameters are characterized; they can be computed with the algorithms given there.

Geiselmann [30] describes an algorithm to produce potential collisions for the $SL(2, 2^n)$ hashing scheme, which is independent of the choice of the irreducible polynomials. The complexity of his algorithm is that of the discrete logarithm problem in $GF(2^n)$ or $GF(2^{2n})$. However, no collisions in the specified range of the hash function have been found using this algorithm. Some pairs of rather long colliding strings are given by Geiselmann for a toy example of $GF(2^{21})$.

Hashing Algorithms

Rivest [58] proposes a hashing algorithm called MD4. It is a software-oriented scheme that is especially designed to be fast on 32-bit machines. The algorithm produces a 128-bit output, so it is not computationally feasible to produce two messages having the same hash value. The scheme provides diffusion and confusion using three Boolean functions. The MD5 hashing algorithm is a strengthened version of MD4 [57]. MD4 has been broken by Dobbertin [26].

HAVAL stands for a one-way hashing algorithm with a variable length of output. It was designed at the University of Wollongong by Zheng, Pieprzyk and Seberry [85]. It compresses a message of an arbitrary length into a digest of either 128, 160, 192, 224 or 256 bits. The security level can be adjusted by selecting 3, 4, or 5 passes. The structure of HAVAL is based on MD4 and MD5. Unlike MD4 and MD5 whose basic operations are done using functions of three Boolean variables, HAVAL employs five Boolean functions of seven variables (each function serves a single pass). All functions used in HAVAL are highly nonlinear, 0-1 balanced, linearly inequivalent, mutually output-uncorrelated and satisfy the strict avalanche criterion (SAC). No attack on HAVAL has been reported so far.

Charnes and Pieprzyk [14] proposed a modified version of HAVAL based on five Boolean functions of five variables. The resulting hashing algorithm is faster than the five pass, seven variable version of the original HAVAL algorithm. They use the same cryptographic criteria that are used to select the Boolean functions in the original scheme. Unlike the seven variable case, the choice of the Boolean functions is fairly restricted in the modified setting. Using the shortest algebraic normal form of the Boolean functions as one of the criteria (to maximize the speed of processing), it is shown that the Boolean functions used are optimal. No attacks have been reported for the five variable version.

Attacks

The best method to evaluate a hashing scheme is to see what attacks an adversary can perform to find collisions. A good hashing algorithm produces a fixed length number which depends on all the bits of the message. It is generally assumed that the adversary knows the hashing algorithm. In a conservative approach, it is assumed that the adversary can perform an adaptive chosen message attack, where it may choose messages, ask for their digests, and try to compute colliding messages. There are several methods for using such pairs to attack a hashing scheme and to calculate colliding messages. Some methods are quite general and can be applied against any hashing scheme; for example, the so-called *birthday attack*. Other methods are applicable only to specific hashing schemes. Some attacks can be used against a wide range of hash functions. For example, the so-called *meet-in-the-middle attack* is applicable to any scheme that uses some sort of block chaining in its structure. As another example, the so-called *correcting block attack* is applicable mainly to hash functions based on modular arithmetic.

Birthday Attack

The idea behind this attack originates from a famous problem from probability theory, called the *birthday paradox*. The paradox can be stated as follows. What is the minimum number of pupils in a classroom so the probability that at least two pupils have the same birthday is greater than 0.5? The answer to this question is 23, which is much smaller than the value suggested by intuition. The justification for the paradox is as follows. Suppose that the pupils are entering the classroom one at a time. The probability that the birthday of the first pupil falls on a specific day of the year is equal to $\frac{1}{365}$. The probability that the birthday of the second pupil is not the same as the first one is equal to $1 - \frac{1}{365}$. If the birthdays of the first two pupils are different, the probability that the birthday of the third pupil is different from the first one and the second one is equal to $1 - \frac{2}{365}$. Consequently, the probability that t students have different birthdays is equal to $(1 - \frac{1}{365})(1 - \frac{2}{365}) \ldots (1 - \frac{t-1}{365})$, and the probability that at least two of them have the same birthday is

$$P = 1 - \left(1 - \frac{1}{365}\right)\left(1 - \frac{2}{365}\right) \cdots \left(1 - \frac{t-1}{365}\right) .$$

It can be easily computed that for $t \geq 23$, this probability is greater than 0.5.

The birthday paradox can be employed for attacking hash functions. Suppose that the number of bits of the hash value is n. An adversary generates r_1 variations of a bogus message and r_2 variations of a genuine message. The probability of finding a bogus message and a genuine message that hash to the same digest is

$$P \approx 1 - e^{-\frac{r_1 r_2}{2^n}}$$

where $r_2 \gg 1$ (see Ohta and Koyama [46]). When $r_1 = r_2 = 2^{\frac{n}{2}}$, the above probability is ≈ 0.63. Therefore any hashing algorithm which produces digests of length around 64 bits is insecure, since the time complexity function for the birthday attack is $\approx 2^{32}$. It is usually recommended that the hash value should be around 128 bits to thwart the birthday attack.

This method of attack does not take advantage of the structural properties of the hash scheme or its algebraic weaknesses. It applies to any hash scheme. In addition, it is assumed that the hash scheme assigns to a message a value which is chosen with a uniform probability among all the possible hash values. Note that if the structure is weak or has certain algebraic properties, the digests do not have a uniform probability distribution. In such cases it may be possible to find colliding messages with a better probability and fewer message-digest pairs.

Meet-in-the-Middle Attack

This is a variation of the birthday attack, but instead of comparing the digests, the intermediate results in the chain are compared. The attack can be made against schemes which employ some sort of block chaining in their structure. In contrast to birthday attack, a meet-in-the-middle attack enables an

attacker to construct a bogus message with a desired digest. In this attack the message is divided into two parts. The attacker generates r_1 variations on the first part of a bogus message. He starts from the initial value and goes forward to the intermediate stage. He also generates r_2 variations on the second part of the bogus message. He starts from the desired false digest and goes backward to the intermediate stage. The probability of a match in the intermediate stage is the same as the probability of success in the birthday attack.

Correcting-Block Attack

In a correcting block attack, a bogus message is concatenated with a block to produce a corrected digest of the desired value. This attack is often applied to the last block and is called *correcting last block* attack, although it can be applied to other blocks as well. Hash functions based on modular arithmetic are especially sensitive to the correcting last block attack (cf. Preneel [50]). The introduction of redundancy into the message in these schemes makes finding a correcting block with the necessary redundancy difficult. However, it makes the scheme less efficient. The difficulty of finding a correcting block depends on the nature of the redundancy introduced. For example, Coppersmith [20] shows that the redundancy built into the CCITT hashing scheme based on modular squaring results in an insecure scheme.

Biham and Shamir [11] have developed a method for attacking block ciphers, known as *differential cryptanalysis*. This is a general method for attacking cryptographic algorithms, including hashing schemes. For example, Berson [8] has applied differential cryptanalysis to MD5.

40.5 MAC

Message authentication codes provide message integrity and are one of the most important security primitives in current distributed information systems. A *message authentication code* (MAC) is a symmetric key cryptographic primitive that consists of two algorithms. A MAC *generation algorithm*, $G = \{G_k : k = 1, \ldots, N\}$ takes an arbitrary message, s, from a given collection $\mathcal{S}$ of messages and produces a *tag*, $t = G_k(s)$, which is appended to the message to produce an *authentic* message, $m = (s.t)$. A *MAC verification* algorithm, $V = \{V_k(.) : k = 1, \ldots, N\}$, takes authenticated messages of the form $(s.t)$, and produces a true or false value, depending on whether the message is authentic. The security of a MAC depends on the best chance that an active spoofer has to successfully substitute a received message $(s.G_k(s))$ for a fraudulent one, $m' = (s', t)$, so that $V_k(m')$ produces a true result. In MAC systems, the communicants share a secret key, and are therefore not distinguishable cryptographically.

The security of MACs can be studied from the point of view of unconditional or computational security.

Unconditionally secure MACs are equivalent to cartesian authentication codes. However, in MAC systems only multiple communications are of interest. In Section 40.2, A-codes that provide protection for multiple transmissions were discussed. In the next section, we present a construction for a MAC that has been the basis of all the recent MAC constructions and has a number of important properties. It is provably secure; the number of key bits required is asymptotically minimal; and it has a fast implementation.

Computationally secure MACs have arisen from the needs of the banking community, cf. Preneel et al., [52]. They are also studied under other names, such as *keyed hash functions* and *keying hash functions*. In Section "Computational Security," we review the main properties and constructions of such MACs.

Unconditionally Secure MACs

When the enemy has unlimited computational resources, attacks against MAC systems and the analysis of security are similar to that of Cartesian A-codes. The enemy observes n codewords of the form $s_i.t_i$, $i = 1, \ldots, n$, in the channel and attempts to construct a fraudulent codeword $s.t$ which is accepted by the receiver. (This is the same as spoofing of order n in an A-code.) If the communicants want to limit the enemy's chance of success to p after n message transmissions, the number of authentication functions

(number of keys) must be greater than a lower bound which depends on p. If the enemy's chance of success in spoofing of order i, $i = 1, \ldots, n$, is p_i, then at least $1/p_1 p_2 \cdots p_n$ keys are required; see Fak [28], Wegman and Carter [80] for a proof of this. For $p_i = p, i = 1, \ldots, n$, the required number of key bits is $-n \log_2 p$. That is, for every message, $-\log_2 p$ key bits are required. This is the absolute minimum for the required number of key bits.

Perfect protection is obtained when the enemy's best strategy is a random choice of a tag and appending it to the message; this strategy succeeds with probability $p = 2^{-k}$, if the size of the tag is k bits. In this case the number of required key bits for every extra message is k.

Wegman and Carter [80] give a general construction for unconditionally secure MACs that can be used for providing protection for an arbitrary number of messages.

Their construction uses *universal classes of hash functions*. Traditionally, a hash function is used to achieve fast average performance over all inputs in varied applications, such as databases. By using a universal class of hash functions it is possible to achieve provable average performance without restricting the input distribution.

Let $h : A \rightarrow B$ be a hash function mapping the elements of a set A to a set B. A *strongly universal$_n$ class* of hash function is a class of hash functions with the property that for n distinct elements $a_1, \ldots, a_n$ of A and n distinct elements $b_1, \ldots, b_n$ of B, exactly $|H|/(b^n)$ functions map a_i to b_i, for $i = 1, \ldots, n$. Strongly universal$_n$ hash functions give perfect protection for multiple messages as follows. The transmitter and the receiver use a publicly known class of strongly universal$_n$ hash functions, and a shared secret key determines a particular member of the class that they will use for their communication. Stinson [70] shows that a class of strongly universal$_2$ that maps a set of a elements to a set of b elements is equivalent to an orthogonal array $OA_\lambda(2, a, b)$ with $\lambda = |H|/b^2$. Similar results can be proved for strongly universal$_n$ classes of hash functions. Because of this equivalence, universal$_n$ hash functions are not a practically attractive solution. In particular, this proposal is limited by the constraints of constructing orthogonal arrays with arbitrary parameters. For a survey of known results on orthogonal arrays, see Beth, Jungnickel and Lenz [10].

Wegman and Carter Construction

Wegman and Carter show that, instead of strongly universal$_n$ one can always use a strongly universal$_2$ family of hash functions, together with a one time pad of random numbers. The system works as follows. Let B denote the set of tags consisting of the sequences of k bit strings. Let $\mathcal{H}$ denote a strongly universal$_2$ class of hash functions mapping $\mathcal{S}$ to B. Two communicants share a key that specifies a function $h \in \mathcal{H}$ together with a pad containing k-bit random numbers. The tag for the j^{th} message s_j is $h(s_j) \oplus r_j$, where r_j is the j^{th} number on the pad. It can be proved that this system limits the enemy's chance of success to 2^{-k} as long as the pad is random and not used repeatedly. The system requires $nk + K$ bits of key, where K is the number of bits required to specify an element of $\mathcal{H}$, n is the number of messages to be authenticated, and k is the size of the tags.

This construction has a number of remarkable properties. Firstly, for large n the key requirement for the system approaches the theoretical minimum of k bits per message. This is because for large n the number of key bits is effectively determined by nk. Secondly, the construction of MAC with provable security for multiple communications is effectively reduced to the construction of a better studied primitive, that is, strongly universal$_2$ class of hash functions. Finally, by replacing the one-time pad with a pseudorandom sequence generator, unconditional security is replaced by computational security.

Wegman and Carter's important observation is as follows. By not insisting on the minimum value for the probability of success in spoofing of order one, i.e., allowing $p_1 > 1/k$, it is possible to reduce the number of functions, and thus the required number of keys. This observation leads to the notion of almost strongly universal$_2$ class.

An *ϵ-almost universal$_2$* (or ϵ-AU$_2$) class of hash functions has the following property. For any pair $x, y \in A, x \neq y$, the number of hash functions h with $h(x) = h(y)$ is at most equal to ϵ. The *ϵ-almost*

strongly-universal$_2$ (or ϵ-ASU$_2$) hash functions have the added property that for any $x \in A$, $y \in B$ the number of functions with $h(x) = y$ is $|H|/|B|$. Using an ϵ-almost strongly universal$_2$ class of functions in the Wegman and Carter construction results in MAC systems for which the probability of success for an intruder is ϵ. Such MACs are called ϵ*-otp-secure*, see Krawczyk [38].

Stinson [72] gives several methods for combining hash functions of class AU$_2$ and ASU$_2$. The following theorem shows that an ϵ-ASU$_2$ class can be constructed from an ϵ-AU$_2$ class.

THEOREM 40.3 *[72] Suppose H_1 is an ϵ_1-AU$_2$ class of hash functions from A_1 to B_1, and suppose H_2 is an ϵ_2-ASU$_2$ class of hash functions from B_1 to B_2. Then there exists an ϵ-ASU$_2$ class H of hash functions from A_1 to B_2, where $\epsilon = \epsilon_1 + \epsilon_2$ and $|H| = |H_1| \times |H_2|$.*

This theorem further reduces the construction of MACs with provable security to the construction of ϵ-AU$_2$ class of hash functions.

Several constructions for ϵ-ASU$_2$ hash functions are given by Stinson [72]. Johansson et al. [37] establish relationships between ASU$_2$ hash functions and error correcting codes. They use geometric error correcting codes to construct new classes of ϵ-ASU$_2$ hash function of smaller size. This reduces the key size.

Krawczyk [38] shows that in the Wegman–Carter construction, ϵ-ASU$_2$ hash functions can be replaced with a less demanding class of hash functions, called ϵ*-otp-secure*. The definition of this class differs from other classes of hash functions, in that it is directly related to MAC constructions and their security, in particular, to the Wegman–Carter construction.

Let $s \in \mathcal{S}$ denote a message that is to be authenticated by a k bit tag $h(s) \oplus r$, constructed by Wegman and Carter's method. An enemy succeeds in a spoofing attack if he can find $s' \neq s$, $t' = h(s') \oplus r$, assuming that he knows H but does not know h and r. A class H of hash functions is ϵ*-otp-secure* if for any message no adversary succeeds in the above attack scenario with probability greater than ϵ.

THEOREM 40.4 *[38] A necessary and sufficient condition for a family H of hash functions to be ϵ-otp-secure is that*

$$\forall a_1 \neq a_2,\ c,\quad Pr_h\left(h\left(a_1\right) \oplus h\left(a_2\right) = c\right) \leq \epsilon\ .$$

The need for high speed MACs has increased with the progress in high speed data communication. A successful approach to the construction of such MACs uses hash function families in which the message is hashed by multiplying it by a binary matrix. Because hashing is achieved with exclusive-or operations, it can be efficiently implemented in software. An obvious candidate for such a class of hash functions, originally proposed by Wegman and Carter [13, 80], is the set of linear transformations from A to B. It is shown that this forms an ϵ-AU$_2$ class of hash functions. However the size of the key—the number of entries in the matrix—is too large, and too many operations are required for hashing. Later proposals by Krawczyk [38] and by Rogaway [61] are aimed at alleviating these problems, and have a fast software implementation. The former uses Topelitz matrices [38], while the latter uses binary matrices with only three ones per column. In both cases, the resulting family is ϵ-AU$_2$.

The design of a complete MAC usually involves a number of hash functions which are combined by methods, similar to those proposed by Stinson [72]. The role of some of the hash functions is to produce high compression (small b), while others produce the desired spread and uniformity (see Rogaway [61]).

Reducing the key size of the hash function is especially important in practical applications, because the one-time pad is replaced by the output of a pseudorandom generator with a short key (of the order of 128 bits). Hence it is desirable to have the key size of the hash function of similar order.

Computational Security

In the computationally secure approach, protection is achieved because excessive computation is required for a successful forgery. Although a hash value can be used as a checksum to detect random changes in the data, a secret key must be used to provide protection against active tampering. Methods for constructing MACs from hash functions have traditionally followed one of the following approaches: the so-called *hash-then-encrypt* and *keying a hash function.*

Hash-Then-Encrypt To construct a MAC for a message x with this method, the hash value of x is calculated and the result is encrypted using an encryption algorithm. This is similar to signature generation, where a public key algorithm is replaced by a private key encryption function.

There are a number of drawbacks to this method. First, the overall scheme is slow. This is because the two primitives used in the construction, i.e., the cryptographic hash functions and encryption functions, are designed for other purposes and have extra security properties which are not strictly required in the construction. Although this construction can produce a secure MAC, the speed of the MAC is bounded by the speed of its constituent algorithms. For example, cryptographic hash functions are designed to be one-way. It is not clear whether this is a required property in the hash-then-encrypt construction, where the output of the hash function is encrypted and one-wayness is effectively obtained through the difficulty of finding the plaintext from the ciphertext.

A serious shortcoming of this method is that existing export restrictions, which usually apply to encryption functions, are inherited by MACs constructed using this method.

Keying a Hash Function In the second approach a secret key is incorporated into a hashing algorithm. This operation is sometimes called *keying a hash function* (see Bellare, Canetti and Krawczyk [5]). This method is attractive, because of the availability of hashing algorithms and their relative speed in software implementation; these algorithms are not subject to export restrictions.

Although this scheme can be implemented more efficiently in software than the previous scheme, the objection to the superfluous properties of the hash functions remains.

The keying method depends on the structure of the hash function. Tsudik [78] proposes three methods of incorporating the key into the data. In the *secret prefix method*, $G_k(s) = H(k\|s)$, while in the *secret suffix*, we have $G_k(s) = H(s\|k)$. Finally, the *envelope method* combines the previous two methods with $G_k(s) = H(k_1\|s\|k_2)$ and $k = k_1\|k_2$.

Instead of including the key into the data, the key information can be included into the hashing algorithm. In iterative hash functions such as MD5 and SHA, the key can be incorporated into the initial vector, compression function or into the output transformation.

There have also been some attempts at defining and constructing *secure keyed hash functions* as independent primitives, namely by Berson, Gong and Lomas [9] and Bakhtiari, Safavi-Naini and Pieprzyk [1]. The former propose a set of criteria for secure keyed hash functions and give constructions using one-way hash functions. The latter argue that the suggested criteria for security is in most cases excessive; relaxing these allows constructions of more efficient secure keyed hash functions. Bakhtiari et al. also give a design of a keyed hash function from scratch. Their design is based mostly on intuitive principles and lacks a rigorous proof of security. A similar approach is taken in the design of MDx-MAC by Preneel and van Oorschot [53], which is a scheme for constructing a MAC from an MD5-type hash function. It is conjectured that if MDx is a secure hash function, then MDx-MAC is a secure MAC.

Security Analysis of Computationally Secure MACs

The security analysis of computational secure MACs has followed two different approaches. In the first approach, the security assessment is based on an analysis of some possible attacks. In the second approach, a security model is developed and used to examine the proposed MAC.

Security Analysis Through Attacks Consider a MAC algorithm that produces MACs of length m using a k bit key. In general an attack might result in a *successful forgery*, or in the *recovery of the key*. According to the classification given by Preneel and van Oorschot [54], a forgery in a MAC can be

either *existential*—the opponent can construct a valid message and MAC without the knowledge of the key pair—or *selective*—the opponent can determine the MAC for a message of his choice. Protection against the former type of attack imposes more stringent conditions than the latter type of attack. A forgery is *verifiable* if the attacker can determine with a high probability whether the attack is successful. In a *chosen text attack* the attacker is given the MACs for the messages of his own choice. In an *adaptive attack* the attacker chooses text for which he can see the result of his previous request before forming his next request. In a *key recovery* attack the aim of the attacker is to find the key. If the attacker is successful, he can perform selective forgery on any message of his choice and the security of the system is totally compromised.

For an *ideal MAC* any method to find the key is as expensive as an exhaustive search of $O(2^k)$ operations. If $m < k$, the attacker may randomly choose the MAC for a message with the probability of success equal to $1/2^m$. However, in this attack the attacker cannot verify whether his attack has been successful.

The complexity of various attacks is discussed by several authors: Tsudik [78], Bakhtiari et al. [2], Bellare et al. [6]. Preneel and van Oorschot [53, 54] propose constructions resistant to such attacks. Some attacks can be applied to all MACs obtained using a specific construction method while other attacks are limited to particular instances of the method.

Formal Security Analysis

The main attempts at formalizing the security analysis of computationally secure MACs are due to Bellare et al. [5], and Bellare and Rogaway [7]. In both papers, an attack model is firstly carefully defined and the security of a MAC with respect to that model is considered. Bellare et al. [5] use their model to prove the security of a generic construction based on pseudorandom functions, while Bellare and Rogaway [7] use their model to prove the security of a generic construction based on hash functions.

MAC from Pseudorandom Functions The formal definition of security given by Bellare et al. [7] assumes that the enemy can ask the transmitter to construct tags for messages of his choice, and also ask the receiver to verify chosen message and tag pairs. The number of these requests is limited, and a limited time t can be spent on the attack. Security of the MAC is expressed as an upper bound on the enemy's chance of succeeding in its best attack.

The construction proposed by Bellare et al. applies to any pseudorandom function. Their proposal, called XOR-MAC, basically breaks a message into blocks. For each block the output of the pseudorandom function is calculated, and the outputs are finally XORed. Two schemes based on this approach are proposed: the randomized XOR scheme and the counter-based scheme.

The pseudorandom function used in the above construction can be an encryption function, like DES, or a hash function, like MD5. It is proved that the counter based scheme is more secure than the randomized scheme, and if DES is used, both schemes are more secure than CBC MAC. Some of the desirable features of this construction are *parallelizability* and *incrementality*. The former means that message blocks can be fed into the pseudorandom function in parallel. The latter refers to the feature of calculating incrementally the value of the MAC for a message s' which differs from s in only a few blocks.

MAC from Hash Functions The model used by Bellare and Rogaway [7] is similar to the above one. The enemy can obtain information by asking queries; however, in this case queries are only addressed to the transmitter.

A family of functions $\{F_k\}$ is *(ϵ, t, q, L)-secure MAC* [6] if any adversary that is not given the key k, is limited to spend total time t, and sees the values of the function F_k computed on q messages $s_1, s_2, \cdots, s_q$ of its choice, each of length at most L, cannot find a message and tag pair (s, t), $s \neq s_i$, $i = 1, \ldots q$, such that $t = F_k(s)$ with probability better that ϵ.

Two general constructions for MAC from hash functions, the so-called NMAC (the nested construction) and HMAC (the hash-based MAC) are given and their security is formally proved.

THEOREM 40.5 *[5] If the keyed compression function f is a (ϵ_f, t, q, L)-secure MAC and the keyed iterated hash function F is (ϵ_F, t, q, L)-weakly collision-resistant, then the NMAC is ($\epsilon_f + \epsilon_F$, t, q, L)-secure MAC.*

Weak collision-resistance is a much weaker notion than the collision resistance of (unkeyed) hash functions, because the enemy does not know the secret key and finding collision is much more difficult in this case. More precisely, a family of *keyed hash functions* $\{F_k\}$ *is* (ϵ, t, q, L)*-weakly collision-resistant* if any adversary that is not given the key k, is limited to spend total time t, and sees the values of the function F_k computed on q messages $m_1, m_2, \ldots, m_q$ of its choice, each of length at most L, cannot find messages m and m' for which $F_k(m) = F_k(m')$ with probability better that ϵ.

With some extra assumptions similar results are proved for the HMAC construction.

A related construction is the *collisionful keyed hash function* proposed by Gong [33]. In his construction, the collisions are selectable and the resulting function is claimed to provide security against password guessing attacks. Bakhtiari, Safavi-Naini and Pieprzyk [3, 4] explore the security of Gong's function and a key exchange protocol based on collisionful hash functions.

Applications

The main application of a MAC is to provide protection against active spoofing (see Wegman and Carter [13]). This is particularly important in open distributed systems such as the Internet. Other applications include secure password checking and software protection. MACs can be used to construct encryption functions and have been used in authentication protocols in place of encryption functions (cf. Bird et al. [12]). An important advantage of MAC functions is that they are not subject to export restrictions. Other applications of MAC functions are to protect software against viruses (cf. Radai [56]), or to protect computer files against tampering. Integrity checking is an important service in a computer operating system which can be automated with software tools.

40.6 Digital Signatures

Digital signatures are meant to be electronic equivalents of handwritten signatures. They should preserve the main features of handwritten signatures. Obviously, it is desirable that digital signatures be as legally binding as handwritten ones. There are three elements in every signature: the signer, the document, and the time of signing.

In most cases, the document already includes a timestamp. A digital signature must reflect both the content of the document and the identity of the signer. The signer is uniquely identified by its secret key. In particular, we require the signature to be

- Unique—a given signature reflects the document and can be generated by the signer only;
- Unforgeable—it must be computationally intractable for an opponent to forge the signature;
- Easy to generate by the signer and easy to verify by recipients; and
- Impossible to deny by the signer (nonrepudiation).

A digital signature differs from a handwritten signature in that it is not physically attached to the document on a piece of paper. Digital signatures have to be related both to the signer and the document by a cryptographic algorithm. Signatures can be verified by any potential recipient. Therefore, the verification algorithm must be public. Signature verification succeeds only when the signer and document match the signature.

There are two general classes of signature schemes:

- One-time signature schemes, and
- Multiple signature schemes.

One-time signature schemes can be used to sign only one message. To sign a second message, the signature scheme has to be reinitialized; however, any signature can be verified repeatedly. Multiple signature schemes can be used to sign several messages without the necessity to re-initialize the signature scheme.

In practice, a signature scheme is required to provide a relatively short signature for a document of an arbitrary length. We sign the document by generating a signature for its digest. The hashing employed to produce the digest must be secure and collision free.

One-Time Signature Schemes

This class of signature schemes can be implemented using any one-way function. These schemes were first developed using private key cryptosystems. We follow the original notation. An encryption algorithm is used as a one-way function. To set up the signature scheme, the signer chooses a one-way function (encryption algorithm). The signer selects an index k (secret key) randomly and uniformly from the set of keys, K. The index determines an instance of the one-way function, i.e., $E_k : \Sigma^n \rightarrow \Sigma^n$ where $\Sigma = \{0, 1\}$; it is known only by the signer. Note that n has to be large enough to avoid birthday attacks.

Lamport Scheme

Lamport's scheme [40] generates signatures for n-bit messages. To sign a message, the signer first chooses randomly n key pairs:

$$(K_{10}, K_{11}), (K_{20}, K_{21}), \ldots, (K_{n0}, K_{n1}) \ . \tag{40.1}$$

The pairs of keys are kept secret and are known to the signer only. Next, the signer creates two sequences, S and R:

$$\begin{aligned} S &= \{(S_{10}, S_{11}), (S_{20}, S_{21}), \ldots, (S_{n0}, S_{n1})\} , \\ R &= \{(R_{10}, R_{11}), (R_{20}, R_{21}), \ldots, (R_{n0}, R_{n1})\} . \end{aligned} \tag{40.2}$$

The elements of S are selected randomly and the elements of R are cryptograms of S so

$$R_{ij} = E_{K_{ij}}\left(S_{ij}\right) \quad \text{for} \quad i = 1, \ldots, n \quad \text{and} \quad j = 0, 1 , \tag{40.3}$$

where E_K is the encryption function of the selected symmetric cryptosystem. S and R are stored in a read-only public register; they are known by the receivers.

The signature of a n-bit message $M = (m_1, \ldots, m_n)$, $m_i \in \{0, 1\}$ for $i = 1, \ldots, n$, is a sequence of cryptographic keys,

$$S(M) = \{K_{1i_1}, K_{2i_2}, \ldots, K_{ni_n}\} \tag{40.4}$$

where $i_j = 0$ if $m_j = 0$; otherwise $i_j = 1$, $j = 1, \ldots, n$. A receiver validates the signature $S(M)$ by verifying whether suitable pairs of S and R match each other for known keys.

Rabin Scheme

In Rabin's scheme [55], a signer begins the construction of the signature by generating $2r$ keys at random:

$$K_1, K_2, \ldots, K_{2r} \ . \tag{40.5}$$

The parameter r is determined by the security requirements. The K_i are secret and known only to the signer. Next, the signer creates two sequences which are needed by the recipients to verify the signature. The first sequence,

$$S = \{S_1, S_2, \ldots, S_{2r}\}$$

comprises of binary blocks chosen at random by the signer. The second,

$$R = \{R_1, R_2, \ldots, R_{2r}\}$$

is created using the sequence S, $R_i = E_{K_i}(S_i)$ for $i = 1, \ldots, 2r$. S and R are stored in a read-only public register.

The signature is generated using the following steps. The message to be signed M is enciphered under keys $K_1, \ldots, K_{2r}$. The cryptograms

$$E_{K_1}(M), \ldots, E_{K_{2r}}(M) \tag{40.6}$$

form the signature $S(M)$. The pair $(S(M), M)$ is sent to the receivers.

To verify the signature, a receiver selects randomly a $2r$-bit sequence σ of r-ones and r-zeros. A copy of σ is forwarded to the signer. Using σ, the signer forms an r-element subset of the keys with the property that K_i belongs to the subset if and only if the ith element of the $2r$-bit sequence is '1'; $i = 1, \ldots, 2r$. The subset of keys is then communicated to the receiver. To verify the key subset, the receiver generates and compares r suitable cryptograms of S with the originals kept in the public register.

Matyas–Meyer Scheme

Matyas and Meyer [41] propose a signature scheme based on the DES algorithm. However, any one-way function can be used in the scheme.

The signer first selects a random matrix $U = [u_{i,j}]$ $i = 1, \ldots, 30$, $j = 1, \ldots, 31$ and $u_{i,j} \in \Sigma^n$. Using U, a 31×31 matrix $KEY = [k_{i,j}]$ is constructed for $k_{i,j} \in \Sigma^n$. The first row of KEY matrix is chosen at random, the other rows are

$$k_{i+1,j} = E_{k_{i,j}}\left(u_{i,j}\right)$$

for $i = 1, \ldots, 30$ and $j = 1, \ldots, 31$. Finally, the signer installs in a public registry the matrix U and the vector $(k_{31,1}, \ldots, k_{31,31})$ (the last row of KEY).

To sign a message $m \in \Sigma^n$ the cryptograms

$$c_i = E_{k_{31,i}}(m) \text{ for } i = 1, \ldots, 31$$

are computed. The cryptograms are considered as integers and ordered according to their values so $c_{i_1} < c_{i_2} < \cdots < c_{i_{31}}$. The signature of m is the sequence of keys

$$SG_k(m) = \left(k_{i_1,1}, k_{i_2,2}, \ldots, k_{i_{31},31}\right) .$$

The verifier takes the message m, recreates the cryptograms c_i and orders them in increasing order. Next, the signature-keys are put into the "empty" matrix KEY in the entries indicated by the ordered sequence of c_i's. The verifier then repeats the signer's steps and computes all keys below the keys of the signature. The signature is accepted if the last row of KEY is identical to the row stored in the registry.

Signature Schemes Based on Public-Key Cryptosystems

RSA Signature Scheme

First, a signer sets up the RSA system [59] with the modulus $N = p \times q$, where the two primes p and q are fixed. Next a random decryption key $d \in Z_N$ is chosen; the encryption key e is

$$e \times d \equiv 1 \pmod{(p-1, q-1)} .$$

The signer publishes both the modulus N and the decryption key d.

Given a message $M \in Z_N$, the signature generated by the signer is

$$S \equiv M^e \pmod{N} .$$

Since the decryption key is public, anyone can verify whether

$$M \equiv S^d \pmod{N} .$$

The signature is considered to be valid if this congruence is satisfied. RSA signatures are subject to various attacks which exploit the commutativity of exponentiation.

ElGamal Signature Scheme

The signature scheme due to ElGamal [27] is based on the discrete logarithm problem. The signer chooses a finite field $GF(p)$ for a sufficiently large prime p. A primitive element $g \in GF(p)$ and a random integer $r \in GF(p)$ are fixed. Next the signer computes

$$K \equiv g^r (\bmod p)$$

and announces K, g and p. To sign a message $M \in GF(p)$, the signer selects a random integer $R \in GF(p)$ such that gcd $(R, p-1) = 1$ and calculates

$$X \equiv g^R (\bmod p) .$$

Using this data following congruence is solved

$$M \equiv r \times X + R \times Y (\bmod p - 1)$$

for Y using Euclid's algorithm. The signature of M is the triple (M, X, Y). Note that r and R are kept secret by the signer. The recipient of the signed message forms

$$A \equiv K^X X^Y (\bmod p)$$

and accepts the message M as authentic if $A \equiv g^M \pmod p$. It is worth noting that knowledge of the pair (X, Y) does not reveal the message M. As a matter of fact, there are many pairs matching the message—for every pair (r, R) there is a pair (X, Y).

Since discrete exponent systems can be based on any cyclic group, the ElGamal signature scheme can be extended to this setting. A modification of ElGamal's signature was proposed as a Digital Signature Standard (DSS) in 1991 [45].

Special Signatures

Sometimes additional conditions are imposed upon digital signatures. Blind signatures are useful in situations where the message to be signed should not be revealed to the signer. Unlike typical digital signatures, the undeniable versions require the participation of the signer in order to verify the signature. Fail-stop signatures are used whenever there is a need for protection against a very powerful adversary. As these signatures require interactions amongst the parties involved, the signatures are sometimes called *signature protocols.*

Blind Signatures

The concept of blind signatures was introduced by Chaum [18]. They are applicable to situations where the holder of a message M needs to get M signed by a signer (which could be a public registry) without revealing the message. This can be done with the following steps.

- The holder of the message first encrypts it.
- The holder sends a cryptogram of the message to the signer.
- The signer generates the signature for the cryptogram and sends it back to the holder.
- The holder decodes the encryption and obtains the signature of the message.

This scheme works if the encryption and signature operations commute, for example, the RSA scheme can be used to implement blind signatures.

Assume that the signer has set up a RSA signature scheme with modulus N and public decryption key d. The holder of the message M selects at random an integer $k \in Z_N$ and computes the cryptogram

$$C \equiv M \times k^d (\bmod N) .$$

The cryptogram C is now sent to the signer who computes the blind signature

$$S_C \equiv \left(M \times k^d\right)^e \ (\mathrm{mod} N) \ .$$

The blind signature S_C is returned to the holder who computes the signature for M as follows:

$$S_M \equiv S_C \times k^{-1} \equiv M^e (\mathrm{mod}\ N) \ .$$

It is not necessary to have special signature schemes to generate blind signatures. It is enough for the holder of the message to use a secure hash function $h()$. To get a (blind) signature from the signer, the holder first compresses the message M using $h()$. The digest $D = h(M)$ is sent to the signer. After signing the digest, the signature $SG_k(D)$ is communicated to the holder who attaches the message M to the signature $SG_k(D)$. Note that the signer cannot recover the message M from the digest, since $h()$ is a one-way hash function. Also the holder cannot cheat by attaching a "false" message unless collisions for the hash function can be found.

Undeniable Signatures

The concept of undeniable signatures is due to Chaum and van Antwerpen [19]. The characteristic feature of undeniable signatures is that signatures cannot be verified without the cooperation of the signer. Assume we have selected a large prime p and a primitive element $g \in GF(p)$. Both p and g are public. The signer randomly selects its secret $k \in GF(p)$ and announces $g^k \pmod p$. For a message M, the signer creates the signature

$$S \equiv M^k (\mathrm{mod}\ p) \ .$$

Verification needs the cooperation of the verifier and signer, and proceeds as follows.

- The verifier selects two random numbers $a, b \in GF(p)$ and sends $C \equiv S^a (g^k)^b \pmod p$ to the signer.
- The signer computes k^{-1} such that $k \times k^{-1} \equiv 1 \pmod{p-1}$ and returns $d = C^{k^{-1}} \equiv M^a \times g^b \pmod p$ to the verifier.
- The verifier accepts or rejects the signature as genuine depending on whether $d \equiv M^a \times g^b \pmod p$.

There are two possible ways in which a verification can fail. Either the signer has tried to disavow a genuine signature or the signature is indeed false. The first possibility is prevented by incorporating a disavowal protocol. The protocol requires two runs for verification. In the first run, the verifier randomly selects two integers $a_1, b_1 \in GF(p)$ and sends $C_1 \equiv S^{a_1}(g^k)^{b_1} \pmod p$ to the signer. The signer returns $d_1 = C_1^{k^{-1}}$ to the verifier. The verifier checks whether

$$d_1 \neq M^{a_1} \times g^{b_1} (\mathrm{mod}\ p) \ .$$

If the congruence is not satisfied, the verifier repeats the process using a different pair $a_2, b_2 \in GF(p)$. The verifier concludes that S is a forgery if and only if

$$\left(d_1 g^{-b_1}\right)^{a_2} \equiv \left(d_2 g^{-b_2}\right)^{a_1} (\mathrm{mod}\ p) \ ;$$

otherwise, the signer is cheating.

Fail-Stop Signatures

The concept of fail-stop signatures was introduced by Pfitzmann and Waidner [48]. Fail-stop signatures protect signatures against a powerful adversary. As usual the signature is produced by a signer who holds a particular secret key. There are, however, many other keys which can be used to produce the same signature and which thus work with the original public key. There is a high probability that the key chosen by the adversary differs from the key held by the signer. Fail-stop signatures provide signing and verification algorithms as well as an algorithm to detect forgery.

Let k be a secret key known to the signer only and K be the public key. The signature on a message M is denoted as $s = SG_k(M)$. A fail-stop signature must satisfy the following conditions:

- An opponent with unlimited computational power can forge a signature with a negligible probability. More precisely, an opponent who knows the pair $(s = SG_k(M), M)$ and the signer's public key K, can create a collection of all keys $K_{s,M}$ such that $k^* \in K_{s,M}$ if and only if $s = SG_{k^*}(M) = SG_k(M)$. The size of $\mathcal{K}_{s,M}$ has to increase exponentially as a function of the security parameter n. Not knowing the secret k, the opponent can only randomly choose an element from $K_{s,M}$. Let this element be k^*. If the opponent signs another message $M^* \neq M$, it is a requirement that $s^* = SG_{k^*}(M^*) \neq SG_k(M^*)$ with a probability close to one.
- There is a polynomial-time algorithm that produces a proof of forgery as output, when given the following inputs: a secret key k, a public key K, a message M, a valid signature s, and a forged signature s^*.
- A signer with polynomially bounded computing power cannot construct a valid signature that it can later deny by proving it to be a forgery.

Clearly, after the signer has provided a proof of forgery, the scheme is compromised and is no longer used. This is why it is called "fail-stop."

40.7 Research Issues and Summary

In this chapter we discussed authentication, hashing, message authentication codes (MACs), and digital signatures. We have presented the fundamental ideas underlying each topic and indicated the current research developments in these topics. This is reflected in our list of references.

We shall now summarize the topics covered in this chapter.

Authentication deals with the problem of providing assurance to a receiver that a communicated message originates from a particular transmitter, and that the received message has the same content as the transmitted message. A typical and widely used application of authentication occurs in computer networks. Here the problem is to provide a protocol to establish the identity of two parties wishing to communicate or make transactions via the network. Motivated by such applications, the theory of authentication codes has developed into a mature area of research, drawing from several areas of mathematics.

Hashing algorithms provide a relatively short digest of a much longer input. Hashing must satisfy the critical requirement that the digests of two distinct messages are distinct. A widely used type of hashing functions are constructed from block encryption ciphers. They have numerous applications in cryptology. Algebraic methods have also been proposed as a source of good hashing functions. These offer some provable security properties.

Message authentication codes or (MACs) are symmetric key primitives providing message integrity against active spoofing, by appending a cryptographic checksum to a message that is verifiable only by the intended recipient of the message. Message authentication is one of the most important ways of ensuring the integrity of information communicated by electronic means. This is especially relevant in the rapidly developing sphere of electronic commerce.

Digital signatures are the electronic equivalents of handwritten signatures. They are designed so as to preserve the essential features of handwritten signatures. They can be used to sign electronic documents and have potential application in legal contexts.

40.8 Defining Terms

Authentication: One of the main two goals of cryptography (the other is secrecy). An authentication system ensures that messages transmitted over a communication channel are authentic.

Cryptology: The art/science of design and analysis of cryptographic systems.

Digital signatures: An asymmetric cryptographic primitive that is the digital counterpart of a signature and links a document to a unique person.

Encryption algorithm: Transforms an input text by "mixing" it with a randomly chosen bit string—the key—to produce the cipher text. In a symmetric encryption algorithm, the plain text can be recovered by applying the key to the cipher text.

Hashing: Hashing is accomplished by applying a function to an arbitrary length message to create a digest/hash value, which is usually of fixed length.

Key: An input provided by the user of a cryptographic system. This piece of information is kept secret and is the source of security in a cryptographic system. Sometimes a part of key information is made public, in which case the secret part is the source of security.

Message authentication codes: A symmetric cryptographic primitive that is used for providing authenticity.

Plain text, cipher text: The cipher text is the "scrambled" version of an original source—the plain text. It is assumed that the scrambled text, produced by an encryption algorithm, can be inspected by persons not having the key and not reveal the content of the source.

Acknowledgments

We thank Anish Mathuria for all his comments and suggestions, which have greatly helped us improve our exposition.

References

[1] Bakhtiari, S., Safavi-Naini, R., and Pieprzyk, J., Keyed hash functions, *Cryptography: Policy and Algorithms Conference,* LNCS Vol. 1029, 201–214, Springer-Verlag, Berlin, 1995.

[2] Bakhtiari, S., Safavi-Naini, R., and Pieprzyk, J., Practical and secure message authentication, *Proc. Second Annual Workshop on Selected Areas in Cryptography (SAC'95),* 55–68, Ottawa, Canada, May 1995.

[3] Bakhtiari, S., Safavi-Naini, R., and Pieprzyk, J., On selectable collisionful hash functions, *Proc. Australasian Conference on Information Security and Privacy,* LNCS Vol. 1172, 287–294, Springer-Verlag, Berlin, 1996.

[4] Bakhtiari, S., Safavi-Naini, R., and Pieprzyk, J., Password-based authenticated key exchange using collisionful hash functions, *Proc. Australasian Conference on Information Security and Privacy,* LNCS Vol. 1172, 299–310, Springer-Verlag, Berlin, 1996.

[5] Bellare, M., Canetti, R., and Krawczyk, H., Keying hash functions for message authentication, *Proc. Crypto'96,* LNCS Vol. 110, 1–15, Springer-Verlag, Berlin, 1996.

[6] Bellare, M., Kilian, J., and Rogaway, P., The security of cipher block chaining, *Proc. Crypto'94,* LNCS Vol. 839, 348–358, Springer-Verlag, Berlin, 1994.

[7] Bellare, M. and Rogaway, P., The exact security of digital signatures—how to sign with RSA and Rabin. *Proc. Eurocrypt'96*, LNCS Vol. 1070, 399–416, Springer-Verlag, Berlin, May 1996. 1987,

[8] Berson, T.A., Differential cryptanalysis mod 2^{32} with applications to MD5, *Proc. Eurocrypt'92,* LNCS, Vol. 658, 71–80, Springer-Verlag, Berlin, 1993.
[9] Berson, T.A., Gong, L., and Lomas, T.M.A., Secure, keyed, and collisionful hash functions, TR SRI-CSL-94-08, *SRI International,* Dec. 1993. Revised version (Sept. 2, 1994).
[10] Beth, T., Jungnickel, D., and Lenz, H., *Design Theory,* Cambridge University Press, Cambridge, 1986.
[11] Biham, E. and Shamir, A., Differential cryptanalysis of DES-like Cryptosystems, *Journal of Cryptology,* 4, 3–72, 1991.
[12] Bird, R., Gopal, I., Herzberg, A., Janson, P., Kutten, S., Molva, R., and Yung, M., The KryptoKnight family of light-weight protocols for authentication and key distribution, *IEEE/ACM Transactions on Networking,* 3, 31–41, 1995.
[13] Carter, J.L. and Wegman, M.N., Universal class of hash functions, *Journal of Computer and System Sciences,* 18(2), 143–154, 1979.
[14] Charnes, C. and Pieprzyk, J., Linear nonequivalence versus nonlinearity, *Proc. Auscrypt'92,* LNCS Vol. 718, 156–164, Springer-Verlag, Berlin, 1993.
[15] Charnes, C. and Pieprzyk, J., Attacking the SL_2 Hashing scheme, *Proc. Asiacrypt'94,* LNCS Vol. 917, 322–330, Springer-Verlag, Berlin, 1995.
[16] C. Charnes, L. O'Connor, J. Pieprzyk, Safavi-Naini, R. and Zheng, Y., Comments on GOST encryption algorithm, *Proc. Eurocrypt'94,* LNCS Vol. 950, 433–438, Springer-Verlag, Berlin, 1995.
[17] Charnes, C. and Pieprzyk, J., Weak parameters for the SL_2 hashing function, Manuscript, 1996.
[18] Chaum, D., Blind signatures for untraceable payments, *Proc. Crypto 82,* 199–203, Plenum Press, New York, 1983.
[19] Chaum, D. and Van Antwerpen, H., Undeniable signatures, *Proc. Crypto 89,* LNCS Vol. 435, 212–217, Springer-Verlag, Berlin, 1990.
[20] Coppersmith, D., Analysis of ISO/CCITT Document X.509 Annex D. Internal Memo, IBM T. J. Watson Center, June 11, 1989.
[21] Daemen, J., Govaerts, R., and Vandewalle, J., A framework for the design of one-way hash functions including cryptanalysis of Damgard one-way function based on a cellular automaton, *Proc. Asiacrypt'91,* LNCS Vol. 739, 82–97, Springer-Verlag, Berlin, 1993.
[22] Damgård, I., A design principle for hash functions, *Proc. Crypto'89,* LNCS Vol. 435, 416–427, Springer-Verlag, Berlin, 1990.
[23] Davies, D.W. and Price, W.L., The application of digital signatures based on public-key cryptosystems, *Proc. Fifth Int. Computer Communications Conference,* 525–530, Oct. 1980.
[24] De Santis, A. and Yung, M., On the design of provably-secure cryptographic hash functions, *Proc. Eurocrypt'90,* LNCS Vol. 473, 377–397, Springer-Verlag, Berlin, 1990.
[25] Desmedt, Y. and Frankel, Y., Shared generation of authenticators and signatures, *Proc. Crypto'91,* LNCS, Vol. 576, 457–469, Springer-Verlag, Berlin, 1992.
[26] Dobbertin, H., Cryptanalysis of MD4, *Proc. Fast Software Encryption Workshop,* LNCS Vol. 1039, 71–82, Springer-Verlag, Berlin, 1996.
[27] El Gamal, T., A public key cryptosystem and a signature scheme based on discrete logarithms, *IEEE Transactions on Information Theory,* 31, 469–472, 1985.
[28] Fak, V., Repeated use of codes which detect deception, *IEEE Transactions on Information Theory,* 25(2), 233–234, Mar. 1979.
[29] Gehrmann, C., van Dijk, M., and Smeets, B., Unconditionally secure group authentication, Submitted for publication.
[30] Geiselmann, W., A note on the hash function of Tillich and Zémor, *Cryptography and Coding,* LNCS Vol. 1025, 257–263, Springer-Verlag, Berlin, 1995.
[31] Gibson, J.K., Discrete logarithm hash function that is collision free and one way, *IEE Proc.-E,* 138(6), 407–427, 1991.

[32] MacWilliams, F.J., Gilbert, E.N., and Sloane, N.J.A., Codes which detect deception, *Bell System Technical Journal,* 53(3), 405–424, 1974.
[33] Gong, L., Collisionful keyed hash functions with selectable collisions, *Information Processing Letters,* 55, 167–170, 1995.
[34] Impagliazzo, R. and Naor, M., Efficient cryptographic schemes as provably secure as subset sum, *Proc. 30th IEEE Symposium on Foundations of Computer Science,* 236–241, 1989.
[35] Johansson, T., Lower bound on the probability of deception in authentication with arbitration, *IEEE Transaction on Information Theory,* 40, 1573–1585, 1994.
[36] Johansson, T., Authentication codes for nontrusting parties obtained from rank metric codes, *Designs, Codes and Cryptography,* 6, 205–218, 1995.
[37] Johansson, T., Kabatianskii, G., and Smeets, B., On the relation between *A*-codes and codes correcting independent errors, *Proc. Eurocrypt'93,* LNCS Vol. 765, 1-11, Springer-Verlag, Berlin, 1994.
[38] Krawczyk, H., LFSR-based hashing and authentication, *Proc. Crypto'94,* LNCS Vol. 839, 129–139, Springer-Verlag, Berlin, 1994.
[39] Kurosawa, K., New bounds on authentication code with arbitration, *Proc. Crypto'94,* LNCS Vol. 839, 140–149, Springer-Verlag, Berlin, 1994.
[40] Lamport, L., Constructing digital signatures from a one-way function, TR CSL-98, SRI International, Oct. 1979.
[41] Matyas, S.M. and Meyer, C.H., Electronic signature for data encryption standard, *IBM Tech. Disc. Bull.,* 24(5), 1981.
[42] Merkle, R.C., A fast software one-way hash function, *Journal of Cryptology,* 3(1), 43–58, 1989.
[43] Merkle, R.C., One way hash functions and DES, *Proc. Crypto'89,* LNCS Vol. 435, 428–446, Springer-Verlag, Berlin, 1990.
[44] Naor, M. and Yung, M., Universal one-way hash functions and their Cryptographic applications, *Proc.* 21*st ACM Symposium on Theory of Computing,* 33–43, Seattle, 1989.
[45] National Institute for Standards and Technology. Digital Signature Standard (DSS), *Federal Register*. 56(169), Aug. 30 1991.
[46] Ohta, K. and Koyama, K., Meet-in-the-middle attack on digital signature schemes, *Proc. Auscrypt'90,* LNCS Vol. 453, 110–121, Springer-Verlag, Berlin, 1990.
[47] Pei, D., Information theoretic bounds for authentication codes and PBIB, *Presented at the Rump session of Asiacrypt'91,*
[48] Pfitzmann, B. and Waidner, M., Fail-stop signatures and their applications, *Proc. Securicom'91,* 338–350, 1991.
[49] Pieprzyk, J. and Sadeghiyan, B., *Design of Hashing Algorithms,* LNCS Vol. 756, Springer-Verlag, New York, 1993.
[50] Preneel, B., *Analysis and Design of Cryptographic Hash Functions,* Ph.D. thesis, Katholieke Universiteit, Leuven, 1993.
[51] Piper, F. and Beker, H., *Cipher Systems,* Northwood Books, London, 1982.
[52] Preneel, B., Chaum, D., Fumy, W., Jansen, C.J.A., Landrock, P., and Roelofsen, G., Race integrity primitives evaluation (RIPE): A Status Report, *Proc. Eurocrypt'91,* LNCS Vol. 547, 547–551, Springer-Verlag, Berlin, 1991.
[53] Preneel, B. and van Oorschot, P.C., MDx-MAC and building fast MACs from hash functions, *Proc. Eurocrypt'96,* LNCS Vol. 1070, 1–14, Springer-Verlag, Berlin, 1996.
[54] Preneel, B. and van Oorschot, P.C., On the security of two MAC Algorithms, *Proc. Eurocrypt'96,* LNCS Vol. 1070, 19–32, Springer-Verlag, Berlin, 1996.
[55] Rabin, M.O., Digitalized signatures, *Foundations of Secure Computation,* 155–168, Academic Press, 1978.
[56] Radai, Y., Checksumming techniques for anti-viral purposes, *International Virus Bulletin Conference,* Sept. 1991.

[57] Rivest, R.L., *RFC* 1321*: The MD5 message-digest algorithm,* Internet Activities Board, Apr. 1992.
[58] Rivest, R.L., The MD4 message digest algorithm, *Proc. Crypto'90,* LNCS Vol. 537, 303–311, Springer-Verlag, Berlin, 1991.
[59] Rivest, R.L., Shamir, A., and Adleman, L.M., A method for obtaining digital signatures and public-key cryptosystems, *Communications of the ACM,* 21(2), 120–126, 1978.
[60] Rees, R.S. and Stinson, D.R., Combinatorial characterization of authentication codes II, preprint.
[61] Rogaway, P., Bucket hashing and its application to fast message authentication, *Proc. Crypto'95*, LNCS Vol. 963, 30–42, Springer-Verlag, Berlin, 1995.
[62] Rompel, J., One-way functions are necessary and sufficient for secure signatures. *Proc.* 22*nd ACM Symposium on Theory of Computing,* 387–394, Baltimore, MD, 1990.
[63] Rosenbaum, V., A lower bound on authentication after having observed a sequence of messages, *Journal of Cryptology,* 6(3), 135–156, 1993.
[64] Safavi-Naini, R., Three systems for shared generation of authenticators, *Proc. Cocoon'96,* LNCS Vol. 1090, 401–411, Springer-Verlag, Berlin, 1996.
[65] Safavi-Naini, R. and Martin, K., Unconditionally secure authentication systems with shared generation of authenticators. Submitted for publication.
[66] Simmons, G.J., A game theory model of digital message authentication, *Congressus Numerantium,* 34, 413–424, 1982.
[67] Simmons, G.J., A survey of information authentication, *Contemporary Cryptology: The Science of Information Integrity,* IEEE Press, 379–419, 1992.
[68] Stinson, D.R., A provably secure hash function equivalent to the discrete logarithm problem, TR CCIS Lincoln, Mar. 1992.
[69] Stinson, D.R., Combinatorial characterisation of authentication codes, *Proc. Crypto'91,* LNCS, Vol. 576, 62–73, Springer-Verlag, Berlin, 1991.
[70] Stinson, D.R., Combinatorial techniques for universal hashing, *Journal of Computer and System Sciences,* 48, 337–346, 1994.
[71] Stinson, D.R., The combinatorics of authentication and secrecy codes, *Journal of Cryptology,* 2(1), 23–49, 1990.
[72] Stinson, D.R., Universal hashing and authentication codes, *Designs, Codes and Cryptography,* 4, 369–380, 1994.
[73] Taylor, R., Near optimal unconditionally secure authentication, *Proc. Eurocrypt'94,* LNCS, Vol. 950, 245–255, Springer-Verlag, Berlin, 1995.
[74] Tillich, J-P. and Zémor, G., Hashing with SL_2. *Proc. Crypto'94,* LNCS Vol. 839, 40–49, Springer-Verlag, Berlin 1994.
[75] Tombak, L. and Safavi-Naini, R., Authentication codes that are r-fold secure against spoofing, *Proc.* 2*nd ACM Conference on Computer and Communication Security,* 166–169, 1994.
[76] Tombak, L. and Safavi-Naini, R., Authentication codes in plaintext and content-chosen attacks, *Designs, Codes and Cryptography,* 6, 83–99, 1995.
[77] Tombak, L. and Safavi-Naini, R., Combinatorial characterization of A-codes with r-fold security, *Proc. Asiacrypt'94,* LNCS Vol. 917, 211–223, Springer-Verlag, Berlin, 1995.
[78] Tsudik, G., Message authentication with one-way hash functions, *IEEE Infocom'92*, 2055–2059, May 1992.
[79] Walker, M., Information theoretic bounds for authentication schemes, *Journal of Cryptology,* 2(3), 133–138, 1990.
[80] Wegman, M.N. and Carter, J.L., New hash functions and their use in authentication and set equality, *Journal of Computer and System Sciences,* 22, 265–279, 1981.
[81] Winternitz, R.S., Producing a one-way hash function from DES, *Proc. Crypto'83,* Plenum Press, New York, 203–207, 1984.

[82] Vanroose, P., Smeets, B., and Wan, Z.-X., On the construction of authentication codes with secrecy and codes withstanding spoofing attacks of order $L \geq 2$, *Proc. Eurocrypt'90,* LNCS Vol. 473, 306–312, Springer-Verlag, Berlin, 1990.
[83] Yung, M. and Desmedt, Y., Arbitrated unconditionally secure authentication can be unconditionally protected against arbiter's attack, *Proc. Crypto'90,* LNCS Vol. 537, 177–188, Springer-Verlag, Berlin, 1990.
[84] Zheng, Y., Matsumoto, T., and Imai, H., Structural properties of one-way hash functions, *Proc. Crypto'90,* LNCS Vol. 537, 285–302, Springer-Verlag, Berlin, 1991.
[85] Zheng, Y., Pieprzyk, J., and Seberry, J., HAVAL - a one-way hashing algorithm with variable length of output, *Proc. Auscrypt'92,* LNCS Vol. 718, 83–104, Springer-Verlag, Berlin, 1993.

Further Information

Current research in cryptology is represented in the proceedings of the conferences CRYPTO, EUROCRYPT, ASIACRYPT, AUSCRYPT. There are also more specialized conferences dealing with topics such as hashing, fast software encryption, and security. The proceedings are published by Springer in their LNCS series. *The Journal of Cryptology, IEEE Proceedings on Information Theory, Designs Codes and Cryptography,* and several other journals publish extended versions of the articles that were presented in the above-mentioned conferences.

41

Crypto Topics and Applications II

Jennifer Seberry,
Chris Charnes,
Josef Pieprzyk, and
Rei Safavi-Naini
University of Wollongong

41.1 Introduction

In this chapter we continue our exposition of the crypto topics that was begun in the previous chapter. This chapter covers secret sharing, threshold cryptography, signature schemes, and finally quantum key distribution and quantum cryptography. As in the previous chapter, we have focused only on the essentials of each topic. We have included in the reference list sufficient items that can be consulted for further details.

First we give a synopsis of the topics that are discussed in this chapter.

Secret sharing is concerned with the problem of how to distribute a secret among a group of participating individuals, or entities, so that only predesignated *collections* of individuals are able to recreate the secret by collectively combining the parts of the secret that were allocated to them. There are numerous applications of secret sharing schemes in practice. One example of secret sharing occurs in banking. For instance, the combination to a vault may be distributed in such a way that only specified collections of employees can open the vault by pooling their portions of the combination. In this way the authority to initiate an action, e.g., the opening of a bank vault, is divided for the purposes of providing security and for added functionality such as auditing if required.

Threshold cryptography is a relatively recently studied area of cryptography. It deals with situations where the authority to initiate or perform cryptographic operations is distributed amongst a group of individuals. Many of the standard operations of single-user cryptography have counterparts in threshold cryptography.

Signature schemes deal with the problem of generating and verifying (electronic) signatures for documents. A subclass of signature schemes is concerned with the shared-generation and shared-verification of signatures, where a collaborating group of individuals is required to perform these actions.

A new paradigm of security has recently been introduced into cryptography with the emergence of the ideas of quantum key distribution and quantum cryptography. While classical cryptography employs various mathematical techniques to restrict eavesdroppers from learning the contents of encrypted messages, in quantum cryptography the information is protected by the laws of physics.

41.2 Secret Sharing

Introduction

Secret sharing is concerned with the problem of distributing a secret among a group of participating individuals, or entities, so that only predesignated collections of individuals are able to recreate the secret by collectively combining their *shares* of the secret.

The earliest and the most widely studied type of secret sharing schemes are called (t, n)-*threshold schemes*. In these schemes the *access structure*—a specification of the participants authorized to recreate the secret—comprises all the possible t-element subsets selected from a n-element set.

The problem of *realizing*, i.e., implementing secret sharing schemes for threshold structures was solved independently by Blakley [12] and Shamir [75] in 1979. Shamir's solution is based on the property of polynomial interpolation in finite fields; Blakley formulated and solved the problem in terms of finite geometries.

In a (t, n)-threshold scheme, each of the n participants holds some *shares* (also called *shadows*) of the secret. The parameter $t \leq n$ is called the *threshold* value. A fundamental property of a (t, n)-threshold scheme is that the secret can only be recreated if at least t shareholders combine their shares, but less than t shareholders cannot recreate the secret. The fact that the key can be recovered from the combined shares of any t-sized subset is a property which makes threshold schemes very useful in key management. Threshold schemes tolerate the invalidation of up to $n - t$ shares—the secret can still be recreated from the remaining intact shares.

Secret sharing schemes are also used to control the authority to perform critical actions. For example, a bank vault can be opened only if say, any two out of three trusted employees of the bank agree to do so by combining their partial knowledge of the vault combination. In this case, even if any one of the three employees is not present at any given time the vault can still be opened, and no single employee has sufficient information about the combination to open the vault.

Secret sharing schemes that do not reveal any information about the shared secret to unauthorized individuals are called *perfect*. This notion will be formally defined in "Models of Secret Sharing." In this survey we discuss both perfect and *nonperfect* schemes, as the latter schemes are proving to be useful in various secret sharing applications.

Besides the (t, n)-threshold structures, more general access structures are encountered in the theory of secret sharing. These will be considered in "General Access Structures." General access structures apply to situations where the trust-status of the participants is not uniform. For example, in the bank scenario described earlier, it might be considered more secure to authorize either the bank manager, or any two out of three senior employees to open the vault.

Since Blakley's and Shamir's papers have appeared, the study of secret sharing has developed into an active area of research in cryptography. The fundamental problem of the theory and practice of secret sharing deals with the issue of how to implement secret sharing schemes for arbitrary access structures.

We shall describe later some of the solutions to this problem. Simmons [79] gives numerous examples of practical situations which require secret sharing schemes. He also gives a detailed account of the geometric approach to secret sharing. Stinson's [84] survey is broader and more condensed.

Simmons [78] discusses secret sharing schemes with extended capabilities. He argues that there are realistic applications in which schemes with extended capabilities are required. We assume there exists a key distribution center (KDC) that is trusted unconditionally.

Models of Secret Sharing

A common model of secret sharing has two phases. In the *initialization phase*, a trusted entity—the *dealer*—distributes shares of a secret to the participants via secure means. In the *reconstruction phase* the authorized participants submit their shares to a *combiner*, who reconstructs the secret on their behalf. It is assumed that the combiner is an algorithm which only performs the task of reconstructing the secret. We denote the sets of all possible secrets and shares by $\mathcal{K}$ and $\mathcal{S}$ respectively; the set of participants in a scheme is denoted by $\mathcal{P}$. Secret sharing schemes can be modeled using the information theory concept of *entropy* (cf. [45]). This approach was initiated by Karnin, Greene and Hellman [54] and developed further by Capocelli et al. [23].

DEFINITION 41.1 A secret sharing scheme is a collection of two algorithms. The first (the dealer) is a probabilistic mapping

$$\mathcal{D} : \mathcal{K} \to \mathcal{S}_1 \times \mathcal{S}_2 \times \ldots \times \mathcal{S}_n$$

where $\mathcal{S}_i \subset \mathcal{S}$ $(i = 1, 2, \ldots, n)$ and $\mathcal{S}_i$ is a subset of shares which is used to generate a share for the participant $P_i \in \mathcal{P}$. The second (the combiner) is a function

$$\mathcal{C} : \mathcal{S}_{i_1} \times \mathcal{S}_{i_2} \times \ldots \times \mathcal{S}_{i_t} \to \mathcal{K}$$

such that if the corresponding subset of participants $\{P_{i_1}, P_{i_2}, \ldots, P_{i_t}\}$ belongs to the access structure Γ, it produces the secret $K \in \mathcal{K}$, i.e.,

$$H\left(K \mid P_{i_1}, P_{i_2}, \ldots, P_{i_t}\right) = 0\,. \tag{41.1}$$

The combiner fails to recompute the secret if the subset of participants does not belong to the access structure Γ, i.e.,

$$H\left(K \mid S_l\right) \geq 0 \tag{41.2}$$

for $S_l = \{s_{i_1}, s_{i_2}, \ldots, s_{i_l}\}$ and $S_l \notin \Gamma$.

In Eq. (41.1), $H(K \mid P_{i_1}, P_{i_2}, \ldots, P_{i_t})$ is calculated with respect to the shares of the participants. A secret sharing scheme is called *perfect* if $H(K \mid S_l) = H(K)$ for any unauthorized subset of participants, i.e., not belonging to an access structure Γ (cf. "General Access Structures").

The following result is proved by Karnin et al. [54].

THEOREM 41.1 *A necessary condition for a perfect threshold scheme is that for each share s_i, the inequality $H(s_i) \geq H(K)$ holds.*

Most of the secret sharing schemes which we discuss satisfy this inequality, but we will also consider in the section on "Nonperfect Schemes" schemes that do not satisfy this inequality; these are called *nonperfect* schemes.

The Matrix Model

A matrix representation of perfect secret sharing schemes was introduced by Brickell and Stinson [21]. The matrix model is often used in theoretical investigations of secret sharing. In this model a perfect secret sharing scheme is formulated as a matrix M that is known by all the participants $\mathcal{P}$ in the scheme. The $|\mathcal{P}| + 1$ columns of M are indexed as follows. The first column corresponds to the dealer $\mathcal{D}$, the remaining columns are indexed by the remaining participants in $\mathcal{P}$. Each row of M contains one of the possible keys K which is to be shared in column $\mathcal{D}$, and the shares of K are located in the remaining columns. When the dealer wants share K, a row r which has K in the $\mathcal{D}$-column is chosen uniformly and randomly. The dealer distributes the shares of K to each participant using the matrix M, i.e., participant P_j receives the entry $M_{r,j}$ as his share.

The general requirements of a perfect secret scheme translate into the following combinatorial conditions in the matrix model, cf. Stinson [84], and Blundo et al. [15]. Suppose that Γ is an access structure.

1. If $\mathcal{B} \in \Gamma$ and $M(r, P) = M(r', P)$ for all $P \in \mathcal{B}$, then $M(r, \mathcal{D}) = M(r', \mathcal{D})$.
2. If $\mathcal{B} \notin \Gamma$, then for every possible assignment f of shares to the participants in $\mathcal{B}$, say $f = (f_P : P \in \mathcal{B})$, a nonnegative integer $\lambda(f, \mathcal{B})$ exists such that

$$|\{r : M(r, P) = f_P \ \forall P \in \mathcal{B},\ M(r, D) = K\}| = \lambda(f, \mathcal{B})$$

is independent of the value of K.

Information Rate

The *information rate* of secret sharing schemes was studied by Brickell and Stinson [21]. It is a measure of the amount of information that the participants need to keep secret in a secret sharing scheme. The *information rate* of a participant P_i in a secret sharing scheme with $|S_i|$ shares is

$$\rho_i = \frac{\log_2 |\mathcal{K}|}{\log_2 |S_i|}.$$

The information rate of the scheme, denoted ρ, is defined to be the minimum of the ρ_i.

A proof of the fact that $\rho \leq 1$ is given by Stinson [84]. This result motivates the definition of *ideal* secret sharing schemes.

DEFINITION 41.2 A perfect secret sharing scheme is called *ideal* if $\rho = 1$; that is, if the size of each participants share, measured in the number of bits, equals the size of the secret.

We now define another measure used to quantify the comparison between secret sharing schemes (cf. "General Access Structures").

DEFINITION 41.3 $\rho^*(\Gamma)$ is the maximum value of ρ for any perfect secret sharing scheme realizing the access structure Γ.

For any access structure it is desirable to implement a secret sharing scheme with information rate close to 1. This minimizes the amount of information that needs to be kept secret by the participants, which means that there is a greater chance of the scheme remaining secure. For example, a (t, n)-threshold scheme implemented as in Shamir's method is ideal, but when the scheme is modified to prevent *cheating* as proposed by Tompa and Woll [87], it is no longer ideal (cf. "The Problem of Cheaters").

Some Known Schemes

We now describe several well-known threshold secret sharing schemes.

Blakley's Scheme

Blakley [12] implements threshold schemes using *projective spaces* over finite fields $GF(q)$. A projective space $PG(t, q)$ is defined from the corresponding $t+1$-dimensional vector space $V(t+1, q)$ by omitting the zero vector of $V(t+1, q)$ and identifying two vectors v and v' satisfying the relation $v = \lambda v'$, where λ is a nonzero element of $GF(q)$. This defines an equivalence relation on $V(t+1, q)$. The set of equivalence classes, i.e., the lines through the origin of $V(t+1, q)$, are the points of $PG(t, q)$; there are $(q^t - 1)/(q - 1)$ such points. Similarly, each k-dimensional subspace of $V(t+1, q)$ corresponds to a $(k-1)$-dimensional subspace of $PG(t, q)$. Every point of $PG(t, q)$ lies on $(q^t - 1)/(q - 1)$ $(t-1)$-dimensional subspaces which are called the *hyperplanes* of $PG(t, q)$.

To realize a (t, n)-threshold scheme, the secret is represented by a point p chosen randomly from $PG(t, q)$; each point p belongs to $(q^t - 1)/(q - 1)$ hyperplanes. The shares of the secret are the n hyperplanes, which are randomly selected and distributed to the participants. If q is sufficiently large and n is not too large, then the probability that any t of the hyperplanes intersect in some point other than p is close to zero; cf. Blakley [12]. Thus generally the secret can be recovered from any t of the n shares. The secret cannot be recovered from the knowledge of less than t hyperplanes, as these will intersect only in some subspace containing p. This scheme is not perfect, since a coalition of unauthorized insider participants has a greater chance of guessing the secret than an unauthorized group of outsider participants.

Blakley's geometric solution to the secret sharing problem has grown into an active area of research. We will cover some of these developments in this survey.

Simmons' Scheme

Simmons formulates secret sharing schemes in terms of *affine spaces* instead of projective spaces. The reasons for using affine spaces instead of projective spaces are explained by Simmons [79]. (There is a correspondence between projective spaces and affine spaces, cf. Beth, Jungnickel and Lenz [9].) Briefly, an affine space $AG(n, q)$ consist of points—the vectors of $V(n, q)$, and a hierarchy of l-dimensional subspaces for $l \leq n$ and their *cosets*. These correspond to the equivalence classes in projective geometry mentioned above, and are called the *flats* of $AG(n, q)$. The equivalence classes of lines, planes, etc., of $AG(t, q)$ are the 1-dimensional, 2-dimensional, etc., flats. A *hyperplane* is a flat of co-dimension one. To realize a (t, n)-threshold scheme in $AG(t, q)$, the secret is represented by a point p chosen randomly from $AG(t, q)$, which lies on a publicly known line V_d (lines have q points). A hyperplane V_i of the *indicator variety* is selected so that V_i intersects V_d in a single point p. The shares of the secret are the subsets of points of V_i. An authorized subset of participants, which spans V_i, enables reconstruction of the secret. If an unauthorized subset of participants attempts to reconstruct the secret, their shares will only span a flat which intersects V_d in the empty set. Thus they gain no information about the secret. The precise amount of information gained by the unauthorized participants about the secret can be expressed in terms of the defining parameters of $AG(n, q)$. These schemes are perfect. Simmons [79] gives a detailed explanation of the implementation of secret sharing schemes using projective and affine spaces.

Shamir's Scheme

Shamir's [75] scheme realizes (t, n)-access structures using on polynomial interpolation over finite fields. In his scheme the secrets $\mathcal{S}$ belong to a prime power finite field $GF(q)$, which satisfies $q \geq n + 1$. In the initialization phase, the dealer $\mathcal{D}$ chooses n distinct nonzero elements $\{x_1, \ldots, x_n\}$ from $GF(q)$ and allocates these to participants $\{P_1, \ldots, P_n\}$. This correspondence is publicly known, and creates undesirable side effects if any of the participants are dishonest; see Section "The Problem of Cheaters." However for now, we will assume that all the participants obey faithfully the protocol for reconstructing the secret.

Fix a random element of $GF(q)$ as the secret K. The shares of K are created using the following protocol.

(a) $\mathcal{D}$ chooses $a_1, a_2, \ldots, a_{t-1}$ from $GF(q)$ randomly, uniformly, and independently.

(b) Let $a(x)$ be a polynomial of degree at most $t-1$, defined as $a(x) = K + a_1 x + a_2 x^2 + \cdots + a_{k-1} x^{k-1}$.

(c) The shares of the secret key are $y_i = a(x_i)$, for $1 \leq i \leq n$.

With the above data, if any t out of the n participants $\{x_{i_1}, \ldots, x_{i_t}\}$ combine their shares $\{y_{i_1}, \ldots, y_{i_t}\}$, then using Lagrangian interpolation, there is a unique polynomial of degree at most $t-1$ passing through the points: $\{(x_{i_1}, y_{i_1}), \ldots, (x_{i_t}, y_{i_t})\}$. So the combined shares of the t participants can be used to recreate the polynomial $a(x)$, and hence the secret, which is $K = a(0)$.

The relation between the secret and the shares is obtained from Lagrange's interpolation formula as

$$K = \sum_{j=1}^{t} y_{i_j} b_j \,, \tag{41.3}$$

where the b_j are defined as

$$b_j = \prod_{\substack{1 \leq k \leq t, \\ k \neq j}} \frac{x_{i_k}}{x_{i_k} - x_{i_j}} \,.$$

Shamir's scheme is computationally efficient in terms of the computational effort required to create the shares and to recover the secret. Also the share size is optimal in an information theoretic sense, cf. Definition 41.2.

The reconstruction phase in Shamir's scheme can also be considered as a system of linear equations, which are defined by the shares K_i. If t shares are submitted to the combiner, the system of linear equations

$$y_{i_j} = K + a_1 x_{i_j} + a_2 x_{i_j}^2 + \cdots + a_{k-1} x_{i_j}^{k-1}, \;\; j = 1, \ldots t$$

can be solved for the unknowns $K, a_1, a_2, \ldots, a_{t-1}$, because the determinant of this system of equations is a nonsingular Vandermonde determinant. (The $\{x_1, \ldots, x_n\}$ are pair-wise distinct.) However, if $t-1$ participants try to reconstruct the secret, they face the problem of solving $t-1$ linear equations in t unknowns. This system of equations has one degree of freedom. Consequently, $t-1$ participants do not obtain any information about the secret, as K was selected uniformly and randomly from $GF(q)$. Shamir's system is perfect.

A (t, t) Threshold Scheme

Karnin et al. [54] describe a secret sharing scheme which realizes (t, t)-access structures. The interest in such schemes is that they can be used as the basis for other cryptographic constructions.

In their scheme, the set of secrets $\mathcal{S}$ is the ring of residue classes Z_m, where m is any integer. (In applications m is large.) The secret K is shared using the following algorithm.

(a) $\mathcal{D}$ secretly chooses randomly, uniformly, and independently $t-1$ elements $y_1, y_2, \ldots, y_{t-1}$ from Z_m; y_t is defined as

$$y_t = K - \sum_{i=1}^{t-1} y_i \bmod m \,.$$

(b) Participant P_i for $1 \leq i \leq t$ receives the share y_i from $\mathcal{D}$.

The above system is perfect, as the following argument shows. The set of shares of $l < t$ participants attempting to reconstruct the secret either contains the share $y_t = K - \sum_{i=1}^{t-1} y_i \bmod m$, or not. In both cases the (unauthorized) participants lack the necessary information to determine K. Shamir's scheme with $t = n$ provides an alternative construction of (t, t)-threshold schemes, using the fields $GF(q)$ instead of Z_m.

Threshold Schemes and Discrete Logarithms

The *discrete logarithm* has been widely employed in the literature to transform threshold schemes into conditionally secure schemes with extra properties. This idea is exploited in the papers by Benaloh [1], Beth [10], Charnes, Pieprzyk and Safavi-Naini [27], Charnes and Pieprzyk [28], Lin and Harn [58], Langford [56], and Hwang and Chang [50].

It is a consequence of the linearity of Eq. (41.3) that Shamir's scheme can be modified to obtain schemes having enhanced properties such as *disenrollment* capability, in which shares from one or more participants can be made incapable of forming an updated secret. (The formal analysis of schemes with this property was given by Blakley et al. [13].) Let $a(x)$ be a polynomial and let $a(i)$ be the shares as in Shamir's scheme. In the modified threshold scheme proposed by Charnes, Pieprzyk and Safavi-Naini [27], $g^{a(0)}$ is the secret and the shares are $s_i = g^{c_i}, c_i = a(i)$. A generator g of the cyclic group of the field $GF(2^n)$ is chosen so that $2^n - 1$ is a Mersenne prime.

The modified (t, n)-threshold schemes are capable of disenrolling participants whose shares have been compromised either through loss or theft, and still maintain the original threshold level. In the event that some of the original shares are compromised, the KDC can issue using a public authenticated channel a new generator g' of the cyclic group of $GF(2^n)$. The shareholders can calculate their new shares $s_i{}'$ from the initial secret data according to

$$s_i{}' = g'^{c_i} .$$

Hwang and Chang [50] used a similar setting to obtain *dynamic* threshold schemes.

Threshold schemes with disenrollment capability, without the assumption of the intractability of the discrete logarithm problem, can be based on *families* of threshold schemes. The properties of these schemes are studied in a paper by Charnes, Pieprzyk and Safavi-Naini [26]; here we provide the basic definition.

DEFINITION 41.4 A threshold scheme family (TSF) is defined by an $(m \times n)$ matrix of shares $[s_{i,j}]$ such that

1. Any row $(s_{i,1}, s_{i,2}, \ldots, s_{i,n})$ represents an instance of $TS_{r_i}(t_i, n)$ where $i = 1, \ldots, m$.
2. Any column $(s_{1,j}, s_{2,j}, \ldots, s_{m,j})$ represents an instance of $TS_{c_j}(t_j, m)$ where $j = 1, \ldots, n$.

A family of threshold schemes in which all rows and all columns are ideal schemes is called an *ideal threshold scheme family*, or ITS family for short. In these schemes it is possible to alter dynamically the threshold values by moving from one level of the matrix to another.

Lin and Harn [58] and Langford [56] use the discrete logarithm to transform Shamir's scheme into a conditionally secure scheme which does not require a trusted KDC. A similar approach is used by Langford [56] to obtain a *threshold signature* scheme. Beth [10] describes a protocol for *verifiable secret sharing* for general access structures based on geometric schemes. The discrete logarithm problem is used to encode the secret and the shares so that they can be publicly announced for verification purposes.

It should be noted that the definition of disenrollment given by Charnes et al. [27] is not the same as that of Blakley et al. [13]. Blakley et al. establish a lower bound on the number of bits required to encode the shares in schemes with disenrollment. Their bound shows that this number grows linearly with the number of disenrollments. They also present two geometric (t, n)-threshold schemes which meet this bound.

It is interesting to note that Benaloh [1] uses the discrete logarithm to transform Shamir's scheme, but for a very different purpose. One of the properties of the discrete logarithm is that the sum of the discrete logarithms of the shares of a secret is equal to the discrete logarithm of the product of the shares of the secret. This property has an application in secret-ballot elections (cf. Benaloh [1]) where, in contrast with schemes mentioned above, the discrete logarithm problem is required to be tractable.

The *homomorphic* property introduced by Benaloh [1] has prompted the question whether similar schemes can be set up in noncommutative groups—other than the additive and multiplicative groups of

finite fields. Frankel and Desmedt [44] prove that perfect homomorphic threshold schemes cannot be set up in non-commutative groups. It is an open problem to find useful applications of homomorphic schemes in abelian groups.

Error Correcting Codes and Secret Sharing

McEliece and Sarwate [61] observe that Shamir's scheme is closely related to Reed–Solomon codes [62]. The advantage of this formulation is that the error correcting capabilities of the Reed–Solomon codes can be translated into desirable secret sharing properties.

Let $(\alpha_0, \alpha_1, \ldots, \alpha_{q-1})$ be a fixed list of the nonzero elements of a finite field $GF(q)$ containing q elements. In a Reed–Solomon code, an information word $\mathbf{a} = (a_0, a_1, \ldots, a_{k-1})$, $a_i \in GF(q)$, is encoded into the codeword $\mathbf{D} = (D_1, D_2, \ldots, D_{r-1})$, where $D_i = \sum_{j=0}^{k-1} a_j \alpha_i{}^j$. In this formulation the secret is $a_0 = -\sum_{i=1}^{r-1} D_i$ and the shares distributed to the participants are the D_i.

In the above formulation of threshold schemes, algorithms such as the errors-and-erasures decoding algorithm can be used to correct t out of s shares where $s - 2t \geq k$ in a (k, n)-threshold scheme, if for some reason these shares were corrupted. The algorithm will also locate which invalid shares D_i were submitted, either as a result of deliberate tampering or as a result of storage degradation.

Karnin et al. [54] realize threshold schemes using linear codes. Massey [59] introduced the concept of *minimal codewords*, and proved that the access structure of a secret sharing scheme based on a $[n, k]$ linear code is determined by the minimal codewords of the *dual* code. To realize a (t, n)-threshold scheme, a linear $[n+1, t; q]$ code $\mathcal{C}$ over $GF(q)$ is selected. If G is the generator matrix of $\mathcal{C}$ and $s \in GF(q)$ is the secret, then the *information vector* $\mathbf{s} = (s_0, s_1, \ldots, s_{t-1})$ is any vector satisfying $s = \mathbf{s} \cdot \mathbf{g}^T$, where $\mathbf{g}^T$ is the first column vector of G. The codeword corresponding to $\mathbf{s}$ is $\mathbf{s}G = (t_0, t_1, \ldots, t_n)$. Each participant in the scheme receives t_i as its share and t_0 is the secret. To recover the secret, first the linear dependency between $\mathbf{g}$ and the other column vectors in the (public) generator matrix G is determined. If $\mathbf{g} = \sum x_j \mathbf{g}_j$ is the linear relation, the secret is given by $\sum x_j t_{i_j}$, where $\{t_{i_1}, t_{i_2}, \ldots, t_{i_t}\}$ is a set of t shares.

Renvall and Ding [69] consider the access structures of secret sharing schemes based on linear codes as used by McEliece and Sarwate and Karnin et al. They determine the access structures that arise from $[n+1, k, n-k+2]$ MDS codes—codes which achieve the *singleton bound* [62]. Bertilsson and Ingemarsson [8] use linear block codes to realize secret sharing schemes for general access structures. Their algorithm takes a description of an access structure by a monotone Boolean formula Γ, and outputs the generator matrix of a linear code which realizes Γ.

Combinatorial Structures and Secret Sharing

There are various connections between combinatorial structures and secret sharing, cf. [9]. Stinson and Vanstone [85], and Schellenberg and Stinson [72] study threshold schemes based on *combinatorial designs*. Stinson [84] uses *balanced incomplete blocks designs* to obtain general bounds on the information rate ρ^* of schemes with access structure based on graphs (cf. "Ideal and Other Schems").

Street [83] surveys *defining* sets for t-designs and *critical sets* for Latin squares, with the view of applying these concepts to multilevel secret sharing schemes, in which a hierarchical structure can be imposed on the shares. To illustrate these methods, we give an example of a (2, 3)-threshold scheme based on a small Latin square, cf. Chaudhry and Seberry [31]. For an example of a scheme with a hierarchical share structure, cf. Street [83].

Let $(i, j; k)$ denote that the value k is in position (i, j) of the Latin square

$$L = \begin{pmatrix} 1 & 2 & 3 \\ 2 & 3 & 1 \\ 3 & 1 & 2 \end{pmatrix}.$$

The shares of the secret, which is L, are: $S = \{(2, 1; 1), (3, 2; 1), (1, 3; 3)\}$.

More recently, critical sets in *Room squares* have been used to realize multilevel secret sharing schemes, cf. Chaudhry and Seberry [31]. Some other approaches to multilevel schemes are considered in the papers by Beutelspacher [11] and Cooper et al. [33]. The schemes based on Latin and Room squares are examples of nonperfect schemes which will be discussed in the section on "Nonperfect Schemes."

The Problem of Cheaters

So far we have assumed that the participants in a secret sharing scheme are honest and obey the reconstruction protocol. However, there are conceivable situations where a dishonest clique of participants (assuming an honest KDC) may attempt to defraud the honest participants by altering the shares they were issued.

In the McEliece and Sarwate formulation of Shamir's scheme, invalid shares can be identified. Schemes with this capability are said to have the *cheater identification* property. A weaker capability ascertains that invalid shares were submitted in the reconstruction phase without necessarily locating the source of these shares; this is called *cheater detection*.

Tompa and Woll [87] show that public knowledge of the ordinates in Shamir's scheme allows a clique of dishonest participants to modify their shares resulting in an invalid secret K' being recreated. Suppose that participants $i_1, i_2, \ldots, i_t$ agree to pool their shares in order to recreate the secret. A dishonest participant, say i_1, can determine a polynomial $\Delta(x)$ of degree at most $t-1$ from $\Delta(0) = -1$ and $\Delta(i_2) = \Delta(i_3) = \ldots = \Delta(i_t) = 0$ using Lagrangian interpolation. Instead of the share originally issued by the dealer, the cheater submits the modified share $a(i_1) + \Delta(i_1)$. Lagrangian interpolation of points using the modified share will result in the polynomial $a(x) + \Delta(x)$ being recreated, instead of the intended polynomial $a(x)$. Now the constant term is $a(0) + \Delta(0) = K - 1$, a legal but incorrect secret. The honest participants believe that the secret is $K - 1$, but the cheater privately recovers the correct secret as $K = (K-1) + 1$.

To prevent this type of cheating, Tompa and Woll define the shares in their scheme as (x_1, d_1), $(x_2, d_2), \ldots$ (x_n, d_n). The dealer chooses randomly and uniformly a permutation $(x_1, x_2, \ldots, x_n)$ of n distinct elements from $\{1, 2, \ldots, q-1\}$, and $d_i = a(x_i)$. The modified scheme resists the above attack for up to $t-1$ cheaters. The expected running time of the scheme is polynomial in $k, n, \log s$ and $\log(1/\epsilon)$, where ϵ is a designated security parameter of the scheme and the secret k is chosen from $\{0, 1, \ldots, s-1\}$. But the participants need to keep secure two shares instead of the usual single share.

Brickell and Stinson [20] modified Blakley's geometric (t, n)-scheme and obtained a scheme in which cheaters can be detected and identified. Blakley et al. [13] proved that this scheme is capable of disenrolling participants; cf. "Threshold Schemes and Discrete Logarithms." To set up the scheme, the dealer performs certain computations such as checking that the shadows are in *general position*. It is an open problem whether these computations can be done efficiently as the numbers of participants increase.

The problem of secret sharing without the usual assumptions about the honesty of the participants, or even the KDC has been considered in the literature. For example, in verifiable secret sharing it is not assumed that the dealer is honest. This problem is studied by Chor et al. [32]. The problem is how to convince the participants in a (t, n)-threshold scheme, that every subset of t shares of a share set $\{s_1, s_2, \ldots, s_n\}$ defines the same secret. This is called t-consistency. In Shamir's scheme, t-consistency is equivalent to the condition that interpolation on the points $(1, s_1)$, $(2, s_2), \ldots, (n, s_n)$ yields a polynomial of degree at most $t-1$. As application of homomorphic schemes, Benaloh [1] gives an *interactive proof* that Shamir's scheme is t-consistent.

General Access Structures

A complete discussion of secret sharing requires the notion of a general *access structure*.

Ito, Saito and Nishizeki [52] describe a method to realize secret sharing schemes for general access structures. They observe that for most applications of secret sharing it suffices to consider *monotone* access structures (MAS), defined as follows.

DEFINITION 41.5 Given a set $\mathcal{P}$ of n participants ($|\mathcal{P}| = n$), a monotone access structure on $\mathcal{P}$ is a family of subsets $\mathcal{A} \subseteq 2^{\mathcal{P}}$ such that

$$\mathcal{A} \subseteq \mathcal{A}' \subseteq \mathcal{P} \Rightarrow \mathcal{A}' \in \mathcal{A} \tag{41.4}$$

The intersection $\mathcal{A}_1 \cap \mathcal{A}_2$, and the union $\mathcal{A}_1 \cup \mathcal{A}_2$ of two monotone access structures is a monotone access structure. If $\mathcal{A}$ is a monotone access structure, then $2^{\mathcal{P}} \setminus \mathcal{A} = \bar{\mathcal{A}}$ is not a MAS. Any MAS can be expressed equivalently by a monotone Boolean function. Conversely, any Boolean expression without negations represents a MAS.

In view of the above observations, we consider the *minimal* authorized subsets of an access structure $\mathcal{A}$ on $\mathcal{P}$. A set $B \in \mathcal{A}$ is minimal authorized, if for each proper subset A of B, it is the case that $A \notin \mathcal{A}$. The set of minimal authorized subsets of $\mathcal{A}$ is called the *basis*. An access structure $\mathcal{A}$ is the unique *closure* of the *basis*, i.e., all subsets of $\mathcal{P}$ that are supersets of the basis elements.

Some examples of inequivalent access structures on four participants are given by the following monotone formulae: $\Gamma_1 = P_1P_2P_3 + P_1P_2P_4 + P_1P_3P_4 + P_2P_3P_4$—a $(3, 4)$-threshold scheme; $\Gamma_2 = P_1P_2 + P_3P_4$; $\Gamma_4 = P_1P_2 + P_2P_3 + P_3P_4$. In these formulae, the P_i's represent the participants in the scheme (sometimes the literals A, B, etc., are used). The authorized subsets in the access structure are specified precisely by these formulae. For example, Γ_2 stipulates that either P_1 AND P_2 OR P_3 AND P_4 are the authorized subsets. (It is known that no threshold scheme can realize the access structure defined by Γ_2. For a proof, cf. Benaloh and Leichter [2].)

The inequivalent access structures on three and four participants, and the information rates of secret sharing schemes realizing these structures are given by Simmons, Jackson and Martin [80] and by Stinson [84]. The information rates of all inequivalent access structures on five participants are discussed by Martin and Jackson [60]. It should be remarked that a practical examination of access structures is probably limited to five participants. For more than five participants, the number of equivalence classes of monotone Boolean formulae becomes too great to consider. However, Martin and Jackson [60] provide inductive methods using which the information rates of an access structure Γ is related to the information rates of smaller access structures that are "embedded" in Γ.

Secret sharing schemes for nonmonotone access structures have also been investigated, cf. Simmons [79].

Realizing General Access Structures

Ito et al. [52] were the first to show how to realize secret sharing schemes for general access structures. Benaloh and Leichter [2] simplified the method of Ito et al.

They show that any monotone access structure can be recognized by a *monotone* Boolean circuit. In a monotone circuit, each variable corresponds to an element of $\mathcal{P}$. The circuit outputs a true value only when the set of variables which take a true value corresponds to an authorized subset of $\mathcal{P}$, i.e., belongs to the access structure. Monotone circuits are described by Boolean formulae which involve only AND and OR operators. Using Benaloh and Leichter's method, one can realise any access structure as a composite of subsecrets. The subsecrets are shared across AND gates by (t, t)-threshold schemes for appropriate t, and all the inputs to the OR gates have the same value.

Simmons, Jackson and Martin [80] show how *cumulative arrays*, first studied by Ito et al. [52], can be used to realize geometric secret sharing schemes for general access structures.

DEFINITION 41.6 A cumulative array $C_{\mathcal{A}} = (\mathcal{S}, f)_{\mathcal{A}}$ for the access structure $\mathcal{A}$ is a pair comprising of the share set $\mathcal{S} = \{s_1, s_2, \ldots\}$, and the dealer function $f : \mathcal{P} \to 2^{\mathcal{S}}$ which assigns subset of shares to each participant.

As an example, consider the following access structure:

$$\mathcal{A} = \text{ closure } \{\{P_1, P_2\}, \{P_2, P_3\}, \{P_3, P_4\}, \{P_1, P_4\}\} ,$$

where $\mathcal{P} = \{P_1, P_2, P_3, P_4\}$. Let $\mathcal{S} = \{s_1, s_2\}$. A cumulative array for this access structure is $f(P_1) = s_1$, $f(P_2) = s_2$, $f(P_3) = s_1$, and $f(P_4) = s_2$.

Perfect geometric secret sharing schemes are obtained from cumulative arrays as follows. Choose a projective space $V_i = PG(m-1, q)$, where m is the number of columns in the cumulative array. In V_i, let $\{s_i, \ldots, s_m, K\}$ be $m+1$ points such that no m points lie on a hyperplane of V_i – the points are in general position. A domain variety V_d is chosen so that $V_i \bigcap V_d = \{K\}$. The set of shares in the geometric scheme is $\{s_i, \ldots, s_m\}$ and K is the secret. The shares are distributed using the cumulative array: participant P_i receives share s_j if and only if the (i, j) entry of the array is one. Note that it could be difficult to verify the general position hypothesis for large m, cf. Brickell and Stinson [20]. Jackson and Martin [53] show that any geometric secret sharing scheme realizing an access structure is "contained" in the cumulative array, which realizes the access structure.

For any access structure $\mathcal{A}$ on the set $\mathcal{P}$, there is a unique minimal cumulative array. Thus to implement geometric secret sharing schemes with the minimal number of shares, we need only consider minimal cumulative arrays. It remains only to have a means by which the minimal cumulative array can be calculated given an arbitrary monotone Boolean function Γ. Such a method was first given by Simmons, Jackson and Martin [80]. It relies on minimizing the Boolean expression which results when the AND and OR operators in Γ are exchanged.

An alternative method for calculating minimal cumulative arrays is described by Charnes and Pieprzyk [29]. Their method has the advantage that the complete truth table of Γ is not required for some Γ, thereby avoiding an exponential time computation. For general Boolean expressions, as the number of variables increases the time complexity of the above method and the method given by [80] is the same.

To describe the method of Charnes and Pieprzyk [29], we require the following.

DEFINITION 41.7 [29] The *representative matrix* M_Γ of a monotone Boolean function $\Gamma(P_1, P_2, \ldots, P_n)$, expressed as a disjunctive sum of r products of n variables, is an $n \times r$ matrix with rows indexed by the P_i and columns by the product terms of the P_i. The (i, j)-entry is one if P_i occurs in the jth product, and is zero otherwise.

For example, if $\Gamma = P_1P_2 + P_2P_3 + P_3P_4$, then M_Γ is the following matrix:

	P_1P_2	P_2P_3	P_3P_4
P_1	1	0	0
P_2	1	1	0
P_3	0	1	1
P_4	0	0	1

Suppose that $\Gamma(P_1, P_2, \ldots, P_n)$ is a monotone formula expressed in *minimal disjunctive form*, i.e., a disjunctive sum of products of the P_i and no product term is contained in any other. Let M_Γ be its representative matrix.

DEFINITION 41.8 [29] A subset $\{P_l, P_m, \ldots\}$ of the variables of $\Gamma(P_1, P_2, \ldots, P_n)$ is a *relation set* if $P_lP_m \ldots$ is represented in M_Γ by the all ones vector.

In the representative matrix above, $\{P_1, P_3\}$, $\{P_2, P_3\}$, and $\{P_2, P_4\}$ are the *minimal* representative

sets, i.e., not contained in any other representative set. The Boolean formula derived from these sets is $P_1P_3 + P_2P_3 + P_2P_4$.

THEOREM 41.2 *[29] Let $\Gamma(P_1, P_2, \ldots, P_n)$ be a monotone formula and M_Γ its representative matrix. Let $\mathcal{R}$ be the collection of minimal relation sets of M_Γ. Then the representative matrix whose rows are indexed by the variables P_i and columns by product terms derived from $\mathcal{R}$ is the minimal cumulative array for $\mathcal{A}$.*

Thus, using the above theorem the matrix

	P_1P_3	P_2P_3	P_2P_4
P_1	1	0	0
P_2	0	1	1
P_3	1	1	0
P_4	0	0	1

is the minimal cumulative array for $\Gamma = P_1P_2 + P_2P_3 + P_3P_4$. To realize Γ as a geometric scheme, we require a projective space $V_i = PG(2, q)$. The secret $K \in V_d$, and the shares $\{s_1, s_2, s_3\}$ are points chosen in general position in V_i. The cumulative array specifies the distribution of the shares: P_1 receives share $\{s_1\}$; P_2 receives shares $\{s_2, s_3\}$; P_3 receives shares $\{s_1, s_2\}$; P_4 receives share $\{s_3\}$. It can be easily verified that only the authorized subsets of participants can recreate the secret, e.g., the combined shares of P_1 and P_2 span V_i, hence these participants can recover the secret as $V_i \cap V_d = \{K\}$. But unauthorized participants, e.g., P_1 and P_3, cannot recover the secret.

An algorithm for calculating cumulative arrays, based on Theorem 41.2, is described by Charnes and Pieprzyk [29]. This algorithm is efficient for those Γ which have columns containing many zeros in M_Γ. Thus in the previous example, the combinations $\{P_1, P_2\}$, $\{P_1, P_4\}$ and $\{P_3, P_4\}$ cannot produce relation sets and can be ignored. Further computational economy is obtained if the Boolean formula has a large degree of symmetry.

Ideal and Other Schemes

Brickell [18] gives a vector space construction for realizing ideal secret sharing schemes for certain types of access structures, Γ. Let ϕ be a function

$$\phi : \mathcal{P} \cup \{\mathcal{D}\} \rightarrow GF(q)^d$$

with the property that $\phi(D)$ can be expressed as a linear combination of the vectors in $< \phi(P_i) : P_i \in B >$ if and only if B is an authorized subset, i.e. $B \in \Gamma$. Then, for any such ϕ, the distribution rules (cf. "The Matrix Model") are for any vector $\mathbf{a} = (a_1, \ldots, s_d)$ in $GF(q)^d$, a distribution rule is given by the inner product of $\mathbf{a}$ and $\phi(x)$ for every $x \in \mathcal{P} \cup \{\mathcal{D}\}$. Under the above conditions, the collection of distribution rules is an ideal secret sharing scheme for Γ. A proof of this result can be found in a paper by Stinson [84].

Shamir's (t, n)-threshold scheme is an instance of the vector space construction. Access structures $\Gamma(G)$, whose basis is the edge set of certain undirected graphs, can also be realized as ideal schemes by this construction. In particular the access structure $\Gamma(G)$, where $G = (V, E)$ is a *complete multigraph* can be realized as an ideal scheme. A proof of this is given by Stinson [84].

A relation between ideal secret sharing schemes and *matroids* was established by Brickell and Davenport [19]. The matroid theory counterpart of a minimal linearly dependent set of vectors in a vector space is called a *circuit*. A *coordinatizable* matroid is one that can be mapped into a vector space over a field in a way that preserves linear independence. Brickell and Davenport [19] prove the following theorem about coordinatizable matroids.

THEOREM 41.3 *[19] Suppose the connected matroid $\mathcal{M} = (X, \mathcal{I})$ is coordinatizable over a finite field. Let $x \in X$ and let $\mathcal{P} = X \setminus \{x\}$. Then there exists an ideal scheme for the connected access structure having basis $\Gamma_0 = \{C \setminus \{x\} : x \in C \in \mathcal{C}\}$, where $\mathcal{C}$ denotes the set of circuits of $\mathcal{M}$.*

There are limits to the access structures that can be realized as ideal secret sharing schemes. This was first established by Benaloh and Leichter [2]. They proved that the access structure on four participants specified by the monotone formula $\Gamma = P_1P_2 + P_2P_3 + P_3P_4$ cannot be realized by an ideal scheme. The relation between the size of the shares and the secret for Γ was made precise by Capocelli et al. [23]. They proved the following information theoretic bound.

THEOREM 41.4 *For the access structure $\Gamma = closure\{\{P_1, P_2\}, \{P_2, P_3\}, \{P_3, P_4\}\}$ on four participants $\{P_1, P_2, P_3, P_4\}$, the inequality $H(P_2) + H(P_3) \geq 3H(K)$ holds for any secret sharing scheme realizing Γ.*

From this theorem it follows that the information rate ρ of any secret sharing scheme realizing Γ satisfies the bound $\rho \leq \frac{2}{3}$. Bounds are also derived by Capocelli et al. [23] for the maximum information rate ρ^* of access structures $\Gamma(G)$, where the graph G is a *path* P_n ($n \geq 3$); a *cycle* C_n, $n \geq 6$, for n even and $n \geq 5$, for n odd; or any *tree* T_n.

Realizing Schemes Efficiently

In view of bounds on the information rates of secret sharing schemes, it is natural to ask whether there exist schemes whose information rates equal the known bounds. For example, for $\Gamma = P_1P_2 + P_2P_3 + P_3P_4$ one is interested in realizations of Γ with $\rho = \frac{2}{3}$.

Stinson [84] used a general method, called *decomposition construction*, to build larger schemes starting from smaller ideal schemes. In this method, the basis Γ_0 of an access structure is decomposed into smaller access structures, as $\Gamma_0 = \cup\Gamma_k$, where the Γ_k are the basis of the constituent access structures which can be realized as ideal schemes. From such decompositions of access structures, Stinson [84] derives a lower bound: $\rho^*(\Gamma) \geq \ell/R$, where ℓ and R are two quantities defined in terms of the ideal decomposition of Γ_0. The decomposition construction and its precursor, the graph decomposition construction (cf. Blundo et al. [15]), can be formulated as linear programming problems in order to derive the best possible information rates that are obtainable using these constructions.

Other ways of realizing schemes with optimal or close to optimal information rates are considered by Charnes and Pieprzyk [30]. Their method combines multiple copies of cumulative arrays using the notion of *composite shares*—combinations of the ordinary shares in cumulative arrays. This procedure is stated as an algorithm that outputs a cumulative array with the best information rate. It is not clear how efficient this algorithm is as the numbers of participants increases. However, the optimal information rates for access structures on four participants given by Stinson [84] can be attained by combining cumulative arrays.

Nonperfect Schemes

It is known that in nonperfect schemes the size of the shares is less that the size of the secret, i.e., $H(s_i) < H(K)$. Because of this inequality, a nonperfect scheme can be used to disperse a computer file to n sites, in such a way that the file can be recovered from its images that are held at any t of the sites ($t \leq n$). Moreover, this can be done so that the size of the images is less than the size of the original file, resulting in an obvious saving of disk space. Making backups of computer files using this method provides insurance against the loss or destruction of valuable data. For details, cf. Karnin et al. [54].

A formal analysis of nonperfect secret sharing schemes is given by Ogata, Kurosawa and Tsujii [66]. Their analysis characterizes, using information theory, secret sharing schemes in which the participants *not*

belonging to an access structure do gain some information about the secret. This condition is precluded in perfect secret sharing schemes.

Ogata et al. [66] define a nonperfect scheme in terms of a triple of access sets $(\Gamma_1, \Gamma_2, \Gamma_3)$, which partition the set of all subsets of the participants $\mathcal{P}$. Γ_1 is the family of access subsets, Γ_2 is the family of semi-access subsets and Γ_3 is the family of non-access subsets. The participants belonging to the semi-access subsets are able to obtain some, but not complete information about the secret. The participants which belong to the non-access subsets gain no information about the secret.

The *ramp schemes* of Blakley and Meadows [14] are examples of nonperfect schemes where the access structure consists of semi-access subsets. Another way of viewing ramp schemes is that the collective uncertainty about a secret gradually decreases as more participants join the collective.

Ogata et al. [66] prove a lower bound on the size of the shares in nonperfect schemes. They also characterize nonperfect schemes for which the size of the shares is $|K|/2$.

Ogata and Kurosawa [65] establish a general lower bound for the sizes of shares in nonperfect schemes. They show that there is an access hierarchy for which the size of the shares is strictly larger than this bound. It is in general a difficult problem to realize nonperfect secret sharing schemes with the optimum share size, as in the case of perfect schemes.

41.3 Threshold Cryptography

There are circumstances where an action requires to be executed by a group of people. For example, to transfer money from a bank a manager and clerk need to concur. A bank vault can be opened only if three high ranking bank employees cooperate. A ballistic missile can be launched if two officers authorize the action.

Democratic groups usually exhibit a flat relational structure where every member has equal rights. On the other hand, in hierarchical groups, the privileges of group members depend on their position in the hierarchy. A member on the level $i - 1$ inherits all the privileges from the level i, as well as additional privileges specific to its position.

Unlike single-user cryptography, threshold or society-oriented cryptography allows groups to perform cryptographic operations such as encryption, decryption, signature, etc. A trivial implementation of group-oriented cryptography can be achieved by concatenating secret sharing schemes and a single user cryptosystem. This arrangement is usually unacceptable as the cooperating subgroup must first recover the cryptographic key. Having access to the key can compromise the system, as its use is not confined to the requested operation. Ideally, the cooperating participants should perform their private computations in one go. Their partial results are then sent to a so-called *combiner* who calculates the final result. Note that at no point is the secret key exposed.

A group-oriented cryptosystem is usually set up by a *dealer* who is a trusted authority. The dealer generates all the parameters, distributes elements via secure channels if the elements are secret, or broadcasts the parameters if they need not be protected. After setting up a group cryptosystem, the dealer is no longer required, as all the necessary information has been deposited with the participants of the group cryptosystem.

If some participants want to cooperate to perform a cryptographic operation, they use a combiner to perform the final computations on behalf of the group. The final result is always correct if the participants belong to the access structure and follow the steps of the algorithm. The combiner fails if the participants do not belong to the access structure, or if the participants do not follow the algorithm (that is, they cheat). The combiner need not be trusted; it suffices to assume that it will perform some computations reliably but not necessarily all.

The *access structure* is the collection of all subsets of participants authorized to perform an action. An example is a (t, n)-threshold scheme, where any t out of n participants are authorized subsets ($t \leq n$).

Threshold cryptography provides tools for groups to perform the following tasks:

- Threshold encryption—a group generates a valid cryptogram which can later be decrypted by a single receiver;
- Threshold decryption—a single sender generates a valid cryptogram which can be decrypted by a group;
- Threshold authentication—a group of senders agrees to co-authenticate the message so the receiver can decide whether the message is authentic or not;
- Threshold signature (multisignature)—a group signs a message which is later validated by a single verifier;
- Threshold pseudorandom generation.

Threshold Encryption

Public-key cryptography can be used as a basis for simple group encryption. Assume that a receiver wants to have a communication channel from a group of n participants $\mathcal{P} = \{P_1, \ldots, P_n\}$. Further suppose that the receiver can decrypt a cryptogram only if all participants cooperate, i.e., a (n, n)-threshold encryption system. Group encryption works as follows.

Assume that the group and the receiver agree to use the RSA cryptosystem with the modulus $N = pq$. The receiver first computes a pair of keys: one for encryption e and the other for decryption d, where $e \times d \equiv 1 \bmod (p-1)(q-1)$. Both keys are secret. The factors p and q are known by the receiver only. The encryption key is communicated to the dealer (via a secure channel). The dealer selects $n-1$ shares e_i of the encryption key at random from the interval $[0, e/n]$. The last share is

$$e_n = e - \sum_{i=1}^{n-1} e_i \ .$$

Each share e_i is communicated to participant P_i via a secure channel $(i = 1, \ldots, n)$.

Now if the group wants to send a message m to the receiver, each participant P_i prepares its partial cryptogram $c_i \equiv m^{e_i} \bmod N$ $(i = 1, \ldots, n)$. After collecting n partial cryptograms, the receiver can recover the message $m \equiv (\prod_{i=1}^{n} c_i)^d \bmod N$. Note that the receiver also plays the role of a combiner. Moreover, the participants need not reconstruct the secret encryption key e and at no stage of decryption is the encryption key revealed—this is a characteristic feature of threshold cryptography.

Many existing secret-key algorithms such as the DES [64], LOKI [22], FEAL [76], or the Russian GOST [82], are not homomorphic. These algorithms cannot be used for threshold encryption. The homomorphic property is necessary in order to generate shares of the key so that partial cryptograms can be combined into a cryptogram for the correct message, cf. [1].

Threshold encryption has not received a great deal of attention, perhaps because of its limited practical significance.

Threshold Decryption

Hwang [49] proposes a cryptosystem for group decryption based on the discrete logarithm problem. In his system it is assumed that the sender knows the participants of the group. The sender encrypts the message using a predetermined (either private or public key) cryptosystem with a secret key known to the sender only. The sender then distributes the secret key among the group of intended receivers using Shamir's (t, n)-threshold scheme. Any t cooperating participants can recover the decryption key and decrypt the cryptogram. In Hwang's scheme, key distribution is based on the Diffie–Hellman [39] protocol. Thus the security of his scheme is equivalent to the security of the discrete logarithm problem. However, the main problem with the above solution is that the key can be recovered by a straightforward application of secret sharing. This violates the fundamental requirement that the decryption key must never be revealed to the group (or combiner).

We consider now an implementation of a scheme for (t, n)-threshold decryption. The group decryption used here is based on the ElGamal public-key cryptosystem [41] and is described by Desmedt and Frankel [35].

The system is set up by the dealer $\mathcal{D}$ who first chooses a Galois field $GF(q)$ such that $q-1$ is a Mersenne prime and $q = 2^{\ell}$. Further $\mathcal{D}$ selects a primitive element $g \in GF(q)$ and a nonzero random integer $s \in GF(q)$. The dealer computes $y = g^s \bmod q$ and publishes the triple (g, q, y) as the public parameters of the system. The dealer then uses Shamir's (t, n)-threshold scheme to distribute the secret s among the n shareholders in such a way that for any subset $\mathcal{B}$ of t participants, the secret $s = \sum_{P_i \in \mathcal{B}} s_i \bmod (q-1)$ (all calculations are performed in $GF(q)$).

Suppose that user A wants to send a message $m \in GF(q)$ to the group. A first chooses at random an integer $k \in GF(q)$ and computes the cryptogram $c = (g^k, my^k)$ for the message m.

Assume that $\mathcal{B}$ is an authorized subset, so it contains at least t participants. The first stage of decryption is executed separately by each participant $P_i \in \mathcal{B}$. P_i takes the first part of the cryptogram and computes $(g^k)^{s_i} \bmod q$. The result is sent to the combiner, who computes $y^k = g^{ks} = \prod_{i \in \mathcal{B}} g^{ks_i}$, and decrypts (using the multiplicative inverse y^{-k}) the cryptogram

$$m \equiv my^k \times y^{-k} \bmod p \, .$$

Group decryption can also be based on a combination of the RSA cryptosystem [70] and Shamir's threshold scheme. The scheme described by Desmedt and Frankel [36] works as follows. The dealer D computes the modulus $N = pq$, where p, q are strong primes, that is, $p = 2p' + 1$ and $q = 2q' + 1$ (where p' and q' are large and distinct primes). The dealer selects at random an integer e such that e and $\lambda(N)$ are coprime ($\lambda(N)$ is the least common multiple of two integers $p-1$ and $q-1$, so $\lambda(N) = 2p'q'$). Next D publishes e and N as the public parameters of the system, but keeps p, q, and d secret (d satisfies the congruence $ed = 1 \bmod \lambda(N)$). It is clear that computing d is easy for the dealer who knows $\lambda(N)$, but is difficult—equivalent to the factoring of N—to someone who does not know $\lambda(N)$. The dealer then uses Shamir's scheme to distribute the secret $s = d - 1$ amongst n participants. The shares are denoted s_i and any t co-operating participants (the set $\mathcal{B}$) can retrieve the secret. We have,

$$s = \sum_{i \in \mathcal{B}} s_i \bmod \lambda(N) \, .$$

Group decryption of the cryptogram $c \equiv m^e \bmod N$ starts from individual computations. Each $P_i \in \mathcal{B}$ calculates its partial cryptogram $c^{s_i} \bmod N$. All the partial cryptograms are sent to the combiner who recovers the message

$$m = \prod_{i \in \mathcal{B}} c^{s_i} \times c \equiv c^{\left(\sum_{i \in \mathcal{B}} s_i + 1\right)} \equiv c^d \equiv \left(m^e\right)^d \bmod N \, .$$

Again the secret $s = d - 1$ is never exposed during the decryption.

Ghodosi, Pieprzyk, and Safavi-Naini [46] proposed a solution to the problem of group decryption which does not require a dealer. It uses the RSA cryptosystem and Shamir's threshold scheme. The system works under the assumption that all participants from $\mathcal{P} = \{P_1, \ldots, P_n\}$ have their entries in a public registry (white pages). The registry provides the public parameters of a given participant. A participant P_i has as its RSA entry N_i, e_i in the registry, and this entry cannot be modified by an unauthorized person.

The sender first selects the group $\mathcal{P} = \{P_1, \ldots, P_n\}$. For the message m $(0 < m < \prod_{i=1}^{n} N_i)$, the sender computes

$$m_i \equiv m \bmod N_i$$

for $i = 1, \ldots, n$. Next the sender selects at random a polynomial $f(x)$ of degree at most t over $GF(p)$, where $p < \min_i N_i$. Let

$$f(x) = a_0 + a_1 x + \cdots + a_{t-1} x^{t-1} \, .$$

The sender computes $c_i = f(x_i)$ for public x_i, $k = f(0)$, $c_i^{e_i} \bmod N_i$, and $m_i^k \bmod N_i$ $(i = 1, \ldots, n)$. Finally, the sender merges $c_i^{e_i} \bmod N_i$ into C_1 and $m_i^k \bmod N_i$ into C_2 using the Chinese Remainder Theorem. The sender broadcasts the tuple (N, p, t, C_1, C_2) as the cryptogram.

The participants check whether they are the intended recipients, by finding the gcd (N_i, N) for instance. Note that the sender can give the list of all participants instead of the modulus N. A participant P_i first recovers the pair $(c_i^{e_i} \bmod N_i)$ and $(m_i^k \bmod N_i)$ from C_1 and C_2, respectively. Using its secret key d_i, the participant retrieves c_i. The c_i are now broadcast so that each participant can reconstruct $f(x)$ and find $k = f(0)$. Note that none of the participants can cheat, as it can be readily verified whether c_i' satisfies the congruence

$$c_i'^{e_i} \equiv C_1 \bmod N_i \ .$$

Knowing k, each participant finds the message $m_i \equiv C_2^{k^{-1}} \bmod N_i$. Although k is public, only participant P_i can find $k^{-1} \times k \bmod (p_i - 1)(q_i - 1)$ from his knowledge of the factorization of $N_i = p_i q_i$. Lastly, all the partial messages are communicated to the combiner who recovers the message m by the Chinese Remainder Theorem.

41.4 Signature Schemes

A signature scheme consists of two algorithms: signature generation and signature verification. Each of these algorithms can be collaboratively performed. A shared-generation scheme allows a group of signers to collaboratively sign a document. In a signature scheme with shared verification, the signature verification requires collaboration of a group. We examine the two types of systems and note that the two can be combined if necessary.

Shared Generation Schemes

In these schemes a signer group P of n participants has a public/private key pair. The private part is shared among members of the group such that each member has part of the private key that is not known to anyone else. The signature scheme is usually based on one of the well known signature schemes such as ElGamal, Schnorr, RSA, and Fiat–Shamir.

The group is created with an access structure that determines the authorized groups of signers. A special case of shared-generation schemes are the *multisignature* schemes, in which collaboration of all members in P is necessary. Most systems proposed for shared generation are of the multisignature type, or its generalization, (t, n)*-threshold signature*. In the latter type of signature each subgroup p, $p \subset P$ of size t, can generate the signature.

A shared-generation scheme can be *sequential* or *simultaneous*. In a sequential scheme each member of the group signs the message and forwards it to the next group member. In some schemes, after the first signer the message is not readable and all subsequent signers must blindly sign the message. In a simultaneous scheme, each group member forms a partial signature which is sent to a combiner who forms the final signature.

There are a number of issues that differentiate shared-generation systems.

1. Mutually trusted party: a system may need a mutually trusted party who is usually active during the key generation phase; it chooses the group secret key and generates secrets for all group members. In systems without a trusted party, each signer produces his secret key and participates in a protocol with other signers to generate the group public key.
2. The security of most signatures schemes is based on the intractability of one of the following problems: discrete logarithm or integer factorization. Shared-generation schemes based on ElGamal and Schnorr signature schemes use the former, while those based on RSA and Fiat–Shamir use the later.

3. Using many/few interactions for producing signature. The amount of interaction between the signers and the trusted third party varies in different schemes.

There are properties—some essential and some desirable—that a shared-generation scheme must satisfy. The *essential* properties are as follows:

A1 Signature generation must require collaboration of all members of the authorized group and no signer in the group should be able to deny his signature. Verification must be possible by any outsider.

A2 An unauthorized group should not be able to forge the signature of an authorized group. It should not also be possible for an authorized group to forge the signature of another authorized group.

A3 No secret information should be derivable from the released group and partial signatures.

The *desirable* properties are as follows:

1. Each signer must have the same power and be able to see the message that he is signing.
2. The order of signing in a sequential scheme should not be fixed.
3. The size of the multisignature should be comparable to, preferably the same as, the size of the individual signature.

For a (t, n) threshold signature scheme (A1) and (A2) reduce to

B1 From any t partial signature the group signature should be easily derivable.

B2 Knowledge of $t-1$ or fewer partial signatures should not reduce the chance of forgery of an unauthorized group.

Constructions

The earliest proposals for shared-generation schemes are by Itakura [51] and by Boyd [16]. Boyd's scheme is a (n, n)-threshold group signature based on RSA, in which if $n > 2$, most participants must blindly sign the message.

Threshold RSA Signature

Desmedt and Frankel [36] construct a simultaneous threshold (t, n) RSA signature that requires a trusted third party to generate and distribute the group public key and the secret keys of the signers.

Their scheme works as follows. In the initialization stage, a trusted KDC (dealer) selects at random a polynomial of degree $t-1$: $f(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_{t-1}x^{t-1}$. The group secret key k is fixed as $a_0 = f(0)$. The dealer gives $y_i = f(x_i)$ to participant P_i, for each i, via a secure channel. The computations are performed in $Z_{\lambda(N)}$, where $\lambda = 2p'q'$ and $p = 2p' + 1$, $q = 2q' + 1$. To sign a message m $(0 \le m < N)$ each participant $P_i \in B$ calculates its partial signature $s_i = m^{k_i} \bmod N$ and transmits the result to the combiner. The combiner computes the signature S of the message m according to the following:

$$S = m \times \prod_{P_i \in B} s_i = m \times \prod_{\substack{P_i \in B \\ i=1}}^{t} m^{k_i} = m \times m^{d-1} = m^d \bmod N .$$

The signature verification is similar to the conventional RSA signature scheme.

Threshold Signature Based on Discrete Logarithm

Ohta and Okamoto [67] propose a sequential multisignature scheme based on the Fiat–Shamir signature scheme. In their scheme, the order of signing is not restricted but the scheme requires a trusted center for key generation.

A variation of group signature is *undeniable group signature*, in which verification requires collaboration of signers. The signature scheme has a "commitment phase" during which t group members work together to sign a message, and a "verification phase" during which all signers work together to prove validity of the signature to an outsider. Harn and Yang [48] propose two (t, n)-threshold schemes, with $t = 1$ and $t = n$. Their schemes do not require a trusted third party and the algorithm is based on the discrete logarithm problem.

Harn [47] proposes three simultaneous multisignature schemes, based on the difficulty of discrete logarithm. Two of these schemes do not require a trusted third party. We briefly review one of the schemes. We use the notation of Harn [47].

Let KDC denote the key distribution center. The KDC selects

1. p, a large prime, in the range $2^{511} \leq p \leq 2^{512}$
2. q, a prime divisor of $p - 1$
3. $\{a_i, i = 0, \ldots, t-1\}$, and $f(x) = a_0 + a_1 x + \cdots + a_{t-1}x^{t-1} \pmod{q}$ where $0 < a_i < q$
4. α, where $\alpha = h^{(p-1)/q} \pmod{p} > 1$. α is a generator with order q in GF(p); p, q and α are made public.

The KDC computes the group public key $y = \alpha^{f(0)} \bmod p$, where $f(0)$ is the group secret key. The KDC also computes public keys for all group members as

$$y_i = \alpha^{f(x_i)} \pmod{p}, \quad \text{for} \quad i = 1, 2, \ldots, n$$

where $f(x_i) \bmod q$ is the share of participant i from the group secret key. (Note that, since α is a generator with order q in $GF(p)$, $\alpha^r \bmod p = \alpha^{r \bmod q}$, for any nonnegative integer r.)

In order to generate the group signature on a message m, each participant of a group B ($|B| \geq t$) randomly selects an integer, $k_i \in [1, q-1]$, and computes a public value, $r_i = \alpha^{k_i} \bmod p$ and broadcasts r_i to all members in B. Knowing all the r_i ($i \in B$), each member of the group B computes

$$r = \prod_{i \in B} r_i \pmod{p}.$$

Participant i computes his partial signature as

$$s_i = m' \times f(x_i) \times \left(\prod_{\substack{i,j \in B \\ i \neq j}} \frac{x_j}{(x_i - x_j)} \right) - k_i \times r \pmod{p}$$

where $H(m) = m'$ (H is a one-way and collision free hash function) and transmits (r_i, s_i) to a designated combiner.

Once the combiner receives the partial signature (r_i, s_i), it is verified using

$$y_i^{(m')^{\left(\prod_{\substack{i,j \in B \\ i \neq j}} \frac{x_j}{(x_i - x_j)}\right)}} = \alpha^{s_i} r_i^r \pmod{p}.$$

If all the partial signatures are verified, then the combiner calculates the group signature (r, s) on message m, where $s = \sum_{i \in B} s_i \pmod{q}$.

An outsider who receives the signature (r, s) on the message m can verify the validity of the signature using the check $y^{m'} = \alpha^s r^r \pmod{p}$. This check works because

$$f(0) = \sum_{i \in B} f(x_i) \prod_{\substack{i,j \in B \\ i \neq j}} \frac{x_j}{(x_i - x_j)} \pmod{q}$$

and thus,

$$\begin{aligned} y^{m'} &= \left(\alpha^{f(0)}\right)^{m'} = (\alpha^{\left(\sum_{i \in B} f(x_i) \prod_{\substack{i,j \in B \\ i \neq j}} \frac{x_j}{(x_i - x_j)}\right)^{m'}} \\ &= \prod_{i \in B} y_i^{(m') (\prod_{\substack{i,j \in B \\ i \neq j}} \frac{x_j}{(x_i - x_j)})} = \prod_{i \in B} \left(r_i^r a_i^s\right) = r^r \alpha^s \ . \end{aligned}$$

An interesting security problem in these schemes, as discussed by Desmedt and Frankel [36] and by Harn [47], is that if more than t signers collaborate they can find the secrets of the system with a high probability, and thus identify the rest of the shareholders. Possible solutions to this problem in the case of discrete logarithm based schemes can be found in a paper by Li, Hwang and Lee [57].

A concept related to threshold signature is t-resilient digital signatures. In these schemes, n members of a group can collaboratively sign a message even if there are t dishonest members. Moreover no subset of t dishonest members can forge a signature.

Desmedt [34] shows that a t-resilient signature scheme with no trusted center can be constructed for any signature scheme using a general multiparty protocol. Cerecedo, Matsumoto and Imai [24] present efficient protocols for the shared generation of signatures based on the discrete logarithm problem, Schnorr's scheme and variants of the ElGamal scheme. Their protocols are based on an efficient multiparty protocol for shared computation of products and they do not need a trusted party. Park and Kurosawa [68] discuss a (t, n)-threshold scheme based on the discrete logarithm, more precisely a version of Digital Signature Standard (DSS), which does not require multiplication and only uses linear combination for combination of shares.

Chang and Liou [25] and Langford [56] propose other signature schemes based on the discrete logarithm problem.

Shared Verification of Signatures

Signature schemes with shared verification are not commonly found in the literature.

De Soete, Quisquater and Vedder [81] propose a system for shared verification of signatures. But their system is not really a signature scheme in the sense that it does not produce a signature for every message. Each user has a secret that enables him to verify himself to others. It requires at least two verifiers for the secret to be verified.

Laih and Yen [55] argue that in some cases it might be necessary to sign a message such that only specified groups of participants can verify the signed message. The main requirements of such schemes are:

1. A can sign any message M for any specified group B.
2. Only the specified group can validate the signature of A. No other group, except B, can validate the signature of A on M.
3. B should not be able to forge A's signature on M for another user C even if B and C conspire.
4. No one should be able to forge A's signature on another message M'.

5. If A disavows his signature, it must be possible for a third party to resolve the dispute between A and B.

The scheme proposed by Laih and Yen [55] is based on Harn's scheme, which is an efficient ElGamal type shared-generation scheme. In the proposed scheme a group of signers can create a digital shared-generation scheme for a specified group, who can collectively check the validity of the signature. The secret key of the users is chosen by the users themselves and each group has a public key for signature generation or verification. Since the private key of the verifiers is not known, dispute settlement by a third party requires an extra protocol between the third party and the verifiers.

41.5 Quantum Key Distribution—Quantum Cryptography

While classical cryptography employs various mathematical techniques to restrict eavesdroppers from learning the contents of encrypted messages, in quantum mechanics the information is protected by the laws of physics. In classical cryptography absolute security of information cannot be guaranteed. However on the quantum level there is a law called the Heisenberg uncertainty principle. This states that even the most refined measurement on a quantum object cannot reveal everything about the object before the measurement was made. This is because the object may be altered by simply taking the measurement. The Heisenberg uncertainty principle and quantum entanglement can be exploited in a system of secure communication, often referred to as "quantum cryptography" [6]. Quantum cryptography provides means for two parties to exchange a enciphering key over a private channel with complete security of communication.

There are at least three main types of quantum cryptosystems for the key distribution:

- *BB protocol*: Cryptosystems with encoding based on two noncommuting observables proposed by Wiesner [89], and by Bennett and Brassard [5].
- *EPR-type*: Cryptosystems with encoding built upon quantum entanglement and the Bell Theorem proposed by Ekert [42].
- *B-type*: Cryptosystems with encoding based on two non-orthogonal state vectors proposed by Bennett [3].

A *BB* quantum cryptosystem can be explained using the following simple example. The system includes a transmitter, Alice, and a receiver, Bob. Alice may use the transmitter to send photons in one of four polarizations: 0, 45, 90, or 135 degrees. Bob at the other end uses the receiver to measure the polarization. According to the laws of quantum mechanics, Bob's receiver can distinguish between rectilinear polarizations (0 and 90), or it can quickly be reconfigured to discriminate between diagonal polarizations (45 and 135); it can never, however, distinguish both types. The key distribution requires several steps. Alice sends photons with one of the four polarizations which are chosen at random. For each incoming photon, Bob chooses at random the type of measurement: either the rectilinear type or the diagonal type. Bob records the results of the measurements but keeps them secret. Subsequently Bob publicly announces the type of measurement (but not the results) and Alice tells the receiver which measurements were of the correct type. Alice and Bob (the sender and the receiver) keep all cases in which Bob's measurements were of the correct type. These cases are then translated into bits (1's and 0's) and thereby become the key. An eavesdropper is bound to introduce errors to this transmission because he/she does not know in advance the type of polarisation of each photon and quantum mechanics does not allow him/her to acquire sharp values of two non-commuting observables (here rectilinear and diagonal polarisations). The two legitimate users of the quantum channel, Alice and Bob, test for eavesdropping by revealing a random subset of the key bits and checking (in public) the error rate. Although they cannot prevent eavesdropping, they will never be fooled by an eavesdropper because any effort to tap the channel, however subtle and sophisticated, will be detected. Whenever they are not happy with the security of the channel they can try to set up the key

distribution again. The mechanism of *privacy amplification* is used to finally distill a secret key between Alice and Bob from these interactions, cf. Bennett, Brassard and Robert [7].

The basic idea of cryptosystems of *EPR*-type is as follows. A sequence of correlated particle pairs is generated, with one member of each pair being detected by Alice and the other by Bob (for example, a pair of so-called Einstein–Podolsky–Rosen photons, whose polarizations are measured by Alice and Bob). An eavesdropper on this communication would have to detect a particle to read the signal, and retransmit it in order for his/her presence to remain unknown. However, the act of detection of one particle of a pair destroys its quantum correlation with the other. Thus Alice and Bob can easily verify whether this has been done, without revealing the results of their own measurements, by communication over an open channel.

In our example we will consider two types of measurement: we consider \ and / to be one type (diagonal) and | and − to be the other (rectilinear).

1. Alice's polarization	|	\	−	|	/	−	−	−	−	\	/
2. Bits Alice sent	0	0	1	0	0	1	1	1	0	1	1
3. Bob's polarization	|	\	|	\	|	|	/	|	−	\	−
4. Bits Bob registered	0	0	1	1	0	1	0	1	0	1	1
5. Alice states publicly whether Bob's polarization was correct or not	Yes	Yes	Yes	No	No	Yes	No	Yes	Yes	Yes	No
6. Alice's remaining bits	0	0	1	*x*	*x*	1	*x*	1	0	1	*x*
7. Bob's remaining bits *x* means discard	0	0	1	*x*	*x*	1	*x*	1	0	1	*x*
8. Alice and Bob	0	0	1	*x*	*x*	1	*x*	1	0	1	*x*
compare bits	0	0	1	*x*	*x*	1	*x*	1	0	1	*x*
chosen at random	OK							OK		OK	
9. Alice's remaining bits	*x*	0	1	*x*	*x*	1	*x*	*x*	0	*x*	*x*
10. Bob's remaining bits	*x*	0	1	*x*	*x*	1	*x*	*x*	0	*x*	*x*
11. The key		0	1			1			0		

Shor's Quantum Factoring Algorithm

Mathematicians have tried hard to solve the key distribution problem, and in the 1970s a clever mathematical discovery called "public key" systems gave an elegant solution. Public-key cryptosystems avoid the key distribution problem, but unfortunately their security depends on unproven mathematical assumptions, such as the difficulty of factoring large integers (RSA, the most popular public key cryptosystem, gets its security from the difficulty of factoring large numbers). An enemy who knows your public key can in principle calculate your private key because the two keys are mathematically related; however, the difficulty of computing the private key from the respective public key is exactly that of factoring big integers.

Difficulty of factoring grows rapidly with the size, i.e., number of digits, of the number we want to factor. To see this, take a number N with ℓ decimal digits ($N \approx 10^{\ell}$) and try to factor it by dividing it by $2,\ 3,\ \ldots, \sqrt{N}$ and checking the remainder. In the worst case approximately $\sqrt{N} \approx 10^{\ell/2}$ divisions may be needed to solve the problem—an exponential increase as a function of ℓ. Now imagine computer capable of performing 10^{10} divisions per second. The computer can then factor any number N, using the trial division method, in about $\sqrt{N}/10^{10}$ seconds. Take a 100-digit number N, so that $N \approx 10^{100}$. The computer will factor this number in about 10^{40} seconds, much longer than 10^{17} seconds—the estimated age of the universe!

It seems that factoring big numbers will remain beyond the capabilities of any realistic computing devices and unless mathematicians or computer scientists come up with an efficient factoring algorithm public-key cryptosystems will remain secure. Or will they? As it turns out we know that this is not the case; the classical, purely mathematical, theory of computation is not complete simply because it does not describe all physically possible computations. In particular it does not describe computations that can be performed by quantum devices. Indeed, recent work in quantum computation shows that a quantum computer can factor much faster than any classical computer.

Quantum computers can compute faster because they can accept as input not just one number, but a coherent superposition of many different numbers, and subsequently perform a computation (a sequence of unitary operations) on all of these numbers simultaneously. This can be viewed as a massive parallel computation, but instead of having many processors working in parallel we have only one quantum processor performing a computation that affects all components of the state vector. To see how it works let us describe Shor's factoring using a quantum computer composed of two quantum registers.

Consider two quantum registers, each register being composed of a certain number of two-state quantum systems which we call "qubits" (quantum bits). We take the first register and place it in a quantum superposition of all the possible integer numbers it can contain. This can be done by starting with all qubits in the 0 states and applying a simple unitary transformation to each qubit which creates a superposition of 0 and 1 states:

$$|0\rangle \longrightarrow \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) \,. \tag{41.5}$$

Imagine a two-qubit register, for example. After this procedure the register will be in a superposition of all four numbers it can contain,

$$\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) = \frac{1}{2}(|00\rangle + |01\rangle + |10\rangle + |11\rangle) \tag{41.6}$$

where 00 is binary for 0, 01 binary for 1, 10 binary for 2 and finally 11 which is binary for 3.

Then we perform an arithmetical operation that takes advantage of quantum parallelism by computing the function $F_N(x)$ for each number x in the superposition. The values of $F_N(x)$ are placed in the second register so that after the computation the two registers become entangled:

$$\sum_x |x\rangle|F_N(x)\rangle \tag{41.7}$$

(we have ignored the normalization constant). Now we perform a measurement on the second register. The measurement yields a randomly selected value $F_N(k)$ for some k. The state of the first register immediately after the measurement, due to the periodicity of $F_N(k)$, is a coherent superposition of all states $|x\rangle$ such that $x = k,\ k+r,\ k+2r,\ \ldots$, i.e., all x for which $F_N(x) = F_N(k)$. The value k is randomly selected by the measurement: therefore, the state of the first register is subsequently transformed via a unitary operation which sets any k to 0 (i.e., $|k\rangle$ becomes $|0\rangle$ plus a phase factor) and modifies the period from r to a multiple of $1/r$. This operation is known as the quantum Fourier transform. The first register is then ready for the final measurement and yields an integer which is the best whole approximation of a multiple of $1/r$. From this result r, and subsequently factors of N, can be easily calculated (see below). The execution time of the quantum factoring algorithm can be estimated to grow as a quadratic function of ℓ, and numbers 100 decimal digits long can be factored in a fraction of a second!

When the first quantum factoring devices are built, the security of public-key cryptosystems will vanish. The mathematical solution to the key distribution problem is shattered by the power of quantum computation. Does it leave us without any means to protect our privacy? Fortunately quantum mechanics after destroying classical ciphers comes to rescue our privacy and offers its own solution to the key distribution problem.

The main reference for this brief account of quantum cryptanalysis is the paper by Shor [77]. A comprehensive exposition of Shor's algorithm for factoring on a quantum computer, together with some

relevant background in number theory, computational complexity theory and quantum computation including remarks about possible experimental realizations has been prepared for the Review of Modern Physics by Artur Ekert and Richard Jozsa and can be obtained via electronic (postscript file) or postal service.

Factoring

Quantum factoring of an integer N is based on calculating the period of the function $F_N(x) = a^x \pmod N$. We form the increasing powers of a until they start to repeat with a period we denote r. Once r is known the factors of N can be found using the Euclidean algorithm to find the greatest common divisor of $a^{r/2} \pm 1$ and N.

Suppose we want to factor 91. Let us take a number at random, say 28, and raise it to the powers 2, 3, ... After 12 iterations we find we have the number 28 repeated and so we use $r = 12$. Hence we want to find gcd(91, $28^6 \pm 1$), which we find to be 1 and 13, respectively. From here we can factorize 91. Classically, calculating r is as difficult as trying to factor N and the execution time is exponential in the number of digits in N. Quantum computers can find r in time which grows only as a quadratic function of the number of digits of N.

Practicalities

The idea of a quantum computer is simple, however its realization is not. Quantum computers require a coherent, controlled evolution for a period of time which is necessary to complete the computation. Many view this requirement as an insurmountable experimental problem; however, others believe that technological progress will sooner or later make such devices feasible. In an ordinary, classical computer, all the bits have a definite state at a given instant in time, say 0 1 1 0 0 0 1 0 1 0 .... However, in a quantum computer the state of the bits is described by a wave equation such as

$$\Psi = a|0\,1\,1\,0\,0\,0\,1\,0\,1\ \ldots\rangle + b|1\,1\,1\,0\,0\,0\,0\,0\,1\ \ldots\rangle\,. \tag{41.8}$$

The coefficients a, b, ... are complex numbers and the probability that the computer is in state 0 1 1 0 0 0 1 0 1 ... is $|a|^2$, that it is in the state 1 1 1 0 0 0 0 0 1 ... is $|b|^2$, and so on. However, describing the state of the computer by a wave function does not merely imply the ordinary uncertainties that we describe using probabilities. For instance, the phases of the complex coefficients a, b, ... have genuine significance: they can describe interference between different states of the computer, which is very useful for quantum computation. The quantum wave function declares that the computer exists in all of its states simultaneously so long as that state is not measured; when we do choose to measure it, a particular state will be observed with the prescribed probability.

DiVincenzo [40] shows how to construct the quantum analogue of the one-bit NOT, or inverter gate, with spectroscopic techniques that have been well known in physics for over 50 years. He then shows that the 1956 Feher procedure for polarization transfer in electron-nucleus double resolution has the protocol for making the two-qubit (quantum bit) XOR gate implicit within it. Finally he shows how the three-qubit AND operation can be performed using three XOR gates and four single-qubit rotations.

In practice we do not know how to "wire up" the components of the XOR gates, and so a quantum computer is still just a theoretical possibility. However, if a quantum computer could be built it would be the end of classical cryptography as we know it.

The field of quantum cryptography was pioneered by Stephen Wiesner [89]. Around 1970 at Columbia University in New York he showed how quantum effects could in theory be use to produce "quantum bank notes" that are immune to counterfeiting. The first feasible cryptosystems were designed between 1982 and 1984 by the American physicist Charles H. Bennett of IBM's Thomas J. Watson Research Center in Yorktown Heights in the U.S. and by the Canadian expert in cryptography, Gilles Brassard from the University of Montreal [5]. In 1989 David Deutsch and Artur Ekert [37, 38] at Oxford University developed

the idea along slightly different lines. Their system is based on another aspect of quantum theory called quantum correlation.

In the early 1980s, secure quantum key distribution based on both the Heisenberg uncertainty principle and quantum correlations evolved into a testable system for practical use, though this is by no means easy to set up. The first apparatus, constructed by Bennett, Brassard and colleagues in 1989 [4] at IBM's research center, was capable of transmitting a secret key over a distance of approximately 30 centimeters.

Since then, other researchers have looked at systems based on correlations of another quantum property of light, called phase. Phase is measure of how far a photon has gone in its cycle of vibration. Information about the key is encoded in this property of phase instead of polarization. This has the advantage that with current technology, phase is easier to handle over a long distance. Since 1991 John Rarity and Paul Tapster of Britain's [43, 86, 88] Defense Research Agency in Malvern have been developing a system to increase the transmission distance. They have designed and tested an optical system good enough to transmit photons that stay correlated in phase over several hundred meters. Rarity and Tapster believe the distances could increase to several kilometers, once advances in technology give new optical fibers and semiconductor photo detectors that allow better transmission and detection, and thereby reduce background errors.

In theory, cryptosystems based on quantum correlations should also allow quantum keys to be stored, by storing photons without performing any measurements. At present, however, photons cannot be kept correlated longer than a small fraction of a second, so they are not a good medium for information storage. But a fraction of a second is long enough for a photon to cover a long distance, so photons are suitable for sending information and for key distribution.

41.6 Research Issues and Summary

In this chapter we discussed secret sharing, threshold cryptography, signature schemes, and finally quantum key distribution and quantum cryptography. As in the previous chapter, we have focused only on the essentials. The list of references expands our exposition by listing items that represent the current research activity in these topics. We give a brief summary of our exposition.

The central problem in secret sharing is how to distribute parts of a secret, to a group individuals, in such a way that only the predesignated individuals can recreate the secret. As well as having direct application in key management, secret sharing schemes are also components of other cryptographic constructions.

Threshold cryptography is concerned with situations where the authority to initiate or perform cryptographic operations is distributed amongst a group of individuals. Many standard constructions of single-user cryptography have counterparts in threshold cryptography.

A signature scheme is an algorithm for generating and verifying (electronic) signatures for documents. A subclass of signature schemes deals with the shared-generation and shared-verification of signatures, where a collaborating group of individuals is required to perform these actions.

A new paradigm of security has recently been introduced into cryptography with the emergence of the ideas of quantum key distribution and quantum cryptography. As opposed to classical cryptography, where various mathematical techniques are used to restrict eavesdroppers from learning the contents of encrypted messages, in quantum key distribution the information is protected by the uncertainty principle of quantum mechanics. A non-Turing model of computation can be also be based on the formalism of quantum mechanics. Various computations, notably the factorization of integers into primes, could be done on such machines with unprecedented parallelism.

41.7 Defining Terms

Access structure: A formal specification of the participants in a secret-sharing scheme which are able to recreate a shared key from their portions of the key.

Geometric secret sharing: A realization of a secret-sharing scheme using finite geometry. Usually either affine or projective geometries are used.

Group cryptosystems: A recent development where cryptographic operations are performed by groups instead of individuals.

Perfect secret sharing: In such a scheme it is impossible to deduce any partial information about a shared key from less than the critical number of shares of the key.

Quantum computation: A theoretical model of computer based on the principles of quantum mechanics. It is known that factoring of integers could be done in polynomial time on such a machine.

Quantum cryptosystem: Methods of securely exchanging private keys across an insecure channel in which the principles of quantum mechanics are used.

Secret sharing: Protecting a secret key by distributing it in such a way that only the authorized individuals can recreate the key.

Signature schemes: An algorithm that generates and verifies a cryptographic signature.

Threshold scheme: A secret-sharing scheme with a uniform access structure in which any collection of shareholders greater than a given threshold can recreate the secret.

Acknowledgments

We thank Anish Mathuria and Hossein Ghodosi for all their comments and suggestions, which have greatly helped us improve our exposition.

References

[1] Benaloh, J.C., Secret sharing homomorphisms: keeping shares of a secret secret. *Proc. Crypto'86,* LNCS Vol. 263, 251–260, Springer-Verlag, Berlin, 1987.

[2] Benaloh, J. and Leichter, J., Generalized secret sharing and monotone functions, *Proc. CRYPTO'88,* LNCS, Vol. 403, 27–35, Springer-Verlag, 1989.

[3] Bennett, C.H., Quantum cryptography using any two nonorthogonal states, *Phys. Rev. Lett.,* 68, 3121–3124, 1992.

[4] Bennett, C.H., Bessette, F., Brassard, G., Salvail, L., and Smolin, J., Experimental quantum cryptography, *J. Cryptology,* 5, 3–28, 1992.

[5] Bennett, C.H. and Brassard, G., Quantum cryptography: Public-key distribution and coin tossing, *Proc. IEEE Int. Conference on Computers, Systems and Signal Processing,* IEEE, 175–179, New York, 1984.

[6] Bennett, C.H., Brassard, G., and Ekert, A.K., Quantum cryptography, *Scientific American,* 50–57, Oct. 1992.

[7] Bennett, C.H., Brassard, G., and Robert, J.-M., Privacy amplification by public discussion, *SIAM Journal on Computing,* 17(2), 210–229, 1988.

[8] Bertilson, M. and Ingemarsson, I., A construction of practical secret sharing schemes using linear block codes, *Proc. AUSCRYPT'92,* LNCS, Vol. 718, 67–79, Springer-Verlag, 1993.

[9] Beth, T., Jungnickel, D., and Lenz, H., *Design Theory,* Cambridge University Press, Cambridge, 1986.

[10] Beth, T., Multifeature security through homomorphic encryption, *Proc. Asiacrypt'94,* LNCS Vol. 917, 3–17, Springer-Verlag, Berlin, 1993.

[11] Beutelspacher, A., Enciphered geometry: Some applications of geometry to cryptography, *Discrete Applied Mathematics,* 37, 59–68, 1988.

[12] Blakley, G.R., Safeguarding cryptographic keys, *Proc. N.C.C.,* AFIPS Conference Proceedings 48, Vol. 48, 313–317, 1979.
[13] Blakley, B., Blakley, G.R., Chan, A.H., and Massey, J.L., Threshold schemes with disenrollment, *Proc. Crypto'92,* LNCS Vol. 740, 546–554, Springer-Verlag, Berlin, 1992.
[14] Blakley, G.R. and Meadows, C., Security of ramp schemes, *Proc. CRYPTO'84,* LNCS, Vol. 196, 242–268, Springer-Verlag, 1985.
[15] Blundo, C., De Santis, A., Stinson, D.R., and Vaccaro, V., Graph decompositions and secret sharing schemes, *Journal of Cryptology,* 8(1), 39–64, 1995.
[16] Boyd, C., Digital Multisignatures, In *Cryptography and Coding,* Beker, H. and Piper, F., Eds., 241–246. Clarendon Press, 1989.
[17] Brassard, G., *Modern Cryptology: A Tutorial,* Springer, Berlin, 1988.
[18] Brickell, E.F., Some ideal secret sharing schemes, *Journal of Combinatorial Mathematics and Combinatorial Computing,* 6, 105–113, 1989.
[19] Brickell, E.F. and Davenport, D.M., On the classification of ideal secret sharing schemes, *Journal of Cryptology,* 4, 123–134, 1991.
[20] Brickell, E.F. and Stinson, D.R., The detection of cheaters in threshold schemes, *Proc. Crypto'88,* LNCS, Vol. 403, 564–577, Springer-Verlag, Berlin, 1990.
[21] Brickell, E.F. and Stinson, D.R., Some improved bounds on the information rate of perfect secret sharing schemes, *Journal of Cryptology,* 5, 153–166, 1992.
[22] Brown, L., Kwan, M., Pieprzyk, J. and Seberry, J., Improving resistance to differential cryptanalysis and the redesign of LOKI, in *Advances in Cryptology—Proceedings of ASIACRYPT '91,* Imai, R.R.H. and Matsumoto, T., Eds., Vol. 739 of *Lecture Notes in Computer Science*, 36–50, Springer-Verlag, 1993.
[23] Capocelli, R., DeSantis, A., Gargano, L., and Vaccaro, V., On the size of shares for secret sharing schemes, *Proc. CRYPTO '91,* LNCS Vol. 576, 101–113, Springer-Verlag, 1992.
[24] Cerecedo, M., Matsumoto, T., and Imai, H., Efficient and secure multiparty generation of digital signatures based on discrete logarithms, *IEICE Trans. Fundamentals,* E76-A(4), 531–545, Apr. 1993.
[25] Chang, C.-C. and Liou, F.-Y., A digital multisignature scheme based upon the digital signature scheme of a modified ElGamal public key cryptosystem, *Journal of Information Science and Engineering,* 10, 423–432, 1994.
[26] Charnes, C., Pieprzyk, J., and Safavi-Naini, R., Families of threshold schemes, *Proc. 1994 IEEE International Symposium on Information Theory,* Trondheim, Norway, 1994.
[27] Charnes, C., Pieprzyk, J., and Safavi-Naini, R., Conditionally secure secret sharing schemes with disenrollment capability, 2*nd ACM Conf. on Computer and Communications Security,* Nov. 2-4 1994, Fairfax, 89–95, VA, ACM 1994.
[28] Charnes, C. and Pieprzyk, J., Disenrollment capability of conditionally secure secret sharing schemes, *Proc. International Symposium on Information Theory and Its Applications (ISITA'94),* Nov. 20-25 1994, 225–227, Sydney, Australia, IEA, NCP 94/9 1994.
[29] Charnes, C. and Pieprzyk, J., Cumulative arrays and generalised Shamir secret sharing schemes, 17*th Australasian Computer Science Conference,* Australian Computer Science Communications, 16(1), 519–528, 1994.
[30] Charnes, C. and Pieprzyk, J., Generalised cumulative arrays and their application to secret sharing schemes, 18*th Australasian Computer Science Conference,* Australian Computer Science Communications, 17(1), 61–65, 1995.
[31] Chaudhry, G. and Seberry, J., Secret sharing schemes based on Room squares, *Combinatorics, Complexity and Logic,* DMTCS'96, 158–167, Springer-Verlag, Berlin, 1996.
[32] Chor, B., Goldwasser, S., Micali, S., and Awerbuch, B., Verifiable secret sharing and achieving simultaneity in the presence of faults, *Proc. 26th IEEE Symp. Found. Comp. Sci.,* 383–395, 1985.

[33] Cooper, J., Donovan, D., and Seberry, J., Secret sharing schemes arising from Latin squares, *Bull. ICA,* 12, 33–43, 1994.

[34] Desmedt, Y., Society and group oriented cryptography: A new concept, In *Advances in Cryptology—Proceedings of CRYPTO '87,* Pomerance, C., Ed., volume 293 of *Lecture Notes in Computer Science,* 120–127. Springer-Verlag, 1988.

[35] Desmedt, Y. and Frankel, Y., Threshold cryptosystems, in *Advances in Cryptology—Proceedings of CRYPTO '89,* Brassard, G., Ed., Vol. 435 of *Lecture Notes in Computer Science*, 307–315, Springer-Verlag, 1990.

[36] Desmedt, Y. and Frankel, Y., Shared generation of authenticators and signatures, in *Advances in Cryptology—Proceedings of CRYPTO'91,* Feigenbaum, J., Ed., Vol. 576 of *Lecture Notes in Computer Science*, 457–469, Springer-Verlag, 1992.

[37] Deutsch, D., Quantum theory, the Church-Turing principle and the universal quantum computer, *Proc. R. Soc. London, Ser. A,* 400, 96–117, 1985.

[38] Deutsch, D., Quantum computational networks, *Proc. R. Soc. London, Ser. A,* 425, 73–90, 1989.

[39] Diffie, W. and Hellman, M., New directions in cryptography, *IEEE Trans. on Inform. Theory,* Vol. IT-22, 644–654, Nov. 1976.

[40] DiVincenzo, D.P., Quantum computation, *Science,* 270, 255–261, 1995.

[41] ElGamal, T., A public key cryptosystem and a signature scheme based on discrete logarithms, *IEEE Trans. on Inform. Theory,* Vol. IT-31, 469–472, Jul. 1985.

[42] Ekert, A.K., Quantum cryptography based on Bell's theorem, *Phys. Rev. Lett.,* 67, 661–663, 1991.

[43] Ekert, A.K., Rarity, J.G., Tapster, P.R., and Palma, G.M., Practical quantum cryptography based on two-photon interferometry, *Phys. Rev. Lett.,* 69, 1293–1295, 1992.

[44] Frankel, Y. and Desmedt, Y., Classification of Ideal Homomorphic Threshold Schemes Over Finite Abelian Groups, *Proc. EUROCRYPT '92,* LNCS, Vol. 658, 25–34, Springer-Verlag, 1993.

[45] Gallagher, R.G., *Information Theory and Reliable Communications,* John Wiley & Sons, New York, 1968.

[46] Ghodosi, H., Pieprzyk, J., and Safavi-Naini, R., Dynamic threshold cryptosystems, *Proceedings of PRAGOCRYPT'96,* CTU Publishing House, Prague, Part 1, 370–379, 1996.

[47] Harn, L., Group-oriented (t, n) threshold digital signature scheme and digital multisignature, *IEEE Proc.-Comput. Digit. Tech.,* 141(5), 307–313, Sep. 1994.

[48] Harn, L. and Yang, S., Group-oriented undeniable signature schemes without the assistance of a mutually trusted party, In *Advances in Cryptology—Proceedings of AUSCRYPT '92,* Seberry, J. and Zheng, Y., Eds., Vol. 718 of *Lecture Notes in Computer Science,* 133–142. Springer-Verlag, 1993.

[49] Hwang, T., Cryptosystem for group oriented cryptography, *Proc. Eurocrypt'90,* LNCS Vol. 473, Springer-Verlag, Berlin, 353–360, 1990.

[50] Hwang, S.-J. and Chang, C.-C., A dynamic secret sharing scheme with cheater detection, *Proc. ACISP'96,* LNCS, Vol. 1172, 48–55, 1996.

[51] Itakura, K. and Nakamura, K., *A Public-Key Cryptosystem Suitable for Digital Multisignature,* NEC J. Res. Dev., 71 edition, Oct. 1983.

[52] Ito, M., Saito, A., and Nishizeki, T., Secret sharing scheme realising general access structure, *Proc. Globecom'87,* 99–102, 1987.

[53] Jackson, W-A. and Martin, K.M., Cumulative arrays and geometric secret sharing schemes, *Proc. Auscrypt'92,* LNCS, Vol. 718, 49–55, Springer-Verlag, Berlin, 1993.

[54] Karnin, E.D., Greene, J.W., and Hellman, M.E., On secret sharing systems, *IEEE Transactions Info. Theory,* IT-29, 1, 35–41, 1983.

[55] Laih, C.-S. and Yen, S.-M., Multi-Signature for Specified Group of Verifiers, *Journal of Information Science and Engineering,* 12(1), 143–152, Mar. 1996.

[56] Langford, S.K., Threshold DSS signatures without a trusted party, *Proc. Crypto'95,* LNCS, Vol. 963, 397–409, Springer-Verlag, Berlin, 1996.
[57] Li, C.-M., Hwang, T., and Lee, N.-Y., Threshold-multisignature schemes where suspected forgery implies traceability of adversarial shareholders, In *Advances in Cryptology—Proceedings of EUROCRYPT '94,* De Santis, A., Ed., volume 950 of *Lecture Notes in Computer Science,* 194–204, Springer-Verlag, 1995.
[58] Lin, H.-Y. and Harn, L., A generalized secret sharing scheme with cheater detection. *Proc. Asiacrypt'91,* LNCS, 149–158, Springer-Verlag, Berlin, 1993.
[59] Massey, J.L., Minimal codewords and secret sharing, *Proc. 6th Joint Swedish-Russian Workshop in Information Theory,* 276–279, 1993.
[60] Martin, K. and Jackson, W.-A., Perfect Secret Sharing Schemes on Five Participants, *Designs Codes and Cryptography,* 9, 267–286, 1996.
[61] McEliece, R.J. and Sarwate, D.V., On sharing secrets and Reed-Solomon codes, *Communications of the ACM,* 24(9), 683–584, 1981.
[62] MacWilliams and Sloane, N.J.A., *Theory of Codes,* North Holland, 1977.
[63] Muller, A., Breguet, J., and Gisin, N., Experimental demonstration of quantum cryptography using polarised photons in optical fibre over more than 1 km, *Europhys. Lett.,* 23, 383–388, 1993.
[64] National Bureau of Standards, Federal Information Processing Standard (FIPS), U.S., Department of Commerce, *Data Encryption Standard*, 46 ed., Jan. 1977.
[65] Ogata, W. and Kurosawa, K., Lower bound on the size of shares of nonperfect secret sharing schemes, *Proc. Asiacrypt'94,* LNCS Vol. 917, 33–41, Springer-Verlag, Berlin, 1995.
[66] Ogata, W., Kurosawa, K., and Tsujii, S., Nonperfect secret sharing schemes, *Proc. Auscrypt'92,* LNCS Vol. 718, 56–66, Springer-Verlag, Berlin, 1993.
[67] Ohta, K. and Okamoto, T., A digital multisignature scheme based on the Fiat-Shamir scheme, In *Advances in Cryptology—Proceedings of ASIACRYPT '91,* Rivest, R.L., Imai, H., and Matsumoto, T., Eds., Vol. 739 of *Lecture Notes in Computer Science,* 139–148, Springer-Verlag, 1993.
[68] Park, C. and Kurosawa, K., New ElGamal type threshold digital signature scheme, *IEICE Trans. Fundamentals,* E79(1), 86–93, Jan. 1996.
[69] Renvall, A. and Ding, C., The access structure of some secret sharing schemes, *Proc. ACISP'96*, LNCS, Vol. 1172, 67–86, Springer-Verlag, 1996.
[70] Rivest, R., Shamir, A., and Adleman, L., A method for obtaining digital signatures and public-key cryptosystems, *Communications of the ACM,* vol. 21, 120–126, Feb. 1978.
[71] Rivest, R., Shamir, A., and Adleman, L., On digital signatures and public-key cryptosystems, MIT Laboratory for Computer Science, Technical Report, MIT/LCS/TR-212, Jan. 1979.
[72] Schellenberg, P.J. and Stinson, D.R., Threshold schemes from combinatorial designs, *Journal of Combinatorial Mathematics and Combinatorial Computing,* 5, 143–160, 1989.
[73] Schneier, B., *Applied Cryptography: Protocols, Algorithms, and Source Code in C,* John Wiley & Sons, New York, 1994.
[74] Seberry, J. and Pieprzyk, J., *Cryptography: an Introduction to Computer Security,* Prentice-Hall, Sydney, 1989.
[75] Shamir, A., How to share a secret, *Communications of the ACM,* 22(11), 612–613, 1979.
[76] Shimizu, A. and Miyaguchi, S., Fast data encipherment algorithm FEAL, *Advances in Cryptology—Proceedings of EUROCRYPT '87,* Chaum, D. and Price, W., Eds., Vol. 304 of *Lecture Notes in Computer Science*, 267–278, Springer-Verlag, 1987.
[77] Shor, P.W., Algorithms for quantum computation: Discrete log and factoring, *Proceedings of the 35th Annual Symposium on the Foundations of Computer Science,* 124–134, Goldwasser, S., Ed., IEEE Computer Society Press, Los Alamitos, CA, 1994.
[78] Simmons, G.J., How to (really) share a secret, *Proc. Crypto'88,* LNCS Vol. 403, 390–448, Springer-Verlag, Berlin, 1989.

[79] Simmons, G.J., An introduction to shared secret and/or shared control schemes and their application, In *Contemporary Cryptology—The Science of Information Integrity,* 441–497, Simmons, G.J., Ed., IEEE Press, New York, 1992.
[80] Simmons, G.J., Jackson, W.-A., and Martin, K., The geometry of shared secret schemes, *Bull. of the ICA,* 1, 71–88, 1991.
[81] De Soete, M., Quisquater, J.-J., and Vedder, K., A signature with shared verification scheme, In Brassard, J., Ed., *Advances in Cryptology—Proceedings of CRYPTO '89,* Vol. 435 of *Lecture Notes in Computer Science,* 253–262. Springer-Verlag, 1990.
[82] NSBS. Processing Information Systems. Cryptographic Protection. Cryptographic Algorithm, GOST 28147-89, C1–C26, National Soviet Bureau of Standards, 1989.
[83] Street, A.P., Defining sets for t-designs and critical sets for Latin squares, *New Zealand Journal of Mathematics,* 21, 133–144, 1992.
[84] Stinson, D.R., An explication of secret sharing schemes, *Designs, Codes and Cryptography,* 2, 357–390, 1992.
[85] Stinson, D.R. and Vanstone, S.A., A combinatorial approach to threshold schemes, *SIAM Journal of Discrete Mathematics,* 1, 230–236, 1988.
[86] Tapster, P.R., Rarity, J.G., and Owens, P.C.M., Violation of Bell's inequality over 4 km of optical fibre, *Phys. Rev. Lett.,* 73, 1923–1926, 1994.
[87] Tompa, M. and Woll, H., How to share a secret with cheaters, *Journal of Cryptology,* 1, 133–138, 1988.
[88] Townsend, P.D., Rarity, J.G., and Tapster, P.R., Enhanced single photon fringe visibility in a 10 km-long prototype quantum cryptography channel, *Electron. Lett.,* 29, 1291–1293, 1993.
[89] Wiesner, S., Conjugate coding, *SIGACT News,* 15, 78–88, 1983. (Original manuscript written circa 1970.)

Further Information

As in the previous chapter we mention the conferences CRYPTO, EUROCRYPT, ASIACRYPT, AUSCRYPT, and conferences dealing with security such as ACISP. Information can also be found in Journals such as the *Communications of the ACM.* Quantum cryptography is also covered in the physics literature, e.g., *Europhys. Letters, Physical Review Letters.*

42

Cryptanalysis

Samuel S. Wagstaff, Jr.
Purdue University

42.1 Introduction

A *cipher* is a secret way of writing in which plaintext is enciphered into ciphertext under the control of a key. Those who know the key can easily decipher the ciphertext back into the plaintext. *Cryptanalysis* is the study of breaking ciphers, that is, finding the key or converting the ciphertext into the plaintext without knowing the key.

For a given cipher, key and plaintext, let M, C, and K denote the plaintext, the corresponding ciphertext and the key, respectively. If E_K and D_K represent the enciphering and deciphering functions when the key is K, then we may write $C = E_K(M)$ and $M = D_K(C)$. For all M and K we must have $D_K(E_K(M)) = M$.

There are four kinds of cryptanalytic attacks. All four kinds assume that the forms of the enciphering and deciphering functions are known.

1. Ciphertext only: Given C, find K or at least M for which $C = E_K(M)$.
2. Known plaintext: Given M and the corresponding C, find K for which $C = E_K(M)$.
3. Chosen plaintext: The cryptanalyst may choose M. He is told C and must find K for which $C = E_K(M)$.
4. Chosen ciphertext: The cryptanalyst may choose C. He is told M and must find K for which $C = E_K(M)$.

The ciphertext only attack is hardest. To carry it out, one may exploit knowledge of the language of the plaintext, its redundancy, or its common words.

An obvious known plaintext attack is to try all possible keys K and stop when one is found with $C = E_K(M)$. This is feasible only when the number of possible keys is small.

The chosen plaintext and ciphertext attacks may be possible when the enemy can be tricked into enciphering or deciphering some text or after the capture of a cryptographic chip with an embedded key.

The cryptanalyst has *a priori* information about the plaintext. For example, he knows that a string like "the" is more probable than the string "wqx." One goal of cryptanalysis is to modify the *a priori* probability distribution of possible plaintexts to make the correct plaintext more probable than the incorrect ones, although not necessarily certain. Shannon's [20] information theory is used to formalize this process. It estimates, for example, the *unicity distance* of a cipher, which is the shortest length of ciphertext needed to make the correct plaintext more probable than the incorrect ones.

42.2 Types of Ciphers

There is no general method of attack on all ciphers. There are many *ad hoc* methods that work on just one cipher or, at best, one type of cipher. We will list some of the kinds of ciphers and describe what methods one might use to break them. In the later sections we will describe some of the more sophisticated methods of cryptanalysis.

Ciphers may be classified as **transposition ciphers, substitution ciphers** or product ciphers.

Transposition ciphers rearrange the characters of the plaintext to form the ciphertext. For example, enciphering may consist of writing the plaintext into a matrix row-wise and reading out the ciphertext column-wise. More generally, one may split the plaintext into blocks of fixed length L and encipher by applying a given **permutation,** the key, to each block. It is easy to recognize transposition ciphers, because the frequency distribution of ciphertext letters is the same as the usual distribution of letters in the language of the plaintext. One may guess the period length L from the message length and the spacing between repeated strings. Once L is guessed, the permutation may be constructed by trial and error using the frequency distribution of pairs or triples of letters.

In a *simple substitution cipher,* the key is a fixed permutation of the alphabet. The ciphertext is formed from the plaintext by replacing each letter by its image under the permutation. These ciphers are broken by trial and error, comparing the frequency distribution of the ciphertext letters with that of the plaintext letters. For example, "e" is the most common letter in English. Therefore, the most frequent ciphertext letter probably is the image of "e." The frequency distribution of pairs or triples of letters helps, too. The pair "th" is common in English. Hence, a pair of letters with high frequency in the ciphertext might be the image of "th."

A *homophonic substitution cipher* uses a ciphertext alphabet larger than the plaintext alphabet, to confound the frequency analysis just described. The key is an assignment of each plaintext letter to a subset of the ciphertext alphabet, called the **homophones** of the plaintext letter. The homophones of different letters are disjoint sets and the size of the homophone set is often proportional to the frequency of the corresponding letter in ordinary plaintext. Thus "e" would have the largest homophone set. Plaintext is enciphered by replacing each letter by a *random* element of its homophone set. As a result, a frequency analysis of single ciphertext letters will find that they have a uniform distribution, which is useless for cryptanalysis. Homophonic substitution ciphers may be broken by analysis of the frequency distribution of pairs and triples of letters.

Polyalphabetic substitution ciphers use multiple permutations (called *alphabets*) of the plaintext alphabet onto the ciphertext alphabet. The key is a description of these permutations. If the permutations repeat in a certain sequence, as in the Beaufort and Vigenère ciphers, one may break the cipher by first determining the number d of different alphabets and then solving d interleaved simple substitution ciphers. Either the Kasiski method or the Index of Coincidence may be used to find d.

The Kasiski method looks for repeated ciphertext strings in the hope that they may be encipherments of the same plaintext word which occurs each time at the same place in the cycle of alphabets. For example, if the ciphertext "wqx" occurs twice, starting in positions i and j, then probably d divides $j - i$. When several repeated pairs are found, d may be the greatest common divisor of a subset of the differences in their starting points.

The Index of Coincidence (IC) (see Friedman [11]) measures frequency variations of letters to estimate the size of d. Let $\{a_1, \ldots, a_n\}$ be the alphabet (plaintext or ciphertext). For $1 \leq i \leq n$, let F_i be the frequency of occurrence of a_i in a ciphertext of length N. The Index of Coincidence is given by the formula

$$IC = \binom{N}{2}^{-1} \sum_{i=1}^{n} \binom{F_i}{2}.$$

Then IC is the probability that two letters chosen at random in the ciphertext are the same letter. One can estimate IC theoretically in terms of d, N and the usual frequency distribution of letters in plaintext. For English, $n = 26$ and the expected value of IC is

$$\frac{1}{d}\left(\frac{N-d}{N-1}\right)(0.066) + \left(\frac{d-1}{d}\right)\left(\frac{N}{N-1}\right)\frac{1}{26}.$$

One estimates d by comparing IC, computed from the ciphertext, with its expected value for various d.

These two methods of finding d often complement each other in that the Index of Coincidence tells the approximate size of d, while the Kasiski method gives a number that d divides. For example, the Index of Coincidence may suggest that d is 5, 6, or 7 and the Kasiski method may predict that d probably divides 12. In this case $d = 6$.

Rotor machines, like the German Enigma, and Hagelin machines are hardware devices that implement polyalphabetic substitution ciphers with very long periods d. They can be cryptanalyzed using group theory. See Barker [1], Kahn [13] and Konheim [14]. The UNIX `crypt(1)` command uses a similar cipher. See Reeds and Weinberger [19] for its cryptanalysis.

A *one-time pad* is a polyalphabetic substitution cipher whose alphabets do not repeat. They are selected by a random sequence which is the key. The key is as long as the plaintext. If the key sequence is truly random and if it is used only once, then this cipher is unbreakable. However, if the key sequence is itself ordinary text, as in a running-key cipher, the cipher can be broken by frequency analysis, beginning with the assumption that usually in each position both the plaintext letter and key letter are letters which occur with high frequency. Frequency analysis works also when a key is reused to encipher a second message. In this case one assumes that in each position the plaintext letters from both messages are letters which occur with high frequency.

A *polygram substitution cipher* enciphers a block of several letters together to prevent frequency analysis. For example, the Playfair cipher enciphers pairs of plaintext letters into pairs of ciphertext letters. It may be broken by analyzing the frequency of pairs of ciphertext letters. See Hitt [12] for details. The Hill cipher treats a block of plaintext or ciphertext letters as a vector and enciphers by matrix multiplication. If the vector dimension is 2 or 3, then the cipher may be broken by frequency analysis of pairs or triples of letters. A known plaintext attack on a Hill cipher is an easy exercise in linear algebra. (Compare with linear feedback shift registers in the next section.)

The Pohlig–Hellman cipher and most public key ciphers are polygram substitution ciphers with a block size several hundred characters long. The domain and range of the substitution mapping is so large that the key must be a compact description of this function, such as exponentiation modulo a large number. See the sections on knapsacks, RSA, integer factoring, and discrete logarithms below for cryptanalysis of such ciphers.

A *product cipher* is a cipher formed by composing several transposition and substitution ciphers. A classic example is the data encryption standard, which alternates transposition and substitution ciphers in 16 rounds. See Diffie and Hellman [9] for general cryptanalysis of DES. See the section on differential cryptanalysis below for a powerful attack on DES and similar product ciphers.

Ciphers are also classified as block or stream ciphers. All ciphers split long messages into blocks and encipher each block separately. Block sizes range from one bit to thousands of bits per block.

- A *block cipher* enciphers each block with the same key.
- A *stream cipher* has a sequence or stream of keys and enciphers each block with the next key. The **key stream** may be periodic, as in the Vigenère cipher or a linear feedback shift register, or not periodic, as in a one-time pad. Ciphertext is often formed in stream ciphers by exclusive-oring the plaintext with the key.

42.3 Linear Feedback Shift Registers

A linear feedback shift register is a device that generates a key stream for a stream cipher. It consists of an n-bit shift register and an XOR gate. Let the shift register hold the vector $R = (r_0, r_1, \ldots, r_{n-1})$. The inputs to the XOR gate are several bits selected (tapped) from fixed bit positions in the register. Let the 1 bits in the vector $T = (t_0, t_1, \ldots, t_{n-1})$ specify the tapped bit positions. Then the output of the XOR gate is the scalar product $TR = \Sigma_{i=0}^{n-1} t_i r_i \bmod 2$, and this bit is shifted into the shift register. Let $R' = (r'_0, r'_1, \ldots, r'_{n-1})$ be the contents of the register after the shift. Then $r'_i = r_{i-1}$ for $1 \le i < n$ and $r'_0 = TR$. In other words, $R' \equiv HR \bmod 2$, where H is the $n \times n$ matrix with T as its first row, 1s just below the main diagonal and 0s elsewhere.

$$H = \begin{pmatrix} t_0 & t_1 & \ldots & t_{n-2} & t_{n-1} \\ 1 & 0 & \ldots & 0 & 0 \\ 0 & 1 & \ldots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \ldots & 0 & 0 \\ 0 & 0 & \ldots & 1 & 0 \end{pmatrix}$$

The bit r_{n-1} is shifted out of the register and into the key stream. One can choose T so that the period of the bit stream is $2^n - 1$, which is the maximum possible period. If n is a few thousand, this length may make the cipher appear secure, but the linearity makes it easy to break.

Suppose $2n$ consecutive key bits $k_0, \ldots, k_{2n-1}$ are known. Let X and Y be the $n \times n$ matrices

$$X = \begin{pmatrix} k_{n-1} & k_n & \ldots & k_{2n-2} \\ k_{n-2} & k_{n-1} & \ldots & k_{2n-3} \\ \vdots & \vdots & \ddots & \vdots \\ k_0 & k_1 & \ldots & k_{n-1} \end{pmatrix}$$

$$Y = \begin{pmatrix} k_n & k_{n+1} & \ldots & k_{2n-1} \\ k_{n-1} & k_n & \ldots & k_{2n-2} \\ \vdots & \vdots & \ddots & \vdots \\ k_1 & k_2 & \ldots & k_n \end{pmatrix}$$

From $R' \equiv HR \bmod 2$ it follows that $Y \equiv HX \bmod 2$, so H may be computed from $H \equiv YX^{-1} \bmod 2$. The inverse matrix $X^{-1} \bmod 2$ is easy to compute by Gaussian elimination or by the Berlekamp–Massey algorithm for n up to at least 10^4. The tap vector T is the first row of H and the initial contents R of the shift register are $(k_{n-1}, \ldots, k_0)$.

See Ding, Xiao and Shan [10] for more information about linear feedback shift registers and variations of them.

42.4 Meet in the Middle Attacks

One might think that a way to make block ciphers more secure is to use them twice with different keys. If the key length is n bits, a brute force known plaintext attack on the basic block cipher would take up to 2^n encryptions. It might appear that the double encryption $C = E_{K_2}(E_{K_1}(M))$, where K_1 and K_2 are independent n-bit keys, would take up to 2^{2n} encryptions to find the two keys. The *meet-in-the-middle attack* is a known plaintext attack which trades time for memory and breaks the double cipher using only about 2^{n+1} encryptions, and space to store 2^n blocks and keys. Let M_1 and M_2 be two known plaintexts, and let C_1 and C_2 be the corresponding ciphertexts. (A third pair may be needed.) For each possible key K store the pair $(K, E_K(M_1))$ in a file. Sort the 2^n pairs by second component. For each possible key K compute $D_K(C_1)$ and look for this value as the second component of a pair in the file. If it is found, the current key might be K_2 and the key in the pair found in the file might be K_1. Check whether $C_2 = E_{K_2}(E_{K_1}(M_2))$ to determine whether K_1 and K_2 really are the keys.

In the case of DES, $n = 56$ and the meet-in-the-middle attack requires enough memory to store 2^{56} plaintext-ciphertext pairs, more than is available now. But some day there may be enough memory to make the attack feasible.

42.5 Differential and Linear Cryptanalysis

Differential cryptanalysis is an attack on DES and some related ciphers which is faster than exhaustive search. It requires about 2^{47} steps rather than the 2^{55} steps needed on average to test all keys.

Note: Although there are 2^{56} keys for DES, one can test both a key K and its complement $\overline{K}$ with a single DES encipherment using the equivalence

$$C = \mathrm{DES}_K(M) \iff \overline{C} = \mathrm{DES}_{\overline{K}}(\overline{M}) .$$

The adjective "differential" here refers to a difference modulo 2 or XOR. The basic idea of differential cryptanalysis is to form many input plaintext pairs M, M^* whose XOR $M \oplus M^*$ is constant and study the distribution of the output XORs $\mathrm{DES}_K(M) \oplus \mathrm{DES}_K(M^*)$. The XOR of two inputs to DES, or a part of DES, is called the *input XOR* and the XOR of the outputs is called the *output XOR*.

DES (see Chapter 38) consists of permutations, XORs and F functions. The F functions are built from expansion (48 bits from 32 bits), S-boxes and permutations. It is easy to see that permutations P, expansions E, and XOR's satisfy these equations:

$$\begin{aligned} P(X) \oplus P(X^*) &= P(X \oplus X^*), \\ E(X) \oplus E(X^*) &= E(X \oplus X^*) \end{aligned}$$

and

$$(X \oplus K) \oplus (X^* \oplus K) = X \oplus X^* .$$

These operations are *linear* (as functions on vector spaces over the field with 2 elements). In contrast, S-boxes are *nonlinear*. They do not preserve XORs:

$$S(X) \oplus S(X^*) \neq S(X \oplus X^*) .$$

Indeed, if one tabulates the distribution of the four-bit output XOR of an S-box as a function of the six-bit input XOR, one obtains an irregular table. We show a portion of this table for the S-box S_1 (Table 42.1).

The row and column labels are hexadecimal numbers. All counts are even numbers since a pair (X, X^*) is counted in an entry if and only if (X^*, X) is counted. Note also that if the input XOR is 0, then the output XOR is 0, too. Hence the first row (0H) has the entry 64 in column 0H and 0 counts for the other

TABLE 42.1 XOR Distribution for S-box S_1

Input	Output XOR										
XOR	0H	1H	2H	3H	4H	5H	6H	7H	...	EH	FH
0H	64	0	0	0	0	0	0	0	...	0	0
1H	0	0	0	6	0	2	4	4	...	2	4
2H	0	0	0	8	0	4	4	4	...	4	2
3H	14	4	2	2	10	6	4	2	...	2	0
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
34H	0	8	16	6	2	0	0	12	...	0	6
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
3FH	4	8	4	2	4	0	2	4	...	2	2

15 columns. The other rows have no entry greater than 16 and 20% to 30% 0 counts. The average entry size is $64/16 = 4$. The 64 rows for each S-box are all different.

Here is an overview of the differential cryptanalysis of DES. Encipher many plaintext pairs having a fixed XOR and save the ciphertext pairs. The input to the F function in the last round is $R_{15} = L_{16}$, which can be computed from the ciphertext by undoing the final permutation. Thus we can find the input XOR and output XOR for each S-box in the last round. Consider one S-box. If we knew the output E of expansion and the input X to the S-box, we could compute 6 bits (K) of the key used here by $K = E \oplus X$, since the F function computes $X = E \oplus K$. For each of the 64 possible 6-bit blocks K of the key, count the number of pairs that result with the known output XOR using this key value in the last round. Compare this distribution with the rows of the XOR table for the S box. When enough pairs have been enciphered, the distribution will be similar to a unique row, and this will give six of the 48 key bits used in the last round. After this has been done for all eight S boxes, one knows 48 of the 56 key bits. The remaining 8 bits can be found quickly by trying the 256 possibilities.

The input XORs to the last round may be specified by using as plaintext XORs certain 64-bit patterns which, with a small positive probability, remain invariant under the first 15 rounds. Plaintext pairs which produce the desired input to the last round are called *right pairs.* Other plaintext pairs are called *wrong pairs.* In the statistical analysis, right pairs predict correct key bits while wrong pairs predict random key bits. When enough plaintext pairs have been analyzed, the correct key bits overcome the random bits by becoming the most frequently suggested values.

The description above gives the *chosen* plaintext attack on DES. A *known* plaintext attack may be performed as follows. Suppose we need m plaintext pairs to perform a chosen plaintext attack. Let $2^{32}\sqrt{2m}$ random known plaintexts be given together with their corresponding ciphertexts. Consider all $(2^{32}\sqrt{2m})^2/2 = 2^{64}m$ possible (unordered) pairs of known plaintext. Since the block size is 64 bits, there are only 2^{64} possible plaintext XOR values. Hence there are about $2^{64}m/2^{64} = m$ pairs giving each plaintext XOR value. Therefore, it is likely that there are m pairs with each of the plaintext XOR's needed for a chosen plaintext attack.

Although we have described the attack for ECB mode, it works also for the other modes (CBC, CFB, and OFB) because it is easy to compute the real input and output of DES from the plaintext and ciphertext in these modes.

The linearity of parts of the DES algorithm mentioned earlier is the basis for Matsui's [16] linear cryptanalysis. In this known plaintext attack on DES, one XOR's together certain plaintext and ciphertext bits to form a linear approximation for the XOR of certain bits of the key. This approximation will be correct with some probability p. Irregularity in the definition of the S-boxes allows one to choose bit positions from the plaintext, ciphertext and key for which the XOR approximation is valid with probability $p \neq 1/2$. This bias can be amplified by computing the XOR of the key bits for many different known plaintext-ciphertext pairs. When enough XOR's of key bits have been guessed, one can compute the key

bits by solving a system of linear equations over the field with 2 elements. Linear cryptanalysis of DES requires about 2^{43} known plaintext-ciphertext pairs to determine the 56-bit key.

Differential and linear cryptanalysis apply also to many substitution/ permutation ciphers similar to DES. These include FEAL, LOKI, Lucifer, and REDOC-II. The same methods can break some hash functions like Snefru and N-hash, by producing two messages with the same hash value.

See Biham and Shamir [3] and [4] for more information about differential cryptanalysis. See Matsui [16] for the details of linear cryptanalysis.

42.6 Knapsack Ciphers

We describe a variation of Shamir's attack on the Merkle–Hellman knapsack. A similar attack will work for almost all known knapsacks, including iterated ones. The only public key knapsack not yet broken is that of Chor and Rivest [6].

Let the public knapsack weights be the positive integers a_i for $1 \le i \le n$. An n-bit plaintext block $x_1, \ldots, x_n$ is enciphered as $E = \Sigma_{i=1}^{n} a_i x_i$.

Let the secret superincreasing knapsack weights be s_i for $1 \le i \le n$, where $\Sigma_{i=1}^{j} s_i < s_{j+1}$ for $1 \le j < n$. Let the secret modulus M and multiplier W satisfy $1 < W < M$, $\gcd(W, M) = 1$ and $M > \Sigma_{i=1}^{n} s_i$. Then $a_i \equiv W^{-1} s_i \bmod M$ or $W a_i \equiv s_i \bmod M$ for $1 \le i \le n$.

To attack this cryptosystem, rewrite the last congruence as $W a_i - M k_i = s_i$ for $1 \le i \le n$, where the k_i are unknown nonnegative integers. Divide by $M a_i$ to get

$$\frac{W}{M} - \frac{k_i}{a_i} = \frac{s_i}{M a_i} .$$

This equation shows that, at least for small i, the fraction k_i/a_i is a close approximation to the fraction W/M. The inequalities $\Sigma_{i=1}^{j} s_i < s_{j+1}$ and $\Sigma_{i=1}^{n} s_i < M$ imply $s_i \le 2^{-n+i} M$. For almost all W we have $a_i \ge M/n^2$. Hence,

$$\frac{W}{M} - \frac{k_i}{a_i} = \frac{s_i}{M a_i} \le \frac{2^{-n+i}}{a_i} \le \frac{2^{-n+i} n^2}{M} .$$

We expect that all k_i and all a_i are about the same size as M, so that

$$\frac{k_i}{k_1} - \frac{a_i}{a_1} = \mathrm{O}(2^{-n+i} n^2 / M) .$$

This says that the vector $(a_2/a_1, \ldots, a_n/a_1)$, constructed from the public weights, is a very close approximation to the vector $(k_2/k_1, \ldots, k_n/k_1)$, which involves numbers k_i from which one can compute the secret weights s_i. Given a vector like $(a_2/a_1, \ldots, a_n/a_1)$, one can find close approximations to it, like $(k_2/k_1, \ldots, k_n/k_1)$, by integer programming or the Lenstra–Lenstra–Lovasz Theorem. After k_1 is found, M and W are easily determined from the fact that k_1/a_1 is a close approximation to W/M. Finally, Shamir showed that if W^*/M^* is any reasonably close approximation to W/M, then W^* and M^* can be used to decrypt every ciphertext.

See Brickell and Odlyzko [2] for more information about breaking knapsacks.

42.7 Cryptanalysis of RSA

Suppose an RSA cipher (see Rivest, Shamir and Adleman [18]) is used with public modulus $n = pq$, where $p < q$ are large secret primes, public enciphering exponent e and secret deciphering exponent d.

Most attacks on RSA try to factor n. One could read the message without factoring n if the sender of the enciphered message packed only one letter per plaintext block. Then the number of different ciphertext

blocks would be small (the alphabet size) and the ciphertext could be cryptanalyzed as a simple substitution cipher.

Here are four cases in which $n = pq$ might be easy to factor.

(1) If p and q are too close, that is, if $q - p$ is not much larger than $n^{1/4}$, then n can be factored by Fermat's Difference of Squares method. It finds x and y so that $n = x^2 - y^2 = (x - y)(x + y)$, as follows. Let $m = \lceil \sqrt{n} \rceil$. Write $x = m$. If $z = x^2 - n$ is a square y^2, then $p = x - y$, $q = x + y$ and we are done. Otherwise, add $2m + 1$ to z and add 2 to m. Then test whether $z = (x + 1)^2 - n$ is a square, and so on. Most z can be eliminated quickly because squares must lie in certain congruence classes. For example, squares must be $\equiv 0, 1, 4$, or $9 \bmod 16$, which eliminates three out of four possible values for z. If $p = k\sqrt{n}$, where $0 < k < 1$, then this algorithm tests about $((1 - k)^2/(2k))\sqrt{n}$ z's to discover p. The Difference of Squares method normally is not used at all to factor large integers. Its only modern use is to insure that no one forms an RSA modulus by multiplying two 100-digit primes having the same high order 45 digits, say. Riesel [17] has a nice description of Fermat's Difference of Squares Method.

(2) If either $p - 1$ or $q - 1$ has a small ($< 10^9$, say) largest prime factor, then n can be factored easily by Pollard's $p - 1$ factoring algorithm. The idea of this method is that if $p - 1$ divides Q and p does not divide b, then p divides $b^Q - 1$. This assertion follows from Fermat's little theorem. Typically, Q is the product of all prime powers below some limit. The method is feasible when this limit is as large as 10^9. If $Q = \prod_{i=1}^{k} q_i$, when q_i are prime powers, then one computes

```
x = b
for i = 1 to k
     x = x^q_i mod n
     if g = gcd(x-1, n) > 1, stop with factor g of n
end
```

This method and the following one have analogues in which $p - 1$ and $q - 1$ are replaced by $p + 1$ and $q + 1$. See Riesel [17] for more details of Pollard's $p - 1$ method. See Williams [21] for details of the $p + 1$ method.

(3) The number $g = \gcd(p - 1, q - 1)$ divides $n - 1$ and may be found by factoring $n - 1$, which may be much easier than factoring n. When g is small, as usually happens, it is not useful for finding p or q. But when g is large, it may help to find p or q. In that case, for each large factor $g < \sqrt{n}$ of $n - 1$ one can seek a factorization $n = (ag + 1)(bg + 1)$.

(4) If n is small, $n < 10^{150}$, say, then one can factor n by a general factoring algorithm like those described in the next section.

42.8 Integer Factoring

We outline the quadratic sieve (QS) and the number field sieve (NFS), which are currently the two fastest known integer factoring algorithms that work for any composite number, even an RSA modulus.

Both of these methods factor the odd composite positive integer n by finding integers x, y so that $x^2 \equiv y^2 \bmod n$ but $x \not\equiv \pm y \bmod n$. The first congruence implies that n divides $(x - y)(x + y)$, while the second congruence implies that n does not divide $x - y$ or $x + y$. It follows that at least one prime factor of n divides $x - y$ and at least one prime factor of n does not divide $x - y$. Therefore, $\gcd(n, x - y)$ is a proper factor of n. (This analysis fails if n is a power of a prime, but an RSA modulus would never be a prime power because it would be easy to recognize.)

However, these two factoring methods ignore the condition $x \not\equiv \pm y \bmod n$ and seek many random solutions to $x^2 \equiv y^2 \bmod n$. If n is odd and not a prime power, at least half of all solutions with $y \not\equiv 0 \bmod n$ satisfy $x \not\equiv \pm y \bmod n$ and factor n. Actually, if n has k different prime factors, the probability of successfully factoring n is $1 - 2^{1-k}$, for each random congruence $x^2 \equiv y^2 \bmod n$ with $y \not\equiv 0 \bmod n$.

In QS, many congruences (called relations) of the form $z^2 \equiv q \bmod n$ are produced with q factored completely. One uses linear algebra over the field GF (2) with two elements to pair these primes and find a subset of the relations in which the product of the qs is a square, y^2, say. Let x be the product of the zs in these relations. Then $x^2 \equiv y^2 \bmod n$, as desired.

The linear algebra is done as follows: Let $p_1, \ldots, p_r$ be all of the primes which divide any q. Write the relations as

$$z_i^2 \equiv q_i = \prod_{j=1}^{r} p_j^{e_{ij}} \bmod n, \text{ for } 1 \le i \le s \ ,$$

where $s > r$. Note that $\prod_{j=1}^{r} p_j^{f_j}$ is the square of an integer if and only if every $f_i \equiv 0 \bmod 2$. Suppose $(t_1, \ldots, t_s)$ is not the zero vector in the s-dimensional vector space $GF(2)^s$, but does lie in the null space of the matrix $[e_{ij}]$, that is, $\Sigma_{i=1}^{s} e_{ij} t_i = 0$ in $GF(2)$. Then

$$\prod_{\substack{i=1 \\ t_i=1}}^{s} q_i = \prod_{j=1}^{r} \prod_{\substack{i=1 \\ t_i=1}}^{s} p_j^{e_{ij}} \ .$$

The exponent on p_j in this double product is $\Sigma_{i=1}^{s} e_{ij} t_i$, which is an even integer since the t vector is in the null space of the matrix. Therefore,

$$\prod_{\substack{i=1 \\ t_i=1}}^{s} q_i$$

is a square because every one of its prime factors appears raised to an even power. Call this square y^2 and let

$$x = \prod_{\substack{i=1 \\ t_i=1}}^{s} z_i \ .$$

Then $x^2 \equiv y^2 \bmod n$.

In QS, the relations $z^2 \equiv q \bmod n$ with q factored completely are produced as follows: Small primes $p_1, \ldots, p_r$ for which n is a quadratic residue (square) are chosen as the *factor base.* Choose many pairs a, b of integers with $a = c^2$ for some integer c, $b^2 \equiv n \bmod a$ and $|b| \le a/2$. Then the quadratic polynomial

$$Q(t) = \frac{1}{a}[(at+b)^2 - n] = at^2 + 2bt + \frac{b^2 - n}{a}$$

will take integer values at every integer t. If a value of t is found for which the right-hand side is factored completely, a relation $z^2 \equiv q \bmod n$ is produced, with $z \equiv (at+b)c^{-1} \bmod n$ and $q = Q(t) = \prod_{j=1}^{r} p_j^{f_j}$, as desired. No trial division by the primes in the **factor base** is necessary. A **sieve** factors millions of $Q(t)$'s at once. Let t_1 and t_2 be the two solutions of

$$(at+b)^2 \equiv n \bmod p_i \text{ in } 0 \le t_1, t_2 < p_i \ .$$

(A prime p is put in the factor base for QS only if this congruence has solutions, that is, only if n is a quadratic residue modulo p.) Then all solutions of $Q(t) \equiv 0 \bmod p_i$ are $t_1 + kp_i$ and $t_2 + kp_i$ for $k \in \mathbf{Z}$. In most implementations, $Q(t)$ is represented by one byte `Q[t]` and $\log p_i$ is added to this byte to avoid division of $Q(t)$ by p_i, a slow operation. The two inner loops are

```
t = t_1
while t < upper_limit
     Q[t] = Q[t] + log p_i
     t = t + p
end
```

and a similar loop for the other root of the quadratic congruence. After the sieve completes, one harvests the relations $z^2 \equiv q \bmod n$ from those t for which `Q[t]` exceeds a threshold. Only at this point is $Q(t)$ formed and factored, the latter operation being done by trial division with the primes in the factor base.

The time QS takes to factor n is about

$$\exp((1+\epsilon(n))(\log n)^{1/2}(\log\log n)^{1/2}) ,$$

where $\epsilon(n) \to 0$ as $n \to \infty$.

We begin our outline of the NFS by describing the special number field sieve (SNFS), which factors numbers of the form $n = r^e - s$, where r and $|s|$ are small positive integers. It uses some algebraic number theory. Choose a small positive integer d, the degree of an extension field. Let k be the least positive integer for which $kd \geq e$. Let $t = sr^{kd-e}$. Let f be the polynomial $X^d - t$. Let $m = r^k$. Then $f(m) = r^{kd} - sr^{kd-e} = r^{kd-e}n$ is a multiple of n. The optimal degree for f is $((3+\epsilon(e))\log n/(2\log\log n))^{1/3}$ as $e \to \infty$ uniformly for r, s in a finite set, where $\epsilon(e) \to 0$ as $e \to \infty$.

Let α be a zero of f. Let $K = \mathbf{Q}(\alpha)$. Assume f is irreducible. The degree of K over $\mathbf{Q}$ is d. Let Q_n denote the ring of rational numbers with denominator **coprime** to n. The subring $Q_n[\alpha]$ of K consists of expressions $\Sigma_{i=0}^{d-1}(s_i/t_i)\alpha^i$ with $s_i, t_i \in \mathbf{Z}$ and $\gcd(n, t_i) = 1$. Define a ring homomorphism $\phi : Q_n[\alpha] \to \mathbf{Z}/n\mathbf{Z}$ by the formula $\phi(\alpha) = (m \bmod n)$. Thus $\phi(a + b\alpha) \equiv a + bm \bmod n$.

For $0 < a \leq A$ and $-B \leq b \leq B$, SNFS uses a sieve (as in QS) to factor $a + bm$ and the norm of $a + b\alpha$ in $\mathbf{Z}$. The norm of $a + b\alpha$ is $(-b)^d f(-a/b)$. This polynomial of degree d has d roots modulo p which must be sieved, just like the two roots of $Q(t)$ in QS. A pair (a, b) is saved in a file if $\gcd(a, b) = 1$, and $a + bm$ and the norm of $a + b\alpha$ both have only small prime factors (ones in the factor base, say).

We use linear algebra to pair the prime factors just as in QS, except that now we must form squares on *both* sides of the congruences. The result is a non-empty set S of pairs (a, b) of coprime integers such that

$$\prod_{(a,b)\in S} (a + bm) \text{ is a square in } \mathbf{Z} ,$$

and

$$\prod_{(a,b)\in S} (a + b\alpha) \text{ is a square in } Q_n[\alpha] .$$

Let the integer x be a square root of the first product. Let $\beta \in Q_n[\alpha]$ be a square root of the second product. We have $\phi(\beta^2) \equiv x^2 \bmod n$, since $\phi(a + b\alpha) \equiv a + bm \bmod n$. Let y be the integer for which $\phi(\beta) \equiv y \bmod n$. Then $x^2 \equiv y^2 \bmod n$, which will factor n with probability at least $1/2$.

In the general NFS, we must factor an arbitrary n for which no obvious polynomial is given. The key properties required of the polynomial are that it is irreducible, that it has moderately small coefficients so that the norm of α is small, and that we know a non-trivial root m modulo n of it. Research in good polynomial selection is still in progress. One simple choice is to let m be some integer near $n^{1/d}$ and write n in radix m as $n = c_d m^d + \cdots + c_0$, where $0 \leq c_i < m$. Then use the polynomial $f(X) = c_d X^d + \cdots + c_0$. With this choice, the time needed by NFS to factor n is

$$\exp(((64/9)^{1/3} + \epsilon(n))(\log n)^{1/3}(\log\log n)^{2/3}) ,$$

where $\epsilon(n) \to 0$ as $n \to \infty$. QS is faster than NFS for factoring numbers up to a certain size, and NFS is faster for larger numbers. This crossover size is between 100 and 150 decimal digits, depending on the implementation and the size of the coefficients of the NFS polynomial.

See Riesel [17] for a description of the quadratic sieve and number field sieve factoring algorithms. See also Lenstra and Lenstra [15] for more information about NFS.

42.9 Discrete Logarithms

The Diffie–Hellman key exchange, the ElGamal public key cryptosystem, the Pohlig–Hellman private key cryptosystem and the **digital signature algorithm** could all be broken if we could compute discrete logarithms quickly, that is, if we could solve the equation $a^x = b$ in a large finite field. For convenience of computation, usually the finite field is either the integers modulo a prime p or the field with 2^n elements.

Consider first the exponential congruence $a^x \equiv b \bmod p$. By analogy to ordinary logarithms, we may write $x = \log_a b$ when p is understood from the context. These *discrete logarithms* enjoy many properties of ordinary logarithms, such as $\log_a bc = \log_a b + \log_a c$, except that the arithmetic with logarithms must be done modulo $p-1$ because $a^{p-1} \equiv 1 \bmod p$. Neglecting powers of $\log p$, the congruence may be solved in $O(p)$ time and $O(1)$ space by raising a to successive powers modulo p and comparing each with b. It may also be solved in $O(1)$ time and $O(p)$ space by looking up x in a precomputed table of pairs $(x, a^x \bmod p)$ sorted by the second coordinate. Shanks' "giant step–baby step" algorithm solves the congruence in $O(\sqrt{p})$ time and $O(\sqrt{p})$ space as follows. Let $m = \lceil \sqrt{p-1} \rceil$. Compute and sort the m ordered pairs $(j, a^{mj} \bmod p)$, for j from 0 to $m-1$, by the second coordinate. Compute and sort the m ordered pairs $(i, ba^{-i} \bmod p)$, for i from 0 to $m-1$, by the second coordinate. Find a pair (j, y) in the first list and a pair (i, y) in the second list. This search will succeed because every integer between 0 and $p-1$ can be written as a two-digit number ji in radix m. Finally, $x = mj + i \bmod p-1$.

There are faster ways to solve $a^x \equiv b \bmod p$ using methods similar to the two integer factoring algorithms QS and NFS. Here is the analogue for QS. Choose a factor base of primes $p_1, \ldots, p_k$. Perform the following precomputation which depends on a and p but not on b. For many random values of x, try to factor $a^x \bmod p$ using the primes in the factor base. Save at least $k+20$ of the factored residues:

$$a^{x_j} \equiv \prod_{i=1}^{k} p_i^{e_{ij}} \bmod p \text{ for } 1 \le j \le k+20 ,$$

or equivalently

$$x_j \equiv \sum_{i=1}^{k} e_{ij} \log_a p_i \bmod p-1 \text{ for } 1 \le j \le k+20 .$$

When b is given, perform the following main computation to find $\log_a b$. Try many random values for s until one is found for which $ba^s \bmod p$ can be factored using only the primes in the factor base. Write it as

$$ba^s \equiv \prod_{i=1}^{k} p_i^{c_i} \bmod p$$

or

$$(\log_a b) + s \equiv \sum_{i=1}^{k} c_i \log_a p_i \bmod p-1 .$$

Use linear algebra as in QS to solve the linear system of congruences modulo $p-1$ for $\log_a b$. One can prove that the precomputation takes time

$$\exp((1+\epsilon(p))\sqrt{\log p \log\log p}) ,$$

where $\epsilon(p) \to 0$ as $p \to \infty$, while the main computation takes time

$$\exp((0.5+\epsilon(p))\sqrt{\log p \log\log p}) ,$$

where $\epsilon(p) \to 0$ as $p \to \infty$.

There is a similar algorithm for solving congruences of the form $a^x \equiv b \bmod p$ which is analogous to NFS and runs faster than the above for large p.

There is a method of Coppersmith [5] for solving equations of the form $a^x = b$ in the field with 2^n elements which is practical for n up to about 1000. Empirically, it is about as difficult to solve $a^x = b$ in the field with p^n elements as it is to factor a general number about as large as p^n. As these are the two problems which must be solved in order to break the RSA and the ElGamal cryptosystems, it is not clear which of these systems is more secure.

42.10 Research Issues and Summary

This article presents an overview of some of the many techniques of breaking ciphers. This subject has a long and rich history. We gave some historical perspective by mentioning some of the older ciphers whose cryptanalysis is well understood. Most of this article deals with the cryptanalysis of ciphers still being used today, such as DES, RSA, Pohlig–Hellman, etc. We still do not know how to break these ciphers quickly, and research into methods for doing this continues today.

New cryptosystems are invented frequently. Most are broken quickly. For example, research continues on multistage knapsacks, some of which have not been broken yet, but probably will be soon. Some variations of differential cryptanalysis being studied now include higher-order differential cryptanalysis and a combination of linear and differential cryptanalysis. Some recent advances in integer factoring algorithms include the use of several primes outside the factor base in QS, the self-initializing QS, faster computation of the square root (β) in the NFS, better methods of polynomial selection for the general NFS, and the lattice sieve variation of the NFS. Most of these advances apply equally to computing discrete logs. A promising source of new cryptosystems is the theory of elliptic curves, in which adding a point to itself many times corresponds to exponentiation in RSA, Pohlig–Hellman, and similar cryptosystems. Cryptanalysis of elliptic curve ciphers is harder than that of exponentiation ciphers because the group structure of an elliptic curve is more complicated than that of the multiplicative group of integers modulo n.

We have omitted discussion of many important ciphers in this short article. Even for the ciphers which were mentioned, the citations are far from complete. See the "Further Information" section below for general references.

42.11 Defining Terms

Breaking a cipher: Finding the key to the cipher by analysis of the ciphertext or, sometimes, both the plaintext and corresponding ciphertext.

Coprime: Relatively prime. Refers to integers having no common factor greater than 1.

Differential Cryptanalysis: An attack on ciphers like DES that tries to find the key by examining how certain differences (XORs) of plaintext pairs affect differences in the corresponding ciphertext pairs.

Digital signature algorithm: A public-key algorithm for signing or authenticating messages. It was proposed by the National Institute of Standards and Technology in 1991.

Factor base: A fixed set of small primes, usually all suitable primes up to some limit, used in factoring auxiliary numbers in the quadratic and number field sieves.

Homophone: One of several ciphertext letters used to encipher a single plaintext letter.

Key stream: A sequence of bits, bytes, or longer strings used as keys to encipher successive blocks of plaintext.

Permutation or transposition cipher: A cipher that enciphers a block of plaintext by rearranging the bits or letters.

Quadratic residue modulo m: A number r that is coprime to m and for which there exists an x so that $x^2 \equiv r \bmod m$.

Sieve: A number theoretic algorithm in which, for each prime number p in a list, some operation is performed on every pth entry in an array.

Substitution cipher: A cipher that enciphers by replacing letters or blocks of plaintext by ciphertext letters or blocks under the control of a key.

XOR: Exclusive-or or sum of bits modulo 2.

References

[1] Barker, W.G., *Cryptanalysis of the Hagelin Cryptograph,* Aegean Park Press, Laguna Hill, CA, 1977.

[2] Brickell, E.F. and Odlyzko, A.M., Cryptanalysis: A survey of recent results, *Proc. IEEE,* 76, 578–593, May 1988.

[3] Biham, E. and Shamir, A., Differential cryptanalysis of DES-like cryptosystems, In *Advances in Cryptology–CRYPTO '90, Lecture Notes in Computer Science 537*, Menezes, A.J. and Vanstone, S.A., Eds., 2–21, Springer-Verlag, Berlin, 1991.

[4] Biham, E. and Shamir, A., *Differential Cryptanalysis of the Data Encryption Standard,* Springer-Verlag, New York, 1993.

[5] Coppersmith, D., Fast evaluation of logarithms in fields of characteristic two, *IEEE Trans. Inform Theory,* 30, 587–594, 1984.

[6] Chor, B. and Rivest, R.L., A knapsack type public key cryptosystem based on arithmetic in finite fields, In *Advances in Cryptology–CRYPTO '84,* 54–65, Springer-Verlag, Berlin, 1985.

[7] Denning, D.E., *Cryptography and Data Security,* Addison-Wesley, Reading, MA, 1983.

[8] Diffie, W. and Hellman, M., New directions in cryptography, *IEEE Trans. on Info. Theory,* IT-22 (6), 644–654, Nov. 1976.

[9] Diffie, W. and Hellman, M., Exhaustive cryptanalysis of the NBS data encryption standard, *Computer,* 10(6), 74–84, Jun. 1977.

[10] Ding, C., Xiao, G., and Shan, W., *The Stability Theory of Stream Ciphers, Lecture Notes in Computer Science 561,* Springer-Verlag, New York, 1991.

[11] Friedman, W.F., *Elements of Cryptanalysis,* Aegean Park Press, Laguna Hills, CA, 1976.

[12] Hitt, P., *Manual for the Solution of Military Ciphers,* 2nd ed., Army Service Schools Press, Fort Leavenworth, KS, 1918.

[13] Kahn, D., *The Codebreakers,* Macmillan, New York, 1967.

[14] Konheim, A.G., *Cryptography, A Primer,* John Wiley & Sons, New York, 1981.

[15] Lenstra, A.K. and Lenstra, Jr., H.W., *The Development of the Number Field Sieve,* Springer-Verlag, New York, 1993.

[16] Matsui, M., Linear cryptanalysis method for DES cipher, In Menezes, A.J. and Vanstone, S.A., Eds., *Advances in Cryptology–EUROCRYPT '93,* 386–397, Springer-Verlag, Berlin, 1994.

[17] Riesel, H., *Prime Numbers and Computer Methods for Factorization,* 2nd ed., Birkhäuser, Boston, 1994.

[18] Rivest, R.L., Shamir, A., and Adleman, L., A method for obtaining digital signatures and public-key cryptosystems, *Comm. ACM,* 21(2), 120–126, Feb. 1978.

[19] Reeds, J.A. and Weinberger, P.J., File security and the UNIX crypt command, *AT&T Tech. J.,* 63, 1673–1683, Oct. 1984.

[20] Shannon, C.E., Communication theory of secrecy systems, *Bell Syst. Tech. J.,* 28, 656–715, 1949.

[21] Williams, H.C., A $p + 1$ method of factoring, *Math. Comp.,* 39, 225–234, 1982.

Further Information

Research on cryptanalysis is published often in the journals *Journal of Cryptology, Cryptologia* and *Computers and Security.* Such research appears occasionally in the journals *Algorithmica, AT&T Technical Journal, Communications of the ACM, Electronics Letters, Information Processing Letters, IBM Journal of Research and Development, IEEE Transactions on Information Theory, IEEE Spectrum, Mathematics of Computation* and *Philips Journal of Research.*

Several annual or semiannual conferences with published proceedings deal with cryptanalysis. CRYPTO has been held since 1981 and has published proceedings since 1982. See CRYPTO '82, CRYPTO '83, etc. EUROCRYPT has published its proceedings in 1982, 1984, 1985, 1987, 1988, etc. The conference AUSCRYPT is held in even numbered years beginning in 1990, while ASIACRYPT is held in odd numbered years beginning in 1991.

Kahn [13] gives a comprehensive history of cryptology up to 1967. Public key cryptography was launched by Diffie and Hellman [8]. Denning [7] and Konheim [14] have lots of information about and examples of old ciphers and some information about recent ones.

43

Pseudorandom Sequences and Stream Ciphers

Andrew Klapper
University of Kentucky

43.1 Introduction

This chapter concerns the generation of **pseudorandom sequences** and the role of these sequences in **stream ciphers.** Pseudorandom sequences are also used in various probabilistic algorithms that arise in cryptography. In this latter context, however, the role of pseudorandomness is essentially the same as in other probabilistic algorithms and we leave this aspect of pseudorandomness to other chapters.

The only completely secure cryptosystem is the **one-time pad.** In this private key system the message alphabet is the integers modulo an integer m. The *key* or *keystream* is a sequence of symbols from the message alphabet generated uniformly independently at random. The key is known to both the sender and receiver. To encrypt a message, the key is added to the message symbol by symbol modulo m. From the point of view of Shannon's **information theory** this system is unconditionally secure. That is, an adversary who knows any subset of the symbols of the key (or, equivalently, any set of known plaintext/ciphertext pairs) can determine any other symbol of the message with probability no better than guessing. Such a system is further advantageous because encryption is as fast as the method of generating the random symbols. The drawback is that sharing the key between the sender and the receiver is no easier than sharing the message, since the key is as large as the message. Thus the one-time pad is only practical when the sender and receiver have access to a secure channel at some point (when they can share the key) and plan to use an insecure channel to share a message at some later time. Such situations are rare.

A stream cipher uses a pseudorandom sequence as the keystream in place of a truly random one. By pseudorandom we mean that the key is apparently random by various criteria that are related to the effectiveness of the system. Since the actual encryption is simple and well understood, the issues surrounding the effectiveness of a stream cipher have to do mainly with the generation of keys. The symbols of the key must be difficult to determine from a subset of the symbols and the key must be

efficiently generated. There is widespread belief in the cryptographic community that stream ciphers are less secure than either well-designed block ciphers or public key systems. Thus they are considered practical only if they achieve much higher speed. We stress, however, that the relative security of types of cryptosystems is largely a matter of belief, often depending on unproved complexity theoretic assumptions.

Stream ciphers can be implemented either in hardware, which has the advantage of speed of execution, or in software, which has the advantages of speed of implementation and portability. The choice often affects the choice of message alphabet. A binary alphabet is typically used in hardware implementations. An alphabet of size 256 is often used in software implementations. In much of this chapter we treat the case of stream ciphers based on binary sequences, since this case has been more extensively studied. In many instances the analysis is considerably simpler with binary sequences. Also, hardware implementations are essentially finite state devices. Their output sequences are thus eventually periodic and this is often an assumption about the sequences used in stream ciphers.

As with most cryptography there are two aspects to the study of stream ciphers: their design and their cryptanalysis. Of course there is a relationship. Stream ciphers must be designed to resist any general cryptanalytic attacks and the cryptanalysis of a system depends on its design.

Classification and Modes of Stream Ciphers

In the greatest generality a stream cipher is a parametrized non-terminating finite automaton with output. More precisely, it consists of

1. A *state space* $\mathcal{A}$, a key space $\mathcal{K}$, and a message alphabet $\mathcal{M}$;
2. A *state change function* $F : \mathcal{K} \times \mathcal{A} \times \mathcal{M} \to \mathcal{A}$; and
3. An *output function* $g : \mathcal{K} \times \mathcal{A} \times \mathcal{M} \to \mathcal{M}$.

For a given message sequence $x_0, x_1, \cdots \in \mathcal{M}$, initial state $\alpha_0 \in \mathcal{A}$, and key $k \in \mathcal{K}$, the stream cipher generates a state sequence and output (i.e., ciphertext) sequence by

$$\begin{aligned} \alpha_{i+1} &= F(k, \alpha_i, x_i) \\ y_i &= g(k, \alpha_i, x_i) \ . \end{aligned}$$

Most commonly

$$g(k, \alpha, x) = x + h(k, \alpha) \bmod m \ ,$$

where $\mathcal{M}$ is the integers modulo an integer m. In this case the sequence $z_i = h(k, \alpha_i)$ is called the keystream and the pair (F, h) is called a **keystream generator.**

Keystream generators may be classified as *synchronous* or *self-synchronizing.* In a self-synchronizing generator the state depends on the previous few output symbols,

$$\alpha_{i+1} = F\left(y_i, \cdots, y_{i-r+1}\right) \ .$$

In effect the state consists of the previous r output symbols. The advantage of such a system is that if symbols are lost, then the receiver can resynchronize after r output symbols have been received. The disadvantage is that distorted symbols cause error propagation for r symbols.

In a synchronous generator the keystream is independent of the message sequence. That is, $F : \mathcal{K} \times \mathcal{A} \to \mathcal{A}$. Synchronous generators are unable to recover from lost symbols without resetting to an initial state. However, an external mechanism can be added to solve the synchronization problem. Distorted symbols result in no error propagation. Nonetheless, in most applications using noisy channels it is likely that an underlying error correction mechanism will be used. Synchronous generators are the most commonly used stream ciphers.

Synchronous generators are often restricted to one of two special cases. In *counter mode,* $\alpha_{i+1} = F(\alpha_i)$ and $z_i = h(k, \alpha_i)$. An example of a counter mode generator is the *cyclotomic generator,* described below.

In *output feedback mode*, $\alpha_{i+1} = F(k, \alpha_i)$ and $z_i = h(\alpha_i)$. Most of the feedback register based generators described in this article run in output feedback mode.

43.2 Underlying Principles

The mathematical tools used to study cryptographically strong pseudorandom sequences are algebra, information theory, and probability theory. In this section various formalizations of the notion of randomness are described. We also describe the structure and analysis of **linear feedback shift registers.** These are fundamental building blocks in the design of keystream generators. Of particular importance in this analysis is the theory of finite fields [22, 23].

Randomness

There are several different notions of randomness that have been applied to sequences. These include *information theoretic randomness, statistical randomness, unpredictability* (a complexity theoretic notion), and *Kolmogorov complexity.* The first three have the greatest relevance for cryptography, and we describe them in more detail next.

Information Theoretic Randomness

The mathematical analysis of cryptographic systems was initiated by Shannon's invention of information theory [37]. Shannon's approach was to analyze the information that can be derived about a system by an adversary who has unlimited computational resources. The analysis is probabilistic. The cryptanalyst has a known ciphertext and knows the probability distribution of plaintexts, keys, and ciphertexts. The *a priori* distribution on the plaintexts arises by assuming they lie in some restricted class of strings such as a natural language. By conditioning this distribution on the known ciphertext, a new distribution on plaintext is determined. The cryptanalyst is successful if one plaintext has conditional probability close to one.

Let X be a discrete random variable whose distribution is given by $p_i = Pr(X = i)$. The **entropy** of X is defined by

$$H(X) = -\sum_{i, p_i \neq 0} p_i \log(p_i) \ .$$

The entropy is a measure of uncertainty about X. Shannon gave a set of properties that the uncertainty intuitively should have and proved that the so-defined entropy function is the unique function that has them. For any X, $H(X) \leq \log(n)$, with equality if and only if $p_i = 1/n$ for all i. The notion of entropy also plays a role in compression. It is a lower bound on the average number of bits required to encode a finite set. An introduction to information theory and its role in coding theory and cryptography can be found in the book by Welsh [38].

If X and Y are two random variables, the notion of entropy extends naturally to the joint entropy $H(X, Y)$, the conditional entropy $H(X|Y)$ (the uncertainty about X if Y is known), and the mutual information

$$I(X; Y) = H(X) - H(X|Y)$$

between X and Y (the information common to X and Y). If $I(X; Y) = 0$, then X and Y are independent, so Y reveals nothing about X. Conversely, if $I(X; Y) = H(X)$, i.e., is maximal, then $H(X|Y) = 0$ so there is no uncertainty about X when Y is known.

Now consider the case of a cryptosystem. Suppose that $X^n = x_1, x_2, \cdots, x_n$ is an n-bit plaintext, K is a key, and $Y^n = y_1, y_2, \cdots, y_n$ is the corresponding n-bit ciphertext. In general,

$$H\left(K|Y^n\right) = H\left(X^n|Y^n\right) + H\left(K|X^n, Y^n\right) \ .$$

If $I(X^n; Y^n) = 0$, then the ciphertext reveals nothing of the plaintext and the system is said to have *perfect secrecy.* In this case

$$H(K) \geq H\left(K|Y^n\right) \geq H\left(X^n|Y^n\right) = H\left(X^n\right) .$$

Thus the uncertainty about the key must be at least as great as the uncertainty about the plaintext. It follows that the average length of the minimal length encoding of the key must be at least as great as that of the plaintext. That is, to achieve perfect secrecy the key must be as long as the plaintext. This is a fundamental limit on the power of private key encryption.

At the other extreme, if $I(X^n; Y^n)$ is asymptotic to $H(X^n)$, then for long enough plaintexts knowledge of the ciphertext essentially determines the plaintext and the system can be broken (although it still may require an exhaustive search to find the plaintext).

In the intermediate cases $0 < I(X^n; Y^n) < rH(X^n)$ for some positive constant $r < 1$. For arbitrarily long messages there is some uncertainty about the plaintext. It follows that

$$H\left(K|Y^n\right) > (1-r)H\left(X^n\right)$$

is bounded away from zero. Such a cryptosystem is called *ideally secure.* This means that the key can never be determined from the ciphertext with perfect certainty. However, the uncertainty may be small enough that significant portions of the plaintext are revealed.

Shannon also defined a measure of how much ciphertext is necessary to determine the key. The *unicity distance* n_u is the minimum length n such that $H(K|Y^n) \sim 0$. Under reasonable assumptions, we have

$$n_u = \frac{H(K)}{1-h} ,$$

where h is the information rate of the plaintext (so $1-h$ is the redundancy).

Statistical Randomness

A sequence is statistically random if various statistical properties are close to those of a truly random sequence. The failure of these properties to hold does not necessarily lead directly to a cryptanalytic attack. However, such failure is reason for concern that an attack may be found in the future. Following are some of the randomness criteria that have been considered for periodic binary sequences. Let $\mathbf{a} = a_0, a_1, a_2, \cdots$ denote such a sequence. Its period, denoted $\rho(\mathbf{a})$, is the least positive integer n such that $a_i = a_{i+n}$ for every i. The period of a keystream must be large.

Balance: A sequence is **balanced** if the number of occurrences of 0 equals the number of occurrences of 1 in each period. In the extreme, a highly unbalanced sequence allows a cryptanalyst to read most of a message either directly or by complementing all bits. In less extreme cases, unbalanced sequences have been shown to leak essential information [18]. Of course if $\rho(\mathbf{a})$ is odd, then $\mathbf{a}$ cannot be perfectly balanced.

Subsequence Distribution: If r is any integer, we can count the number of occurrences of each binary r-tuple $b_0, \cdots, b_{r-1}$ in a period of $\mathbf{a}$ (more precisely, we count the number of indices i such that $0 \leq i \leq n-1$, and $a_i = b_0, a_1 = b_{i+1}, \cdots a_{i_r-1} = b_{r-1}$). It is desirable that the distribution of occurrences be close to uniform. A sequence of period 2^r such that every binary r-tuple occurs exactly once in a period is known as a *de Bruijn sequence.*

Autocorrelations: The **autocorrelation with shift** τ of $\mathbf{a}$ is defined as

$$A_{\mathbf{a}}(\tau) = \sum_{i=0}^{\rho(\mathbf{a})-1} (-1)^{a_i + a_{i+\tau}} ,$$

or, equivalently, the number of bits in one period of $\mathbf{a}$ and a τ shift of $\mathbf{a}$ that are equal minus the number of bits that are not equal. If $\mathbf{a}$ is independent of its τ shift, then the autocorrelation

is zero, while if for each i, a_i determines $a_{i+\tau}$, then the autocorrelation is plus or minus $\rho(\mathbf{a})$. Thus the autocorrelation measures the extent to which a sequence is independent of its shifts. The failure of the autocorrelation to be close to zero can sometimes be used to derive essential information about a cryptosystem.

Run Property: A *run* in a sequence is a maximal subsequence consisting of only zeros or only ones. A binary sequence can be thought of as a series of runs of varying lengths. A sequence has the *run property* if the distribution of runs is close to what would be expected of a truly random sequence. If a run starts at position i, then it has length r with probability 2^{-r}.

Nonlinearity: Generally, linearity can be exploited in constructing cryptanalytic attacks. Thus, loosely speaking, the sequence should not exhibit linear structure. This statement can be interpreted in various ways. For example, the sequence defines a function from its set of indices to $\{0, 1\}$. Whenever we interpret the index set as an algebraic structure (say, modular integers or a finite field), this function should be highly nonlinear. Various approaches have been taken to defining the degree to which such a function is nonlinear.

A Boolean function f on n bits satisfies the **strict avalanche criterion** (or SAC) if $f(\bar{x}) + f(\bar{x} + \bar{y})$ is a balanced function of $\bar{x}$ for every $\bar{y}$ with $||\bar{y}|| = 1$. Here $||\bar{y}||$ denotes the number of ones in $\bar{y}$, known as its Hamming weight. That is, changing any single bit of the input to f gives a function that is uncorrelated with f. The function f satisfies $SAC(k)$ if holding any k bits constant results in a function that satisfies SAC. Preneel et al. gave conditions under which various $SAC(k)$ hold [28]. More generally, f satisfies the *propagation criterion of degree k* if $f(\bar{x}) + f(\bar{x} + \bar{y})$ is a balanced function of $\bar{x}$ for every $\bar{y}$ with $1 \leq ||\bar{y}|| \leq k$.

The *Walsh transform of f* is defined as

$$\hat{F}(\bar{w}) = \sum_{\bar{x}} (-1)^{f(\bar{x}) + \bar{w} \cdot \bar{x}} .$$

Parseval's theorem says that

$$\sum_{\bar{w}} \left(\hat{F}(\bar{w}) \right)^2 = 2^{2n} .$$

A function is *bent* if each Walsh transform achieves exactly the average value implied by Parseval's theorem,

$$\left| \hat{F}(\bar{w}) \right| = 2^{n/2}$$

for every w [30]. That is, the maximum correlation of f with a linear function is as small as possible.

The notions of **correlation immunity** and the degree of *algebraic nonlinearity* are described in subsequent sections.

Unpredictability

In a very different attempt to define a notion of randomness suitable for cryptography, Yao [39] and Blum and Micali [3] considered what would make it effectively impossible for an adversary to determine the next bit of a sequence if a prefix were known. The adversary in their model is assumed to have limited resources, so their definition of unpredictability is complexity theoretic.

In Blum and Micali's model a generator G inputs a security parameter n and a random number $0 \leq i < 2^n$ and outputs a pseudorandom bit sequence $\mathbf{a}$. Such a generator is a **cryptographically strong pseudorandom bit generator** or CSPRB generator if the following hold.

1. The bits a_j are easy to generate. That is, it should take time polynomial in n to output the jth bit.

2. The bits are unpredictable. Given G, n, and $a_0, \cdots, a_{j-1}$, but not i, it should be computationally infeasible to predict a_j with probability significantly greater than 1/2.

More precisely, suppose the output from G has length $p(n)$ if n is the security parameter. A predicting family is a polynomial (in n) size family of circuits

$$C = \left\{ C_n^j : j < p(n) \right\}$$

such that each C_n^j has j inputs and one output. If the input is $a_0, \cdots, a_{j-1}$, and the output is d, let $P_{j,i}$ be the probability that $d = a_j$. Then G passes the **next bit test** C if for every polynomial $q(n)$, every large enough n, every $j < p(n)$, and every $i < 2^n$,

$$P_{j,i} < \frac{1}{2} + \frac{1}{q(n)} .$$

A generator is *perfect* if it passes all polynomial size next bit tests. A different, but equivalent, formulation was given by Yao. In the subsection on universal security in Section 43.3, we describe Blum and Micali's construction of a generator that passes the next bit test assuming the hardness of the discrete log problem.

Feedback Shift Registers

Many devices that are proposed for generating cryptographically strong pseudorandom sequences are based on linear feedback shift registers or LFSRs for short [14]. An LFSR of length r has an r-bit state vector that is updated by shifting by one position and filling the vacated position by a linear function of the previous state. More specifically, if we let the state be $\bar{a} = (a_0, \cdots, a_{r-1})$, $a_i \in \{0, 1\}$, then there is a linear *feedback function*

$$f(\bar{a}) = \left(\sum_{i=0}^{r-1} c_i a_i \right) \bmod 2 ,$$

where each c_i is a bit. The state is updated by replacing $\bar{a}$ with $(a_1, \cdots, a_{r-1}, f(\bar{a}))$. A diagram of an LFSR is given in Fig. 43.1.

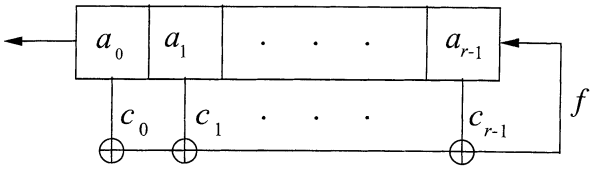


FIGURE 43.1 Linear feedback shift register.

In hardware the state change can be thought of as tapping the cells whose corresponding c_is equal 1 and adding the values modulo 2. Thus LFSRs are extremely fast, especially when implemented in hardware and when the number of tapped cells is small. They can be designed to generate sequences of large period—up to $2^r - 1$. A sequence of period $2^r - 1$ output by an LFSR of length r is called an **m-sequence.** These sequences have many of the desirable statistical properties mentioned in "Statistical Randomness." They are optimally balanced for odd period sequences. Every subsequence of length $t \leq r$ occurs 2^{r-t} times, except the all-zero subsequence which occurs $2^{r-t} - 1$ times. The shifted autocorrelations all equal -1. The distribution of subsequences of each length up to r is nearly uniform and the distribution on runs is nearly perfect.

There is a useful algebraic theory of LFSRs. To describe this, some algebraic background is needed. If $q = 2^r$, then there is a unique *finite field* with q elements, called $GF(q)$. An element α of $GF(q)$ is

called *primitive* if every nonzero element of $GF(q)$ is a power of α. (Compare to Chapter 38, Section 5.9). Primitive elements exist in every $GF(q)$. The minimal degree polynomial with coefficients in $GF(2) = \mathbf{Z}/2\mathbf{Z} = \{0, 1\}$ for which α is a root is called a *primitive polynomial.* We also need the *trace function,* defined by

$$Tr_1^r(x) = x + x^2 + x^4 + \cdots + x^{2^{r-1}} .$$

The trace function maps $GF(q)$ to $GF(2)$, is nonzero, and is $GF(2)$-linear, thinking of $GF(q)$ as a vector space over $GF(2)$. In fact, every such linear function is of the form $x \mapsto Tr_1^r(\gamma x)$ for some $\gamma \in GF(q)$. These notions generalize to a setting where the base field $GF(2)$ is replaced by an arbitrary finite field of arbitrary characteristic. This makes it possible to extend the following analysis of LFSR sequences to LFSRs whose entries lie in an arbitrary finite field.

We can associate to any eventually periodic sequence **a** the *generating function*

$$g(x) = \sum_{i=0}^{\infty} a_i x^i .$$

We can associate to any LFSR with feedback function $\sum_{i=0}^{r-1} c_i a_i$ the *connection polynomial*

$$q(x) = \sum_{i=1}^{r} c_{r-i} x^i - 1 .$$

These are defined over $GF(2)$. Then $g(x)$ is a rational function and, when represented as a quotient of two relatively prime polynomials, the denominator is the connection polynomial of the smallest LFSR that outputs **a**. An LFSR sequence is an m-sequence if and only if the connection polynomial is primitive. The period of **a** is the least n such that $g(x)$ divides $x^n - 1$. Also, it can be shown that if the connection polynomial of an LFSR is irreducible, then there are elements $\gamma, \alpha \in GF(2^r)$ such that

$$a_i = Tr_1^r\left(\gamma \alpha^i\right) .$$

The sequence **a** is an m-sequence if and only if α is primitive. It is these algebraic structures that make it possible to analyze many of the statistical properties of m-sequences mentioned above.

Despite their nice statistical properties m-sequences are cryptologically weak. This is due to the linearity of the feedback function and the associated algebraic structures.

If **a** is any sequence, then the size of the smallest LFSR that outputs **a** is called the **linear span** or **linear complexity** of **a**. We denote this quantity by $\lambda(\mathbf{a})$. There is an algorithm, due to Berlekamp and Massey [24] which, given $2\lambda(\mathbf{a})$ bits of a sequence, outputs a description of a minimal length LFSR that generates **a**. This algorithm is given in Fig. 43.2. At the kth stage the best rational representation of the generating function modulo x^k is found. Details of the proof that the algorithm achieves this and converges in $2\lambda(\mathbf{a})$ steps were given by Massey [24]. Furthermore, the Berlekamp–Massey algorithm can be made efficient by computing the product in step 11 from previous values in linear time. Thus if the algorithm examines T bits of a sequence, then it runs in time $\mathcal{O}(T^2)$. If the period of a sequence is exponentially larger than its linear span, then this time is quite small. This is the case for m-sequences. Thus LFSRs are unsuitable for generating sequences for stream ciphers. Moreover, sequences generated by other means must have large linear span or they will still be susceptible to the Berlekamp–Massey algorithm.

A major goal of research on stream ciphers has been to design efficient keystream generators whose output has large linear span. Despite the cryptographic weakness of LFSRs, their speed, simplicity, and ability to be analyzed make them important building blocks for stream ciphers. They are useful as well in other areas such as spread spectrum systems, radar systems, and Monte Carlo simulation. Many variations on LFSRs have been proposed that gain cryptographic strength by introducing some nonlinearity. Several of these are described in Section 43.3.

BERLEKAMP-MASSEY($\mathbf{a}$)

```
1   input a_i s until the first nonzero a_{k-1} is found
2   g ← a_{k-1} · x^{k-1}
3   p_{k-1} ← 0
4   q_{k-1} ← 1
5   m ← k − 1
6   p_k ← x^{k-1}
7   q_k ← x^{k-1} + 1
8   while there are more bits
9       do  input a new bit a_k
10          g ← g + a_k x^k
11          if g · q_k ≡ p_k ( mod x^{k+1})
12              then q_{k+1} ← q_k
13                   p_{k+1} ← p_k
14              else q_{k+1} ← q_k + x^{k−m} q_m
15                   p_{k+1} ← p_k + x^{k−m} p_m
16                   if deg(q_{k+1}) > deg(q_k)
17                       then m ← k
18          k = k + 1
19  return q_k, p_k
```

FIGURE 43.2 The Berlekamp–Massey algorithm.

It has long been known that a small change in a sequence can result in an enormous change in the linear span. For example, the all 0 sequence has linear span 0, but if we change every nth position to a 1, then the resulting sequence has linear span n (any register of length less than n that outputs this sequence must reach an all 0 state, and will output all 0s from then on, which is a contradiction). Suppose $\mathbf{a}$ is any sequence of period n and $\mathbf{b}$ is another sequence with the same period that differs from $\mathbf{a}$ in a small number k of bits per period. If a cryptanalyst is given a prefix $\bar{u} = a_0, \cdots, a_{m-1}$ of $\mathbf{a}$, she can run the Berlekamp–Massey algorithm for every m-tuple that differs from $\bar{u}$ in at most k positions. Among the resulting generators will be one that outputs $\mathbf{b}$, although it may be problematic deciding which generator to select. This led Ding, Xiao, and Shan [10] to define the **sphere complexity.** Let $O(\mathbf{a},k)$ be the set of sequences of period n that differ from $\mathbf{a}$ in at least 1 and at most k positions. Then the sphere complexity of $\mathbf{a}$ is defined as

$$\mathrm{SC}_k(\mathbf{a}) = \min_{\mathbf{b}\in O(\mathbf{a},k)} \lambda(\mathbf{b}) .$$

Although less important than the linear span, it is desirable that the sphere complexity of a sequence be large.

LFSRs can be generalized to feedback registers whose entries are elements of an arbitrary fixed finite field $GF(q)$. The coefficients a_i are arbitrary elements of $GF(q)$. Such a register outputs a sequence of

elements of $GF(q)$. Essentially everything we have said about LFSRs applies in this more general setting, although some slight modification is necessary in steps 14 and 15 of the Berlekemp–Massey algorithm.

43.3 State of the Art

The importance of cryptanalysis is threefold. First, it reveals weaknesses in existing systems so that users know what to avoid. Second, it provides a limited means of certification: users are more confident in a system that has withstood attack for a reasonable time. This is important in an area where formal proofs of security seem unlikely and systems often intentionally defy formal analysis. Third, the study of cryptanalysis often reveals general principles for the design of future cryptosystems. In addition to the Berlekamp–Massey algorithm there have been several general methods of cryptanalysis of stream ciphers. In the subsection on "Cryptanalysis" we discuss **correlation attacks** [12, 26, 36] and 2-adic rational approximation [20].

The ultimate goal of research on stream ciphers is to provide fast methods of securely transmitting data. Methods that have been proposed in recent years include feedback shift register based methods such as **nonlinear filter generators** [15]; **nonlinear combiners** [31]; **clock-controlled shift registers** [13]; shrinking generators; and cyclotomic generators [9]. There have also been several recent proposals of generators that are not based on shift registers, and are suitable for software implementations. These include RC4 and SEAL [29]. In the subsection on "Keystream Generation" we survey these approaches.

In contrast to public key cryptosystems, the theoretical foundations of stream ciphers are generally weak or only weakly connected to practice. In the subsection on "Universal Security" we discuss complexity theoretic models for stream cipher security [3, 39]; and models for security against broad generalizations of the Berlekamp–Massey algorithm [19].

Cryptanalysis

In this section we describe general methods of cryptanalyzing stream ciphers.

Correlation Attacks

Consider a situation in which one or more feedback registers are made to interact to produce an output sequence. Suppose a cryptanalyst knows the structure of the feedback registers and how they interact but not the start states. This is a typical arrangement when keystream generators are implemented in hardware. The goal of the cryptanalyst then is to determine the initial states of the registers from a known segment of key stream. If the generator is constructed so that the state is large enough, then an exhaustive search is infeasible. The idea behind a correlation attack is to find statistical correlations between keystream bits and state bits. These correlations can then be used to improve searches for the initial states. Typically, exhaustive search is performed on the start state of one of the underlying registers until the correlations match the prediction.

This general framework has been used to attack combination generators, nonlinear filter generators, and various clock controlled shift registers. These attacks are described in greater detail in the sections below on the specific generators.

2-Adic Rational Approximation

The method of 2-adic rational approximation is based on a class of feedback registers invented by Goresky and Klapper [20] that is analogous to LFSRs. These registers, called **feedback with carry shift registers** or FCSRs, are based on algebra over the integers and 2-adic numbers just as LFSRs are based on algebra over polynomials and power series. An FCSR of length r has an r-bit state vector $\bar{a} = (a_0, \cdots, a_{r-1})$ plus an integer memory m. The state is updated similarly to that of an LFSR, but the addition is performed as integers rather than modulo 2. The low bit is fed back to the register and the high

bits are retained in m as a carry to the next state change. More precisely, there is a linear feedback function

$$f(\bar{a}, m) = m + \sum_{i=0}^{r-1} c_i a_i \ ,$$

where each c_i is a bit. The state is updated by replacing $\bar{a}$ with

$$(a_1, \cdots, a_{r-1}, f\ (\bar{a}, m) \bmod 2)$$

and replacing m by

$$\lfloor f\ (\bar{a}, m)\ /2 \rfloor \ .$$

A diagram of an FCSR is given in Fig. 43.3.

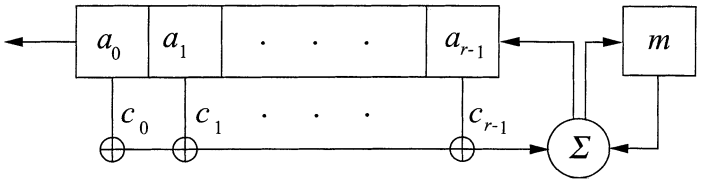


FIGURE 43.3 Feedback with carry shift register.

FCSRs are very simple and fast devices that can output sequences that are exponentially larger than the size of the register. There are many analogies between LFSRs and FCSRs. Associated with the output sequence **a** is the *2-adic number*

$$\alpha = \sum_{i=0}^{\infty} a_i 2^i \ .$$

The algebra of 2-adic numbers is like that of power series, but addition and multiplication are performed with carry to higher terms instead of modulo 2. Associated with the FCSR is the *connection number*

$$q = \sum_{i=1}^{r} c_{r-i} x^i - 1 \ .$$

An FCSR sequence is always eventually periodic. The associated 2-adic number α is rational, and q is the denominator of a rational representation of α. Thus the cryptanalytic problem of finding the smallest FCSR that outputs a given sequence is equivalent to the problem of finding the minimal rational representation for a 2-adic number. This problem was solved by de Weger's algorithm. An adaptive version appears in [20], where the notion of *2-adic span* (the minimal number of bits of storage used by an FCSR that outputs **a**) is defined. As with linear span, in order for a sequence to be cryptographically strong it must have large 2-adic span. It was further shown that sequences generated by summation combiners (see "Nonlinear Combiners") have relatively low 2-adic span. Various generalizations of FCSRs have been proposed by exploiting known generalizations of the algebra of 2-adic numbers.

Despite this cryptanalysis, maximal period FCSR sequences (ones for which 2 is a primitive root modulo the connection number) have many desirable statistical properties. Their periods are exponentially larger than the sizes of their generators, they are balanced, their arithmetic correlations vanish, and they are nearly de Bruijn sequences. Thus they are potential substitutes for m-sequences as building blocks for keystream generators.

Keystream Generation

In this section we survey various methods of generating sequences for stream ciphers. Much of the research in this area has concentrated on generating sequences with large linear span and immunity to correlation attacks.

Linear Congruential Generators

Linear congruential generators are often suggested as pseudorandom generators because they are simple, have large period, and are readily available on computer systems. However, they are highly linear and hence are cryptographically weak.

A linear congruential generator is determined by an integer modulus, m, and a pair of integers, u and v, with $0 < u < m$ and $0 \leq v < m$. A sequence of integers $a_0, a_1, a_2, \cdots$ is generated by choosing a_0 arbitrarily and computing

$$a_n = ua_{n-1} + v \bmod m \ .$$

If v is relatively prime to m and u has maximal order modulo m (that is, $u^i = 1 \bmod m$ only if $\phi(m)$ divides i, where ϕ is Euler's function), then this sequence has maximal period $\phi(m)$.

Several attacks on linear congruential generators have appeared in the literature. The one due to Boyar assumes that u, v, and m are unknown and the cryptanalyst does not know the $\log(\log(m))$ low order bits of each a_n [4]. Lagarias and Reeds have described an attack on a generalization of the linear congruential generator in which the linear function is replaced by an arbitrary polynomial.

Nonlinear Combiners

A nonlinear combiner takes the outputs from a set of k LFSRs and combines them with a nonlinear function $h : GF(2)^k \rightarrow GF(2)$. We denote by $\mathbf{a}^j$ the output from the jth LFSR. Then the output $\mathbf{b}$ of the nonlinear combiner is the sequence whose ith bit is $b_i = h(a_i^1, \cdots, a_i^k)$. A diagram of a nonlinear combiner with $k = 2$ is given in Fig. 43.4.

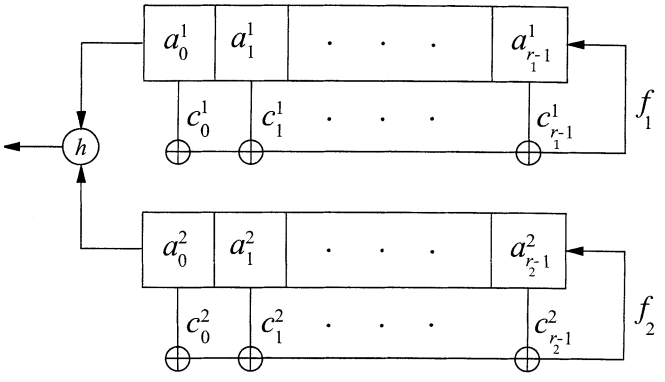


FIGURE 43.4 Nonlinear combiner.

The *Geffe generator* is a special case using three LFSRs. The output from the third register is used to select between the first two. That is, $h(a^1, a^2, a^3) = a^3a^1 \oplus (\neg a^3)a^2$. The period of the Geffe generator is $n_1n_2n_3$ and the linear span is

$$\lambda\left(\mathbf{a}^1\right)\lambda\left(\mathbf{a}^3\right) + \lambda\left(\mathbf{a}^2\right)\left(1 + \lambda\left(\mathbf{a}^3\right)\right) \ .$$

However, state information is leaked since

$$\text{Prob}\left(h\left(a^1, a^2, a^3\right) = a^1\right) = \text{Prob}\left(h\left(a^1, a^2, a^3\right) = a^2\right) = \frac{3}{4}\,.$$

Another special case is the *threshold generator.* This generator combines k LFSR sequences by outputting a 1 if and only if the majority of the outputs are 1. That is,

$$h\left(a^1, \cdots, a^k\right) = \begin{cases} 1 & \text{if } \sum_{i=1}^{k} a^i > \frac{k}{2} \\ 0 & \text{otherwise}\,. \end{cases}$$

In case $k = 3$ the period is $n_1 n_2 n_3$ and the linear span is

$$\lambda\left(\mathbf{a}^1\right)\lambda\left(\mathbf{a}^2\right) + \lambda\left(\mathbf{a}^2\right)\lambda\left(\mathbf{a}^3\right) + \lambda\left(\mathbf{a}^1\right)\lambda\left(\mathbf{a}^3\right)\,.$$

Once again, however, there is a positive correlation between the output sequence $\mathbf{b}$ and each sequence $\mathbf{a}^i$. Specifically, the mutual information $I(\mathbf{b}; \mathbf{a}^i)$ is 0.189 bits.

In the general case it is known that the period is bounded by

$$\rho(\mathbf{b}) \le \text{lcm}\left(\rho\left(\mathbf{a}^1\right), \cdots, \rho\left(\mathbf{a}^k\right)\right)$$

and the linear span is bounded by

$$\lambda(\mathbf{b}) \le h^*\left(\lambda\left(\mathbf{a}^1\right), \cdots, \lambda\left(\mathbf{a}^k\right)\right)\,,$$

where h^* is h thought of as a polynomial over the integers [32]. Further, Key [17] showed that these inequalities become equalities when the $\mathbf{a}^j$ are m-sequences with relatively prime periods. These results generalize to sequences over arbitrary finite fields.

Nonlinear combiners can be further generalized by allowing the combining function to retain a small amount of memory, say m bits. Such a combiner is specified by a pair of functions,

$$h : GF(2)^k \times GF(2)^m \to GF(2)$$

and

$$u : GF(2)^k \times GF(2)^m \to GF(2)^m\,.$$

Thus if the state of the memory is c, then the combiner outputs $b_i = h(a_i^1, \cdots, a_i^k, c)$ and updates the memory by $c = u(a_i^1, \cdots, a_i^k, c)$. Of particular interest is the *summation combiner* which adds its input sequence with carry, using the extra memory bits to save the carry. That is,

$$h\left(a^1, \cdots, a^k, c\right) = c + \sum_j a^j \bmod 2$$

and

$$u\left(a^1, \cdots, a^k, c\right) = \left\lfloor \left(c + \sum_j a^j\right)/2 \right\rfloor\,,$$

where we treat c as an integer. Rueppel [31] showed that if the input sequences to a summation combiner are m-sequences with relatively prime periods $\rho_1, \cdots, \rho_k$, then the output of the sequence has period $\prod_i \rho_i$ and linear span close to its period. Summation combiners are susceptible to a 2-adic rational approximation attack.

Combiner generators are vulnerable to correlation attacks. This was first observed by Siegenthaler. The cryptanalyst is assumed to know the combining function h and the individual LFSRs, but not the initial states of the LFSRs. This is equivalent to saying that the cryptanalyst knows the individual LFSR sequences but not the phase shifts that gave rise to the keystream. The idea of Siegenthaler's correlation attack is to pick one of the input sequences to h and compute the correlation of each phase shift with the known bits of the keystream until one is found that matches the predicted correlation. The time needed to determine the initial states of all the LFSRs is proportional to the sum of their periods times the number of known keystream bits. This is much smaller than the time needed to do an exhaustive search of all states, which is proportional to at least the products of the periods of the sequences. An attack such as this that attacks a piece of a generator at a time is sometimes called a *divide-and-conquer attack.*

To compute statistics it is assumed that each input to h is a sequence of independent uniformly distributed binary random variables. Let $\mathbf{b}$ be the output sequence and suppose $b_0, \cdots, b_{n-1}$ are known. For each j let p_j be the probability that the jth input to h equals the output. If p_j is close to $1/2$, then n must be enormous. On the other hand suppose p_j is far from $1/2$. For each phase shift τ the correlation

$$C_{\mathbf{a}^j,\mathbf{b}}(\tau) \stackrel{\text{def}}{=} \frac{1}{n}\sum_{i=0}^{n-1}(-1)^{a_{i+\tau}^j}(-1)^{b_i}$$

is computed. For some T the phase shifts that give the T best correlations are chosen as candidates. (Best means closest to $2p_j - 1$, the *a priori* expected correlation.) If n and T are large enough, then the probability that the correct phase shift is in the candidate set is large. For example, suppose the length of the jth LFSR is 41, $p_j = 0.75$, and $n = 300$. Then the probability that the correct phase shift is among the best 1000 candidates is 0.98.

In order to build a combiner that resists this correlation attack, it is necessary that all the probabilities p_j be close to $1/2$. However, even if this is the case it may be possible to find a correlation between the output from h and a small subset of its inputs. In this case a similar divide-and-conquer attack can be used. The goal is then to find a combining function h that has no such correlations. This gives rise to the notion of **correlation immunity,** which was defined by Siegenthaler [34].

DEFINITION 43.1 A function $h(x_1, x_2, \cdots, x_k) : GF(2)^k \to GF(2)$ is *mth order correlation immune* if the random variable given by any m-tuple of x_js is statistically independent of $h(x_1, x_2, \cdots, x_k)$.

This condition is equivalent to the condition that, for every choice of binary vector $\bar{w} = (w_1, \cdots, w_k)$ with $||\bar{w}|| \leq m$, $h(x_1, x_2, \cdots, x_k)$ is statistically independent of the inner product $\bar{w} \cdot \bar{x}$. It can also be shown that h is mth order correlation immune if and only if $\hat{H}(\bar{w}) = 0$ whenever $1 \leq ||\bar{w}|| \leq m$ (where $\hat{H}(\bar{w})$ is the Walsh transform of h).

Siegenthaler showed that there is a trade-off between the order of correlation immunity and the attainable nonlinearity. The *nonlinear order* of h is the maximum number of variables in any monomial in the algebraic normal form of h. If h is mth-order correlation immune and $1 \leq m < k$, then its nonlinear order is at most $k - m$. This is lowered to $k - m - 1$ if h is balanced and $m \neq k - 1$, and this is tight. Since both measures cannot be large, memoriless nonlinear combiners are cryptographically weak.

Combiners with memory were invented in part to remedy this. For example, a combiner with output function

$$h(\bar{x}, \bar{y}) = \sum_{i=1}^{k} x_i + g(\bar{y}) \bmod 2$$

is $(n-1)$st-order immune for any nonzero g and any state change function u. Thus g and u can be chosen to satisfy any nonlinearity conditions. Meier and Staffelbach [26] observed that the correlation immunity of such combiners arises because the linear functions applied to input bits in computing the correlations

fail to take earlier bits into account. However, if at stage i one considers all bits

$$\left\{a_\ell^j : 0 \le \ell \le i \text{ and } 1 \le j \le k\right\} ,$$

then there must be correlations between the ith output bit b_i and linear functions of the form

$$\sum_{\ell=0}^{i}\sum_{j=1}^{k} w_{\ell,j} a_\ell^j .$$

For the summation combiner and for general combiners with a single bit of memory Meier and Staffelbach found explicit linear functions for which these correlations are large. This analysis was generalized to combiners with arbitrary amounts of memory by Golić [12], who showed that if the size of the memory is sufficiently large, then the correlations between inputs and outputs can be kept small.

Nonlinear Filter Generators

A nonlinear filter generator applies a nonlinear function h to the state of a linear feedback shift register to produce the output sequence **b**. Thus if the register has length r, then h is a function from $GF(2)^r$ to $GF(2)$. For speed and ease of implementation it is desirable that h have few terms when expressed as a polynomial. If n is the period of the underlying LFSR sequence, then it is possible to generate any sequence of period dividing n by a nonlinear filter generator. However, when the function h is expressed as a polynomial, it may have as many terms as the period of the sequence. Key [17] showed that if h has degree d then

$$\lambda(\mathbf{b}) \le \sum_{i=1}^{d} \binom{r}{i} .$$

This result was generalized by Chan, Goresky, and Klapper [6] to registers with nonlinear feedback functions. It follows from Key's result that h must have high degree. Lower bounds are the real need for cryptographic purposes but such results have been rare. Kumar and Scholtz [21] showed that if h is a bent function and 4 divides r, then

$$\lambda(\mathbf{b}) \ge 2^{r/4}\binom{r/2}{r/4} \sim 2\frac{2^{r/2}}{\sqrt{\pi r}} .$$

Weaker lower bounds for more general classes of filter generators have been shown by Bernasconi and Günther and Rueppel [31]. It is unknown how to construct filter generators with maximal linear span where h has few terms.

Chan and Games considered the following generalization. Let **a** be an m-sequence over a finite field $GF(q)$. Let $h : GF(q) \to GF(2)$. Let $b_i = h(a_i)$. The sequence **b** is called a *geometric sequence.* If q is even, then h can be represented algebraically and results of Herlestam [16] and Brynielsson can be used to show that if $h(x) = \sum_{i=0}^{q-1} c_i x^i$, then

$$\begin{aligned} \lambda(\mathbf{b}) &= \sum_{c_i \neq 0} \lambda(\mathbf{a})^{||i||} \\ &\le q^{\log_2(\lambda(\mathbf{a})+1)} . \end{aligned}$$

To be cryptographically strong, a register of this size would need a linear span close to $q^{\lambda(\mathbf{a})}$. Chan and Games showed that if q is odd, then $\lambda(\mathbf{b})$ can be made as large as $q^{\lambda(\mathbf{a})-1}$. Furthermore, geometric sequences have optimal autocorrelations (and in some cases low cross-correlations). However, Klapper later showed that if one considers these sequences as sequences over $GF(q)$ (that simply happen to have only two values), then the linear span is low and, by exploiting the imbalance of the sequences, the parameter q can be found with a probabilistic attack [18].

Filter generators are vulnerable to correlation attacks. This was first observed by Siegenthaler [35]. The easiest way to see this is to consider a filter generator whose underlying LFSR has length r as a nonlinear combiner with r input sequences. The LFSRs generating the input sequences are identical, but the initial state of the ith register is the second state of the $(i-1)$st register. Then the same correlation attack that was used on a nonlinear combiner can be used. In fact the attack is now faster since all the underlying registers have the same structure. It is only necessary to cycle through the set of initial states once looking for correlations.

Clock-Controlled Generators

A quite different way to introduce nonlinearity in a generator is to irregularly clock certain parts of the generator. A survey of these clock-controlled generators as of 1989 was given by Gollman and Chambers [13]. In a simple case, two LFSRs are used: L_1 and L_2 of lengths n_1 and n_2, respectively. We are also given a function

$$f : \{0, 1\}^{n_1} \to \mathbf{Z} .$$

At each step we clock L_1 once and extract $f(s_i)$ where $s_i = (s_{i,1}, \cdots, s_{i,n_1})$ is the ith state of L_1. Register L_2 then changes state (i.e., is clocked) $f(s_i)$ times and the bit produced by the last state change is taken as the ith output of the clock controlled generator (if $\mathbf{b}$ is the output from L_2, this is $c_i = b_{\sigma(i)}$, where $\sigma(i) = \Sigma_{i=0}^{i} f(s_i)$). The case when $f(s_i) = s_{i,1}$ is called the *stop-and-go generator* and is weak, since each change in the output reveals that a 1 has been generated by L_1. Also, there is a large correlation between consecutive output bits. The strength is improved by taking $f(s_i) = 1 + s_{i,1}$, giving rise to the *step-once-twice generator.*

For general clock-controlled generators, if ρ_i is the period of L_1 and T is the sum of the f values of the states of L_1 in one period, then the period of the output sequence $\mathbf{c}$ is

$$\rho(\mathbf{c}) = \rho_1 \rho_2 / \gcd(T, \rho_2) .$$

Assuming $\gcd(T, \rho_2) = 1$, the period is maximal. We make this assumption for the remainder of this discussion. The linear span of $\mathbf{c}$ is upper bounded by $\lambda(\mathbf{c}) \leq n_2 \rho_1$, hence is large but not maximal. If L_1 generates an m-sequence, then the stop-and-go generator achieves the upper bound. If L_1 and L_2 generate the same m-sequence, then both the stop-and-go and step-once-twice generators achieve the maximum linear span [1].

Several authors have described correlation attacks on clock controlled shift registers. For example, Golić showed that in some circumstances the feedback polynomial and the initial state of the clocked register can be determined [11].

A variation on the stop-and-go generator, called the *alternating step generator,* was proposed by Günther. Two stop-and-go generators are used, sharing the same control (L_1) register. The outputs are then added modulo 2. Let ρ_2 and ρ_2' be the periods of the L_2 registers, which have lengths n_2 and n_2'. Assume that the control register produces a de Bruijn sequence of period 2^m and the connection polynomials for the L_2 registers are irreducible. Then the output period is $2^m \rho_1 \rho_2$. The output linear span satisfies

$$\left(n_2 + n_2'\right) 2^{m-1} < \lambda(\mathbf{c}) \leq \left(n_2 + n_2'\right) 2^m .$$

Günther also gave conditions under which the distribution of subsequences and the autocorrelations are close to ideal. Unfortunately, the alternating step generator is vulnerable to a divide-and-conquer attack against the control register.

Clock-controlled shift registers can be extended by using a cascade of registers, each output sequence clocking the next register. The structure may be modified so that the output from stage $k-1$ both clocks stage i and is added to the output of stage k modulo 2. A diagram of a cascaded clock controlled shift register of height 3 is given in Fig. 43.5.

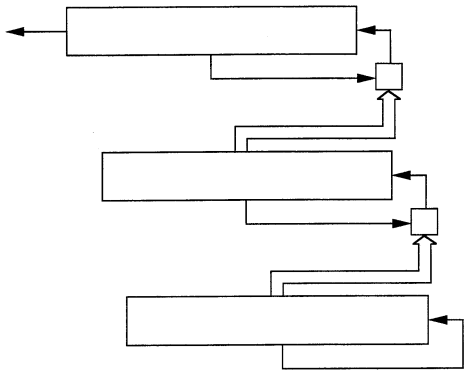


FIGURE 43.5 Cascaded clock controlled shift register of height = 3.

If 0,1 clocking is used (that is, a cascaded stop-and-go generator), each LFSR has maximal period and length n, and all LFSRs have distinct primitive connection polynomials, then the output period is $(2^n-1)^k$ and the linear span is $n(2^n-1)^{k-1}$. For arbitrary s, t clocking suppose the LFSRs have irreducible degree d connection polynomials and period p with $p^2 \not| (2^{p-1}-1)$. Then the output period is p^n and the output linear span is at least $d(p^n-1)/(p-1)$. The distribution of subsequences of any fixed length approaches the uniform distribution as k approaches infinity.

Correlation attacks may be used to recover state information about the last stage of a cascaded clock controlled shift register. Another attack based on iteratively reconstructing the states of the registers is possible in some circumstances [5]. The cryptanalyst is assumed to know the individual shift register sequences but not the correct phase shifts. She attempts to determine the phase shift of each stage in turn, starting with the last stage. In some cases if she guesses the phase of a given stage and reverses the register one step at a time, then the reconstructed output from the preceding stage eventually locks into the correct phase. This allows the reconstruction to be repeated at the preceding stage.

With a *self-clocking generator* a single LFSR is used to clock itself. The case when the clocking function satisfies $f(s_i) = d$ if $s_{i,1} = 0$ and $f(s_i) = k$ if $s_{i,1} = 1$ is called the $[d, k]$ self-decimation generator, to which we restrict our attention. We also assume the underlying LFSR has maximal period $2^n - 1$. The state graph of a $[d, k]$ self-decimation generator may not be purely cyclic. If $d \neq k$, then the (eventual) period is at most $(3/4)(2^n - 1)$. In case $\gcd(d, 2^n - 1) = 1$ and $2d \equiv k \bmod 2^n - 1$ or $2^{n-1}d \equiv k \bmod 2^n - 1$ the period is exactly $(2/3)(2^n - 1)$. The distribution of short subsequences is close to uniform. There is evidence that the linear span is at least 2^{n-1} but this has not been proved. The drawback to the $[d, k]$-self decimation generator is that each output bit reveals a state bit and reveals which state bit is revealed by the next output bit. One way of avoiding this is to use different state bits for output and clocking control.

The Shrinking Generator

A somewhat different type of clocking occurs in the *shrinking generator.* Again, we start with a pair of LFSRs, L_1 and L_2, with lengths n_1 and n_2 and output sequences **a** and **b**. At each stage, if $a_i = 1$, then b_i is output. Otherwise no bit is output. Thus the output **c** is a shrunken version of **b**: $c_j = b_{i_j}$ if i_j is the position of the jth 1 in **a**. If **a** and **b** are m-sequences and $\rho(\mathbf{a})$ and $\rho(\mathbf{b})$ are relatively prime, then **c** has period

$$\rho(\mathbf{c}) = \rho(\mathbf{b})2^{n_1-1} = \left(2^{n_2} - 1\right) 2^{n_1-1} .$$

The linear span of **c** satisfies

$$n_1 2^{n_2-2} < \lambda(\mathbf{c}) \leq n_1 2^{n_2-1} .$$

It can also be shown that the distribution on fixed length subsequences in **c** is close to uniform. One drawback is the irregularity of the output. A string of zeros in **a** leads to a delay in generating the next

bit. Buffering can be used to alleviate this problem. A variation called the *self shrinking generator,* where a single register clocks itself, has also been considered.

Cyclotomic Generator

Ding [9] proved a lower bound on the linear span of a sequence based only on its period. If n is an integer with prime factorization

$$n = \prod_{i=1}^{t} p_i^{e_i} ,$$

q is a prime power relatively prime to n, and $\mathbf{a}$ is a periodic sequence of period n over $GF(q)$, then

$$\lambda(\mathbf{a}) \geq \max \left\{ \text{ord}_{p_i}(q) : 1 \leq i \leq t \right\} .$$

The sphere complexity is similarly bounded. If $k < \min\{||\mathbf{a}||, n - ||\mathbf{a}||\}$, where $||\mathbf{a}||$ is the Hamming weight of a single period of $\mathbf{a}$, then

$$\text{SC}_k(\mathbf{a}) \geq \max \left\{ \text{ord}_{p_i}(q) : 1 \leq i \leq t \right\} .$$

In particular, if the period n is prime and q is a primitive element modulo n, then the linear span is at least $n - 1$, as is the sphere complexity for $k < \min\{||\mathbf{a}||, n - ||\mathbf{a}||\}$. When n is prime, various conditions guarantee that 2 is a primitive element modulo n. For example, (1) $n4t \pm 1$, t an odd prime; or (2) $n = t_1 t_2 \pm 1$, each t_i an odd prime and $2^{t_i} \not\equiv -1 \bmod n$.

Sequences satisfying the hypotheses of the above results can be produced by a class of generators called *cyclotomic generators* whose analysis is based on the theory of cyclotomic numbers. These generators use a base register that counts by ones modulo n. A function h is then applied to the value of the counter to produce an output bit. The simplest case is the cyclotomic generator of order $2k$, for which

$$h(i) = \left(i^{(n-1)/2k} \bmod n \right) \bmod 2 .$$

Here $(x \bmod n) \bmod 2$ means reduce x modulo n to a residue in the range 0 to $n - 1$, then take the parity. It can further be shown that the autocorrelations of these sequences are ideal. Several generalizations of this generator have also been considered by Ding [9].

RC4

RC4 is a byte-oriented stream cipher designed by Rivest for RSA Data Security, Inc. in 1987. It is intended for use in software. Its details were unpublished and proprietary until 1994, when they were anonymously leaked on the sci.crypt newsgroup. The state consists of two nonnegative integer variables x and y, each less than 256, and an array P of 256 bytes. P must contain a permutation of $\{0, 1, \cdots, 255\}$. A sequence of bytes is generated by the pseudocode in Fig. 43.6. All addition is modulo 256.

The key is an array of bytes, $K[0], \cdots, K[m - 1]$ for some m. This is used to initialize the state as follows: The variables x and y are set to 0 and the array P is set to $(0, 1, \cdots, 255)$. Then the RC4 algorithm in Fig. 43.6 is iterated 256 times, with line 3 replaced by

$$y \leftarrow y + P[x] + K[x \bmod m] ,$$

and with line 5 deleted. Finally, x and y are reset to 0. Very little is publicly known about the security of RC4. RSA Data Security, Inc. claims the algorithm is immune to linear and differential cryptanalysis, has no small cycles, and is highly nonlinear.

RC4(**a**)

```
1    while more bytes are needed
2        x ← x + 1
3        y ← y + P[x]
4        swap P[x] and P[y]
5        output P[P[x] + P[y]]
```

FIGURE 43.6 The RC4 stream cipher.

SEAL

Software-optimized encryption algorithm (SEAL) was proposed in 1993 by Rogaway and Coppersmith [29]. Since it is quite new, little is known about its cryptanalysis. SEAL is designed for use in software. It depends on a 32-bit architecture and uses eight 32-bit registers and about 3 kilobytes of cache. It is claimed to have a data rate ten to thirty times faster than DES, the most popular block cipher.

A basic design principle of SEAL is to generate a large table in a preprocessing stage. This stage is relatively slow but is intended to take place concurrently with the (generally slow) key exchange at the beginning of a session. In such a setting the preprocessing of SEAL incurs little extra cost. The table is generated by a complex mix of bit-wise ands, bit-wise ors, bit-wise exclusive-ors, bit-wise complements, concatenations, shifts, and additions modulo 2^{32}.

The input to SEAL is a 160 bit string a and a message size L. For each a, a function SEAL_a is determined from the set of positive integers to the set of infinite binary strings. Encryption is performed by computing the bit-wise exclusive or of the nth message and the first L bits of $\text{SEAL}_a(n)$. In practice only the first $128\lceil L/128\rceil$ bits of $\text{SEAL}_a(n)$ are generated. Again, the computation of $\text{SEAL}_a(n)$ is a complex mix of bit-wise ands, bit-wise ors, bit-wise exclusive-ors, bit-wise complements, concatenations, shifts, and additions modulo 2^{32}, plus table lookups using the tables generated in the preprocessing stage.

The assumed strength of SEAL depends on the fact that, if a is chosen uniformly at random, then $\text{SEAL}_a(n)$ is computationally indistinguishable from a random L-bit function of n.

Universal Security

In this section we consider the existence of universally secure generators from several theoretical viewpoints. We start with complexity theoretic considerations.

Blum–Micali Discrete Log Generator

Blum and Micali designed a generator whose security is based on the assumed hardness of the discrete log problem [3]. Suppose p is a prime number and g is a generator of $\mathbf{Z}_p^*$, the multiplicative group of integers modulo p. If $x \in \mathbf{Z}_p^*$, then the discrete logarithm of x with respect to g is the unique integer k such that $x = g^k \bmod p$. The discrete log problem is to find k given p, g, and x. This problem is widely believed to be computationally infeasible. If x is a quadratic residue modulo p, then its discrete log is of the form $2t$ with $0 \le t < (p-1)/2$. Its square roots are g^t and $g^{t+(p-1)/2}$. The latter is called the principle square root. The discrete log problem is polynomial time reducible to the problem of determining whether an element is a principle square root. Define the predicate

$$\psi_p(x) = \begin{cases} 1 & \text{if } x < (p-1)/2, \\ 0 & \text{otherwise}. \end{cases}$$

The state of a Blum–Micali generator is an element $x \in \mathbf{Z}_p^*$. At each stage the generator outputs $\psi(x)$ and changes state to $g^x \mod p$. It can be shown that if a certain assumption about the infeasibility of solving the discrete log problem with polynomial size circuits is true, then the Blum–Micali generator is perfect in the sense of the subsection "Unpredictability" in Section 43.2 when the initial state (seed) is chosen randomly.

Other Perfect Generators

The Blum–Micali generator is actually an example of a general construction that can be based on any one-way permutation f (in the case of the discrete log generator, $f(x) = g^x$) and binary predicate B on the domain of the predicate (in the case of the discrete log generator, $B(x) = \psi_p(x)$). There is a notion of an unpredictable predicate that guarantees that the resulting generator is perfect. The goal is to design a predicate whose unpredictability is equivalent to some problem that is infeasible. Such a predicate is called *hard core* for f. In reality we have no proof of infeasibility for the problems that arise this way, so belief in the security of such generators depends on belief in the infeasibility of the underlying problem (e.g., the discrete log problem).

One such construction is based on the infeasibility of inverting RSA ciphertexts. If $n = pq$ is a product of two primes and e is relatively prime to $(p-1)(q-1)$, then the domain is $\mathbf{Z}_n^*$. The permutation is $f(x) = x^e$ and the hard core predicate is the least significant bit. It can be shown that if RSA ciphertexts cannot be inverted in expected polynomial time, then the RSA generator is perfect.

Another such construction is based on the infeasibility of computing square roots modulo a product of two primes that are congruent to 3 modulo 4 [2]. This problem is known to be equivalent to factoring the modulus. Here the domain is the set of quadratic residues modulo an RSA modulus (i.e., a product of two large primes), the permutation is squaring, and the hard core predicate is the least significant bit. It can be shown that if it is impossible to compute modular square roots with polynomial size circuits on a polynomial fraction of the moduli n, correctly on all but a polynomial fraction of $\mathbf{Z}_n^*$, then the quadratic residue generator is perfect.

There have been several recent results that show that other bits (in addition to the least significant bit) are hard core, thus expanding the suite of perfect generators. For example, Näslund showed that any bit in a random linear function modulo a random prime is hard core for any one-way function.

Provable Security

Several attacks on stream ciphers synthesize a generator given a prefix of the keystream. The synthesized generator belongs to a specific class of generators $\mathcal{F}$. If enough bits are available, the attack should produce the smallest generator (in the sense of the size of the state space) in $\mathcal{F}$ that outputs the given sequence. The Berlekamp–Massey and the 2-adic rational approximation algorithms are examples of such attacks.

Klapper [19] studied the question of whether there exists a family of generators that resists all such attacks. For a family $\mathcal{F}$ and a sequence $\mathbf{a}$ the size of the smallest generator is denoted $\lambda_{\mathcal{F}}(\mathbf{a})$. Such an attack A is called *effective* if it runs in polynomial time and is successful whenever the number of bits available is at least a fixed polynomial in $\lambda_{\mathcal{F}}(\mathbf{a})$. The existence of an effective $\mathcal{F}$-synthesizing algorithm implies that $\lambda_{\mathcal{F}}(\mathbf{a})$ is a measure of security for stream ciphers, analogous to the linear span. By a diagonalization argument it can be shown that there exists a family of efficiently generated sequences S such that, for any effective algorithm synthesizing generators in a family $\mathcal{F}$, $\lambda_{\mathcal{F}}(\mathbf{a})$ grows superpolynomially in the log of the period of $\mathbf{a} \in S$. Various generalizations of this result are possible for weaker notions of security.

A quite different approach to provable security was taken by Maurer [25]. He considered stream ciphers based on the availability of a public global source of randomness. Maurer described such a *randomized stream cipher* that is perfectly secure with high probability. The proof of this result is based on Shannon's information theory. The sender and receiver need only share a short key, while the cryptanalyst must examine an infeasible number of random bits in order to attack the system. The drawback to this system is the difficulty in making a source of a large number of truly random bits publicly available.

43.4 Research Issues and Summary

In this article we provide a survey of techniques related to pseudorandom sequence generation and its role in cryptography. Stream ciphers, the resulting systems, are generally much faster than other cipher systems. Thus they are useful when high-speed secure communications is needed. Proving their security, however, is more difficult than that of many public key systems. Techniques for generating sequences that have cryptographically strong properties are presented, as well as techniques for cryptanalyzing sequences. A number of measures of randomness are also described.

A variety of sequence generators are presented. Many of these are based on modifications of linear feedback shift registers to eliminate vulnerability to Berlekamp–Massey type attacks. In most cases the best that can be said is that the generators resist certain known attacks and have various desirable randomness properties. In some cases the security of a generator is described in complexity theoretic (and in particular asymptotic) terms. Unfortunately such generators tend to be slow.

A few of the cryptanalytic techniques presented are general and can be tried on any stream cipher. Such techniques give rise to general measures of security. More often, cryptanalytic techniques are specific to generators or classes of generators. Nonetheless, they can sometimes be adapted to other classes of generators. Attempting the cryptanalysis of a generator is important for its certification as a secure system.

One focus of current research is the development of new techniques of keystream generation. There are no known provably secure efficient keystream generators, so their need persists. When each new technique of cryptanalysis is developed, new generators are needed that resist the new attack, as well as all old attacks. The difficulty faced by designers of keystream generators is that the presence of enough structure to prove that a generator resists known attacks and has other desirable properties often leads to the development of new cryptanalytic techniques to which the generator is vulnerable. The ideal would be to design a type of keystream generator that resists all computationally feasible cryptanalytic attacks, is efficient, and has an efficient description. This is a goal for security defined in various models.

Simultaneously, considerable research is focused on developing new cryptanalytic techniques. The challenge is to find exploitable structure in systems that are often designed to have little clear structure or to combine several types of apparently incompatible structure. The search for cryptanalytic tools may at first seem perverse and counter productive. However, when a system has weaknesses it is important that potential users be aware of this. These weaknesses may be known already to hostile cryptanalysts who are unlikely to advertise their knowledge.

43.5 Defining Terms

Autocorrelation: The number of positions where a periodic binary sequence agrees with a shift of itself minus the number of places where it disagrees.

Balance: The number of zeros minus the number of ones in a single period of a periodic binary sequence.

Clock-controlled generator: A keystream generator in which one LFSR is used to determine which output symbols of a second LFSR are used as the final output.

Correlation attack: A cryptanalytic attack on a keystream generator based on improving key search by exploiting correlations between output bits and initial state or seed bits.

Correlation immunity: A measure of resistance of a nonlinear combiner to correlation attacks.

Cryptographically strong pseudorandom bit generator (CSPRB): An efficient family of keystream generators such that it is infeasible to predict bits with probability significantly greater than one half.

Entropy: A formal measure of the uncertainty in a random variable.

Feedback with carry shift register (FCSR): A feedback register similar to an LFSR, but where the addition in the feedback function is performed with carry, the carry being retained for the next stage in extra memory.

Information theory: The mathematical study of the information content of random variables.

Keystream generator: A device or algorithm for generating a pseudorandom sequence.

Linear feedback shift register (LFSR): A device for generating infinite periodic sequences. The state updates by shifting its state vector one position and generating a new state symbol as a linear function of the old state vector. The output is the symbol shifted out of the register.

Linear span: Also called the linear complexity, the length of the shortest linear feedback shift register that outputs a given sequence.

m-Sequence: A maximal period linear feedback shift register sequence.

Next bit test: A family of circuits used to predict the next bit of a sequence given a prefix of the sequence.

Nonlinear filter generator: A linear feedback shift register modified so the output is computed as a nonlinear function of the state.

Nonlinear combiner: A keystream generator in which the outputs of several LFSRs are combined by a nonlinear function. The combining function may have a small amount of memory.

One-time pad: A stream cipher in which the key bits are chosen independently at random.

Pseudorandom sequence: An infinite periodic sequence that behaves like a truly random sequence with respect to various measures of randomness.

Stream cipher: A private key cryptosystem in which the key is added to the plaintext symbol by symbol modulo the size of the plaintext alphabet.

Strict avalanche condition (SAC): A nonlinearity condition. A function satisfies SAC if changing any single bit results in a function uncorrelated with f.

References

[1] Beth, T. and Piper, F., The stop-and-go generator. In *Advances in Cryptology—Eurocrypt '84,* T. Beth, N. Cot, and I. Ingemarsson, Eds., *Lecture Notes in Computer Science,* Vol. 209, 88–92. Springer-Verlag, Berlin, 1985.

[2] Blum, L., Blum, M., and Shub, M., A simple unpredictable pseudo-random number generator. *Siam J. Comput.,* (15), 364–383, 1986.

[3] Blum, M. and Micali, S., How to generate cryptographically strong sequences of pseudorandom bits. *SIAM J. Comput.,* (13), 850–864, 1984.

[4] Boyar, J., Inferring sequences produced by a linear congruential generator missing low-order bits. *J. Crypt.,* (1), 177–184, 1989.

[5] Chambers, W. and Gollmann, D., Lock-in effect in cascades of clock-controlled shift-registers. In *Advances in Cryptology—Eurocrypt '88,* G. Günther, Ed., *Lecture Notes in Computer Science,* Vol. 330, 331–344, Springer Verlag, Berlin, 1988.

[6] Chan, A., Goresky, M., and Klapper, A., On the linear complexity of feedback registers. *IEEE Trans. Inf. Thy.,* (36), 640-645, 1990.

[7] Coppersmith, D., Krawczyk, H., and Mansour, Y., The shrinking generator. In *Advances in Cryptology—Crypto '93,* D. Stinson, Ed., *Lecture Notes in Computer Science,* Vol. 773, 22–39, Springer-Verlag, New York, 1994.

[8] Ding, C., The differential cryptanalysis and design of natural stream ciphers. In *Fast Software Encryption: Proceedings of 1993 Cambridge Security Workshop,* R. Anderson, Ed., *Lecture Notes in Computer Science,* Vol. 809, 101–120, Springer-Verlag, Berlin, 1994.

[9] Ding, C., Binary cyclotomic generators. In *Fast Software Encryption: Proceedings of 1994 Leuven Security Workshop,* B. Perneel, Ed., *Lecture Notes in Computer Science,* Vol. 1008, 29–60, Springer-Verlag, Berlin, 1995.
[10] Ding, C., Xiao, G., and Shan, W., *The Stability Theory of Stream Ciphers, Lecture Notes in Computer Science,* Vol. 561, Springer-Verlag, Berlin, 1991.
[11] Golić, J., Fast correlation attacks on irregularly clocked shift registers. In *Advances in Cryptology—Eurocrypt '95,* J. Quisqater, Ed., *Lecture Notes in Computer Science,* Vol. 921, 248–262, Springer-Verlag, Berlin, 1995.
[12] Golić, J., Correlation properties of a general binary combiner with memory. *J. Crypt.,* (9), 111–126, 1996.
[13] Gollman, D. and Chambers, W., Clock-controlled shift registers: A review. *IEEE J. Selected Areas Commun.,* (7), 525–533, 1989.
[14] Golomb, S., *Shift Register Sequences,* Aegean Park Press, Laguna Hills, CA, 1982.
[15] Groth, E., Generation of binary sequences with controllable complexity. *IEEE Trans. Info. Thy.,* (IT-17), 288–296, 1971.
[16] Herlestam, T., On function of linear shift register sequences. In *Advances in Cryptology—Eurocrypt '85,* E. Pichler, Ed., *Lecture Notes in Computer Science,* Vol. 219, 119–129, Springer-Verlag, Berlin, 1985.
[17] Key, E., An analysis of the structure and complexity of nonlinear binary sequence generators. *IEEE Trans. Info. Thy.,* (IT-22), 732–736, 1976.
[18] Klapper, A., The vulnerability of geometric sequences based on fields of odd characteristic. *J. Crypt.,* (7), 33–51, 1994.
[19] Klapper, A., On the existence of secure feedback registers. In *Advances in Cryptology—Eurocrypt '96,* U. Maurer, Ed., *Lecture Notes in Computer Science,* Vol. 1070, 256–267, Springer-Verlag, Berlin, 1996.
[20] Klapper, A. and Goresky, M., Feedback shift registers, 2-adic span, and combiners with memory. *J. Crypt.,* (10), 111–147, 1997.
[21] Kumar, V. and Scholtz, R., Bounds on the linear span of bent sequences. *IEEE Trans. Info. Thy.,* (IT-29), 854–862, 1983.
[22] Lidl, R. and Niederreiter, H., *Finite Fields: Encyclopedia of Mathematics,* Vol. 20, Cambridge University Press, Cambridge, 1983.
[23] McEliece, R., *Finite Fields for Computer Scientists and Engineers,* Kluwer Academic Publishers, Boston, 1987.
[24] Massey, J., Shift register sequences and BCH decoding. *IEEE Trans. Info. Thy.,* (IT-15), 122–127, 1969.
[25] Maurer, U., A provably-secure strongly-randomized cipher. In *Advances in Cryptology—Crypto '90,* S. Vanstone, Ed., *Lecture Notes in Computer Science,* Vol. 473, 361–373, Springer-Verlag, New York, 1991.
[26] Meier, W. and Staffelbach, O., Correlation properties of combiners with memory in stream ciphers, *J. Crypt.,* (5), 67–86, 1992.
[27] Menezes, A., van Oorschot, P., and Vanstone, S., *CRC Handbook of Applied Cryptography,* CRC Press, Boca Raton, FL, 1996.
[28] Preneel, B., Van Leekwijk, W., Van Linden, L., Govaerts, R., and Vandewalle, J., Boolean functions satisfying higher order propagation criteria. In *Advances in Cryptology—Eurocrypt '90,* I. Damgård, Ed., *Lecture Notes in Computer Science,* Vol. 473, 161–173, Springer-Verlag, Berlin, 1990.
[29] Rogaway, P. and Coppersmith, D., A software optimized encryption algorithm. In *Fast Software Encryption: Proceedings of 1993 Cambridge Security Workshop,* R. Anderson, Ed., *Lecture Notes in Computer Science,* Vol. 809, 56–63, Springer-Verlag, Berlin, 1993.
[30] Rothaus, O., On bent functions. *J. Comb. Thy. (A),* (20), 300–305, 1976.

[31] Rueppel, R., *Analysis and Design of Stream Ciphers,* Springer-Verlag, New York, 1986.
[32] Rueppel, R. and Staffelbach, O., Products of sequences with maximum linear complexity. *IEEE Trans. Info. Thy.,* (IT-33), 124–131, 1987.
[33] Schneier, B., *Applied Cryptography,* John Wiley & Sons, New York, 1996.
[34] Siegenthaler, T., Correlation-immunity of nonlinear combining functions for cryptographic applications. *IEEE Trans. Info. Thy.,* (IT-30), 776–780, 1984.
[35] Siegenthaler, T., Cryptanalyst's representation of nonlinearly filtered ml-sequences. In *Advances in Cryptology—Eurocrypt'85,* E. Pichler, Ed., *Lecture Notes in Computer Science,* Vol. 219, 103–110, Springer-Verlag, Berlin, 1986.
[36] Simmons, G., Ed., *Contemporary Cryptography,* IEEE Press, New York, 1992.
[37] Shannon, C., Communication theory of secrecy systems. *Bell Syst. Tech. J.,* (28), 656–715, 1949.
[38] Welsh, D., *Codes and Cryptography,* Clarendon Press, Oxford, U.K., 1988.
[39] Yao, A., Theory and applications of trapdoor functions. In *Proceedings, 23rd IEEE Symposium on Foundations of Computer Science, 1982,* 80–91, *IEEE Computer Society Press,* Los Alamitos, CA, 1982.

Further Information

Research on pseudorandom sequences and stream ciphers is extensively published in the *IEEE Transactions on Information Theory* and the *Journal of Cryptology.*

Many major results appear initially in the proceedings of the annual conferences Crypto, held in Santa Barbara, California, Eurocrypt, held in various countries in Europe, and Asiacrypt, held in various countries in Asia and Australia. The proceedings of these conferences appear in the Springer-Verlag *Lecture Notes in Computer Science* series. Crypto and Eurocrypt are held under the auspices of the IACR, the International Association for Cryptologic Research.

Many papers have also appeared at specialty workshops such as "Fast Software Encryption", whose proceedings also appear in the Springer-Verlag *Lecture Notes in Computer Science* series.

Information about the IACR and about many conferences on cryptography can be found at the IACR web-site, http://www.iacr.org/~iacr/.

The chapter by Rueppel in Simmon's *Contemporary Cryptography* [36] provides a more detailed summary of the state of the art as of 1990.

Menezes, van Oorschot, and Vanstone's *CRC Handbook of Applied Cryptograpy* [27] provides a thorough summary as of 1996.

A thorough treatment of the basic analysis of linear feedback shift registers can be found in Golomb's classic *Shift Register Sequences* [14].

Many aspects of shift registers, stream ciphers, and linear span are treated in Rueppel's *Analysis and Design of Stream Ciphers* [31].

Many of the more practical aspects of stream ciphers, as well as a thorough bibliography, can be found in Schneier's *Applied Cryptography* [33].

Good sources for the mathematical foundations of the study of pseudorandom sequences are Lidl and Niederreiter's *Finite Fields* [22] and McEliece's *Finite Fields for Computer Scientists and Engineers* [23].

44

Electronic Cash

Stefan Brands
Brands Technologies

44.1 Introduction

Soon you may find yourself e-mailing bytes representing your money to service providers across the ocean, using a smart card or a handheld computer to pay at your local grocery, and making backup copies to protect your money against hard-disk crashes. Now that technological advances in chip manufacturing have brought vast computing powers within everyone's reach, traditional cash is about to be replaced for an invention based on modern cryptography: electronic cash.

This chapter provides an overview of the state-of-the-art techniques for designing electronic cash systems, without going into great technical detail. We begin by quickly reviewing today's payment forms and analyzing their inherent shortcomings, to motivate the subsequent description of properties that are generally deemed desirable for an electronic equivalent of traditional money.

Traditional Cash Payments

Traditional cash is a bearer instrument that can be used spontaneously and instantaneously, to make payments from person to person without the involvement of a bank. It is the preferred method for low- and medium-value purchases, which make up the bulk of our everyday transactions. Cash payments also

offer privacy, because they are not normally traceable by a third party. Together these factors account for the wide acceptability of traditional cash.

Traditional cash also has several shortcomings. Creating cash that is hard to forge, transporting cash from one place to another, protecting cash transport and storage, and replacing worn out coins and bank notes all make traditional cash very costly to handle for banks. Bank notes are easily destroyed and can be forged using sophisticated color copier machines, coins are too heavy to carry around in bulk, and both are easily lost or stolen. Cash comes in fixed denominations and so typically many coins or bank notes need to change hands to pay a single amount. Because coins and bank notes reveal neither the payer's nor the payee's identity, cash is the preferred method of payment for money laundering, bribery, and extortion. Cash, moreover, can be passed on many times without the need for settlement by a bank, further conflicting with the desire of governmental organizations to trace criminal money. Another shortcoming of traditional cash, one that has become particularly urgent in recent years, is the inherent requirement for physical proximity of payer and payee; coins and bank notes cannot be used for payments over the phone or the Internet.

Payments by Instruction

Some of the problems of traditional cash have successfully been addressed over the past few decades with the introduction of checks, debit cards, and credit cards. Instead of value itself, the payer transfers to the payee an *instruction,* directing the payer's bank to transfer a specified amount. After reception of the transferred instruction from the payee's bank, the payer's bank moves the value from source to destination account, both of which are specified by the payment instruction. Because the actual value resides at all times within the banks, the risks of theft and loss are largely overcome. Checks moreover can be mailed by post, and credit cards can be used over the phone or even the Internet.

Payments by instruction also bring forth several problems of their own, mainly originating from the problem of ensuring their authenticity. Handwritten signatures are easily forged and magnetic stripes copied, PINs can be learned through fake point-of-sale terminals, and specified amounts can sometimes be modified by the payee. This can be addressed in part by upgrading to cryptographic authentication methods in combination with chip cards. Replacing magnetic stripe cards by chip cards with protected memory renders unauthorized card duplication almost infeasible, and fake terminal attacks can be overcome by using cards with a microprocessor that enable a PIN or a biometric to be entered into the card itself. The microprocessor can furthermore be used to compute message authentication codes or, in case it has sufficient processing power, digital signatures; these are much harder to forge than handwritten signatures, and moreover can be verified rapidly and with one hundred percent accuracy by processors in point-of-sale modules. The use of secret keys for cryptographic authentication also enables message encryption for protection against wire-tappers, and hence can protect against wire-tapping of credit card numbers sent across the Internet.

A problem that cannot be addressed by authentication techniques is the need for *on-line* verification of debit and credit card payments; for the former it is inherent, and in case of the latter it is needed (at least as a rule) to protect against overdrawing of accounts. On-line processing makes handling costs significantly higher than for traditional cash and is expensive for payees. A system that requires on-line payment verification moreover can suffer severely from a network breakdown or overload; this can lead to serious delays, and merchants may even have to refuse all payments for a while. Add to this the fact that payees typically need to satisfy stringent registration criteria (such as having a store front), and need to acquire expensive tamper-resistant modules, and it is easy to understand why the acceptability of credit and debit cards is much lower than of cash payments. Only guaranteed checks can be verified *off-line* and without special equipment, but these are not readily issued by banks, do not allow instantaneous payment, and are time-consuming to obtain, verify, process, and redeem.

Another inherent problem is that all payments are effortlessly *traceable* by the bank that performs the actual transfer of value, from source to destination account. This enables intrusive profiling of spending

behavior and, by inference drawing on mathematical techniques such as data mining, all manner of other personal information characteristics. This can lead to junk mail, unjustified or wrong assumptions about personal behavior and other characteristics, and outright discrimination. In the wrong hands, detailed information about spending and inferred habits is a valuable tool for victimizing people, be it by criminals to select their targets, or by political aggressors to track down or lock out opponents. Data protection laws can offer only limited protection against abuse of personal information, and cannot protect against the harmful consequences that may result from payment profiling errors.

Electronic Cash Properties

By combining the benefits of traditional cash with those of payments by instruction, while at the same time circumventing the shortcomings of each, a list of desirable properties for an electronic equivalent of traditional cash can be composed.

As with traditional cash, electronic cash should have high acceptability, should be cost-effective for low-value purchases, and should be suitable for payment from person to person. This means that payees should not need costly tamper-resistant modules provided by financial institutions, nor should they need to meet stringent registration criteria.

It should also be possible to verify payments off-line. Off-line payment capability is a prerequisite for platform independence and hence for cross-platform portability, a key feature of an open system. While for a platform such as the Internet, on-line verification need not become a bottleneck until a substantial number of users are active, for most other platforms on-line verification is not cost-effective and can lead to unacceptable delays. Systems requiring on-line payment verification go against the philosophy of cash payment, and cannot be considered true electronic cash.

As with payments by instruction, electronic cash should offer storage and transportation convenience, while protecting users against loss, theft, and accidental destruction. Physical proximity of payer and payee should not be needed, so that electronic cash payments can be made over the phone or the Internet. Furthermore, all manner of electronic cash handling should be dealt with electronically, to ensure cost-effectiveness, instantaneousness, and accuracy; this includes not only creating, issuing, spending, verifying, counting, transporting and revoking of electronic cash, but also fraud detecting and tracing.

The extent to which an electronic cash system offers privacy of payment may well be the decisive factor in its ultimate acceptance in an open consumer market. Three forms of privacy of payment can be distinguished. The first is known as *confidentiality,* and refers to the ability of account holders to hide transaction details from wire-tappers, and to hide information such as purchase descriptions from the bank. Payment confidentiality can be achieved fairly straightforwardly, by encrypting all sensitive data.

The second form of privacy is the ability to hide who is transacting with who, how many times, and so on. In case of Internet payments and the like, traffic analysis may easily reveal such information, regardless of whether messages are encrypted, unless special measures are taken. We do not consider techniques to hinder traffic analysis in this chapter, because they are platform dependent and not necessarily based on cryptography; for example, in the case of Internet payments one can use IP spoofing or anonymous remailers, and in other circumstances traffic analysis may be inherently difficult.

The third form of privacy is payment **untraceability.** Without the use of proper cryptographic design techniques, every electronic cash payment would cause a unique transaction identifier to end up in the computer files of the bank, and electronic cash would be no better in this respect than payments by instruction. Apart from off-line payment capability, it is the possibility to have payment untraceability that sets electronic cash apart from instruction-based payment forms.

In addition to the objective of untraceability of payments, any acceptable design for an electronic cash system should strike a balance between the perfect two-sided untraceability of traditional cash and the governmental desire to discourage criminal uses, such as money laundering, bribery, and extortion.

Achieving many or all of the properties listed above, without giving in on system security, may seem a daunting if not impossible task. However, as we will see in this chapter, with the right choice of

cryptographic techniques small miracles can be accomplished. We will successively examine payment authentication techniques, ways to represent electronic cash, techniques for electronic cash transfer, techniques to guarantee security even when tamper-resistance is compromised, security measures for account holders, and techniques to achieve payment privacy. This is followed by a detailed description of an example electronic cash system, designed on the basis of the considerations and insights of the preceding sections. At the end of this chapter a summary of the material presented is provided.

44.2 Preliminaries

Modeling Electronic Cash

The flow of electronic cash resembles that of traditional cash, but the minting need not necessarily take place by a central party. When electronic cash is backed by some commodity of generally trusted value, such as gold or traditional cash, different banks can be allowed to mint their own electronic cash. Electronic cash then merely serves as a sophisticated front for exchanging "real" money, and needs to be purchased from a minting bank, like any other good or service.

Each participant in an electronic cash system is represented by at least one computing device. When an account holder of some bank wants to withdraw some electronic cash minted by his bank, his computing device engages in an execution of a *withdrawal protocol* with a computing device of the bank. At the end of the protocol execution, the computing device of the account holder holds an amount of electronic cash, represented in either one of two forms discussed later. The bank has charged the account holder, by taking the equivalent amount of "traditional" money out of his bank account and moving it into a *float account*; the electronic cash is prepaid.

To spend electronic cash at a payee that accepts electronic cash issued by his own bank, the account holder interfaces his computing device to one of the payee, and the two computing devices perform an execution of a *payment protocol.* As a result, the representation of the electronic cash amount held by the account holder's device is adjusted, to reflect the new amount. Since the payment is off-line, the payee's computing device should correspondingly represent the received amount in some form. As we will see later on, the payee's representation of electronic cash need not necessarily be the same as that for the payer, and in fact it preferably is not.

Ultimately a party that holds electronic cash issued by the issuing bank needs to sell it back to that bank, to prevent losing money. At the very least, redemption is needed before electronic cash expires, but depending on system design it may also be needed because electronic cash received in a payment cannot be used for subsequent payments, or only up to a predetermined number of times. To this end the party interfaces his computing device to one of his own bank, and the two devices perform an execution of a *deposit protocol.* As a result, the account of the party depositing the electronic cash is credited by his bank with the equivalent amount of money. In case the crediting bank is the same as the issuing bank, it simply takes the money out of its float account. If, however, the crediting bank is the same as the issuing bank, it needs to settle with the issuing bank. Because the design of a suitable clearing infrastructure, no matter how important, is only remotely related to the cryptographic design of electronic cash systems, we do not discuss this further here.

For design purposes, an electronic cash system can conveniently be modeled as consisting of a single bank, one or more payers and one or more payees, each holding one or more computing devices. The core of a cryptographic design of an electronic cash system then consists of the protocols for withdrawal, payment and deposit. Depending on the functionality of the cash system, other cryptographic protocols may be needed.

The computing equipment of the bank can be viewed as a single device, so that one can speak of "the" computing device of the bank. Furthermore, a distinction can be made between *paying devices* and *receiving devices,* which is especially convenient when studying the security of value-transferring protocols.

Depending on system design, a device can be either a paying device or a receiving device, or both. For example, paying devices serve as receiving devices in the withdrawal protocol.

Regardless of the manner in which electronic cash held by a paying device is represented, it is clear that at least part of the device must be *tamper-resistant,* in order to prevent double-spending of electronic cash. Namely, if an attacker can determine the internal state of a paying device, then state freezing or copying enables the same electronic cash to be spent over and over again. This fraud can be detected, if at all, only once the forged cash is deposited to the bank.

Since tamper-resistant paying devices are typically smart cards, their microprocessors are preferably of low complexity and small size. Chip size increases with storage space, and the ability to rapidly perform public-key cryptographic multiplications requires a special cryptographic co-processor. Increased chip size and complexity contribute significantly to manufacturing cost and negatively affect chip reliability and durability. In "Guaranteeing Your Own Privacy" we will see how paying devices can be used in combination with other computing devices, such as a handheld device when paying in the local mall or a desktop computer when making Internet payments. Such an "interposed" computer of the account holder need not be tamper-resistant, can take over the greater part of the computational and storage burden of the paying device, and can offer many security and convenience advantages to its holder.

Authentication Techniques

Of utmost importance for the security of any electronic cash system is that forgery of electronic cash be infeasible, regardless of the form in which it is represented. This implies that receiving devices must be able to distinguish paying devices from attackers who try to pass for paying devices. To prove their authenticity, paying devices necessarily need to be equipped by the bank with a secret key. Correspondingly, receiving devices must be able to recognize whether they are communicating with a device holding a secret key installed by the bank. This requires a secure authentication mechanism.

An easy way for a receiving device to verify the authenticity of a paying device is to require the paying device to reveal its secret key. This, however, enables a wire-tapper to learn the secret key of the paying device, and to subsequently pass for it. More generally, a secure authentication protocol should resist a **replay attack.** This is an attack in which a wire-tapped *transcript* of an execution of an authentication protocol is reused by the attacker in order to pass for a paying device. **Static authentication,** in which the evidence provided to prove authenticity is always the same, does not offer adequate security.

Prevention of replay requires **dynamic authentication.** The key observation is that instead of revealing a secret key, it suffices to prove knowledge of the key, without revealing it. To achieve this seemingly contradictory task, the paying device must perform a computation that can be performed in a reasonable time span, if at all, only when it knows the secret key. The outcome of the computation must be verifiable by the receiving device, so that it can conclude that the outcome must have been generated by a paying device. To prevent a replay attack, the receiving device must ensure that a different computation is performed each time.

Protocols for dynamic authentication are commonly called *challenge–response protocols,* because the task that the receiving device wants to see performed can be regarded as a *challenge* to the paying device, and the result of the computation as the *response.* When designing a challenge–response protocol, care must be taken that wire-tappers cannot (feasibly) compute the secret key by analyzing transcripts of protocol executions, or by selecting challenges by themselves in some clever way.

Techniques for secure challenge–response authentication vary widely. An important distinction is between *conventional* dynamic authentication and dynamic authentication based on *public-key cryptography.* The following overview is focused especially on the applicability to electronic cash design.

Conventional Dynamic Authentication

In case the receiving device already knows the secret key of the paying device, it can verify a response by computing it by itself, and verifying for equality. The result of a computation that is based on a secret key and a message, and that can be verified only by using the same secret key, is called a MAC (message authentication code). Secure MACs are those that resist cryptanalysis even against active attackers, who can request and cryptanalyze arbitrarily many MACs for self-chosen messages. Secure MACs trivially enable the construction of secure challenge–response protocols: to this end the receiving device requests the paying device to compute a MAC for a challenge (and possibly such information as a device ID number). This is also known as symmetric authentication.

The challenge can be chosen either at random from a large domain, possibly by using a pseudo-random number generator, by concatenating a unique ID number of the receiving device to a sequence number (maintained by the receiving device and incremented upon each execution of the authentication protocol), or by using a sufficiently accurate estimate of the receiver's local time and date. In the case of random numbers an attacker who has learned MACs for many messages has negligible probability that a receiving device subsequently requests a MAC for a message seen earlier; in the case of sequence numbers and time/date estimates, challenges are guaranteed to differ with each protocol execution, but they can be anticipated and may conceivably be of greater value for a cryptanalytic attack.

Because receiving devices must know, or at least must be able to generate, the secret keys of the paying devices with which they need to be able to conduct transactions, receiving devices must be tamper-resistant as well. Therefore they must be issued by or on behalf of the bank, which takes care of installing the secret keys into all paying and receiving devices. When multiple banks each issue their own electronic cash, payees need different receiving devices for each bank. (In practice, this can take the form of different tamper-resistant chips plugged into a single standardized receiving module.)

The particular manner in which the secret keys are installed is of great importance to the system security. Three basic approaches are known.

The System-Wide Secret Key With this method, paying and receiving devices all hold the same random secret key, generated and installed by the bank. A major drawback is that an attacker needs merely be able to extract the secret key of any device, by compromising its tamper-resistance, in order to pass for any paying device.

Diversified Keys This method is similar to the preceding, with the difference that each paying device holds a unique secret key. Each receiving device must be able to recognize the secret keys of all paying devices. Since storing all the secret keys is inefficient, and adding new keys each time new paying devices are introduced is cumbersome, each receiving device stores a master key, generated at random and installed by the bank. The secret key of a paying device is computed as a function of the master key and a unique ID number of the paying device, and is called a **diversified key.** To ensure that the master key cannot be computed from a device secret key, the function must at least be one-way. In practice often a DES-based one-way function is used.

In case a secret key of a paying device has been extracted by an attacker, and used fraudulently, once the fraud is detected the compromised device can be traced and subsequently blacklisted. A weak point is that the master key is present in any receiving device, and extraction enables an attacker to pass for a paying device with an arbitrary ID number.

Certification of Diversified Keys A security improvement over the previous method can be achieved by *certification* of paying device ID numbers. To this end the bank stores into each paying device not only an ID number, but also its own digital signature on that ID number (and possibly also information such as an expiration date). In the payment protocol, the paying device transfers its ID number and the digital signature to the receiving device, which establishes the validity of the ID number by applying the public key of the bank to the digital signature. Message authentication then proceeds as with the preceding method.

The advantage over the diversified key method is that an attacker who knows the master key cannot pass for an arbitrary paying device, since he cannot forge signatures of the bank. Hence the attacker can pass only for a paying device for which he has learned the ID number and the digital signature, which conceivably makes it easier for the bank to trace the fraud.

Dynamic Authentication Based on Public-Key Cryptography

The vulnerability that is caused by the omnipresence of a master key can be removed by using *asymmetric* instead of symmetric authentication. As before, to prove its authenticity to a receiving device, a paying device computes a response based on its secret key and a challenge; but this time the receiving device verifies the result by applying the corresponding *public* key of the paying device.

As with the method of the system-wide secret key, it is not acceptable that all paying devices use the same secret key, and storing the public keys of paying devices into receiving devices is problematic for the same reasons as with the diversified key method. Instead, paying devices should provide their public keys to receiving devices during payment. Receiving devices must then be able to verify the validity of public keys, to which end the bank must certify the public keys of all paying devices. MAC certification voids the security benefits of public-key cryptographic authentication, because the bank's authentication key must be known to all receiving devices, so that only digital signatures are acceptable for certification. This also has the advantage that a single receiving device can be used to receive electronic cash issued by different banks.

Two basic public-key cryptographic techniques are known to design secure challenge–response protocols, *zero-knowledge authentication* and *digital signatures.* Zero-knowledge techniques can be used to design authentication protocols for which active attackers, who are allowed to cryptanalyze many protocol transcripts for self-chosen challenges, provably cannot learn any useful information beyond the public key.

In a secure digital signature scheme, it is infeasible for active attackers to subsequently forge a new (message, signature) pair. Hence a secure method of digital signing can be used effectively as a proof of knowledge of the secret key used for signing, without revealing "useful" information to attackers. A drawback is that the resistance against cryptanalytic attacks is harder to prove than for zero-knowledge authentication. On the other hand, a very important advantage of digital signatures over zero-knowledge authentication, as well as over conventional dynamic authentication, is that a digital signature constitutes an unforgeable transcript of a proof of knowledge of the secret key. It also demonstrates that any other information included in the challenge message, such as an amount, has been agreed to by the signer. A transcript of a protocol execution can therefore be used by a receiving device to demonstrate to the bank that a transaction has taken place; this does not require trust of the verifier in the tamper-resistance of the receiving device, and in fact receiving devices need not be tamper-resistant at all.

Many digital signature schemes, such as the RSA scheme in [35] and the Schnorr scheme in [36], are based on number-theoretic primitives. One-way functions built from block-ciphers (such as DES) can be processed two to four orders of magnitude more efficiently, and are of significant interest for electronic cash design because they allow implementation on paying devices with low-cost microprocessors. For this reason, we will now review several of these efficient schemes.

Lamport Signatures The first signature method we consider is the Lamport signature scheme [29]. The public key of a paying device is a set of $2k$ numbers, for an appropriate security parameter k, of the following form:

$$(f(s_{01}), f(s_{11})), (f(s_{02}), f(s_{12})), \ldots, (f(s_{0k}), f(s_{1k})), \tag{44.1}$$

and the secret key consists of the $2k$ preimages s_{ij}. The function $f(\cdot)$ is a one-way function that can be evaluated rapidly; an example is the function that assigns to a 55-bit input x the DES encryption of a fixed message, computed using the 56-bit key that is obtained from concatenating 0 to x. For storage efficiency,

the s_{ij} can be generated in a pseudo-random manner from a single secret, for example by applying a one-way function to i, j, and a randomly generated seed that is kept secret by the paying device.

To compute a Lamport signature on a message m of binary length k, the paying device sends to the receiving device a certificate of the bank on the public key. In addition, it releases, for each bit m_i of m, either $(s_{0i}, f(s_{1i}))$ or $(f(s_{0i}), s_{1i})$, depending on whether m_i is zero or one.

With this method, the binary length of a signature is $2k$ times the binary length of preimages of $f(\cdot)$, assuming inputs and outputs have the same length. In a practical implementation this typically exceeds storage requirements for, say, RSA signatures by an order of magnitude. As noted by Lamport, an alternative way to compute signatures is for the paying device to release a distinct subset of preimages for each message: this enables the signing of messages almost twice as long, using the same key set-up. Another method to achieve the same efficiency improvement is due to [32], in which it is proposed to expand all k-bit messages into codewords, by appending a count of the number of zero-bits; a message can then be signed securely by using a public key consisting of $k + \lceil \log_2 k \rceil$ instead of $2k$ ordered images of secret preimages.

Matrix-Based Signatures A further significant improvement in signature storage can be obtained by an improvement of Merkle [31]. In this improved method a secret key for a paying device consists of a matrix of r rows by $2c$ columns, which for explanatory purposes is best thought of as two matrices of r rows by c columns each, referred to as a "message" matrix and a "control" matrix, respectively. A key is generated by filling both matrices in a special way. To fill matrix j, for $j \in \{1, 2\}$, c random numbers, $s_{1j}, \ldots, s_{cj}$ are generated, and these are used to fill out the rth matrix row. The rest of the matrix is filled out row by row by applying the hash function $f(\cdot)$, as follows:

$$\begin{pmatrix} f^{r-1}(s_{1j}) & f^{r-1}(s_{2j}) & \ldots & f^{r-1}(s_{cj}) \\ \vdots & \vdots & \vdots & \vdots \\ f(s_{1j}) & f(s_{2j}) & \ldots & f(s_{cj}) \\ s_{1j} & s_{2j} & \ldots & s_{cj} \end{pmatrix} \tag{44.2}$$

The top rows of the two matrices are fed into a one-way hash function, referred to as a *compression* function, and the result is the public key for the paying device.

Using this key pair set-up, the paying device can sign any one of up to r^c messages. To sign a message of $\lfloor \log_2 r^c \rfloor$ bits, the message is first expanded into r-ary representation. Denoting the r-ary representation of m by $m_1 m_2 \cdots m_c$, the paying device releases $f^{r-1-m_i}(s_{i1})$ from the message matrix and $f^{m_i}(s_{i2})$ from the control matrix, for each $i \in \{1, \ldots, c\}$. As an example, consider $r = 4$ and $c = 3$. To sign the 6-bit message 010011, with 4-ary expansion 103, the paying device releases the numbers marked by the boxes:

$$\overbrace{\begin{pmatrix} f^3(s_{11}) & \boxed{f^3(s_{21})} & f^3(s_{31}) \\ \boxed{f^2(s_{11})} & f^2(s_{21}) & f^2(s_{31}) \\ f(s_{11}) & f(s_{21}) & f(s_{31}) \\ s_{11} & s_{21} & \boxed{s_{31}} \end{pmatrix}}^{\text{Message matrix}} \quad \overbrace{\begin{pmatrix} f^3(s_{12}) & f^3(s_{22}) & \boxed{f^3(s_{32})} \\ f^2(s_{12}) & f^2(s_{22}) & f^2(s_{32}) \\ \boxed{f(s_{12})} & f(s_{22}) & f(s_{32}) \\ s_{12} & \boxed{s_{22}} & s_{32} \end{pmatrix}}^{\text{Control matrix}} \tag{44.3}$$

To verify the signature, the receiving device computes the top row of each of the two matrices, by applying the appropriate number of one-way function applications to the matrix entries provided. By applying the compression function to the two computed top rows, it can then check whether the result matches the public key of the paying device.

It is important to note the need for the control matrix. If only the message matrix were used, then wire-tapping of a device signature for a message would enable signature forgery for any message that has digits in its r-ary expansion that all are no greater than those in the same position in the r-ary expansion of the first message. With the control matrix, in effect an additional "control" message has to be signed, for which each digit is equal to the additive inverse of the message. Releasing values higher up in the message

matrix thus requires releasing values moving downward in the control matrix, which requires the ability to invert $f(\cdot)$.

Storage of the secret key of the signer can be compressed using the same technique as for Lamport signatures. In particular, in practice the matrix rows may be generated from a short random seed in pseudo-random manner. A further improvement can be achieved by reducing the number of columns needed for the control matrix; instead of using one "control digit" per message digit, one can get away with logarithmically many control digits (and hence logarithmically many columns for the control matrix) by summing the digits of the message and signing the resulting number in a suitable expansion. Note that the dimensions of the message matrix and the control matrix need not be related; we assumed this merely for explanatory purposes.

The security of the Lamport and the matrix-based signature methods has recently been studied in detail in [23]. A further efficiency improvement can be found in [37], where it is observed that a matrix can be used for a larger message space, by defining a one-to-one relation between messages and subsets of entries in the matrix, such that no subset can be computed from any other and each subset contains exactly one entry in each column. This condition is fulfilled by taking entries according to a Latin square, or more generally by taking all subsets with entries such that the sum of the rows that the entries are in is a constant. A generalization of this technique to directed acyclic graphs is studied in [3].

Tree Authentication A serious drawback of both the Lamport and the matrix-based signature methods is that it is insecure for the bank to let paying devices sign more than once with respect to the same public key. As an extreme example, consider a wire-tapper who intercepts two signatures made by the same paying device, in either signature scheme, one for the message having all r-ary digits equal to zero and another having all r-ary digits equal to $r - 1$ (with $r = 2$ for the Lamport signature scheme); from then on he would be able to forge signatures of that paying device for any message. Signature schemes in which a public key may be used for signing only a single message are called *one-time* signature schemes.

An extension of one-time signatures, due to Merkle [32], enables paying devices to use a single certified public key for signing many messages, using a one-time signature for each. The method preserves much of the computational efficiency of the one-time signature scheme, and can be implemented in a standard 8-bit smart card microprocessor. To illustrate the technique, we use for concreteness the matrix-based one-time signature method as the basic building block, following Merkle; it will be clear that this technique can also be applied to any other one-time signature method. Viewing the message matrix and the control matrix as a single matrix, the idea is to take multiple matrix instances as leaves in a tree. Each node value in the level just above the leaves is computed by compressing the top rows of the child matrices. All the other node values in the tree are computed by compressing the child node values, and the value of the root node serves as the public key of the paying device.

To compute a digital signature on a message, the paying device uses a matrix in a leaf of the tree that has not been used before. In addition, the paying device must release the compression values of sufficiently many nodes, to enable the receiving device to compute the root value and thus to verify the public key. In case of a binary tree, the paying device to this end releases the compression result for the sibling leaf matrix and, for each of the internal nodes in the path from the leaf used to the root, the value of the sibling node. A binary tree of depth n enables the paying device to securely make 2^n signatures with respect to the same public key.

As with the one-time signature methods, the key storage space for the paying device can be reduced by generating the bottom rows of the leaf matrices all from a single secret. Only the seed and the certificate on the root value then need to be stored by the paying device. With respect to dynamic storage and computational efficiency for the paying device, one-time signatures are best used up from the leftmost to the rightmost leaf in order. As shown in [32], it is then possible to generate the additional node values for a new one-time signature in an efficient way from the additional node values of the previous one-time signature. According to later investigations, symmetric tree structures are not optimal in this respect; it is better to use a rake-shaped form.

44.3 Electronic Cash Techniques

We are now prepared for an explanation of the techniques for electronic cash design.

Representing Electronic Cash

The presentation thus far has been independent of how electronic cash is represented. Two fundamental ways exist for representing electronic cash in computing devices, whether paying or receiving devices. First is to indicate an amount of electronic cash by means of the value of a counter, maintained in a chip register. For example, 100 electronic dollars spendable up to cent granularity would be represented by a counter value of 10, 000. This representation is often referred to as **register-based cash.** Since money can be forged when counters can be bypassed or updated without bank authorization, the security of systems using this representation of electronic cash relies critically on tamper-resistance.

The other way to represent electronic cash is in the form of cryptographic tokens that are digitally signed by the bank. To each token at least a fixed denomination and currency are assigned, and possibly also attributes such as an expiration date and how many times it may be passed on. The tokens are called **electronic coins,** and must be unforgeable and verifiable solely by using the signature public key of the bank. In its simplest form, each coin is a different (message, signature) pair, from now on referred to as the *two-part form* for coins. The denomination and currency, and any other attributes, can be encoded into the message, or can be indicated by the particular key used by the bank to compute its digital signature. The security of electronic coins relies crucially on the secrecy of the bank's secret key for signing.

Each of the two methods for representing electronic cash has its own characteristic implications for security, functionality, and efficiency of a cash system. The register-based representation has the advantage over electronic coins that storage space is minimal. In contrast, for each electronic coin, storage space must be allocated. Coin verification moreover inherently requires the ability to verify digital signatures, which is more costly than verifying MACs. Furthermore, the complexity of communication and computation for coins increases with the number of coins needed to form an amount (although for some cryptographic embodiments, techniques are known to trade storage space against computational requirements). Another disadvantage is that when coins of appropriate denomination are not at hand, a payment requires change from the receiving device or a new withdrawal. In contrast, with register-based cash any amount less than the current register value (or a predetermined maximum) can be paid.

Cash held by paying and receiving devices can be either register-based or in the form of electronic coins, and within the same system any of the four possible combinations can make sense. Paying devices may even hold part of their value in the form of electronic coins, and the rest in the form of register-based cash (see "Discussion"). The implications of the various combinations will become clear in the rest of this section, and in particular in "Privacy of Payments" we will see that there are valid reasons for preferring coins over register-based cash.

Transferring Electronic Cash

To transfer electronic cash securely from a paying device to a receiving device, secure authentication by the paying device is required. The techniques for authentication detailed in "Authentication Techniques" can all be used in combination with either form of electronic cash representation, but significant differences exist in the manner in which this is accomplished. In any case, the internal cash balance (whether register-based cash or in the form of coins) of a paying device should be decreased before transferring the electronic cash; otherwise the transaction could be interrupted by the holder of the device between the sending of the electronic cash and the internal adjustment (and we do not want to depend on the receiving device to send an acknowledgment).

Transferring Register-Based Cash

When combining an authentication method with a register-based cash representation for paying devices, not only the paying device but also the amount that is transferred must be authenticated. The paying device must decrease its register value to reflect the amount that is transferred, and of course it must have been programmed by the bank to do so only when its current value represents an amount exceeding the payable amount.

In order to authenticate the amount, it can be encoded into the challenge. Because for security the varying part of the challenge must remain intact, the resulting challenge becomes longer. In case a correct response is returned, the receiving device is assured not only that the other device is a paying device, but also that the paying device has decreased its local balance by the specified amount (assuming that its tamper-resistance has not been broken, of course).

The receiving device can store the transferred cash either as register-based cash or in the form of an electronic coin. In the case of zero-knowledge authentication, or authentication by means of a MAC, only the former representation can be used, since the receiving device does not receive information that could only have been created by a paying device or by the bank. In the case of authentication by means of a digital signature computed by the paying device, the digital signature received can serve as an electronic coin, having a denomination specified by the challenge message that has been signed. Alternatively, the use of digital signatures only serves as a more efficient alternative to zero-knowledge authentication (less interaction) and to make an unforgeable transaction log for use by the bank, and the receiving device stores the amount received in the form of register-based cash; a system based on this approach has been proposed in [24].

In case the receiver stores the received amount in the form of a coin, either the coin may be passed on for a subsequent payment, or it can only be deposited. In the latter case one speaks of an **electronic check** payment, because the paying device fills out the amount at payment time and signs it in an unforgeable manner. A check that has not yet been filled out is called a blank check; it has no value when it is issued, and hence need not be prepaid. An important aspect of electronic check payment is that the bank does not need to trust in the tamper-resistance of receiving devices. For security, blank checks should have a maximum spending limit, assigned by the bank at the time of issuing.

Authentication of the paying device can also take place in an indirect manner, by using a *session key*. In the case of conventional authentication, to this end the paying device sends its ID number to the receiving device, which uses the master key to compute the diversified key of the paying device. The receiving device then sends a random challenge to the paying device, and both parties use the secret key of the paying device to compute a MAC for the challenge. However, rather than the paying device sending the MAC to the receiving device, as a response, both parties regard it as a session key. Further messages, such as for the transfer of an amount, are authenticated and/or encrypted using the session key. Since these tasks can be performed only if the session key is known, valid MACs and/or messages decrypting to meaningful messages convince the receiving device indirectly that the other device must be a paying device.

In the case of public-key cryptographic authentication, the two devices can send their certified public keys to one another, and develop a session key by means of Diffie–Hellman key exchange [21] or another method. The session key can be used to compute MACs for subsequent messages and/or to symmetrically encrypt messages. As with session keys derived from MACs, the ability to do any of the above demonstrates indirectly to the receiving device that the other device is a paying device. In contrast to session keys derived using conventional authentication, no master key needs to be held by devices outside the bank.

Alternatively, the following public-key cryptographic technique can be used to form a session key. The bank provides each tamper-resistant paying device with a unique secret key and a corresponding public key. The secret key is the trap-door of a trap-door one-way function $f(\cdot)$, specified by the public key of the device. To transfer cash to a receiving device, the paying device sends to the receiving device its public key and a certificate for it of the bank. The receiving device generates a random challenge x in the domain of $f(\cdot)$, and sends $f(x)$ to the paying device. The paying device uses its trap-door to uncover x,

and both devices use x, or a part of it, as the session key. Again, the actual transfer of the amount can be authenticated by means of a MAC or a digital signature. An advantage over the preceding method is that much faster cryptographic embodiments of this technique for forming session keys are known.

In case of conventional dynamic authentication it is sometimes preferable to reverse the roles of devices that hold a secret master key and devices that hold diversified secret keys. When receiving devices hold diversified keys and paying devices a master key, electronic cash transfer can take place based on the following observation: the ability of a paying device to compute a MAC using the secret key of the receiving device demonstrates to the receiving device that the device it is communicating with knows the master key, and hence that it must be a paying device. This is particularly useful in the withdrawal protocol, because it enables the bank to issue register-based electronic cash to paying devices without having to provide these with its master key.

Transferring Electronic Coins

To issue electronic coins to a paying device of an account holder, the bank computes digital signatures on distinct messages, sends these to the device (possibly encrypted to prevent wire-tapping) and debits the account of its holder by the total value of the coins. To make a payment, the paying device first determines whether coins of appropriate denominations are present. If the amount cannot be made up, either an excess amount must be formed and the receiving device must supply change, or first another withdrawal must be performed.

When coins are in the two-part form, the receiving device must be tamper-resistant and the coins must be encrypted before being transmitted, to protect against coin theft or copying. When using conventional authentication, a session key can be formed by the paying and the receiving device, which is then used by the paying device to encrypt the coins. Before sending out the coins, the paying device erases the coins from memory. The receiving device decrypts, verifies the coins using the public key of the bank, and stores them. Depending on the number of times the coins may be passed on, the receiving device can later on either deposit the coins or use (some of) them to make a payment. As we have seen, the presence of master-keys in devices outside the bank can be avoided by forming session keys using a public-key cryptographic method.

The need for receiving devices to be tamper-resistant can be avoided by defining a coin to be a triple consisting of a secret key, a corresponding public key, and a certificate of the bank on the public key. From now on this is referred to as a coin in the *three-part form.* The secret key of the coin belongs to a paying device, and at least part of it may not be known to anyone else than the device itself (and perhaps the bank). For each triple, the bank charges the account of the device holder for the value of the coin. To spend a coin, the paying device computes a digital signature on a challenge message of the receiving device (zero-knowledge authentication makes the approach pointless), using the secret key of the coin triple, and sends it to the receiving device, together with the public key and the certificate of the coin triple. The receiving device can verify the payment by using the public key of the bank. The received coin cannot be passed on, since the secret key of the coin has not been divulged by the paying device and in a subsequent payment another challenge will have to be signed. Consequently, the receiving device can only deposit the coin. By encoding into the challenge message a unique account identifier, uniquely associated by the bank with the account of the holder of the receiving device, it is effectively ensured that a wire-tapped coin cannot be deposited to another account. Note that this technique can also be applied to electronic checks.

Payments with electronic coins in the three-part form are very similar to electronic check payments. They differ in that for check payment a register-based cash representation is used and blank checks can be issued free of charge; the amount payable is encoded into the challenge message. In contrast, electronic coins are prepaid at withdrawal time, for their denomination value. Furthermore, each electronic coin is a new triple from the bank, while check payments can be made using a single triple (secret key, public key, certificate), installed by the bank in an initial stage. For greater efficiency, the bank can program paying devices such that they assist in spending coins in the three-part form a predetermined number of times

greater than one (a k-spendable coin at withdrawal time then must be prepaid for k times the coin value), but a limit is inevitable in order to determine the value for which the token must be prepaid.

When Tamper-Resistance is Compromised

Apart from stealing the secret key of the bank or discovering an algorithmic break or breakthrough (such as how to invert functions that are believed to be one-way), there are two ways in which an attacker may be able to forge electronic money. Thus far we have only touched on the possibility that tamper-resistance might be compromised, and investigated value transfer methods under the assumption that secret keys in tamper-resistant devices cannot be physically extracted by attackers. In reality, there is no such thing as tamper-proofness. Experience has shown that organized crime can hire expertise comparable to that in national laboratories, and even hackers nowadays have access to sophisticated tools. Moreover, care must be taken that device secrets cannot be extracted in much simpler ways than by direct read-out; see, for example, recent research results on cryptanalysis in the presence of induced hardware faults, initiated by [4]. When secrets can be extracted from paying or receiving devices, nothing distinguishes counterfeit from cash issued by the bank. What financial damages can this result in?

Tamper-resistance is best viewed as a matter of economics. Roughly speaking, a technology for tamper-resistance offers adequate security for the bank if the required cost for breaking tamper-resistance exceeds the fraudulent profit that can be made as a result. When estimating the expected fraudulent profit that can be made in an electronic cash system, one also needs to take into consideration the economics of large-scale cracking; to crack a single smart card, equipment and expertise running into hundreds of thousands of dollars may be needed, but this is largely a one-time investment. The damage that can be done ultimately depends on the measures incorporated into the system for preventing or discouraging forgery of electronic cash.

Fraud Detection

Of primary importance is the capability of the bank to *detect* whether unauthorized electronic cash is being introduced into the system. Unless special measures are incorporated, nothing prevents the bank from accurately keeping track of the electronic cash held at various moments in time by the paying and receiving devices of each of its account holders. To keep track of the flow of electronic cash between devices, transaction transcripts that reveal at least how much cash has been transferred must regularly be made available to the bank. If not, then the bypassing of registers of compromised paying devices, or the multiple spending of coins, cannot be detected in any other way than through the economic side-effect of hyper-inflation. With sufficiently detailed and regular transaction monitoring, the bank can at the very least detect whether more electronic cash is circulating than it has actually issued.

Electronic coins and checks in this respect have an important advantage over other methods: the transaction record is tied in with the received value, into the digitally signed message of the paying device, and hence received coins or checks cannot be deposited without automatically also revealing the corresponding transaction logs. There is also an important distinction between electronic coins and electronic checks. Instead of double-spending a check, an attacker can also make a profit by spending it for the maximum spending limit, while bypassing the register in the paying device. The bank can detect this only by keeping track of the balances of all paying devices. If such a "shadow" balance ever drops below zero, it is clear that forgery has taken place. With electronic coins, the only way for an attacker to make a fraudulent profit is by double-spending withdrawn coins, and so any forgery of electronic cash can alternatively be detected by the bank by maintaining a list of all deposited coins and checking at each deposit for double-spending. This distinction may not be relevant in case checks and coins are fully traceable, because in effect the same intrusive transaction logging takes place. However, it is crucial when privacy measures need to be incorporated, as we will see in "Privacy of Payments."

Fraud Tracing

In case a master key is present in devices outside the bank, an attack on the master key enables cash forgery that cannot be traced to a single device, at least not on the basis of transaction logs. With conventional authentication methods, the ability to extract the master key enables an attacker to perform the authentication for any paying device. When cash held by paying devices is in register-based form, the attacker can forge MACs for any legitimate amount, for any paying device.

In order to be able to *trace* devices that have been compromised, rather than merely those that have been passed for, the bank should not deploy system-wide secret keys. This calls for cash transfer based on public-key cryptographic authentication. When coins in the two-part form are transferred in encrypted form, an attacker needs to wire-tap and decrypt or to physically extract coins from the paying device or the receiving device, in order to double-spend coins. Even though in these cases the bank can determine which devices have been passed for thus far (to this end, transaction dumps should reveal the ID numbers of paying devices), it has no way of tracing the source of the fraud. Namely, when a coin in the two-part form has been double-spent, it is not clear whether the attack has been on the paying device or on any of the receiving devices in the chain of payments with that coin.

An important advantage of coins in the three-part form over coins in the two-part form is that a coin in the three-part form can be double-spent only by physically extracting the coin secret key from the paying device in which the coin is stored; short of a physical attack on the paying device, the secret key of a three-part coin never leaves the paying device. By keeping track of which paying device has stored which coin secret keys, any forgery of money, no matter how little, can always be traced by the bank to the compromised paying device.

Fraud Liability

When counterfeit can be traced to a specific device, it is not necessarily the case that its holder is a criminal. After all, the device might have been compromised by a thief. In order to be able to point the finger to the holder of the device on firm grounds, the bank must require all its account holders to comply with mechanisms for reporting loss, theft or hardware defects. It should also issue personalized paying devices, that may not be swapped. This can easily be achieved by issuing cards with an access control mechanism, such as a PIN or biometric verification; as will be discussed in "Preventing Loss," this is also desirable for account holders, in view of loss tolerance.

Furthermore, measures to ensure that thieves cannot operate stolen devices can be incorporated. In this respect, however, little added value comes from protecting tamper-resistant devices with PINs, passwords, or biometric access mechanisms, because it can be expected that a hardware attacker can bypass these as well. However, a special use of a secret access code can offer added security. To this end the secret key of a paying device is computed as a function of information held by the device and secret access information that is provided by its holder. Each time after having used the secret key to perform a transaction, the device erases it from memory together with the provided access information. This ensures that the secret key is never present for a long time in the memory of the computing device, and is never stored in nonvolatile memory. A randomly chosen eight-character password suffices to prevent a successful exhaustive search using currently available technology. In case a successful forgery due to the compromise of tamper-resistance is traced to a specific device, its holder is either a fraud or has been very sloppy with the secret access information.

To be able to noncontestably demonstrate that a forgery has been made using the keys held by a certain paying device, the trace information that becomes available to the bank in case of forgery should be such that not even the bank itself could have computed the trace information, had there been no forgery. For example, this evidence could be the secret key that the paying device uses to compute digital signatures at payment time (for checks or coins in the three-part form), or the bank's ability to show $k + 1$ signatures with respect to the same secret key of a device that has been programmed to compute only k signatures with the same key. To this end, paying devices must be able to choose their own secret keys, unknown to

the bank, which calls for at least some of the paying device's operations to be under the control of its holder. We will defer a discussion of how to achieve this to the section on "Guaranteeing Your Own Privacy," since this scenario is also very important to achieve privacy of payments.

Fraud Containment

When forgery of money can be traced to paying devices, the bank can blacklist the devices. To this end, it places the device ID numbers (or ranges thereof) in a list that is regularly distributed to all receiving devices, for example whenever these perform a deposit. Alternatively, the bank can blacklist (one-way hashes of) the secret or public keys used by the compromised devices for making payments. Furthermore, the ability to trace compromised paying devices may suffice to stop the fraudsters from continuing their fraud.

In case tracing of compromised devices is not possible, the bank can stop further fraud only by revoking its own keys. In case of conventional authentication this requires the distribution of a new master key, and in case of public-key cryptographic authentication a new public key for certification must be made available.

By refreshing master keys and certification keys on a regular basis, indicated by expiration dates, containment can be obtained even for frauds that are difficult to detect. For the same reason, it is desirable that tamper-resistant devices can make and/or receive payments only until a built-in expiration date.

Security for Account Holders

The major concern thus far has been security for the bank against forgery. For account holders, other security aspects are of importance. Firstly, the electronic cash held by a device should not disappear in any other way than by spending it at the approval of its legitimate holder. Secondly, it should not be possible for an attacker to redirect a payment to a party other than that intended by the legitimate holder of the device. Thirdly, whenever the behavior of an honest account holder is disputed, by the bank or another party, the account holder should be able to substantiate his innocence; this is also in the bank's interest, as discussed in "Fraud Liability."

Preventing Loss

In general, any lost electronic cash that cannot redeemed by the bank, for instance because proper recovery measures have not been incorporated, results in a profit for the bank. Once it becomes aware of the loss, the bank can take the value out of the float account.

There are various ways for an account holder to lose electronic cash. Firstly, the contents of his device may be garbled because of a device crash or a hardware fault, or his device may get lost or stolen; measures to protect against these events are referred to as loss-tolerance measures. Secondly, a payment transaction can be interrupted with the effect that the paying device has debited the payable amount while the other device has not received it; this must be protected against by fault-tolerance measures. Thirdly, another party may be able to withdraw from his account at the bank. Finally, in case of a receiving device that is not tamper-resistant, an attacker may have the device accept bogus money by modifying its software or by substituting its copy of the public key of the bank.

Loss-Tolerance Loss-tolerance is of concern for both paying and receiving devices, and requires that a destroyed, lost, or stolen amount of electronic cash can be recovered. Recovery from a lost or stolen device requires greater caution by the bank than recovery from a crash. Namely, in the case of a crashed device the bank can scrutinize the device to determine whether there is any chance of its holder having extracted the secret key (or having attempted to do so), while in the case of loss or theft the bank cannot *a priori* distinguish between a victimized account holder and one faking loss or theft. In view of this, tamper-resistant devices at the very least should have a suitable access control mechanism (PIN, password, or biometric), so that another party cannot spend (or deposit to their own account) the cash held by a lost

or stolen device without breaking its tamper-resistance. The bank should also require timely reporting of lost or stolen devices, so that device ID numbers, or withdrawn coins or blank checks, can be blacklisted.

The recovery itself can be done either by the victimized account holder himself or in cooperation with the bank. The former is possible only when the account-holder is able to regularly make backup copies of the contents of his device. In systems based on electronic coins in the three part-form or electronic checks, this measure can easily be applied for receiving devices, since these need not be tamper-resistant and received cash can be deposited only to the account specified in the digitally signed challenge message.

Cooperation with the bank is needed whenever a paying device crashes, or a receiving device that holds a secret key of the bank, because parties other than the bank cannot make backup copies of the secrets in these devices. In systems in which the bank can determine how much was present in the device before it crashed, for example by examining transaction logs and perhaps also information of the user about his most recent payments, the bank can issue new device and adjust its cash representation to reflect the value at the moment of the crash, or credit the account of the account holder for the lost value. A discussion of several loss-tolerance schemes for register-based cash can be found in [38].

In case a device is reported to have been lost or stolen, the bank should delay reimbursement, if necessary until the current version of electronic cash has expired, to make sure that cash can either be reimbursed through recovery or be spent by means of the normal paying protocol, but not both. Expiration dates can be built in by the bank changing its keys every now and then; this is desirable anyway, for the purpose of containment in the case of theft of the bank's own secret keys.

In "Discussion" we will see that loss-tolerance can be implemented even in case payments are untraceable.

Fault-Tolerance Fault-tolerance is mainly an implementation issue. To cope with transaction interruption, a device should at all times be prepared to resend the last message it sent out. For security, a device should as a rule resend only exactly the same message that it sent out previously (of course without modifying its internal cash representation a second time). When setting an interruption flag before sending out a message for the first time, upon reset the device can infer from the flag that it needs to do a re-transmittal. Once an acknowledgment is received (this can be implicit in the response message of the other device), or after a time-out, the device can clear its interruption flag. New actions are performed only once the flag has been cleared again. Transaction processing is a specialty in its own right, and the above is obviously an oversimplication to convey the general idea. A thorough account of transaction processing principles and methods can be found in [26].

Account Access Control In order to prevent withdrawal from account by an unauthorized party, the bank should grant account access only to parties that are properly authenticated. To this end, a MAC or a digital signature of the account holder must be required. In the former case, the secret key needed to gain access to an account is known also to the bank, and fraudulent parties (such as bank employees), may be able to gain access to the key. By digitally signing crucial parts of the withdrawal request, the requested amount, and date and time of the request (and perhaps also the most recent account balance), an account holder can always disavow an unauthorized withdrawal. Of course, withdrawal requests must also be protected against replay attacks; for this purpose each message should contain a fresh part (i.e., an account access sequence number, a random challenge, or a time/date estimate), that must be signed along with each request message.

Software Integrity An attacker who gains access to a receiving device must be prevented from modifying the software in order to have it accept bogus information. In case the receiving device is not tamper-resistant, the attacker may change, say, a function pointer, so that the software calls a verification routine that always accepts. Alternatively, an attacker can attempt to substitute the key used by the receiving device for verifying electronic cash payments (the master key or the public key of the bank). The attacker can even prevent the receiving device from being informed about the error at deposit time, by substituting

the error message of the bank by another, since he can also have the receiving device accept arbitrary strings for digital signatures or MACs of the bank. Protection against unauthorized access and computer viruses can be achieved by using measures such as password protection and secure operating system software.

Preventing Payment Redirection

A more subtle way in which an attacker might attempt to steal electronic cash is to redirect to its own receiving device a payment intended for another party, or to redirect it to the paying device of another party that the attacker intends to pay. This is known as a *man-in-the-middle attack*.

One way to prevent the wrong party from being paid is to explicitly direct an electronic cash payment to the (account of the) intended payee, by making a digital signature at payment time. To this end, three-part coins or blank checks can be used. When an ID number or a certified public key of the intended receiving device is known to the paying device, it can be included in the challenge message that is signed. Alternatively, a random number or a digital pseudonym can be used, as long as it is uniquely associated by the bank with the account of the payee; in this manner the payee can remain anonymous to the payer. As mentioned in "Transferring Electronic Coins," this measure also prevents an attacker from depositing wire-tapped electronic coins or checks to his own account.

For payment platforms in which the paying device and the receiving device need to be in physical proximity, a generally applicable technique known as *distance bounding* can be used to prevent a fraudulent receiving device from passing on a payment to another receiving device. The idea is for the receiving device and the paying device to mutually agree on the (random part of the) challenge message that is to be responded to by the paying device, by each sending a series of random bits to the other, one by one and interleaved, and concatenating the bits to form the challenge. By timing the delay, the receiving device can determine an upper bound on its distance to the other device. What makes this approach practical is that today's electronics can easily handle timings of a few nanoseconds, and light can travel only about 30 cm during one nsec. Even the timing between two consecutive periods of a 50 MgHz clock allows light to travel only three M and back. Details and variations of the above technique can be found in [12].

Nonrepudiation

Account holders should be able to disavow erroneous or false incrimination of fraudulent behavior. When parties with differing interests authenticate their messages using MACs or zero-knowledge proofs, the transcript of a communication cannot be used by another party to later on demonstrate that the communication took place, since zero-knowledge transcripts can be formed by anyone. Nonrepudiation requires the use of digital signatures, since these can only be computed by the party associated with the public key needed for verification (assuming proper protection of the secret key). When crucial parts of the withdrawal, payment and deposit protocols are digitally signed, all parties can disavow false claims. Of course, as in "Fraud Liability," this approach works only if each account holder can generate and control his own secret keys, which requires part of (the operations of) his device to be under his control; this is addressed further in "Guaranteeing Your Own Privacy."

Privacy of Payments

Confidentiality of transaction details is easily achieved by means of line encryption and not depositing details such as the purchase description, and the prevention of traffic analysis is a problem largely orthogonal to the design of an electronic cash system. We now turn to what might well be the most complex issue in the design of an electronic cash system: how can we incorporate untraceability of payments without encouraging criminal uses of electronic cash?

Relaxed Monitoring, Anonymous Accounts, and Anonymous Devices

Untraceability can to some extent be incorporated into an electronic cash system by the bank relaxing the requirements for transaction log dumping. When tamper-resistant receiving devices only dump transaction logs with aggregated transaction details, and leave out any information that can be correlated to paying devices or payers, payments cannot be traced. The bank still needs transaction logs to reveal payment amounts and, preferably, time and date of each transaction, in order to be able to detect counterfeit. A distinct problem with this measure is that the detection of counterfeit, originating from a successful hardware attack, is possible only when the aggregated amount of forgery exceeds the total amount in the bank's float account, which clearly is unacceptable for an open system. An improvement can be obtained by categorizing paying devices into several groups, and requiring aggregated transaction records to reveal these group IDs, but this causes a trade-off with payment untraceability. Further serious drawbacks of the approach are that fraud cannot be traced by electronic means only, blacklisting of devices is not possible, and payers have to trust receiving devices to not deposit individual transaction records (unless paying devices do not provide any trace information in the first place, which is the key to the approach in "Blinding"). Relaxed monitoring and security for the bank always need to be traded off with each other.

Another approach towards untraceability is anonymous bank accounts. Although the use of anonymous accounts protects the identity of payers, all the transactions conducted by the same account holder are linkable; a single identification of the account holder in any transaction enables the bank to trace all past and future transactions of the account holder. Increasing the degree to which payments are unlinkable to the payer by allowing users to swap their devices with other users conflicts with the desirability of a device access control mechanism (such as PIN verification). Not only can this approach hardly be said to offer privacy of payments, it also conflicts with the tracing capabilities of the bank in case of forgery. Anonymous accounts moreover are not allowed in most countries, because they interfere with the tax system.

A third straightforward approach is for the bank to issue tamper-resistant paying devices in such a manner that it does not know which account holder receives which device. Again, this approach cannot be said to offer privacy, because all payments made with the same tamper-resistant device are linkable. It is also difficult to run the device distribution process in a manner that randomly distributes the devices over sufficiently many account holders, and ultimately the provider has to be trusted to properly conduct the distribution. Furthermore, by withdrawing cash from a named account the identity of the holder of the device is revealed. Other drawbacks are that the tracing capabilities of the bank are seriously reduced, tracing cannot be conducted electronically on the basis of transaction logs, and it is difficult to assess to which extent the holder of the traced device is responsible for a fraud originating from that device.

Blinding

A much better way to ensure untraceability of payments is by application of special cryptographic techniques in the design of an electronic cash system. These can ensure that the information that the bank learns about its account holders and their devices is uncorrelated to the information that is revealed by devices when making payments.

In a basic cryptographic paradigm due to Chaum [15], a receiver can obtain from a signer a digital signature on a message, in such a manner that the message and the signature remain completely unknown to the signer. More specifically, in each execution of the protocol, the receiver can obtain with uniform probability a single pair (message, signature) from the set of all possible such pairs, and the signer has maximal uncertainty about the particular pair it has issued. The receiver is said to **blind** the execution of the protocol, and the protocol is called a blind signature issuing protocol.

Efficient blind signature issuing protocols are known for a variety of practical digital signature schemes. The most efficient such protocol known to date is for RSA signatures, and is due to Chaum [15]. It enables a receiver to obtain a pair $(m, m^{1/e} \bmod n)$, where n is an RSA modulus, e is an RSA encryption exponent

(co-prime to $\varphi(n)$) and m is a message satisfying an appropriate redundancy pattern (alternatively, m is a one-way hash of a message), as follows:

Step 1. The receiver picks at random a message m from the message space, and a random **blinding** factor r from $\mathbb{Z}_n^*$. The receiver then sends $m_0 := r^e m \bmod n$ to the signer;

Step 2. The signer sends $s_0 := m_0^{1/e} \bmod n$ to the receiver; and

Step 3. The receiver computes $s := r^{-1} s_0 \bmod n$.

It is easy to see that s is a digital signature of the signer on m, and that the condition for a blind signature issuing protocol is fulfilled.

The basic blinding technique can be used straightforwardly to design an untraceable electronic cash system, as follows. The bank issues electronic coins in the two-part form (message, signature), by means of a blind signature issuing protocol. To this end, the three steps above are performed in parallel, one execution for each requested coin, and the account holder in Step 1 specifies the desired number of coins and their denominations. The account holder also digitally signs its request message (and a fresh part, for replay prevention) to prove to be the holder of the account. For each denomination, the bank uses a different RSA exponent, and all RSA exponents are co-prime. To make a payment, the paying device selects coins of the appropriate denominations, encrypts them using an authenticated public key of the tamper-resistant receiving device, erases the coins and then transfers the encrypted payment message.

By following the blind signature issuing protocol, using properly generated random numbers, all payments are anonymous and unlinkable. In fact, even with infinite computing power the signer cannot trace payments, if only account holders use "genuinely random" blinding factors. For greater efficiency (and reconstructability), the blinding factors applied by an account holder may all be generated from a single secret key by using a pseudo-random number generator, but then untraceability is only computational. A particular danger of this is that many years from now it may be feasible to retroactively trace payments, by analyzing the archived deposit databases of the bank; the expected progression in sheer computing power and advances in algorithmics make this a realistic scenario. Surveillance and tracking capabilities are the primary tools of political oppressors to resist opposition, and changes in a political climate are not always predictable. For this reason it is preferable for any electronic cash design for national or global use to be independent of the particular manner in which blinding factors are generated.

By regarding the public key of a coin in the three-part form, (secret key, public key, certificate), as the message in a blind signature issuing protocol, it follows that a paying device can withdraw completely blinded coin triples by using the above blinding technique. Alternatively, blinded triples serve as blank checks, in combination with a register-based cash representation in the paying device. Note that the bank does not need to know the secret keys of the triples obtained by the paying device. As we have seen in "Fraud Liability" this is desirable for the bank, because it is then able to come up with an incontestable proof in case a k-spendable coin or blank check has been used at least $k + 1$ times. Of course, the bank must also prove that the coin or blank check has indeed been withdrawn by the account holder, and not been constructed by itself; this can be accomplished by showing the (digitally signed) request message that the account holder sent in Step 1 of the execution of the issuing protocol in which the coin or blank check was issued.

A general drawback of the basic blinding technique is that the bank can never *trace* forgery: even fraudulent payments are untraceable. For the same reason the bank cannot apply blacklisting to contain further fraud. Moreover, in case of checks substantial amounts of forged cash may be injected into the system without the bank even being able to detect this. As a consequence, the basic blinding technique is appropriate only to the design of electronic coin systems in which payments are deposited *on-line;* in case of a double-spending attempt, the bank can then simply tell the receiving device to not accept the payment. By having the paying device encrypt its coins for the bank, instead of for the payee's device, neither the paying nor the receiving device needs to be tamper-resistant. On the downside, one of the two major advantages of prepaid electronic cash over instruction-based payment forms, off-line payment

verification, has now been sacrificed completely in order to achieve the other, untraceability. We want to achieve *both* properties.

One-Show Blinding

In order to enable the bank to trace double-spent coins, without sacrificing the untraceability of coins that have been spent only once, the paradigm of **one-show blinding** has been introduced in [18]. This paradigm requires the construction of an issuing protocol and a payment protocol that securely act in concert, in the following manner. The payment protocol must be such that a signature of the paying device on one challenge message does not reveal any information that helps tracing, but any two different signatures, for the same coin, reveal trace information. This trace information must be encoded by the bank into each coin that is issued. To this end, coins are represented in the three-part form, (secret key, public key, certificate), and the issuing protocol must be such that the coin public key and the coin certificate can be fully blinded by the paying device, while the bank must make sure that the coin secret key contains an identifier (note that the tamper-resistance of the paying device may already have been defeated by an attacker before the issuing takes place), at least with substantial probability.

For the payment protocol, a one-time signature scheme can be used, with the property that the computation of two signatures with respect to the same coin public key enables the computation of the coin secret key, or at least of the identifier. To deposit the received coin, the receiving device transfers to the bank the coin public key, the coin certificate, the challenge and the signature of the paying device. The bank verifies the information by checking its own certificate, the signature, and the uniqueness and correct formation of the challenge message. Any double-spending of the same coin (for which the tamper-resistance of the paying device must be compromised) results in the deposit of a second signature with respect to the same coin public key, but on a different challenge message. From any two such different signatures the bank can compute the identifier, and hence trace the compromised paying device. Of course, this paradigm can also be used for electronic checks.

Two fundamentally different approaches are known for designing issuing protocols for the one-show blinding paradigm.

Cut-and-Choose Blinding The first approach can be inferred from Chaum et al. [18], and is commonly called **cut-and-choose blinding.** To retrieve a coin of the specified three-part form, the paying device and the bank engage in performing in parallel a great many executions of a basic blind signature issuing protocol, with the notable difference that the bank completes its part of the protocol only for one of the protocol runs, which it chooses at random. Moreover it does so only when the paying device can demonstrate for all the other protocol runs that it has properly encoded the required identifier into its messages (whence the name "cut-and-choose"); this is called *opening* of a blinded candidate.

To illustrate this process, consider using the blind RSA signature issuing protocol described in "Blinding" as the underlying blind signature issuing protocol. The paying device generates independently at random many "candidate" key pairs, (secret key, public key), such that each secret key contains the identifier in a prescribed manner. It blinds each of the public keys, in the same manner as one blinds messages in the basic blind RSA signature issuing protocol, and sends the resulting blinded public keys to the bank. The bank then requests the paying device to "open" all but one of the submitted blinded public keys, by revealing for each blinded public key the secret key and the blinding factor used in its construction. The bank verifies whether the opened candidates have been constructed properly and, if so, discards them and computes its RSA root of the remaining unopened candidate. Upon receiving the signed blinded candidate, the paying device removes the blinding factor, as in Step 3 of the blind RSA signature issuing protocol.

A very serious shortcoming of this straightforward implementation is that the probability of detecting a fraud increases only linearly in the number of protocol runs, and thus a huge amount of data must be exchanged in order to achieve a sufficiently low probability of successful deception. In the scheme actually proposed in [18], the security level is *exponentially* related to the number of protocol runs, an important improvement. Their construction, which is very specific and does not seem to allow for generalization, is

as follows. The pair (secret key, public key) of a coin triple is a key pair for making a Lamport signature (see "Dynamic Authentication based on Public-Key Cryptography"), but with a twist. Specifically, the public key is a set of k numbers, for an appropriate security parameter k, of the following form:

$$g\left(f\left(s_{01}\right), f\left(s_{11}\right)\right), \; g\left(f\left(s_{02}\right), f\left(s_{12}\right)\right), \ldots, g\left(f\left(s_{0k}\right), f\left(s_{1k}\right)\right) \; . \tag{44.4}$$

The functions $f(\cdot)$ and $g(\cdot)$ are collision-intractable, and the $2k$ secrets s_{ij} of the paying device are of the following form:

$$s_{0j} = \left(I \oplus a_j, b_j\right) \quad \text{and} \quad s_{1j} = \left(a_j, c_j\right), \quad \forall j \in \{1, \ldots, k\} \, , \tag{44.5}$$

for numbers (a_j, b_j, c_j) chosen at random by the paying device. The number I is an identifier, uniquely associated by the bank with the paying device. The certificate of the bank is an RSA signature on the public key. To retrieve a triple of the specified form, the paying device sends to the bank $2k$ blinded $g(f(s_{0j}), f(s_{1j}))$ terms, and opens only half of these, randomly selected by the bank. If the verification succeeds, the bank signs *the product* of the remaining k numbers. Upon removing the product of the k corresponding blinding factors, the paying device is left with one coin triple.

Of course, this time the paying device can slip in an erroneously formed $g(\cdot)$ term with probability $1/2$, and more generally n erroneously formed terms with probability close to $1/2^n$. To defeat successful spending of malformed coins, the payment protocol for a coin is designed as follows. The paying device sends the coin public key and the coin certificate to the receiving device, and in addition computes a Lamport signature on a k-bit challenge message, m. Denoting the binary expansion of m by $m_1 \cdots m_k$, the paying device releases for each bit m_i either $(s_{0i}, f(s_{1i}))$ or $(f(s_{0i}), s_{1i})$, depending on whether m_i is zero or one.

It is easy to see that one signature does not help the bank to trace the payment device, assuming that $f(\cdot)$ hides its first argument unconditionally. To guarantee that the probability of untraceable repeated spending is no greater than $1/2^n$, any two challenges must be guaranteed to differ in at least $2n$ bit positions (note that the possibility of payees cooperating with the payer must be taken into account). To this end, challenges can be expanded into code words with minimum distance $2n$. Double-spending the same coin (for which the tamper-resistance of the paying device must be compromised) results in the deposit of a second Lamport signature with respect to the same coin public key, but on a different challenge message. From any two such different signatures the bank can compute the identifier, I. Namely, for any two bit-positions where the two challenge messages differ, the bank knows both s_{0i} and s_{1i} and hence can compute a candidate identifier, by taking the exclusive-or of their first arguments; the majority candidate must then be the identifier sought for.

Note that the linear growth in the challenge length this time brings exponentially growing security, and that a guaranteed minimum distance of n may suffice to achieve the specified security level in case identifiers of account holders are random secrets and are never transmitted in the clear. Namely, in that case an attacker cannot encode valid identifiers of other account holders into his own coins, so that in case of double-spending with overwhelming probability only one of the computed candidate identifiers is valid.

Although ingenious, this improved realization of cut-and-choose blinding is still far from practical. For a practical implementation, k must be taken to be, say, 70, with 40 bits of each challenge containing a varying part, a receiving device ID and possibly an amount (in case of an electronic check), and the other 30 bits being the required hamming distance to get a sufficiently low probability of successful fraud without traceability. With practical choices for the numbers a_j, b_j, c_j and the blinding factors, the resulting signature size exceeds that for an RSA signature by about two orders of magnitude. Not only must all this information be transmitted, it must also be stored by the bank, for a duration at least as long as the validity of its certification key. Furthermore, the size of the data transmitted in the withdrawal of a single untraceable electronic coin or check is several hundred times the size of an RSA signature.

Restrictive Blinding The second approach to design issuing protocols for the one-show blinding paradigm is called **restrictive blinding.** It avoids the expensive cut-and-choose of the first approach, by inherently restricting the paying device in the manner in which it can blind the secret key for coin or check triples; the bank can then encode the identifier itself into the coin or check, by encoding it into the blinding-invariant part of the secret key.

In [9] a general technique is described to design efficient restrictive blind issuing protocols based on any so-called Fiat–Shamir type signature scheme (another protocol appears in [5], but its design does not follow a generally applicable technique). An important ingredient to this technique is the use of *secret-key certificates,* instead of public-key certificates, for coin or check triples. Although a secret-key certificate is not a digital signature of the bank on the public key of a triple, it offers the same functionality as a digital signature; see [8, 10] for details. The extra flexibility that comes from this can be exploited in the design of restrictive blind issuing protocols.

Following is an example of this technique, based on the Schnorr digital signature scheme [36] . The secret key of the bank is a k-tuple $(x_1, \ldots, x_k)$, where $k-1$ is the number of "identifiers" that the bank can encode independently into each triple that it issues; $k = 2$ suffices for electronic cash applications, as will be shown below. Its corresponding public key is

$$p,\ q,\ \mathcal{H}(\cdot),\ (g_0, g_1, \ldots, g_k)\ , \tag{44.6}$$

where q and p are primes such that q evenly divides $p-1$. The secret key $(x_1, \ldots, x_k)$ is chosen at random by the bank from $(\mathbb{Z}_q)^k$, g_0 is a random number of order q in $\mathbb{Z}_p^*$, and $\mathcal{H}(\cdot)$ is a collision-intractable hash function. (Alternatively, all computations can be performed on an elliptic curve of order q over a finite field; this allows working with much smaller numbers than needed in case of $\mathbb{Z}_p^*$, since the best known algorithms for discrete logarithms in such groups are exponential rather than subexponential.) The rest of the public key is generated deterministically:

$$g_i := g_0^{x_i} \bmod p \quad \forall i \in \{1, \ldots, k\}\ . \tag{44.7}$$

With the current state of computers and cryptographic knowledge, a 20-byte prime q and a 100-byte prime p should suffice for long-term security.

A certificate of the bank on a public key h of order q in $\mathbb{Z}_p^*$ for a paying device is a pair (r, c) such that

$$c = \mathcal{H}\left(h, g_0^c\, h^r \bmod p\right)\ . \tag{44.8}$$

Corresponding to the public key h of the paying device is a secret key $(y_1, \ldots, y_k)$ in $(\mathbb{Z}_q)^k$, such that

$$h = g_1^{y_1} \cdots g_k^{y_k} \bmod p\ . \tag{44.9}$$

The following issuing protocol enables the paying device to obtain a triple $(y_1, \ldots, y_k),\ h,\ (r, c)$ from the bank, in such a manner that the public key h and the certificate (r, c) are fully blinded, while the paying device cannot prevent the bank from encoding into the secret key $k-1$ numbers, $(I_2, \ldots, I_k)$, where $x_1 + \sum_{i=2}^{k} x_i I_i \neq 0 \bmod q$:

Step 1. The bank generates at random a number $w_0 \in \mathbb{Z}_q$, and sends $a_0 := g_0^{w_0} \bmod p$ to the paying device.

Step 2. The paying device generates at random three numbers $\alpha_1 \in \mathbb{Z}_q^*$, $\alpha_2, \alpha_3 \in \mathbb{Z}_q$, and computes

$$h := \left(g_1 g_2^{I_2} \cdots g_k^{I_k}\right)^{\alpha_1} \bmod p \tag{44.10}$$

and

$$c := \mathcal{H}\left(h, a_0 g_0^{\alpha_2} \left(g_1 g_2^{I_2} \cdots g_k^{I_k}\right)^{\alpha_3} \bmod p\right)\ . \tag{44.11}$$

The paying device then sends $c_0 := c - \alpha_2 \bmod q$ to the bank. Note that almost all of the workload can be precomputed, except for the modular multiplication by a_0 and the evaluation of $\mathcal{H}(\cdot)$.
Step 3. The bank sends $r_0 := (x_1 + x_2 I_2 + \cdots + x_k I_k)^{-1}(w_0 - c_0) \bmod q$ to the paying device.
Step 4. The paying device computes $r := \alpha_1^{-1}(r_0 + \alpha_3) \bmod q$.

It is not hard to show that if r_0 in Step 3 is such that

$$g_0^{c_0}\left(g_1 g_2^{I_2} \cdots g_k^{I_k}\right)^{r_0} = a_0 \bmod p\ , \tag{44.12}$$

which the paying device can verify if it so desires, then

- The pair (r, c) is a secret-key certificate of the bank on h;
- The pair $h,\ (r, c)$ is completely hidden from the "view" of the bank in the execution of the issuing protocol, regardless of the choice of $(I_2, \ldots, I_k)$; and
- The secret key $(y_1, \ldots, y_k)$ known by the paying device for h is such that $y_1^{-1} y_i = I_i \bmod q$, for all $i \in \{2, \ldots, k\}$. (Note that $\alpha_1 = 0 \bmod q$ results in an invalid public key.)

It can be proved, under a plausible assumption, that no conspiracy of attackers, each possibly with their own different tuple $(I_2, \ldots, I_k)$, can feasibly retrieve $l + 1$ different triples by performing l executions of the issuing protocol with the bank, for any $l \geq 0$, even if the executions can be arbitrarily interleaved. It can furthermore be proved that if the bank follows the protocol, then no conspiracy of up to $k - 1$ attackers can feasibly retrieve one triple (secret key, public key, certificate) from the bank for which the secret key does not contain any of the tuples $(I_2, \ldots, I_k)$ used by the bank in Step 3 of each of the protocol executions; if this were not the case, then it would be feasible to compute from scratch a pair $h, (r, c)$ such that $c = \mathcal{H}(h, g^c h^r \bmod p)$ without knowing $\log_g h \bmod q$, which is believed to be infeasible. It is believed that this result holds even for conspiracies of k or more attackers, although this conjecture has not been proved.

We now take $k = 2$, for simplicity, and show that the paying device can use the withdrawn triple to make a payment in accordance with the one-show blinding paradigm: two payments with the same triple enable the bank to compute I_2, while one payment does not reveal any information correlated to the execution of the protocol in which the triple was obtained. As before, the payment can be either an electronic check or an electronic coin payment. With h equal to $g_1^{y_1} g_2^{y_2} \bmod p$, the paying device can compute an Okamoto digital signature [33] on a challenge message m, as follows. It generates two random numbers s_1, s_2 from $\mathbb{Z}_q$, and computes $b := g_1^{s_1} g_2^{s_2} \bmod p$ and $d = \mathcal{H}(m, h, b)$. It then computes $r_1 := y_1 d + s_1 \bmod q$ and $r_2 := y_2 d + s_2 \bmod q$, and sends the signature (d, r_1, r_2) to the receiving device, together with the public key h and the certificate (r, c). The receiving device accepts the signature (r_1, r_2) if and only if the following holds:

$$d = \mathcal{H}\left(m, h, g_1^{r_1} g_2^{r_2} h^{-d} \bmod p\right)\ . \tag{44.13}$$

In addition, the receiving device must of course verify the certificate of the bank. At a later stage, the receiving device deposits the payment transcript. The bank verifies it in the same manner as described for the receiving device, and checks for double-depositing and double-spending.

It can be proved that the payment is unconditionally untraceable. Suppose now that the paying device is compromised by an attacker, and the attacker spends the same triple a second time, with respect to another challenge message, m^*, but using the same b. The signature for the new challenge message, m^*, is a triple (d^*, r_1^*, r_2^*) such that

$$d^* = \mathcal{H}\left(m^*, h, g_1^{r_1^*} g_2^{r_2^*} h^{-d^*} \bmod p\right)\ . \tag{44.14}$$

It follows from Eqs. (44.13) and (44.14), and the collision-intractability of $\mathcal{H}(\cdot)$, that

$$h = g_1^{\left(r_1 - r_1^*\right)/(d - d^*)} g_2^{\left(r_2 - r_2^*\right)/(d - d^*)} \bmod p\ . \tag{44.15}$$

Since $h = 1$ is not accepted by receiving devices (the bank will not redeem it, because it is not a generator), it follows that $\alpha_1 \neq 0 \bmod q$. But then

$$\left(r_2 - r_2^*\right) / \left(r_1 - r_1^*\right) = y_2/y_1 = \alpha_1 I_2/\alpha_1 = I_2 \bmod q \ . \tag{44.16}$$

To ensure that the attacker cannot use another b for the second spending of the same triple, b must be part of the triple. To this end, the bank requires in the payment protocol the use of a number b that has been hashed along by the paying device when computing c in Step 2 of the withdrawal protocol; this effectively turns the Okamoto signature scheme into a one-time signature scheme. Clearly, (r, c) must then satisfy $c = \mathcal{H}(h, b, g_0^c h^r \bmod p)$.

This technique can actually be used for a much more general paradigm than one-show blinding, because multiple identifiers can be encoded independently by the issuer. These identifiers may serve the role of credential values (attributes), about which all manner of properties can subsequently be demonstrated in the showing protocol. Specifically, returning to the general form $h = \prod_{i=1}^{k} g_i^{y_i} \bmod p$, Brands [11] shows how one can rapidly demonstrate that the numbers $y_1, \ldots, y_k$ satisfy a satisfiable formula from propositional logic, where the atomic propositions are linear relations modulo q, without revealing anything beyond the validity of the formula. This showing technique enables the paying device to rapidly demonstrate formulas such as

$$\text{“}\left[(5y_2 - 3y_3 = 5) \ \text{AND} \ (2y_3 + 3y_5 = 7)\right] \ \text{OR NOT} \ (y_2 + 4y_6 - 3y_8 = 5) \ .\text{”} \tag{44.17}$$

It can be ensured that if and only if the number of formulas (not necessarily the same) demonstrated with respect to the same h exceeds a predetermined threshold, then the bank can compute the secret key $(y_1, \ldots, y_k)$ of the triple. In the example electronic cash system described in Section 44.4, this general showing protocol technique is used as follows: by taking I_2 to be an identifier of the paying device, and letting I_3 denote a coin denomination specifier of the coin triple, the paying device at payment time can prove to the receiving device the denomination of the coin, without revealing anything about I_2. To this end, it proves the formula

$$\text{“NOT} \ (y_1 = 0) \ \text{AND} \ (y_3 = I_3 y_1) \ .\text{”} \tag{44.18}$$

If the coin is double-spent, I_2 can be computed and hence the coin can be traced to the compromised device.

Guaranteeing Your Own Privacy

The one-show blinding paradigm is not sufficient to design an untraceable electronic cash system. In fact, it is only half of the work. Namely, we have thus far assumed that paying devices are tamper-resistant and are issued by or on behalf of the bank; otherwise an attacker can easily forge money. However, if we do not entrust the bank with detailed information about the spending habits of its account holders, can we trust the bank to program paying devices so as to properly protect our privacy? This question is particularly relevant when one realizes that the idea of blinding fundamentally relies on the ability to produce random numbers that are unpredictable to the bank, and that today's smart cards can produce only pseudo-random numbers, on the basis of a random seed value. This seed value must be installed by the bank itself or at least under its supervision, because it must be guaranteed to be random and secret; but with the bank knowing all seed values, it can compute all the pseudo-random numbers that any paying device will ever compute, and thus the whole idea of blinding becomes pointless. Even if the bank were to tell us that it does not know the seed values, or that paying devices produce random numbers by postprocessing bits sampled from an internal source of “true” randomness (such as a noise diode), this cannot be publicly verified.

When transactions are conducted by directly interfacing a tamper-resistant paying device to a receiving device, it cannot be assessed whether the paying device secretly transfers additional information, such as a device ID, which would also make blinding pointless. A solution to this problem is to let all transfer of

information between the paying device and receiving devices flow through a computer that is *interposed* and trusted by the payer (a "user-controlled" computer). Its hardware and software may be purchased on the free market, and a knowledgeable user may even engage in manufacturing his own software and/or hardware. A desktop computer serves as a natural interposed computer when making Internet payments, and a handheld device is a natural candidate for paying at the local grocery store. When the paying device is in the form of a PC Card or a smart card, which must be inserted into a slot or a smart card reader, the user-controlled interposed computer can easily check that no data is transmitted in addition to that specified in the protocol description.

Usage of a user-controlled computer with a keyboard and display also offers another important advantage. Namely, the user can enter his password, PIN, or biometric using the computer's keyboard, and can read out the balance and payment information from the computer's display instead of having to rely on someone else's device. This makes fake terminal attacks impossible.

A further advantage is that the user-controlled computer can safeguard the user's secret keys, and can make, store (for the purpose of nonrepudiation), and verify all manner of digital signatures. It can also keep its own transaction logs, provide functions for cash-management, and so on. In addition, it may be able to take over part of the workload and/or storage burden of the tamper-resistant paying device, perhaps even to the extent that the tamper-resistant device does not need to perform any heavy number-theoretic operations.

However, without special measures the paying device may still be able to leak out covert information, by using a **subliminal channel** made available through the protocols used for communicating with the outside world (notably the protocols for withdrawal and payment). This leakage is called **outflow.** As a simple example, consider the paying device in the RSA-based one-show blinding system detailed in the section on "Blinding": in the secret values a_i, b_i, and c_i, some of which are revealed during payment, the paying device can easily encode an identifier and a great deal of other covert information, by using an encoding that only the receiving device, or perhaps only the bank, can recognize. Conversely, a receiving device may be able to transmit messages to the paying device through a subliminal channel, for example to instruct it to halt or to supply outflow in case its internal state adheres the supplied inflow information. This is called **inflow,** and can be achieved for instance by encoding the instruction into the random part of a challenge message.

In [16], and later in [20], a paradigm is proposed in which the interposed computer performs a much more active role than to merely check for the flow of additional information. Known as the "wallet-with-observer" paradigm, the idea is that proper cryptographic design of the withdrawal and the payment protocol may enable the interposed computer of the payer to ensure that the blinding factors used in the withdrawal protocol are unpredictable to the bank, and that no subliminal channels can exist.

To ensure that withdrawn coin triples are indeed uncorrelated to the view of bank, Chaum [16, 20] has the interposed computer develop any blinding factors and coin secret keys jointly with the paying device, in such a manner that they are randomized; randomized numbers cannot contain a subliminal channel.

In [17], an electronic cash system is described in which the interposed computer can prevent outflow and ensure correctness of the blinding process (a summary appears in [2]). Although this system suffers from several shortcomings, described shortly, we review it here for educational purposes and because it is the first (and, for a long time, the only) such system to have been proposed. Payments in the system are made using blank checks, as described in "Transferring Register-Based Cash." Blank checks are represented by triples (secret key, public key, certificate), which can be withdrawn from the bank by means of a basic blind signature issuing protocol. The (secret key, public key) pair of a triple is used to make a matrix-based signature, as described in "Dynamic Authentication based on Public-Key Cryptography," and the certificate is a blind RSA signature of the bank on the public key (as explained in "Blinding"). The following protocol steps take place:

Step 1. The tamper-resistant paying device and the user-controlled interposed computer together generate a mutually random number. The paying device ensures that the interposed computer does not learn it, while the interposed computer ensures that the check secret key, that will be derived from the random number by the paying device, cannot contain outflow. To this end, the paying device sends a commit on a (pseudo) random number, the interposed computer returns a random number, and the paying device combines the two numbers. The paying device does not (yet) open its commit to demonstrate its honest behavior in this process, because the interposed computer is not allowed to learn secret keys of check triples (this would enable multiple use of a blank check, without needing to break the tamper-resistance of the paying device).

Step 2. The paying device uses the mutually random number to compute the bottom row of a matrix, and to subsequently fill out all the entries in the matrix (as described in "Dynamic Authentication based on Public-Key Cryptography"). The paying device then computes the check public key by compressing the entries in the top row, and provides the public key to the interposed computer. Note that only the paying device knows the secret key (the entries in the matrix).

Step 3. The interposed computer now offers a blinded form of the public key to the bank, in order to obtain a blind RSA signature on it. The paying device may not develop the blinding factor itself, since in Step 6 the bank could then learn the check public key; on the other hand, the interposed computer may not determine the blinding factor itself, because that would enable it to have the bank sign any information, including public keys for which it knows the check secret key itself. Therefore, the interposed computer and the paying device develop the blinding factor in a mutually random manner, using the method of Step 1. The paying device afterwards opens its commit to show that it behaved properly.

Step 4. Although the interposed computer is assured that the blinding factor is really blind, it has no assurance that the paying device in Step 1 actually used its contribution in the formation of the check secret key. This is of importance because in the payment protocol (Step 7) the paying device will reveal matrix entries, and otherwise there could be a great deal of outflow. Hence at this point the interposed computer can request the paying device to open its commit of Step 1; the decision of whether to make the request or not is made by flipping a (not necessarily unbiased) coin. Because with the opening of the commit the interposed computer learns all the information needed to compute the matrix entry and thus the check secret key, the paying device in this case will not further assist the interposed computer, to prevent it from obtaining a blind RSA signature of the bank; instead, it halts the current protocol execution, and a new protocol execution must be started at Step 1.

Step 5. In case the interposed computer has not requested opening of the commit in Step 4, the protocol execution continues. Because the bank must make sure that it blindly signs only check public keys for which the interposed computer does not know the check secret key, the paying device computes a MAC for the blinded public key (using a secret key known also to the bank; this can be a diversified key) and provides it to the interposed computer. (Alternatively, it could compute a digital signature, but this requires greater computing power.)

Step 6. The interposed computer sends the blinded public key and the MAC to the bank, in the withdrawal protocol. The bank uses the secret key of the paying device to verify the MAC and, if it is correct, returns its RSA root of the blinded public key. The interposed computer removes the blinding factor and verifies the result, as described in "Blinding". The result of the actions up to this point is that the paying device and the interposed computer have together obtained a blank check triple, with the secret key being known only to the paying device.

Step 7. To make a payment with this blank check, the interposed computer passes the challenge message of the receiving device on to the paying device. The paying device reveals to the interposed computer the proper entries of the secret key, as described in "Dynamic Authentication based on Public-Key Cryptography". Upon verifying that these are correct (to prevent outflow), the interposed computing device passes them on to the receiving device.

This protocol has many shortcomings. A first shortcoming is that the paying device needs to perform a public-key cryptographic operation. Namely, in Step 5 it needs to authenticate the blinded public key, and hence must be able to compute the blinded public key or at least verify its correct formation. As shown in [1], probabilistic verification can be used to alleviate the task for the paying device. To this end, the paying device verifies a modularly reduced version of the blinding factor, using a secret prime modulus that has been installed by the bank during initialization of the paying device. This modulus can be chosen fairly small for a practical security level (typically eight bytes suffices), but its secrecy is of utmost importance and hence each paying device is preferably assigned a unique prime. The downside of this method is that it causes a lot of overhead communication and computation between the paying device and the interposed computer.

Another shortcoming is that the MAC of the paying device can contain outflow: any covert message (sufficiently short in comparison to the size of the MAC, for security) could be exclusive-ored into the MAC by the paying device, and extracted by the bank by removing the MAC. This particular problem can be overcome by a minor variation of Step 6, as described in [17]: instead of having the interposed computer pass on the MAC in Step 6, the bank computes it by itself and returns, say, the bit-wise exclusive-or of its RSA root and an expanded form of the MAC (for example, the expansion can be the symmetric encryption of the blinded public key, using the MAC as the encryption key). Only in case the interposed computer has received the MAC from the paying device can it remove the expanded MAC and extract the check certificate.

Yet another shortcoming is in the manner in which the interposed computer in Step 4 verifies that the secret key, and hence the signature at payment time, cannot contain outflow: verification takes place using a cut-and-choose strategy, much like with the cut-and-choose approach to one-show blinding. Hence the probability that cheating is detected increases only linearly with the average number of requests for commit opening.

Furthermore, the commit function in Step 1 could have a trapdoor, known to the paying device, in case of which it could always perform the opening in Step 5 correctly and still cause outflow. These problems can be overcome by the more drastic change of using a number-theoretic signature scheme to define the (secret key, public key) pair. In fact, this is the only difference of the protocols in [16, 20] (neither of which deals with the specific problem of electronic cash design) in comparison to the above protocol of Chaum [17]. As an example, consider using the Schnorr signature scheme: in Step 1, the paying device would select a random $x_1 \in \mathbb{Z}_q$ and send $h_1 := g^{x_1} \bmod p$ to the interposed computer; the interposed computer would return a random $x_2 \in \mathbb{Z}_q$, and the paying device would use $h := g^{x_1+x_2} \bmod p$ as the public key. The interposed computer could then check the correctness by verifying that $h = h_1 g^{x_2} \bmod p$, without needing to know the contribution x_1 of the paying device. Of course, the drawback of this approach is the much greater computational burden for the paying device; it now definitely needs to have a cryptographic co-processor in order to complete its computations in reasonable time.

Another problem that cannot be overcome is that the bank can retroactively trace all payments once the paying device is returned to the bank, if only the paying device stores the random numbers developed in Step 1 or, more efficiently, a few bytes of the random challenges received in Step 7; the bank can then match this information against the deposited information. More generally, the bank at issuing time cannot encode additional information, such as a check spending limit or (in case the triple would serve as a coin) a denomination, without the paying device needing to know it: such information can be encoded only by the choice of the encryption exponent and/or the RSA modulus used by the bank, and the paying device for the purpose of Step 5 needs to know both.

Still another problem is that the interposed computer in Step 7 cannot prevent the receiving device from encoding inflow into the fresh part of its challenge message, unless this is generated in a mutually agreed manner.

However, by far the most serious shortcoming and criticism of the above protocol is that withdrawn triples can be spent many times in an untraceable manner in case the paying device is compromised, because check triples are completely blinded. Incorporating the cut-and-choose blinding technique into

the above protocol would make the workload for withdrawing a check triple unacceptable: the paying device would have to assist in the construction of all blinded candidates, would have to authenticate all of them, and would have to assist in the opening of half of the candidates.

In [5, 6] techniques are introduced that are extensions of the restrictive blinding technique and the showing technique described in "One-Show Blinding." These techniques do not suffer from any of the above shortcomings. Specifically, they enable the paying device and the interposed computer to obtain a triple (secret key, public key, certificate), by performing a restrictive blind issuing protocol with the bank, in such a manner that

- The public key and the certificate can be blinded efficiently by the interposed computer, because it may determine the blinding factors by itself. In fact, the paying device need not take part in the withdrawal protocol at all, so that its holder can leave it in a safe place;
- The paying device assists in making a payment by providing a response to a challenge of the interposed computer. The only computational task that the paying device has to perform is to compute responses, and this can be done so rapidly that the paying device need merely have a simple 8-bit smart card microprocessor (in particular, payments can easily be made using several coins instead of a single check);
- Part of the secret key of each triple is a unique identifier of the paying device, and the paying device does not need to store dynamically any information from the triple: all it ever needs to do is increment sequence numbers and compute responses;
- Inflow and outflow in the withdrawal and in the payment protocol are actively prevented by the interposed computer, by blinding all communication between the paying device and the outside world on the flight (instead of using passive prevention by means of cut-and-choose verification). Moreover, this can be done at virtually no computational cost;
- Even in case the paying device stores all the challenges it receives during payments (this is all it can learn during the time it is held by the account holder), and is returned afterwards to the bank, the stored information cannot be used by the bank to retroactively trace payments by matching it against deposit information; the view of the paying device in the payment protocol is statistically uncorrelated to the view of the receiving device, at least for all receiving device views involving the same number of coins for each denomination. Moreover, by allowing the storage space allocated by the paying device to grow linearly in the number of coins that are withdrawn, it can even be ensured that the paying device cannot learn the denominations of the coins that are spent.

In Section 44.4 an example is described of a practical electronic cash system that is based on these techniques.

One-Sided vs. Two-Sided Untraceability

The one-show blinding and interposed-computer techniques suffice to offer payer untraceability. Even though payers are untraceable, accounts are all named and hence compliance with existing tax regulations is maintained. Since payees are known to the bank, payments are only *one-sided untraceable,* in the sense that payers can trace their own payments (with the help of the bank). Namely, coins and checks are deposited to named accounts, and can be recognized by their payers. Moreover, by disclosing the blinding factors used to withdraw their coins and checks, payers can provide incontestable proofs of payment. One-sided untraceability makes electronic cash unattractive for criminal uses: although a money launderer or a bribed user may readily accept criminal money at one particular moment, he may not be willing to trust the payer to not give evidence against him at a later stage, and victims of extortion have no reason to withhold evidence at all.

However, it is not sufficient for an electronic cash system to support one-sided untraceability. Indeed, one-sided untraceability can always be converted into *two-sided untraceability* (also called payee untrace-

ability), unless special measures are taken. Namely, an account holder can make a payment to a payee by withdrawing electronic coins or checks using blinded candidates that are provided by the payee; the account holder then merely acts as an intermediary between his account and the payee, providing one-time account access to the payee, and cannot learn what coin or check the payee ends up with. To this end, an extortioner may force the account holder to run modified software on his interposed computer, or a willing account holder may provide two-sided untraceability as a service for money laundering purposes. Because proximity of the payer and the payee is not required at any time the payee can remain anonymous throughout (for example, on the Internet all communication can be through spoofed IP addresses or anonymous remailers). In the case of fully blinded two-part coins, or three-part coins and checks, the payee can subsequently at his leisure deposit the coins or checks to his own account, by making a payment to himself; nobody, including the payer, can trace the payment.

To counter this strategy, electronic coins are best implemented in the three-part form, (secret key, public key, certificate), with the secret key containing a unique secret identifier known only to the paying device of the account holder. This is easily achieved by using the restrictive blinding technique. In that case the above strategy enables the other party to receive the public key and the certificate of a blinded coin or check, but not the required secret key. Nevertheless, it may still be possible to use this pair, by letting the paying device of the account holder cooperate also when spending the coin or check, using further active blinding to maintain payee untraceability. By designing the cash system so that the paying device responds only when provided with a challenge message that contains a payee (account) identifier, the untraceable payee cannot make a payment to himself without identifying his identifier to the paying device of the account holder (and hence to the interposed computer of the account holder, if present). Therefore, to maintain payee untraceability, the payee can use the electronic cash only to make a payment to another party, in return for goods or services, which makes the whole idea of conversion to two-sided untraceability pointless, because the account holder might as well have paid the other party directly; moreover, the third party may be of help in tracing.

Note that this measure to prevent two-sided untraceability conversion conflicts with measures to ensure that any information stored by the paying device cannot be used by the bank to retroactively trace payments upon return of the paying device; the (paying device of the) account holder learns the account identifier of the payee. More generally, if the views of the paying device and the payee are statistically uncorrelated, then two-sided untraceability conversion can always be accomplished. For a large-scale electronic cash implementation, it may hence be desirable to prevent only inflow and outflow, while ensuring that the views of the paying device and the receiving device are strongly correlated.

An attempt to circumvent this conflict might be to require the account holder at withdrawal time to demonstrate (in zero-knowledge or otherwise) that its blinding factors have been derived from a secret key, which corresponds to a registered public key of the account holder, according to a predetermined method. However, in the known blind and one-show blind signature issuing protocols this can be achieved only by using completely impractical general multiparty computations or inefficient cut-and-choose verification by the bank. Another serious problem, which moreover is inherent, is that privacy of payments can only be computational: with sufficient computing power the bank can compute the blinding factors by itself, for example by computing the secret key of the account holder.

44.4 An Example Electronic Cash System

In this section we describe an example electronic cash system, to illustrate many of the foregoing design principles and techniques. For the sake of brevity, the bank is denoted by $\mathcal{B}$, the account holder by $\mathcal{A}$, his (tamper-resistant) paying device by $\mathcal{P}$, his interposed computer by $\mathcal{C}$, and the receiving device by $\mathcal{R}$. Receiving devices need not be tamper-resistant. For the sake of clarity of the description, we do not incorporate fault-tolerance measures.

Bank Set-Up

$\mathcal{B}$ generates a secret key $(x_1,\ x_2,\ x_3)$ and a corresponding public key:

$$p,\ q,\ \mathcal{H}(\cdot),\ (g_0,\ g_1,\ g_2,\ g_3)\ . \tag{44.19}$$

This key pair will be used for the computation of coin certificates in the restrictive blind coin issuing protocol. As in the restrictive blind issuing protocol in "One-Show Blinding," p and q are primes such that q divedes $p-1$; $\mathcal{H}(\cdot)$ is a collision-intractable hash function; x_1, x_2, x_3 are three random numbers from $\mathbb{Z}_q$; g_0 is an element of order q in $\mathbb{Z}_p^*$; and, $g_1 = g_0^{x_1} \bmod p$, $g_2 = g_0^{x_2} \bmod p$ and $g_3 = g_0^{x_3} \bmod p$.

For concreteness, it is assumed that $\mathcal{B}$ issues electronic coins of denomination $2^{\texttt{index}}$, for $\texttt{index} \in \{0, \ldots, l\}$, for some limit l and measured in some appropriate unit (for example, dollar cents); alternatively, any other mapping from the set of valid index numbers to the set of valid denominations may be used.

Opening an Account

When $\mathcal{A}$ opens an account, $\mathcal{B}$ provides $\mathcal{A}$ with a tamper-resistant paying device $\mathcal{P}$. $\mathcal{P}$ holds a randomly chosen secret key $I \in \mathbb{Z}_q$, installed by $\mathcal{B}$ and serving as an identifier of $\mathcal{P}$; $\mathcal{B}$ will be able to trace double-spending to the compromised paying device by computing I. The corresponding public key $h := g_2^I \bmod p$ is made available to $\mathcal{C}$.

$\mathcal{P}$ uses a pseudo-random number generator, PRGN$\mathcal{P}$, that takes as inputs triples of the form

$$\text{(seed, index value, sequence number)}\ . \tag{44.20}$$

The seed is a random secret of $\mathcal{P}$, known also to $\mathcal{B}$; for efficiency, we take it to be the same as I. The design of PRGN$\mathcal{P}$ is such that a simple 8-bit smart card micro-processor can evaluate it within a few hundredths of a second; one can build it, for example, from a one-way hash function with pseudo-random properties, such as SHA.

For each coin denomination, $\mathcal{P}$ keeps track of a sequence number, seqnum$\mathcal{P}$(index), which it increments each time when it assists in spending a coin of that denomination. The pseudo-random number output by PRGN$\mathcal{P}$, on input (I, index, seqnum$\mathcal{P}$(index)), is $\mathcal{P}$'s contribution to the secret key of the coin of that denomination and sequence number; $\mathcal{B}$ encodes it into the coin at withdrawal time (note that $\mathcal{B}$ can compute the pseudo-random numbers of $\mathcal{P}$), and it is needed again to spend the coin.

$\mathcal{C}$ for each coin denomination keeps track of its own sequence number, seqnum$\mathcal{C}$(index). It increments this number upon each coin withdrawal. $\mathcal{C}$ also keeps track of a copy of each seqnum$\mathcal{P}$(index), and in any case can always request the current values from $\mathcal{P}$, to stay in synch with $\mathcal{P}$.

Initially, seqnum$\mathcal{P}$(index) and seqnum$\mathcal{C}$ (index) are set to zero, for each coin denomination. At any moment in time, if $\mathcal{C}$ has properly performed the withdrawal protocol, $\mathcal{C}$ and $\mathcal{P}$ together hold seqnum$\mathcal{C}$(index) - seqnum$\mathcal{P}$ (index) coins of denomination $2^{\texttt{index}}$. In practice, one can use four bytes of storage space for each sequence number, three of which serve to store the actual number and one containing an error-correcting code.

$\mathcal{C}$ also generates its own key pair for message signing purposes, and $\mathcal{B}$ registers $\mathcal{C}$'s message public key with the account.

Coin Withdrawal Protocol

To withdraw an electronic coin with denomination $2^{\texttt{index}}$, $\mathcal{C}$ and $\mathcal{B}$ perform the following withdrawal protocol:

Step 1. $\mathcal{C}$ sends to $\mathcal{B}$ a digitally signed withdrawal request, specifying the account of $\mathcal{A}$, index and seqnum$\mathcal{C}$(index). To prevent replay of withdrawal requests, the withdrawal request also contains, say, an account access counter.

Step 2. If the signature on the withdrawal request is correct, $\mathcal{B}$ generates two random numbers, $w_0, v \in \mathbb{Z}_q$. The number v is computed in a pseudo-random manner, as follows:

$$v := \mathrm{PRGN}\mathcal{P}(I,\ \mathtt{index},\ \mathtt{seqnum}\mathcal{C}(\mathtt{index}))\ . \tag{44.21}$$

$\mathcal{B}$ then computes

$$a_0 := g_0^{w_0} \bmod p \quad \text{and} \quad u := g_2^{v} \bmod p\ , \tag{44.22}$$

and sends (a_0, u) to $\mathcal{C}$.

Step 3. $\mathcal{C}$ generates a random number $\alpha_1 \in \mathbb{Z}_q^*$ and five random numbers $\alpha_2, \alpha_3, \alpha_4, \alpha_5, \alpha_6 \in \mathbb{Z}_q$. $\mathcal{C}$ then computes

$$h' := (g_1\, h\, g_3^{\mathtt{index}})^{\alpha_1} \bmod p\ , \tag{44.23}$$

$$c := \mathcal{H}(h',\ u\,(h')^{\alpha_4}\, g_2^{\alpha_5}\, h^{\alpha_6} \bmod p,\ a_0\, g_0^{\alpha_2}\, (g_1\, h\, g_3^{\mathtt{index}})^{\alpha_3} \bmod p) \tag{44.24}$$

and

$$c_0 := c - \alpha_2 \bmod q\ . \tag{44.25}$$

$\mathcal{C}$ then sends c_0 to $\mathcal{B}$. (Note that $\mathcal{C}$ can perform the bulk of the workload in a precomputation phase, before connecting to $\mathcal{B}$; virtually the only on-line computations are the multiplications by a_0 and u.)

Step 4. $\mathcal{B}$ computes

$$r_0 := (x_1 + x_2\, I + x_3\, \mathtt{index})^{-1}\ (w_0 - c_0) \bmod q\ , \tag{44.26}$$

charges the account of $\mathcal{A}$ by $2^{\mathtt{index}}$ units, and sends r_0 to $\mathcal{C}$. $\mathcal{B}$ also increases the account access counter for $\mathcal{A}$ by one.

Step 5. $\mathcal{C}$ computes

$$r := \alpha_1^{-1}\ (r_0 + \alpha_3) \bmod q\ . \tag{44.27}$$

$\mathcal{C}$ stores $(\alpha_1, \alpha_4, \alpha_5, \alpha_6)$ and (r, c) onto its "coin stack" of denomination $2^{\mathtt{index}}$ (note that these are all "small" numbers), indexed by $\mathtt{seqnum}\mathcal{C}(\mathtt{index})$, and increments $\mathtt{seqnum}\mathcal{C}(\mathtt{index})$ and its account access counter both by one.

In case $\mathcal{A}$ wants to withdraw many coins, not necessarily having the same denomination, the request in Step 1 should specify all coins requested; Steps 2 through 5, which must be performed once for each coin, can be performed in parallel for all coins. The withdrawal protocol thus consists of four message transmissions, independent of the number of coins requested.

For authentication and nonrepudiation, the messages sent in Steps 2 through 4 should also be digitally signed by the sender. The digital signature of $\mathcal{C}$ on its message in Step 3 (and details of the preceding messages) can serve to $\mathcal{B}$ as a proof that $\mathcal{C}$ has requested the withdrawal of the specified amount, and authorizes $\mathcal{B}$ to debit $\mathcal{A}$'s account. By requiring $\mathcal{B}$ to be able to resend its response(s) in Step 4 whenever requested to do so, at least until a time-out or until $\mathcal{C}$ requests withdrawal of a new set of coins (this proves that $\mathcal{C}$ has successfully completed the previous withdrawal session), it is ensured that $\mathcal{B}$ obtains $\mathcal{C}$'s debit authorization if and only if $\mathcal{C}$ is able to obtain the requested coins (possibly with the help of a judge).

In Step 5, $\mathcal{C}$ may wish to verify that

$$g_0^{c_0}\ \left(g_1\, h\, g_3^{\mathtt{index}}\right)^{r_0} = a_0 \bmod p\ . \tag{44.28}$$

For efficiency, this verification can normally be omitted. In particular, if r_0 is incorrect then $\mathcal{C}$ will notice this because $\mathcal{R}$ will not accept when the coin is spent; $\mathcal{C}$ can then use the digital signature of $\mathcal{B}$ on its message in Step 5, to demonstrate that r_0 has been formed incorrectly by $\mathcal{B}$. This requires $\mathcal{C}$ to store the digital signature of $\mathcal{B}$ on its message in Step 4 (and details of the preceding messages) at least until the withdrawn coins have been spent.

Furthermore, all messages may be encrypted, for confidentiality. This can be done using symmetric encryption, using a mutually known random session key; $\mathcal{C}$ can generate the session key in Step 1, and send it along with the withdrawal request in encrypted form (using a special encryption public key of $\mathcal{B}$). For efficiency, the RSA signature scheme may be used for message signing, the RSA encryption scheme for session key encryption, and triple-DES for message encryption.

Coin Payment Protocol

If $\texttt{seqnum}\mathcal{C}(\texttt{index})$ exceeds $\texttt{seqnum}\mathcal{P}(\texttt{index})$ then $\mathcal{C}$ and $\mathcal{P}$ together hold $\texttt{seqnum}\mathcal{C}(\texttt{index})$ - $\texttt{seqnum}\mathcal{P}(\texttt{index})$ coins of denomination $2^{\texttt{index}}$. To transfer a coin to $\mathcal{R}$, $\mathcal{C}$ and $\mathcal{R}$ perform the following payment protocol (with $\mathcal{P}$ necessarily assisting $\mathcal{C}$):

Step 1. A challenge message m is determined by $\mathcal{C}$ and $\mathcal{R}$, by concatenating an account identifier of $\mathcal{R}$, the number $\texttt{index}$, and a fresh part (such as a random number of $\mathcal{R}$, or a sufficiently accurate time/date estimate by $\mathcal{C}$ that is approved by $\mathcal{R}$). $\mathcal{C}$ retrieves $(\alpha_1, \alpha_4, \alpha_5, \alpha_6)$ and (r, c), indexed by $\texttt{seqnum}\mathcal{P}(\texttt{index})$ in its coin stack for coins of denomination $2^{\texttt{index}}$. $\mathcal{C}$ then recomputes h' from α_1 (of course, $\mathcal{C}$ could alternatively have stored h' in its coin stack at withdrawal time), and computes

$$d := \mathcal{H}(m,\ h',\ (r, c)) \quad \text{and} \quad e := d + \alpha_6 \bmod q\ . \tag{44.29}$$

$\mathcal{C}$ then sends e to $\mathcal{P}$, together with $\texttt{index}$, to indicate the denomination of the coin it wants to spend.
Step 2. $\mathcal{P}$ computes

$$y := I\, e + \text{PRGN}\mathcal{P}(I,\ \texttt{index},\ \texttt{seqnum}\mathcal{P}(\texttt{index})) \bmod q\ , \tag{44.30}$$

increments $\texttt{seqnum}\mathcal{P}(\texttt{index})$ by one, and sends y to $\mathcal{C}$.
Step 3. $\mathcal{C}$ computes

$$r_1 := y + \alpha_5 \bmod q, \quad \text{and} \quad r_2 := -\alpha_1^{-1} d + \alpha_4 \bmod q \tag{44.31}$$

and sends

$$h',\ (r, c),\ (d, r_1, r_2) \tag{44.32}$$

to $\mathcal{R}$ (and possibly also missing details of m).
Step 4. For the challenge message m approved or decided on by $\mathcal{R}$, $\mathcal{R}$ verifies that

$$c \ = \ \mathcal{H}\left(h',\ g_1^d\, g_2^{r_1}\, g_3^{d\,\texttt{index}}\, (h')^{r_2} \bmod p,\ g_0^c\, (h')^r \bmod p\right) \tag{44.33}$$

and

$$d \ = \ \mathcal{H}(m,\ h',\ (r, c))\ . \tag{44.34}$$

If the verification holds, $\mathcal{R}$ accepts the coin payment.

In Step 3, $\mathcal{C}$ may wish to verify that

$$g_2^y\, h^{-e} = u \bmod p\ . \tag{44.35}$$

For efficiency this can normally be omitted. Even if the verification is never performed, if y is incorrect then $\mathcal{R}$ will not accept the payment, while at most one bit of outflow can result (namely, whether the coin is correct or not).

In a typical scenario, this payment protocol is performed simultaneously for multiple coins of suitable denominations, in order to make up the exact amount payable. For increased efficiency, a single m and a

single d can be used for all coins. To this end, m must specify `index` for each of the coins (alternatively, and more compactly, the payable amount may be specified), and to compute d the hash function $\mathcal{H}(\cdot)$ must take as inputs m and, for each coin, $(h',\ (r, c))$ of that coin.

In Step 4, $\mathcal{R}$ can send a digitally signed receipt to $\mathcal{C}$. For confidentiality, $\mathcal{C}$ and $\mathcal{R}$ may use session key encryption. To this end, $\mathcal{C}$ can generate a random session key in Step 3, publicly encrypt it using a (certified) public key of $\mathcal{R}$, symmetrically encrypt the payment message using the session key, and send both encryptions over to $\mathcal{R}$.

Coin Deposit Protocol

To deposit the received coin at a convenient moment later on, $\mathcal{R}$ performs the following deposit protocol with $\mathcal{B}$:

Step 1. $\mathcal{R}$ sends to $\mathcal{B}$ the payment transcript:

$$h',\ (r, c),\ (d, r_1, r_2)\ ,\ m\ . \tag{44.36}$$

Step 2. $\mathcal{B}$ verifies that the purportedly fresh part of the challenge message m has not been used before by $\mathcal{R}$ (note that for this reason the use of an accurate time and date indication is preferred, since $\mathcal{B}$ then merely needs to store the most recent time and date indication of $\mathcal{R}$); this excludes double-depositing by $\mathcal{R}$. If this is the case, $\mathcal{B}$ verifies the payment transcript in the same way as specified for $\mathcal{R}$ in Step 4 of the payment protocol, reading `index` from m. If the verification holds, $\mathcal{B}$ credits the account that is indicated by the account identifier in m by $2^{\texttt{index}}$ units.

Of course, $\mathcal{R}$ can deposit multiple payment transcripts at once, and should preferably sign its deposit request in Step 1. Likewise, $\mathcal{B}$ in Step 2 should return a digitally signed statement, to inform $\mathcal{R}$ of which payment transcripts have been accepted. To hide who is depositing how much, both parties can encrypt their messages using a session key suggested by $\mathcal{R}$, which $\mathcal{R}$ sends to $\mathcal{B}$ in public-key encrypted form.

Forgery Detecting and Tracing

At a suitable moment, which might be at the time of deposit by $\mathcal{R}$ but could also be later on, $\mathcal{B}$ checks for double-spending. To this end, $\mathcal{B}$ verifies whether $\mathcal{H}(h')$ already appears in a "forgery-detect" database. If this is the case, $\mathcal{B}$ subsequently traces the physically compromised paying device, by retrieving from a "forgery-trace" database the "old" pair (d^*, r_1^*) (indexed by the hash of h' that is already in the forgery-detect database) and computing

$$\left(d - d^*\right)^{-1} \left(r_1 - r_1^*\right) \bmod q\ ; \tag{44.37}$$

this number is equal to I. In addition, $\mathcal{B}$ can blacklist $\mathcal{H}(h')$.

On the other hand, if the coin has not been double-spent, $\mathcal{B}$ stores $\mathcal{H}(h')$ in the forgery-detect database and the pair (d, r_1) in the forgery-trace database, indexed by $\mathcal{H}(h')$. The forgery-detect database can be maintained on computer hard-disks, while the forgery-trace database can be stored on tape or WORM disks.

Discussion

There are several noteworthy aspects in the design of this example cash system. Firstly, the withdrawal protocol is the restrictive blind issuing protocol discussed in "One-Show Blinding." Secondly, the interposed computer computes the blinding factors by itself, and in fact the tamper-resistant paying device

does not take part in the withdrawal protocol. When assisting in a coin payment, the paying device performs a simple computation that can be performed rapidly by a simple smart card processor. Furthermore, it does not store any part of the withdrawn coins; it merely increments coin sequence numbers that occupy preallocated storage space, which can be well below 100 bytes in a practical implementation. Hence the desired low complexity for paying devices is achieved. Thirdly, the receiving device need not be tamper-resistant, and payment transcripts can be deposited only to the account of the payee indicated in the challenge message. Finally, the active blinding by the interposed computer in the payment protocol ensures unconditional protection against inflow and outflow.

Moreover, it can be proved that even if $\mathcal{P}$ stores the challenges e received in all executions of the payment protocol performed by $\mathcal{C}$, and $\mathcal{B}$ analyzes all information stored by $\mathcal{P}$ upon its return, the payments of $\mathcal{C}$ still cannot be traced, regardless of the strategy followed by $\mathcal{B}$, $\mathcal{P}$ and all receiving devices, and regardless of their computing power. Of course, this untraceability holds only amongst all payments made with the same number of coins and of the same denominations, because $\mathcal{P}$ does learn the coin denominations involved in a payment. This can be improved on, by using a single list of sequence number instead of one for each denomination, but then the amount of data that must be stored by the paying device grows linearly instead of logarithmically with the number of coins that have been withdrawn. Namely, at withdrawal time it cannot be predicted in which order coin denominations will be addressed at payment time, and in the worst case the paying device must hop almost randomly through the list of sequence numbers to select coins of the appropriate denomination, keeping track of all the sequence numbers used thus far.

In order to keep the size of the forgery-detect and forgery-trace databases manageable, $\mathcal{B}$ should regularly change its own secret key for certifying coins. Once coins have expired for deposit, $\mathcal{B}$ can erase the databases or archive them. The use of coin expiration dates also helps to contain the financial damage that can be done by an attacker who gets hold of $\mathcal{B}$'s secret certification key. In fact, for this purpose it is highly desirable that $\mathcal{B}$ can at any instant declare the current coin version invalid, and start issuing coins using a new certification key. To make this work smoothly in practice, coin versions should be given a separate expiration dates for at least the withdrawal and the deposit protocols.

In view of the possibility of a crash, loss or theft of $\mathcal{C}$ and/or $\mathcal{P}$, $\mathcal{C}$ should generate backup copies of all its coins. $\mathcal{C}$ can make a backup each time when it withdraws new coins, and any coins that have been spent since the last backup can be removed or overwritten. Specifically, for each withdrawn coin that has not yet been spent, a backup entry is kept of the form

$$\texttt{index},\ \alpha_1,\ b, (r, c)\ . \tag{44.38}$$

Here, b denotes $u\,(h')^{\alpha_4}\,g_2^{\alpha_5}\,h^{\alpha_6} \bmod p$; this number is computed by $\mathcal{C}$ in Step 3 of the withdrawal protocol, for inclusion in the hash function. (To reduce storage space, $\mathcal{H}(b)$ can be stored instead of b itself; the definition of the coin certificate, (r, c), must then be modified correspondingly.) In case $\mathcal{A}$ wants to recover, $\mathcal{C}$ reads the backup entries, and sends them to $\mathcal{B}$. Of course, in case $\mathcal{C}$ was involved in the crash, loss or theft, this requires $\mathcal{A}$ to first obtain a new $\mathcal{C}$ and/or to reinstall $\mathcal{C}$'s software. For each coin, the provided entry information is sufficient for $\mathcal{B}$ to reconstruct h' and to check (r, c), and in particular to verify that the information indeed specifies a coin it had issued to $\mathcal{A}$. $\mathcal{B}$ reimburses $\mathcal{A}$ for all the coins that have not been spent, to which end it matches h' against the forgery-detect database. In case the paying device has been reported lost or stolen, $\mathcal{B}$ should not unconditionally reimburse $\mathcal{A}$ until the expiry date for payment or deposit of the current coin version, to prevent abuse of the recovery procedure for the purpose of double-spending. Note that the recovery method is independent of the manner in which blinding factors are computed, and of whether or not the account holder has a paying device at his disposal at the time of recovery (issuing a new one requires a physically secure channel, while $\mathcal{A}$ may want to suspend his account). Moreover, the backup information cannot be used by a thief to spend the coins, since part of the secret key of each coin triple is not stored. Finally, $\mathcal{B}$ can incontestably prove to $\mathcal{A}$ that a coin offered for recovery appears already in its deposit database by showing the signature (d, r_1, r_2), which it could not have made by itself. To this end it also needs to store r_2 at deposit time.

For cross-platform portability, the same paying device can be used in combination with different user-controlled computer devices. Of course, this requires a mechanism for the various user-controlled computers to exchange coin information, since coins withdrawn using one computer may need to be spent using another.

In line with our earlier discussions, a minor variation allows the protocols to be used for electronic checks, which can be spent for any amount up to a predetermined maximum. A special value of `index` can be used to denote checks (alternatively, several extra values are defined, each specifying a different maximum spending limit). The only required change to the protocols is that the challenge message m in the payment protocol must now specify in addition the amount for which the check is filled out, and d must be computed by $\mathcal{P}$ itself in Step 2; correspondingly, there is no use for α_6 anymore, and $\mathcal{C}$ in Step 1 of the payment protocol must provide $\mathcal{P}$ with m, h' and (r, c), instead of with e. (Alternatively, $\mathcal{C}$ provides m and a hash of $(h', (r, c))$ to $\mathcal{P}$, and $\mathcal{P}$ computes d by hashing these two numbers. This approach is especially convenient when using a single m and d for multiple coins, because $\mathcal{P}$ can then be given m and a hash of all $(h', (r, c))$, and these two numbers then form the input to $\mathcal{H}(\cdot)$ for computing d; of course, the verification by $\mathcal{R}$ and $\mathcal{B}$ must be modified correspondingly. Furthermore, parts of m may be hashed, perhaps even with a random salt, to prevent $\mathcal{P}$ from learning or recognizing them.) Before sending out y in Step 2, $\mathcal{P}$ must decrement its register value by the amount for which the check is filled out. Note that now an additional protocol is needed, for withdrawing electronic cash; what we called the withdrawal protocol for coins is now the issuing protocols for blank checks. This can easily be constructed using the conventional authentication techniques described in "Authentication Techniques."

The check variation has the drawback that $\mathcal{B}$ can retroactively trace all payments of $\mathcal{C}$ once $\mathcal{P}$ is returned to $\mathcal{B}$, if only $\mathcal{P}$ stores for each execution of the payment protocol m or a relevant part of it. On the other hand, having $\mathcal{P}$ compute d by itself has the advantage for law enforcement that one-sided untraceability cannot be converted into two-sided untraceability, as discussed in "One-Sided Versus Two-Sided Untraceability." This adjustment can also be applied to the described coin protocols.

A more serious drawback of the check variation, which as we have seen is unavoidable, is that by physically extracting the secret key I of $\mathcal{P}$, an attacker can introduce counterfeit without detection by the bank. The following trade-off is believed to be of practical interest: the bank issues coins as well as checks (in both cases preventing conversion to two-sided untraceability), and paying devices use checks only to pay fractional amounts that would otherwise require too many coins. The bank specifies a low spending limit for checks, and monitors the number of checks withdrawn by each of its account holders in order to limit excessive check withdrawal. This ensures that an attacker cannot make a significant profit without being exposed.

44.5 Summary and Research Issues

As we have seen in this chapter, techniques for designing electronic cash systems vary widely and range from elementary to complex. Assuming that it is infeasible for attackers to compromise tamper-resistance, register-based electronic cash is preferable over electronic coins for reason of efficiency. We have seen that payment authentication must take place on the basis of dynamic authentication, in the form of challenge-response protocols, to prevent replay attacks. Authentication can be performed using symmetric cryptography, whereby MACs are communicated, or by using public-key cryptography. The former approach is (in general) much more efficient than the latter, but the presence of system-wide secrets in receiving devices is inherent to this approach, and hence tamper-proofness of receiving devices is critical. In the latter approach, it is not of concern to the bank whether receiving devices are tamper-resistant, if only electronic cash is issued in the form of coins or checks in the three-part form. This can be accomplished in one of the following two ways: either paying devices withdraw electronic coins, or they hold register-based cash and digitally sign the payable amount using a blank check. When tamper-resistance can be relied upon, achieving security against forgery is mainly a matter of designing cash transfer protocols that resist cryptanalysis by wire-tappers.

We have also seen that an important aspect of system design is to ensure security for the bank, under the assumption that secrets in tamper-resistant devices can be extracted and registers bypassed. It is here that crucial design decisions must be made. When electronic cash is represented in the form of register-based cash, a forgery can be detected by the bank only by monitoring how much cash ought to be present in each paying device at various moments in time. This requires receiving devices to deposit transaction transcripts that reveal at least a paying device ID and the transferred amount. In case receiving devices are tamper-proof, the bank can trust receiving devices for performing truthful depositing of transaction logs. In case receiving devices are not tamper-resistant, or at least their tamper-resistance is not relied on, truthful depositing of transaction details can be ensured only by issuing electronic cash in the three-part form (coins or blank checks).

Because such monitoring is in conflict with privacy of payments, we have also discussed techniques for incorporating untraceability of payments. The straightforward approaches of relaxed monitoring, anonymous accounts, and anonymously issued paying devices were all seen to offer only a very weak form of privacy (all payments are linkable, and trust in the bank or receiving devices is required), and moreover inevitably cause a trade-off with security for the bank. Much better are cryptographic techniques based on the paradigm of blinding. When electronic coins or check triples are withdrawn by means of a basic blind signature protocol, all information can be blinded by the withdrawing party, and forgery can at best be detected by the bank (untraceability is perfect even for attackers who manage to forge electronic cash). An important improvement is one-show blinding, for which two techniques exist: cut-and-choose blinding and restrictive blinding. The former is very expensive, and for practical purposes only the latter is acceptable.

The one-show blinding paradigm is only half of the work that is needed to design an untraceable electronic cash system, because paying devices that are tamper-resistant cannot be guaranteed to compute random blinding factors, or to otherwise follow the protocols; and in case paying devices are not tamper-resistant but user-controlled, there is no prior restraint of double-spending. As we have seen, the other half of the work can be accomplished by having the account holder interpose a computer of his own between his paying device and any outside devices, to ensure that the paying device cannot leak information that helps the bank to trace his payments. The cash system should then be designed such that the interposed computer can prevent inflow and outflow by blinding on the flight all communication between the paying device and the outside world, and during withdrawal can compute the blinding factors either by itself or randomize them.

By ensuring that paying devices are provided with the challenge information that needs to be signed at payment time, one-sided untraceability cannot be converted into two-sided untraceability, thus making electronic cash unattractive for criminal uses such as money laundering, bribery, and extortion. If this property is not deemed relevant, it can even be ensured that paying devices cannot learn information that enables the bank to trace payments when analyzing their contents upon return. Finally, we have seen a detailed example of a practical electronic cash system, based on the discussed principles and techniques.

An important area for further research is to rigorously prove the cryptographic security of practical electronic cash systems, by proving all manner of attacks as hard as breaking well-understood cryptographic primitives. For a start in this direction, see [7].

44.6 Defining Terms

Blinding: A paradigm according to which a receiver in an execution of a protocol obtains digitally signed information that remains hidden from the issuer. Also refers to the cryptographic actions by a party interposed between two other parties in a cryptographic protocol, to destroy subliminal channels.

Cut-and-choose blinding: A technique for designing an issuing protocol for the one-show blinding paradigm. A great many basic blind issuing protocols are performed in parallel, and the signer

completes the blind signature issuing protocol only for some of these, after having verified for the rest that the blinded candidates contain an identifier.

Diversified key: A secret key that is computed by hashing at least a master secret key and an identifier, using a one-way hash function.

Dynamic authentication: A method for a device to prove its authenticity, in such a manner that replay attacks have negligible or zero probability of success.

Electronic check: A method for transferring electronic cash. A blank check is a triple (secret key, public key, certificate), issued by the bank to a paying device that holds cash in register-based form. At payment time the check is filled out by the paying device for any amount up to a predetermined maximum amount, by signing the amount and subtracting it from its internal cash balance. Blank checks are not prepaid, because they do not carry value.

Electronic coin: A method for representing electronic cash. A coin is a prepaid public-key cryptographic token, digitally signed by the bank, to which a fixed value and currency are assigned at issuing time. Coins can be either in the two-part form, (message, signature), or in the three-part form, (secret key, public key, certificate).

Inflow: Covert information that is communicated by an outside device to a paying device, through a subliminal channel in a system protocol.

One-show blinding: A cryptographic paradigm, requiring the design of a signature issuing protocol and a corresponding signature showing protocol, such that showing an obtained signature once is untraceable, while showing it twice allows a built-in identifier to be computed.

Outflow: Covert information that is communicated by a paying device to an outside device, through a subliminal channel in a system protocol.

Register-based cash: A method for representing electronic cash. The amount of cash held by a tamper-resistant device is indicated by the value of a counter, maintained in a register in a chip.

Replay attack: An attack whereby information revealed by a party when proving its authenticity is reused to pass a subsequent authenticity test.

Restrictive blinding: A technique for designing an issuing protocol for the one-show blinding paradigm. The issuer can issue a triple (secret key, public key, certificate) in such a manner that the receiver can blind the public key and the certificate, but not a nontrivial blinding-invariant part of the secret key; in this part one or more identifiers can be encoded.

Static authentication: A method whereby one device proves its authenticity to another by always revealing the same predetermined secret, such as a password or an identifier. Not secure against replay attacks.

Subliminal channel: A channel present in a cryptographic protocol, by means of which a party can secretly communicate information to another party, in a manner unrecognizable by interposed parties.

Untraceability: In an untraceable electronic cash system, payments cannot be traced to the payer by examining information revealed through system protocols, and payments by the same payer are unlinkable. Untraceable cash systems can be designed using cryptographic blinding techniques.

References

[1] Bos, J., Verification of RSA Computations on a Small Computer. *Practical Privacy,* Ph.D. Thesis, ISBN 90-6196-405-9, 1992.

[2] Bos, J. and Chaum, D., SmartCash: A Practical Electronic Payment System. *Centrum voor Wiskunde en Informatica technical reports,* CS-R9035, 1990.
[3] Bleichenbacher, D. and Maurer, U., Directed Acyclic Graphs, One-way Functions and Digital Signatures. *Advances in Cryptology—CRYPTO '94.* Y.G. Desmedt, Ed., *Lecture Notes in Computer Science,* Vol. 839, Springer-Verlag, 75–82, 1994.
[4] Boneh, D., DeMillo, R., and Lipton, R., On the Importance of Checking Computations for Faults. *Advances in Cryptology—EUROCRYPT'97,* W. Fumy, Ed., *Lecture Notes in Computer Science,* Vol. 233, Springer-Verlag, 37–51, 1997.
[5] Brands, S., Untraceable off-line cash in wallets with observers. *Advances in Cryptology—CRYPTO '93,* D.R. Stinson, Ed., *Lecture Notes in Computer Science,* Vol. 773, Springer-Verlag, 302–318, 1994.
[6] Brands, S., Electronic Cash on the Internet. *Proceedings of the ISOC Symposium on Network and Distributed System Security,* San Diego, CA, Febr. 16-17, 64–84, 1995.
[7] Brands, S., Off-Line Electronic Cash Based on Secret-Key Certificates. *Proceedings of the Second International Symposium of Latin American Theoretical Informatics,* R. Baeza-Yates, E. Goles, P.V. Goblete, Eds., *Lecture Notes in Computer Science,* Vol. 911, Springer-Verlag, 131–166, 1995.
[8] Brands, S., Secret-Key Certificates. *Centrum voor Wiskunde en Informatica technical report,* CS-R9510, 1995.
[9] Brands, S., Restrictive Blinding of Secret-Key Certificates. *Advances in Cryptology—EUROCRYPT '95.* L.C. Guillou and J.-J. Quisquater, Eds., *Lecture Notes in Computer Science,* Vol. 921, Springer-Verlag, 231–247, 1995.
[10] Brands, S., Secret-Key Certificates (Continued). *Centrum voor Wiskunde en Informatica technical report,* Report CS-R9555, 1995.
[11] Brands, S., Rapid Demonstration of Linear Relations Connected by Boolean Operators. *Advances in Cryptology—EUROCRYPT'97,* W. Fumy, Ed., *Lecture Notes in Computer Science,* Vol. 1233, Springer-Verlag, 318–333, 1997.
[12] Brands, S. and Chaum, D., Distance Bounding. *Advances in Cryptology—EUROCRYPT '93.* T. Helleseth, Ed., *Lecture Notes in Computer Science,* Vol. 765, Springer-Verlag, 344–359, 1994.
[13] Brickell, E., Gemmell, P., and Kravitz, D., Trustee-Based Tracing Extensions to Anonymous Cash and the Making of Anonymous Change. *Proceedings of the 6th Annual Symposium on Discrete Algorithms,* 457–466, 1995.
[14] Camenisch, J., Maurer, U., and Stadler, M., Digital Payment Systems with Passive Anonymity-Revoking Trustees. *Proceedings of Computer Security—ESORICS '96, Lecture Notes in Computer Science,* Vol. 1146, Springer-Verlag, 31–43, 1996.
[15] Chaum, D., Blind Signatures for Untraceable Payments. *Advances in Cryptology—CRYPTO '82,* R.L. Rivest, A. Sherman and D. Chaum, Eds., Vol. 0, Plenum Press, 199–203, 1983.
[16] Chaum, D., Card-Computer Moderated Systems. Patent no. US 4,926,480, 1988.
[17] Chaum, D., Optionally moderated transaction systems. Patent no. EP 0 439 847 A1, 1990.
[18] Chaum, D., Fiat, A., and Naor, M., Untraceable Electronic Cash. *Advances in Cryptology—CRYPTO '88.* S. Goldwasser, Ed., *Lecture Notes in Computer Science,* Vol. 403, Springer-Verlag, 319–327, 1988.
[19] Chaum, D. and Pedersen, T., Transferred Cash Grows in Size. *Advances in Cryptology—Proceedings of EUROCRYPT '92,* R.A. Rueppel, Ed., *Lecture Notes in Computer Science,* Vol. 658, Springer-Verlag, 390–407, 1993.
[20] Chaum, D. and Pedersen, T., Wallet Databases with Observers. *Advances in Cryptology—CRYPTO '92,* Ernest F. Brickell, Ed., *Lecture Notes in Computer Science,* Vol. 740, Springer-Verlag, 89–105, 1993.
[21] Diffie, W. and Hellman, M., New Directions in Cryptography. *IEEE Transactions on Information Theory,* Vol. IT-22, 644–654, 1976.

[22] Eng, T. and Okamoto, T., Single-Term Divisible Electronic Coins. *Advances in Cryptology—Proceedings of EUROCRYPT '94,* Alfredo De Santis, Ed., *Lecture Notes in Computer Science,* Vol. 950, Springer-Verlag, 306–319, 1994.
[23] Even, S., Goldreich, O., and Micali, S., On-Line/Off-Line Digital Signatures. *J. Cryptology,* Vol. 9, No. 1, 35–67, 1996.
[24] Even, S., Goldreich, O., and Yacobi, Y., Electronic Wallet. *Advances in Cryptology—Proceedings of Crypto '83,* D. Chaum, Ed., Vol. 0, Plenum Press, 383–386, 1984.
[25] Froomkin, M., Flood Control on the Information Ocean: Living With Anonymity, Digital Cash and Distributed Databases. *15 Pitt. J. Law & Commerce,* No. 395, 1996.
[26] Gray, J. and Reuter, A., *Transaction Processing: Concepts and Techniques,* Morgan Kaufman, San Francisco, 1993.
[27] IEEE Spectrum, *Technology and the Electronic Economy,* (Special issue on electronic money.) Febr. 1997.
[28] Jones, D., *Mondex: A House of Smart-Cards?; With e-cash, privacy is illusory and security is questionable,* The Convergence, jul. 1997.
[29] Lamport, L., Constructing Digital Signatures from a One Way Function. *SRI International Report,* No. CSL–98, 1979.
[30] Law, L., Sabett, S., and Solinas, J., How to Make a Mint: the Cryptography of Anonymous Electronic Cash. National Security Agency, Office of Information Security Research and Technology, Cryptology Division, June 14, 1996.
[31] Merkle, R., Matrix Digital Signature For Use with the Data Encryption Algorithm. *IBM Tech. Discl. Bulletin,* Vol. 28, No. 2, 603–604, 1985.
[32] Merkle, R., A Digital Signature Based on a Conventional Encryption Function. *Advances in Cryptology—CRYPTO '87,* C. Pomerance, Ed., *Lecture Notes in Computer Science,* Vol. 293, Springer-Verlag, 369–378, 1988.
[33] Okamoto, T., Provably Secure and Practical Identification Schemes and Corresponding Signature Schemes. *Advances in Cryptology—CRYPTO '92,* E.F. Brickell, Ed., *Lecture Notes in Computer Science,* Vol. 740, Springer-Verlag, 31–53, 1992.
[34] Pedersen, T., Electronic Payments of Small Amounts. *Aarhus University Technical Report,* DAIMI PB-495, Denmark, 1995.
[35] Rivest, R., Shamir, A., and Adleman, L., A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. *Comm. ACM,* Vol. 21, 120–126, 1978.
[36] Schnorr, C., Efficient Signature Generation by Smart Cards. *J. Cryptology,* Vol. 4, No. 3, 161–174, 1991.
[37] Vaudenay, S., One-Time Identification with Low Memory. *Proceedings of EUROCODE '92,* Udine, Italy, *CISM Courses and Lectures,* No. 339, Springer-Verlag, 31–53, 1993.
[38] Waidner, M. and Pfitzmann, B., Loss-Tolerance for Electronic Wallets. *20th International Symposium on Fault-Tolerant Computing,* Newcastle upon Tyne, U.K., 140–147, 1990.

Further Information

Literature

Public literature on the design of electronic cash systems is scarce and dispersed. Since 1995 several books have been published about electronic commerce, but these are all nontechnical and focus on credit-card systems. The main source for articles on the cryptographic design of electronic cash systems are the annual Eurocrypt and Crypto conferences. Interesting technical information may also be found in proceedings of smart-card conferences, and by entering relevant keywords into a search engine for the World Wide Web. Patents are another useful source of information on techniques related to electronic cash systems.

In this chapter we have not discussed all manner of functionality extensions and variations that have been proposed for untraceable electronic cash. To name a few, off-line transferability for coins in the three-part form is discussed in [19]; a technique for tick payments for blank checks is introduced in [34]; escrow functionality for electronic coin systems is introduced in [13], and improved by [14]; and divisibility of electronic coins is addressed, among others, by [22]. An account of legal aspects of untraceable electronic cash can be found in [25], and a compact overview of cryptographic techniques for untraceable cash in [30]. The February issue of [27] is devoted entirely to technological aspects of electronic money.

Electronic Cash Today

Most prepaid electronic purse initiatives are based on symmetric authentication and encryption. Many of these are modeled after the Proton system, a nonanonymous register-based cash system developed by Banksys, the association of Belgium banks. One of first electronic purse projects of significance has been a Danish initiative allied to Visa International, called Danmønt. Other national purse scheme initiatives are underway in the Netherlands (Chipknip, by Interpay), Switzerland (Telekurs bank consortium) and Portugal (Multibanco Electronic Purse). A related technology called Visa Cash has been developed by Visa International, and has been piloted at the 1996 Summer Olympics.

International standardization efforts for electronic purse schemes are the CEN Intersector Electronic Purse and the EMV 3.0 specifications.

Another development is the Mondex prepaid system of Mondex International Limited, which enables repeated off-line transferability of value from person to person. All devices are tamper-resistant, and offer a keyboard and display. The version currently in operation uses conventional cryptographic authentication. In September 1995, a Fair Trading Act complaint was filed against Mondex for falsely advertising their system as anonymous, and the claim was upheld after a nine-month investigation by the Fair Trading Office; see [28] and the references provided therein.

Citibank is developing a cash system called the Electronic Monetary System (EMS), with functionality similar to that of Mondex. In particular, cash can repeatedly be transferred from person to person, without the involvement of central party, and all devices (Money Modules) are tamper-resistant and have a keypad and a display. Instead of using a register-based cash representation, cash appears to be stored and transferred in the form of two-part coins. Each note carries a complete electronic audit trail, and (invisibly to the user) electronic notes are submitted to the bank for validation and control whenever a withdrawal or deposit is made. Security is based on the ability of the bank to trace all transactions to devices.

CAFE is a European Commission sponsored project, involving thirteen leading European parties from academic research and industries, that has tested an off-line electronic cash system in the commission headquarters in Brussels. The CAFE system is based on the one-show blinding paradigm and the wallet-with-observer paradigm, and more specifically on the public-key cryptographic techniques of Brands [5, 6] for withdrawing, paying, and depositing. Transactions are conducted using a tamper-resistant smart card inserted into a handheld computer with a keyboard and display, and limited off-line transferability is offered.

Four financial institutions (the Mark Twain bank, the Merita bank, the Deutsche bank and the Advance bank) and the Sweden Post offer (or are about to) a system for untraceable electronic cash payments over the Internet. This system is a software-only on-line payment system based on Chaum's basic blind RSA signatures, developed by Amsterdam-based DigiCash. Because two-sided untraceability is not prevented, this system is open to criminal uses such as extortion, bribery and money laundering.

Recently, CyberCash has launched a payment system called CyberCoin, also for on-line payments over the Internet. This system has been designed to have functionality similar to the DigiCash system, but lacks provisions for anonymity. According to public statements from CyberCash, value resides at all times within the bank and the system actually is an instruction-based system.

45

Parallel Computation: Models and Complexity Issues[1]

Raymond Greenlaw
Armstrong Atlantic State University

H. James Hoover
University of Alberta

45.1 Introduction

The theory of **parallel computation** is concerned with the development and analysis of parallel computing models, the techniques for solving and classifying problems on such models, and the implications of this work.

[1]Raymond Greenlaw — This research partially supported by National Science Foundation grant CCR-9209184; a Fulbright Scholarship, Senior Research Award; and a Spanish Fellowship for Scientific and Technical Investigations. H. James Hoover — This research partially supported by the Natural Sciences and Engineering Research Council of Canada grant OGP 38937.

Despite technology that continually improves the performance of individual processors, humans are adept at inventing problems for which a single processor is simply too slow. Only by using many processors in parallel is there any hope of quickly solving such problems. In principle, having more processors working on a problem means it can be solved much faster. Ideally, we expect a speedup of the following form:

$$(\text{Parallel Time}) = \frac{(\text{Sequential Time})}{(\text{Number of Processors})}$$

In practice, parallel algorithms seldom attain ideal *speedup*,[2] are more complex to design, and are more difficult to implement than single-processor algorithms. The designer of a parallel algorithm must grapple with these fundamental issues:

1. What parallel resources are available,
2. What kind of speedup is desired,
3. What machine architecture should be used,
4. How the problem may be decomposed to exploit parallelism, and
5. Whether the problem is even amenable to a parallel attack?

The dilemma of parallel computation is that at some point every problem begins to resist speedup. The reality is that as one adds processors one must devote (disproportionately) more resources to interprocessor communication, and one must deal with more problems caused by processors waiting for others to finish. Furthermore, some problems appear to be so resistant to speedup via parallelism that they have earned the name *inherently sequential.*

Pragmatic versus Asymptotic Parallelism

The practice of parallel computation can be loosely divided into the pragmatic and the asymptotic.

The goal of *pragmatic parallel computation* is simply to speed up the computation as much as possible using whatever parallel techniques and equipment are available. For example, doing arithmetic on 32 bits in parallel, overlapping independent I/O operations on a storage subsystem, fetching instructions and precomputing branch conditions, or using 4 processors interconnected on a common bus to share workload are all pragmatic uses of parallelism. Pragmatic parallelism is tricky, very problem-specific, and highly effective at obtaining the modest factor of 3 or 4 speedup that can suddenly make a problem reasonable to solve. This article is not directly concerned with pragmatic parallel computation as defined above; the models and techniques we explain in this article are.

Asymptotic parallel computation, in contrast to pragmatic, is more concerned with the architecture for general parallel computation; parallel algorithms for solving broadly applicable, fundamental problems; and the ultimate limitations of parallel computation. For example, are shared memory multiprocessors more powerful than mesh-connected parallel machines? Given an unlimited number of processors, just how fast can one do n-bit arithmetic? Or, can every polynomial time sequential problem be solved in polylogarithmic time on a parallel machine? Asymptotic parallel computation does provide tools for the pragmatic person too, for example, the algorithm designer with many processors available can make use of the results from this field.

The field of asymptotic parallel computation can be subdivided into the following main areas:

1. Models: comparing and evaluating different parallel machine architectures.

[2]The speedup is simply the ratio (Sequential Time/Parallel Time). In general, the speedup is less than the number of processors.

2. Algorithm design: using a particular architecture to solve a specific problem.
3. Computational complexity: classifying problems according to their intrinsic difficulty of solution.

Many books are devoted to the design of parallel algorithms, for example [17, 23, 38]. The focus of this article is on models of parallel computation and complexity.

When discussing parallel computation it is common to adopt the following informal definitions:

- A problem is **feasible** if it can be solved by a parallel algorithm with worst-case time and processor complexity $n^{O(1)}$.
- A problem is **feasible highly parallel** if it can be solved by an algorithm with worst-case time complexity $(\log n)^{O(1)}$ and processor complexity $n^{O(1)}$.
- A problem is **inherently sequential** if it is feasible but has no feasible highly parallel algorithm for its solution.

Chapter Overview

Section 45.2—two fundamental models of parallel computation, the parallel random access machines and uniform Boolean circuit families.

Section 45.3—the fundamental parallel complexity classes P and NC.

Section 45.4—other important parallel models and mutual simulations.

Section 45.5—P-completeness, the theory of inherently sequential problems.

Section 45.6—various examples of P-**complete** problems.

Section 45.7—research issues and summary.

Section 45.8—defining terms.

"Further Information"— references for further reading.

45.2 Two Fundamental Models of Parallel Computation

Introduction

An important part of parallel computation involves the description, classification, and comparison of abstract models of parallel machines.

One of the prerequisites to studying a problem is specifying the resources that we are concerned about consuming, and the abstract machine model on which we want to compute our solutions. For practicing programmers, these two items are usually dictated by the real machines at their disposal. For example, having only a small number of processors, each with a very large memory, is a much different situation from having tens of thousands of processors each with limited memory. In the first situation we wish to minimize the number of processors required by our solution, and will choose a model that ignores memory consumption. In the second situation, where minimizing local memory may be more important, we choose a model that provides many processors (although not extravagantly many) and in which all basic operations are assumed to take the same amount of time to execute. The single biggest issue to be faced from the standpoint of developing a parallel algorithm is the *granularity* of the parallel operations. Should the problem be broken up into very small units of computations, which can be done in parallel but then may incur costly communication overhead, or should the problem be divided into relatively large chunks that can be done in parallel but where each chunk is processed sequentially?

The opposite ends of our granularity spectrum are captured by the two main models of parallel computation: *parallel random access machines* and *uniform Boolean circuit families.* For high-level, coarse-

granularity situations the preferred model is the PRAM, while for more detailed questions of implementability and small resource bounds, the finer granularity uniform Boolean circuit model is used.

Although there are many differences between parallel models, for feasible, highly parallel computations, most models are equivalent to within a polynomial in both time and hardware resources, simultaneously. By this we mean that if the size-n instances of some problem can be solved in time $T(n) = (\log n)^{O(1)}$ and processors $P(n) = n^{O(1)}$ on a machine from model M_1, then there exists a machine from model M_2 that can solve the size n instances of the problem in time $T(n)^{O(1)}$ and $P(n)^{O(1)}$ processors. Thus, if a problem is feasibly highly parallel on one model, it is so on all other equivalent models. (See Section 45.4 for more details.) This says that the class of problems solvable by feasibly highly parallel computations is robust and insensitive to minor variations in the underlying computational model.

Parallel Random Access Machines

One of the natural ways of modeling parallel computation is the generalization of the single processor into a *shared memory multiprocessor,* as illustrated in Fig. 45.1. This machine consists of many powerful independent machines (usually viewed as being of the same type) that share a large common memory which is used for computation and communication. The parallel machines envisioned in such a model contain very large numbers of processors, on the order of millions. Such enormous machines do not yet exist, but machines with 64,000 processors can be built now, and economics rather than technology is the main factor that limits their size.

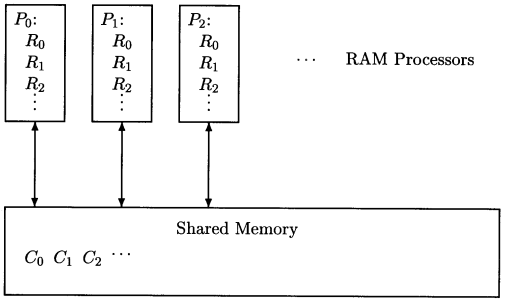


FIGURE 45.1 The Parallel Random Access Machine (PRAM). A PRAM consists of a large number of processors connected to a common memory. Each processor is quite powerful by itself, and operates independently of the others. Processors communicate by passing information through the shared memory. Arbitration may be required when more than one processor attempts to access a given memory cell.

The formal model of the shared memory multiprocessor is called a *parallel random access machine* (PRAM), and was introduced independently in [15] and [18, 19]. The PRAM model consists of a collection of random access machine (RAM) processors that run in parallel and communicate via a common memory. The basic PRAM model consists of an unbounded collection of numbered RAM processors $P_0, P_1, P_2, \ldots$ and an unbounded collection of shared memory cells $C_0, C_1, C_2, \ldots$ (see Fig. 45.1). Inputs and outputs to the PRAM computation are placed in shared memory to allow concurrent access. Each instruction is executed in unit time, synchronized over all active processors.

Each RAM processor P_i has its own local memory, knows its own index i, and has instructions for direct and indirect read/write access to the shared memory. Local memory is unbounded, and consists of memory cells $R_0, R_1, R_2, \ldots$, with each cell capable of holding an integer of unbounded size. The usual complexity measures for each individual processor's RAM computation are *time,* in the form of the number of instructions executed, and *space,* in the form of the number of memory cells accessed.

A typical PRAM instruction set, with addressing modes, is given in Table 45.1. In this simple machine, local memory cell R_0 serves as an accumulator so that at most one read and one write to shared memory occurs for each instruction. The multiply and divide instructions take only a constant operand in order to prevent the rapid generation and testing of very large numbers. These restrictions also prevent the consumption of an exponential amount of space in polynomial time.

TABLE 45.1 Sample PRAM Instruction Set

Instruction	Description
$\alpha \leftarrow \beta$	move data to cell with address α from cell with address β Address — Description R_i — local cell R_i R_{R_i} — local cell with address given by contents of R_i C_i — shared cell C_i C_{R_i} — shared cell with address given by contents of R_i
IDENT	load the processor number into R_0
CONST c	load the constant c into R_0
ADD α	add contents of α to R_0
SUB α	subtract contents of α from R_0
MULT c	multiply contents of R_0 by constant c
DIV c	divide contents of R_0 by constant c and truncate
GOTO i	branch to instruction i
IFZERO i	branch to instruction i if contents of R_0 is 0
HALT	stop execution of this processor

Two important technical issues must be dealt with by the model. The first is the manner in which a finite number of the processors from the potentially infinite pool are activated for a computation. A common way is for processor P_0 to have a special activation register that specifies the maximum index of an active processor. Any nonhalted processor with an index smaller than the value in the register can execute its program. Initially only processor P_0 is active, and all others are suspended waiting to execute their first instruction. P_0 then computes the number of processors required for the computation and loads this value into the special register. Computation proceeds until P_0 halts, at which point all active processors halt. The SIMDAG model is an example of a PRAM using such a convention [19]. Another common approach is to have active processors explicitly activate new ones via FORK instructions [15].

The second technical issue concerns the way in which simultaneous access to shared memory is arbitrated. In all models, it is assumed that the basic instruction cycle separates shared memory read operations from write operations. Each PRAM instruction is executed in a cycle with three phases. First the read operation (if any) from shared memory is performed, then the computation associated with the instruction (if any) is done, and finally the write operation (if any) to shared memory is performed. This eliminates read/write conflicts to shared memory, but does not eliminate all access conflicts. This is dealt with in a number of ways as described in Table 45.2 (see [48] and [14] for more details). All of these variants of the PRAM are deterministic, except for the ARBITRARY CRCW-PRAM, for which it is possible that repeated executions on identical inputs result in different outputs.

Any given PRAM computation will use some specific time and hardware resources. The complexity measure corresponding to *time* is simply the time taken by the longest running processor. The measure corresponding to *hardware* is the maximum number of active processors during the computation.

TABLE 45.2 How Different PRAM Models Resolve Access Conflicts

CRCW	Concurrent-Read Concurrent-Write
	Allows simultaneous reads and writes to the same memory cell with a mechanism for arbitrating simultaneous writes to the same cell:
	PRIORITY — only the write by the lowest numbered contending processor succeeds.
	COMMON — the write succeeds only if all processors are writing the same value.
	ARBITRARY — any one of the writes succeeds.
CREW	Concurrent-Read Exclusive-Write
	Allows simultaneous reads of the same memory cell, but only one processor may attempt to write to a cell.
CROW	Concurrent-Read Owner-Write
	A common restriction of the CREW-PRAM that preassigns an owner to each common memory cell. Simultaneous reads of the same memory cell are allowed, but only the owner can write to the cell, thus ensuring exclusive-write access.
EREW	Exclusive-Read Exclusive-Write
	Requires that no two processors simultaneously access any given memory cell.

Our standard PRAM model will be the CREW-PRAM with a processor activation register in processor P_0. This means that processor P_0 is guaranteed to have run for the duration of the computation, and the largest value in the activation register is an upper bound on the number of processors used. We also need to specify how the inputs are provided to a PRAM computation, how the outputs are extracted, and how the cost of the computation is accounted:

DEFINITION 45.1 Let M be a PRAM. The **input/output conventions** for M are as follows. An input $x \in \{0, 1\}^n$ is presented to M by placing the integer n in shared memory cell C_0, and the bits $x_1, \ldots, x_n$ of x in shared memory cells $C_1, \ldots, C_n$. M displays its output $y \in \{0, 1\}^m$ similarly: integer m in shared memory cell C_0, and the bits $y_1, \ldots, y_m$ of y in shared memory cells $C_1, \ldots, C_m$.

M computes in **parallel time** $T(n)$ **and processors** $P(n)$ if and only if for every input $x \in \{0, 1\}^n$, machine M halts within at most $T(n)$ time steps, activates at most $P(n)$ processors, and presents some output $y \in \{0, 1\}^*$.

M computes in **sequential time** $T(n)$ if and only if it computes in parallel time $T(n)$ using 1 processor.

With these conventions in place, and having decided on one version of the PRAM model to be used for all computations, we can talk about a function being computed in parallel time $T(n)$ and processors $P(n)$.

DEFINITION 45.2 Let f be a function from $\{0, 1\}^*$ to $\{0, 1\}^*$. The function f is **computable in parallel time** $T(n)$ **and processors** $P(n)$ if and only if there is a PRAM M that on input x outputs $f(x)$ in time $T(n)$ and processors $P(n)$.

Note that no explicit accounting is made of the local or shared memory used by the computation. Since the PRAM is prevented from generating large numbers, that is, for $T \geq \log n$ no number may exceed $O(T)$ bits in T steps, a computation of time T with P processors cannot store more than $O(PT^2)$ bits of information. Hence, for our purposes P and T together adequately characterize the memory requirement of the computation, and there is no need to parameterize it separately.

All of the various PRAM models are polynomially equivalent with respect to feasible, highly parallel computations, and so any one is suitable for defining the complexity classes P and NC that we present in Section 45.3. The final important point to note about the PRAM model is that it is generally not difficult to see (in principle) how to translate an informally described parallel algorithm into a PRAM algorithm.

Uniform Boolean Circuit Families

Boolean circuits are designed to capture very fine-grained parallel computation at the resolution of a single bit—they are a formal model of the combinational logic circuit. Circuits are basic technology, consisting of very simple logical gates connected by "bit-carrying wires". They have no memory and no notion of state. Circuits avoid almost all issues of machine organization and instruction repertoire. Their computational components correspond directly with devices that we can actually fabricate, although the circuit model is still an idealization of real electronic devices. The circuit model ignores a host of important practical considerations such as circuit area, volume, pin limitations, power dissipation, packaging, and signal propagation delay. Such issues are addressed more accurately by more complex VLSI models (see [34]). But for much of the study of parallel computation, the Boolean circuit model provides a good compromise between simplicity and realism.

Each circuit is an acyclic directed graph in which the edges carry unidirectional logical signals and the vertices compute elementary Boolean logical functions. Formally we denote the logical functions by the sets $B_k = \{f \mid f : \{0,1\}^k \rightarrow \{0,1\}\}$, that is, each B_k is the set of all k-ary Boolean functions. We refer informally to such functions by strings "1", "0", "$\neg$", "$\wedge$", "$\vee$", "NOT", "AND", "OR", and so on. The entire graph computes a Boolean function from the inputs to the outputs in a natural way.

DEFINITION 45.3 A **Boolean circuit** α is a labeled finite oriented directed acyclic graph. Each vertex v has a type $\tau(v) \in \{I\} \cup B_0 \cup B_1 \cup B_2$. A vertex v with $\tau(v) = I$ has indegree 0 and is called an **input**. The inputs of α are given by a tuple $\langle x_1, \ldots, x_n \rangle$ of distinct vertices. A vertex v with outdegree 0 is called an **output**. The outputs of α are given by a tuple $\langle y_1, \ldots, y_m \rangle$ of distinct vertices. A vertex v with $\tau(v) \in B_i$ must have indegree i and is called a **gate**.

Note that fanin is less than or equal to two but fanout is unrestricted. Inputs and gates can also be outputs. See Fig. 45.2 for an example.

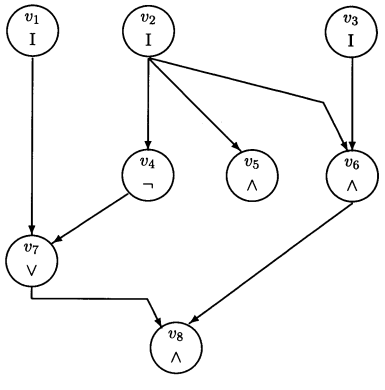


FIGURE 45.2 A Sample Boolean Circuit. A Boolean circuit with inputs $\langle v_1, v_2, v_3 \rangle$ and outputs $\langle v_5, v_8 \rangle$. It has size 8, depth 3, and width 2. Input v_2 has fanout 3. Gate v_4 has fanin 1 and fanout 1.

DEFINITION 45.4 A Boolean circuit α with inputs $\langle x_1, \ldots, x_n \rangle$ and outputs $\langle y_1, \ldots, y_m \rangle$ **computes a function** $f : \{0,1\}^n \rightarrow \{0,1\}^m$ in the following way: input x_i is assigned a value $\nu(x_i)$ from $\{0,1\}$ representing the ith bit of the argument to the function. Every other vertex v is assigned the unique value $\nu(v) \in \{0,1\}$ obtained by applying $\tau(v)$ to the value(s) of the vertices incoming to v. The value of the function is the tuple $\langle \nu(y_1), \ldots, \nu(y_m) \rangle$ in which output y_j contributes the jth bit of the output.

The most common resource measures of interest for a circuit are its *size* and *depth.*

DEFINITION 45.5 The **size** of α, denoted size(α), is the number of vertices in α. The **depth** of α, denoted depth(α), is the length of the longest path in α from an input to an output.

A less common measure is *width,* which intuitively corresponds to the maximum number of gate values, not counting inputs, that need to be preserved when the circuit is evaluated level-by-level.

DEFINITION 45.6 The **width** of α, denoted width(α), is

$$\max_{0<i<\text{depth}(\alpha)} \left| \left\{ v \ : \ \begin{array}{l} 0 < d(v) \leq i \text{ and} \\ \text{there is an edge from vertex } v \text{ to a vertex } w, d(w) > i \end{array} \right\} \right|$$

where d(w), the depth of w, is the length of the longest path from any input to vertex w.

Each circuit α is described by a binary string denoted by $\overline{\alpha}$. This description can be thought of as a blueprint for that circuit, or alternatively as a parallel program executed by a universal circuit simulator. In any case, although we speak of circuits, we actually generate and manipulate circuit descriptions (exactly as we manipulate programs and not Turing machines). One common description is the *standard encoding,* the precise details of which are not important to this chapter (see [39]). The main point is that circuit descriptions are simple objects to generate and manipulate.

An individual circuit with n inputs and m outputs is a finite object computing a function from binary strings of length n to binary strings of length m. In contrast to a PRAM computation in which one algorithm handles all possible lengths of inputs, different circuits are required for different length inputs. The collection of different circuits for the various input lengths is called a *circuit family.*

The simplest kind of circuit family is used for computing some function f whose output length m is a function, possibly constant, only of the length of the input. That is, the length of $f(x)$ on n-bit inputs x is some function $m(n)$. In this case we can represent the function $f : \{0,1\}^* \rightarrow \{0,1\}^*$ by an infinite sequence of circuits, $\{\alpha_n\}$, where circuit α_n computes f restricted to inputs of length n. Such a sequence is called a *Boolean circuit family.*

DEFINITION 45.7 A **Boolean circuit family** $\{\alpha_n\}$ is a collection of circuits, each α_n computing a function $f^n : \{0,1\}^n \rightarrow \{0,1\}^{m(n)}$. The **function computed by** $\{\alpha_n\}$, denoted f_α, is the function $f_\alpha : \{0,1\}^* \rightarrow \{0,1\}^*$, defined by $f_\alpha(x) \equiv f^{|x|}(x)$.

Functions where the output length varies with the value of x as well as its length can be handled by special encodings (similar to the output convention used for PRAMs). The case where the length of the output is always 1 is particularly important for defining formal languages.

DEFINITION 45.8 Let $\{\alpha_n\}$ be a Boolean circuit family that computes the function $f_\alpha : \{0,1\}^* \rightarrow \{0,1\}$. The **language decided by** $\{\alpha_n\}$, denoted by L_α, is the set $L_\alpha = \{x \in \{0,1\}^* \mid f_\alpha(x) = 1\}$.

Defining an infinite collection of circuits with no computational constraints gives *nonuniform* circuit families. Nonuniform circuit families are unexpectedly powerful in that they can "compute" noncomputable functions. These circuit families are widely used as objects of lower bound proofs, where their power merely serves to strengthen the significance of the lower bounds. However, they are a somewhat unsatisfactory model in which to consider upper bounds. In particular, there may be no effective way, given n, to obtain a description of the n-th circuit α_n.

The *uniform* approach gives an explicit algorithm for constructing the elements of the circuit family. Each circuit family is defined by a program in some computational model that takes n as input and then outputs the encoding $\overline{\alpha_n}$ of the nth member. In doing so, an infinite object, the family, is effectively described by a finite object, the program. The question then becomes, how much computational power is permitted in producing the description $\overline{\alpha_n}$?

Borodin, arguing that the circuit constructor should have no more computational power than the object it constructs, introduced the notion of *uniformity* [1]. One such example of a weak constructor is a Turing machine that is limited to only $O(\log n)$ work space on inputs of length n. Such machines have limited computing power but can still describe a wide class of useful circuit families.

DEFINITION 45.9 A family $\{\alpha_n\}$ of Boolean circuits is **logarithmic space uniform** if the transformation $1^n \rightarrow \bar{\alpha_n}$ can be computed in $O(\log(\text{size}(\alpha_n)))$ space on a deterministic Turing machine.

Note how the complexity of producing the description of α_n is expressed in terms of the size of the resulting circuit. Logarithmic space uniformity is sometimes called *Borodin–Cook uniformity,* and was first mentioned in [5]. This notion of uniformity has the desirable property that the description $\overline{\alpha_n}$ can be produced in polynomial time sequentially, or in polylogarithmic time in parallel with a polynomial number of processors. Thus, the circuit constructor is reasonable from both sequential and parallel perspectives.

Equivalence of PRAMs and Uniform Boolean Circuit Families

We have remarked that many parallel models are equivalent with respect to feasible highly parallel algorithms. That is, if a problem has a feasible highly parallel solution on one model, then it also has one on any equivalent model. Originally the notion of feasible and highly parallel came from the observation that certain problems had polylogarithmic time and polynomial processor solutions on many different models. This ability to support feasibly highly parallel algorithms has now become the defining characteristic of all "reasonable" parallel models. In order for any new parallel model to be considered reasonable, it must be able to relatively efficiently simulate some existing reasonable model and vice versa. (See the remarks in Section 45.7 about more realistic parallel models.)

In this sense, PRAMs and uniform Boolean circuits are both reasonable parallel models, and can simulate each other in a way that maintains feasible highly parallel computations. This result, as other simulation results, is quite technical, and sensitive to the precise details of each model involved. The basic ideas are relatively simple, and easily worked to completion by those with a penchant for detail (see [21] for more details). To simulate a circuit operating on input x of length n, the PRAM first uses the uniformity condition to compute the description of the circuit from the family that handles the inputs of length n. Then it initializes the inputs and does a gate-by-gate simulation of the circuit, evaluating in parallel all the gates at the same level. The simulation of a PRAM by a circuit is a bit more complex. The circuit for inputs of length n has to account for the worst case time and processor consumption of the PRAM, and the product of these two gives the size of the circuit. The circuit is then organized as a sequence of layers, each layer of which simulates one time step of each processor of the PRAM.

THEOREM 45.1 *A function f from $\{0,1\}^*$ to $\{0,1\}^*$ can be computed by a logarithmic space uniform Boolean circuit family $\{\alpha_n\}$ with depth$(\alpha_n) = (\log n)^{O(1)}$ and size$(\alpha_n) = n^{O(1)}$ if and only if f can be computed by a CREW-PRAM M on inputs of length n in time $T(n) = (\log n)^{O(1)}$ and processors $P(n) = n^{O(1)}$.*

Similar simulation results among the various models of parallel computation (see Section 45.4) allow us to observe that if a problem is inherently sequential on one reasonable parallel model, then it is inherently sequential on all other reasonable models.

45.3 Fundamental Parallel Complexity Classes

Introduction

The most important complexity classes in parallel computation are P, NC, and the class of P-complete problems. The problems in P are considered to be easy to solve on a single processor; they are *tractable* or *feasible.* Such problems may still take unacceptable amounts of sequential time to solve and so are ideal candidates for parallel computation. NC consists of those problems in P that can be solved very fast in parallel. The P-complete problems appear to be outside of NC. The classes NC and P are defined formally in the next section, and the P-complete problems, which require more technical background, are defined formally in Section "Completeness."

Nick's Class (NC) and Polynomial Time (P)

Because of their simplicity, language recognition and decision problems are the standard mechanisms for defining the computational classes of complexity theory. We refer the reader to Chapters 27, 28, and 29 for background and more in-depth discussion. The following definitions aid in defining the classes[3] NC and P.

DEFINITION 45.10 Let L be a language over $\{0, 1\}^*$. The **characteristic function** of L is the function f_L defined on all $x \in \{0, 1\}^*$ such that $f_L(x) = 1$ if $x \in L$, and $f_L(x) = 0$ if $x \notin L$.

DEFINITION 45.11 A language $L \subseteq \{0, 1\}^*$ is **decidable in sequential time** $T(n)$ if and only if the characteristic function of L can be computed in sequential time $T(n)$.

DEFINITION 45.12 A language $L \subseteq \{0, 1\}^*$ is **decidable in parallel time** $T(n)$ **with** $P(n)$ **processors** if and only if the characteristic function of L is computable in parallel time $T(n)$ and processors $P(n)$.

A single sequential processor running in polynomial time can easily simulate a polynomial number of processors running in polynomial time, and conversely. This means we can ignore sequential computation and restrict our attention to PRAMs, as per the following lemma.

LEMMA 45.1 A language L is decidable in sequential time $n^{O(1)}$ if and only if L is decidable in parallel time $n^{O(1)}$ with $n^{O(1)}$ processors.

We can now define the class of feasible highly parallel problems, NC, and the class of polynomial time sequential problems, P.

DEFINITION 45.13 The class NC is the set of all languages L that are decidable in parallel time $(\log n)^{O(1)}$ and processors $n^{O(1)}$.

DEFINITION 45.14 The class P is the set of all languages L that are decidable in parallel time $n^{O(1)}$ and processors $n^{O(1)}$.

[3] NC was named after Nick Pippenger.

Many complexity classes have functional analogs, typically denoted by prefixing the class with the letter F. For example, FNC denotes "function NC" — the class of functions computable in the resource bounds of the class NC. Similarly, FP denotes "function P." We assume the reader has an intuition for what these classes are.

From Lemma 45.1, we know that $NC \subseteq P$. The important open question for parallel computation is whether this inclusion is proper.

A Basic Example—Parallel Sums

Many problems in P have truly dramatic speed improvements when solved in parallel. Here we consider a simple but important problem, **parallel sums,** defined as follows:

Given: Natural numbers $a_1, a_2, \ldots, a_n$, and t.

Problem: Is $t = a_1 + a_2 + \cdots + a_n$?

We can solve this problem *sequentially* by adding numbers together two at a time. Such a procedure requires $n - 1$ additions to compute the total, and the total can be checked against t to solve the problem. Parallel sums is an example of a problem that is *sequentially feasible.* Now consider solving parallel sums using a number of processors operating in parallel.

With *limited parallelism,* which means using a fixed number of processors, we can never achieve more than a *constant factor* of improvement over the best-known sequential time. For example, with two processors we can never be more than twice as fast as with one. But suppose that processors were so plentiful and inexpensive that we could consider using thousands or more in parallel. This introduces a qualitative change in the way we approach parallel computation. What kind of speedup could be achieve if we had essentially an unlimited number of processors?

With *polynomially bounded parallelism,* we have the potential to achieve more than a constant factor speedup. For example, suppose that we use $P(n) = n/2$ processors to solve an instance of parallel sums consisting of n numbers and a value t. That is, we have one processor for each pair of numbers. The computation can then be organized as a binary tree in which we add as many pairs of numbers as possible in each time step. The time to compute the addition of n numbers is the height of the tree, so the computation can be done in $O(\log n)$ elapsed time using $n/2$ processors assuming a unit cost for addition.

We must emphasize the importance of this example. Using only a relatively small number of processors (about half the size of the problem instance), we have achieved an *exponential*[4] improvement in the solution of parallel sums by going from $O(n)$ sequential time to $O(\log n)$ parallel time. Problems that exhibit this kind of improvement are exactly those that belong to the class NC.

Does *NC* Equal *P*?

Many of the problems in P have highly parallel solutions similar to parallel sums, and one wonders if perhaps every problem in P can be solved fast on a parallel machine.

> *Does every problem with a feasible sequential solution also have a feasible highly parallel solution? That is, does NC equal P?*

Unfortunately, it seems that some problems in P do not lie in NC, and one of the main tasks of complexity theory is to identify these suspected *inherently sequential problems.* The theory of P-completeness described in Section 45.5 provides evidence suggesting NC and P are indeed different classes.

[4] By exponential we mean going from a polynomial function to a polylogarithmic function. Therefore, an improvement from n^3 to $(\log n)^2$ is considered an exponential improvement.

45.4 Parallel Models and Simulation Results

Introduction

Parallel models of computation vary widely in *modes of communication, instruction sets,* and in what constitutes a *processor.* There is no clearly superior architecture for a general purpose parallel computer, and the same is true for the theoretical models of parallel computation. Thus there is room for inventing new architectures and computational features. These *new* models are evaluated by two basic methods: assessing how well they speed up existing sequential computation classes, and how well they do in mutual simulations with existing parallel models. The comparison with existing parallel models also provides information about how easy the model is to use and how close the model is to a real machine. The ultimate aim of this exercise is to identify the key attributes that must be present in any reasonable parallel model.

In this section we introduce a number of synchronous parallel models and some simulation results. A parallel machine is **synchronous** if all processors must complete the execution of their current instruction before any processor begins execution of its next instruction. Brief descriptions, providing only a taste of the models, are given in "The Fixed Structure Models" and "The Modifiable Structure Models". Many of the important technical issues concerning the models can be extrapolated from the PRAM and Boolean circuit models given in "Parallel Random Access Machines" and "Uniform Boolean Circuit Families." The meanings of the names of complexity classes used in this section are easily inferred, and are mostly historical. For example, UAG-TIME stands for uniform aggregate time and ATIME, SPACE represents alternating Turing machines with simultaneous time and space bounds.

The simulation results that relate two seemingly different parallel models M_1 and M_2 follow a basic pattern. If any problem Π can be solved on a machine of type M_1 using T_{M_1} time and P_{M_1} processors, can Π be solved on an M_2 machine in some related bounds? For example, using $f_T(T_{M_1})$ time and $f_P(P_{M_1})$ processors, where f_T and f_P are two well-behaved functions relating M_1's resources of time and processors respectively, to M_2's. What about vice versa? Do time and processor bounds for problem Π on model M_2 imply time and processor bounds on M_1?

The simulation results that relate a parallel model and a sequential model have a different purpose. Here the focus is on how parallel resources relate to sequential resources. This provides an understanding of what kinds of sequential problems can be automatically parallelized. A useful framework for this was proposed by [6]; we follow his discussion and build upon it in the next section.

Cook's Classification Scheme for Parallel Models

The various parallel models, which are defined in the following sections, are grouped into two classes: the *fixed structure* models and the *modifiable structure* models. "The fixed structure models correspond to sequential machines with a fixed storage structure such as Turing machine tape". They consist of *alternating Turing machines, bounded fanin uniform Boolean circuit families, conglomerates, k-PRAMs,* and *uniform aggregates.* This class represents parallel models whose interconnection pattern is fixed throughout a computation.

"The *modifiable structure* models, correspond to the modifiable sequential Storage Modification Machines [41] and RAMs." They consist of *hardware modification machines, MRAMs* (standard RAMs with a multiplication instruction [22]), *PRAMs, SIMDAGs, unbounded fanin uniform Boolean circuit families,* and *vector machines.* This class represents parallel machines whose communication links are allowed to vary during a computation.

The above mentioned parallel machines can be grouped according to their *time* resource usage as related to *sequential space* on a Turing machine. For a fixed structure model X, the following relationship typically holds:

$$\text{X-TIME}(S(n)) \subseteq \text{DSPACE}(S(n)) \subseteq \text{NSPACE}(S(n)) \subseteq \text{X-TIME}(S(n)^2) \tag{45.1}$$

whereas a modifiable structure model Y usually satisfies

$$\text{Y-TIME}(S(n)) \subseteq \text{DSPACE}(S(n)^2) \subseteq \text{Y-TIME}(S(n)^2) \tag{45.2}$$

For more on DSPACE (deterministic Turing machine space) and NSPACE (nondeterministic Turing machine space) see Chapter 27. Equation[5] (45.2) is not known to be true for some of the models in the modifiable class. For example, it is not known how to simulate $S(n)$ space bounded Turing machines by vector machines or MRAMs running in time $S(n)$. The SIMDAG can simulate NSPACE($S(n)$) in linear time.

The Fixed Structure Models

Uniform Families of Bounded Fanin Boolean Circuits

As presented in Section "Uniform Boolean Circuit Families" the resources of interest for circuits are depth, size, and width. Circuit depth is a useful measure of parallel time because a lower bound on the running time for a problem Π using the circuit model can be applied to parallel machines that are implemented using circuit technology, or to parallel models that are equivalent to circuits with respect to the time resource. The fixed structure of circuits as opposed to the programmable nature of some other parallel models facilitates the proof of lower bounds as noted by [43].

A shortcoming of the circuit model is that it does not provide a very good measure of hardware. The size of the circuit provides an upper bound on the hardware size but intuitively this is a poor bound because a circuit is not allowed to reuse any of its gates. Width seems to provide a reasonable measure of hardware size for synchronous circuits (see Theorem 45.4). Uniform circuit depth corresponds to sequential space in the following way:

THEOREM 45.2 *[1] For* $T(n) \geq \log n$ $UDEPTH(T(n)) \subseteq DSPACE(T(n)) \subseteq NSPACE(T(n)) \subseteq UDEPTH(T(n)^2)$.

The time bound $T(n)$ has to be computable within the same resource constraints established by the uniformity conditions of the model. This is in practice only a technical annoyance, and so from now on unless otherwise stated we assume that $T(n) \geq \log n$, and that it is constructible in $O(\log n)$ space.

Uniform Aggregates

Aggregates, defined in [11], are circuits with feedback that have a special input convention that does not count the hardware required to hold the input. The feedback allows gates to be reused during a computation and thus reflects more accurately the amount of hardware required to solve problems. The resources of interest for an aggregate are *hardware* and *time*. Aggregates, like circuits, work for only one fixed input length per aggregate, and thus they require the same notions of family and uniformity as for circuits. The unusual input convention for aggregates allows a computation to use less than linear hardware. Aggregates like circuits provide a reasonable measure of parallel time. In fact, the following theorem shows that they provide the same measure of time as circuits. (All of these are results from [11] and [12]).

[5] As is customary, we refer to the relations in (1) and (2) as Eqs. (1) and (2), respectively, even though they are not mathematical equations but only LaTeX equations.

THEOREM 45.3 *UAG-TIME*$(T(n)) = UDEPTH(T(n))$.

As a corollary to Theorems 45.2 and 45.3, we get the following result.

COROLLARY 45.1 UAG-TIME$(T(n)) \subseteq$ DSPACE$(T(n)) \subseteq$ NSPACE$(T(n)) \subseteq$ UAG-TIME$(T(n)^2)$.

Theorem 45.2 and Corollary 45.1 provide evidence for the Parallel Computation Thesis that is discussed further in its own section. The following result relates uniform synchronous circuit width to uniform aggregate hardware. A circuit is **synchronous** if all inputs to a gate v come from gates at depth $(d(v) - 1)$ (see Definition 45.5).

THEOREM 45.4 *DSPACE*$(S(n)) = UAG\text{-}HARDWARE(S(n)) = USWIDTH(S(n))$.

Width represents the maximum number of gates that have to be active at any one time during a level-by-level evaluation of a circuit, and thus gives a reasonable measure for hardware size.

Conglomerates

A conglomerate is defined as a collection of interconnected finite state machines M_0, M_1, M_2, ... [18]. Each M_i is deterministic and has $r \geq 1$ inputs and 1 output. The conglomerate is unlike a circuit in that it may have loops, and unlike an aggregate in that its input convention implies hardware size $\Omega(n)$. A uniformity requirement on the connection function of conglomerates similar to what one does for circuits and aggregates is imposed. On a given input of size n, the first n machines each contain one bit of the input. All machines begin in a designated initial state. The computation halts if M_0 enters a special final state.

THEOREM 45.5 *[19] If C is a conglomerate whose connection function f can be computed in space n on a Turing machine then CONG-TIME*$(T(n)) \subseteq DSPACE(T(n)) \subseteq NSPACE(T(n)) \subseteq CONG\text{-}TIME(T(n)^2)$.

Time is the only resource that has been formally studied for the conglomerate. The obvious definition of hardware for a conglomerate is the number of finite state machines that are "used" during a computation. For values of hardware size $\Omega(n)$, conglomerates provide the same measure of hardware that aggregates do.

k-PRAMs

The k-PRAM [40] does not have any global memory structure. Instead a processor is allowed to communicate with another processor through a *call* by passing parameters or via a *return* through the *channel registers.* A processor is also allowed the following instruction: if "child processor" returned then label_1 else label_2. This provides an additional way to communicate because the return time can affect the parent's computation. Communication between processors in a k-PRAM is much more restricted than in PRAM models whose processors share a global memory. The constant k indicates the branching factor out of a processor. The inability of a processor to communicate with an unbounded number of processors is the motivation for placing the model in the fixed structure class. Theorem 45.6 shows that time on this nondeterministic model is polynomially related to sequential space, although this result is not as tight as for some of the other parallel models.

THEOREM 45.6 *[40] If* $T(n) \geq n$ *is k-PRAM-countable then NSPACE*$(T(n)) \subseteq k\text{-}NPRAM\text{-}TIME(T(n)^2)$. *If* $T(n) \geq n$ *is RAM-constructible then k-NPRAM-TIME*$(T(n)) \subseteq NSPACE(T(n)^3)$.

Alternating Turing Machines

An alternating Turing machine [2] is a generalization of a nondeterministic Turing machine to have universal acceptance states as well as existential acceptance states. (See Chapter 27.) ATM resources are *time, space,* and *alternation,* where **alternation** counts the maximum number of changes between existential and universal acceptance states, in an accepting subtree of an ATM computation.

ATMs are particularly suited for problems that can be modeled as two person games. One drawback of the ATM as a parallel model is that there does not seem to be a resource measure for the ATM that corresponds to hardware [6].

The result stated below demonstrates the close relationship between ATMs and uniform families of bounded fanin Boolean circuits.

THEOREM 45.7 *[39] If S and T are computable in deterministic time $O(T(n))$ then ATIME, SPACE$(T(n), S(n))$ = UDEPTH, SIZE$(T(n), 2^{O(S(n))})$.*

The next theorem of Ruzzo and Tompa, (see [43] for a proof) shows that ATMs with alternation considered as a resource are a good model of unbounded fanin, parallel computation (see Section "SIMDAGs and CRCW-PRAMs").

THEOREM 45.8 *If $T(n)$ and $S(n)$ are suitable functions then ATM-ALT, SPACE$(T(n), S(n))$ = CRCW-PRAM-TIME, PROC$(T(n), 2^{O(S(n))})$.*

The Modifiable Structure Models

SIMDAGs and CRCW-PRAMs

The SIMDAG was formally defined in [18]. SIMDAG is an acronym for *S*ingle *I*nstruction Stream, *M*ultiple *Da*ta Stream, *G*lobal Memory. The model is now more commonly known as the CRCW-PRAM. Goldschlager proposed a fair cost per instruction version of the SIMDAG, called the *charged* SIMDAG. The charged SIMDAG satisfies Eq. (45.1), whereas the following theorem holds for the SIMDAG.

THEOREM 45.9 *[18] If $T(n) \geq \log n$ is SIMDAG-countable then NSPACE$(T(n)) \subseteq$ SIMDAG-TIME$(T(n)) \subseteq$ DSPACE$(T(n)^2)$.*

Uniform Families of Unbounded Fanin Boolean Circuits

Lower bounds for unbounded fanin Boolean circuits were proved in [16]. The following theorem by Stockmeyer and Vishkin relates CRCW-PRAMs to uniform families of unbounded fanin Boolean circuits in such a way that these lower bounds also hold for the CRCW-PRAM.

THEOREM 45.10 *[43] There is a constant c and a function $q(P, T, n)$ bounded above by a polynomial in P, T, and n such that the following holds. Let M be a CRCW-PRAM with processor bound $P(n)$ that operates in time $T(n)$. There is a constant d_M and, for each n, a circuit C_n of size $d_M q(P(n), T(n), n)$ and depth $cT(n)$ such that C_n realizes the input-output behavior of M on inputs of size n.*

Combining Theorem 45.10 with results from [16], we get the following corollary.

COROLLARY 45.2 [43] A CRCW-PRAM with a polynomially bounded number of processors that operates in constant time cannot compute parity, multiply integers, find the transitive closure of a graph, determine whether a graph has a perfect matching, or sort integers.

This corollary illustrates why Theorem 45.10 is a useful step toward unifying the modifiable parallel models. It allows the lower bounds already known for one model to be transferred over to another. It is possible to simulate unbounded fanin Boolean circuits via PRAMs [43]. See Section "Equivalence of PRAMs and Uniform Boolean Circuit Families" for the bounded fanin case of this simulation.

Hardware Modification Machines

The hardware modification machine (HMM) (see [10, 11, 12]) was defined to measure hardware more effectively. A hardware modification machine is made up of a finite collection of finite state machines. The machines are connected together in the same manner as in the conglomerate, the main difference being that the connections may be changed locally *during* a computation. A single HMM works for all input lengths so there is no need to define uniformity. The resources of interest for HMMs are *time* and *hardware*. The input convention for the HMM is such that we can consider hardware values that are sublinear. The following theorem shows that the HMM satisfies Eq. (45.2).

THEOREM 45.11 *[11, 12]* $DSPACE(T(n)) \subseteq HMM\text{-}TIME(T(n)) \subseteq DSPACE(T(n)^2)$.

This theorem illustrates that despite the HMM's processors being allowed only a bounded number of connections at any step during the computation, the HMM can still simulate DSPACE linearly. The following theorem shows the relationship between DSPACE and HMM hardware.

THEOREM 45.12 *[11, 12]* $HMM\text{-}HARDWARE(H(n)) \subseteq DSPACE(H(n)(\log n + \log H(n)))$.

Other interesting results for *parallel pointer machines* are given in [32].

Parallel Computation Thesis

The **Parallel Computation Thesis** is that time-bounded parallel machines are polynomially related to space-bounded sequential machines [18]. That is, for any function $T(n)$,

$$\text{PARALLEL-TIME}(T(n)^{O(1)}) = \text{SEQUENTIAL-SPACE}(T(n)^{O(1)}) \; . \tag{45.3}$$

The Parallel Computation Thesis, by constructively relating sequential space to parallel time, provides an automatic mechanism for obtaining a fast parallel algorithm from a highly space efficient sequential algorithm.

Any model satisfying Eqs. (45.1) or (45.2) gives support to the Parallel Computation Thesis. For example, Theorem 45.2, Corollary 45.1, Theorem 45.5, and Theorem 45.11 are results of the form described in Eq. (45.1) for uniform bounded fanin circuits, uniform aggregates, conglomerates, and HMMs, respectively. The next few results provide additional evidence supporting the Parallel Computation Thesis for vector machines, CREW-PRAMs, and ATMs.

THEOREM 45.13 *[36] If* $T(n)$ *is VM-countable and* $T(n) \geq \log n$ *then* $NSPACE(T(n)) \subseteq VM\text{-}TIME(T(n)^2)$. *If* $T(n)$ *is RAM-constructible and* $T(n) \geq \log n$ *then* $VM\text{-}TIME(T(n)) \subseteq DSPACE(T(n)^2)$.

THEOREM 45.14 *[15]* $CREW\text{-}PRAM\text{-}TIME(T(n)) \subseteq DSPACE(T(n)^2) \subseteq CREW\text{-}PRAM\text{-}TIME(T(n)^2)$.

THEOREM 45.15 *[2]* $ATIME(T(n)) \subseteq DSPACE(T(n)) \subseteq NSPACE(T(n)) \subseteq ATIME(T(n)^2)$.

The proofs of the simulation of sequential space by parallel time are similar to, and motivated by, Savitch's Theorem (see Chapter 27). The general structure of such proofs is as follows:

1. Let M be an $S(n)$ space bounded Turing machine (deterministic or nondeterministic) with initial configuration C_0 and unique final configuration C_f.
2. Observe that M has at most $2^{O(S(n))}$ possible configurations.
3. Construct the state transition matrix or transition graph for M.
4. Compute the transitive closure by repeated squaring of the transition matrix or perform path doubling in the transition graph.
5. Accept if and only if the entry at position C_0C_f is 1 in the transition matrix, or if and only if there is a path from C_0 to C_f in the transition graph.

For the other direction, of simulating parallel time by sequential space, the basic technique is to perform a depth-first search on the instructions executed by the machine being simulated. This is done using a recursive procedure. The recursive procedure's initial call often has the form VERIFY(q, x, t), where q is a final condition, x is the input, and t is the time bound. The procedure VERIFY checks to see that the *final* state or *accept* instruction is reached on input x in time t. The recursion depth is the same as the running time, say $T(n)$, of the machine being simulated. In situations where the recursion requires only constant space at each level, the whole simulation is performed using linear space. If the recursion needs space $T(n)$ in each call to store parameters and return values, then the simulation takes quadratic space.

45.5 *P*-Completeness Theory

The following is a basic presentation of P-completeness theory. For an in-depth study, see [21].

Reducibility

The theory of P-completeness is useful for categorizing problems that are potentially inherently sequential; it parallels the theory of NP-completeness. One of the key ideas required in the development of the theory is the ability to relate one problem to another, via the notion of *reducibility* (see Chapter 28 for more on this subject). Here we focus on one of the most basic forms.

DEFINITION 45.15 A language L is NC **many-one reducible** or NC **reducible** to L', written $L \leq_m^{NC} L'$, if and only if there is a function f in FNC such that $x \in L$ if and only if $f(x) \in L'$.

Since NC reducibility is transitive, we can use a series of individual reductions to achieve a more complicated reduction.

LEMMA 45.2 NC reducibility is transitive. That is, whenever $L \leq_m^{NC} L'$ and $L' \leq_m^{NC} L''$, then $L \leq_m^{NC} L''$.

In many reductions from a language L to a language L', the exact complexity of L' is unknown. Although this gives us no absolute information about the computational complexity of L, it still provides useful information about the relative difficulties of the two languages. In particular, assuming the reduction is not too powerful, it implies that L is no more difficult to decide than L'. It is important to note that if the reduction is allowed too much power, it will mask the complexity of L'. The following shows that NC reducibility has the appropriate level of power, since it preserves membership in NC.

LEMMA 45.3 If $L' \in NC$ and $L \leq_m^{NC} L'$ then $L \in NC$.

A less powerful notion called AC^0 reducibility is discussed at length in Chapter 28.

Completeness

The idea of P-completeness is to identify those problems in P that are the "hardest".

DEFINITION 45.16 A language L is P-**hard under** NC **reducibility** if and only if $L' \leq_m^{NC} L$ for every $L' \in P$. A language L is P-**complete under** NC **reducibility** if and only if $L \in P$ and L is P-hard.

THEOREM 45.16 *If any P-complete language is in NC then NC equals P.*

Theorem 45.16 tells us that the fundamental question of whether NC equals P has the same answer as the question of whether any P-complete problem is in NC. Much evidence has accumulated showing that it is unlikely that NC equals P, and so the P-complete problems are likely to be inherently sequential. Thus, when trying to design a highly parallel algorithm, one should avoid solving P-complete problems. That is, do not make use of a subroutine call to solve a P-complete problem since this will create a parallel bottleneck.

Proof Methodology for *P*-Completeness

The steps required to show that a language L is P-complete are as follows:

1. Demonstrate that L is in P
   (a) provide an algorithm A for L
   (b) prove algorithm A is correct
   (c) prove algorithm A runs in polynomial time
2. Prove that L is P-hard
   (a) for all $L' \in P$, prove that $L' \leq_m^{NC} L$ (this is usually done by providing a function f reducing a known P-complete problem to L)
   (b) prove f is a valid reduction
   (c) prove f is in FNC

In the next section we examine the most fundamental P-complete problems. These are the problems that, by application of Lemma 45.2, are most frequently used to demonstrate a new problem is P-complete.

45.6 Examples of *P*-Complete Problems

Our goal in this section is to acquaint the reader with a number of P-complete problems. The full proofs of P-completeness may be found in [21].

Generic Machine Simulation

The canonical device for performing sequential computations is the Turing machine, with its single processor and serial access to memory. Of course, the usual machines that we call sequential are not

nearly so primitive, but fundamentally they all suffer from the same bottleneck created by having just one processor. So to say that a problem is inherently sequential is to say that solving it on a parallel machine is not substantially better than solving it on a Turing machine. What could be more sequential than the problem of simulating a Turing machine computation? If we could just discover how to simulate efficiently, in parallel, every Turing machine that uses polynomial time, then every feasible sequential computation could be translated automatically into a highly parallel form. Thus, we are interested in the following problem.

DEFINITION 45.17 Generic Machine Simulation Problem (GMSP)
Given: A string x, a description $\overline{M}$ of a Turing machine M, and an integer t coded in unary. The input is coded as $x\#\overline{M}\#^t$, where # is a delimiter character not otherwise present in the string and $\#^t$ is the unary encoding of t.
Problem: Does M accept x within t steps?

Intuitively at least, it is easy to see that this problem is solvable in polynomial time sequentially—just interpret M's program step-by-step on input x until either M accepts or t steps have been simulated, whichever comes first. Such a step-by-step simulation of an arbitrary Turing machine by a fixed one is the essence of the fundamental result that universal Turing machines exist. Given a reasonable encoding $\overline{M}$ of M, the simulation of it by the universal machine will take time polynomial in t and the lengths of x and $\overline{M}$, which in turn is polynomial in the length of the universal machine's input. (This is why we insist that t be encoded in unary.)

It is easy to NC reduce an arbitrary language L in P to the generic machine simulation problem. Let M_L be a Turing machine recognizing L in polynomial time and let $r(n) = n^{O(1)}$ be an easy-to-compute upper bound on that running time. To accomplish the reduction, given a string x, simply generate the string $f(x) = x\#\overline{M_L}\#^{r(|x|)}$. Then $f(x)$ will be a "yes" instance of the generic machine simulation problem if and only if x is in L. This transformation is easily performed in NC (see [21] for details).

THEOREM 45.17 *The generic machine simulation problem is P-complete.*

GMSP is in fact a bit too generic to be very useful for proving other problems are P-complete. A problem that is better suited for this is described in the next section.

The Circuit Value Problem and Variants

The fundamental P-complete problem is the **circuit value problem**, defined as follows:

DEFINITION 45.18 Circuit Value Problem (CVP)
Given: An encoding $\overline{\alpha}$ of a Boolean circuit α, inputs $x_1, \ldots, x_n$, and a designated output y.
Problem: Is output y of α TRUE on input $x_1, \ldots x_n$?

THEOREM 45.18 *[31] The Circuit Value Problem is P-complete.*

The circuit value problem plays the same role in P-completeness theory that *satisfiability* (SAT) [3] does in NP-completeness theory. Like SAT, CVP is the fundamental P-complete problem in the sense that it is most frequently used to show that other problems are P-complete. Also like SAT, CVP has many restricted variants that are P-complete and can often simplify the construction of reductions. Table 45.3 summarizes these variants of CVP.

TABLE 45.3 Other P-Complete Variants of the Circuit Value Problem

Topologically Ordered CVP	A **topological ordering** is a vertex numbering so that the source vertex of each edge is less than the sink vertex. All CVP variants here remain P-complete even if we require that the vertices be presented in topological order in the circuit encoding (and thus the evaluation order is provided in the CVP instance).
NORCVP	CVP is restricted to contain only NOR gates. Reductions are often simpler when only one gate type needs to be simulated.
Monotone CVP (MCVP)	CVP restricted to **monotone** gates, that is, AND's and OR's. Useful in the situation where negations are hard to simulate.
Alternating, Monotone CVP (AMCVP)	A special case of MVCP. A monotone circuit is **alternating** if on any path from an input to an output the gates alternate between OR and AND. For AMCVP, inputs must connect only to OR gates, and outputs must be OR gates. Reductions often replace individual gates by certain small "gadgets." The alternating property reduces the number and kinds of interactions between gadgets that must be considered.
Fanin 2, Fanout 2 AMCVP (AM2CVP)	A restriction of AMCVP, where all gates are restricted to have fanin and fanout two, with the obvious exception of inputs and outputs. Having a fixed fanout often simplifies reductions.
Synchronous AM2CVP (SAM2CVP)	Restriction of AM2CVP to synchronous circuits (defined in "Uniform Aggregates"). All output vertices are required to be on the highest level of the circuit, so that it can be partitioned into layers, with all edges going from one layer to the next higher one, and all outputs on the last layer. Note that in a circuit that is both alternating and synchronous, all gates on any given level must be of the same type. The fanin two and fanout two restrictions further imply that every level contains exactly the same number of vertices. This structural regularity simplifies some reductions.

There are two key ideas in the proof of Theorem 45.18. The first is that for any circuit there is a simple sequential algorithm that given any input to the circuit evaluates individual gates of the circuit in a fixed order, evaluating each exactly once, and arriving at the circuit's designated output value in polynomial time. Thus CVP is in P.

The second key idea (see Chapter 27 for full details) is used to show that CVP is P-complete. Every language $L \in P$ has an associated Turing machine that decides membership of a length n string x in L in time $T(n)$, polynomial in n. Any polynomially time-bounded Turing machine computation can only use a polynomial amount of tape, and this means that the state of the tape at any instant can be simulated by a constant depth polynomially sized circuit "slice". On an input of length n, the Turing machine runs for at most $T(n)$ steps, and so at most $T(n)$ tape simulation slices are required. Thus there is a circuit family associated with L such that the nth circuit of the family decides membership of all strings of length n. This reduces the question of $x \in L$ to whether a specific polynomially sized circuit with input x outputs the value 1, and therefore shows that CVP is P-hard.

Additional *P*-Complete Problems

There are hundreds of P-complete problems (see [21]), from many areas of computer science: circuit complexity, graph theory, graph searching, combinatorial optimization and flow, local optimality, logic, formal languages, algebra, geometry, real analysis, games, and miscellaneous topics. Here are three of the more interesting P-complete problems. This first problem is from graph theory and was proved P-complete in [7].

DEFINITION 45.19 Lexicographically First Maximal Independent Set (LFMIS)

Given: An undirected graph $G = (V, E)$ with an ordering on the vertices and a designated vertex v.

Problem: Is vertex v in the lexicographically first maximal independent set of G?

Below we prove LFMIS is P-complete.

THEOREM 45.19 *[7] The Lexicographically First Maximal Independent Set Problem is P-complete.*

PROOF Membership in P follows from the standard greedy algorithm: vertices are processed in numerical order and added to the independent set if they do not have an edge to any vertex already in the independent set.

Completeness follows by reducing the NOR circuit value problem (NORCVP) to LFMIS using the construction given in [21]. Without loss of generality, we assume the instance α of NORCVP has its gates numbered (starting from 1) in topological order with inputs numbered first and outputs last. Suppose y is the designated output gate in the instance of NORCVP. We construct from α an instance of LFMIS, namely an undirected graph G. The graph G will be exactly the same as the graph underlying the circuit α, except that we add a new vertex, numbered 0, that is adjacent to all 0-inputs of α. It is easy to verify by induction that a vertex i in G is included in the lexicographically first maximal independent set if and only if either i equals 0 (the new vertex), or gate i in α has value TRUE. A choice of v equal to y completes the reduction.

The proof that the reduction can be performed in NC amounts to showing that the required edges from the new vertex 0 to the 0-inputs can be produced easily. The remainder of the circuit connections are easy to output directly.

The next problem is from graph searching and was proved P-complete in [37].

DEFINITION 45.20 Lexicographically First Depth-first Search Ordering

Given: An undirected graph $G = (V, E)$ with fixed ordered adjacency lists, and two designated vertices u and v.

Problem: Is vertex u visited before vertex v in the depth-first search of G induced by the order of the adjacency lists?

The final problem is from formal languages and was proved P-complete in [26].

DEFINITION 45.21 Context-Free Grammar Membership

Given: A context-free grammar $G = (N, T, P, S)$ and a string $x \in T^*$.

Problem: Is $x \in L(G)$?

It is likely that all these problems are inherently sequential. However, each of the problems has a related variant that is feasibly highly parallel. For example, each of these are in NC: finding *some* maximal independent set, finding the lexicographically first depth-first numbering for directed *acyclic* graphs, and context-free grammar membership for ϵ-free (ϵ denotes the empty string) grammars. References to these algorithms can be found in [21]. In practice, when the problem you want to solve is P-complete, there is often a closely related variant that is feasible and highly parallel. Whether this variant is useful to you is of course another matter.

Open Problems

There are a few simple problems for which it is unknown whether the problem is in NC or is P-complete. We give some of these here. The original references for these, and other, problems may be found in [21].

DEFINITION 45.22 Integer Greatest Common Divisor (IntegerGCD)
Given: Two n-bit positive integers a and b.
Problem: Compute the greatest common divisor of a and b.

DEFINITION 45.23 Lexicographically First Maximal Matching (LFMM)
Given: An undirected graph $G = (V, E)$ with an ordering on its edges and distinguished edge $e \in E$.
Problem: Is e in the lexicographically first maximal matching of G? A matching is **maximal** if it cannot be extended.

DEFINITION 45.24 Directed or Undirected Depth-First Search (DFS)
Given: A graph $G = (V, E)$ and a vertex s.
Problem: Construct a depth-first search numbering of G starting from vertex s.

DEFINITION 45.25 Maximum Matching (MM)
Given: An undirected graph $G = (V, E)$.
Problem: Find a maximum matching of G. A **matching** is a subset of edges $E' \subseteq E$ such that no two edges in E' share a common endpoint. A matching is **maximum** if no matching of larger cardinality exists.

DEFINITION 45.26 Subtree Isomorphism (STI)
Given: Two unrooted trees $T = (V, E)$ and $T' = (V', E')$.
Problem: Is T *isomorphic* to a subtree of T'?

It is notable that LFMM is *CC-complete* (circuits composed of comparators), and that DFS, MM, and STI have randomized feasible highly parallel algorithms (i.e., they are in RNC, which is the class random NC).

45.7 Research Issues and Summary

Many issues in parallel computation remain unresolved. The big question of asymptotic complexity theory is whether every feasible sequential problem has a feasible highly parallel solution, that is, does NC equal P? Although we have strong evidence[6] suggesting there are fundamental limits to the speedup attainable through parallel computation, this evidence does not constitute proof. It seems that deciding this question will require a major breakthrough in complexity theory. Another important complexity question is whether randomization helps in highly parallel computation. It is clear that NC is a subset of RNC (the randomized version of NC), but not known if the two classes are equal. Finally, there are many problems that have feasible highly parallel algorithms but whose algorithms are not **optimal**. That is, their **work** (time-processor product) is more than a constant factor greater than the running time of the best

[6] We have never explicitly stated the evidence in this chapter due to its technical nature. Chapter 5 of [21] goes into the evidence in detail.

known sequential algorithm. Attempting to improve the time or processor bounds for such problems is one way of understanding the relationship between sequential and parallel computation.

From the standpoint of algorithm design, the most important issue is finding a parallel model that is more realistic. That is, a model that reflects the kinds of parallel machines that will actually be built in the foreseeable future, and on which algorithms will be implemented. The works [8, 9, 44, 45], are some first steps in this direction. The main assumption of these models is that all parallel machines will essentially consist of a moderate number (thousands not millions) of highly powerful individual processors each with substantial memory, and interconnected by a high (but not infinite) capacity network.

Each model is parameterized so that any given instance of these kinds of parallel machines can be modeled so as to account for the architectural features which dominate the design of a parallel algorithm for that machine. The parameters include the number of processors, the communication bandwidth on the interconnection network, the *latency* or *delay* in transmitting over the network, granularity, and the overhead of initiating communication over the network.

The presence of these parameters forces the designer of the parallel algorithm not only to account for the computational aspects of the problem (deciding what is to be done and by what processor), but also to consider the communication aspects of the problem (where should data be placed and how should accesses be scheduled). By putting equal emphasis on computation and communication, such models more accurately capture reality, while avoiding architectural details like the kind of interconnection network. The task is now to develop parallel algorithms that can be used for as wide a range of parameter values as possible, and thus be robust over a much broader class of real machines.

45.8 Defining Terms

Circuit Value Problem: [31]
Given: An encoding $\overline{\alpha}$ of a Boolean circuit α, inputs $x_1, \ldots, x_n$, and a designated output y.
Problem: Is output y of α TRUE on input $x_1, \ldots x_n$?

NC**:** The set of all languages L that are decidable in parallel time $(\log n)^{O(1)}$ and processors $n^{O(1)}$.

NC **many-one reducibility:** A language L is NC **many-one reducible** or NC **reducible** to L', written $L \leq_m^{NC} L'$, if there is a function f in FNC such that $x \in L$ if and only if $f(x) \in L'$.

P**:** The set of all languages L that are decidable in sequential time $n^{O(1)}$.

Parallel computation thesis: Sequential space is polynomially related to parallel time.

Path Systems: [4]
Given: A path system $P = (X, R, S, T)$, where $S \subseteq X$, $T \subseteq X$, and $R \subseteq X \times X \times X$.
Problem: Is there an admissible vertex in S? A vertex x is **admissible** if and only if $x \in T$, or there exists admissible $y, z \in X$ such that $(x, y, z) \in R$.

P**-complete:** A language L is P**-hard under** NC **reducibility** if $L' \leq_m^{NC} L$ for every $L' \in P$. A language L is P**-complete under** NC **reducibility** if $L \in P$ and L is P-hard.

Acknowledgments

A warm thanks to Mike Atallah for inviting us to work with him on this exciting project. A special thanks to Larry Ruzzo for many enlightening discussions about the material in this chapter. Thanks to the anonymous referees for carefully reading the chapter and providing us with many helpful suggestions. This chapter was written while Ray was on sabbatical at the Universitat Politècnica de Catalunya in Barcelona. The department's hospitality is greatly appreciated.

References

[1] Borodin, A., On relating time and space to size and depth. *SIAM Journal on Computing,* 6(4), 733–744, 1977.

[2] Chandra, A.K., Kozen, D.C., and Stockmeyer, L.J., Alternation. *Journal of the ACM,* 28(1), 114–133, 1981.

[3] Cook, S.A., The complexity of theorem proving procedures. In *Conference Record of Third Annual ACM Symposium on Theory of Computing,* pages 151–158, Shaker Heights, OH, 1971.

[4] Cook, S.A., An observation on time-storage trade off. *Journal of Computer and System Sciences,* 9(3), 308–316, 1974.

[5] Cook, S.A., Deterministic CFL's are accepted simultaneously in polynomial time and log squared space. In *Conference Record of the Eleventh Annual ACM Symposium on Theory of Computing,* 338–345, Atlanta, GA, 1979. See also [49].

[6] Cook, S.A., Towards a complexity theory of synchronous parallel computation. *L'Enseignement Mathématique,* XXVII(1–2), 99–124, 1981. Also in [35, pages 75–100].

[7] Cook, S.A., A taxonomy of problems with fast parallel algorithms. *Information and Control,* 64(1–3), 2–22, 1985.

[8] Culler, D., Karp, R., Patterson, D., Sahay, A., Schauser, K.E., Santos, E., Subramonian, R., and von Eicken, T., Logp: Towards a realistic model of parallel computation. In *Proceedings of the Fifth Annual ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming,* 1–12, ACM, 1993.

[9] de la Torre, P. and Kruskal, C., Towards a single model of efficient computation in real parallel machines. *Future Generations Computer Systems,* 8, 395–408, 1992. Preliminary version in *PARLE'91: Parallel Architectures and Languages Europe,* Lecture Notes in Computer Science, Aars, van Leeuwen and Rems, Eds., Springer-Verlag, 6–24, Jul. 1991.

[10] Dymond, P.W., *Simultaneous Resource Bounds and Parallel Computation,* Ph.D. Thesis, University of Toronto. Department of Computer Science Technical Report 145/80, 1980.

[11] Dymond, P.W. and Cook, S.A., Hardware complexity and parallel computation. In *21st Annual Symposium on Foundations of Computer Science,* 360–372, Syracuse, NY. IEEE, 1980.

[12] Dymond, P.W. and Cook, S.A., Complexity theory of parallel time and hardware. *Information and Computation,* 80(3), 205–226, 1989.

[13] Dymond, P.W. and Ruzzo, W.L., Parallel random access machines with owned global memory and deterministic context-free language recognition. In *Automata, Languages, and Programming: 13th International Colloquium,* Kott, L., Ed., Vol. 226: *Lecture Notes in Computer Science,* 95–104, Rennes, France. Springer-Verlag, 1986.

[14] Fich, F.E., The complexity of computation on the parallel random access machine. In [38], chapter 20, 843–899, 1993.

[15] Fortune, S. and Wyllie, J.C., Parallelism in random access machines. In *Conference Record of the Tenth Annual ACM Symposium on Theory of Computing,* 114–118, San Diego, CA, 1978.

[16] Furst, M.L., Saxe, J.B., and Sipser, M., Parity, circuits, and the polynomial-time hierarchy. *Mathematical Systems Theory,* 17(1), 13–27, 1984.

[17] Gibbons, A.M. and Rytter, W., *Efficient Parallel Algorithms,* Cambridge University Press, 1988.

[18] Goldschlager, L.M., *Synchronous Parallel Computation,* Ph.D. Thesis, University of Toronto. Computer Science Department Technical Report 114, 1977.

[19] Goldschlager, L.M., A universal interconnection pattern for parallel computers. *Journal of the ACM,* 29(4), 1073–1086, 1982.

[20] Greenlaw, R., Polynomial completeness and parallel computation. In [38], chapter 21, 901–953, 1993.

[21] Greenlaw, R., Hoover, H.J., and Ruzzo, W.L., *Limits to Parallel Computation: P-Completeness Theory,* Oxford University Press, 1995.

[22] Hartmanis, J. and Simon, J., On the power of multiplication in random access machines. In *15th Annual Symposium on Switching and Automata Theory,* 13–23, 1974.
[23] JáJá, J., *An Introduction to Parallel Algorithms,* Addison-Wesley, 1992.
[24] Johnson, D.S., The *NP*-completeness column: An ongoing guide (7th). *Journal of Algorithms,* 4(2), 189–203, 1983.
[25] Johnson, D.S., A catalog of complexity classes. In [47], chapter 2, 67–161, 1990.
[26] Jones, N.D. and Laaser, W.T., Complete problems for deterministic polynomial time. In *Conference Record of Sixth Annual ACM Symposium on Theory of Computing,* 40–46, Seattle, WA, 1974.
[27] Karp, R.M. and Ramachandran, V., Parallel algorithms for shared-memory machines. In [47], chapter 17, 869–941, 1990.
[28] Kindervater, G.A.P. and Lenstra, J.K., An introduction to parallelism in combinatorial optimization. In *Parallel Computers and Computation,* van Leeuwen, J. and Lenstra, J.K., Eds., Vol. 9: *CWI Syllabus,* 163–184. Center for Mathematics and Computer Science, Amsterdam, The Netherlands, 1985a.
[29] Kindervater, G.A.P. and Lenstra, J.K., Parallel algorithms. In *Combinatorial Optimization: Annotated Bibliographies,* O'hEigeartaigh, M., Lenstra, J.K., and Rinnooy Kan, A.H.G., Eds., chapter 8, 106–128. John Wiley & Sons, Chichester, UK, 1985b.
[30] Kindervater, G.A.P. and Trienekens, H.W.J.M., Experiments with parallel algorithms for combinatorial problems. Technical Report 8550/A, Erasmus University Rotterdam, Econometric Inst., 1985.
[31] Ladner, R.E., The circuit value problem is log space complete for *P*. *SIGACT News,* 7(1), 18–20, 1975.
[32] Lam, T.W. and Ruzzo, W.L., The power of parallel pointer manipulation. In *Proceedings of the 1989 ACM Symposium on Parallel Algorithms and Architectures,* 92–102, Santa Fe, NM, 1989.
[33] Leighton, F.T., *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes,* Morgan Kaufmann, 1992.
[34] Lengauer, T., VLSI theory. In [47], chapter 16, 837–868, 1990.
[35] Logic and Algorithmic, *Logic and Algorithmic,* An International Symposium Held in Honor of Ernst Specker, Zürich, Febr. 5–11, 1980. Monographie No. 30 de L'Enseignement Mathématique, Université de Genève, 1982.
[36] Pratt, V.R. and Stockmeyer, L.J., A characterization of the power of vector machines. *Journal of Computer and System Sciences,* 12(2), 198–221, 1976.
[37] Reif, J.H., Depth-first search is inherently sequential. *Information Processing Letters,* 20(5), 229–234, 1985.
[38] Reif, J.H., Ed., *Synthesis of Parallel Algorithms,* Morgan Kaufmann, 1993.
[39] Ruzzo, W.L., On uniform circuit complexity. *Journal of Computer and System Sciences,* 22(3), 365–383, 1981.
[40] Savitch, W.J. and Stimson, M.J., Time bounded random access machines with parallel processing. *Journal of the ACM,* 26(1), 103–118, 1979.
[41] Schönhage, A., Storage modification machines. *SIAM Journal on Computing,* 9(3), 490–508, 1980.
[42] Spirakis, P.G., Fast parallel algorithms and the complexity of parallelism (basic issues and recent advances). In *Parcella'88. Fourth International Workshop on Parallel Processing by Cellular Automata and Arrays Proceedings,* Wolf, G., Legendi, T., and Schendel, U., Eds., volume 342 of *Lecture Notes in Computer Science,* 177–189, Berlin, East Germany. Springer-Verlag, 1988 (published 1989).
[43] Stockmeyer, L.J. and Vishkin, U., Simulation of parallel random access machines by circuits. *SIAM Journal on Computing,* 13(2), 409–422, 1984.

[44] Valiant, L.G., A bridging model for parallel computation. *Communications of the ACM,* 33(8), 103–111, 1990a.
[45] Valiant, L.G., General purpose parallel architectures. In [47], chapter 18, 943–971, 1990b.
[46] van Emde Boas, P., The second machine class: Models of parallelism. In *Parallel Computers and Computation,* van Leeuwen, J. and Lenstra, J.K., Eds., Vol. 9: *CWI Syllabus,* 133–161, 1985. Center for Mathematics and Computer Science, Amsterdam, The Netherlands.
[47] van Leeuwen, J., Ed., *Handbook of Theoretical Computer Science,* Vol. A: Algorithms and Complexity. M.I.T. Press/Elsevier, 1990.
[48] Vishkin, U., Synchronous parallel computation—a survey. Preprint. Courant Institute, New York University, 1983.
[49] von Braunmühl, B., Cook, S.A., Mehlhorn, K., and Verbeek, R., The recognition of deterministic CFL's in small time and space. *Information and Control,* 56(1-2), 34–51, 1983.

Further Information

Much of our material is adapted from *Limits to Parallel Computation* [21], and many more details can be found there. The relationships among various parallel models can be found in the excellent surveys [6, 14, 27]. Additional papers surveying other aspects of parallel models and parallel computing include [24, 29, 42, 46, 48].

The book *Efficient Parallel Algorithms* [17] has a brief discussion of parallel models of computation followed by substantial material on parallel algorithms. The text *An Introduction to Parallel Algorithms* [23] devotes a chapter to discussing parallel models and then extensively delves into parallel algorithms. The text *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes* [33] contains a detailed discussion of many different types of parallel models and algorithms. The collection *Synthesis of Parallel Algorithms* [38], contains about twenty chapters organized around parallel algorithms for particular types of problems, together with an introductory chapter on P-completeness [20], and one surveying PRAM models [14]. The chapter "Parallel Algorithms for Shared-Memory Machines" by [27] in the *Handbook of Theoretical Computer Science* [47] describes a variety of highly parallel algorithms. In the same handbook, the chapter "A Catalog of Complexity Classes" [25] is a thorough overview of basic complexity theory and of the current state of knowledge about most complexity classes. It is an excellent reference for establishing the context of each class and its established relationships to others. The papers [28, 29, 30] provide extensive bibliographies of papers about parallel algorithms and parallel algorithm development for combinatorial optimization problems.

46

Algorithmic Techniques for Networks of Processors

Russ Miller
State University of New York at Buffalo

Quentin F. Stout
University of Michigan

46.1 Introduction

This chapter is concerned with designing algorithms for machines constructed from multiple processors. In particular, we discuss algorithms for machines in which the processors are connected to each other by some simple, systematic, interconnection pattern. For example, consider a chess board, where each square represents a processor (for example, a processor similar to one in a home computer) and every generic processor is connected to its 4 neighboring processors (those to the north, south, east, and west). This is an example of a *mesh computer,* a network of processors that is important for both theoretical and practical reasons.

The focus of this chapter is on algorithmic techniques. Initially, we define some basic terminology that is used to discuss parallel algorithms and parallel architectures. Following this introductory material, we define a variety of **interconnection networks,** including the mesh (chess board), which are used to allow processors to communicate with each other. We also define an abstract parallel model of computation, the *PRAM,* where processors communicate with memory instead of with each other. We then discuss several parallel programming paradigms, including the use of high-level data movement operations, **divide-and-conquer, pipelining,** and **master–slave.** Finally, we discuss the problem of mapping the structure of an inherently parallel problem onto a target parallel architecture. This mapping problem can arise in a variety

of ways, and with a wide range of problem structures. In some cases, finding a good mapping is quite straightforward, but in other cases it is a computationally intractable NP-complete problem.

46.2 Terminology

In order to initiate our investigation, we first define some basic terminology that will be used throughout the remainder of this chapter.

Shared Memory vs. Distributed Memory

In a **shared memory** machine, there is a single global image of memory that is available to all processors in the machine, typically through a common bus or switching network (see Fig. 46.1). This model is similar to a blackboard, where any processor can read or write to any part of the board (memory), and where all communication is performed through messages placed on the board.

Each processor in a **distributed memory** machine has access only to its private (local) memory (see Fig. 46.1). In this model, processors communicate by sending messages to each other, with the messages being sent through some form of interconnection network. This model is similar to a school in which each professor occupies a unique classroom equipped with a blackboard. For professor W to access information maintained on the board of professor X, W sends a message to X requesting the information, and X sends a message back with the information. In this classroom scenario, messages might be transmitted by students running through the halls. In such message-passing systems, the overhead and delay can be significantly reduced if it can be arranged so that X sends the information to W without a request being sent. This is particularly useful if it can be arranged so that the data from X arrives before W needs to use it, for then W will not be delayed waiting for the data. This analogy represents an important aspect of developing efficient programs for distributed memory machines, especially general-purpose machines in which communication can take place concurrently with calculation so that the communication time is effectively hidden.

For small shared memory systems, it may be that the network is such that each processor can access all memory cells in the same amount of time. For example, many symmetric multiprocessor (SMP) systems have this property. However, since memory takes space, systems with a large number of processors are typically constructed as modules (i.e., a processor/memory pair) that are connected to each other via an interconnection network. Thus, while memory may be logically shared in such a model, in terms of performance each processor acts as if it is distributed, with some memory being "close" (fast access) to the processor and some memory being "far" (slow access) from the processor. Notice the similarity to distributed memory machines, where there is a significant difference in speed between a processor accessing its own memory versus a processor accessing the memory of a distant processor. Such shared memory machines are called NUMA (nonuniform memory access) machines, and often the most efficient programs for NUMA machines are developed by using algorithms efficient for distributed memory architectures, rather than using ones optimized for uniform access shared memory architectures.

The efficient use of an interconnection network in a parallel computer is often an important consideration in developing and tuning parallel programs. For example, in either shared or distributed memory machines, communication will be delayed if a packet of information must pass through many communication links. Similarly, communication will be delayed by contention if many packets need to pass through the same link. As an example of contention at a link, in a distributed memory machine configured as a binary tree of processors, suppose that all leaf processors on one side of the machine need to exchange values with all leaf processors on the other side of the machine. Then a bottleneck occurs at the root since the passage of information proceeds in a sequential manner through the links in and out of the root.

Both shared and distributed memory systems can also suffer from contention at the destinations. In a distributed memory system, too many processors may simultaneously send messages to the same processor,

which causes a processing bottleneck. In a shared memory system, there may be memory contention, where too many processors try to simultaneously read or write from the same location.

Another common feature of both shared and distributed memory systems is that the programmer has to be sure that computations are properly synchronized, i.e., that they occur in the correct order. This tends to be easier in distributed memory systems, where each processor controls the access to its data, and the messages used to communicate data also have the side-effect of communicating the status of the sending processor. For example, suppose processor W is calculating a value, which will then be sent to processor R. If the program is constructed so that R does not proceed until the message from W arrives, then it is guaranteed of using the correct value in the calculations. In a shared memory system, the programmer needs to be more careful. For example, in the same scenario, W may write the new value to a memory location that R reads. However, if R reads before W has written, then it may proceed using the wrong value. This is known as a *race condition,* where the correctness of the calculation depends on the order of the operations. To avoid this, various locking or signaling protocols need to be enforced so that R does not read the location until after W has written to it. Race conditions are a common source of programming errors, and are often difficult to locate because they disappear when a deterministic, serial debugging approach is used.

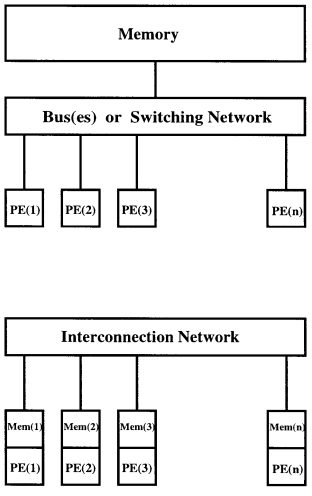


FIGURE 46.1 Shared memory (top) and distributed memory (bottom) machines. (*PE* is used represent a processing element and *Mem* is used to represent memory.)

Flynn's Taxonomy

In 1966, Michael Flynn classified computer architectures with respect to the *instruction stream,* that is, the sequence of operations performed by the computer, and the *data stream,* that is, the sequence of items operated on by the instructions [4]. While extensions and modifications to Flynn's taxonomy have appeared, Flynn's original taxonomy [5] is still widely used. Flynn characterized an architecture as belonging to one of the following four classes.

- Single-Instruction Stream, Single-Data Stream (SISD)
- Single-Instruction Stream, Multiple-Data Stream (SIMD)
- Multiple-Instruction Stream, Single-Data Stream (MISD)

- Multiple-Instruction Stream, Multiple Data Stream (MIMD)

Standard serial computers fall into the *single-instruction stream, single data stream (SISD)* category, in which one instruction is executed per unit time. This is the so-called von Neumann model of computing, in which the stream of instructions and the stream of data can be viewed as being tightly coupled, so that one instruction is executed per unit time to produce one useful result. Modern "serial" computers include various forms of modest parallelism in their execution of instructions, but most of this is hidden from the programmer and only appears in the form of faster execution of a sequential program.

A *single-instruction stream, multiple-data stream (SIMD)* machine typically consists of multiple processors, a control unit (controller), and an interconnection network, as shown in Fig. 46.2. The control unit

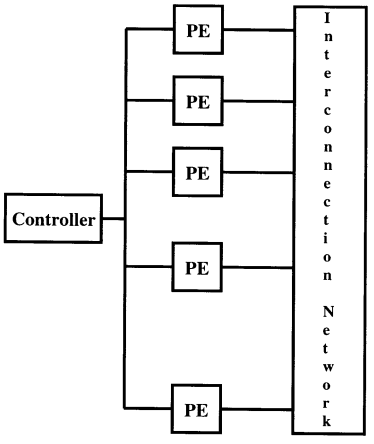


FIGURE 46.2 A SIMD machine. (*PE* is used to represent a processing element.)

stores the program and broadcasts the instructions to all processors simultaneously. Active processors execute the instruction on the contents of their own local memory. Through the use of a *mask*, processors may be in either an active or inactive state at any time during the execution of the program. Masks can be dynamically determined, based on local data or the processor's coordinates. Note that one side-effect of having a centralized controller is that the system is synchronous, so that no processor can execute a second instruction until all processors are finished with the first instruction. This is quite useful in algorithm design, as it eliminates many race conditions and makes it easier to reason about the status of processors and data.

Multiple-instruction stream, single-data stream (MISD) machines consist of two or more processors that perform separate instructions on the same data. This model is rarely implemented.

A *multiple-instruction stream, multiple-data stream (MIMD)* machine typically consists of multiple processors and an interconnection network. In contrast to the single-instruction stream model, the multiple-instruction stream model allows each of the processors to store and execute its own program, providing multiple instruction streams. Each processor fetches its own data on which to operate. (Thus, there are multiple data streams, as in the SIMD model.) Often, all processors are executing the same program, but may be in different portions of the program at any given instant. This is the *single-program multiple-data (SPMD)* style of programming, which is an important mode of programming because it is rarely feasible to have a large number of different programs for different processors. The SPMD style, like the SIMD architectures, also makes it somewhat simpler to reason about the status of data structures and processors.

MIMD machines have emerged as the most commonly used general-purpose parallel computers, and are available in a variety of configurations. Both shared and distributed memory machines are available,

as are mixed architectures where small numbers of processors are grouped together as a shared memory symmetric multiprocessor, and these SMPs are linked together in a distributed memory fashion.

Granularity

When discussing parallel architectures, the term *granularity* is often used to refer to the relative number and complexity of the processors. A *fine-grained machine* typically consists of a relatively large number of small, simple processors (in terms of local memory and computational power), while a *coarse-grained machine* typically consists of relatively few processors, each of which is large and powerful. Fine-grained machines typically fall into the SIMD category, where all processors operate in lockstep fashion (i.e., synchronously) on the contents of their own small, local, memory. Coarse-grained machines typically fall into the shared memory MIMD category, where processors operate asynchronously on the large, shared memory. Medium-grained machines are typically built from commodity microprocessors, and are found in both distributed and shared memory models, almost always in MIMD designs.

For a variety of reasons, medium-grained machines currently dominate the parallel computer marketplace in terms of number of installations. Because they utilize commodity processors and have the ability to efficiently perform as general-purpose (parallel) machines, medium-grained machines tend to have cost/performance advantages over systems utilizing special-purpose processors. In addition, they can also exploit much of the software written for their component processors. Fine-grained machines are difficult to use as general-purpose computers because it is often difficult to determine how to efficiently distribute the work to such simple processors. However, fine-grained machines can be quite effective in tasks such as image processing or pattern matching.

By analogy, one can also use the granularity terminology to describe data and algorithms. For example, a database is a coarse-grained view of data, while considering the individual records in the database is a fine-grained view of the same data.

46.3 Interconnection Networks

In this section, we discuss interconnection networks that are used for communication among processors in a distributed memory machine. First, we define some terminology. The *degree of processor* P is the number of other processors that P is directly connected to via bidirectional communication links. The *degree of the network* is the maximum degree of any processor in the network. The *distance* between two processors is the number of communication links on a shortest path between the processors. The *communication diameter* of the network is the maximum, over all pairs of processors, of the distance between the processors. The *bisection bandwidth* of the network corresponds to the minimum number of communication links that need to be removed (or cut) in order to partition the network into two pieces, each with the same number of processors. Goals for interconnection networks include minimizing the degree of the processors (to minimize the cost of building a processor), minimizing the communication diameter (to minimize the communication time for any single message), and maximizing the bisection bandwidth (to minimize contention when many messages are being sent concurrently). Unfortunately, these design goals are in conflict. Other important design goals include simplicity (to reduce the design costs for the hardware and software) and scalability (so that similar machines, with a range of sizes, can be produced).

Before defining some network models (i.e., distributed memory machines characterized by their interconnection networks), we briefly discuss the **parallel random access machine (PRAM),** which is an idealized parallel model of computation, with a unit-time communication diameter. The PRAM is a shared memory machine that consists of a set of identical processors, where all processors have unit-time access to any memory location. The appeal of a PRAM is that one can ignore issues of communication when designing algorithms, focusing instead on obtaining the maximum parallelism possible in order to mini-

mize the running time necessary to solve a given problem. The PRAM model typically assumes a SIMD strategy, so that operations are performed synchronously. If multiple processors try to simultaneously read or write from the same memory location, then a memory conflict occurs. There are several variations of the PRAM model targeted at handling these conflicts, ranging from the Exclusive Read Exclusive Write (EREW) model, which prohibits all such conflicts, to Concurrent Read Concurrent Write (CRCW) models, which have various ways of resolving the effects of simultaneous writes. One popular intermediate model is the concurrent read exclusive write (CREW) PRAM, in which there may be concurrent reads to a memory location, but not concurrent writes. For example, a classroom is usually conducted in a CREW manner. In the classroom, even if several students are writing simultaneously on the blackboard, they are doing so in different locations, and hence there are no write conflicts.

The PRAM does not use a regular interconnection scheme for communication and the unit-time memory access requirement is not scalable (i.e., it is not realistic for a large number of processors and memory). However, in creating parallel programs, it is sometimes useful to describe a PRAM algorithm and then either perform a stepwise simulation of every PRAM operation on the target machine, or perform a higher-level simulation by using **global operations.** In such settings, it is often useful to design the algorithm for a powerful CRCW PRAM model, since often the CRCW PRAM can solve a problem faster or more naturally than an EREW PRAM. Since one is not trying to construct an actual PRAM, objections to the difficulty of implementing CRCW are not relevant; rather, having a simpler and/or faster algorithm is the dominant consideration.

In the remainder of this section, several specific interconnection networks are defined. See Fig. 46.3 for illustrations of these. The networks defined in this section are among the most commonly utilized networks. However, additional networks have appeared in both the literature and in real machines, and variations of the basic networks described here are numerous.

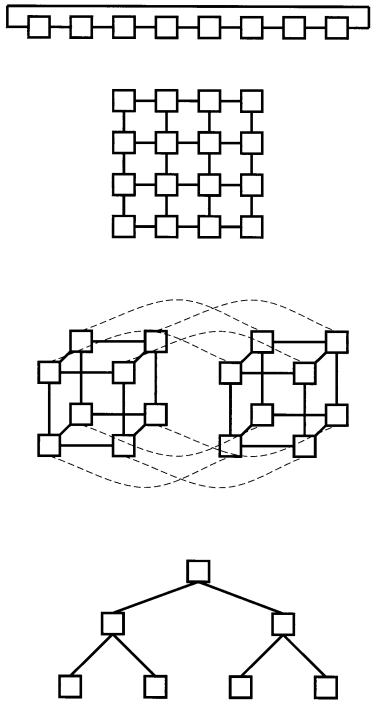


FIGURE 46.3 Sample interconnection networks (from top to bottom): ring, mesh, hypercube, and tree.

46.4 Ring

In a *ring* network, the n processors are connected in a circular fashion so that processor P_i is directly connected to processors P_{i-1} and P_{i+1} (the indices are computed modulo n, so that processors P_0 and P_{n-1} are connected). While the degree of the network is only 2, the communication diameter is $\lfloor n/2 \rfloor$, which is quite high. The bisection bandwidth is only 2, which is quite low.

Meshes and Tori

The n processors of a *two-dimensional square mesh* network are configured so that an interior processor $P_{i,j}$ is connected to its four neighbors, processors $P_{i-1,j}$, $P_{i+1,j}$, $P_{i,j-1}$, and $P_{i,j+1}$. The four corner processors are each connected to their two neighbors, while the remaining processors that are on the edge of the mesh are each connected to three neighbors. So, by increasing the degree of the network to 4, as compared to the degree 2 of the ring, the communication diameter of the network is reduced to $2(\sqrt{n}-1)$, and the bisection bandwidth is increased to $\sqrt{n}$. The diameter is further reduced, to $2\lfloor \sqrt{n}/2 \rfloor$, and the bisection bandwidth is increased, to $2\sqrt{n}$, in a *two-dimensional torus,* which has all the connections of the two-dimensional mesh plus connections between the first and last processors in each row and column. Meshes and tori of higher dimensions can be constructed, where the degree of a d-dimensional mesh or torus is $2d$, and, when n is a perfect dth power, the diameter is either $d(n^{1/d}-1)$ or $d\lfloor n^{1/d}/2 \rfloor$, respectively, and the bisection bandwidth is either $n^{(d-1)/d}$ or $2n^{(d-1)/d}$, respectively. Notice that the ring is a one-dimensional torus.

For a two-dimensional mesh, and similarly for higher-dimensional meshes, the mesh can be rectangular, instead of square. This allows a great deal of flexibility in selecting the size of the mesh, and the same flexibility is available for tori as well.

Hypercube

A *hypercube* with n processors, where n is an integral power of 2, has the processors indexed by the integers $\{0, \ldots, n-1\}$. Viewing each integer in this range as a $(\log_2 n)$-bit string, two processors are directly connected if and only if their indices differ by exactly one bit. Some advantages of a hypercube are that the communication diameter is only $\log_2 n$ and the bisection bandwidth is $n/2$. A disadvantage of the hypercube is that the number of communication links needed by each processor grows as $\log_2 n$, unlike the fixed degree for processors in ring and mesh networks. This makes it difficult to manufacture reasonably generic hypercube processors that could scale to extremely large machines, though in practice this is not a concern because the cost of an extremely large machine would be prohibitive.

Tree

A *complete binary tree* of height k, $k \geq 0$ an integer, has $n = 2^{k+1} - 1$ processors. The root node is at level 0 and the 2^k leaves are at level k. Each processor at level $1, \ldots, k-1$ has two children and one parent, the root processor does not have a parent processor, and the leaves at level k do not have children processors. Notice that the degree of the network is 3 and that the communication diameter is $2k = 2\lfloor \log_2 n \rfloor$. One severe disadvantage of a tree is that when extensive communication occurs, all messages traveling from one side of the tree to the other must pass through the root, causing a bottleneck. This is because the bisection bandwidth is only 1. Fat trees, introduced by Leiserson [9], alleviate this problem by increasing the bandwidth of the communication links near the root. This increase can come from changing the nature of the links, or, more easily, by using parallel communication links. Other generalizations of binary trees include complete t-ary trees of height k, where each processor at level $0, \ldots, k-1$ has t children. There are $(t^{k+1}-1)/(t-1)$ processors, the maximum degree is $t+1$, and the diameter is $2k = 2\lfloor \log_t n \rfloor$.

46.5 Designing Algorithms

Viewed from the highest level, many parallel algorithms are purely sequential, with the same overall structure as an algorithm designed for a more standard "serial" computer. That is, there may be an input and initialization phase, then a computational phase, and then an output and termination phase. The differences, however, are manifested within each phase. For example, during the computational phase, an efficient parallel algorithm may be inherently different from its efficient sequential counterpart.

For each of the phases of a parallel computation, it is often useful to think of operating on an entire structure simultaneously. This is a SIMD-style approach, but the operations may be quite complex. For example, one may want to update all entries in a matrix, tree, or database, and view this as a single (complex) operation. For a fine-grained machine, this might be implemented by having a single (or few) data item per processor, and then using a purely parallel algorithm for the operation. For example, suppose an $n \times n$ array A is stored on an $n \times n$ two-dimensional torus, so that $A(i, j)$ is stored on processor $P_{i,j}$. Suppose one wants to replace each value $A(i, j)$ with the average of itself and the four neighbors $A(i-1, j)$, $A(i+1, j)$, $A(i, j-1)$ and $A(i, j+1)$, where the indices are computed modulo n (i.e., "neighbors" is in the torus sense). This average filtering can be accomplished by just shifting the array right, left, up, and down by one position in the torus, and having each processor average the four values received along with its initial value.

For a medium- or coarse-grained machine, operating on entire structures is most likely to be implemented by blending serial and parallel approaches. On such an architecture, each processor uses an efficient serial algorithm applied to the portion of the data in its processor, and communicates with other processors in order to exchange critical data. For example, suppose the $n \times n$ array of the previous paragraph is stored in a $p \times p$ torus, where p evenly divides n, so that $A(i, j)$ is stored in processor $P_{\lfloor ip/n \rfloor, \lfloor jp/n \rfloor}$. Then, in order to do the same average filtering on A, each processor $P_{k,l}$ still needs to communicate with its torus neighbors $P_{k\pm1,l}$, $P_{k,l\pm1}$, but now sends them either the leftmost or rightmost column of data, or the topmost or bottommost row. Once a processor receives its boundary set of data from its neighboring processors, it can then proceed serially through its subsquare of data and produce the desired results. To maximize efficiency, this can be performed by having each processor send the data needed by its neighbors, then perform the filtering on the part of the array that it contains that does not depend on data from the neighbors, and then finally perform the filtering on the elements that depend on the data from neighbors. Unfortunately, while this maximizes the possible overlap between communication and calculation, it also complicates the program because the order of computations within a processor needs to be rearranged.

Global Operations

To manipulate entire structures in one step, it is useful to have a collection of operations that perform such manipulations. These *global operations* may be very problem-dependent, but certain ones have been found to be widely useful. For example, the average filtering example above made use of shift operations to move an array around. *Broadcast* is another common global operation, used to send data from one processor to all other processors. Extensions of the broadcast operation include simultaneously performing a broadcast within every (predetermined and distinct) subset of processors. For example, suppose matrix A has been partitioned into submatrices allocated to different processors, and one needs to broadcast the first row of A so that if a processor contains any elements of column i then it obtains the value of $A(1, i)$. In this situation, the more general form of a subset-based broadcast can be used.

Besides operating within subsets of processors, many global operations are defined in terms of a commutative, associative, semigroup operator $\otimes$. Examples of such semigroup operators include `minimum`, `maximum`, `or`, `and`, `sum`, and `product`. For example, suppose there is a set of values $V(i)$, $1 \le i \le n$, and

the goal is to obtain the maximum of these values. Then $\otimes$ would represent maximum, and the operation of applying $\otimes$ to all n values is called *reduction.* If the value of the reduction is broadcast to all processors, then it is sometimes known as *report.* A more general form of the reduction operation involves labeled data items, i.e., each data item is embedded in a record that also contains a label, where at the end of the reduction operation the result of applying $\otimes$ to all values with the same label will be recorded in the record.

Global operations provide a useful way to describe major actions in parallel programs. Further, since several of these operations are widely useful, they are often made available in highly optimized implementations. The language APL provided a model for several of these operations, and some parallel versions of APL have appeared. Languages such as C* [15] and FORTRAN 90 [3] also provide for some forms of global operations, as do message-passing systems such as MPI [14]. Reduction operations are so important that most parallelizing compilers detect them automatically, even if they have no explicit support for other global operations.

Besides broadcast, reduction, and shift, other important global operations include the following.

Sort: Let $X = \{x_0, x_1, \ldots, x_{n-1}\}$ be an ordered set such that $x_i < x_{i+1}$, for all $0 \leq i < n-1$. (That is, X is a subset of a linearly ordered data type.) Given that the n elements of X are arbitrarily distributed among a set of p processors, the sort operation will (re)arrange the members of X so that they are ordered with respect to the processors. That is, after sorting, elements $x_0, \ldots, x_{\lfloor n/p \rfloor}$ will be in the first processor, elements $x_{\lfloor n/p \rfloor + 1}, \ldots, x_{\lfloor 2n/p \rfloor}$ will be in the second processor, and so forth. Note that this assumes an ordering on the processors, as well as on the elements.

Merge: Suppose that sets D_1 and D_2 are subsets of some linearly ordered data type, and D_1 and D_2 are each distributed in an ordered fashion among disjoint sets of processors $\mathcal{P}_1$ and $\mathcal{P}_2$, respectively. Then the merge operation combines D_1 and D_2 to yield a single sorted set stored in ordered fashion in the entire set of processors $\mathcal{P} = \mathcal{P}_1 \cup \mathcal{P}_2$.

Associative Read/Write: These operations start with a set of *master* records indexed by unique keys. In the associative read, each processor specifies a key and ends up with the data in the master record indexed by that key, if such a record exists, or else a flag indicating that there is no such record. In the associative write, each processor specifies a key and a value, and each master record is updated by applying $\otimes$ to all values sent to it. (Master records are generated for all keys written.)

These operations are extensions of the CRCW PRAM operations. They model a PRAM with associative memory and a powerful combining operation for concurrent writes. On most distributed memory machines, the time to perform these more powerful operations is within a multiplicative constant of the time needed to simulate the usual concurrent read and concurrent write, and the use of the more powerful operations can result in significant algorithmic simplifications and speedups.

Compression: Compression moves data into a region of the machine where optimal interprocessor communication is possible. For example, compressing k items in a fine-grained two-dimensional mesh will move them to a $\sqrt{k} \times \sqrt{k}$ subsquare.

Parallel Prefix (Scan): Given a set of values a_i, $1 \leq i \leq n$, the *parallel prefix* computation determines $p_i = a_1 \otimes a_2 \otimes \cdots \otimes a_i$, for all i. This operation is available in APL, where it is called *scan.* Note that the hardware feature known as "fetch-and-op" implements a variant of parallel prefix, where "op" is $\otimes$ and the ordering of the processors is not required to be deterministic.

All-to-All Broadcast: Given data $D(i)$ in processor i, every processor receives a copy of $D(i)$, for all i.

All-to-All Personalized Communication: Every processor P_i has a data item $D(i, j)$ that is sent to processor P_j, for all $i \neq j$.

Example: Maximal Point Problem

As an example of the use of global operations, consider the following problem from computational geometry. Let S be a finite set of planar (i.e., two-dimensional) points. A point $p = (p_x, p_y)$ in S is a *maximal point* of S if $p_x > q_x$ or $p_y > q_y$, for every point $(q_x, q_y) \neq p$ in S. The *maximal point problem* is to determine all maximal points of S. See Fig. 46.4. The following parallel algorithm for the maximal point problem was apparently first noted by Atallah and Goodrich [2].

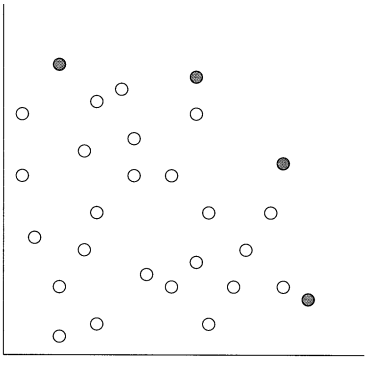


FIGURE 46.4 The maximal points of the set are shaded.

1. Sort the n planar points in reverse order by x-coordinate, with ties broken by reverse order by y-coordinate. Let (i_x, i_y) denote the coordinates of the ith point after the sort is complete. Therefore, after sorting, the points will be ordered so that if $i < j$ then either $i_x > j_x$ or $i_x = j_x$ and $i_y > j_y$.
2. Use parallel prefix on the i_y values, where the operation $\otimes$ is taken to be maximum. The resulting values $\{L_i\}$ are such that L_i is the largest y-coordinate of any point with index less than i.
3. The point (i_x, i_y) is an extreme point if and only if $i_y > L_i$.

The running time $T(n)$ of this algorithm is given by

$$T(n) = Sort(n) + Prefix(n) + O(1) , \tag{46.1}$$

where $Sort(n)$ is the time to sort n items and $Prefix(n)$ is the time to perform parallel prefix. On all parallel architectures known to the authors, $Prefix(n) = O(Sort(n))$, and hence on such machines the time of the algorithm is $\Theta(Sort(n))$. It is worth noting that for the sequential model, it has been shown [7] that the problem of determining maximal points is as hard as sorting.

Divide-and-Conquer

Divide-and-conquer is a powerful algorithmic paradigm that exploits the repeated subdivision of problems and data into smaller, similar problems. It is quite useful in parallel computation because the logical subdivisions into subproblems can correspond to physical decomposition among processors, where eventually the problem is broken into subproblems that are each contained within a single processor. These small subproblems are typically solved by an efficient sequential algorithm within each processor.

As an example, consider the problem of labeling the figures of a black/white image, where the interpretation is that of black objects on a white background. Two black pixels are defined to be *adjacent* if they are vertical or horizontal neighbors, and *connected* if there is a path of adjacent black pixels between them.

A *figure* (i.e., *connected component*) is defined to be a maximally connected set of black pixels in the image. The figures of an image are said to be *labeled* if every black pixel in the image has a label, with two black pixels having the same label if and only if they are in the same figure.

We utilize a generic parallel divide-and-conquer solution for this problem, given, for example, in [11]. Suppose that the $n \times n$ image has been divided into p subimages, as square as possible, and distributed one subimage per processor. Each processor labels the subimage it contains, using whatever serial algorithm is best and using labels that are unique to the processor (so that no two different figures can accidentally get the same label). For example, often the label used is a concatenation of the row and column coordinates of one of the pixels in the figure. Notice that so as long as the global row and column coordinates are used, the labels will be unique. After this step, the only figures that could have an incorrect global label are those that lie in two or more subimages, and any such figures must have a pixel on the border of each subimage it is in (see Fig. 46.5). To resolve these labels, a record is prepared for each black pixel on the border of a subimage, where the record contains information about the pixel's position in the image, and its current label. There are far fewer such records than there are pixels in the original image, yet they contain all of the information needed to determine the proper global labels for figures crossing subimages. The problem of reconciling the local labels may itself be solved via divide-and-conquer, repeatedly merging results from adjacent regions, or may be solved via other approaches. Once these labels have been resolved, information is sent back to the processors generating the records, informing them of the proper final label.

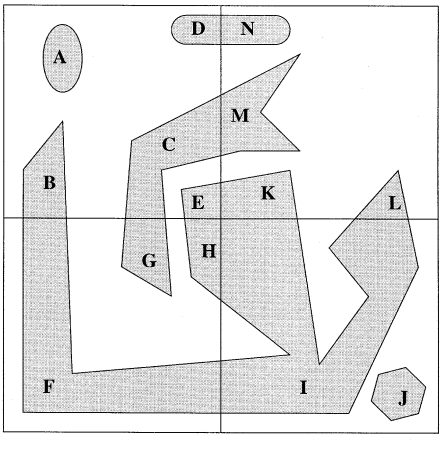


FIGURE 46.5 Divide-and-conquer for labeling figures. The 14 labels shown were generated after each quadrant performed its own local labeling algorithm. While the labels are unique, they need to be resolved globally. Notice that once the labels are resolved (not shown), the image will have only 5 unique labels, one corresponding to each of the 5 figures.

One useful feature of many of the networks described in the section on Interconnection Networks is that they can be divided into similar subnetworks, in a manner that matches the divide-and-conquer paradigm. For example, if the component labeling algorithm just described were performed on a mesh computer, then each subregion of the image would correspond to a subsquare of the mesh. As another example, consider an implementation of quicksort on a hypercube. Suppose a pivot is chosen and that the data is partitioned into items smaller than the pivot and items larger than the pivot. Further, suppose that the hypercube is logically partitioned into two subcubes, where all of the small items are moved into one subcube and all of the large items are moved into the other subcube. Now, the quicksort routine

may proceed recursively within each subcube. Because the recursive divide-and-conquer occurs within subcubes, all of the communication will occur within the subcubes and not cause contention with the other subcube.

Master–Slave

One algorithmic paradigm based on real-world organization paradigms is the master–slave (sometimes referred to as manager–worker) paradigm. In this approach, one processor acts as the master, directing all of the other slave processors. For example, many branch-and-bound approaches to optimization problems keep track of the best solution found so far, as well as a list of subproblems that need to be explored. In a master–slave implementation, the master maintains both of these items and is responsible for parceling out the subproblems to the slaves. The slaves are responsible for processing the subproblems and reporting the result to the master (which will determine if it is the current best solution), reporting new subproblems that need to be explored to the master, and notifying the master when it is free to work on a new subproblem. There are many variations on this theme, but the basic idea is that one processor is responsible for overall coordination, and the other processors are responsible for solving assigned subproblems. Note that this is a variant of the SPMD style of programming, in that there are two programs needed, rather than just one.

Pipelining and Systolic Algorithms

Another common parallel algorithmic technique is based on models that resemble an assembly line. A large problem, such as analyzing a number of images, may be broken into a sequence of steps that must be performed on each image (e.g., filtering, labeling, scene analysis). If one had three processors, and if each step takes about the same amount of time, one could start the first image on the first processor that does the filtering. Then the first image is passed on to the next processor for labeling, while the first processor starts filtering the second image. In the third time step, the initial image is at the third processor for scene analysis, the second image is at the second processor for labeling, and the third image is at the first processor for filtering. This form of processing is called *pipelining,* and it maps naturally to a parallel computer configured as a linear array (i.e., a one-dimensional mesh or, equivalently, a ring without the wraparound connection).

This simple scenario can be extended in many ways. For example, as in a real assembly line, the processors need not all be identical, and may be optimized for their task. Also, if some task takes longer to perform than others, then more than one processor can be assigned to it. Finally, the flow may not be a simple line. For example, an automobile assembly process may have one line working on the chassis, while a different line is working on the engine, and eventually these two lines are merged. Such generalized pipelining is called *systolic processing.* For example, some matrix and image-processing operations can be performed in a two-dimensional systolic manner (see [16]).

46.6 Mappings

Often, a problem has a natural structure to be exploited for parallelism, and this needs to be mapped onto a target machine. Several examples follow.

- The average filtering problem, discussed in the section on Designing Algorithms, has a natural array structure that can easily be mapped onto a mesh computer. If, however, one had the same problem, but a tree computer, then the mapping might be much more complicated.
- Some artificial intelligence paradigms exploit a blackboard-like communication mechanism that naturally maps onto a shared memory machine. However, a blackboard-like approach is more difficult to map onto a distributed-memory machine.

- Finite-element decompositions have a natural structure whereby calculations at each grid point depend only on values at adjacent points. A finite-element approach is frequently used to model automobiles, airplanes, and rocket exhaust, to name a few. However, the irregular (and perhaps dynamic) structure of such decompositions might need to be mapped onto a target parallel architecture that bears little resemblance to the finite-element grid.
- A more traditional example consists of porting a parallel algorithm designed for one parallel architecture onto another parallel architecture.

In all of these examples, one starts with a source structure that needs to be mapped onto a target machine. The goal is to map the source structure onto the target architecture so that calculation and communication steps on the source structure can be efficiently performed by the target architecture. Usually, the most critical aspect is to map the calculations of the source structure onto the processors of the target machine, so that each processor performs the same amount of calculations. For example, if the source is an array, and each position of the array represents calculations that need to be performed, then one tries to map the array onto the machine so that all processors contain the same number of entries. If the source model is a shared-memory paradigm with agents reading from a blackboard, then one would map the agents to processors, trying to balance the computational work.

Besides trying to balance the computational load, one must also try to minimize the time spent on communication. The approaches used for these mappings depend on the source structure and target architecture, and some of the more widely used approaches are discussed in the following subsections.

Simulating Shared Memory

If the source structure is a shared memory model, and the target architecture is a distributed memory machine, then besides mapping the calculations of the source onto the processors of the target, one must also map the shared memory of the source onto the distributed memory of the target.

To map the memory onto the target machine, suppose that there are memory locations $0 \ldots n-1$ in the source structure, and p processors in the target. Typically one would map locations $0 \ldots \lfloor n/p - 1 \rfloor$ to processor 0 of the target machine, locations $\lfloor n/p \rfloor \ldots \lfloor 2n/p - 1 \rfloor$ to processor 1, and so forth. Such a simple mapping balances the amount of memory being simulated by each target processor, and makes it easy to determine where data is located. For example, if a target processor needs to read from shared memory location i, it sends a message to target processor $\lfloor ip/n \rfloor$ asking for the contents of simulated shared memory location i.

Unfortunately, some shared memory algorithms utilize certain memory locations far more often than others, which can cause bottlenecks in terms of getting data in and out of processors holding the popular locations. If popular memory locations form contiguous blocks, then this congestion can be alleviated by stripping (mapping memory location i to processor $i \bmod p$) or other mappings. Replication (having copies of frequently read locations in more than one processor) or adaptive mapping (dynamically moving simulated memory locations from heavily loaded processors to lightly loaded ones) are occasionally employed to relieve congestion, but such techniques are more complicated and involve additional overhead.

Simulating Distributed Memory

It is often useful to view distributed memory machines as graphs. Processors in the machine are represented by vertices of the graph, and communication links in the machine are represented by edges in the graph. Similarly, it is often convenient to view the structure of a problem as a graph, where vertices represent work that needs to be performed, and edges represent values that need to be communicated in order to perform the desired work. For example, in a finite-element decomposition, the vertices of a decomposition might represent calculations that need to be performed, while the edges correspond to flow of data. That is, in

a typical finite-element problem, if there is an edge from vertex p to vertex q, then the value of q at time t depends on the values of q and p at time $t-1$. (Most finite-element decompositions are symmetric, so that p at time t would also depend on q at time $t-1$.) Questions about mapping the structure of a problem onto a target architecture can then be answered by considering various operations on the related graphs.

The best situation is when the graph representing the structure of a problem is a subgraph of the graph representing the target architecture. For example, if the structure of a problem was represented as a connected string of p vertices and the target architecture was a ring of p processors, then the mapping of the problem onto the architecture would be straightforward and efficient. In graph terms, this is described through the notion of embedding. An *embedding* of an undirected graph $G = (V, E)$ (i.e., G has vertex set V and edges E) into an undirected graph $G' = (V', E')$ is a mapping ϕ of V into V' such that

- every pair of distinct vertices $u, v \in V$, map to distinct vertices $\phi(u), \phi(v) \in V'$, and
- for every edge $\{u, v\} \in E$, $\{\phi(u), \phi(v)\}$ is an edge in E'.

Let G represent the graph corresponding to the structure of a problem (i.e., the *source structure*) and let G' represent the graph corresponding to the target architecture. Notice that if there is an embedding of G into G', then values that need to be communicated may be transmitted by a single communication step in the target architecture represented by G'. The fact that embeddings map distinct vertices of G to distinct vertices of G' ensures that a single calculation step for the problem can be simulated in a single calculation step of the target architecture.

One reason that hypercube computers were quite popular is that many graphs can be embedded into the hypercube (graph). An embedding of the one-dimensional ring of size 2^d into a d-dimensional hypercube is called a *d-dimensional Gray code*. In other words, if $\{0, 1\}^d$ denotes the set of all d-bit binary strings, then the d-dimensional Gray code G_d is a 1-1 map of $0 \ldots 2^d - 1$ onto $\{0, 1\}^d$, such that $G_d(j)$ and $G_d((j+1) \bmod 2^d)$ differ by a single bit, for $0 \le j \le 2^d - 1$. The most common Gray codes, called *reflected binary* Gray codes, are recursively defined as follows: $\mathcal{G}_d$ is a 1–1 mapping from $\{0, 1, \ldots, 2^d - 1\}$ onto $\{0, 1\}^d$, given by $\mathcal{G}_1(0) = 0$, $\mathcal{G}_1(1) = 1$, and for $d \ge 2$,

$$\mathcal{G}_d(x) = \begin{cases} 0\mathcal{G}_{d-1}(x) & 0 \le x \le 2^{d-1} - 1 \\ 1\mathcal{G}_{d-1}\left(2^d - 1 - x\right) & 2^{d-1} \le x \le 2^d - 1 \,. \end{cases} \tag{46.2}$$

Alternatively, the same Gray code can be defined in a nonrecursive fashion as $\mathcal{G}_d(x) = x \oplus \lfloor x/2 \rfloor$, where x and $\lfloor x/2 \rfloor$ are interpreted as d-bit strings. Further, the inverse of the reflected binary Gray code can be determined by

$$\mathcal{G}_d^{-1}\left(y_0 \ldots y_{d-1}\right) = x_0 \ldots x_{d-1} \,, \tag{46.3}$$

where $x_{d-1} = y_{d-1}$, and $x_i = y_{d-1} \oplus \cdots \oplus y_i$ for $0 \le i < d-1$.

Meshes can also be embedded into hypercubes. Let M be a d-dimensional mesh of size $m_1 \times m_2 \times \cdots \times m_d$, and let $r = \Sigma_{i=1}^{d} \lceil \log_2 m_i \rceil$. Then M can be embedded into the hypercube of size 2^r. To see this, let $r_i = \lceil \log_2 m_i \rceil$, for $1 \le i \le d$. Let ϕ be the mapping of mesh node $(a_1, \ldots, a_d)$ to the hypercube node which has as its label the concatenation $G_{r_1}(a_1) \cdot \ldots \cdot G_{r_d}(a_d)$, where G_{r_i} denotes any r_i-bit Gray code. Then ϕ is an embedding. Wrapped dimensions can be handled using reflected Gray codes rather than arbitrary ones. (A mesh M is *wrapped* in dimension j if, in addition to the normal mesh adjacencies, vertices with indices of the form $(a_1, \ldots, a_{j-1}, 0, a_{j+1}, \ldots, a_d)$ and $(a_1, \ldots, a_{j-1}, m_j - 1, a_{j+1}, \ldots, a_d)$ are adjacent. A torus is a mesh wrapped in all dimensions.) If dimension j is wrapped and m_j is an integral power of 2, then the mapping ϕ suffices. If dimension j is wrapped and m_j is even, but not an integral power of 2, then to ensure that the first and last nodes in dimension j are mapped to adjacent hypercube nodes, use ϕ, but replace $G_{r_j}(a_j)$ with

$$\begin{cases} \mathcal{G}_{r_j}\left(a_j\right) & \text{if } 0 \le a_j \le m_j/2 - 1 \\ \mathcal{G}_{r_j}\left(a_j + 2^{r_j} - m_j\right) & \text{if } m_j/2 \le a_j \le m_j - 1 \,, \end{cases} \tag{46.4}$$

where $\mathcal{G}_{r_j}$ is the r_j-bit reflected binary Gray code. This construction ensures that $\mathcal{G}_{r_j}(m_j/2 - 1)$ and $\mathcal{G}_{r_j}(2^{r_j} - m_j/2)$ differ by exactly one bit (the highest-order one), which in turns ensures that the mapping takes mesh nodes neighboring in dimension j to hypercube neighbors.

Any tree T can be embedded into a $(|T| - 1)$-dimensional hypercube, where $|T|$ denotes the number of vertices in T, but this result is of little use since the target hypercube is exponentially larger than the source tree. Often one can map the tree into a more reasonably sized hypercube, but it is a difficult problem to determine the minimum dimension needed, and there are numerous papers on the subject.

In general, however, one cannot embed the source structure into the target architecture. For example, a complete binary tree of height 2, which contains 7 processors, cannot be embedded into a ring of any size. Therefore, one must consider weaker mappings, which allow for the possibility that the target machine has fewer processors than the source and does not contain the communication links of the source. A *weak embedding* of a directed source graph $G = (V, E)$ into a directed target graph $G' = (V', E')$ consists of

- a map ϕ_v of V into V', and
- a map ϕ_e of E onto *paths* in G', such that if $(u, v) \in E$ then $\phi_e((u, v))$ is a path from $\phi_v(u)$ to $\phi_v(v)$.

(Note that if G is undirected, each edge becomes two directed edges that may be mapped to different paths in G'. Most machines that are based on meshes, tori, or hypercubes have the property that a message from processor P to processor Q may not necessarily follow the same path as a message sent from processor Q to processor P, if P and Q are not adjacent.) The map ϕ_v shows how computations are mapped from the source onto the target, and the map ϕ_e shows the communication paths that will be used in the target.

There are several measures that are often used to describe the quality of a weak embedding (ϕ_v, ϕ_e) of G into G', including the following.

Processor Load: The maximum, over all vertices $v' \in V'$, of the number of vertices in V mapped onto v' by ϕ_v. Note that if all vertices of the source structure represent the same amount of computation, then the processor load is the maximum computational load by any processor in the target machine. The goal is to make the processor load as close as possible to $|V|/|V'|$. If vertices do not all represent the same amount of work, then one should use labeled vertices, where the label represents the amount of work, and try to minimize the maximum, over all vertices $v' \in V'$, of the sum of the labels of the vertices mapped onto v'.

Link Load (Link Congestion): The maximum, over all edges $(u', v') \in E'$, of the number of edges $(u, v) \in E$ such that (u', v') is part of the path $\phi_e((u, v))$. If all edges of the source structure represent the same amount of communication, then the link load represents the maximum amount of communication contending for a single communication link in the target architecture. As for processor load, if edges do not represent the same amount of communication, then weights should be balanced instead.

Dilation: The maximum, over all edges $(u, v) \in E$, of the path length of $\phi_e((u, v))$. The dilation represents the longest delay that would be needed to simulate a single communication step along an edge in the source, if that was the only communication being performed.

Expansion: The ratio of the number of vertices of G' divided by the number of vertices of G. As was noted in the example of trees embedding into hypercubes, large expansion is impractical. In practice, usually the real target machine has far fewer processors than the idealized source structure, so expansion is not a concern.

In some machines, dilation is an important measure of communication delay, but in most modern general-purpose machines it is far less important because each message has a relatively large start-up time that may be a few orders of magnitude larger than the time per link traversed. Link contention may still be a problem in such machines, but some solve this by increasing the bandwidth on links that would have heavy contention. For example, as noted earlier, *fat-trees* [9] add bandwidth near the root to avoid the

bottlenecks inherent in a tree architecture. This increases the bisection bandwidth, which reduces the link contention for communication that poorly matches the basic tree structure.

For machines with very large message start-up times, often the number of messages needed becomes a dominant communication issue. In such a machine, one may merely try to balance calculation load and minimize the number of messages each processor needs to send, ignoring other communication effects. The number of messages that a processor needs to send can be easily determined by noting that processors p and q communicate if there are adjacent vertices u and v in the source structure such that ϕ_v maps u to p and v to q.

For many graphs that cannot be embedded into a hypercube, there are nonetheless useful weak embeddings. For example, keeping the expansion as close to 1 as is possible (given the restriction that a hypercube has a power of 2 processors), one can map the complete binary tree onto the hypercube with unit link congestion, dilation two, and unit processor contention [8].

In general, however, finding an optimal weak embedding for a given source and target is an NP-complete problem. This problem, sometimes known as the *mapping problem,* is often solved by various heuristics. This is particularly true when the source structure is given by a finite-element decomposition or other approximation schemes for real entities, for in such cases the sources are often quite large and irregular. Fortunately, the fact that such sources often have an underlying geometric basis makes it easier to find fairly good mappings rather quickly.

For example, suppose the source structure is an irregular grid representing the surface of a three-dimensional airplane, and the target machine is a two-dimensional mesh. One might first project the airplane onto the x-y plane, ignoring the z-coordinates. Then one might locate a median x-coordinate, call it $\bar{x}$, where half of the plane's vertices lie to the left of $\bar{x}$ and half to the right. The vertices may then be mapped so that those that lie to the left of $\bar{x}$ are mapped onto the left half of the target machine, and those vertices that lie to the right of $\bar{x}$ are mapped to the right half of the target. In the left half of the target, one might locate the median y-coordinate, denoted $\bar{y}$, of the points mapped to that half, and map the points above $\bar{y}$ to the top-left quadrant of the target, and map points below $\bar{y}$ to the bottom-left. On the right half a similar operation would be performed for the points mapped to that side. Continuing in this recursive, divide-and-conquer manner, eventually the target machine would have been subdivided down into single processors, at which point the mapping would have been determined. This mapping is fairly straightforward, balances the processor load, and roughly keeps points adjacent in the grid near to each other in the target machine, and hence it does a reasonable approximation of minimizing communication time.

Note that if message start-up time is very high, then this probably would not minimize the number of messages sent by each processor, and in such a situation it may be better to partition the plane by cutting along only, say, the x-axis at each step. Each processor would end up with a cross-sectional slab, with all of the source vertices in given range of x-coordinates. If grid edges are not longer than the width of such a slab, then each processor would have to send messages to only two processors, namely the processor with the slab to the left and the processor with the slab to the right.

Other complications can arise because the nodes or edges of such sources may not all represent the same amount of computation or calculation, respectively, in which case weighted mappings are appropriate. A variety of programs are available that perform such mappings, and over time the quality of the mapping achieved, and the time to achieve it, has significantly improved. For irregular source structures, such packages are generally superior to what one would achieve without considerable effort.

A more serious complication is that the natural source structure may be dynamic, adding nodes or edges over time. In such situations one often needs to dynamically adjust the mapping to keep the computational load balanced and keep communication minimal. This introduces additional overhead, which one must weigh against the costs of not adjusting the imbalance. Often the dynamical remappings are made incrementally, moving only a little of the data to correct the worst imbalances. Deciding how often to check for imbalance, and how much to move, typically depends quite heavily on the problem being solved.

46.7 Research Issues and Summary

The development of parallel algorithms and efficient parallel programs lags significantly behind that of standard serial computers. This is perhaps due to the fact that only recently have parallel computers become commercially available. Therefore, parallel computing is in a rapidly growing phase, with important research and development still needed in almost all areas. Extensive theoretical and practical work continues in discovering parallel programming paradigms, in developing a wide range of efficient parallel algorithms, in developing ways to describe and manage parallelism, in developing techniques to automatically detect parallelism, and in developing libraries of parallel routines.

Another factor that has hindered parallel algorithm development is the fact that there are many different parallel computing models. As noted earlier, architectural differences can significantly affect the efficiency of an algorithm, and hence parallel algorithms have traditionally been tied to specific parallel models. One advance is that various hardware and software approaches are being developed to help hide some of the architectural differences. Thus, one may have, say, a distributed memory machine, but have a software system that allows the programmer to view it as a shared memory machine. While it is true that a programmer will usually only be able to achieve the highest performance by directly optimizing the code for a target machine, in many cases acceptable performance can be achieved without tying the code to excessive details of an architecture. This, then, allows code to be ported to a variety of machines, which encourages code development. In the past, extensive code revision was needed every time the code was ported to a new parallel machine, and this strongly discouraged many users who did not want to plan for an unending parade of changes.

Another factor that has limited parallel algorithm development is that most computer scientists were not trained in parallel computing. As the field matures, more courses will incorporate parallel computing and the situation will improve. However, many thousands of small shared-memory systems have already been purchased, often to be used as departmental or corporate compute servers. Unfortunately, do to the dearth of parallel programmers, many of these systems are used only to run concurrent serial programs, or to run turnkey parallel programs (such as databases). There is a serious need for professionals who are able to utilize the full power of these parallel machines, since there are a great many problems which are beyond the power of single processors.

46.8 Defining Terms

Distributed memory: Each processor only has access to only its own private (local) memory, and communicates with other processors via messages.

Divide-and-conquer: A programming paradigm whereby large problems are solved by decomposing them into smaller, yet similar, problems.

Global operations: Parallel operations that affect system-wide data structures.

Interconnection network: The communication system that links together all of the processors and memory of a parallel machine.

Master–slave: A parallel programming paradigm whereby a problem is broken into a collection of smaller problems, with a master processor keeping track of the subproblems and assigning them to the slave processors.

Parallel random access machine (PRAM): A theoretical shared-memory model, where typically the processors all execute the same instruction synchronously, and access to any memory location occurs in unit time.

Pipelining: A parallel programming paradigm that abstracts the notion of an assembly line. A task is broken into a sequence of fixed subtasks corresponding to the stations of an assembly line. A series of similar tasks is solved by starting one task through the subtask sequence, then starting

the next task through as soon as the previous task has finished its first subtask. At any point in time, several tasks are in various stages of completion.

Shared memory: All processors have the same global image of (and access to) all of the memory.

Single program multiple data (SPMD): The dominant style of parallel programming, where all of the processors utilize the same program, though each has its own data.

References

[1] Akl, S.G. and Lyon, K.A., *Parallel Computational Geometry,* Prentice-Hall, Englewood Cliffs, NJ, 1993.

[2] Atallah, M.J. and Goodrich, M.T., Efficient parallel solutions to geometric problems, *Journal of Parallel and Distributed Computing,* 3, (1986), 492-507, 1986.

[3] Brainerd, W.S., Goldberg, C., and Adams, J.C., *Programmers Guide to FORTRAN 90,* McGraw-Hill, New York, 1990.

[4] Flynn, M.J., Very high-speed computing systems, *Proc. of the IEEE,* 54(12), 1901–1909, 1966.

[5] Flynn, M.J., Some computer organizations and their effectiveness, *IEEE Transactions on Computers,* C-21, 948–960, 1972.

[6] JáJá, J., *An Introduction to Parallel Algorithms,* Addison-Wesley, Reading, MA, 1992.

[7] Kung, H.T., Luccio, F., and Preparata, F.P., On finding the maxima of a set of vectors, *Journal of the ACM,* 22(4), 469–476, 1975.

[8] Leighton, F.T., *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes,* Morgan Kaufmann, San Mateo, CA, 1992.

[9] Leiserson, C.E., Fat-trees: Universal networks for hardware-efficient supercomputing, *IEEE Transactions on Computers,* C-34(10), 892–901,1985.

[10] Li, H. and Stout, Q.F., *Reconfigurable Massively Parallel Computers,* Prentice Hall, Englewood Cliffs, NJ, 1991.

[11] Miller, R. and Stout, Q.F., *Parallel Algorithms for Regular Architectures: Meshes and Pyramids,* The MIT Press, Cambridge, MA, 1996.

[12] Quinn, M.J., *Parallel Computing Theory and Practice,* McGraw-Hill, New York, 1994.

[13] Reif, J., Ed., *Synthesis of Parallel Algorithms,* Morgan Kaufmann, San Mateo, CA, 1993.

[14] Snir, M., Otto, S.W., Huss-Lederman, S., Walker, D.W., and Dongarra, J., *MPI: The Complete Reference,* The MIT Press, Cambridge, MA, 1995.

[15] Thinking Machines Corporation, *C* Programming Guide,* Version 6.0.2, Cambridge, MA, 1991.

[16] Ullman, J.D., *Computational Aspects of VLSI,* Computer Science Press, Rockville, MD, 1984.

Further Information

A good introduction to parallel computing at the undergraduate level is *Parallel Computing: Theory and Practice* by Michael J. Quinn. This book provides a nice introduction to parallel computing, including parallel algorithms, parallel architectures, and parallel programming languages. *Parallel Algorithms for Regular Architectures: Meshes and Pyramids* by Russ Miller and Quentin F. Stout focuses on fundamental algorithms and paradigms for fine-grained machines. It advocates an approach of designing algorithms in terms of fundamental data movement operations, including sorting, concurrent read, and concurrent write. Such an approach allows one to port algorithms in an efficient manner between architectures. *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes* is a comprehensive book by F. Thomson Leighton that also focuses on fine-grained algorithms for several traditional interconnection networks. Finally, for the reader interested in algorithms for the PRAM, *An Introduction to Parallel*

Algorithms by J. JáJá covers fundamental algorithms in geometry, graph theory, and string matching. It also includes a chapter on randomized algorithms.

There are several professional societies that sponsor conferences, publish books, and publish journals in the area of parallel algorithms. These include the Association for Computing Machinery (ACM), which can be found at http://www.acm.org, The Institute for Electrical and Electronics Engineers, Inc. (IEEE), which can be found at http://www.ieee.org, and the Society for Industrial and Applied Mathematics (SIAM), which can be found at http://www.siam.org.

Since parallel computing has become so pervasive, most computer science journals cover work concerned with parallel and distributed systems. For example, one would expect a journal on programming languages to publish articles on languages for shared-memory machines, distributed memory machines, networks of workstations, and so forth. For several journals, however, the primary focus is on parallel algorithms. These journals include the *Journal for Parallel and Distributed Computing,* published by Academic Press (http://www.apnet.com), the *IEEE Transactions on Parallel and Distributed Systems* (http://computer.org/pubs/tpds), and for results that can be expressed in a condensed form, *Parallel Processing Letters,* published by World Scientific. Finally, several comprehensive journals should be mentioned that publish a fair number of articles on parallel algorithms. These include the *IEEE Transactions on Computers, Journal of the ACM,* and *SIAM Journal on Computing.*

Unfortunately, due to very long delays from submission to publication, most results that appear in journals (with the exception of *Parallel Processing Letters*) are actually quite old. (A delay of 3–5 years from submission to publication is not uncommon.) Recent results appear in a timely fashion in conferences, most of which are either peer reviewed or panel reviewed. The first conference devoted primarily to parallel computing is the International Conference on Parallel Processing (ICPP), which had its inaugural conference in 1972. Many landmark papers have been presented at ICPP, especially during the 1970s and 1980s. Proceedings from this conference have been published in recent years by the IEEE Computer Society and CRC. In recent years, the International Parallel Processing Symposium (IPPS) (http://www.ippsxx.org) has emerged as the premier conference devoted to parallel computing. IPPS is quite comprehensive in that in addition to the conference, it offers a wide variety of workshops and tutorials. Another conference that has matured nicely in recent years is the ACM Symposium on Parallel and Distributed Processing (SPDP). It is interesting to note that both IPPS and SPDP initially started as regional conferences. In fact, in 1998, IPPS and SPDP will hold a combined conference. A conference that focuses on very theoretical, primarily PRAM-based, algorithms is the ACM Symposium on Parallel Algorithms and Architectures (SPAA). This conference is an offshoot of the premier theoretical conferences in computer science, ACM Symposium on Theory of Computing (STOC) and IEEE Symposium on Foundations of Computer Science (FOCS). A conference that focuses on very large parallel systems is SC 'XY (http://www.supercomp.org), where XY represents the last two digits of the year. This conferences includes the presentation of the Gordon Bell Prize for best parallelization. Awards are given in various categories, such as highest sustained performance and best price/performance.

Finally, a variety of sites exist that can be used to effectively navigate the web, including the IEEE Technical Committee on Parallel Processing (IEEETCPP), which can be found at http://www.cs.buffalo.edu/tcpp. This site contains links to conferences, journals, people in the field, bibliographies on parallel processing, on-line course material, books, and so forth. Another nice site (http://www.computer.org/parascope) is currently maintained by David A. Bader (University of New Mexico) with support from the IEEE. Finally, several newsgroups cater to parallel computing, including comp.parallel and comp.arch.

47

Parallel Algorithms

Guy E. Blelloch
Carnegie Mellon University

Bruce M. Maggs
Carnegie Mellon University

47.1 Introduction

The subject of this chapter is the design and analysis of parallel algorithms. Most of today's algorithms are sequential, that is, they specify a sequence of steps in which each step consists of a single operation. These algorithms are well suited to today's computers, which basically perform operations in a sequential fashion. Although the speed at which sequential computers operate has been improving at an exponential rate for many years, the improvement is now coming at greater and greater cost. As a consequence, researchers have sought more cost-effective improvements by building "parallel" computers—computers that perform multiple operations in a single step. In order to solve a problem efficiently on a parallel computer, it is usually necessary to design an algorithm that specifies multiple operations on each step, i.e., a parallel algorithm.

As an example, consider the problem of computing the sum of a sequence A of n numbers. The standard algorithm computes the sum by making a single pass through the sequence, keeping a running sum of

the numbers seen so far. It is not difficult however, to devise an algorithm for computing the sum that performs many operations in parallel. For example, suppose that, in parallel, each element of A with an even index is paired and summed with the next element of A, which has an odd index, i.e., $A[0]$ is paired with $A[1]$, $A[2]$ with $A[3]$, and so on. The result is a new sequence of $\lceil n/2 \rceil$ numbers that sum to the same value as the sum that we wish to compute. This pairing and summing step can be repeated until, after $\lceil \log_2 n \rceil$ steps, a sequence consisting of a single value is produced, and this value is equal to the final sum.

The parallelism in an algorithm can yield improved performance on many different kinds of computers. For example, on a parallel computer, the operations in a parallel algorithm can be performed simultaneously by different processors. Furthermore, even on a single-processor computer the parallelism in an algorithm can be exploited by using multiple functional units, pipelined functional units, or pipelined memory systems. Thus, it is important to make a distinction between the parallelism in an algorithm and the ability of any particular computer to perform multiple operations in parallel. Of course, in order for a parallel algorithm to run efficiently on any type of computer, the algorithm must contain at least as much parallelism as the computer, for otherwise resources would be left idle. Unfortunately, the converse does not always hold: some parallel computers cannot efficiently execute all algorithms, even if the algorithms contain a great deal of parallelism. Experience has shown that it is more difficult to build a general-purpose parallel computer than a general-purpose sequential computer.

The remainder of this chapter consists of nine sections. We begin in Section 47.2 with a discussion of how to model parallel computers. Next, in Section 47.3 we cover some general techniques that have proven useful in the design of parallel algorithms. Sections 47.4 through Section 47.8 present algorithms for solving problems from different domains. We conclude in Section 47.9 with a discussion of current research topics, a collection of defining terms, and finally sources for further information.

Throughout this chapter, we assume that the reader has some familiarity with sequential algorithms and asymptotic notation and analysis.

47.2 Modeling Parallel Computations

The designer of a sequential algorithm typically formulates the algorithm using an abstract model of computation called the *random-access machine* (RAM) model [2, Chapter 1]. In this model, the machine consists of a single processor connected to a memory system. Each basic CPU operation, including arithmetic operations, logical operations, and memory accesses, requires one time step. The designer's goal is to develop an algorithm with modest time and memory requirements. The random-access machine model allows the algorithm designer to ignore many of the details of the computer on which the algorithm will ultimately be executed, but captures enough detail that the designer can predict with reasonable accuracy how the algorithm will perform.

Modeling parallel computations is more complicated than modeling sequential computations because in practice parallel computers tend to vary more in organization than do sequential computers. As a consequence, a large portion of the research on parallel algorithms has gone into the question of modeling, and many debates have raged over what the "right" model is, or about how practical various models are. Although there has been no consensus on the right model, this research has yielded a better understanding of the relationship between the models. Any discussion of parallel algorithms requires some understanding of the various models and the relationships among them.

In this chapter we divide parallel models into two classes: **multiprocessor models** and **work-depth models.** In the remainder of this section we discuss these two classes and how they are related.

Multiprocessor Models

A *multiprocessor model* is a generalization of the sequential RAM model in which there is more than one processor. Multiprocessor models can be classified into three basic types: local memory machine models, modular memory machine models, and parallel random-access machine (PRAM) models. Figure 47.1 illustrates the structure of these machine models. A local memory machine model consists of a set of n processors each with its own local memory. These processors are attached to a common communication network. A modular memory machine model consists of m memory modules and n processors all attached to a common network. An n-processor PRAM model consists of a set of n processors all connected to a common shared memory [32, 37, 38, 77].

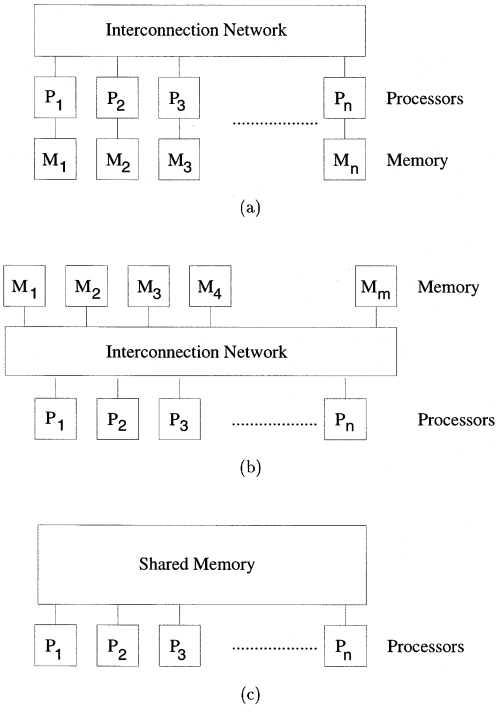


FIGURE 47.1 The three types of multiprocessor machine models. (a) A local memory machine model. (b) A modular memory machine model. (c) A parallel random-access machine (PRAM) model.

The three types of multiprocessors differ in the way that memory can be accessed. In a local memory machine model, each processor can access its own local memory directly, but can access the memory in another processor only by sending a memory request through the network. As in the RAM model, all local operations, including local memory accesses, take unit time. The time taken to access the memory in another processor, however, will depend on both the capabilities of the communication network and the pattern of memory accesses made by other processors, since these other accesses could congest the network. In a modular memory machine model, a processor accesses the memory in a memory module by sending a memory request through the network. Typically the processors and memory modules are arranged so that the time for any processor to access any memory module is roughly uniform. As in a local memory machine model, the exact amount of time depends on the communication network and the memory access pattern. In a **PRAM model,** a processor can access any word of memory in a single step. Furthermore, these accesses can occur in parallel, i.e., in a single step, every processor can access the shared memory.

The PRAM models are controversial because no real machine lives up to its ideal of unit-time access to shared memory. It is worth noting, however, that the ultimate purpose of an abstract model is not to directly model a real machine, but to help the algorithm designer produce efficient algorithms. Thus, if an algorithm designed for a PRAM model (or any other model) can be translated to an algorithm that runs efficiently on a real computer, then the model has succeeded. In "Emulations Among Models" we show how an algorithm designed for one parallel machine model can be translated so that it executes efficiently on another model.

The three types of multiprocessor models that we have defined are broad and allow for many variations. The local memory machine models and modular memory machine models may differ according to their network topologies. Furthermore, in all three types of models, there may be differences in the operations that the processors and networks are allowed to perform. In the remainder of this section we discuss some of the possibilities.

Network Topology

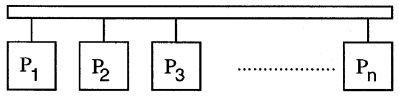
A network is a collection of switches connected by communication channels. A processor or memory module has one or more communication ports that are connected to these switches by communication channels. The pattern of interconnection of the switches is called the network topology. The topology of a network has a large influence on the performance and also on the cost and difficulty of constructing the network. Figure 47.2 illustrates several different topologies.

The simplest network topology is a bus. This network can be used in both local memory machine models and modular memory machine models. In either case, all processors and memory modules are typically connected to a single bus. In each step, at most one piece of data can be written onto the bus. This data might be a request from a processor to read or write a memory value, or it might be the response from the processor or memory module that holds the value. In practice, the advantage of using a bus is that it is simple to build and, because all processors and memory modules can observe the traffic on the bus, it is relatively easy to develop protocols that allow processors to cache memory values locally. The disadvantage of using a bus is that the processors have to take turns accessing the bus. Hence, as more processors are added to a bus, the average time to perform a memory access grows proportionately.

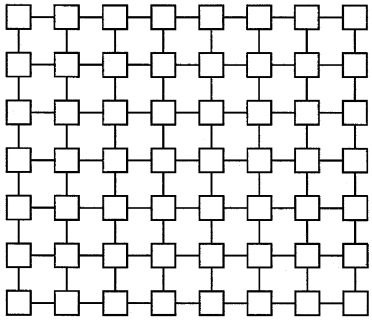
A two-dimensional *mesh* is a network that can be laid out in a rectangular fashion. Each switch in a mesh has a distinct label (x, y) where $0 \leq x \leq X - 1$ and $0 \leq y \leq Y - 1$. The values X and Y determine the length of the sides of the mesh. The number of switches in a mesh is thus $X \cdot Y$. Every switch, except those on the sides of the mesh, is connected to four neighbors: one to the north, one to the south, one to the east, and one to the west. Thus, a switch labeled (x, y), where $0 < x < X - 1$ and $0 < y < Y - 1$, is connected to switches $(x, y + 1)$, $(x, y - 1)$, $(x + 1, y)$, and $(x - 1, y)$. This network typically appears in a local memory machine model, i.e., a processor along with its local memory is connected to each switch, and remote memory accesses are made by routing messages through the mesh. Figure 47.2(b) shows an example of an 8×8 mesh.

Several variations on meshes are also popular, including three-dimensional meshes, toruses, and hypercubes. A *torus* is a mesh in which the switches on the sides have connections to the switches on the opposite sides. Thus, every switch (x, y) is connected to four other switches: $(x, y + 1 \bmod Y)$, $(x, y - 1 \bmod Y)$, $(x + 1 \bmod X, y)$, and $(x - 1 \bmod X, y)$. A hypercube is a network with 2^n switches in which each switch has a distinct n-bit label. Two switches are connected by a communication channel in a hypercube if and only if the labels of the switches differ in precisely one bit position. A hypercube with 16 switches is shown in Fig. 47.2(c).

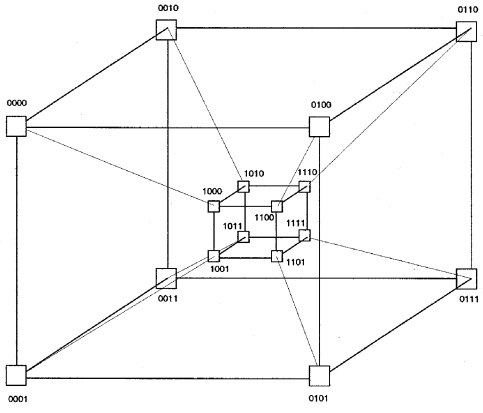
A *multistage network* is used to connect one set of switches called the *input switches* to another set called the *output switches* through a sequence of stages of switches. Such networks were originally designed for telephone networks [15]. The stages of a multistage network are numbered 1 through L, where L is the depth of the network. The switches on stage 1 are the input switches, and those on stage L are the output switches. In most multistage networks, it is possible to send a message from any input switch to any output



(a) Bus



(b) 2-dimensional Mesh



(c) Hypercube

FIGURE 47.2 Bus, mesh, and hypercube network topologies.

switch along a path that traverses the stages of the network in order from 1 to L. Multistage networks are frequently used in modular memory computers; typically processors are attached to input switches, and memory modules are attached to output switches. A processor accesses a word of memory by injecting a memory access request message into the network. This message then travels through the network to the appropriate memory module. If the request is to read a word of memory, then the memory module sends the data back through then network to the requesting processor. There are many different multistage network topologies. Figure 47.3(a), for example, shows a depth-2 network that connects 4 processors to 16 memory modules. Each switch in this network has two channels at the bottom and four channels at the top. The ratio of processors to memory modules in this example is chosen to reflect the fact that, in practice, a processor is capable of generating memory access requests faster than a memory module is capable of servicing them.

A *fat-tree* is a network structured like a tree [56]. Each edge of the tree, however, may represent many communication channels, and each node may represent many network switches (hence the name "fat"). Figure 47.3(b) shows a fat-tree with the overall structure of a binary tree. Typically the capacities of the edges near the root of the tree are much larger than the capacities near the leaves. For example, in this tree the two edges incident on the root represent 8 channels each, while the edges incident on the leaves

represent only 1 channel each. A natural way to construct a local memory machine model is to connect a processor along with its local memory to each leaf of the fat-tree. In this scheme, a message from one processor to another first travels up the tree to the least common-ancestor of the two processors, and then down the tree.

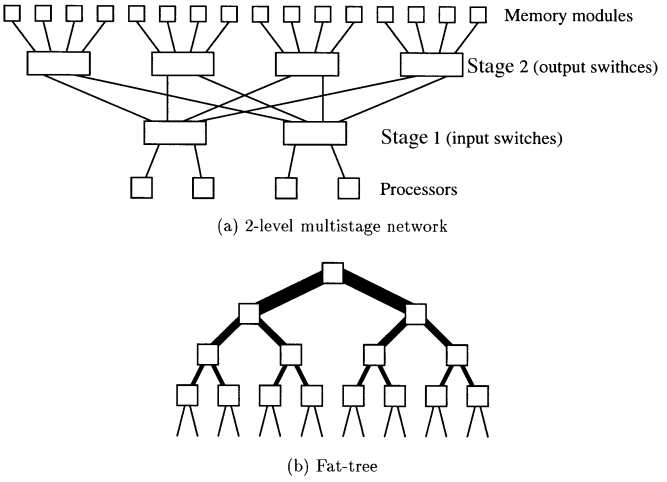


FIGURE 47.3 Multistage and fat-tree network topologies.

Many algorithms have been designed to run efficiently on particular network topologies such as the mesh or the hypercube. For extensive treatment such algorithms, see [55, 67, 73, 80]. Although this approach can lead to very fine-tuned algorithms, it has some disadvantages. First, algorithms designed for one network may not perform well on other networks. Hence, in order to solve a problem on a new machine, it may be necessary to design a new algorithm from scratch. Second, algorithms that take advantage of a particular network tend to be more complicated than algorithms designed for more abstract models like the PRAM models, because they must incorporate some of the details of the network. Nevertheless, there are some operations that are performed so frequently by a parallel machine that it makes sense to design a fine-tuned network-specific algorithm. For example, the algorithm that routes messages or memory access requests through the network should exploit the network topology. Other examples include algorithms for broadcasting a message from one processor to many other processors, for collecting the results computed in many processors in a single processor, and for synchronizing processors.

An alternative to modeling the topology of a network is to summarize its routing capabilities in terms of two parameters, its latency and bandwidth. The latency, L, of a network is the time it takes for a message to traverse the network. In actual networks this will depend on the topology of the network, which particular ports the message is passing between, and the congestion of messages in the network. The latency is often modeled by considering the worst-case time assuming that the network is not heavily congested. The bandwidth at each port of the network is the rate at which a processor can inject data into the network. In actual networks this will depend on the topology of the network, the bandwidths of the network's individual communication channels and, again, the congestion of messages in the network. The bandwidth often can be usefully modeled as the maximum rate at which processors can inject messages into the network without causing it to become heavily congested, assuming a uniform distribution of message destinations. In this case, the bandwidth can be expressed as the minimum *gap* g between successive injections of messages into the network.

Three models that characterize a network in terms of its latency and bandwidth are the Postal model [14], the Bulk-Synchronous Parallel (BSP) model [85], and the LogP model [29]. In the Postal model, a network

is described by a single parameter L, its latency. The Bulk-Synchronous Parallel model adds a second parameter g, the minimum ratio of computation steps to communication steps, i.e., the gap. The LogP model includes both of these parameters, and adds a third parameter o, the overhead, or wasted time, incurred by a processor upon sending or receiving a message.

Primitive Operations

A machine model must also specify the types of operations that the processors and network are permitted to perform. We assume that all processors are allowed to perform the same local instructions as the single processor in the standard sequential RAM model. In addition, processors may have special instructions for issuing nonlocal memory requests, for sending messages to other processors, and for executing various global operations, such as synchronization. There may also be restrictions on when processors can simultaneously issue instructions involving nonlocal operations. For example a model might not allow two processors to write to the same memory location at the same time. These restrictions might make it impossible to execute an algorithm on a particular model, or make the cost of executing the algorithm prohibitively expensive. It is therefore important to understand what instructions are supported before one can design or analyze a parallel algorithm. In this section we consider three classes of instructions that perform nonlocal operations: (1) instructions that perform concurrent accesses to the same shared memory location, (2) instructions for synchronization, and (3) instructions that perform global operations on data.

When multiple processors simultaneously make a request to read or write to the same resource—such as a processor, memory module, or memory location—there are several possible outcomes. Some machine models simply forbid such operations, declaring that it is an error if two or more processes try to access a resource simultaneously. In this case we say that the model allows only *exclusive* access to the resource. For example, a PRAM model might only allow exclusive read or write access to each memory location. A PRAM model of this type is called an exclusive-read exclusive-write (**EREW**) PRAM model. Other machine models may allow unlimited access to a shared resource. In this case we say that the model allows *concurrent* access to the resource. For example, a concurrent-read concurrent-write (**CRCW**) PRAM model allows both concurrent read and write access to memory locations, and a CREW PRAM model allows concurrent reads but only exclusive writes. When making a concurrent write to a resource such as a memory location there are many ways to resolve the conflict. The possibilities include choosing an arbitrary value from those written (arbitrary concurrent write), choosing the value from the processor with lowest index (priority concurrent write), and taking the *logical or* of the values written. A final choice is to allow for *queued* access. In this case concurrent access is permitted but the time for a step is proportional to the maximum number of accesses to any resource. A queue-read queue-write (QRQW) PRAM model allows for such accesses [36].

In addition to reads and writes to nonlocal memory or other processors, there are other important primitives that a model might supply. One class of such primitives support synchronization. There are a variety of different types of synchronization operations and the costs of these operations vary from model to model. In a PRAM model, for example, it is assumed that all processors operate in lock step, which provides implicit synchronization. In a local-memory machine model the cost of synchronization may be a function of the particular network topology. A related operation, broadcast, allows one processor to send a common message to all of the other processors. Some machine models supply more powerful primitives that combine arithmetic operations with communication. Such operations include the prefix and **multiprefix** operations, which are defined in Sections "Scans" and "Multiprefix and Fetch-and-Add."

Work-Depth Models

Because there are so many different ways to organize parallel computers, and hence to model them, it is difficult to select one multiprocessor model that is appropriate for all machines. The alternative to focusing

on the machine is to focus on the algorithm. In this section we present a class of models called work-depth models. In a *work-depth* model, the cost of an algorithm is determined by examining the total number of operations that it performs, and the dependencies among those operations. An algorithm's *work* W is the total number of operations that it performs; its *depth* D is the longest chain of dependencies among its operations. We call the ratio $\mathcal{P} = W/D$ the *parallelism* of the algorithm.

The **work-depth models** are more abstract than the multiprocessor models. As we shall see however, algorithms that are efficient in work-depth models can often be translated to algorithms that are efficient in the multiprocessor models, and from there to real parallel computers. The advantage of using a work-depth model is that there are no machine-dependent details to complicate the design and analysis of algorithms. Here we consider three classes of work-depth models: circuit models, vector machine models, and language-based models. We will be using a language-based model in this chapter, so we will return to these models in Section "Model Used in this Chapter." The most abstract work-depth model is the *circuit model.* A circuit consists of nodes and directed arcs. A node represents a basic operation, such as adding two values. Each input value for an operation arrives at the corresponding node via an incoming arc. The result of the operation is then carried out of the node via one or more outgoing arcs. These outgoing arcs may provide inputs to other nodes. The number of incoming arcs to a node is referred to as the *fan-in* of the node and the number of outgoing arcs is referred to as the *fan-out.* There are two special classes of arcs. A set of *input arcs* provide input values to the circuit as a whole. These arcs do not originate at nodes. The *output arcs* return the final output values produced by the circuit. These arcs do not terminate at nodes. By definition, a circuit is not permitted to contain a directed cycle. In this model, an algorithm is modeled as a family of directed acyclic circuits. There is a circuit for each possible size of the input.

Figure 47.4 shows a circuit for adding 16 numbers. In this figure all arcs are directed toward the bottom. The input arcs are at the top of the figure. Each + node adds the two values that arrive on its two incoming arcs, and places the result on its outgoing arc. The sum of all of the inputs to the circuit is returned on the single output arc at the bottom.

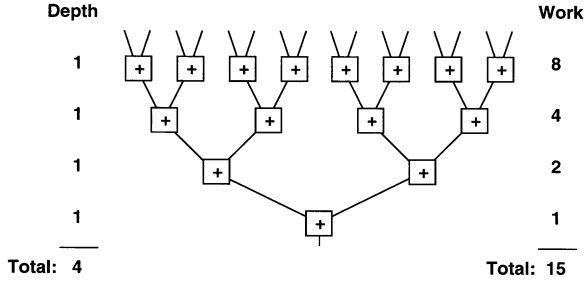


FIGURE 47.4 Summing 16 numbers on a tree. The total depth (longest chain of dependencies) is 4 and the total work (number of operations) is 15.

The **work** and **depth** of a circuit are measured as follows. The work is the total number of nodes. The work in Fig. 47.4, for example, is 15. (The work is also called the *size* of the circuit.) The depth is the number of nodes on the longest directed path from an input arc and an output arc. In Fig. 47.4, the depth is 4. For a family of circuits, the work and depth are typically parameterized in terms of the number of inputs. For example, the circuit in Fig. 47.4 can be easily generalized to add n input values for any n that is a power of two. The work and depth for this family of circuits is $W(n) = n - 1$ and $D(n) = \log_2 n$.

Circuit models have been used for many years to study various theoretical aspects of parallelism, for example to prove that certain problems are difficult to solve in parallel. See [48] for an overview.

In a *vector model* an algorithm is expressed as a sequence of steps, each of which performs an operation on a vector (i.e., sequence) of input values, and produces a vector result [19, 69]. The work of each step is

equal to the length of its input (or output) vector. The work of an algorithm is the sum of the work of its steps. The depth of an algorithm is the number of vector steps.

In a *language* model, a work-depth cost is associated with each programming language construct [20, 22]. For example, the work for calling two functions in parallel is equal to the sum of the work of the two calls. The depth, in this case, is equal to the maximum of the depth of the two calls.

Assigning Costs to Algorithms

In the work-depth models, the cost of an algorithm is determined by its work and by its depth. The notions of work and depth can also be defined for the multiprocessor models. The work W performed by an algorithm is equal to the number of processors multiplied by the time required for the algorithm to complete execution. The depth D is equal to the total time required to execute the algorithm.

The depth of an algorithm is important, because there are some applications for which the time to perform a computation is crucial. For example, the results of a weather forecasting program are useful only if the program completes execution before the weather does!

Generally, however, the most important measure of the cost of an algorithm is the work. This can be argued as follows. The cost of a computer is roughly proportional to the number of processors in the computer. The cost for purchasing time on a computer is proportional to the cost of the computer multiplied by the amount of time used. The total cost of performing a computation, therefore, is roughly proportional to the number of processors in the computer multiplied by the amount of time, i.e., the work.

In many instances, the cost of running a computation on a parallel computer may be slightly larger than the cost of running the same computation on a sequential computer. If the time to completion is sufficiently improved, however, this extra cost can often be justified. As we shall see, however, there is often a tradeoff between time-to-completion and total work performed. To quantify when parallel algorithms are efficient in terms of cost, we say that a parallel algorithm is **work-efficient** if asymptotically (as the problem size grows) it requires at most a constant factor more work than the best sequential algorithm known.

Emulations Among Models

Although it may appear that a different algorithm must be designed for each of the many parallel models, there are often automatic and efficient techniques for translating algorithms designed for one model into algorithms designed for another. These translations are **work-preserving** in the sense that the work performed by both algorithms is the same, to within a constant factor. For example, the following theorem, known as Brent's Theorem [24], shows that an algorithm designed for the circuit model can be translated in a work-preserving fashion to a PRAM model algorithm.

THEOREM 47.1 *[Brent's Theorem] Any algorithm that can be expressed as a circuit of size (i.e., work) W and depth D and with constant fan-in nodes in the circuit model can be executed in $O(W/P + D)$ steps in the CREW PRAM model.*

PROOF The basic idea is to have the PRAM emulate the computation specified by the circuit in a level-by-level fashion. The level of a node is defined as follows. A node is on level 1 if all of its inputs are also inputs to the circuit. Inductively, the level of any other node is one greater than the maximum of the level of the nodes that provide its inputs. Let l_i denote the number of nodes on level i. Then, by assigning $\lceil l_i/P \rceil$ operations to each of the P processors in the PRAM, the operations for level i can be performed in $O(\lceil l_i/P \rceil)$ steps. Concurrent reads might be required since many operations on one level might read

the same result from a previous level. Summing the time over all D levels, we have

$$\begin{aligned} T_{PRAM}(W, D, P) &= O\left(\sum_{i=1}^{D} \left\lceil \frac{l_i}{P} \right\rceil\right) \\ &= O\left(\sum_{i=1}^{D} \left(\frac{l_i}{P} + 1\right)\right) \\ &= O\left(\frac{1}{P}\left(\sum_{i=1}^{D} l_i\right) + D\right) \\ &= O\left(W/P + D\right) . \end{aligned}$$

The last step is derived by observing that $W = \sum_{i=1}^{D} l_i$, i.e., that the work is equal to the total number of nodes on all of the levels of the circuit.

The total work performed by the PRAM, i.e., the processor-time product, is $O(W + PD)$. This emulation is work-preserving to within a constant factor when the parallelism ($\mathcal{P} = W/D$) is at least as large as the number of processors P, for in this case the work is $O(W)$. The requirement that the parallelism exceed the number of processors is typical of work-preserving emulations.

Brent's Theorem shows that an algorithm designed for one of the work-depth models can be translated in a work-preserving fashion to a multiprocessor model. Another important class of work-preserving translations are those that translate between different multiprocessor models. The translation we consider here is the work-preserving translation of algorithms written for the PRAM model to algorithms for a modular memory machine model that incorporates the feature of network topology. In particular we consider a *butterfly machine* [55, Chapter 3.6] model in which P processors are attached through a butterfly network of depth $\log P$ to P memory banks. We assume that, in constant time, a processor can hash a virtual memory address to a physical memory bank and an address within that bank using a sufficiently powerful hash function. This scheme was first proposed by Karlin and Upfal [47] for the EREW PRAM model. Ranade [72] later presented a more general approach that allowed the butterfly to efficiently emulate CRCW algorithms.

THEOREM 47.2 *Any algorithm that takes time T on a P-processor PRAM model can be translated into an algorithm that takes time $O(T(P/P' + \log P'))$, with high probability, on a P'-processor butterfly machine model.*

Sketch of proof: Each of the P' processors in the butterfly emulates a set of P/P' PRAM processors. The butterfly emulates the PRAM in a step-by-step fashion. First, each butterfly processor emulates one step of each of its P/P' PRAM processors. Some of the PRAM processors may wish to perform memory accesses. For each memory access, the butterfly processor hashes the memory address to a physical memory bank and an address within the bank, and then routes a message through the network to that bank. These messages are pipelined so that a butterfly processor can have multiple outstanding requests. Ranade proved that if each processor in the P-processor butterfly sends at most P/P' messages, and if the destinations of the messages are determined by a sufficiently powerful random hash function, then the network can deliver all of the messages, along with responses, in $O(P/P' + \log P')$ time. The $\log P'$ term accounts for the latency of the network, and for the fact that there will be some congestion at memory banks, even if each processor sends only a single message.

This theorem implies that the emulation is work preserving when $P \geq P' \log P'$, i.e., when the number of processors employed by the PRAM algorithm exceeds the number of processors in the butterfly by a factor of at least $\log P'$. When translating algorithms from one multiprocessor model (e.g., the PRAM

model), which we call the *guest model,* to another multiprocessor model (e.g., the butterfly machine model), which we call the *host model,* it is not uncommon to require that the number of guest processors exceed the number of host processors by a factor proportional to the latency of the host. Indeed, the latency of the host can often be hidden by giving it a larger guest to emulate. If the bandwidth of the host is smaller than the bandwidth of a comparably sized guest, however, it is usually much more difficult for the host to perform a work-preserving emulation of the guest.

For more information on PRAM emulations, the reader is referred to [43, 86]

Model Used in This Chapter

Because there are so many work-preserving translations between different parallel models of computation, we have the luxury of choosing the model that we feel most clearly illustrates the basic ideas behind the algorithms, a work-depth language model. Here we define the model that we use in this chapter in terms of a set of language constructs and a set of rules for assigning costs to the constructs. The description here is somewhat informal, but should suffice for the purpose of this chapter. The language and costs can be properly formalized using a profiling semantics [22].

Most of the syntax that we use should be familiar to readers who have programmed in Algol-like languages, such as Pascal and C. The constructs for expressing parallelism, however, may be unfamiliar. We will be using two parallel constructs—a parallel *apply-to-each* construct and a *parallel-do* construct—and a small set of parallel primitives on sequences (one-dimensional arrays). Our language constructs, syntax and cost rules are loosely based on the NESL language [20].

The *apply-to-each* construct is used to apply an expression over a sequence of values in parallel. It uses a set like notation. For example, the expression

$$\{a * a : a \in [3, -4, -9, 5]\}$$

squares each element of the sequence $[3, -4, -9, 5]$ returning the sequence $[9, 16, 81, 25]$. This can be read: "in parallel, for each a in the sequence $[3, -4, -9, 5]$, square a." The apply-to-each construct also provides the ability to subselect elements of a sequence based on a filter. For example

$$\{a * a : a \in [3, -4, -9, 5] | a > 0\}$$

can be read: "in parallel, for each a in the sequence $[3, -4, -9, 5]$ such that a is greater than 0, square a." It returns the sequence $[9, 25]$. The elements that remain maintain the relative order from the original sequence.

The *parallel-do* construct is used to evaluate multiple statements in parallel. It is expressed by listing the set of statements after the keywords **in parallel do**. For example, the following fragment of code calls FUNCTION1 on (X) and assigns the result to A and in parallel calls FUNCTION2 on (Y) and assigns the result to B.

in parallel do
  $A :=$ FUNCTION1(X)
  $B :=$ FUNCTION2(Y)

The parallel-do completes when all of the parallel subcalls complete.

Work and depth are assigned to our language constructs as follows. The work and depth of a scalar primitive operation is one. For example, the work and depth for evaluating an expression such as $3 + 4$ is one. The work for applying a function to every element in a sequence is equal to the sum of the work for each of the individual applications of the function. For example, the work for evaluating the expression

$$\{a * a : a \in [0..n)\},$$

which creates an n-element sequence consisting of the squares of 0 through $n-1$, is n. The depth for applying a function to every element in a sequence is equal to the maximum of the depths of the individual applications of the function. Hence, the depth of the previous example is one. The work for a parallel-do construct is equal to the sum of the work for each of its statements. The depth is equal to the maximum depth of its statements. In all other cases, the work and depth for a sequence of operations are equal to the sums of the work and depth for the individual operations.

In addition to the parallelism supplied by apply-to-each, we use four built-in functions on sequences, *distribute,* ++ (append), *flatten,* and ← (write,) each of which can be implemented in parallel. The function *distribute* creates a sequence of identical elements. For example, the expression

$$distribute(3, 5)$$

creates the sequence

$$[3, 3, 3, 3, 3] .$$

The ++ function appends two sequences. For example $[2, 1]++[5, 0, 3]$ create the sequence $[2, 1, 5, 0, 3]$. The *flatten* function converts a nested sequence (a sequence in which each element is itself a sequence) into a flat sequence. For example,

$$flatten([[3, 5], [3, 2], [1, 5], [4, 6]])$$

creates the sequence

$$[3, 5, 3, 2, 1, 5, 4, 6] .$$

The ← function is used to write multiple elements into a sequence in parallel. It takes two arguments. The first argument is the sequence to modify and the second is a sequence of integer-value pairs that specify what to modify. For each pair (i, v) the value v is inserted into position i of the destination sequence. For example

$$[0, 0, 0, 0, 0, 0, 0, 0] \leftarrow [(4, -2), (2, 5), (5, 9)]$$

inserts the -2, 5 and 9 into the sequence at locations 4, 2 and 5, respectively, returning

$$[0, 0, 5, 0, -2, 9, 0, 0] .$$

As in the PRAM model, the issue of concurrent writes arises if an index is repeated. Rather than choosing a single policy for resolving concurrent writes, we will explain the policy used for the individual algorithms. All of these functions have depth one and work n, where n is the size of the sequence(s) involved. In the case of ←, the work is proportional to the length of the sequence of integer-value pairs, not the modified sequence, which might be much longer. In the case of ++, the work is proportional to the length of the second sequence.

We will use a few shorthand notations for specifying sequences. The expression $[-2..1]$ specifies the same sequence as the expression $[-2, -1, 0, 1]$. Changing the left or right bracket surrounding a sequence to a parenthesis omits the first or last elements, e.g., $[-2..1)$ denotes the sequence $[-2, -1, 0]$. The notation $A[i..j]$ denotes the subsequence consisting of elements $A[i]$ through $A[j]$. Similarly, $A[i, j)$ denotes the subsequence $A[i]$ through $A[j-1]$. We will assume that sequence indices are zero based, i.e., $A[0]$ extracts the first element of the sequence A.

Throughout this chapter our algorithms make use of random numbers. These numbers are generated using the functions *rand_bit(),* which returns a random bit, and *rand_int(h),* which returns a random integer in the range $[0, h-1]$.

47.3 Parallel Algorithmic Techniques

As in sequential algorithm design, in parallel algorithm design there are many general techniques that can be used across a variety of problem areas. Some of these are variants of standard sequential techniques, while others are new to parallel algorithms. In this section we introduce some of these techniques, including parallel divide-and-conquer, randomization, and parallel pointer manipulation. We will make use of these techniques in later sections.

Divide-and-Conquer

A divide-and-conquer algorithm first splits the problem to be solved into subproblems that are easier to solve than the original problem, and then solves the subproblems, often recursively. Typically the subproblems can be solved independently. Finally, the algorithm merges the solutions to the subproblems to construct a solution to the original problem.

The divide-and-conquer paradigm improves program modularity, and often leads to simple and efficient algorithms. It has therefore proven to be a powerful tool for sequential algorithm designers. Divide-and-conquer plays an even more prominent role in parallel algorithm design. Because the subproblems created in the first step are typically independent, they can be solved in parallel. Often the subproblems are solved recursively and thus the next divide step yields even more subproblems to be solved in parallel. As a consequence, even divide-and-conquer algorithms that were designed for sequential machines typically have some inherent parallelism. Note however, that in order for divide-and-conquer to yield a highly parallel algorithm, it is often necessary to parallelize the divide step and the merge step. It is also common in parallel algorithms to divide the original problem into as many subproblems as possible, so that they can all be solved in parallel.

As an example of parallel divide-and-conquer, consider the sequential mergesort algorithm. Mergesort takes a set of n keys as input and returns the keys in sorted order. It works by splitting the keys into two sets of $n/2$ keys, recursively sorting each set, and then merging the two sorted sequences of $n/2$ keys into a sorted sequence of n keys. To analyze the sequential running time of mergesort we note that two sorted sequences of $n/2$ keys can be merged in $O(n)$ time. Hence the running time can be specified by the recurrence

$$T(n) = \begin{cases} 2T(n/2) + O(n) & n > 1 \\ O(1) & n = 1 \end{cases} \tag{47.1}$$

which has the solution $T(n) = O(n \log n)$. Although not designed as a parallel algorithm, mergesort has some inherent parallelism since the two recursive calls are independent, thus allowing them to be made in parallel. The parallel calls can be expressed as

ALGORITHM: MERGESORT(A)
1 **if** ($|A| = 1$) **then return** A
2 **else**
3 **in parallel do**
4 $L :=$ MERGESORT($A[0..|A|/2)$)
5 $R :=$ MERGESORT($A[|A|/2..|A|)$)
6 **return** MERGE(L, R)

Recall that in our work-depth model we can analyze the depth of an algorithm that makes parallel calls by taking the maximum depth of the two calls, and the work by taking the sum of the work of the two calls. We assume that the merging remains sequential so that the work and depth to merge two sorted sequences of $n/2$ keys is $O(n)$. Thus for mergesort the work and depth are given by the recurrences

$$W(n) = 2W(n/2) + O(n) \tag{47.2}$$

$$D(n) = \max(D(n/2), D(n/2)) + O(n) \quad (47.3)$$
$$= D(n/2) + O(n) \quad (47.4)$$

As expected, the solution for the work is $W(n) = O(n \log n)$, i.e., the same as the time for the sequential algorithm. For the depth, however, the solution is $D(n) = O(n)$, which is smaller than the work. Recall that we defined the parallelism of an algorithm as the ratio of the work to the depth. Hence, the parallelism of this algorithm is $O(\log n)$ (not very much). The problem here is that the merge step remains sequential, and is the bottleneck.

As mentioned earlier, the parallelism in a divide-and-conquer algorithm can often be enhanced by parallelizing the divide step and/or the merge step. Using a parallel merge [52], two sorted sequences of $n/2$ keys can be merged with work $O(n)$ and depth $O(\log \log n)$. Using this merge algorithm, the recurrence for the depth of mergesort becomes

$$D(n) = D(n/2) + O(\log \log n) \quad (47.5)$$

which has solution $D(n) = O(\log n \log \log n)$. Using a technique called **pipelined divide-and-conquer** the depth of mergesort can be further reduced to $O(\log n)$ [26]. The idea is to start the merge at the top level before the recursive calls complete.

Divide-and-conquer has proven to be one of the most powerful techniques for solving problems in parallel. In this chapter, we will use it to solve problems from computational geometry, for sorting, and for performing fast Fourier transforms. Other applications range from solving linear systems, to factoring large numbers, to performing n-body simulations.

Randomization

Random numbers are used in parallel algorithms to ensure that processors can make local decisions that, with high probability, add up to good global decisions. Here we consider three uses of randomness.

Sampling One use of randomness is to select a representative sample from a set of elements. Often, a problem can be solved by selecting a sample, solving the problem on that sample, and then using the solution for the sample to guide the solution for the original set. For example, suppose we want to sort a collection of integer keys. This can be accomplished by partitioning the keys into buckets and then sorting within each bucket. For this to work well, the buckets must represent nonoverlapping intervals of integer values, and each bucket must contain approximately the same number of keys. **Random sampling** is used to determine the boundaries of the intervals. First each processor selects a random sample of its keys. Next all of the selected keys are sorted together. Finally these keys are used as the boundaries. Such random sampling is also used in many parallel computational geometry, graph, and string matching algorithms.

Symmetry Breaking Another use of randomness is in **symmetry breaking.** For example, consider the problem of selecting a large independent set of vertices in a graph in parallel. (A set of vertices is *independent* if no two are neighbors.) Imagine that each vertex must decide, in parallel with all other vertices, whether to join the set or not. Hence, if one vertex chooses to join the set, then all of its neighbors must choose not to join the set. The choice is difficult to make simultaneously for each vertex if the local structure at each vertex is the same, for example if each vertex has the same number of neighbors. As it turns out, the impasse can be resolved by using randomness to break the symmetry between the vertices [58].

Load Balancing A third use of randomness is load balancing. One way to quickly partition a large number of data items into a collection of approximately evenly sized subsets is to randomly assign each element to a subset. This technique works best when the average size of a subset is at least logarithmic in the size of the original set.

Parallel Pointer Techniques

Many of the traditional sequential techniques for manipulating lists, trees, and graphs do not translate easily into parallel techniques. For example, techniques such as traversing the elements of a linked list, visiting the nodes of a tree in postorder, or performing a depth-first traversal of a graph appear to be inherently sequential. Fortunately these techniques can often be replaced by parallel techniques with roughly the same power.

Pointer Jumping One of the oldest parallel pointer techniques is **pointer jumping** [88]. This technique can be applied to either lists or trees. In each pointer jumping step, each node in parallel replaces its pointer with that of its successor (or parent). For example, one way to label each node of an n-node list (or tree) with the label of the last node (or root) is to use pointer jumping. After at most $\lceil \log n \rceil$ steps, every node points to the same node, the end of the list (or root of the tree). This is described in more detail in Section "Pointer Jumping."

Euler Tour Technique An Euler tour of a directed graph is a path through the graph in which every edge is traversed exactly once. In an undirected graph each edge is typically replaced by two oppositely directed edges. The Euler tour of an undirected tree follows the perimeter of the tree visiting each edge twice, once on the way down and once on the way up. By keeping a linked structure that represents the Euler tour of a tree it is possible to compute many functions on the tree, such as the size of each subtree [83]. This technique uses linear work, and parallel depth that is independent of the depth of the tree. The Euler tour technique can often be used to replace a standard traversal of a tree, such as a depth-first traversal.

Graph Contraction **Graph contraction** is an operation in which a graph is reduced in size while maintaining some of its original structure. Typically, after performing a graph contraction operation, the problem is solved recursively on the contracted graph. The solution to the problem on the contracted graph is then used to form the final solution. For example, one way to partition a graph into its connected components is to first contract the graph by merging some of the vertices with neighboring vertices, then find the connected components of the contracted graph, and finally undo the contraction operation. Many problems can be solved by contracting trees [64, 65], in which case the technique is called **tree contraction.** More examples of graph contraction can be found in Section 47.5.

Ear Decomposition An ear decomposition of a graph is a partition of its edges into an ordered collection of paths. The first path is a cycle, and the others are called ears. The end-points of each ear are anchored on previous paths. Once an ear decomposition of a graph is found, it is not difficult to determine if two edges lie on a common cycle. This information can be used in algorithms for determining biconnectivity, triconnectivity, 4-connectivity, and planarity [60, 63]. An ear decomposition can be found in parallel using linear work and logarithmic depth, independent of the structure of the graph. Hence, this technique can be used to replace the standard sequential technique for solving these problems, depth-first search.

Other Techniques

Many other techniques have proven to be useful in the design of parallel algorithms. Finding small graph separators is useful for partitioning data among processors to reduce communication [75, Chapter 14]. Hashing is useful for load balancing and mapping addresses to memory [47, 87]. Iterative techniques are useful as a replacement for direct methods for solving linear systems [18].

47.4 Basic Operations on Sequences, Lists, and Trees

We begin our presentation of parallel algorithms with a collection of algorithms for performing basic operations on sequences, lists, and trees. These operations will be used as subroutines in the algorithms that follow in later sections.

Sums

As explained near the beginning of this chapter, there is a simple recursive algorithm for computing the sum of the elements in an array.

ALGORITHM: SUM(A)
1 **if** $|A| = 1$ **then return** $A[0]$
2 **else return** SUM($\{A[2i] + A[2i+1] : i \in [0..|A|/2)\}$)

The work and depth for this algorithm are given by the recurrences

$$W(n) = W(n/2) + O(n) \tag{47.6}$$
$$D(n) = D(n/2) + O(1) \tag{47.7}$$

which have solutions $W(n) = O(n)$ and $D(n) = O(\log n)$. This algorithm can also be expressed without recursion (using a **while** loop), but the recursive version foreshadows the recursive algorithm for the **scan** function.

As written, the algorithm only works on sequences that have lengths equal to powers of 2. Removing this restriction is not difficult by checking if the sequence is of odd length and separately adding the last element in if it is. This algorithm can also easily be modified to compute the "sum" using any other binary associative operator in place of $+$. For example the use of max would return the maximum value in the sequence.

Scans

The *plus-scan* operation (also called all-prefix-sums) takes a sequence of values and returns a sequence of equal length for which each element is the sum of all previous elements in the original sequence. For example, executing a plus-scan on the sequence [3, 5, 3, 1, 6] returns [0, 3, 8, 11, 12]. An algorithm for performing the scan operation [81] is shown below.

ALGORITHM: SCAN(A)
1 **if** $|A| = 1$ **then return** $[0]$
2 **else**
3 $S = \text{SCAN}(\{A[2i] + A[2i+1] : i \in [0..|A|/2)\})$
4 $R = \{\textbf{if } (i \bmod 2) = 0 \textbf{ then } S[i/2] \textbf{ else } S[(i-1)/2] + A[i-1] : i \in [0..|A|)\}$
5 **return** R

The algorithm works by element-wise adding the even indexed elements of A to the odd indexed elements of A, and then recursively solving the problem on the resulting sequence (Line 3). The result S of the recursive call gives the plus-scan values for the even positions in the output sequence R. The value for each of the odd positions in R is simply the value for the preceding even position in R plus the value of the preceding position from A.

The asymptotic work and depth costs of this algorithm are the same as for the SUM operation, $W(n) = O(n)$ and $D(n) = O(\log n)$. Also, as with the SUM operation, any binary associative operator can be used in place of the $+$. In fact the algorithm described can be used more generally to solve various recurrences, such as the first-order linear recurrences $x_i = (x_{i-1} \otimes a_i) \oplus b_i$, $0 \leq i \leq n$, where $\otimes$ and $\oplus$ are both binary associative operators [51].

Scans have proven so useful in the design of parallel algorithms that some parallel machines provide support for scan operations in hardware.

Multiprefix and Fetch-and-Add

The *multiprefix* operation is a generalization of the scan operation in which multiple independent scans are performed. The input to the multiprefix operation is a sequence A of n pairs (k, a), where k specifies a key and a specifies an integer data value. For each key value, the multiprefix operation performs an independent scan. The output is a sequence B of n integers containing the results of each of the scans such that if $A[i] = (k, a)$ then

$$B[i] = sum(\{b : (t, b) \in A[0..i) \mid t = k\})$$

In other words, each position receives the sum of all previous elements that have the same key. As an example,

$$\text{MULTIPREFIX}([(1, 5), (0, 2), (0, 3), (1, 4), (0, 1), (2, 2)])$$

returns the sequence

$$[0, 0, 2, 5, 5, 0]$$

The *fetch-and-add* operation is a weaker version of the multiprefix operation, in which the order of the input elements for each scan is not necessarily the same as the order in the input sequence A. In this chapter we do not present an algorithm for the multiprefix operation, but it can be solved by a function that requires work $O(n)$ and depth $O(\log n)$ using concurrent writes [61].

Pointer Jumping

Pointer jumping is a technique that can be applied to both linked lists and trees [88]. The basic pointer jumping operation is simple. Each node i replaces its pointer $P[i]$ with the pointer of the node that it points to, $P[P[i]]$. By repeating this operation, it is possible to compute, for each node in a list or tree, a pointer to the end of the list or root of the tree. Given a set P of pointers that represent a tree (i.e., pointers from children to parents), the following code will generate a pointer from each node to the root of the tree. We assume that the root points to itself.

ALGORITHM: POINT_TO_ROOT(P)
1 **for** j **from** 1 **to** $\lceil \log |P| \rceil$
2 $\quad P := \{P[P[i]] : i \in [0..|P|)\}$

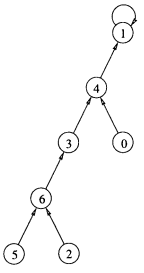
The idea behind this algorithm is that in each loop iteration the distance spanned by each pointer, with respect to the original tree, will double, until it points to the root. Since a tree constructed from $n = |P|$ pointers has depth at most $n - 1$, after $\lceil \log n \rceil$ iterations each pointer will point to the root. Because each iteration has constant depth and performs $\Theta(n)$ work, the algorithm has depth $\Theta(\log n)$ and work $\Theta(n \log n)$.

Figure 47.5 illustrates algorithm POINT_TO_ROOT applied to a tree consisting of seven nodes.

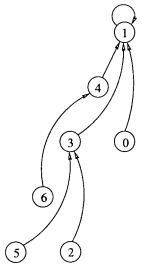
List Ranking

The problem of computing the distance from each node to the end of a linked list is called *list ranking*. Function POINT_TO_ROOT can be easily modified to compute these distances, as shown below.

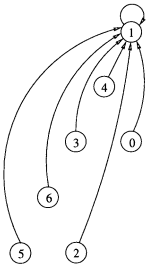
ALGORITHM: LIST_RANK(P)
1 $V = \{$**if** $P[i] = i$ **then** 0 **else** $1 : i \in [0..|P|)\}$
2 **for** j **from** 1 **to** $\lceil \log |P| \rceil$
3 $\quad V := \{V[i] + V[P[i]] : i \in [0..|P|)\}$
4 $\quad P := \{P[P[i]] : i \in [0..|P|)\}$
5 **return** V



(a) The input tree $P = [4, 1, 6, 4, 1, 6, 3]$.



(b) The tree $P = [1, 1, 3, 1, 1, 3, 4]$ after one iteration of the algorithm.



(c) The final tree $P = [1, 1, 1, 1, 1, 1, 1]$.

FIGURE 47.5 The effect of two iterations of algorithm POINT_TO_ROOT.

In this function, $V[i]$ can be thought of as the distance spanned by pointer $P[i]$ with respect to the original list. Line 1 initializes V by setting $V[i]$ to 0 if i is the last node (i.e., points to itself), and 1 otherwise. In each iteration, Line 3 calculates the new length of $P[i]$. The function has depth $\Theta(\log n)$ and work $\Theta(n \log n)$.

It is worth noting that there are simple sequential algorithms that perform the same tasks as both functions POINT_TO_ROOT and LIST_RANK using only $O(n)$ work. For example, the list ranking problem can be solved by making two passes through the list. The goal of the first pass is simply to count the number of elements in the list. The elements can then be numbered with their positions from the end of the list in a second pass. Thus, neither function POINT_TO_ROOT nor LIST_RANK are work-efficient, since both require $\Theta(n \log n)$ work in the worst case. There are, however, several work-efficient parallel solutions to both of these problems.

The following parallel algorithm uses the technique of random sampling to construct a pointer from each node to the end of a list of n nodes in a work-efficient fashion [74]. The algorithm is easily generalized to solve the list-ranking problem.

1. Pick m list nodes at random and call them the *start* nodes.
2. From each start node u, follow the list until reaching the next start node v. Call the list nodes between u and v the *sublist* of u.

3. Form a shorter list consisting only of the start nodes and the final node on the list by making each start node point to the next start node on the list.
4. Using pointer jumping on the shorter list, for each start node create a pointer to the last node in the list.
5. For each start node u, distribute the pointer to the end of the list to all of the nodes in the sublist of u.

The key to analyzing the work and depth of this algorithm is to bound the length of the longest sublist. Using elementary probability theory, it is not difficult to prove that the expected length of the longest sublist is at most $O((n \log m)/m)$. The work and depth for each step of the algorithm are thus computed as follows.

1. $W(n, m) = O(m)$ and $D(n, m) = O(1)$
2. $W(n, m) = O(n)$ and $D(n, m) = O((n \log m)/m)$
3. $W(n, m) = O(m)$ and $D(n, m) = O(1)$
4. $W(n, m) = O(m \log m)$ and $D(n, m) = O(\log m)$
5. $W(n, m) = O(n)$ and $D(n, m) = O((n \log m)/m)$

Thus, the work for the entire algorithm is $W(m, n) = O(n + m \log m)$, and the depth is $O((n \log m)/m)$. If we set $m = n/\log n$, these reduce to $W(n) = O(n)$ and $D(n) = O(\log^2 n)$.

Using a technique called *contraction,* it is possible to design a list ranking algorithm that runs in $O(n)$ work and $O(\log n)$ depth [8, 9]. This technique can also be applied to trees [64, 65].

Removing Duplicates

This section presents several algorithms for removing the duplicate items that appear in a sequence. Thus, the input to each algorithm is a sequence, and the output is a new sequence containing exactly one copy of every item that appears in the input sequence. It is assumed that the order of the items in the output sequence does not matter. Such an algorithm is useful when a sequence is used to represent an unordered set of items. Two sets can be merged, for example, by first appending their corresponding sequences, and then removing the duplicate items.

Approach 1: Using an Array of Flags

If the items are all nonnegative integers drawn from a small range, we can use a technique similar to bucket sort to remove the duplicates. We begin by creating an array equal in size to the range, and initializing all of its elements to 0. Next, using concurrent writes we set a flag in the array for each number that appears in the input list. Finally, we extract those numbers with flags that have been set. This algorithm is expressed as follows.

ALGORITHM: REM_DUPLICATES(V)
1 RANGE := 1+ MAX(V)
2 FLAGS := *distribute* (0,RANGE) $\leftarrow$ $\{(i, 1) : i \in V\}$
3 **return** $\{j : j \in [0..\text{RANGE})|$ FLAGS$[j] = 1\}$

This algorithm has depth $O(1)$ and performs work $O(|V| + \text{MAX}(V))$. Its obvious disadvantage is that it explodes when given a large range of numbers, both in memory and in work.

Approach 2: Hashing

A more general approach is to use a hash table. The algorithm has the following outline. The algorithm first creates a hash table that contains a prime number of entries, where the prime is approximately

twice as large as the number of items in the set V. A prime size is best, because it makes designing a good hash function easier. The size must also be large enough that the chance of collisions in the hash table is not too great. Let m denote the size of the hash table. Next, the algorithm computes a hash value $hash(V[j], m)$ for each item $V[j] \in V$, and attempts to write the index j into the hash table entry $hash(V[j], m)$. For example, Fig. 47.6 describes a particular hash function applied to the sequence [69, 23, 91, 18, 42, 23, 18]. We assume that if multiple values are simultaneously written into the same memory location, one of the values will be correctly written (the arbitrary concurrent write model). An item $V[j]$ is called a *winner* if the index j is successfully written into the hash table. In our example, the winners are $V[0]$, $V[1]$, $V[2]$, and $V[3]$, i.e., 69, 23, 91, and 18. The winners are added to the duplicate-free sequence that is being constructed, and then set aside. Among the losers, we must distinguish between two types of items, those that were defeated by an item with the same value, and those that were defeated by an item with a different value. In our example, $V[5]$ and $V[6]$ (23 and 18) were defeated by items with the same value, and $V[4]$ (42) was defeated by an item with a different value. Items of the first type are set aside because they are duplicates. Items of the second type are retained, and the algorithm repeats the entire process on them using a different hash function. In general, it may take several iterations before all of the items have been set aside, and in each iteration the algorithm must use a different hash function.

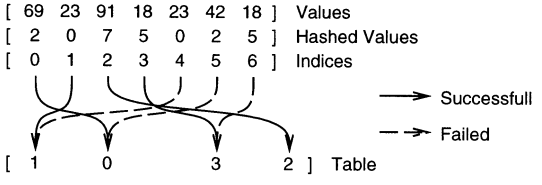


FIGURE 47.6 Each key attempts to write its index into a hash table entry.

The code for removing duplicates using hashing is shown below.

ALGORITHM: REMOVE_DUPLICATES(V)

```
1  m := NEXT_PRIME(2 * |V|)
2  TABLE := distribute(−1, m)
3  i := 0
4  RESULT := {}
5  while |V| > 0
6     TABLE := TABLE ← {(hash(V[j], m, i), j) : j ∈ [0..|V|)}
7     WINNERS := {V[j] : j ∈ [0..|V|) | TABLE[hash(V[j], m, i)] = j}
8     RESULT := RESULT ++ WINNERS
9     TABLE := TABLE ← {(hash(k, m, i), k) : k ∈ WINNERS}
10    V := {k ∈ V | TABLE[hash(k, m, i)] ≠ k}
11    i := i + 1
12 return RESULT
```

The first four lines of function REMOVE_DUPLICATES initialize several variables. Line 1 finds the first prime number larger than $2 * |V|$ using the built-in function NEXT_PRIME. Line 2 creates the hash table, and initializes its entries with an arbitrary value (-1). Line 3 initializes i, a variable that simply counts iterations of the **while** loop. Line 4 initializes the sequence RESULT to be empty. Ultimately, RESULT will contain a single copy of each distinct item from the sequence V.

The bulk of the work in function REMOVE_DUPLICATES is performed by the **while** loop. While there are items remaining to be processed, the code performs the following steps. In Line 6, each item $V[j]$

attempts to write its index j into the table entry given by the hash function $hash(V[j], m, i)$. Note that the hash function takes the iteration i as an argument, so that a different hash function is used in each iteration. Concurrent writes are used so that if several items attempt to write to the same entry, precisely one will win. Line 7 determines which items successfully wrote indices in Line 6, and stores the values of these items in an array called WINNERS. The winners are added to the RESULT in Line 8. The purpose of Lines 9 and 10 is to remove all of the items that are either winners or duplicates of winners. These lines reuse the hash table. In Line 9, each winner writes its value, rather than its index, into the hash table. In this step there are no concurrent writes. Finally, in Line 10, an item is retained only if it is not a winner, and the item that defeated it has a different value.

It is not difficult to prove that, provided that the hash values $hash(V[j], m, i)$ are random and sufficiently independent, both between iterations and within an iteration, each iteration reduces the number of items remaining by some constant fraction until the number of items remaining is small. As a consequence, $D(n) = O(\log n)$ and $W(n) = O(n)$.

The remove-duplicates algorithm is frequently used for set operations. For instance, given the code for REMOVE_DUPLICATES, it is easy to write the code for the set union operation.

47.5 Graphs

Graph problems are often difficult to parallelize since many standard sequential graph techniques, such as depth-first or priority-first search, do not parallelize well. For some problems, such as minimum-spanning tree and biconnected components, new techniques have been developed to generate efficient parallel algorithms. For other problems, such as single-source shortest paths, there are no known efficient parallel algorithms, at least not for the general case.

We have already outlined some of the parallel graph techniques in Section 47.3. In this section we describe algorithms for breadth-first search, for finding connected components, and for finding minimum spanning trees. These algorithms use some of the general techniques. In particular, randomization and graph contraction will play an important role in the algorithms. In this chapter we limit ourselves to algorithms on sparse undirected graphs. We suggest the following sources for further information on parallel graph algorithms [75, Chapters 2-8], [45, Chapter 5], [35, Chapter 2].

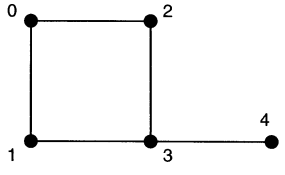
Graphs and Graph Representations

A *graph* $G = (V, E)$ consists of a set of *vertices* V and a set of *edges* E in which each edge connects two vertices. In a *directed graph* each edge is directed from one vertex to another, while in an *undirected graph* each edge is symmetric, i.e., goes in both directions. A *weighted graph* is a graph in which each edge $e \in E$ has a weight $w(e)$ associated with it. In this chapter we use the convention that $n = |V|$ and $m = |E|$. Qualitatively, a graph is considered sparse if m is much less than n^2 and dense otherwise. The *diameter* of a graph, denoted $D(G)$, is the maximum, over all pairs of vertices (u, v), of the minimum number of edges that must be traversed to get from u to v.

There are three standard representations of graphs used in sequential algorithms: edge lists, adjacency lists, and adjacency matrices. An *edge list* consists of a list of edges, each of which is a pair of vertices. The list directly represents the set E. An *adjacency list* is an array of lists. Each array element corresponds to one vertex and contains a linked list of pointers to the neighboring vertices, i.e., the linked list for a vertex v contains pointers to the vertices $\{u|(v, u) \in E\}$). An *adjacency matrix* is an $n \times n$ array A such that A_{ij} is 1 if $(i, j) \in E$ and 0 otherwise. The adjacency matrix representation is typically used only when the graph is dense since it requires $\Theta(n^2)$ space, as opposed to $\Theta(n + m)$ space for the other two representations. Each of these representations can be used to represent either directed or undirected graphs.

For parallel algorithms we use similar representations for graphs. The main change we make is to replace the linked lists with arrays. In particular the edge-list is represented as an array of edges and the

adjacency-list is represented as an array of arrays. Using arrays instead of lists makes it easier to process the graph in parallel. In particular, they make it easy to grab a set of elements in parallel, rather than having to follow a list. Figure 47.7 shows an example of our representations for an undirected graph. Note that for the edge-list representation of the undirected graph each edge appears twice, once in each direction (this property is important for some of the algorithms described in this chapter[1]). To represent a directed graph we simply only store the edge once in the desired direction. In the text we will refer to the left element of an edge pair as the *source vertex* and the right element as the *destination vertex*.



(a)

[(0,1), (0,2), (2,3), (3,4), (1,3), (1,0), (2,0), (3,2), (4,3), (3,1)]

(b)

[[1, 2], [0, 3], [0, 3], [1, 2, 4], [3]]

(c)

FIGURE 47.7 Representations of an undirected graph. (a) A graph G with 5 vertices and 5 edges. (b) An edge-list representation of G. (c) The adjacency-list representation of G. Values between square brackets are elements of an array, and values between parentheses are elements of a pair.

In designing algorithms, it is sometimes more efficient to use an edge list and sometimes more efficient to use an adjacency list. It is therefore important to be able to convert between the two representations. To convert from an adjacency list to an edge list (representation (c) to representation (b) in Fig. 47.7) is straightforward. The following code will do it with linear work and constant depth:

$$flatten(\{\{(i, j) : j \in G[i]\} : i \in [0..|G|\})$$

where G is the graph in the adjacency list representation. For each vertex i this code pairs up i with each of i's neighbors. The *flatten* is used since the nested apply-to-each will return a sequence of sequences that needs to be flattened into a single sequence.

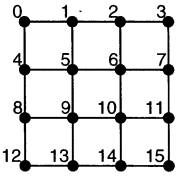
To convert from an edge list to an adjacency list is somewhat more involved, but still requires only linear work. The basic idea is to sort the edges based on the source vertex. This places edges from a particular vertex in consecutive positions in the resulting array. This array can then be partitioned into blocks based on the source vertices. It turns out that since the sorting is on integers in the range $[0..|V|)$, a radix sort can be used (see Section "Radix Sort"), which requires linear work. The depth of the radix sort depends on the depth of the multiprefix operation (see Section "Multiprefix and Fetch-and-Add").

Breadth First Search

The first algorithm we consider is parallel breadth-first search (BFS). BFS can be used to solve problems such as determining if a graph is connected or generating a spanning tree of a graph. Parallel BFS is similar to the sequential version, which starts with a source vertex s and visits levels of the graph one after the other using a queue to keep track of vertices that have not yet been visited. The main difference is that each level is going to be visited in parallel and no queue is required. As with the sequential algorithm each vertex will only be visited once and each edge at most twice, once in each direction. The work is therefore

[1] If space is of serious concern, the algorithms can be easily modified to work with edges stored in just one direction.

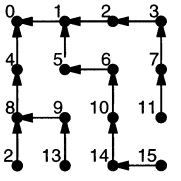
linear in the size of the graph, $O(n+m)$. For a graph with diameter D, the number of levels visited by the algorithm will be at least $D/2$ and at most D, depending on where the search is initiated. We will show that each level can be visited in constant depth, assuming a concurrent-write model, so that the total depth of parallel BFS is $O(D)$.



(a)

Step	Frontier
0	[0]
1	[1, 4]
2	[2, 5, 8]
3	[3, 6, 9, 12]
5	[7, 10, 13]
6	[11, 14]
7	[15]

(b)



(c)

FIGURE 47.8 Example of parallel breadth-first search. (a) A graph G. (b) The frontier at each step of the BFS of G with $s = 0$. (c) A BFS tree.

The main idea of parallel BFS is to maintain a set of frontier vertices that represent the current level being visited, and to produce a new frontier on each step. The set of frontier vertices is initialized with the singleton s (the source vertex). A new frontier is generated by collecting all of the neighbors of the current frontier vertices in parallel and removing any that have already been visited. This is not sufficient on its own, however, since multiple vertices might collect the same unvisited vertex. For example, consider the graph in Fig. 47.8. On step 2 vertices 5 and 8 will both collect vertex 9. The vertex will therefore appear twice in the new frontier. If the duplicate vertices are not removed the algorithm can generate an exponential number of vertices in the frontier. This problem does not occur in the sequential BFS because vertices are visited one at a time. The parallel version therefore requires an extra step to remove duplicates.

The following function performs a parallel BFS. It takes as input a source vertex s and a graph G represented as an adjacency-array, and returns as its result a breadth-first-search tree of G. In a BFS tree each vertex visited at level i points to one of its neighbors visited at level $i-1$ (see Fig. 47.8(c)). The source s is the root of the tree.

ALGORITHM: BFS(s, G)

```
1  FRONT := [s]
2  TREE := distribute(−1, |G|)
3  TREE[s] := s
4  while (|FRONT| ≠ 0)
5     E := flatten({{(u, v) : u ∈ G[v]} : v ∈ FRONT})
6     E' := {(u, v) ∈ E | TREE[u] = −1}
7     TREE := TREE ← E'
8     FRONT := {u : (u, v) ∈ E' | v = TREE[u]}
9  return TREE
```

In this code FRONT is the set of frontier vertices, and TREE is the current BFS tree, represented as an array of indices (pointers). The pointers (indices) in TREE are all initialized to -1, except for the source s which is initialized to point to itself. Each vertex in TREE is set to point to its parent in the BFS tree when it is visited. The algorithm assumes the arbitrary concurrent write model.

We now consider each iteration of the algorithm. The iterations terminate when there are no more vertices in the frontier (Line 4). The new frontier is generated by first collecting into an edge-array the set of edges from current frontier vertices to the neighbors of these vertices (Line 5). An edge from v to u is kept as the pair (u, v) (this is backwards from the standard edge representation and is used below to write from v to u). Next, the algorithm subselects the edges that lead to unvisited vertices (Line 6). Now for each remaining edge (u, v) the algorithm writes the source index v into the BFS tree entry for the destination vertex u (Line 7). In the case that more than one edge has the same destination, one of the source indices will be written arbitrarily—this is the only place that the algorithm uses a concurrent write. These indices become the parent pointers for the BFS tree, and are also used to remove duplicates for the next frontier set. In particular, the algorithm checks whether each edge succeeded in writing its source by reading back from the destination. If an edge reads the value it wrote, its destination is included in the new frontier (Line 8). Since only one edge that points to a given destination vertex will read back the same value, no duplicates will appear in the new frontier.

The algorithm requires only constant depth per iteration of the while loop. Since each vertex and its associated edges are visited only once, the total work is $O(m + n)$. An interesting aspect of this parallel BFS is that it can generate BFS trees that cannot be generated by a sequential BFS, even allowing for any order of visiting neighbors in sequential BFS. We leave the generation of an example as an exercise. We note, however, that if the algorithm used a priority concurrent write (see Section "Model Used in this Chapter") on Line 7, then it would generate the same tree as a sequential BFS.

Connected Components

We now consider the problem of labeling the connected components of an undirected graph. The problem is to label all the vertices in a graph G such that two vertices u and v have the same label if and only if there is a path between the two vertices. Sequentially the connected components of a graph can easily be labeled using either depth-first or breadth-first search. We have seen how to perform a breadth-first search, but the technique requires depth proportional to the diameter of a graph. This is fine for graphs with small diameter, but does not work well in the general case. Unfortunately, in terms of work, even the most efficient polylogarithmic-depth parallel algorithms for depth-first search and breadth-first search are very inefficient. Hence, the efficient algorithms for solving the connected components problem use different techniques.

The two algorithms we consider are based on *graph contraction*. Graph contraction works by contracting the vertices of a connected subgraph into a single vertex. The techniques we use allow the algorithms to make many such contractions in parallel across the graph. The algorithms therefore proceed in a sequence of steps, each of which contracts a set of subgraphs, and forms a smaller graph in which each subgraph has been converted into a vertex. If each such step of the algorithm contracts the size of the graph by a constant fraction, then each component will contract down to a single vertex in $O(\log n)$ steps. By running the contraction in reverse, the algorithms can label all the vertices in the components. The two algorithms we consider differ in how they select subgraphs for contraction. The first uses randomization and the second is deterministic. Neither algorithm is work efficient because they require $O((n + m) \log n)$ work for worst-case graphs, but we briefly discuss how they can be made work efficient in Section "Improved Versions of Connected Components." Both algorithms require the concurrent write model.

Random Mate Graph Contraction

The random mate technique for graph contraction is based on forming a set of star subgraphs and contracting the stars. A *star* is a tree of depth one—it consists of a root and an arbitrary number of children. The random mate algorithm finds a set of nonoverlapping stars in a graph, and then contracts each star into a single vertex by merging each child into its parent. The technique used to form the stars uses randomization. For each vertex the algorithm flips a coin to decide if that vertex is a parent or a

child. We assume the coin is unbiased so that every vertex has a 50% probability of being a parent. Now for each child vertex the algorithm selects a neighboring parent vertex and makes that parent the child's root. If the child has no neighboring parent, it has no root. The parents are now the roots of a set of stars, each with zero or more children. The children either belong to one of these stars or are left on their own (if they had no neighboring parents). The algorithm now contracts these stars. When contracting, the algorithm updates any edge that points to a child of a star to point to the root. Figure 47.9 illustrates a full contraction step. This contraction step is repeated until all components are of size 1.

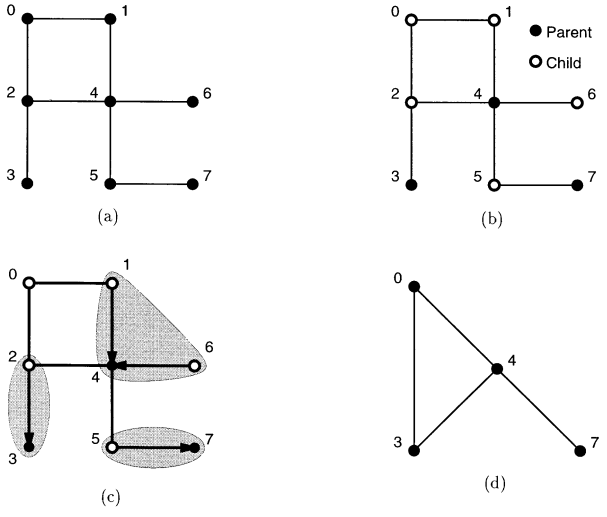


FIGURE 47.9 Example of one step of random mate graph contraction. (a) The original graph G. (b) G after selecting the parents randomly. (c) The stars formed after each child is assigned a neighboring parent as its root (each star is shaded). (d) The graph after contracting each star and relabeling the edges to point to the roots. Children with no neighboring parent remain in the graph.

To analyze the number of contraction steps required to complete we need to know how many vertices the algorithm removes on each contraction step. First we note that a contraction step is only going to remove children, and only if they have a neighboring parent. The probability that a vertex will be deleted is therefore the probability that a vertex is a child multiplied by the probability that at least one of its neighbors is a parent. The probability that it is a child is 1/2 and the probability that at least one neighbor is a parent is at least 1/2 (every vertex that is not fully contracted has one or more neighbors). The algorithm is therefore expected to remove at least 1/4 of the remaining vertices at each step, and since this is a constant fraction, it is expected to complete in $O(\log n)$ steps. The full probabilistic analysis is somewhat involved since it is possible to have a streak of bad flips, but it is not too hard to show that the algorithm is very unlikely to require more than $O(\log n)$ contraction steps.

The following algorithm uses random mate contraction for labeling the connected components of a graph. The algorithm works by contracting until each component is a single vertex and then re-expanding so that it can label all vertices in that component with the same label. The input to the algorithm is a graph G in the edge-list representation (note that this is a different representation than used in BFS), along with the labels of the vertices. The labels of the vertices are initialized to be unique indices in the range $0..|V|-1$. The output of the algorithm is a label for each vertex such that two vertices will have the same label if and only if they belong to the same component. In fact, the label of each vertex will be the original label of one of the vertices in the component.

ALGORITHM: CC_RANDOM_MATE(LABELS, E)

```
1  if (|E| = 0) then return LABELS
2  else
3     CHILD := {rand_bit() : v ∈ [1..n]}
4     HOOKS := {(u, v) ∈ E | CHILD[u] and ∧¬ CHILD[v]}
5     LABELS := LABELS ← HOOKS
6     E' := {(LABELS[u], LABELS[v]) : (u, v) ∈ E | LABELS[u] ≠ LABELS[v]}
7     LABELS' := CC_RANDOM_MATE(LABELS, E')
8     LABELS' := LABELS' ← {(u, LABELS' [v]) : (u, v) ∈ HOOKS}
9     return LABELS'
```

The algorithm works recursively by (a) executing one random-mate contraction step, (b) recursively applying itself to the contracted graph, and (c) reexpanding the graph by passing the labels from each root of a contracted star [from step (a)] to its children. The graph is therefore contracted while going down the recursion and expanded while coming back up. The termination condition is that there are no remaining edges (Line 1). To form stars for the contraction step the algorithm flips a coin for each vertex (Line 3) and subselects all edges HOOKS that go from a child to a parent (Line 4). We call these edges *hook edges* and they represent a superset of the star edges (each child can have multiple hook edges, but only one root in a star). For each hook edge the algorithm writes the parent's label into the child's label (Line 5). If a child has multiple neighboring parents, then one of the parent's labels is written arbitrarily—we assume an arbitrary concurrent write. At this point each child is labeled with one of its neighboring parents, if it has one. The algorithm now updates each edge by reading the labels from its two endpoints and using these as its new endpoints (Line 6). In the same step, the algorithm removes any edges that are within the same star. This gives a new sequence of edges E'. The algorithm has now completed the contraction step, and calls itself recursively on the contracted graph (Line 7). The LABELS$'$ returned by the recursive call are passed on to the children of the stars, effectively expanding the graph (Line 8). The same hooks that were used for contraction can be used for this update.

Two things should be noted about this algorithm. First, the algorithm flips coins on all of the vertices on each step even though many have already been contracted (there are no more edges that point to them). It turns out that this will not affect our worst-case asymptotic work or depth bounds, but it is not difficult to flip coins only on active vertices by keeping track of them—just keep an array of the labels of the active vertices. Second, if there are cycles in the graph, then the algorithm will create redundant edges in the contracted subgraphs. Again, keeping these edges is not a problem for the correctness or cost bounds, but they could be removed using hashing as discussed in "Removing Duplicates."

To analyze the full work and depth of the algorithm we note that each step only requires constant depth and $O(n+m)$ work. Since the number of steps is $O(\log n)$ with high probability, as mentioned earlier, the total depth is $O(\log n)$ and the work is $O((n+m)\log n)$, both with high probability. One might expect that the work would be linear since the algorithm reduces the number of vertices on each step by a constant fraction. We have no guarantee, however, that the number of edges is also going to contract geometrically, and in fact for certain graphs they will not. In "Improved Versions of Connected Components" we discuss how to improve the algorithm so that it is work-efficient.

Deterministic Graph Contraction

Our second algorithm for graph contraction is deterministic [41]. It is based on forming a set of disjoint subgraphs, each of which is tree, and then using the POINT_TO_ROOT routine ("Pointer Jumping") to contract each subgraph to a single vertex. To generate the trees, the algorithm hooks each vertex into a neighbor with a smaller label (by *hooking* a into b we mean generating a directed edge from a to b). Vertices with no smaller-labeled neighbors are left unhooked. The result of the hooking is a set of disjoint trees since hooking only from larger to smaller guarantees there are no cycles, and every node

is hooked into at most one parent. Figure 47.10 shows an example of a set of trees created by hooking. Since a vertex can have more than one neighbor with a smaller label, a given graph can have many different hookings. For example, in Fig. 47.10 vertex 2 could have hooked into vertex 1, rather than vertex 0.

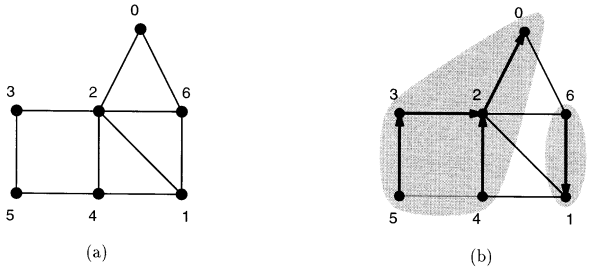


FIGURE 47.10 Tree-based graph contraction. (a) A graph G. (b) The hook edges induced by hooking larger to smaller vertices and the subgraphs induced by the trees.

The following algorithm performs the tree-based graph contraction. We assume that the labels are initialized to the indices of the vertices.

ALGORITHM: CC_TREE_CONTRACT (LABELS, E)
1 **if** ($|E| = 0$)
2 **then return** LABELS
3 **else**
4 HOOKS := $\{(u, v) \in E \mid u > v\}$
5 LABELS := LABELS $\leftarrow$ HOOKS
6 LABELS := POINT_TO_ROOT(LABELS)
7 E' := {(LABELS[u],LABELS[v]) : $(u, v) \in E \mid$ LABELS[u] $\neq$ LABELS[v]}
8 **return** CC_TREE_CONTRACT(LABELS, E')

The structure of the algorithm is similar to the random-mate graph contraction algorithm. The main differences are how the hooks are selected (Line 4), the pointer jumping step to contract the trees (Line 6), and the fact that no relabeling is required when returning from the recursive call. The hooking step simply selects edges that point from larger numbered vertices to smaller numbered vertices. This is called a *conditional hook*. The pointer jumping step uses the algorithm in Section "Pointer Jumping." This labels every vertex in the tree with the root of the tree. The edge relabeling is the same as in the random-mate algorithm. The reason the contraction algorithm does not need to relabel the vertices after the recursive call is that the pointer jumping step will do the relabeling.

Although the basic algorithm we have described so far works well in practice, in the worst case it can take $n - 1$ steps. Consider the graph in Fig. 47.11(a). After hooking and contracting only one vertex has been removed. This could be repeated up to $n - 1$ times. This worst-case behavior can be avoided by trying to hook in both directions (from larger to smaller and from smaller to larger) and picking the hooking that hooks more vertices. We make use of the following lemma.

LEMMA 47.1 Let $G = (V, E)$ be an undirected graph in which each vertex has at least one neighbor. Then either $|\{u \mid (u, v) \in E, u < v\}| \geq |V|/2$ or $|\{u \mid (u, v) \in E, u > v\}| > |V|/2$.

PROOF Every vertex must either have a neighbor with a lesser index or a neighbor with a greater index.

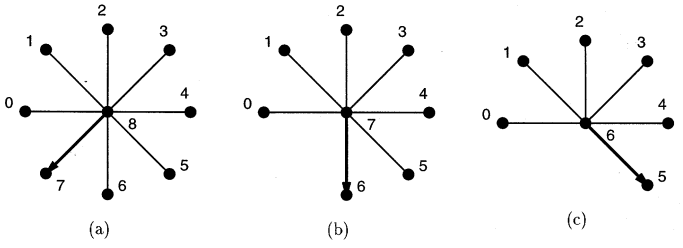


FIGURE 47.11 A worst-case graph. (a) A star graph G with the maximum index at the root of the star. (b) G after one step of contraction. (c) G after two steps of contraction.

This means that if we consider the set of vertices with a lesser neighbor and the set of vertices with a greater neighbor, then one of those sets must consist of at least one half the vertices.

This lemma will guarantee that if we try hooking in both directions and pick the better one we will remove at least 1/2 of the vertices on each step, so that the number of steps is bounded by $\lceil \log_2 n \rceil$.

We now consider the total cost of the algorithm. The hooking and relabeling of edges on each step takes $O(m)$ work and constant depth. The tree contraction using pointer jumping on each step requires $O(n \log n)$ work and $O(\log n)$ depth, in the worst case. Since there are $O(\log n)$ steps, in the worst case, the total work is $O((m + n \log n) \log n)$ and depth $O(\log^2 n)$. However, if we keep track of the active vertices (the roots) and only pointer jump on active vertices, then the work is reduced to $O((m+n) \log n)$, since the number of vertices decreases geometrically in each step. This requires that the algorithm expands the graph on the way back up the recursion as done for the random-mate algorithm. The total work with this modification is the same work as the randomized technique, although the depth has increased.

Improved Versions of Connected Components

There are many improvements to the two basic connected component algorithms we described. Here we mention some of them.

The deterministic algorithm can be improved to run in $O(\log n)$ depth with the same work bounds [13, 79]. The basic idea is to interleave the hooking steps with the **shortcutting** steps. The one tricky aspect is that we must always hook in the same direction (e.g., from smaller to larger), so as not to create cycles. Our previous technique to solve the star-graph problem therefore does not work. Instead each vertex checks if it belongs to any tree after hooking. If it does not then it can hook to any neighbor, even if it has a larger index. This is called an *unconditional hook.*

The randomized algorithm can be improved to run in optimal work, $O(n + m)$ [33]. The basic idea is to not use all of the edges for hooking on each step, and instead use a sample of the edges. This technique, first applied to parallel algorithms, has since been used to improve some sequential algorithms, such as deriving the first linear-work algorithm for finding a minimum spanning tree [46].

Another improvement is to use the EREW model instead of requiring concurrent reads and writes [42]. However this comes at the cost of greatly complicating the algorithm. The basic idea is to keep circular linked lists of the neighbors of each vertex, and then to splice these lists when merging vertices.

Extensions to Spanning Trees and Minimum Spanning Trees

The connected-component algorithms can be extended to finding a spanning tree of a graph or minimum spanning tree of a weighted graph. In both cases we assume the graphs are undirected.

A *spanning tree* of a connected graph $G = (V, E)$ is a connected graph $T = (V, E')$ such that $E' \subseteq E$ and $|E'| = |V| - 1$. Because of the bound on the number of edges, the graph T cannot have any cycles and therefore forms a tree. Any given graph can have many different spanning trees.

It is not hard to extend the connected-component algorithms to return the spanning tree. In particular, whenever components are hooked together the algorithm can keep track of which edges were used for hooking. Since each edge will hook together two components that are not connected yet, and only one edge will succeed in hooking the components, the collection of these edges across all steps will form a spanning tree (they will connect all vertices and have no cycles). To determine which edges were used for contraction, each edge checks if it successfully hooked after the attempted hook.

A *minimum spanning tree* of a connected weighted graph $G = (V, E)$ with weights $w(e)$ for $e \in E$ is a spanning tree $T = (V, E')$ of G such that

$$w(T) = \sum_{e \in E'} w(e) \tag{47.8}$$

is minimized. The connected-component algorithms can also be extended to find a minimum spanning tree. Here we will briefly consider an extension of the random-mate technique. Let us assume, without loss of generality, that all of the edge weights are distinct. If this is not the case, then lexicographical information can be added to the edges weights to break ties. It is well known that if the edge weights are distinct, then there is a unique minimum spanning tree. Furthermore, given any $W \subset V$, the minimum weight edge from W to $V - W$ must be in the minimum spanning tree. As a consequence, the minimum edge incident on a vertex will be in the minimum spanning tree. This will be true even after we contract subgraphs into vertices, since each subgraph is a subset of V.

For the minimum-spanning-tree algorithm, we modify the random mate technique so that each child u instead of picking an arbitrary parent to hook into, finds the incident edge (u, v) with minimum weight and hooks into v if it is a parent. If v is not a parent, then the child u does nothing (it is left as an orphan). Figure 47.12 illustrates the algorithm. As with the spanning-tree algorithm, we keep track of the hook

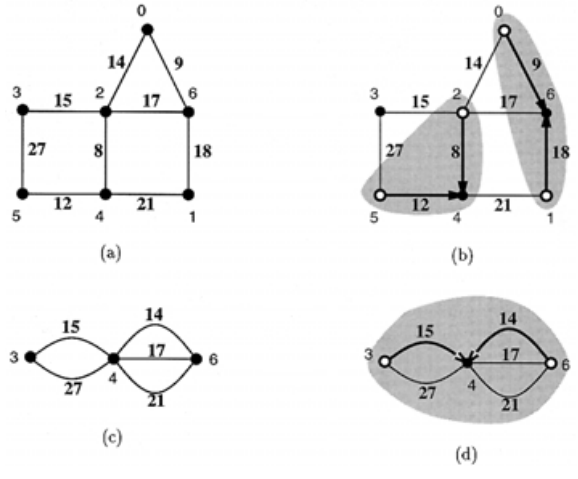


FIGURE 47.12 Example of the minimum-spanning-tree algorithm. (a) The original weighted graph G. (b) Each child (light) hooks across its minimum weighted edge to a parent (dark), if the edge is incident on a parent. (c) The graph after one step of contraction. (d) The second step in which children hook across minimum weighted edges to parents.

edges and add them to a set E'. This new rule will still remove $1/4$ of the vertices on each step on average since a vertex has $1/2$ probability of being a child, and there is $1/2$ probability that the vertex at the other end of the minimum edge is a parent. The one complication in this minimum spanning-tree algorithm is finding for each child the incident edge with minimum weight. Since we are keeping an edge list, this is not

trivial to compute. If the algorithm used an adjacency list, then it would be easy, but since the algorithm needs to update the endpoints of the edges, it is not easy to maintain the adjacency list. One way to solve this problem is to use a priority concurrent write. In such a write, if multiple values are written to the same location, the one coming from the leftmost position will be written. With such a scheme the minimum edge can be found by presorting the edges by weight so the lowest weighted edge will always win when executing a concurrent write. Assuming a priority write, this minimum-spanning-tree algorithm has the same work and depth as the random-mate connected-components algorithm.

There is also a linear-work logarithmic-depth randomized algorithm for finding a minimum-spanning tree [27], but it is somewhat more complicated than the linear-work algorithms for finding connected components.

47.6 Sorting

Sorting is a problem that admits a variety of parallel solutions. In this section we limit our discussion to two parallel sorting algorithms, QuickSort and radix sort. Both of these algorithms are easy to program, and both work well in practice. Many more sorting algorithms can be found in the literature. The interested reader is referred to [3, 45, 55] for more complete coverage.

QuickSort

We begin our discussion of sorting with a parallel version of QuickSort. This algorithm is one of the simplest to code.

ALGORITHM: QUICKSORT(A)
1 **if** $|A| = 1$ **then return** A
2 $i := rand\_int(|A|)$
3 $p := A[i]$
4 **in parallel do**
5 $L := \text{QUICKSORT}(\{a : a \in A \mid a < p\})$
6 $E := \{a : a \in A \mid a = p\}$
7 $G := \text{QUICKSORT}(\{a : a \in A \mid a > p\})$
8 **return** $L + + E + + G$

We can make an optimistic estimate of the work and depth of this algorithm by assuming that each time a partition element p is selected, it divides the set A so that neither L nor H has more than half of the elements. In this case, the work and depth are given by the recurrences

$$W(n) = 2W(n/2) + O(n) \tag{47.9}$$

$$D(n) = D(n/2) + 1 \tag{47.10}$$

which have solutions $W(n) = O(n \log n)$ and $D(n) = O(\log n)$. A more sophisticated analysis [50] shows that the expected work and depth are indeed $W(n) = O(n \log n)$ and $D(n) = O(\log n)$, independent of the values in the input sequence A.

In practice, the performance of parallel QuickSort can be improved by selecting more than one partition element. In particular, on a machine with P processors, choosing $P - 1$ partition elements divides the keys into P sets, each of which can be sorted by a different processor using a fast sequential sorting algorithm. Since the algorithm does not finish until the last processor finishes, it is important to assign approximately the same number of keys to each processor. Simply choosing $p - 1$ partition elements at random is unlikely to yield a good partition. The partition can be improved, however, by choosing a larger number,

sp, of candidate partition elements at random, sorting the candidates (perhaps using some other sorting algorithm), and then choosing the candidates with ranks $s, 2s, \ldots, (p-1)s$ to be the partition elements. The ratio s of candidates to partition elements is called the *oversampling ratio*. As s increases, the quality of the partition increases, but so does the time to sort the sp candidates. Hence there is an optimum value of s, typically larger than one, that minimizes the total time. The sorting algorithm that selects partition elements in this fashion is called *sample sort* [23, 76, 89].

Radix Sort

Our next sorting algorithm is radix sort, an algorithm that performs well in practice. Unlike QuickSort, radix sort is not a *comparison sort*, meaning that it does not compare keys directly in order to determine the relative ordering of keys. Instead, it relies on the representation of keys as b-bit integers.

The basic radix sort algorithm (whether serial or parallel) examines the keys to be sorted one "digit" position at a time, starting with the least significant digit in each key. Of fundamental importance is that this intermediate sort on digits be *stable*: the output ordering must preserve the input order of any two keys with identical digit values in the position being examined.

The most common implementation of the intermediate sort is as a counting sort. A counting sort first counts to determine the *rank* of each key—its position in the output order—and then permutes the keys by moving each key to the location indicated by its rank. The following algorithm performs radix sort assuming one-bit digits.

ALGORITHM: RADIX_SORT(A, b)

```
1  for i from 0 to b − 1
2      FLAGS := {(a >> i) mod 2 : a ∈ A}
3      NOTFLAGS := {1 − b : b ∈ B}
4      R0 := SCAN(NOTFLAGS)
5      s0 := SUM(NOTFLAGS)
6      R1 := SCAN(FLAGS)
7      R := {if FLAGS[j] = 0 then R0[j] else R1[j] + s0 : j ∈ [0..|A|)}
8      A := A ← {(R[j], A[j]) : j ∈ [0..|A|)}
9  return A
```

For keys with b bits, the algorithm consists of b sequential iterations of a **for** loop, each iteration sorting according to one of the bits. Lines 2 and 3 compute the value and inverse value of the bit in the current position for each key. The notation $a >> i$ denotes the operation of shifting a to the right by i bit positions. Line 4 computes the rank of each key that has bit value 0. Computing the ranks of the keys with bit value 1 is a little more complicated, since these keys follow the keys with bit value 0. Line 5 computes the number of keys with bit value 0, which serves as the rank of the first key that has bit value 1. Line 6 computes the relative order of the keys with bit value 1. Line 7 merges the ranks of the even keys with those of the odd keys. Finally, Line 8 permutes the keys according to rank.

The work and depth of RADIX_SORT are computed as follows. There are b iterations of the **for** loop. In each iteration, the depths of Lines 2, 3, 7, 8, and 9 are constant, and the depths of Lines 4, 5, and 6 are $O(\log n)$. Hence the depth of the algorithm is $O(b \log n)$. The work performed by each of Lines 2 through 9 is $O(n)$. Hence, the work of the algorithm is $O(bn)$.

The radix sort algorithm can be generalized so that each b-bit key is viewed as b/r blocks of r bits each, rather than as b individual bits. In the generalized algorithm, there are b/r iterations of the **for** loop, each of which invokes the SCAN function 2^r times. When r is large, a multiprefix operation can be used for generating the ranks instead of executing a SCAN for each possible value [23]. In this case, and assuming the multiprefix operation runs in linear work, it is not hard to show that as long as $b = O(\log n)$ (and

$r = \log n$, so that there are only $O(1)$ iterations), the total work for the radix sort is $O(n)$, and the depth is the same order as the depth of the multiprefix operation.

Floating-point numbers can also be sorted using radix sort. With a few simple bit manipulations, floating-point keys can be converted to integer keys with the same ordering and key size. For example, IEEE double-precision floating-point numbers can be sorted by first inverting the mantissa and exponent bits if the sign bit is 1, and then inverting the sign bit. The keys are then sorted as if they were integers.

47.7 Computational Geometry

Problems in computational geometry involve calculating properties of sets of objects in k-dimensional space. Some standard problems include finding the minimum distance between any two points in a set of points (closest-pair), finding the smallest convex region that encloses a set of points (convex-hull), and finding line or polygon intersections. Efficient parallel algorithms have been developed for most standard problems in computational geometry. Many of the sequential algorithms are based on divide-and-conquer and lead in a relatively straightforward manner to efficient parallel algorithms. Some others are based on a technique called plane sweeping, which does not parallelize well, but for which an analogous parallel technique, the *plane sweep tree* has been developed [1, 10]. In this section we describe parallel algorithms for two problems in two dimensions—closest pair and convex hull. For convex hull we describe two algorithms. These algorithms are good examples of how sequential algorithms can be parallelized in a straightforward manner.

We suggest the following sources for further information on parallel algorithms for computational geometry: [6, 39], [45, Chapter 6], and [75, Chapters 9 and 11],

Closest Pair

The *closest-pair problem* takes a set of points in k dimensions and returns the two points that are closest to each other. The distance is usually defined as Euclidean distance. Here we describe a closest-pair algorithm for two-dimensional space, also called the planar closest-pair problem. The algorithm is a parallel version of a standard sequential algorithm [16, 17], and for n points, it requires the same work as the sequential versions, $O(n \log n)$, and has depth $O(\log^2 n)$. The work is optimal.

The algorithm uses divide-and-conquer based on splitting the points along lines parallel to the y axis, and is expressed as follows.

ALGORITHM: CLOSEST_PAIR(P)
1 **if** ($|P| < 2$) **then return** (P, ∞)
2 $x_m := \text{MEDIAN}(\{x : (x, y) \in P\})$
3 $L := \{(x, y) \in P \mid x < x_m\}$
4 $R := \{(x, y) \in P \mid x \geq x_m\}$
5 **in parallel do**
6 $(L', \delta_L) := \text{CLOSEST\_PAIR}(L)$
7 $(R', \delta_R) := \text{CLOSEST\_PAIR}(R)$
8 $P' := \text{MERGE\_BY\_Y}(L', R')$
9 $\delta_P := \text{BOUNDARY\_MERGE}(P', \delta_L, \delta_R, x_m)$
10 **return** (P', δ_P)

This function takes a set of points P in the plane and returns both the original points sorted along the y axis, and the distance between the closest two points. The sorted points are needed to help merge the results from recursive calls, and can be thrown away at the end. It would not be difficult to modify the routine to return the closest pair of points in addition to the distance between them. The function

works by dividing the points in half based on the median x value, recursively solving the problem on each half and then merging the results. The MERGE_BY_Y function merges L' and R' along the y axis and can use a standard parallel merge routine. The interesting aspect of the code is the BOUNDARY_MERGE routine, which works on the principle described by Bentley and Shamos [16, 17], and can be computed with $O(\log n)$ depth and $O(n)$ work. We first review the principle and then show how it can be performed in parallel.

The inputs to BOUNDARY_MERGE are the original points P sorted along the y axis, the closest distance within L and R, and the median point x_m. The closest distance in P must be either the distance δ_L, the distance δ_R, or a distance between a point in L and a point in R. For this distance to be less than δ_L or δ_R, the two points must lie within $\delta = \min(\delta_L, \delta_R)$ of the line $x = x_m$. Thus, the two vertical lines at $x_l = x_m - \delta$ and $x_r = x_m + \delta$ define the borders of a region M in which the points must lie (see Fig. 47.13). If we could find the closest distance in M, call it δ_M, then the closest overall distance would be $\delta_P = \min(\delta_L, \delta_R, \delta_M)$.

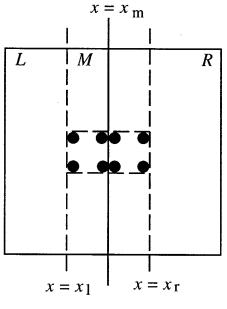


FIGURE 47.13 Merging two rectangles to determine the closest pair. Only eight points can fit in the $2\delta \times \delta$ dashed rectangle.

To find δ_M we take advantage of the fact that not many points can be packed close together within M since all points within L or R must be separated by at least δ. Figure 47.13 shows the tightest possible packing of points in a $2\delta \times \delta$ rectangle within M. This packing implies that if the points in M are sorted along the y axis, each point can determine the minimum distance to another point in M by looking at a fixed number of neighbors in the sorted order, at most 7 in each direction. To see this consider one of the points along the top of the $2\delta \times \delta$ rectangle. To determine if there are any points below it that are closer than δ we need only to consider the points within the rectangle (points below the rectangle must be further than δ away). As the figure illustrates, there can be at most 7 other points within the rectangle. Given this property, the following function performs the border merge.

ALGORITHM: BOUNDARY_MERGE$(P, \delta_L, \delta_R, x_m)$

```
1  δ := min(δ_L, δ_R)
2  M := {(x, y) ∈ P | (x ≥ x_m − δ) ∧ (x ≤ x_m + δ)}
3  δ_M := min({min({distance(M[i], M[i + j]) : j ∈ [1..7]})
4              : i ∈ [0..|P| − 7)}
5  return min(δ, δ_M)
```

For each point in M this function considers the seven points following it in the sorted order and determines the distance to each of these points. It then takes the minimum over all distances. Since the distance relationship is symmetric, there is no need to consider points appearing before a point in the sorted order.

The work of BOUNDARY_MERGE is $O(n)$ and the depth is dominated by the taking the minimum, which has $O(\log n)$ depth.[2] The work of the merge and median steps in CLOSEST_PAIR are also $O(n)$, and the depth of both are bounded by $O(\log n)$. The total work and depth of the algorithm can therefore be expressed by the recurrences

$$W(n) = 2W(n/2) + O(n) \quad (47.11)$$

$$D(n) = D(n/2) + O(\log n) \quad (47.12)$$

which have solutions $W(n) = O(n \log n)$ and $D(n) = O(\log^2 n)$.

Planar Convex Hull

The convex-hull problem takes a set of points in k dimensions and returns the smallest convex region that contains all the points. In two dimensions the problem is called the planar convex-hull problem, and it returns the set of points that form the corners of the region. These points are a subset of the original points. We will describe two parallel algorithms for the planar convex-hull problem. They are both based on divide-and-conquer, but one does most of the work before the divide step, and the other does most of the work after.

QuickHull

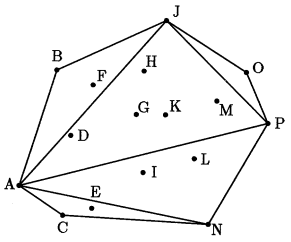
The parallel *QuickHull* algorithm is based on the sequential version [71], so named because of its similarity to the QuickSort algorithm. As with QuickSort, the strategy is to pick a "pivot" element, split the data into two sets based on the pivot, and recurse on each of the sets. Also as with QuickSort, the pivot element is not guaranteed to split the data into equal-sized sets, and in the worst case the algorithm requires $O(n^2)$ work. In practice, however, the algorithm is often very efficient, probably the most practical of the convex hull algorithms. At the end of the section we briefly describe how the splits can be made so that the work is guaranteed to be bounded by $O(n \log n)$.

The QuickHull algorithm is based on the recursive function SUBHULL, which is expressed as follows.

ALGORITHM: SUBHULL(P, p_1, p_2)
1 $P' := \{p \in P \mid \text{LEFT\_OF?}(p, (p_1, p_2))\}$
2 **if** ($|P'| < 2$)
3 **then return** $[p_1] + + P'$
4 **else**
5 $i := \text{MAX\_INDEX}(\{\text{DISTANCE}(p, (p_1, p_2)) : p \in P'\})$
6 $p_m := P'[i]$
7 **in parallel do**
8 $H_l := \text{SUBHULL}(P', p_1, p_m)$
9 $H_r := \text{SUBHULL}(P', p_m, p_2)$
10 **return** $H_l + + H_r$

This function takes a set of points P in the plane and two points p_1 and p_2 that are known to lie on the convex hull, and returns all the points that lie on the hull clockwise from p_1 to p_2, inclusive of p_1, but not of p_2. For example in Fig. 47.14 SUBHULL($[A, B, C, \ldots, P]$, A, P) would return the sequence $[A, B, J, O]$.

[2] The depth of finding the minimum or maximum of a set of numbers can actually be improved to $O(\log \log n)$ with concurrent reads [78].



[A B C D E F G H I J K L M N O P]
A [B D F G H J K M O] P [C E I L N]
A [B F] J [O] P N [C E]
A B J O P N C

FIGURE 47.14 An example of the *QuickHull* algorithm.

The function SUBHULL works as follows. Line 1 removes all the elements that cannot be on the hull because they lie to the right of the line from p_1 to p_2. Determining which side of a line a point lies on can easily be calculated with a few arithmetic operations. If the remaining set P' is either empty or has just one element, the algorithm is done. Otherwise the algorithm finds the point p_m farthest from the line (p_1, p_2). The point p_m must be on the hull since as a line at infinity parallel to (p_1, p_2) moves toward (p_1, p_2), it must first hit p_m. In Line 5 the function MAX_INDEX returns the index of the maximum value of a sequence, which is then used to extract the point p_m. Once p_m is found, SUBHULL is called twice recursively to find the hulls from p_1 to p_m, and from p_m to p_2. When the recursive calls return, the results are appended.

The following function uses SUBHULL to find the full convex hull.

ALGORITHM: QUICKHULL(P)
1 $X := \{x : (x, y) \in P\}$
2 $x_{\min} := P[\text{MIN\_INDEX}(X)]$
3 $x_{\max} := P[\text{MAX\_INDEX}(X)]$
4 **return** SUBHULL($P, x_{\min}, x_{\max}$) $+ +$ SUBHULL($P, x_{\max}, x_{\min}$)

We now consider the cost of the parallel QuickHull, and in particular the SUBHULL routine, which does all the work. The call to MAX_INDEX uses $O(n)$ work and $O(\log n)$ depth. Hence, the cost of everything other than the recursive calls is $O(n)$ work and $O(\log n)$ depth. If the recursive calls are balanced so that neither recursive call gets much more than half of the data then the number of levels of recursion will be $O(\log n)$. This will lead to the algorithm running in $O(\log^2 n)$ depth. Since the sum of the sizes of the recursive calls can be less than n (e.g., the points within the triangle AJP will be thrown out when making the recursive calls to find the hulls between A and J and between J and P), the work can be as little as $O(n)$, and often is in practice. As with QuickSort, however, when the recursive calls are badly partitioned the number of levels of recursion can be as bad as $O(n)$ with work $O(n^2)$. For example, consider the case when all the points lie on a circle and have the following unlikely distribution. The points x_{min} and x_{max} appear on opposite sides of the circle. There is one point that appears half way between x_{min} and x_{max} on the sphere and this point becomes the new x_{max}. The remaining points are defined recursively. That is, the points become arbitrarily close to x_{min} (see Fig. 47.15).

Kirkpatrick and Seidel [49] have shown that it is possible to modify QuickHull so that it makes provably good partitions. Although the technique is shown for a sequential algorithm, it is easy to parallelize. A simplification of the technique is given by Chan et al. [25]. Their algorithm admits even more parallelism and leads to an $O(\log^2 n)$-depth algorithm with $O(n \log h)$ work where h is the number of points on the convex hull.

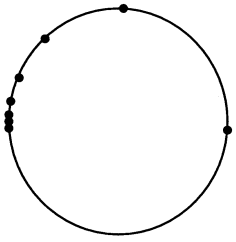


FIGURE 47.15 Contrived set of points for worst-case QuickHull.

MergeHull

The MergeHull algorithm [68] is another divide-and-conquer algorithm for solving the planar convex hull problem. Unlike QuickHull, however, it does most of its work after returning from the recursive calls. The function is expressed as follows.

ALGORITHM: MERGEHULL(P)
1 **if** ($|P| < 3$) **then return** P
2 **else**
3 **in parallel do**
4 H_L = MERGEHULL($P[0..|P|/2)$)
5 H_R = MERGEHULL($P[|P|/2..|P|)$)
6 **return** JOIN_HULLS(H_L, H_R)

This function assumes the input P is presorted according to the x coordinates of the points. Since the points are presorted, H_L is a convex hull on the left and H_R is a convex hull on the right. The JOIN_HULLS routine is the interesting part of the algorithm. It takes the two hulls and merges them into one. To do this it needs to find lower and upper points l_1 and u_1 on H_L and l_2 and u_2 on H_R such that l_1, l_2 and u_1, u_2 are successive points on H (see Fig. 47.16). The lines b_1 and b_2 joining these upper and lower points are called the upper and lower bridges, respectively. All the points between l_1 and u_1 and between u_2 and l_2 on the "outer" sides of H_L and H_R are on the final convex hull, while the points on the "inner" sides are not on the convex hull. Without loss of generality we only consider how to find the upper bridge b_1. Finding the lower bridge b_2 is analogous.

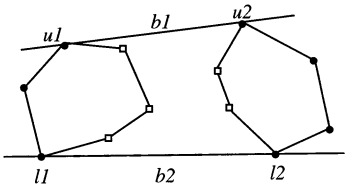


FIGURE 47.16 Merging two convex hulls.

To find the upper bridge one might consider taking the points with the maximum y values on H_L and H_R. This approach does not work in general, however, since u_1 can lie as far down as the point with the minimum x or maximum x value (see Fig. 47.17). Instead there is a nice solution due to Overmars and van Leeuwen [68] based on a dual binary search. Assume that the points on the convex hulls are given in order (e.g., clockwise). At each step the binary search algorithm will eliminate half of the remaining points from consideration in either H_L or H_R or both. After at most $\log |H_L| + \log |H_R|$ steps the search will be left with only one point in each hull, and these will be the desired points u_1 and u_2. Figure 47.18 illustrates the rules for eliminating part of H_L or H_R on each step.

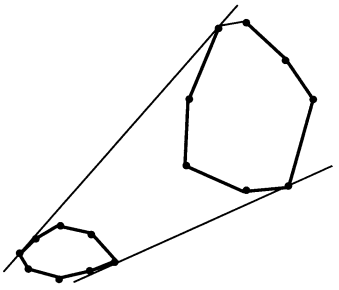


FIGURE 47.17 A bridge that is far from the top of the convex hull.

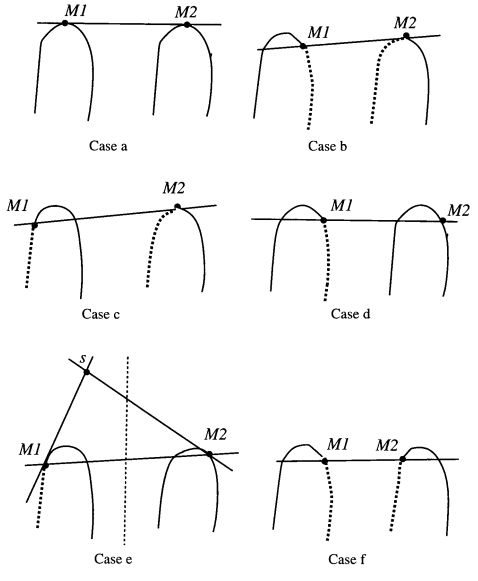


FIGURE 47.18 Cases used in the binary search for finding the upper bridge for the MergeHull. The points $M1$ and $M2$ mark the middle of the remaining hulls. In case (a), all of H_L and H_R lie below the line through $M1$ and $M2$. In this case, the line segment between $M1$ and $M2$ is the bridge. In the remaining cases, dotted lines represent the parts of the hulls H_L and H_R that can be eliminated from consideration. In cases (b) and (c), all of H_R lies below the line through $M1$ and $M2$, and either the left half of H_L or the right half of H_L lies below the line. In cases (d) through (f), neither H_L nor H_R lies entirely below the line. In the case (e), the region to eliminate depends on which side of a line separating H_L and H_R the intersection of the tangents appears. The mirror images of cases (b) through (e) are also used. Case (f) is actually an instance of case (d) and its mirror image.

We now consider the cost of the algorithm. Each step of the binary search requires only constant work and depth since we need only to consider the two middle points M_1 and M_2, which can be found in constant time if the hull is kept sorted. The cost of the full binary search to find the upper bridge is therefore bounded by $D(n) = W(n) = O(\log n)$. Once we have found the upper and lower bridges we need to remove the points on H_L and H_R that are not on H and append the remaining convex hull points. This requires linear work and constant depth. The overall costs of MERGEHULL are therefore

$$D(n) = D(n/2) + O(\log n) = O(\log^2 n) \tag{47.13}$$

$$W(n) = 2W(n/2) + O(n) = O(n \log n) \tag{47.14}$$

This algorithm can be improved to run in $O(\log n)$ depth using one of two techniques. The first involves modifying the search for the bridge points so that it runs in constant depth with linear work [12]. This

involves sampling every $\sqrt{n}$th point on each hull and comparing all pairs of these two samples to narrow the search down to regions of size $\sqrt{n}$ in constant depth. The regions can then be finished in constant depth by comparing all pairs between the two regions. The second technique [1, 11] uses divide-and-conquer to separate the point set into $\sqrt{n}$ regions, solves the convex hull on each region recursively, and then merges all pairs of these regions using the binary-search method. Since there are $\sqrt{n}$ regions and each of the searches takes $O(\log n)$ work, the total work for merging is $O((\sqrt{n})^2 \log n) = O(n \log n)$ and the depth is $O(\log n)$. This leads to an overall algorithm that runs in $O(n \log n)$ work and $O(\log n)$ depth. The algorithms above require concurrent reads (**CREW**). The same bounds can be achieved with exclusive reads (EREW) [66].

47.8 Numerical Algorithms

There has been an immense amount of work on parallel algorithms for numerical problems. Here we briefly discuss some of the problems and results. We suggest the following sources for further information on parallel numerical algorithms [75, Chapters 12-14], [45, Chapter 8], [53, Chapters 5, 10 and 11] and [18].

Matrix Operations

Matrix operations form the core of many numerical algorithms and led to some of the earliest work on parallel algorithms. The most basic matrix operation is matrix multiplication. The standard triply nested loop for multiplying two dense matrices is highly parallel since each of the loops can be parallelized:

ALGORITHM: MATRIX_MULTIPLY(A, B)
1 $(l, m) := dimensions(A)$
2 $(m, n) := dimensions(B)$
3 **in parallel for** $i \in [0..l)$ **do**
4 **in parallel for** $j \in [0..n)$ **do**
5 $R_{ij} := sum(\{A_{ik} * B_{kj} : k \in [0..m)\})$
6 **return** R

If $l = m = n$, this routine does $O(n^3)$ work and has depth $O(\log n)$, due to the depth of the summation. This has much more parallelism than is typically needed, and most of the research on parallel matrix multiplication has concentrated on what subset of the parallelism to use so that communication costs can be minimized. Sequentially it is known that matrix multiplication can be performed using less than $O(n^3)$ work. Strassen's algorithm [82], for example, requires only $O(n^{2.81})$ work. Most of these more efficient algorithms are also easy to parallelize because they are recursive in nature (Strassen's algorithm has $O(\log n)$ depth using a simple parallelization).

Another basic matrix operation is to invert matrices. Inverting dense matrices has proven to be more difficult to parallelize than multiplying dense matrices, but the problem still supplies plenty of parallelism for most practical purposes. When using Gauss–Jordan elimination, two of the three nested loops can be parallelized, leading to an algorithm that runs with $O(n^3)$ work and $O(n)$ depth. A recursive block-based method using matrix multiplication leads to the same depth, although the work can be reduced by using a more efficient matrix-multiplication algorithm. There are also more sophisticated, but less practical, work-efficient algorithms with depth $O(\log^2 n)$ [28, 70].

Parallel algorithms for many other matrix operations have been studied, and there has also been significant work on algorithms for various special forms of matrices, such as tridiagonal, triangular, and sparse matrices. Iterative methods for solving sparse linear systems has been an area of significant activity.

Fourier Transform

Another problem for which there is a long history of parallel algorithms is the discrete Fourier transform (DFT). The fast Fourier transform (FFT) algorithm for solving the DFT is quite easy to parallelize and, as with matrix multiply, much of the research has gone into reducing communication costs. In fact the butterfly network topology is sometimes called the FFT network, since the FFT has the same communication pattern as the network [55, Section 3.7]. A parallel FFT over complex numbers can be expressed as follows:

ALGORITHM: $\mathrm{FFT}(A)$
1 $n := |A|$
2 **if** $(n = 1)$ **then return** A
3 **else**
4 **in parallel do**
5 $\mathrm{EVEN} := \mathrm{FFT}(\{A[2i] : i \in [0..n/2)\})$
6 $\mathrm{ODD} := \mathrm{FFT}(\{A[2i+1] : i \in [0..n/2)\})$
7 **return** $\{\mathrm{EVEN}[j] + \mathrm{ODD}[j] e^{2\pi ij/n} : j \in [0..n/2)\} + +\{\mathrm{EVEN}[j] - \mathrm{ODD}[j] e^{2\pi ij/n} : j \in [0..n/2)\}$

The algorithm simply calls itself recursively on the odd and even elements and then puts the results together. This algorithm does $O(n \log n)$ work, as does the sequential version, and has a depth of $O(\log n)$.

47.9 Research Issues and Summary

Recent work on parallel algorithms has focused on solving problems from domains such as pattern matching, data structures, sorting, computational geometry, combinatorial optimization, linear algebra, and linear and integer programming. For pointers to this work, see Section "Further Information."

Algorithms have also been designed specifically for the types of parallel computers that are available today. Particular attention has been paid to machines with limited communication bandwidth. For example, there is a growing library of software developed for the BSP model [40, 62, 85].

The parallel computer industry has been through a period of financial turbulence, with several manufacturers failing or discontinuing sales of parallel machines. In the past few years, however, a large number of inexpensive small-scale parallel machines have been sold. These machines typically consist of 4 to 8 commodity processors connected by a bus to a shared-memory system. As these machines reach the limit in size imposed by the bus architecture, manufacturers have reintroduced parallel machines based on the hypercube network topology (e.g., [54]).

47.10 Defining Terms

CREW: This refers to a shared memory model that allows for concurrent reads (CR) but only exclusive writes (EW) to the memory.

CRCW: A shared memory model that allows for concurrent reads (CR) and concurrent writes (CW) to the memory.

Depth: The longest chain of sequential dependencies in a computation.

EREW: A shared memory model that allows for only exclusive reads (ER) and exclusive writes (EW) to the memory.

Graph contraction: Contracting a graph by removing a subset of the vertices.

List contraction: Contracting a list by removing a subset of the nodes.

Multiprefix: A generalization of the scan (**prefix sums**) operation in which the partial sums are grouped by keys.

Multiprocessor model: A model of parallel computation based on a set of communicating sequential processors.

Pipelined divide-and-conquer: A divide-and-conquer paradigm in which partial results from recursive calls can be used before the calls complete. The technique is often useful for reducing the depth of an algorithm.

Pointer jumping: In a linked structure, replacing a pointer with the pointer it points to. Used for various algorithms on lists and trees.

PRAM model: A multiprocessor model in which all processors can access a shared memory for reading or writing with uniform cost.

Prefix sums: A parallel operation in which each element in an array or linked-list receives the sum of the previous elements.

Random sampling: Using a randomly selected sample of the data to help solve a problem on the whole data.

Recursive doubling: The same as pointer jumping.

Scan: A parallel operation in which each element in an array receives the sum of all the previous elements.

Shortcutting: Same as pointer jumping.

Tree contraction: Contracting a tree by removing a subset of the nodes.

Symmetry breaking: A technique to break the symmetry in a structure such as a graph which can locally look the same to all the vertices. Usually implemented with randomization.

Work: The total number of operations taken by a computation.

Work-depth model: A model of parallel computation in which one keeps track of the total work and depth of a computation without worrying about how it maps onto a machine.

Work-efficient: A parallel algorithm is work-efficient if asymptotically (as the problem size grows) it requires at most a constant factor more work than the best known sequential algorithm (or the optimal work).

Work-preserving: A translation of an algorithm from one model to another is work-preserving if the work is the same in both models, to within a constant factor.

References

[1] Aggarwal, A., Chazelle, B., Guibas, L., Dúnlaing, C.O., and Yap, C., Parallel computational geometry, *Algorithmica,* 3(3), 293–327, 1988.

[2] Aho, A.V., Hopcroft, J.E., and Ullman, J.D., *The Design and Analysis of Computer Algorithms,* Addison-Wesley, Reading, MA, 1974.

[3] Akl, S.G., *Parallel Sorting Algorithms,* Academic Press, Orlando, FL, 1985.

[4] Akl, S.G., *The Design and Analysis of Parallel Algorithms,* Prentice Hall, Englewood Cliffs, NJ, 1989.

[5] Akl, S.G., *Parallel Computation: Models and Methods,* Prentice Hall, Englewood Cliffs, NJ, 1997.

[6] Akl, S.G. and Lyons, K.A., *Parallel Computational Geometry,* Prentice Hall, Englewood Cliffs, NJ, 1993.

[7] Almasi, G.S. and Gottlieb, A., *Highly Parallel Computing,* Benjamin/Cummings, Redwood City, CA, 1989.

[8] Anderson, R.J. and Miller, G.L., A simple randomized parallel algorithm for list-ranking. *Information Processing Letters,* 33(5), 269–273, Jan. 1990.

[9] Anderson, R.J. and Miller, G.L., Deterministic parallel list ranking. *Algorithmica,* 6(6), 859–868, 1991.
[10] Atallah, M.J., Cole, R., and Goodrich, M.T., Cascading divide-and-conquer: A technique for designing parallel algorithms, *SIAM Journal of Computing,* 18(3), 499–532, June 1989.
[11] Atallah, M.J. and Goodrich, M.T., Efficient parallel solutions to some geometric problems, *Journal of Parallel and Distributed Computing,* 3(4), 492–507, Dec. 1986.
[12] Atallah, M.J. and Goodrich, M.T., Parallel algorithms for some functions of two convex polygons, *Algorithmica,* 3(4), 535–548, 1988.
[13] Awerbuch, B. and Shiloach, Y., New connectivity and MSF algorithms for shuffle-exchange network and PRAM, *IEEE Transactions on Computers,* C–36(10), 1258–1263, Oct. 1987.
[14] Bar-Noy, A. and Kipnis, S., Designing broadcasting algorithms in the postal model for message-passing systems, *Mathematical Systems Theory,* 27(5), 341–452, Sept./Oct. 1994.
[15] Beneš, V.E., *Mathematical Theory of Connecting Networks and Telephone Traffic,* Academic Press, New York, 1965.
[16] Bentley, J.L.,. Multidimensional divide-and-conquer, *Communications of the Association for Computing Machinery,* 23(4), 214–229, Apr. 1980.
[17] Bentley, J.L. and Shamos, M.I., Divide-and-conquer in multidimensional space, In *Conference Record of the Eighth Annual ACM Symposium on Theory of Computing,* 220–230, May 1976.
[18] Bertsekas, D.P. and Tsitsiklis, J.N., *Parallel and Distributed Computation: Numerical Methods,* Prentice-Hall, Englewood Cliffs, NJ, 1989.
[19] Blelloch, G.E., *Vector Models for Data-Parallel Computing,* MIT Press, Cambridge, MA, 1990.
[20] Blelloch, G.E., Programming parallel algorithms, *Communications of the ACM,* 39(3), 85–97, Mar. 1996.
[21] Blelloch, G.E., Chandy, K.M., and Jagannathan, S., Eds., *Specification of Parallel Algorithms. Volume 18 of DIMACS Series in Discrete Mathematics and Theoretical Computer Science,* American Mathematical Society, Providence, RI, 1994.
[22] Blelloch, G.E. and Greiner, J., Parallelism in sequential functional languages, In *Proceedings of the Symposium on Functional Programming and Computer Architecture,* 226–237, June 1995.
[23] Blelloch, G.E., Leiserson, C.E., Maggs, B.M., Plaxton, C.G., Smith, S.J., and Zagha, M., An experimental analysis of parallel sorting algorithms, *Theory of Computing Systems,* 31(2), 135–167, Mar./Apr. 1998.
[24] Brent, R.P., The parallel evaluation of general arithmetic expressions, *Journal of the Association for Computing Machinery,* 21(2), 201–206, Apr. 1974.
[25] Chan, T.M., Snoeyink, J., and Yap, C.-K., Output-sensitive construction of polytopes in four dimensions and clipped Voronoi diagrams in three, In *Proceedings of the 6th Annual ACM–SIAM Symposium on Discrete Algorithms,* 282–291, Jan. 1995.
[26] Cole, R., Parallel merge sort, *SIAM Journal of Computing,* 17(4), 770–785, Aug. 1988.
[27] Cole, R., Klein, P.N., and Tarjan, R.E., Finding minimum spanning forests in logarithmic time and linear work, In *Proceedings of the 8th Annual ACM Symposium on Parallel Algorithms and Architectures,* 243–250, June 1996.
[28] Csanky, L., Fast parallel matrix inversion algorithms, *SIAM Journal on Computing,* 5(4), 618–623, Apr. 1976.
[29] Culler, D.E., Karp, R.M., Patterson, D., Sahay, A., Santos, E.E., Schauser, K.E., Subramonian, R., and von Eicken, T., LogP: A practical model of parallel computation, *Communications of the Association for Computing Machinery,* 39(11), 78–85, Nov. 1996.
[30] Cypher, R. and Sanz, J.L.C., *The SIMD Model of Parallel Computation,* Springer-Verlag, New York, 1994.
[31] Eppstein, D. and Galil, Z., Parallel algorithmic techniques for combinatorial computation, *Annual Review of Computer Science,* 3, 233–83, 1988.

[32] Fortune, S. and Wyllie, J., Parallelism in random access machines, In *Conference Record of the Tenth Annual ACM Symposium on Theory of Computing,* 114–118, May 1978.
[33] Gazit, H., An optimal randomized parallel algorithm for finding connected components in a graph, *SIAM Journal on Computing,* 20(6), 1046–1067, Dec. 1991.
[34] Gelernter, D., Nicolau, A., and Padua, D., Eds., *Languages and Compilers for Parallel Computing. Research Monographs in Parallel and Distributed,* MIT Press, Cambridge, MA, 1990.
[35] Gibbons, A. and Rytter, W., *Efficient Parallel Algorithms,* Cambridge University Press, Cambridge, England, 1988.
[36] Gibbons, P.B., Matias, Y., and Ramachandran, V., The QRQW PRAM: Accounting for contention in parallel algorithms, In *Proceedings of the 5th Annual ACM–SIAM Symposium on Discrete Algorithms,* 638–648, Jan. 1994.
[37] Goldschlager, L.M., A unified approach to models of synchronous parallel machines, In *Conference Record of the Tenth Annual ACM Symposium on Theory of Computing,* 89–94, May 1978.
[38] Goldschlager, L.M., A universal interconnection pattern for parallel computers, *Journal of the Association for Computing Machinery,* 29(3), 1073–1086, Oct. 1982.
[39] Goodrich, M.T., Parallel algorithms in geometry, In *CRC Handbook of Discrete and Computational Geometry,* J.E. Goodman and J.O'Rourke, Eds., 669–682. CRC Press, Boca Raton, FL, 1997.
[40] Goudreau, M., Lang, K., Rao, S., Suel, T., and Tsantilas. T., Towards efficiency and portability: Programming with the BSP model, In *Proceedings of the 8th Annual ACM Symposium on Parallel Algorithms and Architectures,* 1–12, June 1996.
[41] Greiner, J. and Blelloch, J.E., Connected components algorithms, In *High Performance Computing: Problem Solving with Parallel and Vector Architectures,* G.W. Sabot, Ed., Addison Wesley, Reading, MA, 1995.
[42] Halperin, S. and Zwick, U., An optimal randomised logarithmic time connectivity algorithm for the EREW PRAM, *Journal of Computer and Systems Sciences,* 53(3), 395–416, Dec. 1996.
[43] Harris, T.J., A survey of PRAM simulation techniques, *ACM Computing Surveys,* 26(2), 187–206, June 1994.
[44] Hennessy, J.L. and Patterson, D.A., *Computer Architecture: A Quantitative Approach,* Morgan Kaufmann, San Francisco, 2nd ed., 1996.
[45] JáJá, J., *An Introduction to Parallel Algorithms,* Addison Wesley, Reading, MA, 1992.
[46] Karger, D.R., Klein, P.N., and Tarjan, R.E., A randomized linear-time algorithm to find minimum spanning trees, *Journal of the Association for Computing Machinery,* 42(2), 321–328, Mar. 1995.
[47] Karlin, A.R. and Upfal, E., Parallel hashing: an efficient implementation of shared memory, *Journal of the Association for Computing Machinery,* 35(5), 876–892, Oct. 1988.
[48] Karp, R.M. and Ramachandran, V., Parallel algorithms for shared-memory machines, In *Handbook of Theoretical Computer Science,* J.van Leeuwen, Ed., Vol. A: *Algorithms and Complexity,* 869–941. Elsevier Science Publishers, Amsterdam, The Netherlands, 1990.
[49] Kirkpatrick, D.G. and Seidel, R., The ultimate planar convex hull algorithm? *SIAM Journal on Computing,* 15(1), 287–299, Feb. 1986.
[50] Knuth, D.E., *Sorting and Searching,* Vol. 3 of *The Art of Computer Programming,* Addison-Wesley, Reading, MA, 1973.
[51] Kogge, P.M. and Stone, H.S., A parallel algorithm for the efficient solution of a general class of recurrence equations, *IEEE Transactions on Computers,* C–22(8), 786–793, Aug. 1973.
[52] Kruskal, C.P., Searching, merging, and sorting in parallel computation, *IEEE Trans. Comput.,* C–32(10), 942–946, Oct. 1983.
[53] Kumar, V., Grama, A., Gupta, A., and Karypis, G., *Introduction to Parallel Computing: Design and Analysis of Algorithms,* Benjamin/Cummings, Redwood City, CA, 1994.

[54] Laudon, J. and Lenoski, D., The SGI Origin: a ccNUMA highly scalable server, In *Proceedings of the 24th Annual International Symposium on Computer Architecture,* 241–251, June 1997.
[55] Leighton, F.T., *Introduction to Parallel Algorithms and Architectures: Arrays • Trees • Hypercubes,* Morgan Kaufmann, San Mateo, CA, 1992.
[56] Leiserson, C.E., Fat-trees: Universal networks for hardware-efficient supercomputing, *IEEE Transactions on Computers,* C–34(10), 892–901, Oct. 1985.
[57] Lengauer, T., VLSI theory, In *Handbook of Theoretical Computer Science,* J.van Leeuwen, Ed., Vol. A: *Algorithms and Complexity,* 837–868. Elsevier Science Publishers, Amsterdam, The Netherlands, 1990.
[58] Luby, M., A simple parallel algorithm for the maximal independent set problem, *SIAM Journal on Computing,* 15(4), 1036–1053, Nov. 1986.
[59] Lynch, N.A., *Distributed Algorithms,* Morgan Kaufmann, San Francisco, 1996.
[60] Maon, Y., Schieber, B., and Vishkin, U., Parallel ear decomposition search (EDS) and st-numbering in graphs, *Theoretical Comput. Sci.,* 47, 277–298, 1986.
[61] Matias, Y. and Vishkin, U., On parallel hashing and integer sorting, *Journal of Algorithms,* 12(4), 573–606, Dec. 1991.
[62] McColl, W.F., BSP programming. In *Specification of Parallel Algorithms,* Vol. 18: *DIMACS Series in Discrete Mathematics and Theoretical Computer Science,* G.E. Blelloch, K.M. Chandy, and S.Jagannathan, Ed., 25–35. American Mathematical Society, Providence, RI, May 1994.
[63] Miller, G.L. and Ramachandran, V., A new graph triconnectivity algorithm and its parallelization, *Combinatorica,* 12(1), 53–76, 1992.
[64] Miller, G.L. and Reif, J.H., Parallel tree contraction part 1: Fundamentals, In *Randomness and Computation,* Vol. 5: *Advances in Computing Research,* S.Micali, Ed., 47–72. JAI Press, Greenwich, CT, 1989.
[65] Miller, G.L. and Reif, J.H., Parallel tree contraction part 2: Further applications, *SIAM Journal of Computing,* 20(6), 1128–1147, Dec. 1991.
[66] Miller, R. and Stout, Q.F., Efficient parallel convex hull algorithms, *IEEE Transactions on Computers,* 37(12), 1605–1619, Dec. 1988.
[67] Miller, R. and Stout, Q.F., *Parallel Algorithms for Regular Architectures,* MIT Press, Cambridge, MA, 1996.
[68] Overmars, M.H. and van Leeuwen, J., Maintenance of configurations in the plane, *Journal of Computer and System Sciences,* 23(2), 166–204, Oct. 1981.
[69] Pratt, V.R. and Stockmeyer, L.J., A characterization of the power of vector machines, *Journal of Computer and System Sciences,* 12(2), 198–221, Apr. 1976.
[70] Preparata, F. and Sarwate, D., An improved parallel processor bound in fast matrix inversion, *Information Processing Letters,* 7(3), 148–150, Apr. 1978.
[71] Preparata, F.P. and Shamos, M.I., *Computational Geometry—An Introduction,* Springer-Verlag, New York, 1985.
[72] Ranade, A.G., How to emulate shared memory, *Journal of Computer and System Sciences,* 42(3), 307–326, June 1991.
[73] Ranka, S. and Sahni, S., *Hypercube Algorithms: With Applications to Image Processing and Pattern Recognition,* Springer-Verlag, New York, 1990.
[74] Reid-Miller, M., List ranking and list scan on the Cray C90, *Journal of Computer and Systems Sciences,* 53(3), 344–356, Dec. 1996.
[75] Reif, J.H., Ed., *Synthesis of Parallel Algorithms,* Morgan Kaufmann, San Mateo, CA, 1993.
[76] Reif, J.H. and Valiant, L.G., A logarithmic time sort for linear size networks, *Journal of the Association for Computing Machinery,* 34(1), 60–76, Jan. 1987.
[77] Savitch, W.J. and Stimson, M., Time bounded random access machines with parallel processing, *Journal of the Association for Computing Machinery,* 26(1), 103–118, Jan. 1979.

[78] Shiloach, Y. and Vishkin, U., Finding the maximum, merging and sorting in a parallel computation model, *Journal of Algorithms,* 2(1), 88–102, Mar. 1981.
[79] Shiloach, Y. and Vishkin, U., An $O(\log n)$ parallel connectivity algorithm, *Journal of Algorithms,* 3(1), 57–67, Mar. 1982.
[80] Siegel, H.J., *Interconnection Networks for Large-Scale Parallel Processing: Theory and Case Studies,* 2nd ed., McGraw–Hill, New York, 1990.
[81] Stone, H.S., Parallel tridiagonal equation solvers, *ACM Transactions on Mathematical Software,* 1(4), 289–307, Dec. 1975.
[82] Strassen, V., Gaussian elimination is not optimal, *Numerische Mathematik,* 14(3), 354–356, 1969.
[83] Tarjan, R.E. and Vishkin, V., An efficient parallel biconnectivity algorithm, *SIAM Journal of Computing,* 14(4), 862–874, Nov. 1985.
[84] Ullman, J.D., *Computational Aspects of VLSI,* Computer Science Press, Rockville, MD, 1984.
[85] Valiant, L.G., A bridging model for parallel computation, *Communications of the ACM,* 33(8), 103–111, Aug. 1990.
[86] Valiant, L.G., General purpose parallel architectures, In *Handbook of Theoretical Computer Science,* J.van Leeuwen, Ed., Vol. A: *Algorithms and Complexity,* 943–971. Elsevier Science Publishers, Amsterdam, The Netherlands, 1990.
[87] Vishkin, U., Parallel-design distributed-implementation (PDDI) general purpose computer, *Theoretical Computer Science,* 32(1–2), 157–172, July 1984.
[88] Wyllie, J.C., The complexity of parallel computations, Technical Report TR-79-387, Department of Computer Science, Cornell University, Ithaca, NY, Aug. 1979.
[89] Yang, M.C.K., Huang, J.S., and Chow, Y., Optimal parallel sorting scheme by order statistics, *SIAM Journal on Computing,* 16(6), 990–1003, Dec. 1987.

Further Information

In a chapter of this length, it is not possible to provide comprehensive coverage of the subject of parallel algorithms. Fortunately, there are several excellent textbooks and surveys on parallel algorithms including [4, 5, 6, 18, 30, 31, 35, 45, 48, 53, 55, 67, 75, 80].

There are many technical conferences devoted to the subjects of parallel computing and computer algorithms, so keeping abreast of the latest research developments is challenging. Some of the best work in parallel algorithms can be found in conferences such as the ACM Symposium on Parallel Algorithms and Architectures, the IEEE International Parallel Processing Symposium, the IEEE Symposium on Parallel and Distributed Processing, the International Conference on Parallel Processing, the International Symposium on Parallel Architectures, Algorithms, and Networks, the ACM Symposium on the Theory of Computing, the IEEE Symposium on Foundations of Computer Science, the ACM–SIAM Symposium on Discrete Algorithms, and the ACM Symposium on Computational Geometry.

In addition to parallel algorithms, this chapter has also touched on several related subjects, including the modeling of parallel computations, parallel computer architecture, and parallel programming languages. More information on these subjects can be found in [7, 43, 44, 86], and [21, 34], respectively. Other topics likely to interest the reader of this chapter include distributed algorithms [59] and VLSI layout theory and computation [57, 84].

48

Distributed Computing: A Glimmer of a Theory

Eli Gafni
U.C.L.A.

48.1 Introduction

What Is the Area About?

Distributed computing theory develops computability and complexity theories for models whose computation involves many processors interacting in certain limited unpredictable manner through some communication objects, where a **processor** is shorthand for a sequential piece of code that includes instructions, some of which involve access to the communication objects.

One of the defining characteristics of the models tackled by distributed computing is a **nondeterminism** due to run-time events whose possibility of occurrence or the order in which they might occur is unpredictable. Moreover, distributed computing takes as its primary domain models whose nondeterminism gives their processors differing views of the world. Thus, for example, distributed computing will examine a model in which the failure of a processor is noticeable to some of its processors but not to others, but it will not examine a model in which the outcome of a flip of a coin (i.e., a randomized outcome) is made available to all processors simultaneously. In the parlance of knowledge [20], distributed computing studies systems in which the current state of a processor is not common knowledge, even when the input is known to all of the processors.

Research in distributed computing has focused on a number of models distinguished by the different communication objects and different timing constraints. Among those investigated most extensively are the asynchronous message-passing model, the asynchronous shared-memory model, asynchronous models in which at most t processors may fail-stop, and synchronous models with Byzantine, omission, or fail-stop failures.

What Is this Chapter About?

Here I argue that in the chaos that has characterized the field of distributed computing theory, we have begun to see signs of an underlying order. In the 1980s, as Lynch and Lamport [25] have observed, "the field seemed to consist of a collection of largely unrelated results about individual models." They were right at the time. Researchers concentrated on obtaining efficient solutions to problems, and although this was and is an important and productive line of research, most of the ideas leading to efficiency seem ad hoc and are closely tied to a specific model and, at times, to a specific problem instance. Few of the ideas leading to efficient solutions supply a methodology that can be applied in other contexts.

Furthermore, the arguments that have led to the most fundamental impossibility results of the field seem to share no common thread. One fundamental negative result, FLP impossibility [16], states that no agreement can be reached in an asynchronous system in which even a single processor may fail-stop unannounced. The original arguments that led to this result were procedural and very much tied to the properties of the operations specific to the model. Concurrently, the impossibility of reaching Byzantine agreement in time equal to the number of faults was proven [15] through the use of a **chain** of compatible views that led from a run in which one decision has to be taken to a run in which another decision has to be taken. Consequently, the argument went, there must be a link in the chain at which incompatible decisions are taken by different processors in the same run. At about the same time, the procedural inductive proof of the nonexistence of a solution to the so-called two-armies problem was replaced by a slick proof using a newly developed knowledge theory [20, 18]. Although the connection between the latter two impossibility results, one in the context of synchronous systems and the other in the context of asynchronous systems, was at the time vaguely understood, the connection between them and FLP impossibly seemed nonexistent.

In recent years tools have been developed, and research has been fruitful in making connections between results and models in a way that parallels what has been done in complexity theory. Where complexity theory is not much help in supplying a receipt to determine whether a specific problem at hand is $\Theta(n^2)$ or $\Theta(n^3)$ other then referral to some generic problems that have been thoroughly investigated, distributed computing can similarly provide only hints in the form of pointers to similar problems, in determining time and message complexity of a specific problem. Where complexity theory has tool and techniques to categorize problems in a "broad-brush," e.g., P vs. NP, distributed computing theory has over recent years built a formidable machinery to classify problems by the models in which they are solvable.

Problems have been identified that are analogous to **complete** problems. They characterize a model. The problem is solvable in the characterized model, and any model in which the problem is solvable, has a set of solvable problems that is a superset of the problems solvable in the characterized model. Collections of models, whose set of solvable problems is identical, have been identified, where in some models the level of the unpredictability of the possible order of events is lesser than in others. A problem is then checked for solvability in the model that exhibit the least unpredictability, if one exists. It provides, on the one hand, short uncluttered impossibility argument, or on the other hand, it provides a succinct protocol. Such a protocol may be viewed as written in a high-level programming language, with a guaranteed automatic compilation of the protocol to a protocol in the target model.

Some starts that stalled, and some partial successes in the direction indicated above occurred in the 1980s. Knowledge theory [20] was developed, showed some promise for certain problems, but eventually led, in my opinion, to no major breakthrough, in the understanding of distributed computing. A way of coaching 1-resilient models within graph theory led to a complete characterization of these models [9], providing an automatic impossibility proof on one hand or a protocol on the other, to the question of a problem

solvability in the model. This raised the hope that protocol derivation, or the realization of its nonexistence, can in general be automated. And finally, a helpful instance of the "high-level-programming," mentioned above, was discovered with the realization of the equivalence between message passing systems and shared memory systems on the one hand, and between shared memory and atomic-snapshot shared-memory on the other. These instances of model equivalence allowed the investigation of the power of message-passing systems, within the much less detailed model of atomic-snapshot shared-memory.

The break-through was enabled when researchers started to zooming on complete problems. Herlihy [19] showed that the number processor that can achieve consensus characterizes and differentiates certain models. While consensus takes n input values and allows a single output, Chaudhuri proposed to check for models that allow several but less than n outputs. In 1993 three teams were able to show that her problems are also complete for certain models. The techniques employed resulted in the revelation of a connection between distributed computing and topology, be it algebraic [22], combinatorial [5], or point-set topology [29].

Extensive research effort followed. If by "theory" one means a body of tightly related results, building one upon the other, and readily providing explanations to many observable phenomena, then, as I will attempt to show in this chapter, a theory of distributed computing has emerged.

The survey in this chapter has a narrow focus. The advances outlined above are presented. The utility of the "modern tools" that result in are demonstrated in deriving past results. The next section, on models, discusses distributed systems in general and defines the various notions of emulation and tasks which are central to this chapter. We then embark upon a sequence of emulations. We start with the asynchronous shared-memory model for two and three processors, characterizing them for fail-stop and Byzantine failures and, via transformation, we apply all these results to the message-passing model. Then we investigate the shared-memory model in the synchronous domain, and we show how an impossibility result in the asynchronous model translates into lower bound on complexity in the synchronous model. The next step is to link the two models through the notion of failure detectors, and the new notion of an iterated model, and to explicate the utility of characterizing a synchronous system as an asynchronous one with a (weak) failure detector [12]. The concluding section raises some of the major questions that, in my opinion, are facing this emerging theory of distributed computing. Obviously, the chapter leaves unaddressed many of the very large number of techniques developed in distributed computing; in most cases, how they may be incorporated into the framework reported here is not yet clear. The reader is encouraged to look up this wealth of techniques in the excellent textbook by Lynch [26], and two of the surveys by Lamport and Lynch [25], and Attiya [6].

48.2 Models

An asynchronous distributed computation model is the set of all sequential interleaving of communication actions performed by sequential processes or processors on shared communication objects. A communication action may be thought of as the invocation of a remote procedure call by a processor at an object. An object may be thought of as a processor that executes the remote procedure call, changes its state, and responds to the invoking processor by returning a value. The returned value causes the invoking processor to change its state, which in turn determines the next parameter for the next access to a communication object. In this chapter we assume that processors never halt, and therefore after each return of an invocation, they enter a state in which a new invocation is enabled. In the network model, processors access unidirectional point-to-point communication channels. A single communication object is associated with two processors, called sender and receiver, respectively. The sender can invoke an action **send**(m) on the object. The effect of the action is to place a message m in the buffer of the object. After placing the message in its buffer the object responds by returning an *ok* to the sender. The receiver invokes an action **receive** which moves a message from the buffer of the object to the receiver, if such a message exists, or the object

responds by notifying exception otherwise. In the shared memory model, communication objects are read/write registers on which the action of read and write can be invoked.

In this chapter we assume that communication objects do not fail. Yet, in light of the view of a communication object as a "restricted" processor, it is not surprising that when communication failures are taken into account [3], they give rise to results reminiscent of processor failures.

Given a **protocol** — the instantiation of processors with codes — and the initial conditions of processors and objects, we define a space R of **runs** to be a subset of the infinite sequences of processors names. Since we assume that a processor has a single enabled invocation at a time, such a sequence when interpreted as the order in which enabled invocations were executed completely determines the evolution of the computation. Before the system starts all runs in which the processor has its current input are possible. As the system evolves, the local state of the processor excludes some runs. Thus with a local state of a processor we associate a **view** — the set of all runs in R that are not excluded by the local state. By making processors maintain their history in local memory we may assume that consecutive views of a processor are monotonically nondecreasing. Thus, with each run $r \in R$ of a protocol p we can associate a limit view $\lim(V_i(r, p))$ of processor P_i. A protocol f is **full-information** if for all i, r, and p we have $\lim(V_i(r, f)) \subseteq \lim(V_i(r, p))$. Intuitively, a full-information protocol does not economize on the size of its local state, or the size of the parameter to its object invocation. Models which are **oblivious,** that is, the sequence of communication objects a processor will access is the same for all protocols, possess a full-information protocol. All the models in this chapter do. In the rest of this chapter, a protocol stands for the full-information one, and correspondingly a model is associated with a single protocol — its full-information protocol. One can define the notion of full-information protocol with respect to a specific protocol in a nonoblivious model, but we will not need this notion here. A sequence of runs $r_1, r_2, \ldots$ converges to a run r, if r_k and r share a longer and longer prefix as k increases.

It can be observed from the definition of a view, that two views of the same processor, are either disjoint, or related by containment. Given an intermediate view $V_i(r)$ of a processor P_i in run r, we say that a processor **outputs** its view in r if for all P_j which have infinitely many distinct views in r, $\lim(V_j(r)) \subseteq V_i(r)$. Processor P_i is **faulty** in r if it outputs finitely many views. Processor P_i is **participating** in r if it outputs any nontrivial view. Otherwise it is **sleeping** in r. A model A with n processors with communication object O_A **wait-free emulates** a model B with n processors and communication objects O_B if there is a map m from runs R_A in A to runs R_B in B such that

1. The sets of sleeping processors and faulty processors in r and $m(r)$ are identical.
2. The map m is continuous with respect to prefixes. That is if $r_1, r_2, \ldots$ in A converges to r, then $m(r_1), m(r_2), \ldots$ in B converges to $m(r)$. This captures the idea that the map does not predict the future.
3. The map m does not utilize detailed information about the past of a run, if this detailed information is not available through processors' views. Formally, for all P_j nonfaulty and for all r in A, $m(\lim(V_j(r))) \subseteq \lim(V_j(m(r)))$

We say that A **nonblocking** emulates B if we relax the first condition by allowing the mapping m to fail any nonfaulty processor as long as an infinite sequence is mapped into an infinite sequence. Two models are wait-free (nonblocking) equivalent if they wait-free (nonblocking) emulate each other.

A specification of a problem Π on n processors, is a relation from runs to sets of "output-sequences." Each output in the sequence is associate with a unique processor. A model with n processors wait-free solves Π if there exists a map from views to outputs, such the map of the projection of a run on views that are output in the run, is an output-sequence that relates to the run. A problem Π on n processors is non-blocking solvable in a model, if the relaxed problem $\bar{\Pi}$ is wait-free solvable, where $\bar{\Pi}$ takes each element in Π and closes the output-sequence set with respect to removal of infinite suffixes of processors' outputs (as long as the sequence remains infinite).

A **task** is a relaxation of a problem Π such that only bounded prefixes of output-sequences matter. That is to say that past some number of outputs any output is acceptable. Since the notion of participating set is invariant over models, the runs that are distinguished by different output requirements in a task are those that differ in their participating set. Thus in this chapter we employ the notion of task in this restricted sense. In the **consensus** task a processor first outputs its private value, which is either 0 or 1, and then outputs a consensus value. Consensus values agree, and match the input of at least one of the participating processors. In the **election** task a processor outputs its ID and then outputs an election value which is an ID of a participating processor. All election values agree. A run with a single participating processor is a **solo-execution** of that processor.

A model is t**-resilient** if we require that it solves a problem only over runs in which at most t processors are faulty.

A **synchronous** model is one which progresses in rounds. In each round all the communication actions enabled by the beginning of the round are executed by the end of the round.

48.3 Asynchronous Models

Two-Processor Shared-Memory Model

Consider a two-processor single-writer/multi-reader (SWMR) shared-memory system. In such a system, there are two processors P_1 and P_0, and two shared-memory cells C_1 and C_0. Processor P_i writes exclusively to C_i, but it can read the other cell. Both shared-memory cells are initialized to $\perp$. W.l.o.g. computation proceeds with each processor alternately writing to its cell and reading the cell of the other processor.

Can this two-processor system 1-resiliently (wait-free in this case, since for $n = 2$, $n - 1 = 1$) elect one of the processors as the leader? No one-step full-information protocol, and consequently no one-step protocol at all, for solving this problem exists. Consider the state of processor P_1 after writing and reading. It could have read what processor P_0 wrote (denoted by $P_1 : w_0$), or it could have missed what processor P_0 wrote (denoted by $P_1 : \perp$). Thus, we have four possible views, two for each processor, after one step.

In the graph whose nodes are these views, two views are connected by an undirected edge if there is an execution that gives rise to the two views. The resulting graph appears in Fig. 48.1. Since a processor has a single view in an execution, edges connect nodes labeled by distinct processor IDs. The two nodes of distinct IDs which do not share an edge are $P_1 : \perp$ and $P_0 : \perp$. This follows from the fact that in shared memory in which processors first write and then read, the processor that writes second must read the value of the processor that writes first.

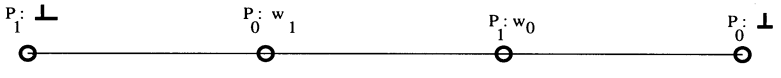


FIGURE 48.1 One-step view graph.

The edge $\{P_1 : \perp, P_0 : w_1\}$ corresponds to the execution: P_1 writes, P_1 reads, P_0 writes, P_0 reads. If we could map the view of a processor after one step into an output, then processor P_1 in this edge is bound to elect P_1, since the possibility of a solo execution by the processor has not yet been eliminated. Similarly, in the edge $\{P_0 : \perp, P_1 : w_0\}$, processor P_0 is bound to elect P_0. Thus, no matter what processor is elected by P_1 and P_0 in the views $P_1 : w_0$ and $P_0 : w_1$, respectively, we are bound to create an edge where at one end P_1 is elected and at the other end P_0 is elected. Because there is an execution in which both processors are elected, we must conclude that there is no one-step 1-resilient protocol for election with two processors.

To confirm that no k-step full-information protocol exists, we could draw the graph of the views after k steps, and observe that the graph contains a path connecting the views $\{P_1 : \perp\}$ and $\{P_0 : \perp\}$.

It is not easy to see that indeed the observation above holds. Given that our goal is an argument that will generalize to more than two processors, we have to be able to get a handle on the general explicit structure of the shared-memory model for any number of processors. This has been an elusive challenge. Instead, we turn to iterated shared memory, a model in which the structure of the graph of a k-step two-processor full-information protocol is easily verified to be a path. We then argue that for two processors, the shared-memory model and the iterated shared-memory model are nonblocking equivalent. We then show that this line of argumentation generalizes to any number of processors.

Two-Processor Iterated Shared-Memory Model

For any model M in which the notion of one-shot use of the model exists, one can define the iterated counterpart $\bar{M}$ of M. In $\bar{M}$, the processors go through a sequence of stages of one-shot use of M in which the output of the $(k-1)$th stage is in turn the input to the kth stage.

To iterate the two-processor SWMR shared-memory model, we take two sequences of cells: $C_{1,1}, C_{1,2}, C_{1,3}, \ldots$, and $C_{0,1}, C_{0,2}, C_{0,3}, \ldots$. Processor P_1 writes its input to $C_{1,1}$ and then reads $C_{0,1}$. Inductively, P_1 then takes its view after reading $C_{0,(k-1)}$, writes this view into $C_{1,k}$, reads $C_{0,k}$, and so on.

The iterated model is related to the notion of a **communication-closed layer** [14]. This accounts for why algorithms in the iterated model are easy both to understand and to prove correct: one may imagine that there is a barrier synchronization after each stage such that no processor proceeded to the current stage until all processors had executed (asynchronously) the previous stage.

In an execution, if the view of P_1 after reading $C_{0,(k-1)}$ is X and the corresponding view for P_0 is Y, then the graph of the views after the kth stage, given the view after the $k-1$'st stage, appears in Fig. 48.2. This graph is the same as the graph for the one-shot SWMR shared-memory model when X is the input to P_1 (and stands therefore for w_1) and Y is the input to P_0 (and stands therefore for w_0). To get the graph of all the possible views after the kth stage, we inductively take the graph after the $(k-1)$th stage, replace each edge with a path of three edges, and label the nodes appropriately. Thus, after k stages, we get a path of 3^k edges. At one end of this path is a view that is bound to output P_1, at the other a view bound to be output P_0, which leads to the conclusion that there is no k-stage protocol in the model for any k.

FIGURE 48.2 One-step view graph after the kth stage with input X,Y.

It is easy to see that the shared-memory model nonblocking implements its iterated version by dividing cell C_i into a sequence of on-demand virtual cells $C_{i,1}, C_{i,2}, C_{i,3}, \ldots$. Processors then "pretend" to read only the appropriate cell. To the see that a non-blocking emulation in the reverse direction is also possible we consider processors to $WriteRead$ sequence numbers. Processor P_1 keeps an estimate v_{p_0} of the last sequence number P_0 wrote. To $WriteRead_1(v)$, processor P_1 writes the pair v_{p_0}, v into the next cell in the sequence. It then read the other. If it contains $\perp$, or it contains the same pair it has written, then the operation terminates, and P_1 returns the pair it wrote to its cell. Otherwise, it updates v_{p_0} to the maximum between the value it held and the value it read, and continues [8].

Characterization of Solvability for Two Processors

What tasks can two processors in shared memory solve 1-resiliently? They can solve, for instance, the following task. Processor P_1 in a solo execution outputs 1, and processor P_0 in a solo execution outputs 10. In every case, the two processors must output values between 1 and 10 whose absolute difference is exactly 1. This task can be solved easily, since in the iterated shared-memory model, after three stages the path contains

10 nodes and these nodes can be associated one-to-one with the integers 1 through 10. The task may be represented using domino pieces. There is an infinite number of pieces, each labeled P_1, x on one side and P_0, y on the other side, where all x and y are real numbers and $|x - y| = 1,\ 1 \leq x \leq 10,\ 1 \leq y \leq 10$. The task is solvable iff one can create a domino path with the pieces such that one side of the path is labeled P_1, 1 and the other side P_0, 10. It is easy to see that if processor P_0 had to output 11 (rather than 10) in a solo execution, the problem would not be solvable.

A generalized version is solvable if processors in solo executions output their integer inputs (which come, for the moment, from a bounded domain of integers) and the tuples of inputs are such that one input value is odd and the other is even. To see that the input (1,8) is solvable, take the output from the second stage of the iterated shared-memory model and fold three consecutive view edges on a single output edge. In algebraic and combinatorial topology, an edge is called a **1-simplex,** a node is called a **0-simplex,** an edge that is subdivided into a path is called a **one-dimensional subdivided simplex,** and a graph is called a **complex.** Thus, for a problem to be solvable 1-resiliently by two processors, the output complex must contain a subdivided simplex in which the labels on the two boundary nodes are the processors with their corresponding solo-execution outputs and the ID labels on the path alternate (colored subdivided simplex).

What if we want to solve the infinite version of the task where the possible integer inputs are not bounded? In this case, the difficulty is that we cannot place an *a priori* upper bound on the number of iterated steps we need to take in the iterated model. One solution is to map the infinite line to a semicircle, then do the appropriate convergence on the semicircle, and map back. Another solution, denoted by Π, proceeds as follows. Processor P_1 with input k_1 takes k_1 steps if it reads $\perp$ continuously. Otherwise, after P_1 reads a value for P_0, it stops once it reads $\perp$ or reads that P_0 has read a value from P_1. Clearly, if the input values are k_1 and k_2, then the view complex is a path of length $k_1 + k_2$ that can be folded into the interval $[k_1, k_2]$ with enough views to cover all the integers (see [4]).

In the case of shared memory (not iterated), if a processor halts once it takes some number of steps or once it observes the other processor take one step, we say that the processor halts within a unit of time. Consider the full-information version of Π (a protocol in the iterated model). An execution of the full-information protocol can be interpreted as an execution of a nonblocking emulation of the atomic-snapshot shared-memory model. If we take this view, then Π translates into a unit-time algorithm.

This conclusion, in fact, holds true for any task that is solvable 1-resiliently by two processors. It is easy to see that by defining the unit as any desirable $\epsilon > 0$, we can get two processors to output real numbers that are within an ϵ-ball (ϵ-agreement). To solve any problem, we fix an embedding of a path that may account for the solvability of the task. Consequently, there exists an ϵ such that for any interval I of length ϵ, all the simplexes that overlap the interval have a common intersection. Processors then conduct ϵ-agreement on the path, and each processor adopts as an output the node of its label which is closest to its ϵ-agreement output.

This view of convergence does not generalize easily to more than two processors. Therefore, we propose another interpretation for the two-processor convergence process [8]. After an $\epsilon/2$-agreement as above, P_i observes the largest common intersection s_i of the simplexes overlapping the ϵ-length interval that is centered around P_i's ϵ-agreement value. It must be that $s_1 \cup s_2$ is a simplex and that $s_1 \cap s_2 \neq \emptyset$. P_i posts s_i in shared memory. It then takes the intersection of the s_j's it observes posted. If a node labeled by P_i's ID is in the intersection, P_i outputs that node. Otherwise, P_i sees only one node, v, in the intersection, and v has a label different from P_i's ID. In this case, P_i outputs one of the nodes labeled by its own ID which appear in a simplex along with v (these nodes are said to be "in the **link** of v").

Thus, since solvability amounts to ϵ-agreement, and ϵ-agreement can be achieved 1-resiliently within a unit of time, we conclude that any task 1-resiliently solvable by two processors can be solved within a unit of time.

Three-Processor 2-Resilient Model

We consider now the three-processor 2-resilient SWMR shared-memory model. W.l.o.g. processors alternate between writing and reading the other two cells one by one. Obviously, we cannot elect a leader (since two processors cannot), but perhaps we can solve the $(3, 2)$ set-consensus problem in which processors elect at most two leaders (i.e., each processor outputs an ID of a participating processor, and the union of the outputs is of cardinality at most 2).

The structure of the full-information protocol of the one-shot shared-memory model for three processors is not as easy to identify as that for two processors. Two-processor executions have many hidden properties that are lost when we have three processors. For example, in a two-processor execution, when a processor reads the value of the other processor's cell, then in conjunction with the value it has last written to its own cell, the processor has an instantaneous view of how the memory looks, as if it read both cell in a single atomic operation. Such an instantaneous copy is called an **atomic snapshot** (or, for short, a snapshot) [1]. In a three-processor system, this property is lost; we cannot interpret a read operation as returning a snapshot.

To get the effect of processor P_1 reading cells C_2 and C_0 instantaneously, we have P_1 read C_2 and C_0 repeatedly until the values the processor reads do not change over two consecutive repetitions. If all values written are distinct, then the values the processor reads reside simultaneously in the memory in an instant that is after the first of the two consecutive repetitions and before the second of the repetitions. Thus, P_1 may safely return these values.

Clearly, three processors can 2-resiliently nonblocking implement one-shot snapshot memory—a memory in which a processor writes its cell and then obtains a vector of values for all cells, and this vector is a snapshot. Yet, a one-shot snapshot may give processors the following views: $P_1 : w_1, w_2, w_0$, $P_2 : w_1, w_2, w_0$, $P_0 : \perp, w_2, w_0$. This is the result of the execution: P_2 writes, P_0 writes, P_0 takes a snapshot, P_1 writes, P_1 and P_2 take a snapshot. Can we require that the set of processors S_i that return at most i values return snapshots only of values written by processors from S_i? In the example above, we have $S_2 = \{P_0\}$, but P_0 returns a value from P_2 which is not in S_2. A snapshot whose values are restricted in this way is called an **immediate snapshot** [7, 29]. A recursive distributed procedure will return immediate snapshots (program for P_i):

1. Procedure-immediate-snapshot $ISN(P_i, k)$.
2. Write input to cell $C_{k,i}$.
3. For $j = 1$ to n, read cell $C_{k,j}$.
4. If the number of values ($\neq \perp$) is k, then return everything read; else, call immediate snapshot $ISN(P_i, k-1)$.

We assume that all cells are initialized to $\perp$ and that in a system with n processors, P_i starts by calling immediate snapshot $ISN(P_i, n)$.

It is not difficult to prove that the view complex of a one-shot immediate snapshot is a subdivided simplex. Simple extension of the algorithm outlined for two processors shows that, in general, the iterated immediate-snapshot model nonblocking implements the shared-memory atomic-snapshot model [8]. The view complex for three processors is shown in Fig. 48.3.

We now argue by way of example that after the first stage in the iterated immediate-snapshot model, processors cannot elect two leaders. By definition, the view $P_i : w_i$ has to be mapped to P_i. The views $P_i : w_i, w_j$ are mapped to P_i or P_j. The rest are mapped to any processor ID. Such a mapping of views to processor IDs constitutes a Sperner coloring of the subdivided simplex [28]. The Sperner Lemma then says that there must be a triangle colored by the three colors. Since a triangle is at least one execution, we have proven that no election of two leaders is possible. The argument we made about the coloring of a path by two processors' IDs is just the one-dimensional instance of the Sperner Lemma. By the recursive properties of iterated models, we conclude that the structure of a k-step three-processor 2-resilient iterated immediate snapshot is the structure of a subdivided triangle.

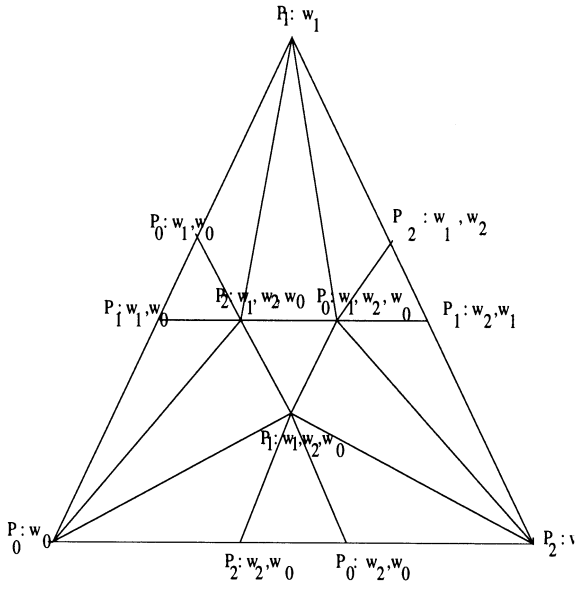


FIGURE 48.3 Three-processor one-shot immediate-snapshot view complex.

What nontrivial tasks can we solve? For one, the task of producing a one-shot immediate snapshot is far from trivial. In general, we can solve anything that the view complex of a sufficiently large number of iterations of one-shot immediate snapshots can map to, color and boundary preserving, simplicially. This includes any subdivided triangle A [23].

We show algorithmically how any three processors converge to a triangle on a colored subdivided triangle A. Embedding a reasonably large-enough complex of the iterated immediate snapshots embedded over A yields two-dimensional ϵ-agreement over A, since triangles of the view complex can be inscribed in smaller and smaller circles as a function of the number of iterations we take. Thus, as we did for two processors, we may argue now that an $\epsilon > 0$ exists such that for any ϵ-ball in A, all simplexes that overlap the ball have a common intersection. P_i then conducts $\epsilon/2$-agreement and posts the largest such intersection s_i of the simplexes overlapping the $\epsilon/2$-radius ball whose center is P_i's $\epsilon/2$-agreement value. P_i takes the intersection of the s_is posted and, if P_i's color is present, adopts the value of that A node. Otherwise, P_i removes a node of P_i's color, if one exists, from the union of the simplexes P_i observed to get a simplex x_i and then starts a new ϵ-agreement from a node of P_i's color in the link of x_i.

As we argued for two processors, when there are three processors, at least one processor will terminate after the first ϵ-agreement. If two processors show up for the second agreement, they have identified the processor that will not proceed; the link is at worst a closed path. The convergence of the remaining two processors is interpreted to take place on one side of the closed path rather than on the other, and the decision about on which side convergence takes place is made according to a predetermined rule that is a function of the starting nodes. The convergence of two processors on a path was outlined in the previous section.

To see that ϵ-agreement for three processors is solvable 2-resiliently on a triangle within a unit of time, we notice that if we again take an iterated algorithm with the stopping rule that processors halt once they reach some bound or once they learn that they read from each other, as in the case of two processors, it can easily be argued inductively that we get a subdivided simplex "growing from the center." As we increase the bound on the solo and pairs executions, we "add a layer" around the previous subdivided simplex. Thus, we have a mesh that becomes as fine as we need and results in ϵ-agreement. Our stopping rule, when converted via nonblocking emulation of shared memory by iterated shared memory, results in a unit-time algorithm. Since we have seen that solving a task amounts to ϵ-agreement on the convex hull,

we conclude that in the snapshot model, an algorithm exists for any wait-free solvable task such that at least one processor can obtain an output within a unit of time (see [4]).

Now, if we view three-processor tasks as a collection of triangular domino pieces that can be reshaped into triangles of any size we want, we see that again the question of solvability amounts to a tiling problem. Where in the two-processor case we had two distinct domino ends that we had to tile in between, we now have three distinct domino corners, corresponding to solo executions, and three distinct domino sides, corresponding to executions in which only two processors participated. To solve a three-processor task, we have to pack the domino pieces together to form a subdivided triangle that complies with the boundary conditions on the sides. Unfortunately, this two-dimensional tiling problem is undecidable in general [17].

Three-Processor 1-Resilient Model

What kind of three-processor tasks are 1-resiliently solvable [9]? It stands to reason that such systems are more powerful than two-processor 1-resiliency. Perhaps the two nonfaulty processors can gang up on the faulty one and decide.

In light of our past reasoning, the natural way to answer this question is to ask what the analogue to the one-shot immediate snapshot is in this situation. A little thought shows that the analogue comprises two stages of the wait-free one-shot immediate snapshot where the nodes that correspond to solo executions have been removed. (We need to two stages because one stage with solo executions removed is a task that is not solvable 1-resiliently by three processors.) This structure can be solved as a task by the three-processor 1-resilient model, and the iteration of this structure is a model that can nonblocking implement a shared-memory model of the three-processor 1-resilient model.

An inductive argument shows that this structure is connected, and consequently consensus is impossible. Furthermore, the link of any node is connected. If we have a task satisfying these conditions, the convergence argument of the previous section shows how to solve the task. We start at possibly only two nodes because one processor may wait on one of the other two. By simple connectivity, we can converge so that at least one processor terminates. The other two can converge on the link of the terminating processor since this link is connected.

Thus a task is solvable 1-resiliently in a system with three processors if it contains paths with connected links, connecting solo executions of two processors. Checking whether this holds amounts to a reachability problem and therefore is decidable.

Models with Byzantine Failure

What if a processor not only may fail to take further steps but also may write anything to its cell in the asynchronous SWMR memory? Such a processor is said to fail in a Byzantine fashion. Byzantine failures have been dealt with in the asynchronous model within the message passing system [10]. Here we define it for the shared-memory environment and show that the essential difficulty is actually in transforming a message-passing system to a shared-memory one with write-once cells.

Why a Byzantine failure is not more harmful than a regular fail-stop failure when in a write-once cells shared-memory environment? If we assume that cell C_i is further subdivided into sub-cells $C_{i1}, C_{i2}, \ldots$ and that these subcells are write-once only, we can nonblocking emulate iterated snapshot shared memory in which a processor writes and then reads all cells in a snapshot. All processors now have to comply with writing in the form of snapshots—namely, they must write a set of values from the previous stage, otherwise what they write will be trivially discarded. Yet, faulty processors may post snapshots that are inconsistent with the snapshots posted by other, nonfaulty processors. We observe that we can resolve inconsistent snapshots by letting their owners to revise these snapshots to snapshots that are the union of the inconsistent ones. This allows processors to affirm some snapshots and not affirm others. A processor that observes another snapshot that is inconsistent with its own snapshot will suggests the union snapshot

if its snapshot has not been affirmed yet. This processor waits with a snapshot consistent with the others until one of its snapshots has been affirmed. It then writes one of its affirmed snapshots as a final one. Thus, all other processors may check that a processor's final snapshot have been affirmed.

A processor affirms a snapshot by observing it to be consistent with the rest and writing an affirmation for it. If we wait for at least $n/2 + 1$ affirmations (discard a processor which affirms two inconsistent snapshots as faulty), then it can be seen that no inconsistent snapshots will be affirmed. In other words, we have transformed Byzantine failure to fail-stop at the cost of nonblocking emulating the original algorithm if it was not written in the iterated style.

The next problem to be addressed is how to nonblocking emulate subdivision to a write-once subcell of a cell C_i when a processor may overwrite the values previously written to the cell. The following procedure nonblocking emulates a read of $C_{i,k}$:

1. If read $C_{i,k} \neq \perp$ or if read $f + 1$ processors claiming a value $v \neq \perp$ for $C_{i,k}$, claim v for $C_{i,k}$.
2. If read $2f + 1$ processors claiming v for $C_{i,k}$ or if read $f + 1$ processors wrote Confirm(v) for $C_{i,k}$, write Confirm(v) for $C_{i,k}$.
3. If read $2f + 1$ processors claiming Confirm(v) for $C_{i,k}$, accept v for $C_{i,k}$.

Clearly, once a processor accepts a value, all processors will accept that value eventually and so a value may not change.

Since the availability of write-once cells allows Byzantine agreement we conclude that like Byzantine agreement the implementation of write-once cells requires that less of a third of the processors fail. In hindsight, we recognize that Bracha discovered in 1987 that a shared-memory system with Byzantine failure and write-once cells can be nonblocking implemented on a message-passing system with the same failures as the shared-memory system provided that $3f < n$ [10]. It took another three years for researchers to independently realize the simpler transformation from message passing to shared memory for fail-stop faults [2].

Message-Passing Model

Obviously, shared memory can nonblocking emulate message passing. The conceptual breakthrough made by Attiya, Bar-Noy, and Dolev (ABD) [2] was to realize that for $2f < n$, message passing can f-resiliently wait-free emulate f-resilient shared memory, as follows. To emulate a write, a processor takes the latest value it is to write, sends the value to all of the processors, and waits for acknowledgment from a majority. To read, a processor asks for the latest value for the cell from a majority of the processors and then writes it. A processor keeps an estimate of the latest value for a cell; this estimate is the value with the highest sequence number that the processor has ever observed for the cell. Since two majorities intersect, the emulation works.

All results of shared memory now apply verbatim, by either the fail-stop or the Byzantine transformation, to message passing. To derive all the above directly in message passing is complex, because, in one way or the other, hidden ABD or Bracha transformations sit there and obscure the real issues. The ABD and Bracha transformations have clarified why shared memory is a much cleaner model than message passing to think and argue about.

From Safe-Bits to Atomic Snapshots

We now show that in retrospect the result of the research conducted during the second part of the 1980s on the power of shared-memory made out of safe-bits is not surprising.

We started this section with a SWMR shared-memory model. Can such a model be nonblocking implemented from the most basic primitives? Obviously, this is a question about the power of models, specifically about relaxing the atomicity of read-write shared-memory registers. This problem was raised

by Lamport [24] in 1986, and quite a few researchers have addressed it since then. Here we show that the nonblocking version of the problem can be answered trivially.

The primitive Lamport considers is a single-writer/single-reader safe-bit. A **safe-bit** is an object to which the writer writes a bit and from which the reader can read provided that the interval of operation execution of reading does not overlap with writing.

We now show how a stage of the iterated immediate-snapshot model can be nonblocking implemented by safe-bits. We assume an unlimited number of safe-bits, all initialized to 0, per pair of processors. To write a value, a processor writes it in unary, starting at a location known to the reader. To read, a processor counts the number of 1s it encounters until it meets a 0.

We observe that the recursive program for one-shot immediate snapshot itself consists of stages. At most k processors arrive at the immediate-snapshot stage called k (notice that these stages run from n down to 1). All we need is for at least one out of the k processors to remain trapped at stage k. The principle used to achieve this is the flag principle, namely, if k processors raise a flag and then count the number of flags raised, at least one processor will see k flags. For this principle to hold, we do not need atomic variables. Processors first write to all the other processors that they are at stage k and only then start the process of reading, so the last processor to finish writing will always read k flags.

Given that a processor writes its input to all other processors before it starts the immediate-snapshot stage, when one processor encounters another in the immediate snapshot and needs to know the other processor's input, the input is already written.

This shows that any task solvable by the most powerful read-write objects can be solved by single-writer/single-reader safe-bits. Can one nonblocking emulate by safe-bits any object that can be read-write emulated? The ingenious transformations cited in the introduction have made it clear that the answer to this question is "yes." Nonetheless, the field still awaits a theory that will allow us to deal with wait-free emulation in the same vein as we have dealt with nonblocking emulation here.

Geometric Protocols

What if we draw a one-dimensional immediate snapshot complex on a plan and repeat subdivide it forever? We can then argue that each point in our drawing corresponds to an infinite run with respect to views that are outputed. Obviously such a construction can be done for any dimension. We obtain an embedding of the space of runs in the Euclidean unit simplex. Such an embedding was the quest that eluded the authors of [29]. An embedding gives rise to "geometric-protocols."

Consider the problem of 2-processors election when we are given that the infinite symmetric run will not happen. One may work out an explicit protocol (which is not trivial). Geometrically, we take an embedding, and a processor waits until its view is completely on one side of the symmetric run. It then decides according to the solo execution that side contains.

48.4 Synchronous Systems

Shared-Memory Model

In retrospect, given the iterated-snapshots model in which conceptually processors go in lock-step, and its equivalence to the asynchronous shared memory, it is hard to understand the dichotomy between synchrony and asynchrony. In one case, the synchronous, we consider processors that are not completely coordinated because of various type of failures. In the other case, the asynchronous, processors are not coordinated because of speed mismatch. Why is this distinction of such fundamental importance? In fact, we argue that it is not. To exemplify this we show how one can derive a result in one model from a result in the other.

We consider the SWMR shared-memory model where its computation evolves in rounds. In other words, all communication events in the different processors and communication objects proceed in lockstep. At

the beginning of a round, processors write their cells and then read all cells in any order. Anything written in the round by a processor is read by all processors. This model may be viewed as a simple variant of the parallel RAM (PRAM) model, in which processors read all cells, rather than just one cell, in a single round.

With no failures, asynchronous systems can emulate synchronous ones. What is the essential difference between synchronous and asynchronous systems when failures are involved? A 1-resilient asynchronous system can "almost" emulate a round of the synchronous system. It falls short because one processor (say, if we are dealing with the synchronous shared-memory system below) misbehaves—but this processor does not really misbehave: what happens is that some of the other processors miss it because they read its cell too early, and they cannot wait on it, since one processor is allowed to fail-stop. If we call this processor faulty, we have the situation where we have at most one fault in a round, but the fault will shift all over the place and any processor may look faulty sooner or later. In contrast, in a synchronous system, a faulty behavior is attributed to some processor, and we always assume that the number of faults is less than n, the number of processors in the system.

We now introduce the possibility of faults into the synchronous system, and we will examine three types of faults. The first type is the analogue of fail-stop: a processor dies in the middle of writing, and the last value it wrote may have been read by some processors and not read by the others.

The algorithm to achieve consensus is quite easy. Each processor writes an agreement proposal at each round and then, at the next round, proposes the plurality of values it has read (with a tie-breaking rule common to all processors). At the first round, processors propose their initial value. After $t + 1$ rounds, where t is an upper bound on the number of faults, a processor decides on the value it would have proposed at round $t + 2$. The algorithm works because, by the pigeon principle, there is a **clean round,** namely, a round at which no processor dies. At the clean round, all processors take the plurality of common information, which results in a unanimous proposal at the next round. This proposal will be sustained to the end. Various techniques exist that can lead to early termination in executions with fewer faults than expected.

The next type of fault is omission [27]: a processor may be resurrected from fail-stop and continue to behave correctly, only to later die again to be perhaps resurrected again, and so on.

We reduce omission failure to fail-stop. A processor P_i that fails to read a value of P_j at round k goes into a protocol introduced in the next section to commit P_j as faulty. Such a protocol has the property that if P_i succeeds, then all processors will consider P_j faulty in the next round. Otherwise P_i obtain a value for P_j. We see that if a correct processor P_i fails to read a value from P_j, all processors will stop reading cell C_j at the next round, which is exactly the effective behavior of a fail-stop processor.

The last type of fault is Byzantine. We assume $n = 3f + 1$. Here, in addition to omitting a value, a processor might not obey the protocol and instead write anything (but not different values ($\neq \perp$) to different processors). We want to achieve the effect of true shared memory, in which if a correct processor reads a value, then all other processors can read the same value after the correct processor has done so.

We encountered the same difficulty in the asynchronous case, with the difference that in the asynchronous case things happen "eventually." If we adopt the same algorithm, whatever must happen eventually (asynchronous case) translates into happening in some finite number of rounds (synchronous case). Moreover, if something that is supposed to happen for a correct processor in the asynchronous case does not happen for a processor in the synchronous case within a prescribed number of rounds, the other processors infer that the processor involved is faulty and announce it as faulty. Thus, the asynchronous algorithm for reading a value which we gave in the previous section translates to (code for processor P_i):

1. Round 1: $v := read(cell)$.
2. Round 2: Write v, read values for v from all nonfaulty processors.
3. Round 3: If $2f + 1$ for value $v \neq \perp$ in Round 2, write $confirm(v)$; else, write $v := faulty$, read cells.

4. Round 4: If $2f + 1$ for v in Round 3 or if $f + 1$ for $confirm(v)$, write $confirm(v)$; else, $v := faulty$. If $2f + 1$ $confirm(v)$ up to now, then $accept(v)$. If $v = faulty$ and $accept(v)$, consider the processor that wrote v faulty.

A little thought shows that the "eventual" of the asynchronous case translates into a single round in the synchronous case. If a value is accepted, it will be accepted by all correct processors in the next round. If a value is not accepted by the end of the third round, then all correct processors propose $v := faulty$. In any case, at the end of the fourth round, either a real value or $v = faulty$ or both will be accepted. After a processor P_i is accepted as faulty, at the next round (which may be at the next phase) all processors will accept it as faulty and will ignore it. Thus, ignoring for the moment the problem that a faulty processor may write incorrect values, we have achieved the effect of write-once shared memory with fail-stop.

To deal with the issue of a faulty processor writing values a correct processor would not write, we can check on previous writes and see whether the processor observes its protocol. Here we do not face the difficulty we faced in the asynchronous case: a processor cannot avoid reading a value of a correct processor, and a processor may or may not (either is possible) read a value from a processor that failed. Nevertheless, a processor's value may be inconsistent with the values of correct processors. We notice that correct processors are consistent among themselves. Processors then can draw a "conflict" graph and, by eliminating edges to remove conflicts, declare the corresponding processors faulty. Since an edge contains at least one faulty processor, we can afford the cost of failing a correct processor.

We now argue a lower bound of $f + 1$ rounds for consensus for any of our three failure modes. It suffices to prove this bound for fail-stop failure, because fail-stop is a special case of omission failure and of Byzantine failure. Suppose an $(m < t + 1)$-rounds consensus algorithm exists for the fail-stop type of failure. We emulate the synchronous system by a 1-resilient asynchronous system in iterated atomic-snapshot shared memory. At a round, there is a unique single processor whose value other processors may fail to read. We consider such a processor to be faulty. We have seen that, without any extra cost, if a correct processor fails to read a value of processor P_j, then all correct processors will consider P_j faulty in the next round, which amounts to fail-stop. Thus, the asynchronous system emulates m rounds of the synchronous system. At each simulated round, there is at most one fault, for a total of at most f faults. This means that the emulated algorithm should result in consensus in the 1-resilient asynchronous system, which is impossible.

We can apply the same logic for set consensus and show that with f faults and k-set consensus, we need at least $\lfloor f/k \rfloor + 1$ rounds. This involves simulating the algorithm in a k-resilient asynchronous atomic-snapshot shared-memory system in which k-set consensus is impossible. (The first algorithm in this section automatically solves k-set consensus in the prescribed number of rounds; see [11]).

Notice that the above impossibility result for the asynchronous model translated into a lower bound on round complexity in the synchronous model. We now present the failure-detector framework, a framework in which speed mismatch is "transformed" into failure, and thus unifying synchrony and asynchrony.

48.5 Failure Detectors: A Bridge Between Synchronous and Asynchronous Systems

In the preceding section, we have seen that a 1-resilient asynchronous system looks like a synchronous system in which at each round a single but arbitrary processor may fail. Thus, synchronous systems can achieve consensus because in this setting when one processor considers another processor faulty, it is indeed the case, and one processor is never faulty. In the asynchronous setting, the processor considered faulty in a round is not faulty, only slow; we make a mistake in declaring it faulty.

Chandra and Toueg [13] have investigated the power of systems via the properties of their of augmenting subsystem called **failure-detector** (FD) that issues faulty declarations. The most interesting FDs are those with properties called S and $\diamond S$.

FD S can be described as follows:

- All processors that take finitely many steps in the underlying computation (faulty processors) will eventually be declared forever faulty by all the local FDs of processors that took infinitely many steps (correct processors), and
- Some correct processor is never declared faulty by the local FDs of correct processors.

In $\diamond S$, these properties hold eventually. Chandra, Hadzilacos, and Toueg [12] then showed that of all the FDs that provide for consensus, $\diamond S$ is the weakest: any FD that provides for consensus can nonblocking emulate $\diamond S$.

Thus, in a sense, if we take a system with timing constraints and call it synchronous if the constraints provide for consensus, then we have an alternative way of checking on whether the constraints provide for nonblocking emulation of $\diamond S$.

In many systems, this alternative technique is natural. For instance, consider a system in which the pending operations stay pending for at most some bounded but otherwise unknown real time and the processors can delay themselves for times that grow longer and longer without bound (of course, a delay operation is not pending until the prescribed delay time has elapsed). It is easy to see that this system can nonblocking implement $\diamond S$ and, as a result, achieve consensus. Thus, we have a variety of consensus algorithms from which to choose. In fact, we can transform any synchronous consensus algorithm for omission to an algorithm for $\diamond S$. The transformation proceeds in two stages. We first transform a synchronous omission algorithm A to an algorithm B for consensus in S. We accomplish this by nonblocking emulating A round by round, where each processor waits on the others until either the expected value is observed or the processor waited on is declared faulty by the FD. Such failure can be considered omission failure. Since in S, one of the processors is never declared faulty, we get that the number of omission faults is less than n, and the emulation results in consensus.

To transform a consensus algorithm A in S to a consensus algorithm B for $\diamond S$, we use a layer algorithm called Eventual. We assume A is safe in the sense that when running A in $\diamond S$, the liveness conditions that result in consensus are weakened but the safety conditions that make processors in A over S agree or preserve validity (output has to be one of the inputs) are maintained.

Algorithm Eventual has the property that a processor's output consists of either committing to some input value or adopting one.

1. If all processors start with the same input, then all commit to that input.
2. If one processor commits to an input value, then all other processors commit to or adopt that value.

The consensus algorithm B for shared memory with $\diamond S$ is to run alternately with A followed by Eventual. The output of A is the input to Eventual, and the output of Eventual is the input to A. When a processor commits to a value in Eventual, it outputs it as the output of B. Algorithm Eventual is simple. Processors repeatedly post their input values and take snapshots until the number of postings exceeds n. If a processor observes only a single value, it commits to that value; otherwise, it adopts a plurality value.

The notion of FDs is very appealing. It unifies synchronous and asynchronous systems. In the FD framework, all systems are of the same type but each system possesses FDs with distinctive properties. Research into the question of how this unified view can be exploited in distributed computing—for example, it might enable understanding of the "topology" of FDs—has hardly begun.

Another direction of possible research is to enrich the semantics of failure-detector. If we take the iterated snapshot system we may consider attaching a separate failure detector subsystem to each layer. A processor which is "late" arriving to a layer is declared faulty by that layer subsystem. We may now investigate the power of such system as a function of the properties of their failure detectors. We can for instance model a **t-resilient system** as a system in which at most t processors will be declared faulty at a layer. When one

considers such a system, the dichotomy of systems between synchronous and asynchronous is completely blurred. The traditional failure detector of Chandra and Toueg cannot capture such a property.

48.6 Research Issues and Summary

In this chapter, we have considered a body of fairly recent research in distributed computing whose results are related and draw upon one another. We have argued that a unified view of these results derives from the synergy between the application of results from topology and the use of the interpretive power of distributed computing to derive transformations that make topology apply.

Many interesting and important questions, aside from matters of complexity, remain. The most practical of these questions concern computations that are amenable to algorithms whose complexity is a function of the concurrency rather than the size of the system. Such algorithms are usually referred to as **fast.** Examples of tasks that can be solved by application of fast algorithms are numerous, but a fundamental understanding of exactly what, why, and how computations are amenable to such algorithms is lacking. Extension of the theory presented in this chapter to nonterminating tasks and to long-lived objects is next on the list of questions.

48.7 Defining Terms

t-Resilient system: A system in which at most t processors are faulty.

0-Simplex: A singleton set. item[1-Dimensional subdivided simplex:] An embedding of 1-simplex that is partitioned into 1-simplexes (a path).

1-Simplex: A set consisting of two elements.

Atomic snapshot: An atomic read operation that returns the entire shared memory.

Chain-of-runs: A sequence of runs in which two consecutive runs are indistinguishable to a processor.

Cleanround: A round in which no new faulty behavior is exhibited.

Communication-closed-layers: A distributed program partitioned to layers that communicate remotely only among themselves and communicate locally unidirectionally.

Complete problem: A problem in a model that characterizes the set of all other problems in the model in the sense that they are reducible to it.

Complex: A set of simplexes that is closed under subset.

Consensus problem: A decision task in which all processors agree on a single input value.

Election: A consensus over the IDs as inputs.

Failure-detector: An oracle that updates a processor on the operational status of the rest of the processors.

Fast solution: A solution to a problem whose complexity depends on the number of participating processors rather than the size of the entire system.

Faulty processor: A processor whose view is output finitely many times in a run.

Full-information protocol: A protocol that induces the finest partition of the set of runs, compared to any other protocol in the model.

Immediate snapshots: A restriction of the atomic snapshots that achieves a certain closure property that atomic snapshots do not have.

Link (of a simplex in a complex): The set of all simplexes in a complex that are contained in a simplex with the given simplex.

Nonblocking (emulation): An emulation that may increase the set of faulty processors.

Oblivious (to a parameter): Not a function of that parameter.

Outputs: Map over views that are eventually known to all nonfaulty processors.

Participating (set of processors): Processors in a run that are not sleeping.

Processor: A sequential piece of code.

Protocol: A set of processors whose codes refer to common communication objects.

Run: An infinite sequence of global "instantaneous-description" of a system, such that one element is the preceding one after the application of a pending operation.

Safe-bit: A single-writer single-reader register bit whose read is defined and returns the last value written only if does not overlap a write operation to the register.

Sleeping (in a run): A processor whose state in a run does not change.

Solo-execution: A run in which only a single processor is not sleeping.

Task: A relation from inputs and set of participating processors to outputs.

View: A set of runs compatible with a processor's local state.

Wait-free (solution): A solution to a nontermination problem in which a processor that is not faulty outputs infinitely many output values.

Acknowledgment

I am grateful to Hagit Attiya for detailed illuminating comments on an earlier version of this chapter.

References

[1] Afek, Y., Attiya, H., Dolev, D., Gafni, E., Merrit, M., and Shavit, N., Atomic snapshots of shared memory. In *Proceedings of the 9th ACM Symposium on Principles of Distributed Computing,* 1–13, 1990.

[2] Attiya, H., Bar-Noy, A., and Dolev, D., Sharing memory robustly in message-passing systems. *Journal of the ACM,* 42(1), 124–142, Jan. 1995.

[3] Afek, Y., Greenberg, D.S., Merritt, M., and Taubenfeld, G., Computing with faulty shared objects. *Journal of the Association of the Computing Machinery,* 42, 1231–1274, 1995.

[4] Attiya, H., Lynch, N., and Shavit, N., Are wait-free algorithms fast? *Journal of the ACM,* 41(4), 725–763, Jul. 1994.

[5] Attiya, H. and Rajsbaum, The combinatorial structure of wait-free solvable tasks. In *WDAG: International Workshop on Distributed Algorithms,* Springer-Verlag, 1996.

[6] Attiya, H., Distributed computing theory. In *Handbook of Parallel and Distributed Computing,* A.Y. Zomaya, Ed., McGraw-Hill, New York, 1995.

[7] Borowsky, E. and Gafni, E., Immediate atomic snapshots and fast renaming. In *Proceedings of the 12th ACM Symposium on Principles of Distributed Computing,* 41–51, 1993.

[8] Borowsky, E. and Gafni, E., A simple algorithmically reason characterization of wait-free computations. In *Proceedings of the 16th ACM Symposium on Principles of Distributed Computing,* 189–198, 1997.

[9] Biran, O., Moran, S., and Zaks, S., A combinatorial characterization of the distributed tasks which are solvable in the presence of one faulty processor. In *Proceedings of the 7th ACM Symposium on Principles of Distributed Computing,* 263–275, 1988.

[10] Bracha, G., Asynchronous byzantine agreement protocols. *Information and Computation,* 75(2), 130–143, Nov. 1987.

[11] Chaudhuri, S., Herlihy, M., Lynch, N.A., and Tuttle, M.R., A tight lower bound for k-set agreement. In *34th Annual Symposium on Foundations of Computer Science,* 206–215, Palo Alto, CA, IEEE, 3–5 Nov. 1993.
[12] Chandra, T.D., Hadzilacos, V., and Toueg, S., The weakest failure detector for solving consensus. *Journal of the ACM,* 43(4), 685–722, Jul. 1996.
[13] Chandra, T.D. and Toueg, S., Unreliable failure detectors for reliable distributed systems. *Journal of the ACM,* 43(2), 225–267, Mar. 1996.
[14] Elrad, T.E. and Francez, N., Decomposition of distributed programs into communication closed layers. *Science of Computer Programming,* 2(3), 1982.
[15] Fischer, M.J. and Lynch, N.A., A lower bound on the time to assure interactive consistency. *Information Processing Letters,* 14(4), 183–186, 1982.
[16] Fischer, M., Lynch, N., and Paterson, M., Impossibility of distributed consensus with one faulty process. *Journal of the ACM,* 32(2), 374–382, 1985.
[17] Gafni, E. and Koutsoupias, E.,. 3-processor tasks are undecidable. In *Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing,* 271, ACM, Aug. 1995.
[18] Gray, J.N., Notes on data base operating systems. In *LNCS, Operating Systems, an Advanced Course,* Bayer, Graham, Seegmuller Eds., Vol. 60, Springer Verlag, Heidelberg, 1978.
[19] Herlihy, M.P., Wait-free synchronization. *ACM Transactions on Programming Languages and Systems,* 11(1), 124–149, Jan. 1991. Supersedes 1988 PODC version.
[20] Halpern, J.Y. and Moses, Y., Knowledge and common knowledge in a distributed environment. *Journal of the ACM,* 37(3), 549–587, Jul. 1990.
[21] Herlihy, M. and Rajsbaum, S., Algebraic topology and distributed computing—A primer. *Lecture Notes in Computer Science,* 1000, 203, 1995.
[22] Herlihy, M. and Shavit, N., The asynchronous computability theorem for t-resilient tasks. In *Proceedings of the 25th ACM Symposium on the Theory of Computing,* 111–120, 1993.
[23] Herlihy, M. and Shavit, N., A simple constructive computability theorem for wait-free computation. In *Proceedings of the 26th ACM Symposium on the Theory of Computing,* 1994.
[24] Lamport, L., On interprocess communication.
Distributed Computing, 1, 77–101, 1986.
[25] Lamport, L. and Lynch, N., Distributed computing: Models and methods. In *Handbook of Theoretical Computer Science,* J. van Leewen, Ed., Vol. B: *Formal Models and Semantics,* chapter 19, 1157–1199, MIT Press, New York, 1990.
[26] Lynch, N., *Distributed Algorithms,* Morgan Kaufmann, San Francisco, 1996.
[27] Neiger, G. and Toueg, S., Automatically increasing the fault-tolerance of distributed algorithms. *Journal of Algorithms,* 11(3), 374–419, Sept. 1990.
[28] Spanier, E.H., *Algebraic Topology*. Springer-Verlag, New York, 1966.
[29] Saks, M. and Zaharoglou, F., Wait-free k-set agreement is impossible: The topology of public knowledge. In *Proceedings of the 26th ACM Symposium on the Theory of Computing,* 101–110, 1993.

Further Information

Current research on the theoretical aspects of distributed computing is reported in the proceedings of the annual ACM Symposium on Principles of Distributed Computing (PODC) and the annual International Workshop on Distributed Algorithms on Graphs (WDAG). Relatively recent books and surveys are [6, 21, 26].