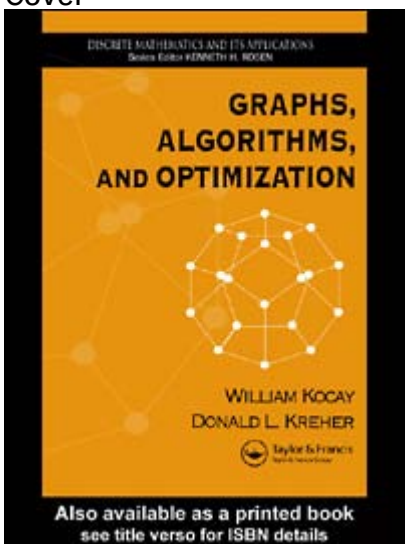
Cover



title:	Graphs, Algorithms, and Optimization Discrete Mathematics and Its Applications
author:	Kocay, William.; Kreher, Donald L.
publisher:	CRC Press
isbn10 | asin:	0203489055
print isbn13:	9780203620892
ebook isbn13:	9780203489055
language:	English
subject	Graph algorithms.
publication date:	2005
lcc:	QA166.245.K63 2005eb
ddc:	511/.5
subject:	Graph algorithms.

cover

Page i
DISCRETE MATHEMATICS AND ITS APPLICATIONS
Series Editor KENNETH H.ROSEN
GRAPHS, ALGORITHMS, AND OPTIMIZATION

page_i

Page ii
DISCRETE MATHEMATICS AND ITS APPLICATIONS
Series Editor
Kenneth H.Rosen, Ph.D.
Charles J.Colbourn and Jeffrey H.Dinitz, The CRC Handbook of Combinatorial Designs
Charalambos A.Charalambides, Enumerative Combinatorics
Steven Furino, Ying Miao, and Jianxing Yin, Frames and Resolvable Designs: Uses, Constructions, and Existence
Randy Goldberg and Lance Riek, A Practical Handbook of Speech Coders
Jacob E.Goodman and Joseph O'Rourke, Handbook of Discrete and Computational Geometry, Second Edition
Jonathan Gross and Jay Yellen, Graph Theory and Its Applications
Jonathan Gross and Jay Yellen, Handbook of Graph Theory
Darrel R.Hankerson, Greg A.Harris, and Peter D.Johnson, Introduction to Information Theory and Data Compression, Second Edition
Daryl D.Harms, Miroslav Kraetzl, Charles J.Colbourn, and John S.Devitt, Network Reliability: Experiments with a Symbolic Algebra Environment
David M.Jackson and Terry I.Visentin, An Atlas of Smaller Maps in Orientable and Nonorientable Surfaces
Richard E.Klima, Ernest Stitzinger, and Neil P.Sigmon, Abstract Algebra Applications with Maple
Patrick Knupp and Kambiz Salari, Verification of Computer Codes in Computational Science and Engineering
Donald L.Kreher and Douglas R.Stinson, Combinatorial Algorithms: Generation Enumeration and Search

Charles C.Lindner and Christopher A.Rodgers, Design Theory
Alfred J.Menezes, Paul C.van Oorschot, and Scott A.Vanstone, Handbook of Applied Cryptography
Richard A.Mollin, Algebraic Number Theory
Richard A.Mollin, Fundamental Number Theory with Applications
Richard A.Mollin, An Introduction to Cryptography
Richard A.Mollin, Quadratics

Page iii
Continued Titles
Richard A.Mollin, RSA and Public-Key Cryptography
Kenneth H.Rosen, Handbook of Discrete and Combinatorial Mathematics
Douglas R.Shier and K.T.Wallenius, Applied Mathematical Modeling: A Multidisciplinary Approach
Douglas R.Stinson, Cryptography: Theory and Practice, Second Edition
Roberto Togneri and Christopher J.deSilva, Fundamentals of Information Theory and Coding Design
Lawrence C.Washington, Elliptic Curves: Number Theory and Cryptography
Kun-Mao Chao and Bang Ye Wu, Spanning Trees and Optimization Problems
Juergen Bierbrauer, Introduction to Coding Theory
William Kocay and Donald L.Kreher, Graphs, Algorithms, and Optimization

Page iv
This page intentionally left blank.

Page v
DISCRETE MATHEMATICS AND ITS APPLICATIONS
Series Editor KENNETH H.ROSEN
GRAPHS, ALGORITHMS, AND OPTIMIZATION
WILLIAM KOCAY
DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF MANITOBA
DONALD L.KREHER
DEPARTMENT OF MATHEMATICAL SCIENCES
MICHIGAN TECHNOLOGICAL UNIVERSITY

CHAPMAN & HALL/CRC
A CRC Press Company
Boca Raton London NewYork Washington, D.C.

Page vi
This edition published in the Taylor & Francis e-Library, 2005.

Library of Congress Cataloging-in-Publication Data
Kocay, William.
Graphs, algorithms, and optimization/William Kocay, Donald L.Kreher.
p. cm.—(Discrete mathematics and its applications)
Includes bibliographical references and index.
ISBN 1-58488-396-0 (alk. paper)
1. Graph algorithms. I. Kreher, Donald L. II. Title. III. Series.
QA166.245.K63 2004
511′.5–dc22 2004056153

ISBN 0-203-48905-5 Master e-book ISBN
ISBN 0-203-62089-5 (OEB Format)
International Standard Book Number 1-58488-396-0 (Print Edition)
Library of Congress Card Number 2004056153

Page vii
The authors would like to take this opportunity to express their appreciation and gratitude to the following people who have had a very significant effect on their mathematical development:
Adrian Bondy, Earl Kramer, Spyros Magliveras, Ron Read, and Ralph Stanton.
This book is dedicated to the memory of
William T.Tutte, (1917–2002)
“the greatest of the graphmen”

Page viii
This page intentionally left blank.

Page ix
Preface
Our objective in writing this book is to present the theory of graphs from an algorithmic viewpoint. We present the graph theory in a rigorous, but informal style and cover most of the main areas of graph theory. The ideas of surface topology are presented from an intuitive point of view. We have also included a discussion on linear programming that emphasizes problems in graph theory. The text is suitable for students in computer science or mathematics programs.
Graph theory is a rich source of problems and techniques for programming and data structure development, as well as for the theory of computing, including NP-completeness and polynomial reduction.
This book could be used a textbook for a third or fourth year course on graph algorithms which contains a programming content, or for a more advanced course at the fourth year or graduate level. It could be used in a course in which the programming language is any major programming language (e.g., C, C++, Java). The algorithms are presented in a generic style and are not dependent on any particular programming language.
The text could also be used for a sequence of courses like “Graph Algorithms I” and “Graph Algorithms II”. The courses offered would depend on the selection of chapters included. A typical course will begin with Chapters 1, 2, 3, and 4. At this point, a number of options are available.
A possible first course would consist of Chapters 1, 2, 3, 4, 6, 8, 9, 10, 11, and 12, and a first course stressing optimization would consist of Chapters 1, 2, 3, 4, 8, 9, 10, 14, 15, and 16. Experience indicates that the students consider these substantial courses. One or two chapters could be omitted for a lighter course.
We would like to thank the many people who provided encouragement while we wrote this book, pointed out typos and errors, and gave useful suggestions. In particular, we would like to convey our thanks to Ben Li and John van Rees of the University of Manitoba for proofreading some chapters.
William Kocay
Donald L.Kreher

Page x

This page intentionally left blank.

Page xi

William Kocay obtained his Ph.D. in Combinatorics and Optimization from the University of Waterloo in 1979. He is currently a member of the Computer Science Department, and an adjunct member of the Mathematics Department, at the University of Manitoba, and a member of St. Paul's College, a college affiliated with the University of Manitoba. He has published numerous research papers, mostly in graph theory and algorithms for graphs. He was managing editor of the mathematics journal *Ars Combinatoria* from 1988 to 1997. He is currently on the editorial board of that journal. He has had extensive experience developing software for graph theory and related mathematical structures.

Donald L.Kreher obtained his Ph.D. from the University of Nebraska in 1984. He has held academic positions at Rochester Institute of Technology and the University of Wyoming. He is currently a University Professor of Mathematical Sciences at Michigan Technological University, where he teaches and conducts research in combinatorics and combinatorial algorithms. He has published numerous research papers and is a co-author of the internationally acclaimed text "Combinatorial Algorithms: Generation Enumeration and Search", CRC Press, 1999. He serves on the editorial boards of two journals.

Professor Kreher is the sole recipient of the 1995 Marshall Hall Medal, awarded by the Institute of Combinatorics and its Applications.

Page xii

This page intentionally left blank.

Page xiii

Contents

1 Graphs and Their Complements **1**
1.1 Introduction 1
Exercises 6
1.2 Degree sequences 8
Exercises 17
1.3 Analysis 18
Exercises 21
1.4 Notes 22
2 Paths and Walks **23**
2.1 Introduction 23
2.2 Complexity 27
Exercises 27
2.3 Walks 28
Exercises 29
2.4 The shortest-path problem 30
2.5 Weighted graphs and Dijkstra's algorithm 33
Exercises 36
2.6 Data structures 37
2.7 Floyd's algorithm 42
Exercises 44
2.8 Notes 45
3 Some Special Classes of Graphs **47**
3.1 Bipartite graphs 47
Exercises 49
3.2 Line graphs 50
Exercises 51
3.3 Moore graphs 52
Exercises 57

Page xiv

3.4 Euler tours 58

3.4.1 An Euler tour algorithm 59
Exercises 62
3.5 Notes 62
4 Trees and Cycles 63
4.1 Introduction 63
Exercises 64
4.2 Fundamental cycles 65
Exercises 65
4.3 Co-trees and bonds 67
Exercises 69
4.4 Spanning tree algorithms 70
4.4.1 Prim's algorithm 72
Data structures 74
Exercises 75
4.4.2 Kruskal's algorithm 76
Data structures and complexity 77
4.4.3 The Cheriton-Tarjan algorithm 78
Exercises 79
4.4.4 Leftist binary trees 79
Exercises 86
4.5 Notes 87
5 The Structure of Trees 89
5.1 Introduction 89
5.2 Non-rooted trees 90
Exercises 92
5.3 Read's tree encoding algorithm 92
5.3.1 The decoding algorithm 95
Exercises 96
5.4 Generating rooted trees 97
Exercises 105
5.5 Generating non-rooted trees 105
Exercises 106
5.6 Prüfer sequences 106
5.7 Spanning trees 109
5.8 The matrix-tree theorem 111
Exercises 116
5.9 Notes 117

Page xv

6 Connectivity 119
6.1 Introduction 119
Exercises 121
6.2 Blocks 122
6.3 Finding the blocks of a graph 125
Exercises 128
6.4 The depth-first search 128
6.4.1 Complexity 135
Exercises 136
6.5 Notes 137
7 Alternating Paths and Matchings 139
7.1 Introduction 139
Exercises 143
7.2 The Hungarian algorithm 143
7.2.1 Complexity 147
Exercises 148
7.3 Perfect matchings and 1-factorizations 148
Exercises 151
7.4 The subgraph problem 152
7.5 Coverings in bipartite graphs 154

7.6 Tutte's theorem 155
Exercises 158
7.7 Notes 158
8 Network Flows 161
8.1 Introduction 161
8.2 The Ford-Fulkerson algorithm 165
Exercises 175
8.3 Matchings and flows 176
Exercises 177
8.4 Menger's theorems 178
Exercises 180
8.5 Disjoint paths and separating sets 180
Exercises 183
8.6 Notes 185
9 Hamilton Cycles 187
9.1 Introduction 187
Exercises 190
9.2 The crossover algorithm 191
9.2.1 Complexity 193
Exercises 196

9.3 The Hamilton closure 197
Exercises 199
9.4 The extended multi-path algorithm 200
9.4.1 Data structures for the segments 204
Exercises 204
9.5 Decision problems, NP-completeness 205
Exercises 213
9.6 The traveling salesman problem 214
Exercises 216
9.7 The ΔTSP 216
9.8 Christofides' algorithm 218
Exercises 220
9.9 Notes 221
10 Digraphs 223
10.1 Introduction 223
10.2 Activity graphs, critical paths 223
10.3 Topological order 225
Exercises 228
10.4 Strong components 229
Exercises 230
10.4.1 An application to fabrics 236
Exercises 236
10.5 Tournaments 238
Exercises 240
10.6 2-Satisfiability 240
Exercises 243
10.7 Notes 243
11 Graph Colorings 245
11.1 Introduction 245
11.1.1 Intersecting lines in the plane 247
Exercises 248
11.2 Cliques 249
11.3 Mycielski's construction 253
11.4 Critical graphs 254
Exercises 255
11.5 Chromatic polynomials 256
Exercises 258

11.6 Edge colorings 258
11.6.1 Complexity 268
Exercises 269

Page xvii

11.7 NP-completeness 269
11.8 Notes 274
12 Planar Graphs 275
12.1 Introduction 275
12.2 Jordan curves 276
12.3 Graph minors, subdivisions 277
Exercises 282
12.4 Euler's formula 282
12.5 Rotation systems 284
12.6 Dual graphs 286
12.7 Platonic solids, polyhedra 290
Exercises 291
12.8 Triangulations 292
12.9 The sphere 295
12.10 Whitney's theorem 297
12.11 Medial digraphs 300
Exercises 301
12.12 The 4-color problem 301
Exercises 305
12.13 Straight-line drawings 305
12.14 Kuratowski's theorem 309
Exercises 312
12.15 The Hopcroft-Tarjan algorithm 312
12.15.1 Bundles 316
12.15.2 Switching bundles 318
12.15.3 The general Hopcroft-Tarjan algorithm 321
12.16 Notes 325
13 Graphs and Surfaces 327
13.1 Introduction 327
13.2 Surfaces 329
13.2.1 Handles and crosscaps 336
13.2.2 The Euler characteristic and genus of a surface 337
Exercises 340
13.3 Graph embeddings, obstructions 341
13.4 Graphs on the torus 342
Exercises 349
13.4.1 Platonic maps on the torus 349
13.4.2 Drawing torus maps, triangulations 352
Exercises 357
13.5 Graphs on the projective plane 357

Page xviii

13.5.1 The facewidth 364
13.5.2 Double covers 368
Exercises 370
13.6 Embedding algorithms 372
Exercises 381
13.7 Heawood's map coloring theorem 382
Exercises 384
13.8 Notes 385
14 Linear Programming 387
14.1 Introduction 387
14.1.1 A simple example 387

14.1.2 Simple graphical example 389
14.1.3 Slack and surplus variables 391
Exercises 394
14.2 The simplex algorithm 395
14.2.1 Overview 395
14.2.2 Some notation 395
14.2.3 **Phase 0:** finding a basis solution 396
14.2.4 Obtaining a basis feasible solution 397
14.2.5 The tableau 398
14.2.6 **Phase 2:** improving a basis feasible solution 399
14.2.7 Unbounded solutions 403
14.2.8 Conditions for optimality 405
14.2.9 **Phase 1:** initial basis feasible solution 407
14.2.10 An example 411
14.3 Cycling 413
Exercises 415
14.4 Notes 416
15 The Primal-Dual Algorithm 417
15.1 Introduction 417
15.2 Alternate form of the primal and its dual 423
15.3 Geometric interpretation 424
15.3.1 Example 424
15.4 Complementary slackness 429
15.5 The dual of the shortest-path problem 430
Exercises 433
15.6 The primal-dual algorithm 434
15.6.1 Initial feasible solution 437
15.6.2 The shortest-path problem 440
15.6.3 Maximum flow 444

Page xix

Exercises 446
15.7 Notes 446
16 Discrete Linear Programming 449
16.1 Introduction 449
16.2 Backtracking 450
16.3 Branch and bound 453
Exercises 463
16.4 Unimodular matrices 465
Exercises 467
16.5 Notes 468

Bibliography 469
Index 477

Page xx
This page intentionally left blank.

Page 1
1
Graphs and Their Complements
1.1 Introduction
The diagram in Figure 1.1 illustrates a graph. It is called the graph of the cube. The edges of the geometric cube correspond to the line segments connecting the nodes in the graph, and the nodes correspond to the corners of the cube where the edges meet. They are the vertices of the cube.

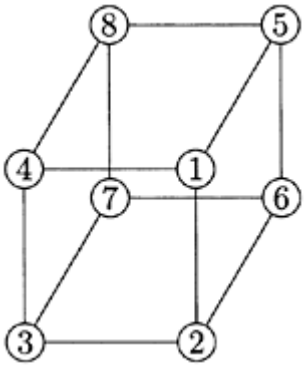


FIGURE 1.1
The graph of a cube

This diagram is drawn so as to resemble a cube, but if we were to rearrange it, as in Figure 1.2, it would still be the graph of the cube, although it would no longer look like a cube. Thus, a graph is a graphical representation of a relation in which edges connect pairs of vertices.

DEFINITION 1.1: A *simple graph G* consists of a *vertex set V(G)* and an *edge set E(G),* where each edge is a pair *{u,u}* of vertices *u,* $v \in V(G)$.

We denote the set of all pairs of a set *V* by $\binom{V}{2}$. Then $E(G) \subseteq \binom{V(G)}{2}$. In

---

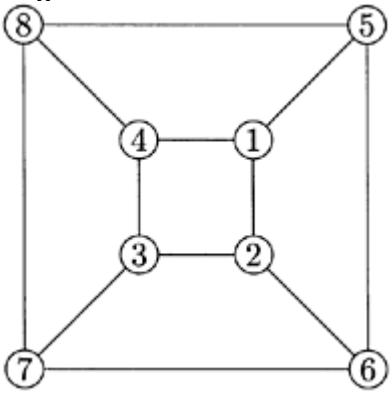


FIGURE 1.2
The graph of the cube

the example of the cube, *V(G)*={1, 2, 3, 4, 5, 6, 7, 8}, and *E(G)*={12, 23, 34, 14, 15, 26, 37, 48, 56, 67, 78, 58}, where we have used the shorthand notation *uu* to stand for the pair *{u,u}*. If *u,* $v \in V(G)$, then $u \longrightarrow v$ means that *u* is joined to *u* by an edge. We say that *u* and *u* are *adjacent*. We use this notation to remind us of the linked list data structure that we will use to store a graph in the computer. Similarly, $u \not\longrightarrow v$ means that *u* is not joined to u. We can also express these relations by writing $uv \in E(G)$ or $uv \notin E(G)$, respectively. Note that in a simple graph if $u \longrightarrow v$, then $v \longrightarrow u$. If *u* is adjacent to each of *u*1, *u*2,..., *u*k, then we write $u \longrightarrow \{u_1, u_2, \ldots, u_k\}$.

These graphs are called *simple* graphs because each pair *u, u* of vertices is joined by at most one edge. Sometimes we need to allow several edges to join the same pair of vertices. Such a graph is also called a *multigraph*. An edge can then no longer be defined as a pair of vertices, (or the multiple edges would not be distinct), but to each edge there still corresponds a pair *{u,u}*. We can express this formally by saying that a graph *G* consists of a vertex set *V(G),* an edge set *E(G),* and a correspondence $\psi : E(G) \rightarrow \binom{V(G)}{2}$. Given an edge $e \in E(G)$, *ψ(e)* is a pair *{u,u}* which are the *endpoints* of *e*. Different edges can then have the same endpoints. We shall use simple graphs most of the time, which is why we prefer the simpler definition, but many of the theorems and techniques will apply to multigraphs as well.

This definition can be further extended to graphs with *loops* as well. A loop is an edge in which both endpoints are equal. We can include this in the general definition of a graph by making the mapping $\psi : E(G) \rightarrow \binom{V(G)}{2} \cup V(G)$. An edge $e \in E(G)$ for which $\psi(e) = u \in V(G)$ defines a loop. Figure 1.2 shows a graph with multiple edges and loops. However, we shall use simple graphs most of the time, so that an

edge will be considered to be a pair of vertices.
The number of vertices of a graph *G* is denoted |*G*|. It is called the *order* of *G*.

Page 3

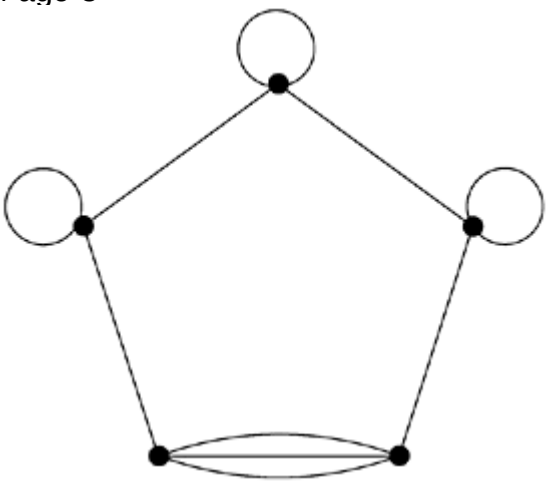


FIGURE 1.3
A multigraph

The number of edges is *ε(G)*. If *G* is simple, then obviously $\varepsilon(G) \leq \binom{|G|}{2}$, since $E(G) \subseteq \binom{V(G)}{2}$. We shall often use node or point as synonyms for vertex.

Many graphs have special names. The *complete graph Kn* is a simple graph with |*Kn*|=*n* and $\varepsilon = \binom{n}{2}$. The empty graph $\overline{K}_n$ is a graph with $|\overline{K}_n| = n$ and *ε*=0. $\overline{K}_n$ is the complement of *Kn*.

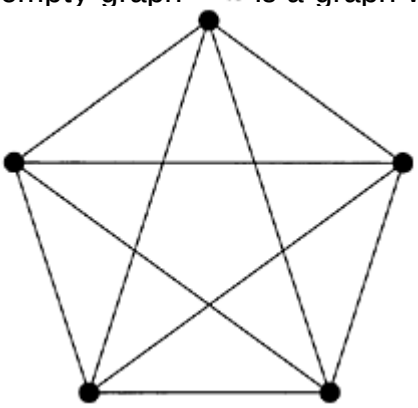


FIGURE 1.4
The complete graph *K*5

DEFINITION 1.2: Let G be a simple graph. The complement of *G* is $\overline{G}$, where $V(\overline{G}) = V(G)$ and $E(\overline{G}) = \binom{V(G)}{2} \setminus E(G)$.

$E(\overline{G})$ consists of all those pairs *uu* which are not edges of *G*. Thus, $uv \in E(\overline{G})$

Page 4

if and only if $uv \notin E(G)$.Figure 1.5 show a graph and its complement.

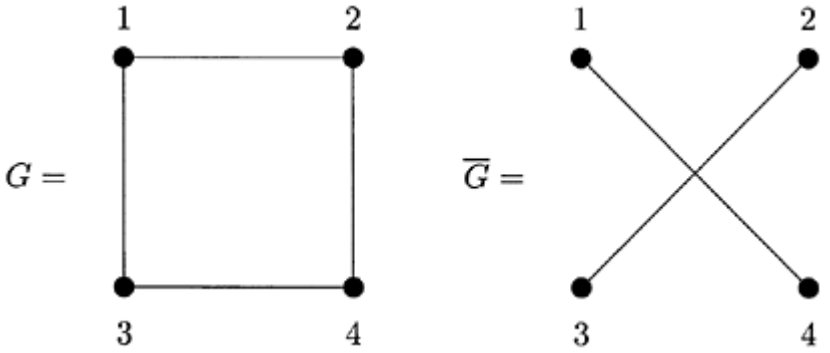


FIGURE 1.5
A graph and its complement

Figure 1.6 shows another graph and its complement. Notice that in this case, when $\overline{G}$ is redrawn, it looks identical to *G*.

In a certain sense, this *G* and $\overline{G}$ are the same graph. They are not equal, since $E(G) \neq E(\overline{G})$, but it is clear that they have the same structure. If two graphs have the same structure, then they can only differ in the names of the vertices. Therefore, we can rename the vertices of one to make it exactly equal to the other graph. In the example above, we can rename the vertices of *G* by the mapping *θ* given by

$$\begin{array}{c|ccccc} k: & 1 & 2 & 3 & 4 & 5 \\ \hline \theta(k): & 1 & 3 & 5 & 2 & 4 \end{array},$$

then *θ(G)* would equal $\overline{G}$. This kind of equivalence of graphs is known as *isomorphism*. Observe that a one-to-one mapping *θ* of the vertices of a graph *G* can be extended to a mapping of the edges of *G* by defining *θ({u,u})={θ(u),θ(u)}*.

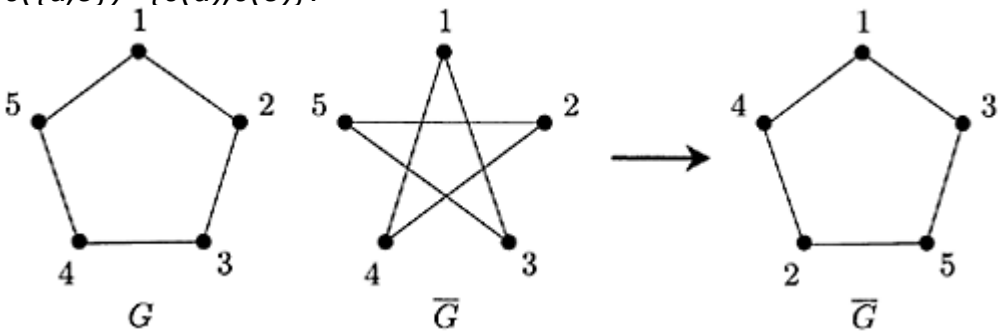


FIGURE 1.6
Another graph and its complement

Page 5

DEFINITION 1.3: Let *G* and *H* be simple graphs. *G* and *H* are *isomorphic* if there is a one-to-one correspondence *θ*: *V(G)→V(H)* such that *θ(E(G))= E(H)*, where $\theta(E(G)) = \{\theta(uv) : uv \in E(G)\}$.

We write $G \cong H$ to denote isomorphism. If $G \cong H$, then $uv \in E(G)$ if and only if $\theta(uv) \in E(H)$. One way to determine whether $G \cong H$ is to try and redraw *G* so as to make it look identical to *H*. We can then read off the mapping *θ* from the diagram. However, this is limited to small graphs. For example, the two graphs *G* and *H* shown in Figure 1.7 are isomorphic, since the drawing of *G* can be transformed into *H* by first moving vertex 2 to the bottom of the diagram, and then moving vertex 5 to the top. Comparing the two diagrams then gives the mapping

$$\begin{array}{c|cccccc} k: & 1 & 2 & 3 & 4 & 5 & 6 \\ \hline \theta(k): & 6 & 4 & 2 & 5 & 1 & 3 \end{array}$$

as an isomorphism.

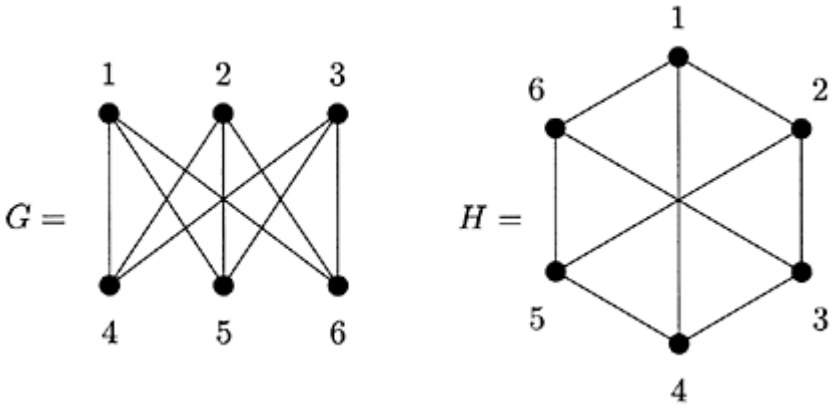


FIGURE 1.7
Two isomorphic graphs

It is usually more difficult to determine when two graphs *G* and *H* are not isomorphic than to find an isomorphism when they are isomorphic. One way is to find a portion of *G* that cannot be part of *H*. For example, the graph H of Figure 1.7 is not isomorphic to the graph of the prism, which is illustrated in Figure 1.8, because the prism contains a triangle, whereas H has no triangle. A *subgraph* of a graph *G* is a graph *X* such that $V(X) \subseteq V(G)$ and $E(X) \subseteq E(G)$. If $\theta: G \rightarrow H$ is a possible isomorphism, then $\theta(X)$ will be a subgraph of *H* which is isomorphic to *X*. A subgraph *X* is an *induced subgraph* if for every $u, v \in V(X) \subseteq V(G), uv \in E(X)$ if and only if $uv \in E(G)$.

The *degree* of a vertex $u \in V(G)$ is DEG*(u),* the number of edges which contain *u*. If *k*=DEG*(u)* and $u \longrightarrow \{u_1, u_2, \ldots, u_k\}$, then $\theta(u) \longrightarrow$

Page 6

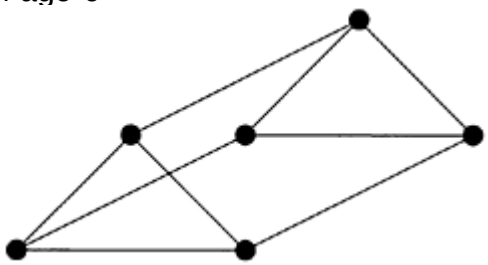


FIGURE 1.8
The graph of the prism

$\{\theta(u_1), \theta(u_2), \ldots, \theta(u_k)\}$, so that DEG*(u)*=DEG*(θ(u))*. Therefore a necessary condition for *G* and *H* to be isomorphic is that they have the same set of degrees. The examples of Figures 1.7 and 1.8 show that this is not a sufficient condition.

In Figure 1.6, we saw an example of a graph *G* that is isomorphic to its complement. There are many such graphs.

DEFINITION 1.4: A simple graph *G* is *self-complementary* if $G \cong \overline{G}$.

LEMMA 1.1 If G is a self-complementary graph, then |*G*|≡0 *or* 1 (mod 4).

PROOF If $G \cong \overline{G}$, then $\varepsilon(G) = \varepsilon(\overline{G})$. But $E(\overline{G}) = \binom{V(G)}{2} \setminus E(G)$, so that $\varepsilon(\overline{G}) = \binom{|G|}{2} - \varepsilon(G) = \varepsilon(G)$, so $\varepsilon(G) = \frac{1}{2}\binom{|G|}{2} = |G|(|G|-1)/4$. Now |*G*| and |*G*|−1 are consecutive integers, so that one of them is odd. Therefore |*G*|≡0 (mod 4) or |*G*|≡1 (mod 4).

So possible orders for self-complementary graphs are 4, 5, 8, 9, 12, 13,..., 4*k,* 4*k*+1,....

Exercises

1.1.1 The four graphs on three vertices in Figure 1.9 have 0, 1, 2, and 3 edges, respectively. Every graph on three vertices is isomorphic to one of these four. Thus, there are exactly four different isomorphism types of graph on three vertices.

Find all the different isomorphism types of graph on 4 vertices (there are 11 of them). *Hint:* Adding an edge to a graph with *ε*=*m,* gives a graph with *ε*=*m*+1. Every graph with *ε*=*m*+1 can be obtained in this way.

Table 1.1 shows the number of isomorphism types of graph up to 10 vertices.

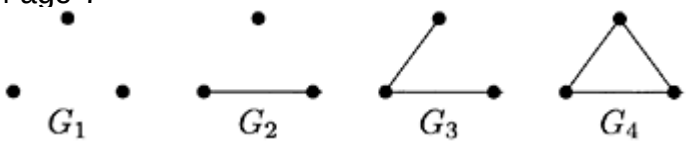


FIGURE 1.9
Four graphs on three vertices

TABLE 1.1
Graphs up to 10 vertices

n	No. graphs
1	1
2	2
3	4
4	11
5	34
6	156
7	1,044
8	12,346
9	247,688
10	12,005,188

1.1.2 Determine whether the two graphs shown in Figure 1.10 are isomorphic to each other or not. If they are isomorphic, find an explicit isomorphism.

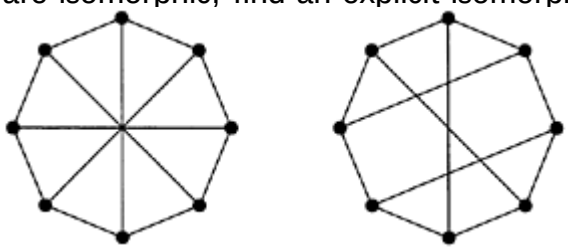


FIGURE 1.10
Two graphs on eight vertices

1.1.3 Determine whether the three graphs shown in Figure 1.11 are isomorphic to each other or not. If they are isomorphic, find explicit isomorphisms.

1.1.4 Find a self-complementary graph on four vertices.

1.1.5 Figure 1.6 illustrates a self-complementary graph, the pentagon, with five vertices. Find another self-complementary graph on five vertices.

1.1.6 We have seen that the pentagon is a self-complementary graph. Let *G* be the pentagon shown in Figure 1.6, with *V(G)*={*u*1, *u*2, *u*3, *u*4, *u*5}. Notice that

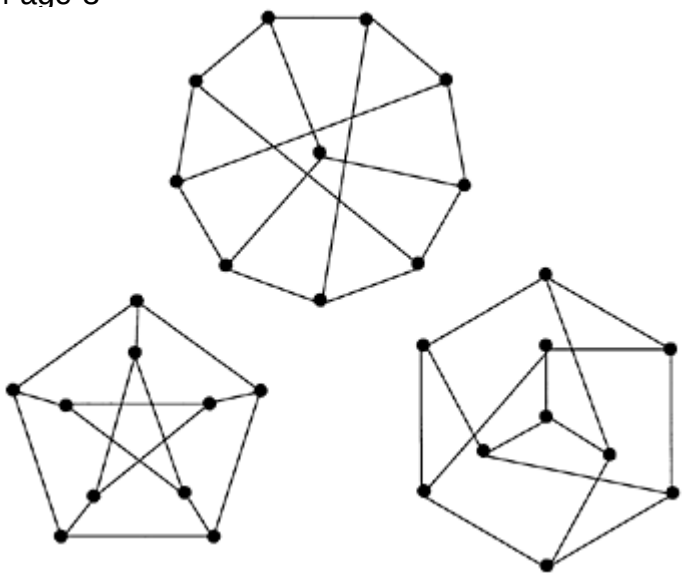


FIGURE 1.11
Three graphs on 10 vertices

$\theta=(u1)(u2, u3, u5, u4)$ is a permutation which maps G to $\overline{G}$; that is, $\theta(G)=\overline{G}$, and $\theta(\overline{G})=G$. θ is called a *complementing* permutation. Since $u2u3 \in E(G)$, it follows that $\theta(u_2u_3)=u_3u_5 \in E(\overline{G})$. Consequently, $\theta(u_3u_5)=u_5u_4 \in E(G)$ again. Applying θ twice more gives $\theta(u_5u_4)=u_4u_2 \in E(\overline{G})$ and $\theta(u4u2)= u2u3$, which is where we started. Thus, if we choose any edge $uiuj$ and successively apply θ to it, we alternately get edges of G and $\overline{G}$. It follows that the number of edges in the sequence so-obtained must be even. Use the permutation (1,2,3,4) (5,6,7,8) to construct a self-complementary graph on eight vertices.

1.1.7 Can the permutation (1, 2, 3, 4, 5) (6, 7, 8) be used as a complementing permutation? Can (1, 2, 3, 4, 5, 6) (7, 8) be? Prove that the only requirement is that every sequence of edges obtained by successively applying θ be of even length.

1.1.8 If θ is any permutation of {1, 2,..., n}, then it depends only on the cycle structure of θ whether it can be used as a complementing permutation. Discover what condition this cycle structure must satisfy, and prove it both necessary and sufficient for θ to be a complementing permutation.

1.2 Degree sequences

THEOREM 1.2 For any simple graph G we have

$$\sum_{u \in V(G)} \text{DEG}(u) = 2\varepsilon(G).$$

Page 9

PROOF An edge uu has two endpoints. Therefore each edge will be counted twice in the summation, once for u and once for u.

We use $\delta(G)$ to denote the minimum degree of G; that is, $\delta(G) = \text{MIN}\{\text{DEG}(u) \mid u \in V(G)\}$. $\Delta(G)$ denotes the maximum degree of G. By Theorem 1.2, the average degree equals $2\varepsilon/|G|$, so that $\delta \le 2\varepsilon/|G| \le \Delta$.

COROLLARY 1.3 The number of vertices of odd degree is even.

PROOF Divide $V(G)$ into $Vodd=\{u|\text{DEG}(u)$ is odd$\}$, and $Veuen=\{u|$ deg(u) is even$\}$. Then $2\varepsilon = \sum_{u \in V_{odd}} \text{DEG}(u) + \sum_{u \in V_{even}} \text{DEG}(u)$. Clearly 2ε and $\sum_{u \in V_{even}}$ DEG(u) are both even. Therefore, so is $\sum_{u \in V_{odd}} \text{DEG}(u)$, which means that $|Vodd|$ is even.

DEFINITION 1.5: A graph G is a *regular* graph if all vertices have the same degree. G is *k-regular* if it is regular, of degree k.

For example, the graph of the cube (Figure 1.1) is 3-regular.

LEMMA 1.4 If G is simple and $|G| \ge 2$, then there are always two vertices of the same degree.

PROOF In a simple graph, the maximum degree $\Delta \le |G|-1$. If all degrees were different, then they would be 0, 1, 2,..., $|G|-1$. But degree 0 and degree $|G|-1$ are mutually exclusive. Therefore there must be two vertices of the same degree.

Let $V(G)=\{u1, u2,..., un\}$. The *degree sequence* of G is

$$\text{DEG}(G)=(\text{DEG}(u1), \text{DEG}(u2),..., \text{DEG}(un))$$

where the vertices are ordered so that

$$\text{DEG}(u1) \ge \text{DEG}(u2) \ge ... \ge \text{DEG}(un).$$

Sometimes it's useful to construct a graph with a given degree sequence. For example, can there be a simple graph with five vertices whose degrees are (4, 3, 3, 2, 1)? Since there are three vertices of odd degree, Corollary 1.3 tells us that there is no such graph. We say that a sequence

$$D=(d1, d2,..., dn),$$

Page 10

is graphic if

$$d1 \ge d2 \ge ... \ge dn,$$

and there is a simple graph G with $\text{DEG}(G)=D$. So (2, 2, 2, 1) and (4, 3, 3, 2, 1) are not graphic, whereas (2, 2, 1, 1), (4, 3, 2, 2, 1) and (2, 2, 2, 2, 2, 2, 2) clearly are.

Problem 1.1: Graphic
Instance: a sequence $D=(d1,d2,...,dn)$.
Question: is D graphic?
Find: a graph G with $\text{DEG}(G)=D$, if D is graphic.

For example, (7, 6, 5, 4, 3, 3, 2) is not graphic; for any graph G with this degree sequence has $\Delta(G)=|G|=7$,

which is not possible in a simple graph. Similarly, (6, 6, 5, 4, 3, 3, 1) is not graphic; here we have $\Delta(G)=6$, $|G|=7$ and $\delta(G)=1$. But since two vertices have degree $|G|-1=6$, it is not possible to have a vertex of degree one in a simple graph with this degree sequence.
When is a sequence graphic? We want a construction which will find a graph G with $\mathrm{DEG}(G)=D$, if the sequence D is graphic.
One way is to join up vertices arbitrarily. This does not always work, since we can get stuck, even if the sequence is graphic. The following algorithm always produces a graph G with $\mathrm{DEG}(G)=D$, if D is graphic.

procedure GRAPHGEN*(D)*
Create vertices *u*1, *u*2,..., *un*
comment: upon completion, *ui* will have degree *D[i]*
graphic←**false** "assume not graphic"
i←1
while *D[i]*>0

```
do { k←D[i]
   { if there are at least k vertices with DEG > 0
     { then { join u_i to the k vertices of largest degree
              { decrease each of these degrees by 1
              { D[i] ← 0
              { comment: vertex u_i is now completely joined
     { else exit        "u_i cannot be joined"
   { i ← i + 1
```

graphic←**true**

This uses a reduction. For example, given the sequence

$$D=(3, 3, 3, 3, 3, 3),$$

the first vertex will be joined to the three vertices of largest degree, which will then reduce the sequence to (*, 3, 3, 2, 2, 2), since the vertex marked by an asterisk is now completely joined, and three others have had their degree reduced by 1. At the next stage, the first remaining vertex will be joined to the three vertices of largest degree, giving a new sequence (*, *, 2, 2, 1, 1). Two vertices are now completely joined. At the next step, the first remaining vertex will be joined to two vertices, leaving (*, *, *, 1, 1, 0). The next step joins the two remaining vertices with degree one, leaving a sequence (*, *, *, *, 0, 0) of zeroes, which we know to be graphic.
In general, given the sequence

$$D=(d1,d2,...,dn)$$

where

$$d1\geq d2\geq...\geq dn,$$

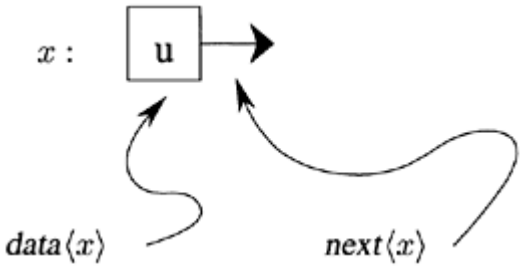
the vertex of degree $d1$ is joined to the $d1$ vertices of largest degree. This leaves the numbers

$$d_2-1, d_3-1, \ldots, d_{d_1+1}-1, d_{d_1+2}, \ldots, d_n,$$

in some order. If we rearrange them into descending order, we get the reduced sequence D'. Write

$$D'=(d'_2, d'_3 \ldots, d'_n),$$

where the first vertex $u1$ has been deleted. We now do the same calculation, using D' in place of D.
Eventually, after joining all the vertices according to their degree, we either get a graph G with $\mathrm{Deg}(G)=D$ or else at some stage, it is impossible to join some vertex ui.
An excellent data structure for representing the graph G for this problem is to have an *adjacency list* for each vertex $v \in V(G)$. The adjacency list for a vertex $v \in V(G)$ is a linked list of the vertices adjacent to u. Thus it is a data structure in which the vertices adjacent to u are arranged in a linear order. A node x in a linked list has two fields: $data\langle x\rangle$, and $next\langle x\rangle$.



Page 12

Given a node *x* in the list, $data\langle x\rangle$ is the data associated with *x* and $next\langle x\rangle$ points to the successor of *x* in the list or $next\langle x\rangle = \text{NIL}$ if *x* has no successor. We can insert data *u* into the list pointed to by *L* with procedure LISTINSERT(), and the first node on list *L* can be removed with procedure LISTREMOVEFIRST().

procedure LISITINSERT*(L, u)*
x←NEWNODE()
$data\langle x\rangle \leftarrow u$
$next\langle x\rangle \leftarrow L$
L←*x*
procedure LISTREMOVEFIRST*(L)*
x←*L*
$L \leftarrow next\langle x\rangle$
FREENODE*(x)*

We use an array *AdjList*[·] of linked lists to store the graph. For each vertex $v \in V(G)$, *AdjList[v]* points to the head of the adjacency lists for *v*. This data structure is illustrated in Figure 1.12.

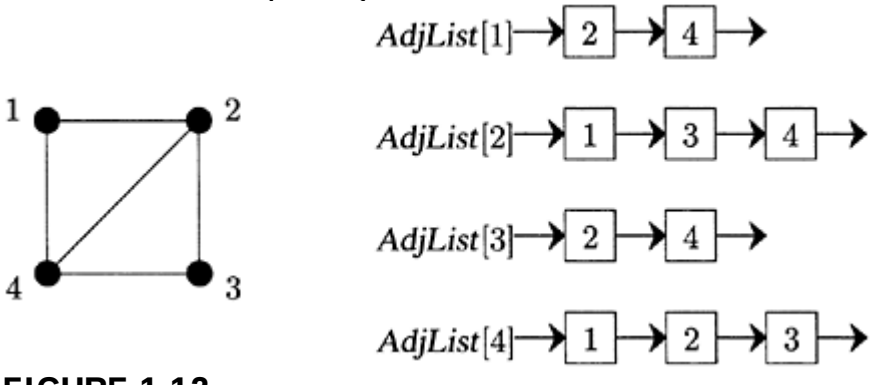


FIGURE 1.12
Adjacency lists of a graph

We can use another array of linked lists, *Pts[k],* being a linked list of the vertices *ui* whose degree-to-be *di*=*k*. With this data structure, Algorithm 1.2.1 can be written as follows:

Page 13

Algorithm 1.2.1: GRAPHGEN*(D)*
comment: {Assume D is not graphic. / Create and initialize the linked lists $Pts[k]$.
graphic←**false**
for *k*←0 **to** *n*−1 **do** *Pts[k]*←NIL
for *k*←1 **to** *n* **do** LISTINSERT*(Pts[D[k]], k)*
comment: Begin with vertex of largest degree.
for *k*←*n* **downto** 1
do while *Pts[k]*≠NIL

```
do { comment: These points are to have degree k.
     x ← Pts[k]
     u ← data⟨x⟩
     LISTREMOVEFIRST(Pts[k])
     comment: { Join u to the next k vertices v of largest degree.
              { If this is not possible, then D is not graphic so exit.
     i ← k
     for j ← 1 to k
     do { while Pts[i] = NIL do { i ← i − 1
        {                       { if i = 0 exit
        { x = Pts[i]
        { v = data⟨x⟩
        { LISTREMOVEFIRST(Pts[i])
        { LISTINSERT(AdjList[u], v)
        { LISTINSERT(AdjList[v], u)
        { LISTINSERT(TempList[i], v)
     comment: { For each such v joined to u if v is on list Pts[j],
              { then transfer v to Pts[j − 1]
     for j ← k downto 1
     do { while TempList[j] ≠ NIL
        {    do { x = TempList[j]
        {       { v = data⟨x⟩
        {       { LISTREMOVEFIRST(TempList[j])
        {       { LISTINSERT(Pts[j − 1], v)
     comment: u is now completely joined. Choose the next point.
comment: Now every vertex has been successfully joined.
graphic ← true
```

This program is illustrated in Figure 1.13 for the sequence *D*=(4, 4, 2, 2, 2, 2), where *n*=6. The diagram shows the linked lists before vertex 1 is joined to vertices 2, 3, 4, and 5, and the new configuration after joining. Care must be

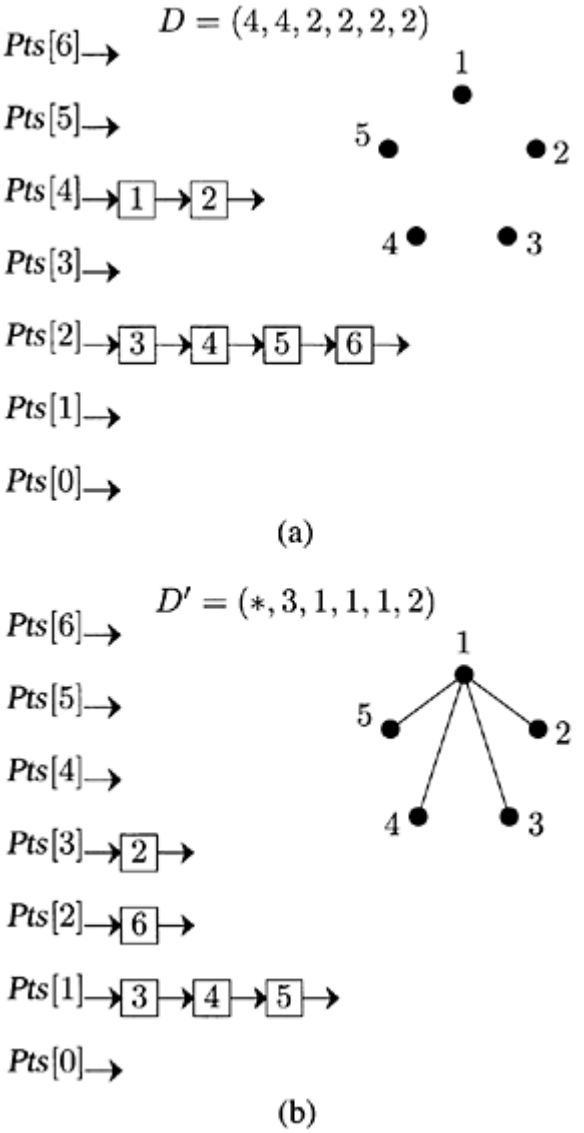


FIGURE 1.13
The linked lists *Pts[k]***. (a) Before 1 is joined to 2, 3, 4, and 5. (b) After 1 is joined to 2, 3, 4, and 5.**

used in transferring the vertices *u* from *Pts[j]* to *Pts*[*j*−1], since we do not want to join *u* to *u* more than once. The purpose of the list *Pts*[0] is to collect vertices which have been transferred from *Pts*[1] after having been joined to *u*. The degrees *d*1, *d*2,..., *dn* need not necessarily be in descending order for the program to work, since the points are placed in the lists *Pts[k]* according to their degree, thereby sorting them into buckets. Upon completion of the algorithm vertex *k* will have degree *dk*. However, when this algorithm is done by hand, it is much more convenient to begin with a sorted list of degrees; for example, *D*=(4, 3, 3, 3, 2, 2, 2, 2, 1), where *n*=9. We begin with vertex *u*1, which is to have degree four. It will be joined to the vertices *u*2, *u*3, and *u*4, all of degree three, and to one of *u*5, *u*6, *u*7, and *u*8, which have degree two. In order to keep the list of degrees sorted, we choose *u*8. We then have $u_1 \longrightarrow \{u_2, u_3, u_4, u_8\}$, and *D* is reduced to (*, 2, 2, 2, 2, 2, 2, 1, 1). We then choose *u*2 and join it to *u*6 and *u*7, thereby further reducing *D* to (*, *, 2, 2, 2, 2, 1, 1, 1, 1). Continuing in this way, we obtain a graph *G*.
In general, when constructing *G* by hand, when *uk* is to be joined to one of *ui* and *uj,* where *di*=*dj* and *i*<*j,* then join *uk* to *uj before ui,* in order to keep *D* sorted in descending order.
We still need to prove that Algorithm 1.2.1 works. It accepts a possible degree sequence

$$D=(d1,d2,\ldots dn),$$

and joins *u*1 to the *d*1 vertices of largest remaining degree. It then reduces *D* to new sequence

$$D' = (d'_2, d'_3, \ldots d'_n).$$

THEOREM 1.5 **(Havel-Hakimi theorem)** *D is graphic if and only if D′ is graphic.*

PROOF Suppose *D′* is graphic. Then there is a graph *G′* with degree sequence *D′,* where *V(G′)*={*u*2, *u*3,..., *un*} with $\text{DEG}(u_i) = d'_i$. Furthermore

$$D' = (d'_2, d'_3, \ldots, d'_n)$$

consists of the degrees

$$\{d_2 - 1, d_3 - 1, \ldots, d_{d_1+1} - 1, d_{d_1+2}, \ldots, d_n\}$$

arranged in descending order. Create a new vertex *u*1 and join it to vertices of degree

$$d_2 - 1, d_3 - 1, \ldots, d_{d_1+1} - 1.$$

Then DEG(*u*1)=*d*1. Call the new graph *G*. Clearly the degree sequence of *G* is

D=(*d*1, *d*2,..., *dn*).

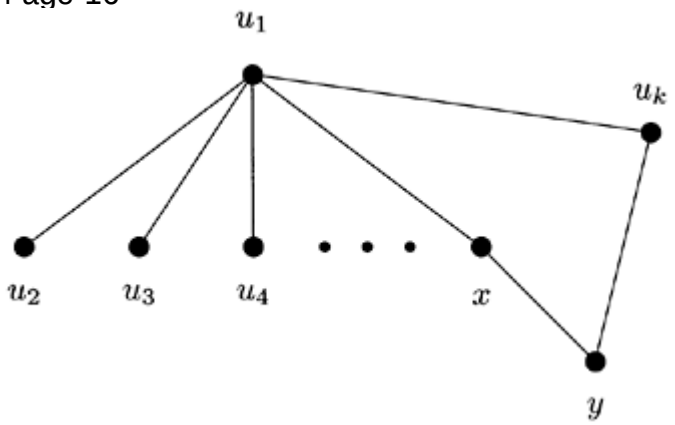


FIGURE 1.14
Vertices adjacent to *u*1

Therefore *D* is graphic.

Now suppose *D* is graphic. Then there is a graph *G* with degree sequence

D=(*d*1,*d*2,...,*dn*),

where *V(G)*={*u*1,*u*2,...,*un*}, with DEG*(ui)*=*di*. If *u*1 is adjacent to vertices of degree $d_2, d_3, \ldots, d_{d_1+1}$, then *G′*=*G*−*u*1 has degree sequence *D′*, in which case *D′* is graphic.

Otherwise, *u*1 is not adjacent to vertices of degree $d_2, d_3, \ldots, d_{d_1+1}$. Let *uk* (where *k*≥2) be the *first* vertex such that *u*1 is not joined to *uk,* but is joined to *u*2, *u*3,..., *uk*−1. (Maybe *k*=2.)

Now DEG(*u*1)=*d*1≥*k,* so *u*1 is joined to some vertex *x*≠*u*2,*u*3,...*uk*−1. *uk* is the vertex of next largest degree, so DEG*(uk)*≥DEG*(x)*. Now *x* is joined to *u*1, while *uk* is not. Therefore, there is some vertex *y* such that $u_k \longrightarrow y$ but $x \not\longrightarrow y$. Set *G*←*G*+*xy*+*u*1*uk*−*u*1*x*−*uky*.

The degree sequence of *G* has not changed, and now $u_1 \longrightarrow \{u_2, u_3, \ldots, u_k\}$. Repeat until $u_1 \longrightarrow \{u_2, u_3, \ldots, u_{d_1+1}\}$. Then *G′*=*G*−*u*1 has degree sequence *D′,* so that *D′* is graphic.

Therefore we know the algorithm will terminate with the correct answer, because it reduces *D* to *D′*. So we have an algorithmic test to check whether *D* is graphic and to generate a graph whenever one exists.

There is another way of determining whether *D* is graphic, without constructing a graph.

THEOREM 1.6 **(Erdös-Gallai theorem)** *Let D*=(*d*1,*d*2,...,*dn*), *where d*1≥

*d*2≥...≥*dn. Then D is graphic if and only if*

1. $\sum_{i=1}^{n} d_i$ *is even; and*
2. $\sum_{i=1}^{k} d_i \le k(k-1) + \sum_{i=k+1}^{n} \text{MIN}(k, d_i)$, *for k*=1,2,...,*n*.

PROOF Suppose *D* is graphic. Then $\sum_{i=1}^{n} d_i = 2\varepsilon$, which is even. Let *V*1 contain the *k* vertices of largest degree, and let *V*2=*V(G)*−*V*1 be the remaining vertices. See Figure 1.15.

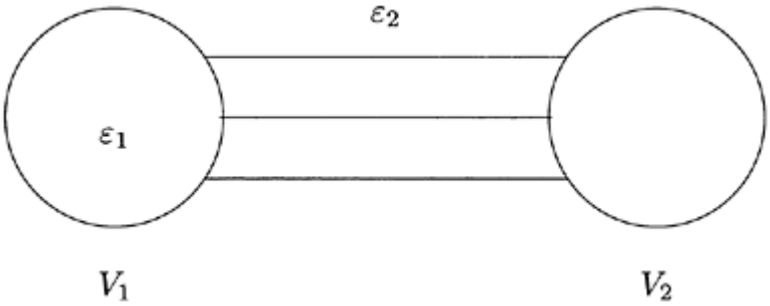


FIGURE 1.15
The vertices *V*1 of largest degree and the remaining vertices *υ*2

Suppose that there are *ε*1 edges within *V*1 and *ε*2 edges from *V*1 to *V*2. Then $\sum_{i=1}^{k} d_i = 2\varepsilon_1 + \varepsilon_2$, since each edge within *V*1 is counted twice in the sum, once for each endpoint, but edges between *V*1 and *V*2 are counted once only. Now $\varepsilon_1 \leq \binom{k}{2}$, since *V*1 can induce a complete subgraph at most. Each vertex $v \in V_2$ can be joined to at most *k* vertices in *V*1, since |*V*1|=*k,* but *u* can be joined to at most DEG*(u)* vertices in *V*1, if DEG*(u)*<*k*. Therefore *ε*2, the number of edges between *V*1 and *V*2, is at most $\sum_{v \in V_2} \text{MIN}(k, \text{DEG}(v))$, which equals $\sum_{i=k+1}^{n} \text{MIN}(k, d_i)$. This now gives $\sum_{i=1}^{k} d_i = 2\varepsilon_1 + \varepsilon_2 \leq k(k-1) + \sum_{i=k+1}^{n} \text{MIN}(k, d_i)$
The proof of the converse is quite long, and is not included here. A proof by induction can be found in the book by HARARY [59].
Conditions 1 and 2 of the above theorem are known as the *Erdös-Gallai conditions*.

Exercises

1.2.1 Prove Theorem 1.2 for arbitrary graphs. That is, prove

THEOREM 1.7 For any graph G we have

$$\sum_{u \in V(G)} Deg(u) + \ell = 2\varepsilon(G).$$

where *ℓ* is the number of loops in *G* and DEG*(u)* is the number of edges incident on *u*. What formula is obtained if loops count two toward DEG*(u)*?

1.2.2 If *G* has degree sequence *D*=(*d*1, *d*2,..., *dn*), what is the degree sequence of $\overline{G}$?

1.2.3 We know that a simple graph with *n* vertices has at least one pair of vertices of equal degree, if *n*≥2. Find all simple graphs with exactly one pair of vertices with equal degrees. What are their degree sequences? *Hint:* Begin with *n*=2, 3, 4. Use a recursive construction. Can degree 0 or n−1 occur twice?

1.2.4 Program the GRAPHGEN() algorithm. Input the sequence *D*=(*d*1, *d*2,..., *dn*) and then construct a graph with that degree sequence, or else determine that the sequence is not graphic. Use the following input data:
(a) 4 4 4 4 4
(b) 3 3 3 3 3 3
(c) 3 3 3 3 3 3 3 3
(d) 3 3 3 3 3 3 3 3 3
(e) 2 2 2 2 2 2 2 2 2 2
(f) 7 6 6 6 5 5 2 1

1.2.5 Let *D*=(*d*1, *d*2,..., *dn*), where *d*1≥*d*2≥...≥*dn*. Prove that there is a multigraph with degree sequence *D* if and only if $\sum_{i=1}^{n} d_i$ is even, and *d*1≤ $\sum_{i=2}^{n} d_i$.

1.3 Analysis

Let us estimate the number of steps that Algorithm 1.2.1 performs. Consider the loop structure
for *k*←*n* **downto** 1
do while Pts[k]≠**NIL**
do {...
The for-loop performs *n* iterations. For many of these iterations, the contents of the while-loop will not be executed, since *Pts[k]* will be NIL. When the contents of the loop are executed, vertex *u* of degree-to-be *k* will be joined to *k* vertices. This means that *k* edges will be added to the adjacency lists of the graph *G* being constructed. This takes 2*k* steps, since an edge *uu* must be added to both *GraphAdj[u]* and *GraphAdj[u]*. It also makes *DEG(u)=k*. When edge *uu* is added, *u* will be transferred from *Pts[j]* to *Pts*[*j*−1], requiring additional *k* steps. Once *u* has been joined, it is removed from the list. Write $\varepsilon = \frac{1}{2}\sum_i d_i$,

Page 19
the number of edges of *G* when *D* is graphic. Then, in all, the combination for-while-loop will perform exactly 2*ε* steps adding edges to the graph and a further *ε* steps transferring vertices to other lists, plus *n* steps for the *n* iterations of the for-loop. This gives a total of 3*ε*+*n* steps for the for-while-loop. The other work that the algorithm performs is to create and initialize the lists *Pts*[·], which takes 2*n* steps altogether. So we can say that in total, the algorithm performs 3*ε*+3*n* steps.
Now it is obvious that each of these "steps" is composed of many other smaller steps, for there are various comparisons and assignments in the algorithm which we have not explicitly taken account of (they are subsumed into the steps we have explicitly counted). Furthermore, when compiled into assembly language, each step will be replaced by many smaller steps. Assembly language is in turn executed by the microprogramming of a computer, and eventually we come down to logic gates, flip-flops, and registers. Because of this fact, and because each computer has its own architecture and machine characteristics, it is customary to ignore the constant coefficients of the graph parameters *ε* and n, and to say that the algorithm has *order ε*+*n,* which is denoted by *O*(*ε*+*n*), pronounced "big Oh of *ε*+*n*". A formal definition is provided by Definition 1.6. Even though the actual running time of a given algorithm depends on the architecture of the machine it is run on, the programmer can often make a reasonable estimate of the number of steps of some constant size (e.g., counting one assignment, comparison, addition, multiplication, etc. as one step), and thereby obtain a formula like 3*ε*+3*n*. Such an algorithm will obviously be superior to one which takes 15*ε*+12*n* steps of similar size. Because of this fact, we shall try to obtain formulas of this form whenever possible, as well as expressing the result in a form like *O*(*ε*+*n*).
The *complexity* of an algorithm is the number of steps it must perform, in the worst possible case. That is, it is an *upper bound* on the number of steps. Since the size of each step is an unknown constant, formulas like 5*n*2/6 and 25*n*2 are both expressed as *O*(*n*2). We now give a formal definition of this notation.
DEFINITION 1.6: Suppose $f : \mathbb{Z}^+ \to \mathbb{R}$ and $g : \mathbb{Z}^+ \to \mathbb{R}$. We say that *f(n)* is *O(g(n))* provided that there exist constants *c*>0 and *n*0≥0 such that 0≤*f(n)*≤*c*·*g(n)* for all *n*≥*n*0.
In other words, *f(n)* is *O(g(n))* provided that *f(n)* is bounded above by a constant factor times *g(n)* for large enough *n*. For example, the function 5*n*3+ 2*n*+1 is *O*(*n*3), because for all *n*≥1, we have

$$5n3+2n+1 \le 5n3+2n3+n3=8n3.$$

Hence, we can take *c*=8 and *n*0=1, and Definition 1.6 is satisfied.
The notation *f(n)* is Ω*(g(n))* ("big omega") is used to indicate that *f(n)* is bounded below by a constant factor times *g(n)* for large enough *n*.

Page 20
DEFINITION 1.7: Suppose $f : \mathbb{Z}^+ \to \mathbb{R}$ and $g : \mathbb{Z}^+ \to \mathbb{R}$. We say that *f(n)* is Ω*(g(n))* provided that there exist constants *c*>0 and *n*0≥0 such that *f(n)*≥*c*·*g(n)*≥0 for all *n*≥*n*0.
We say that *f(n)* is Θ*(g(n))* ("big theta") when *f(n)* is bounded above and below by constant factors times *g(n)*. The constant factors may be different. More precisely:
DEFINITION 1.8: Suppose $f : \mathbb{Z}^+ \to \mathbb{R}$ and $g : \mathbb{Z}^+ \to \mathbb{R}$. We say that *f(n)* is Θ*(g(n))* provided that there exist constants *c, c'*>0 and *n*0≥0 such that 0≤*c*·*g(n)*≤*f(n)*≤*c'*·*g(n)* for all *n*≥*n*0.
If *f(n)* is Θ*(g(n)),* then we say that *f* and *g* have the same *growth rate*.
The big *O*-notation is a method of indicating the *qualitative* nature of the formula, whether quadratic, linear, logarithmic, exponential, etc. Notice that "equations" involving *O*(·) are not really equations, since *O*(·) can only be used in this sense on the right hand side of the equals sign. For example, we could also have shown that 10*n*2+4*n*−4 is *O*(*n*3) or that 10*n*2+4*n*−4 is *O*(2n), but these expressions are not equal to each other. Given a complexity formula like 10*n*2+4*n*−4, we want the smallest function *f(n)* such that 10*n*2+4*n*−4 is *O(f(n))*. Among the useful rules for working with the *O*-notation are the following sum and product rules.
*THEOREM 1.8 Suppose that the two functions f*1*(n) and f*2*(n) are both O(g(n)). Then the function f*1*(n)*+*f*2*(n) is O(g(n))*.
*THEOREM 1.9 Suppose that f1(n) is O(g1(n)) and f*2*(n) is O(g2(n)). Then the function f*1*(n) f*2*(n) is O*(*g*1*(n) g*1*(n)*).
As examples of the use of these notations, we have that *n*2 is *O*(*n*3), *n*3 is Ω(*n*2), and 2*n*2+3*n*−sin *n*+1/*n* is Θ(*n*2).
Several properties of growth rates of functions that arise frequently in algorithm analysis follow. The first of these says that a polynomial of degree *d,* in which the high-order coefficient is positive, has growth rate *nd*.
THEOREM 1.10 Suppose that ad>0. *Then the function a*0+*a*1*n*+…+*adndis* Θ*(nd)*.
The next result says that *logarithmic growth* does not depend on the base to which logarithms are computed.

It can be proved easily using the formula $\log_a n = \log_a b \cdot \log_b n$.
THEOREM 1.11 The function $\log_a n$ *is* $\Theta(\log_b n)$ *for any* $a, b > 1$.

Page 21
The next result can be proved using Stirling's formula. It gives the growth rate of the factorial function in terms of exponential functions.

THEOREM 1.12 The function n! is $\Theta(n^{n+1/2}e^{-n})$.

Exercises

1.3.1 Show that if G is a simple graph with n vertices and ε edges, then $\log \varepsilon = O(\log n)$.

1.3.2 Consider the following statements which count the number of edges in a graph, whose adjacency matrix is *Adj*.

Edges←0
for *u*←1 **to** *n*−1
do for *v*←*u*+1 **to** *n*
do if *Adj[u,u]*=1
then *Edges*←*Edges*+1

Calculate the number of steps the algorithm performs. Then calculate the number of steps required by the following statements in which the graph is stored in adjacency lists:

Edges←0
for *u*←1 **to** *n*−1
do for each $v \longrightarrow u$
do if *u*<*v*
then *Edges*←*Edges*+1

What purpose does the condition *u*<*v* fulfill, and how can it be avoided?

1.3.3 Use induction to prove that the following formulas hold:

(a) $1 + 2 + 3 + \cdots + n = \binom{n+1}{2}$

(b) $\binom{2}{2} + \binom{3}{2} + \binom{4}{2} + \cdots + \binom{n}{2} = \binom{n+1}{3}$.

(c) $\binom{t}{t} + \binom{t+1}{t} + \binom{t+2}{t} + \cdots + \binom{n}{t} = \binom{n+1}{t+1}$.

1.3.4 Show that $3n^2 + 12n = O(n^2)$; that is, find constants A and N such that $3n^2 + 12n < An^2$ whenever $n \geq N$.

1.3.5 Show that $\log(n+1) = O(\log n)$, where the logarithm is to base 2.

1.3.6 Use the answer to the previous question to prove that

$$(n+1)\log(n+1) = O(n \log n).$$

1.3.7 Prove that if $f1(n)$ and $f2(n)$ are both $O(g(n))$, then $f1(n) + f2(n)$ is $O(g(n))$.

1.3.8 Prove that if $f1(n)$ is $O(g1(n))$ and $f2(n)$ is $O(g2(n))$, then $f1(n)f2(n)$ is $O(g1(n)g1(n))$.

Page 22

1.4 Notes

Some good general books on graph theory are BERGE [14], BOLLOBÁS [18], BONDY and MURTY [19], CHARTRAND and LESNIAK [24], CHARTRAND and OELLERMANN [25], DIESTEL [35], GOULD [53], and WEST [123]. A very readable introductory book is TRUDEAU [115]. GIBBONS [51] is an excellent treatment of graph algorithms. A good book discussing the analysis of algorithms is PuRDOM and BROWN [96]. AHO, HOPCROFT, and ULLMAN [1], SEDGEWICK [108] and WEISS [122] are all excellent treatments of data structures and algorithm analysis.

Page 23

2

Paths and Walks

2.1 Introduction

Let u and v be vertices of a simple graph G. A *path* P from u to v is a sequence of vertices $u0, u1, \ldots, uk$ such that $u = u0$, $v = uk$, $u_i \longrightarrow u_{i+1}$, and all the ui are distinct vertices. The length of a path P is $l(P)$, the number of *edges* it uses. In this example, $l(P) = k$, and P is called a uv-path of length k. A uv-path of length 4 is illustrated in Figure 2.1, with dashed edges.

A *cycle* C is a sequence of vertices $u0, u1, \ldots, uk$ forming a $u0uk$-path, such that $u_k \longrightarrow u_0$. The length of C is $l(C)$, the number of edges that it uses. In this case, $l(C) = k+1$.

A *uv*-path *P connects u* to *v*. The set of all vertices connected to any vertex *u* forms a subgraph *Cu*, the *connected component* of *G* containing *u*. It will often be the case that *Cu* contains all of *G,* in which case *G* is a *connected graph*. *w(G)* denotes the number of distinct connected components of *G*. The graph of Figure 2.1 is disconnected, with *w*=3.
There are several ways of finding the connected components of a graph *G*. One way to find the sets *Cu* for a graph *G* is as follows:
procedure COMPONENTS*(G)*
for each $u \in V(G)$
do initialize *Cu* to contain only *u*
for each $u \in V(G)$
do $\begin{cases} \textbf{for each } v \longrightarrow u \\ \quad \textbf{do if } C_u \neq C_v \textbf{ then } \textsc{Merge}(C_u, C_v) \end{cases}$
The inner for-loop ensures that, upon completion, if $u \longrightarrow v$, then *Cu*=*Cv,* for any vertices *u* and *?*. Therefore, if *P*=(*u*0,*u*1,...,*uk*) is any path, we can

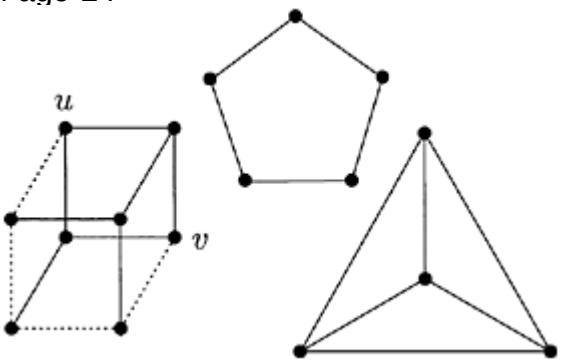
Page 24


FIGURE 2.1
A graph with three components
be sure that $C_{u_0} = C_{u_1} = \cdots = C_{u_k}$, so that when the algorithm terminates, each *Cu* will contain all the vertices connected to *u* by any path; that is, *Cu* will be the connected component containing *u*.
The complexity of the algorithm naturally depends upon the data structures used to program it. This algorithm is a perfect example of the use of the *merge-find* data structure. Initially, each *Cu*=*{u}* and *Cu*=*{u}*. When the edge *uu* is examined, *Cu* and *Cu* are merged, so that now *Cu*=*Cu*=*{u,u}*. The two operations which need to be performed are to determine whether *Cu*=*Cu*, and to merge *Cu* and *Cu* into one. This can be done very efficiently by choosing a vertex in each *Cu* as *component representative*.
uRep←COMPREP*(Cu)*
vRep←COMPREP*(Cu)*
if *uRep*≠*vRep*
then MERGE*(Cu,Cu)*
Initially, *Cu*=*{u},* so that *u* begins as the representative of *Cu*. Associated with each vertex *u* is a pointer toward the representative of the component containing *u*. To find the representative of *Cu,* we start at *u* and follow these pointers, until we come to the component representative. The component representative is marked by a pointer that is negative. The initial value is—1. The pointers are easily stored as an array, *CompPtr*.
COMPREP() is a recursive procedure that follows the component pointers until a negative value is reached.

Page 25
procedure COMPREP*(u)*
if *CompPtr[u]*<0
then return *(u)*
$$\textbf{else} \begin{cases} theRep \leftarrow \textsc{CompRep}(CompPtr[u]) \\ CompPtr[u] \leftarrow theRep \\ \textbf{return } (theRep) \end{cases}$$

The assignment

$$CompPtr[u] \leftarrow theRep$$

is called *path compression*. It ensures that the next time *CompPtr(u)* is computed, the representative will be found more quickly. The algorithm COMPONENTS() can now be written as follows:

Algorithm 2.1.1: COMPONENTS*(G)*

```
    n←|G|
    for u←1 to n
do  do CompPtr[u]← −1
    for u←1 to n
do  { for each v ⟶ u
      do { uRep ← CompRep(u)
           vRep ← CompRep(v)
           if uRep ≠ vRep
             then Merge(uRep, vRep)
```

The essential step in merging *Cu* and *Cu* is to assign either

$$CompPtr[vRep] \leftarrow uRep$$

or

$$CompPtr[uRep] \leftarrow vRep$$

The best one to choose is that which merges the smaller component onto the larger. We can determine the size of each component by making use of the negative values of *CompPtr[uRep]* and *CompPtr[vRep]*. Initially, *CompPtr[u]*=−1, indicating a component of size one.

---

procedure MERGE*(uRep,vRep)*

uSize← *-CompPtr[uRep]*

vSize← *-CompPtr[vRep]*

if *uSize*<*vSize*

$$\textbf{then} \begin{cases} CompPtr[uRep] \leftarrow vRep \\ CompPtr[vRep] \leftarrow -(uSize + vSize) \end{cases}$$

$$\textbf{else} \begin{cases} CompPtr[vRep] \leftarrow uRep \\ CompPtr[uRep] \leftarrow -(uSize + vSize) \end{cases}$$

When *Cu* and *Cv* are merged, the new component representative (either *uRep* or *vRep*) has its *CompPtr*[·] assigned equal to—(*uSize*+*vSize*). The component pointers can be illustrated graphically. They are shown in Figure 2.2 as arrows. The merge operation is indicated by the dashed line.

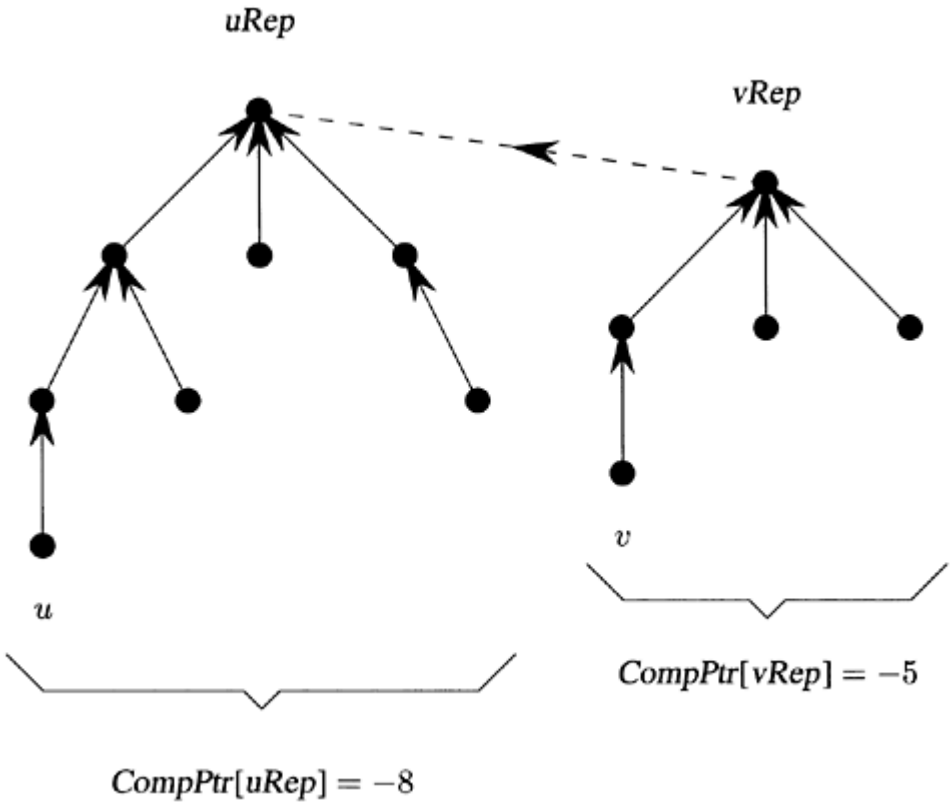


FIGURE 2.2
Component representatives

2.2 Complexity

The components algorithm is very efficient. The for-loop which initializes the *CompPtr* array requires n steps. If adjacency lists are used to store G, then the total number of times that the body of the main loop is executed is

$$\sum \text{DEG}(u) = 2\varepsilon.$$

Thus COMPREP() is called 4ε times. How many times is MERGE() called? At each merge, two existing components are replaced by one, so that at most $n-1$ merges can take place. Each merge can be performed using four assignments and a comparison. It takes n steps to initialize the *CompPtr* array. Thus the total number of steps is about $6n+4\varepsilon\cdot$(number of steps per call to COMPREP()). The number of steps each call to COMPREP() requires depends on the depth of the trees which represent the components. The depth is changed by path compression, and by merging. It is proved in AHO, HOPCROFT, and ULLMAN [1], that if there are a total of n points involved, the number of steps required is $O(a(n))$, where $a(n)$ is the inverse of the function $A(n)$, defined recursively as follows.

$$A(1)=1$$
$$A(k)=2A(k-1)$$

Thus, $A(2)=2^1=2$, $A(3)=2^2=4$, $A(4)=2^4=16$, $A(5)=2^{16}=65536$, etc. It follows that $a(n)\leq 5$, for all $n\leq 65536$. So the complexity of Algorithm 2.1.1 is almost linear, namely, $O(n+\varepsilon a(n))$, where $a(n)\leq 5$, for all practical values of n.

Exercises

2.2.1 Assuming the data structures described above, program the COMPONENTS() algorithm, merging the smaller component onto the larger. Include an integer variable *NComps* which contains the current number of components. Upon completion, its value will equal $w(G)$.

2.2.2 Algorithm 2.1.1 computes the connected components Cu using the array *CompPtr*. If we now want to print the vertices of each distinct Cu, it cannot be done very efficiently. Show how to use linked lists so that for each component, a list of the vertices it contains is available. Rewrite the MERGE() procedure to include this. Is the complexity thereby affected?

2.2.3 In the Algorithm 2.1.1 procedure, the for-loop

for $u \leftarrow 1$ **to** n **do**

executes the statement *uRep*←COMPREP*(u)* once for every $v \longrightarrow u$. Show how to make this more efficient by taking the statement *uRep*←COMPREP*(u)* out of the *u*-loop, and modifying the MERGE() procedure slightly. Calculate the new complexity.

2.2.4 Let *n*=|*G*|. Show that if $\varepsilon > \binom{n-1}{2}$, then *G* is connected. *Hint:* If *G* is disconnected, there is a component of size *x*<*n*. What is the maximum number of edges *G* can then have?

2.2.5 Show that if $\delta > \lfloor (n-1)/2 \rfloor$, then *G* is connected.

2.2.6 Show that if *G* is disconnected, then $\overline{G}$ is connected.

2.2.7 Show that if *G* is simple and connected but not complete, then *G* has three vertices *u, u,* and *w* such that $u \longrightarrow v, w$, but $v \not\longrightarrow w$.

2.2.8 A *longest path* in a graph *G* is any path *P* such that *G* contains no path longer than *P*. Thus a graph can have several different longest paths (all of the same length, though). Show that *ℓ(P)≥δ(G),* for any longest path. *Hint:* Consider an endpoint of *P*.

2.2.9 Show that every graph *G* has a cycle of length at least *δ(G)*+1, if *δ(G)*≥2. *Hint:* Consider a longest path.

2.2.10 Prove that in a connected graph, any two longest paths have at least one vertex in common.

2.3 Walks

Paths do not contain repeated vertices or edges. A *walk* in *G* is any sequence of vertices *u*0, *u*1,..., *uk* such that $u_i \longrightarrow u_{i+1}$. Thus, in a walk, edges and vertices may be repeated. Walks are important because of their connection with the adjacency matrix of a graph. Let *A* be the adjacency matrix of *G*, where *V(G)*={*u*1,*u*2..., *un*}, such that row and column *i* of *A* correspond to vertex *ui*.

THEOREM 2.1 Entry [i,j] of Akis the number of walks of length k from vertex ui to uj.

PROOF By induction on *k*. When *k*=1, there is a walk of length 1 from *ui* to *uj* if and only if $u_i \longrightarrow u_j$, in which case entry *A[i,j]*=1. Assume it's true whenever *k*≤*t* and consider *At*+1. Let *W* be a *uiuj*-walk of length *t*+1, where *t*≥2. If *ul* is the vertex before *uj* on *W,* then *W* can be written as (*W′, u*l*, uj*), where *W′* is a *uiul*-walk of length *t*. Furthermore, every *uiul*-walk of length *t* gives a *uiuj*-walk of length *t*+1 whenever $u_l \longrightarrow u_j$. Therefore the number of

uiuj-walks of length *t*+1 is

$$\sum_l (\text{the number of } u_i u_l - \text{walks of length } t)(A[l,j]).$$

But the number of *uiul*-walks of length *t* is *At[i,l],* so that the number of *uiuj*-walks of length *t*+1 is

$$\sum_{l=1}^{n} A^t[i,l]A[l,j],$$

which equals *At*+1*[i,j]*. Therefore the result is true when *k*=*t*+1. By induction, it's true for all values of *k*. Notice that this result is also true for multigraphs, where now *A[i,j]* is the number of edges joining *ui* to *uj*. For multigraphs, a walk *W* must be specified by giving the sequence of edges traversed, as well as the sequence of vertices, since there can be more than one edge joining the same pair of vertices.

Exercises

2.3.1 Show that *A*2*[i,j]* equals the number of *uiuj*-paths of length 2, if *i*≠*j,* and that *A*2*[i,i]*=DEG*(ui)*.

2.3.2 Show that *A*3*[i,i]* equals the number of triangles containing vertex *ui*. Find a similar interpretation of *A*3*[i,j],* when *i*≠*j*. (A triangle is a cycle of length 3.)

2.3.3 *Ak* contains the number of walks of length *k* connecting any two vertices. Multiply *Ak* by *xk*, the *kth* power of a variable *x,* and sum over *k,* to get the matrix power series *I*+*Ax*+*A*2*x*2+*A*3*x*3+..., where *I* is the identity matrix. The sum of this power series is a matrix whose *ijth* entry is a function of *x* containing the number of uiuj-walks of each length, as the coefficient of *xk*. Since the power series expansion of (1−*a*)−1 is 1+*a*+*a*2+*a*3+..., we can write the above matrix as (*I*−*Ax)*−1. That is, the inverse of the matrix *(I*−*Ax*) is the *walk generating matrix*. Find the walk generating matrix for the graph of Figure 2.3.

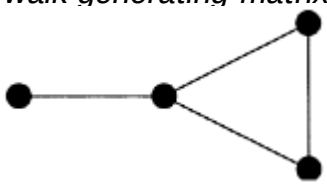


FIGURE 2.3
Compute the number of walks in this graph.

Page 30

2.4 The shortest-path problem

The *distance* from vertex *u* to *v* is DIST*(u,v)*, the length of the shortest *uv*-path. If *G* contains no *uv*-path, then DIST*(u,v)*=∞. In this section we study the following two problems.

Problem 2.1:
Shortest Path

Instance: a graph *G* and a vertex *u*.
Find: DIST*(u,v)*, for all *u*, $v \in V(G)$.

Problem 2.2:
All Paths

Instance: a graph *G*.
Find: DIST*(u, v)*, for all *u*, $v \in V(G)$.

Given a vertex *u*, one way of computing DIST*(u,v)*, for all *v*, is to use a *breadth-first search* (BFS), as is done in procedure BFS().

procedure BFS*(G,u)*
comment: $\begin{cases} ScanQ \text{ is a queue of vertices} \\ dist[v] \text{ will equal DIST(u,v), upon completion} \end{cases}$
for each $v \in V(G)$
do *dist[v]*←8
dist[u]←0
place *u* on *ScanQ*
repeat
select *v* for the head of *ScanQ*
for each $w \longrightarrow v$
do if *w* not on *ScanQ*
then $\begin{cases} \text{add } w \text{ to the end of } ScanQ \\ dist[w] \leftarrow dist[v] + 1 \end{cases}$
advance *ScanQ*
until all of *ScanQ* has been processed

Procedure BFS() uses a type of data structure called a *queue*. A queue is an ordered list in which we usually access only the first or the *head* of the list and new items are only placed at the end or *tail* of the list. This is similar to one's

Page 31

experience of waiting in line at the checkout counter of a store. The person at the head of the line is processed first by the checker and the new customers enter at the end of the line. One of the most convenient ways to store a queue is as an array. For when an algorithm builds a queue on an array, all the vertices visited are on the array when the algorithm completes, ready for input to the next procedure. BFS() works in this way.

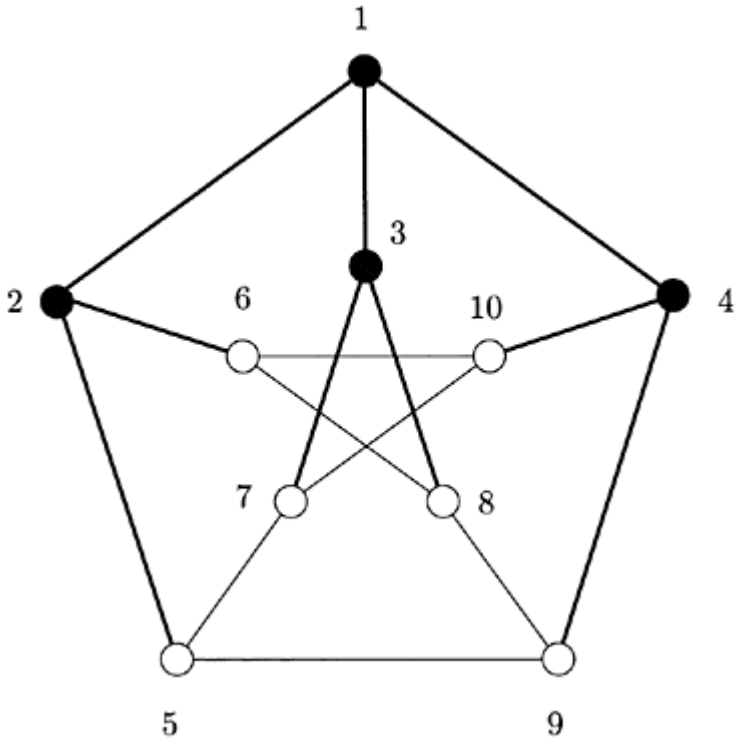


FIGURE 2.4
A breadth-first search

The breadth-first search *(BFS)* algorithm is a fundamental algorithm in graph theory. It appears in various guises wherever shortest paths are useful (e.g., network flows, matching theory, coset enumeration, etc.). Figure 2.4 shows the result of applying a BFS to the Petersen graph, where the vertices are numbered according to the order in which they were visited by the algorithm, and shaded according to their distance from vertex 1. The thicker edges show the shortest paths found by the algorithm.

Notice that the first vertex on the *ScanQ* is *u,* whose *dist[u]*=DIST*(u,u)*=0. The next vertices to be placed on the queue will be those adjacent to *u,* that is, those at distance 1. When they are placed on the queue, their distance will be computed as

$$dist[\cdot] \leftarrow dist[u]+1.$$

So we can say that initially, that is, up to vertices of distance one, vertices are placed on the queue in order of their distance from *u;* and that when each vertex *w* is placed on *ScanQ, dist[w]* is made equal to DIST*(u,w)*. Assume that this is true for all vertices of distance *k* or less, where $k \geq 1$. Consider when *u* is chosen as the first vertex of distance *k* on *ScanQ*. The for-loop examines all vertices $w \longrightarrow v$. If *w* on *ScanQ* already, then there is a *uw*-path of length$\leq k$, and *w* is ignored. If *w* is not on *ScanQ,* then DIST*(u,w)*$> k$. The *uw*-path via *u* has length $k+1$, so *w* is added to the queue, and *dist[w]* is set equal to *dist[u]*$+1 = k+1$. Since every vertex at distance $k+1$ is adjacent to a vertex at distance *k,* we can be sure that when all vertices *u* on *ScanQ* at distance *k* have been scanned, all vertices at distance $k+1$ will be on the queue. Thus the assertion that vertices are placed on the queue in order of their distance from *u,* and that when each vertex *w* is placed on *ScanQ, dist[w]* is made equal to DIST*(u,w),* is true up to distance $k+1$. By induction, it is true for all distances.

This proof that the BFS() algorithm works illustrates how difficult and cumbersome it can be to prove that even a simple, intuitively "obvious" algorithm works correctly. Nevertheless, it is important to be able to prove that algorithms work correctly, especially the more difficult algorithms. Writing down a proof for an "obvious" algorithm will often reveal hidden bugs that it contains. This proof also illustrates another feature, namely, proofs that algorithms work tend to use induction, often on the number of iterations of a main loop.

The complexity of the BFS() is very easy to calculate. The main operations which are performed are

1. Scan all $w \longrightarrow v$.

2. Select the next $v \in ScanQ$.

3. Determine whether $w \in ScanQ$.

The first operation is most efficiently done if *G* is stored in adjacency lists. We want the second and third operations to take a constant number of steps. We store *ScanQ* as an integer array, and also store a boolean

array *onScanQ* to tell whether $w \in ScanQ$. The revised algorithm is Algorithm 2.4.1.

Page 33

Algorithm 2.4.1: BFS*(G,u)*

global *n*
for *u*←1 **to** *n*
do $\begin{cases} dist[v] \leftarrow \infty \\ onScanQ[v] \leftarrow \textbf{false} \end{cases}$
dist[u]←0
ScanQ[1]←*u*
onScanQ[u]←**true**
QSize←1
k←1
repeat
u←*ScanQ[k]*
for each $w \longrightarrow v$
do if not *onScanQ[w]*
then $\begin{cases} QSize \leftarrow QSize + 1 \\ ScanQ[QSize] \leftarrow w \\ onScanQ[w] \leftarrow \textbf{true} \\ dist[w] \leftarrow dist[v] + 1 \end{cases}$
k←*k*+1
until *k*>*QSize*

The initialization takes 2*n* steps. The repeat-loop runs at most *n* times. At most *n* vertices are placed on the queue. The for-loop over all $w \longrightarrow v$ requires

$$\sum_v \text{DEG}(v) = 2\varepsilon$$

steps, all told. This assumes that the adjacent vertices are stored in a linked list—the for-loop traverses the adjacency list. Therefore the total number of steps executed is at most

$$3n+2\varepsilon = O(n+\varepsilon) = O(\varepsilon).$$

Notice that in this program we could have dispensed with the array *onScanQ*, by using instead *dist[w]*=∞ to determine *w* is on *ScanQ*. Because a breath-first search always uses a queue but not always a *dist*[·] array, we have kept the boolean array, too.

2.5 Weighted graphs and Dijkstra's algorithm

A breath-first search calculates DIST*(u,u)* correctly because in a simple graph, each edge has "length" one; that is, the length of a path is the number of edges

Page 34

it contains. In a more general application where graphs are used to model a road network, or distribution network, etc., we may want to assign a length≥1 to each edge. This is illustrated in Figure 2.5.

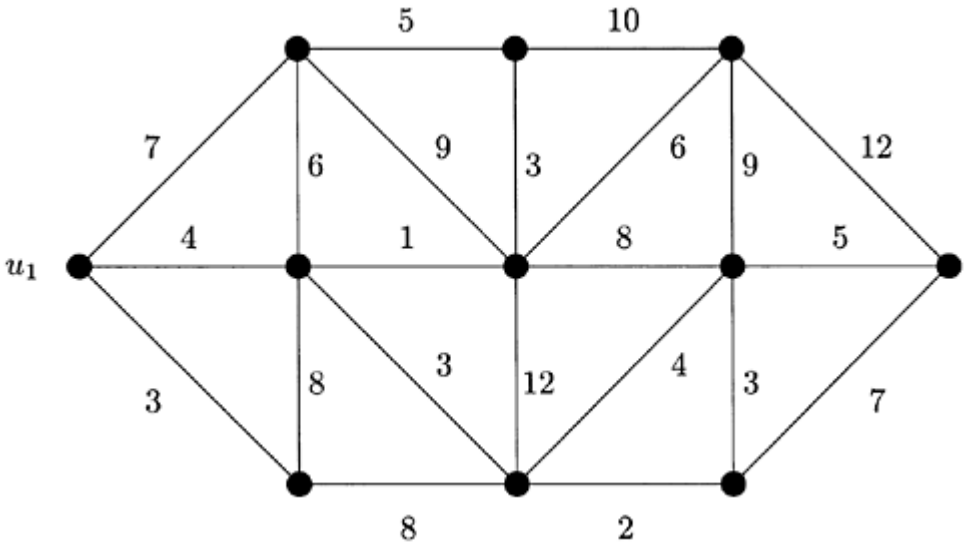


FIGURE 2.5
A weighted graph

This is an example of a *weighted graph*. Each edge $uv \in E(G)$ is assigned a *positive integral weight* WT*(uu)*. WT*(uu)* may represent length, weight, cost, capacity, etc., depending on the application. In a weighted graph, the length of a path *P*=(*u*0,*u*1,..., *uk*) is

$$\ell(P) = \sum_{i=0}^{k-1} \text{WT}(u_i u_{i+1}).$$

The distance between two vertices is now defined as

DIST*(u,u)*=MIN{*ℓ(P): P* is a *uu*-path}

A breath-first search will not compute DIST*(u,u)* correctly in a weighted graph, because a path with more edges may have the shorter length. There are many algorithms for computing shortest paths in a weighted graph. Dijkstra's algorithm is one.

Page 35

procedure DIJKSTRA*(u)*

comment:
- Compute DIST(u, v), for all $v \in V(G)$
- *dist*$[v]$ will equal DIST(u, v) upon completion.
- Vertices are chosen as $u_1, u_2, \ldots, u_n$,
- in order of their distance from u.

*u*1←*u* "the nearest vertex to *u*."

for *k*←1 **to** *n*-1

do
- **comment:**
  - $u_1, u_2, \ldots, u_k$ are currently known
  - in this iteration, u_{k+1} is selected
- select v, the nearest vertex to u_1, such that $v \notin \{u_1, u_2, \ldots, u_k\}$
- $u_{k+1} \leftarrow v$
- assign *dist*$[u_{k+1}]$
- **comment:** $u_1, u_2, \ldots, u_{k+1}$ are now known

comment: all *dist[ui]* are now known

Dijkstra's algorithm is an example of a so-called "greedy" or "myopic" algorithm, that is, an algorithm which always selects the next nearest, or next best, etc., on each iteration. Many problems can be solved by greedy algorithms.

We need to know how to choose *v,* the next nearest vertex to *u*1, in each iteration. On the first iteration, it will be the vertex *v* adjacent to *u*1 such that WT(*u*1*v*) is minimum. This will {*u*1*, u*2} such that DIST(*u*1, *u*1) and DIST(*u*1, *u*2) are known. On the next iteration, the vertex *?* chosen will be adjacent to one of *u*1 or *u*2. The distance to *u*1 will then be either

DIST(*u*1,*u*1)+WT(*u*1*v*)

or

$$\text{DIST}(u1, u2)+\text{WT}(u2v),$$

and *v* will be the vertex for which this sum is minimum.
In general, at the beginning of iteration *k,* vertices *u*1, *u*2*…*, *uk* will have been chosen, and for these vertices,

$$\text{DIST}[ui]=\text{DIST}(u1,ui).$$

The next nearest vertex *v* must be adjacent to some ui, so that the shortest *u*1*v*-path will have length dist*[ui]*+WT*(uiv),* for some *i. v* is chosen as the vertex for which this value is a minimum. This is illustrated in Figure 2.6. The refined code for Dijkstra's algorithm is Algorithm 2.5.1.

Page 36

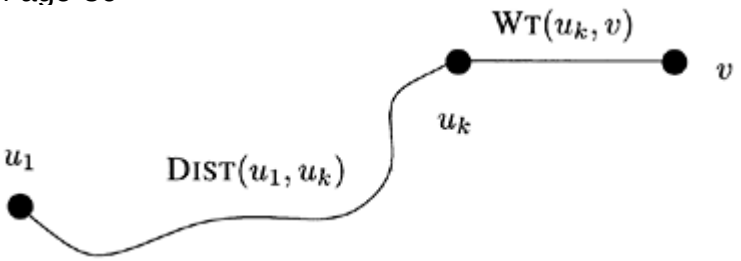


FIGURE 2.6
A shortest *u*1*u*-path, via vertex *uk*

Algorithm 2.5.1: DIJKSTRA*(u)*

comment: Compute $\text{DIST}(u, v)$, for all $v \in V(G)$; *dist*$[v]$ will equal $\text{DIST}(u, v)$ upon completion. Vertices are chosen as $u_1, u_2, \ldots, u_n$, in order of their distance from u

for each u
do *dist[u]*←∞
*u*1←*u* "the nearest vertex to *u*"
dist[u]←0
for *k*←1 **to** *n*−1
do
- **comment:** $u_1, u_2, \ldots, u_k$ are currently known; in this iteration, u_{k+1} is selected
- **for each** $v \longrightarrow u_k$ such that $v \notin \{u_1, u_2, \ldots, u_k\}$
  - **do** *dist*$[v] \leftarrow \text{MIN}(dist[v], dist[u_k] + \text{WT}(u_k v))$
- pick $v \notin \{u_1, u_2, \ldots, u_k\}$ such that *dist*$[v]$ is minimum
- $u_{k+1} \leftarrow v$
- **comment:** *dist*$[u_{k+1}]$ now equals $\text{DIST}(u_1, u_{k+1})$, and $u_1, u_2, \ldots, u_{k+1}$ are now known

Exercises
2.5.1 Prove that Dijkstra's algorithm works. Use induction on the number *k* of iterations to prove that at the beginning of iteration *k,* each *dist[ui]*=DIST(*u*1,*ui*), and that for all *u*≠*ui,* for any *i, dist[u]* equals the length of a shortest *u*1*u*-path using only the vertices {*u*1,*u*2,…,*uk*−1,*u*}. Conclude that after *n*−1 iterations, all distances *dist[u]*=DIST(*u*1,*u*).

Page 37
2.5.2 Assuming that *G* is stored in adjacency lists, and that the minimum *dist[v]* is computed by scanning all *n* vertices, show that the complexity of Dijkstra's algorithm is *O*(*e*+*n*2).

2.6 Data structures

When computing the distances DIST(*u*1, *v*), it would also be a good idea to store a shortest *u*1*v*-path. All the *u*1*v*-paths can easily be stored using a single array
PrevPt[v]: the previous point to *v* on a shortest *u*1v-path.
Initially, *PrevPt[u]*←0. When *dist[v]* and *dist[uk]*+WT*(ukv)* are compared, if the second choice is smaller, then assign *PrevPt[v]*←*uk*. The shortest *u*1*v*-path can then be printed by the following loop:

repeat
output *(v)*
v←*PrevPt[v]*
until *v*=0
The complexity of Dijkstra's algorithm was calculated in Exercise 2.5.2 above as $O(n2+e)$. The term $O(n2)$ arises from scanning up to *n* vertices in order to select the minimum vertex *?*. This scanning can be eliminated if we store the vertices in a partially ordered structure in which the minimum vertex is always readily available. A *heap* is such a structure. In a heap *H,* nodes are stored so that the smallest element is always at the top.
A heap is stored as an array, but is viewed as the partially ordered structure shown above. Its elements are not sorted, but satisfy the *heap property,* namely that *H[i]*=*H*[2*i*] and *H[i]*=*H*[2*i*+1]; that is, the value stored in each node is less than or equal to that of either of its children. Therefore, *H*[1] is the smallest entry in the array.
The heap shown above has depth four; that is, there are four levels of nodes. A heap of depth *k* can contain up to *2k*-1 nodes, so that the depth needed to store *N* values is the smallest value of *k* such that *2k*-1=*N,* namely, $k = \lceil \log(N+1) \rceil$, where the log is to base 2.
If the value stored in a node is changed so that the heap property is no longer satisfied, it is very easy to update the array so that it again forms a heap. For example, if *H*[10] were changed to 4, then the following loop will return *H* to heap form. The movement of data is shown in Figure 2.8.

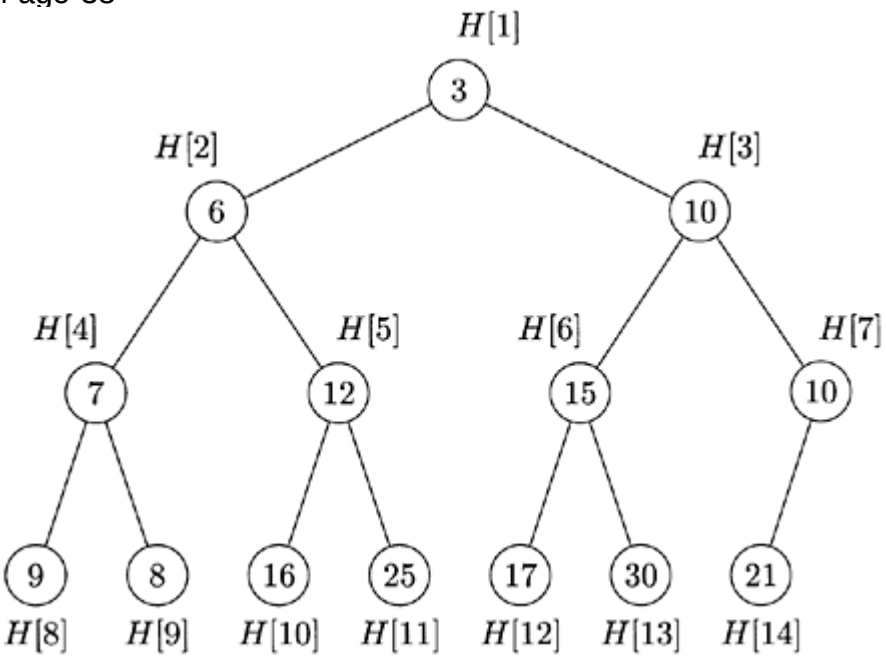


FIGURE 2.7
A heap

procedure FLOATUP*(k)*
comment: Element *H[k]* floats up to its proper place in the heap
temp←*H[k]*
j←*k*/2
while *temp*<*H[j]* **and** *j*>0
do $\begin{cases} H[k] \leftarrow H[j] \\ k \leftarrow j \\ j \leftarrow k/2 \end{cases}$
H[k]←*temp*

Notice the circular movement of data when an altered element floats up to its proper place in the heap. If some entry in the heap were made larger, say *H*[1] became equal to 10, then a similar loop (Procedure FLOATDOWN) would cause the new value to float down to its proper place. Since the depth of a heap containing *N* items is $\lceil \log(N+1) \rceil$, the number of items moved is at most $1+\lceil \log(N+1) \rceil$.

Page 39

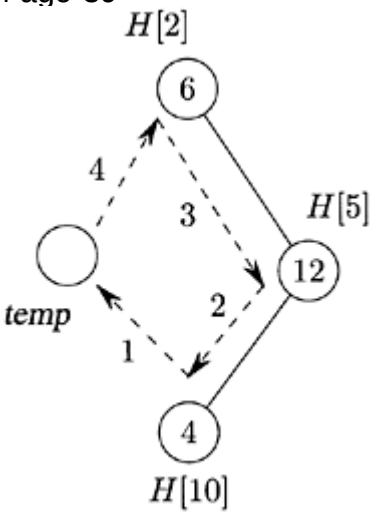


FIGURE 2.8
Updating a heap with FLOATUP.

procedure FLOATDOWN*(k)*

comment: the entry at $H[k]$ has been increased – it now floats down the heap to its correct location. There are currently n entries in the heap, $H[1]$ to $H[n]$. The array $H[\cdot]$ has been dimensioned so that $H[0]$ is also available as a sentinel. $H[0]$ contains a large value bigger than any valid heap entry.

temp←*H[k]*
while *k*+*k*=*n*
do
$i \leftarrow k + k$ "the left child of $H[k]$".
$j \leftarrow i + 1$ "the right child of $H[k]$"
if $j > n$
  then $j \leftarrow 0$ "the sentinel at $H[0]$"
if $H[i] > H[j]$
  then $i \leftarrow j$
comment: $H[i]$ is now the smaller child
if *temp* $\leq H[i]$
  then break "break out of loop"
$H[k] \leftarrow H[i]$
$k \leftarrow i$

H[k]←*temp*

In order to extract the smallest item from a heap of *N* elements, we take its value from *H*[1], and then perform the following steps:

Page 40

H[1]←*H[N]*
N←*N*-1
FLOATDOWN(1)

The new *H*[1] floats down at most $1 + \lceil \log\ N \rceil$ steps.

There are two ways of building a heap.

procedure BUILDHEAPTOPDOWN*(H,N)*

comment: The array H contains N entries transform it into a heap

k←1
while *k*<*N*

do { **comment:** the first k values in H already form a heap
$k \leftarrow k+1$
FLOATUP(k)

Using this method, the elements in entries 1, 2,..., *k* already form a subheap with *k* entries. On each iteration, a new entry is allowed to float up to its proper position so that the first *k*+1 values now form a heap. There are two nodes on level two of the heap. The FLOATUP() operation for each of these may require up to 1+2=3 data items to be moved. On level three there are four nodes. FLOATUP() may require up to 1+3=4 data items to be moved for each one. In general, level *k* contains 2*k*-1 nodes, and FLOATUP() may need up to 1+*k* data items to be moved for each. The total number of steps to create a heap with $d = \lceil \log(N+1) \rceil$ levels in this way is therefore at most

$$S = 3 \cdot 2^1 + 4 \cdot 2^2 + 5 \cdot 2^3 + \ldots + (1+d)2^{d-1} = \sum_{k=2}^{d-1} (1+k)2^{k-1}.$$

Therefore

$$2S = 3.22 + 4.22 + 5.23 + \ldots + (1+d)2d,$$

so that

$$\begin{aligned} 2S - S &= (1+d)2d - 3 \cdot 21 - (22 + 23 + \ldots + 2d - 1) \\ &= (1+d)2d - 5 - (1 + 2 + 22 + 23 + \ldots + 2d - 1) \\ &= (1+d)2d - 5 - (2d - 1) \\ &= d2d - 4 \end{aligned}$$

Thus, it takes *O*(*N* log *N*) steps to build a heap in this way.
The second way of building a heap is to use FLOATDOWN().

---

procedure BUILDHEAPBOTTOMUP*(H,N)*

comment: { The array H contains N entries
transform it into a heap

k←*N*/2
while *k*≥1

do { **comment:** { the substructures at nodes $H[2k]$ and $H[2k+1]$
already form subheaps
FLOATDOWN(k)
$k \leftarrow k-1$

This way is much more efficient, requiring only *O(N)* steps, as is proved in Exercise 2.7.1 below.
We can use a heap *H* to store the values *dist[v]* in Dijkstra's algorithm. The main loop now looks like this.
*u*1←*u* "the nearest vertex to *u.*"
for *k*←1 **to** *n*-1

do { **comment:** $u_1, u_2, \ldots, u_k$ are currently known
for each $v \longrightarrow u_k$ such that $v \notin u_1, u_2, \ldots, u_k$
do { **if** $dist[v] > dist[u_k] + \text{WT}(u_k v)$
then { $dist[v] \leftarrow dist[u_k] + \text{WT}(u_k v)$
FLOATUP(v) "which entry corresponds to v?"
choose u_{k+1} using $H[1]$
$H[1] \leftarrow H[n-k]$
remove last entry from H
FLOATDOWN(1)
comment: $u_1, u_2, \ldots, u_{k+1}$ are now known

Notice that the FLOATUP*(v)* operation requires that we also know which node *H[k]* in the heap corresponds to

the vertex *v,* and vice versa. This can be done with an array mapping vertices into the heap. Let us work out the complexity of Dijkstra's algorithm using this data structure. It is not possible to get an accurate estimate of the number of steps performed in this case, but only an upper bound. The initialization of the heap and *dist*[·] array take *O(n)* steps. The inner for-loop executes a total of at most 2*e* if-statements, so that at most 2*e* FLOATUP()'s are performed, each requiring at most $1 + \lceil \log(n+1) \rceil$ steps. There are also *n*-1 FLOATDOWN()'S performed. Thus the complexity is now

$$O((2\varepsilon + n)(1 + \lceil \log(n+1) \rceil)) = O(\varepsilon \log\ n).$$

This may be better or worse that the previous estimate of $O(n2)$ obtained when the minimum vertex is found by scanning up to *n* vertices on each iteration. If the graph has few edges, say $\Sigma \leq \Delta n/2$, where the maximum degree? is some fixed constant, or a slowly growing function of n, then Dijkstra's algorithm will certainly be much more efficient when a heap is used. Furthermore, it must be remembered that the complexity estimate using the heap is very much an *upper bound,* whereas the other method will always take at least $O(n2)$ steps. If the number of edges is large, say $\Sigma=O(n2)$, then the heap-version of Dijkstra's algorithm can spend so much time keeping the heap up-to-date, that no increase in efficiency is obtained.

2.7 Floyd's algorithm

Floyd's algorithm solves the All Paths Problem, computing a matrix of values *Dist[u,v]*=DIST*(u,v),* for all *u, v* $v \in V(G)$. Initially, *Dist*[·,·] equals the weighted adjacency matrix *A,* where

$$A[u,v] = \begin{cases} \text{WT}(u,v), & \textbf{if } u \longrightarrow v, \\ \infty, & \textbf{if } u \not\longrightarrow v, \\ 0, & \textbf{if } u = v. \end{cases}$$

Floyd's algorithm is extremely simple to program.

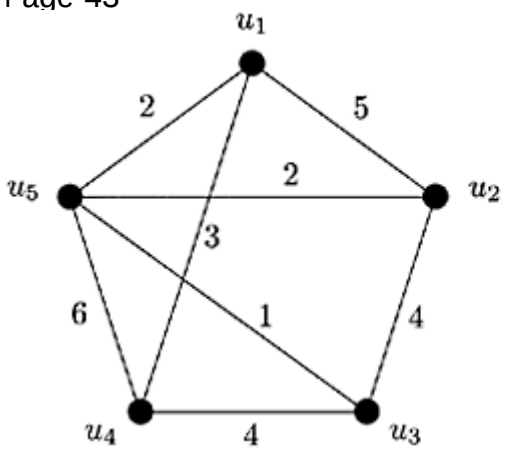
procedure FLOYD*(Dist)*
comment: *Dist[u,v]* will equal DIST*(u,v),* upon completion
for *k*←1 **to** *n*

$$\textbf{do} \begin{cases} \textbf{for } v \leftarrow 1 \textbf{ to } n-1 \\ \quad \textbf{do} \begin{cases} \textbf{for } w \leftarrow v+1 \textbf{ to } n \\ \quad \textbf{do } Dist[v,w] \leftarrow \text{MIN}(Dist[v,w], Dist[v,u_k] + Dist[u_k,w]) \end{cases} \end{cases}$$

The for-loops for *v* and *w* together examine $\binom{n}{2}$ pairs *vw* for each value of *u,* so the complexity of the algorithm is

$$n\binom{n}{2} = \frac{1}{2}n^3 - \frac{1}{2}n^2 = O(n^3).$$

The graph is stored as a weighted adjacency matrix, in which non-adjacent vertices *v, w* can be considered to be joined by an edge of weight ∞. Figure 2.9 shows a weighted graph on which the reader may like to work Floyd's algorithm by hand.



	u_1	u_2	u_3	u_4	u_5
u_1	0	5	∞	3	2
u_2	5	0	4	∞	2
u_3	∞	4	0	1	4
u_4	3	∞	1	0	6
u_5	2	2	4	6	0

FIGURE 2.9

A complete weighted graph and its weighted adjacency matrix.
Let the vertices of *G* be named *u*1, *u*2,..., *un*. In order to prove that Floyd's algorithm works, we prove by induction, that at the end of kth iteration of the for-loop for *u, Dist[v,w]* is the length of the shortest *uw*-path which uses only vertices *u,w,* and *u*1, *u*2,..., *uk*. When *k*=0, that is, before the first iteration, *Dist[u,w]* is the length of the edge *uw,* that is, the length of the shortest path using only vertices *u* and *w*. At the end of the first iteration, *Dist[u,w]*=MIN(WT*(u,w),* WT(*u,u*1)+WT(*u*1,*w*)). This is the length of the shortest *uw*-path using only vertices *u, w,* and *u*1, since that path either uses *u*1, or else consists only of the edge *uw*. Thus, the statement is true when *k*=1.

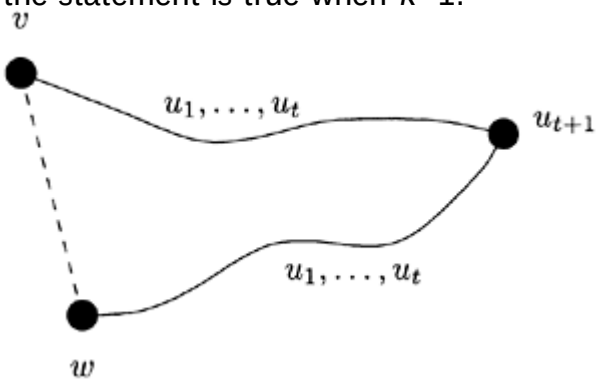


FIGURE 2.10
A path via*ut*+1.
Assume that it is true whenever *k*≤*t,* and consider iteration *t*+1. At the end

of the iteration, each

Dist[v, w]=MIN(*Dist[v, w],Dist*[*v, ut*+1]+*Dist*[*ut*+1,*w*]).

If the shortest *vw*-path using only vertices *v, w, u*1, *u*2,..., *ut*+1 does not use *ut*+1, then its length is the previous value of *Dist[v, w]* from iteration *t*. If the path does use *ut*+1, then the length is given by the second term above. Therefore, at the end of the iteration, the value of *Dist[v, w]* is as required. By induction, it follows that at the end of the *n*th iteration, *Dist[v, w]*=DIST*(v, w),* for all *v* and *w*. Floyd's algorithm finds all distances in the graph. It always takes $n\binom{n}{2} = O(n^3)$ steps, irrespective of the number of edges of *G*. When there are few edges, it is faster to use Dijkstra's algorithm *n* times, once for every starting vertex *u*. This gives a complexity of *O(en* log *n*), using a heap, which can be less than *O*(*n*3).

Exercises

2.7.1 Calculate the number of steps needed to construct a heap using the BUILDHEAPBOTTOMUP() procedure.

2.7.2 The repeat-loop of the FLOATUP() procedure described above requires *k*+2 data items to be moved when an entry floats up *k* nodes in the heap. If FLOATUP() is programmed by swapping adjacent elements instead of moving them in a cycle, calculate the number of items moved when an entry floats up *k* nodes. Which is more efficient?

2.7.3 The type of heap discussed above is called a *binary heap,* since each node *H[k]* has two children, *H*[2*k*] and *H*[2*k*+1]. The depth of a binary heap with *N* elements is $\lceil \log(N+1) \rceil$. In a *ternary heap,* node *H[k]* has three children, *H*[3*k*], *H*[3*k*+1], and *H*[3*k*+2]. What is the depth of a ternary heap with *N* nodes? Calculate the number of steps needed to construct it using the BUILDHEAPBOTTOMUP() procedure.

2.7.4 Program Dijkstra's algorithm using a binary heap.

2.7.5 Show how to store a complete set of shortest paths in Floyd's algorithm, using a matrix *PrevPt[v, w],* being the previous point to *v* on a shortest *vw*-path. What should the initial value of *PrevPt[v, w]* be, and how and when should it be modified?

2.7.6 Ford's algorithm. Consider the following algorithm to find DIST*(u, v),* for a given vertex $u \in V(G)$ and all vertices $v \in V(G)$.

procedure FORD*(u)*
for each $v \in V(G)$
do *dist[v]*←∞
dist[u]←0
while there is an edge *vw* such that *dist[w]*>*dist[v]*+WT*[vw]*
do *dist[w]*←*dist[v]*+WT*[vw]*

Page 45
Prove that Ford's algorithm correctly computes DIST*(u, u)*. What data structures are necessary for an efficient implementation of Ford's algorithm? Analyze the complexity of Ford's algorithm. Give a numerical estimate of the number of steps, as well as a formula of the form $O(\cdot)$.

2.8 Notes

WEISS [122] contains an excellent treatment of the merge-find data structure and heaps. Dijkstra's shortest-path algorithm and Floyd's algorithm are described in most books on algorithms and data structures.

Page 46
This page intentionally left blank.

Page 47
3

Some Special Classes of Graphs

3.1 Bipartite graphs

A graph *G* is said to be *bipartite* if *V(G)* can be divided into two sets *X* and *Y* such that each edge has one end in *X* and one end in *Y*. For example, the cube is a bipartite graph, where the bipartition *(X,Y)* is illustrated by the coloring of the nodes in Figure 3.1.

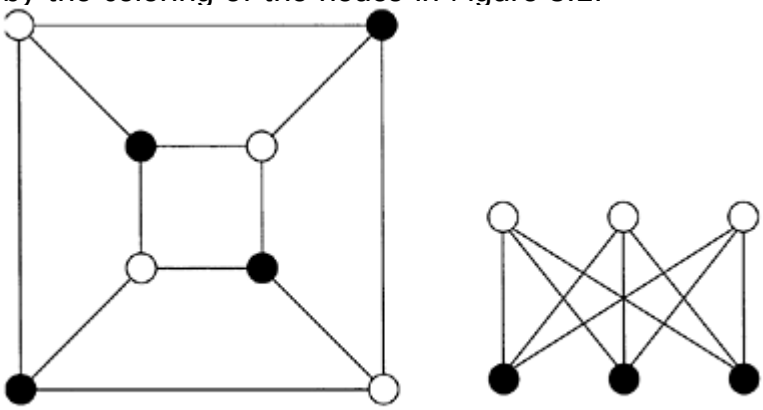


FIGURE 3.1
Two bipartite graphs

The maximum number of edges in a simple bipartite graph in which *X* and *Y* are the two sides of the bipartition is clearly $|X|\cdot|Y|$. The *complete bipartite* graph *Km,n* has $|X|=m$, $|Y|=n$, and $\varepsilon=mn$. For example, *K*3,3 is illustrated in Figure 3.1.

Page 48
LEMMA 3.1 A simple, bipartite graph G has at most $|G|2/4$ *edges*.

PROOF Let *G* have bipartition *(X,Y),* where $|X|=x$ and $|Y|=n-x$, where $n=|G|$. Then $\varepsilon\le x(n-x)=nx-x2=n2/4-(n/2-x)2\le n2/4$.

If $C=(x1, y1, x2, y2,\ldots)$ is a cycle in a bipartite graph *G,* then consecutive vertices of *C* must be alternately in *X* and *Y,* the two sides of the bipartition. It follows that *ℓ(C)* is even. In fact, any graph in which all cycles have even length must be bipartite.

THEOREM 3.2 G is bipartite if and only if all cycles of G have even length.

PROOF Let *G* be a connected graph in which all cycles have even length. Pick any $x \in V(G)$ and set $X=\{u:$ DIST*(x,u)* is even$\}$, and $Y=V(G)-X$. Clearly *X* and *Y* partition *V(G)* into two parts. We must show that there are no edges with both endpoints in *X* or *Y*. Suppose that uu is an edge with $u,v \in X.$ Let *Pu* be a shortest *xu*-path, that is, a path of length DIST*(x,u),* and let *Pu* be a shortest *xu*-path. Then *ℓ(Pu)* and *ℓ(Pu)* are both even. Say $\ell(Pu)\le\ell(Pu)$. *Pu* and *Pu* both begin at point *x*. They do not both contain *u,* or *Puuu* would be a shortest *xu*-path of length *ℓ(Pu)*+1, an odd number. So let *z* be the last point in common to *Pu* and *Pu*. This defines the cycle *C=Pu[z,u]uuPu[u,z]*. Here *Pu[z,u]* denotes the portion of *Pu* from *z* to *u* and *Pu[u,z]* denotes

the portion of *Pu* from *u* to *z*. The length of *C* is then $\ell(Pu[z,u])+\ell(Pu[u,z])+1 = \ell(Pu)+\ell(Pu)-2\mathrm{DIST}(x,z)+1$, which is odd, a contradiction. Therefore no edge *uu* has both endpoints in *X*. Similarly, no edge *uu* has both endpoints in *Y*. Since a graph is bipartite if and only if every component is bipartite, this completes the proof.

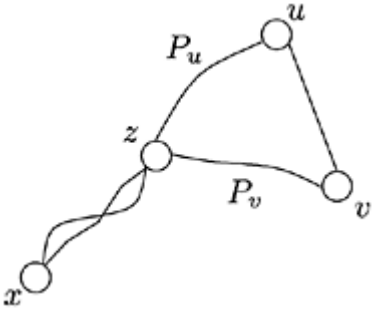


FIGURE 3.2
Two paths in a bipartite graph

Page 49

LEMMA 3.3 If G is a k-regular bipartite graph, where k>0, *with bipartition (X,Y), then*|*X*|=|*Y*|.

PROOF Since each edge has one end in *X,* we can write $\varepsilon = \Sigma_{x\in X}\mathrm{DEG}(x) = k\cdot|X|$. Similarly, $\varepsilon = \Sigma_{y\in Y}\mathrm{DEG}(y) = k\cdot|Y|$. Therefore *k*·|*X*|=*k*·|*Y*|. Since *k*>0, it follows that |*X*|=|*Y*|.

Exercises

3.1.1 The *k-cube Qk* is a graph whose vertex set consists of all binary vectors of length *k:*

$$V(Q_k) = \{(a_1, a_2, \ldots, a_k) : a_i \in \{0,1\}\}$$

Thus there are *2k* vertices. The edges of *Qk* are formed by joining two vertices $\overrightarrow{a} = (a_1, a_2, \ldots, a_k)$ and $\overrightarrow{b} = (b_1, b_2, \ldots, b_k)$ if $\overrightarrow{a}$ and $\overrightarrow{b}$ differ in exactly one coordinate, that is, *ai*=*bi* for all *i* but one. *Q*3 is displayed in Figure 3.3. Prove that *Qk* is bipartite. Describe a bipartition of *Qk*.

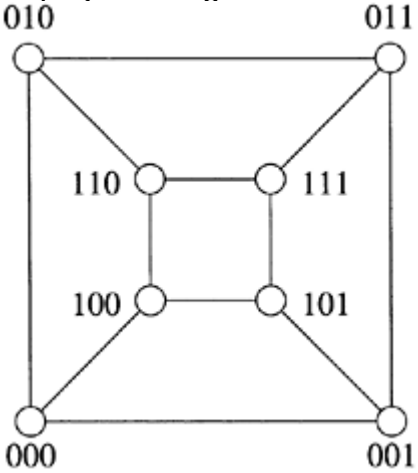


FIGURE 3.3
The 3-cube, Q3

3.1.2 Prove that $\varepsilon(Qk) = k2k-1$.

3.1.3 Describe in pseudo-code an algorithm to find a bipartition of *G,* or to determine that *G* is not bipartite. Describe the data-structures needed, and calculate the complexity (should be $O(\varepsilon)$).

3.1.4 Let *G* be a bipartite simple graph with bipartition *(X,Y)* and n vertices. Let δx be the minimum degree among the vertices of *X,* and δY be the minimum degree

Page 50

among the vertices of *Y*. Show that if $\delta X+\delta Y > n/2$, then *G* is connected, where $\delta X, \delta Y > 0$.

3.2 Line graphs

Two edges of a graph *G* are adjacent if they share a common endpoint. The *linegraph* of *G* is a graph *L(G)* which describes the adjacencies of the edges of *G*. Thus, every vertex of *L(G)* corresponds to an edge *uu* of *G,* so that $|L(G)| = \varepsilon(G)$. This is illustrated in Figure 3.4.

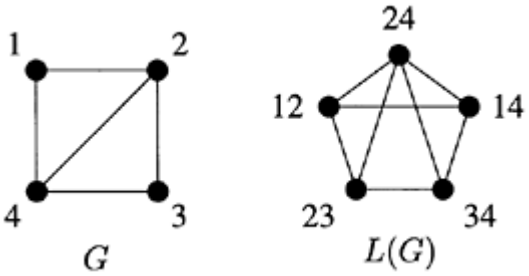


FIGURE 3.4
Constructing a line-graph

A line-graph can always be decomposed into complete subgraphs. For a vertex $v \in V(G)$ lies on DEG*(u)* distinct edges all of which share the endpoint *u*. The DEG*(u)* corresponding vertices of *L(G)* form a complete subgraph containing $\binom{\text{DEG}(u)}{2}$ edges. Every edge of *L(G)* is contained in exactly one such complete subgraph.

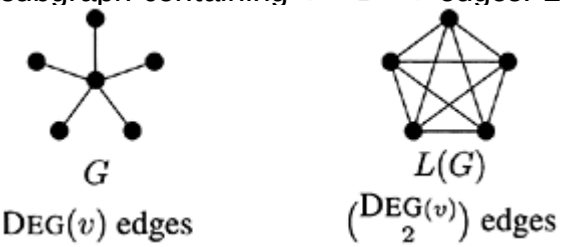


FIGURE 3.5
Complete subgraph in a line-graph

This gives the following theorem:

$$\varepsilon(L(G)) = \sum_{u \in V(G)} \binom{\text{DEG}(u)}{2}$$

THEOREM 3.4

Exercises

3.2.1 Find the line-graph of the cube.

3.2.2 Construct $\overline{L(K_5)}$ and show that it is isomorphic to the Petersen graph.

3.2.3 Let *G* be any graph. If we insert a vertex of degree two into each edge, we obtain a new graph *S(G),* called the *subdivision graph* of *G*. For example, *S*(*K*4) is illustrated in Figure 3.6. Prove that *S(G)* is always bipartite, and find a formula for *ε(S(G))*.

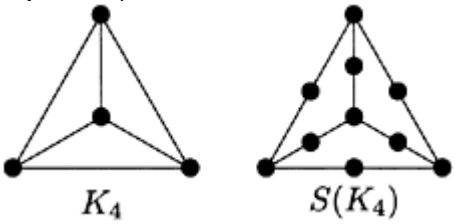


FIGURE 3.6
Subdivision graph of *K*4

3.2.4 The graph *P* in Figure 3.7 is called the 3-prism. Find the line-graphs of the subdivision graphs of *K*4 and *P*. Draw them as neatly as possible. What can you say in general about constructing the line-graph of the subdivision graph of a 3-regular graph?

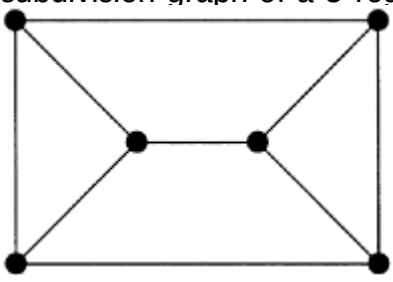


FIGURE 3.7
The 3-prism

3.2.5 We know that

$$\sum_u \text{DEG}(u) = 2\varepsilon(G),$$

Page 52
and that

$$\sum_u \binom{\text{DEG}(u)}{2} = \varepsilon(L(G)).$$

Can you find a similar way of interpreting

$$\sum_u \binom{\text{DEG}(u)}{3}?$$

Assume first that there are no triangles in *G*.
3.2.6 Suppose that a graph *G* is represented by its adjacency matrix. Write a program to print out the adjacency lists of *L(G),* but do not store either adjacency lists or an adjacency matrix for *L(G);* just print it out. Also print out a list of the edges of *G,* in order to give a numbering to the vertices of *L(G)*.
3.2.7 Notice that $L(K_3) \cong L(K_{1,3}) \cong K_3$ (see Figure 3.8). Prove that if *G* and *H* are any other graphs, then $G \cong H$ if $L(G) \cong L(H)$.

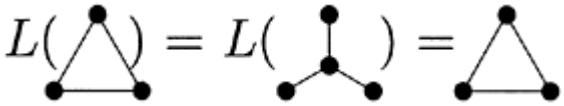


FIGURE 3.8
Two graphs with isomorphic line-graphs
3.3 Moore graphs
The length of the shortest cycle in a graph *G* is called its *girth,* denoted γ*(G)*. For example, the cube has girth four. Graphs with fixed degree *k* and fixed girth often have interesting properties. For example, let *G* be a *k*-regular graph of girth four, and pick any vertex *u* in *G*. There are *k* vertices at distance one from *u*. Since *G* has no triangles, there are at least *k*−1 vertices at distance two from *u,* as shown in Figure 3.9. Therefore, |*G*|≥1+ *k*+(*k*−1)=2*k*. There is only one such graph with |*G*|=2*k,* and that is the complete bipartite graph *Kk, k*.
Now let *G* be a *k*-regular graph of girth five, and let *u* be any vertex, There are *k* vertices at distance one from *u*. Since *G* has no 4-cycles, each point at distance one is adjacent to *k*−1 more vertices at distance two, so that |*G*|≥ 1+*k*+*k*(*k*−1)=*k*2+1.
Problem. Are there any *k*-regular graphs *G* of girth five with |*G*|=*k*2+1?
These graphs are called *Moore graphs*. Let *n*=|*G*|. A l-regular graph cannot have γ=5, so *k*≥2. If *k*=2, then *n*=22+1=5. *G* is a cycle of length five. This is the unique Moore graph of degree two.

Page 53

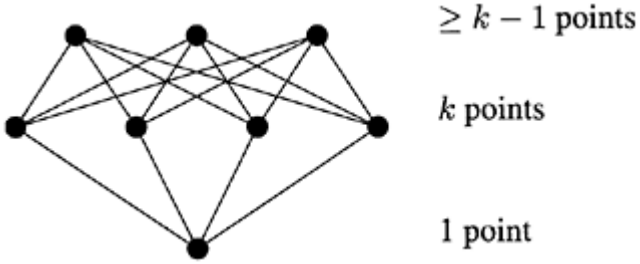


FIGURE 3.9
Kk,k
If *k*=3, then *n*=32+1=10. There are three vertices at distance one from *u,* and six at distance two, as illustrated in Figure 3.10. Consider vertex *u*6. $u_6 \not\to u_5$, since this would create a triangle, whereas γ=5. Without loss of generality, we can take *u*6→*u*8. Were we now to join *u*6→*u*7, this would create a 4-cycle (*u*6, *u*7, *u*3, *u*8), which is not allowed. Therefore, without loss of generality, we can take *u*6→*u*9. This is shown in Figure 3.10.

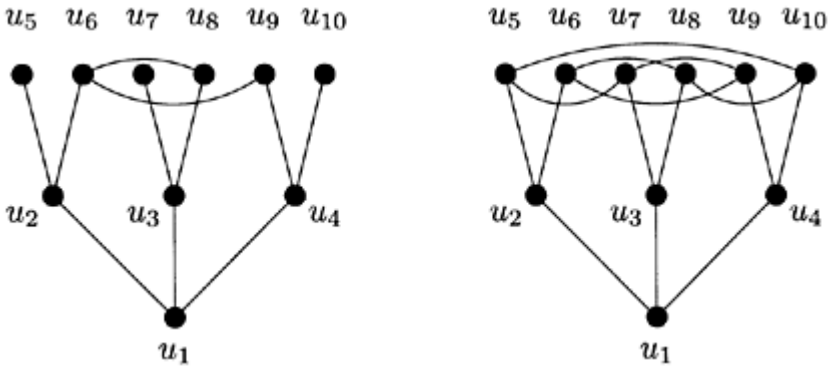


FIGURE 3.10
A Moore graph of degree three

There is now only one way of completing the graph so as to maintain γ=5. Vertex *u*9 cannot be joined to *u*5, *u*8, or *u*10. Therefore *u*9→*u*7. Similarly *u*8→*u*10, etc. The completed graph is shown in Figure 3.10, and has been redrawn in Figure 3.11 (check that this is the same graph).

Thus, we have proved the following.

THEOREM 3.5 The Petersen graph is the unique Moore graph of degree three.

There is a very elegant theorem proving that Moore graphs can exist only for

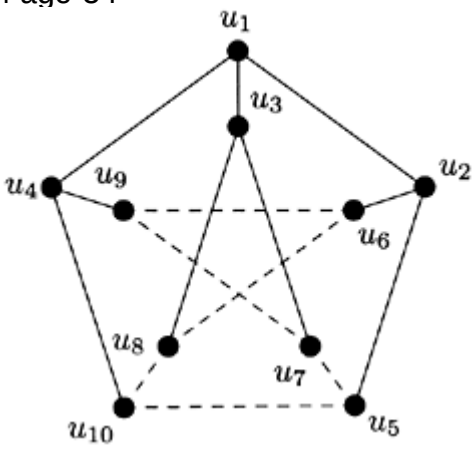


FIGURE 3.11
The Petersen graph

special values of *k*.

THEOREM 3.6 A Moore graph of degree k can exist only if k=2, 3, 7, *or 57*.

PROOF Let *G* be a Moore graph with adjacency matrix *A* and consider *A*2. Entry *[i,j]* of *A*2 is the number of uiuj-paths of length two. If $u_i \longrightarrow u_j$, then there is no 2-path from *ui* to *uj,* since γ=5. Therefore, *[A2]ij*=0 if *[A]ii*=1.

If $u_i \not\longrightarrow u_j$ then DIST*(ui,uj)*>1. It is shown in Exercise 3.3.3 that DIST*(ui,uj)* is always at most 2.

Therefore, if $u_i \not\longrightarrow u_j$ there must be a 2-path connecting *ui* to *uj*. There cannot be two such 2-paths, for that would create a 4-cycle containing *ui* and *uj*. Therefore, [*A*2]*ij*=1 if *[A]ij*=0.

It follows that the matrix *A*2+*A* consists of all 1's off the diagonal. The diagonal elements all equal *k,* the degree of *G*. The number of vertices is *n*= *k*2+1.

$$A^2 + A = \begin{bmatrix} k & & 1 \\ & k & \\ 1 & & \ddots \\ & & & k \end{bmatrix}_{n \times n}$$

We can find the eigenvalues of this matrix. Write *B*=*A*2+*A*. If x is an eigenvector of *A* with eigenvalue *α*, then

$$Bx=(A2+A)x=AAx+Ax=\alpha Ax+\alpha x=(\alpha 2+\alpha)x$$

so that *β*=*α*2+*α* is an eigenvalue of *B*. To find the eigenvalues of *B,* we solve det(*λI*−*B*)=0 for *λ*.

Page 55

$$\det(\lambda I - B) = \begin{bmatrix} \lambda - k & & & -1 \\ & \lambda - k & & \\ & -1 & \ddots & \\ & & & \lambda - k \end{bmatrix}_{n \times n}$$

Adding rows 2 to *n* onto row 1 gives

$$\begin{vmatrix} \lambda - k - n + 1 & \lambda - k - n + 1 & \cdots & \\ -1 & \lambda - k & & \\ & & \ddots & \\ & & & \lambda - k \end{vmatrix}_{n \times n}$$

$$= (\lambda - k - n + 1) \begin{vmatrix} 1 & 1 & \cdots & \\ -1 & \lambda - k & & \\ & & \ddots & \\ & & & \lambda - k \end{vmatrix}_{n \times n}$$

Now add the first row to each row to get

$$(\lambda - k - n + 1) \begin{vmatrix} 1 & 1 & \cdots & \\ 0 & \lambda - k + 1 & & \\ & & \ddots & \\ & & & \lambda - k + 1 \end{vmatrix}_{n \times n}$$

$$= (\lambda - k - n + 1)(\lambda - k + 1)^{n-1} = 0.$$

Therefore the eigenvalues of *B* are

$$\lambda = \begin{cases} \beta_1 = k + n - 1 & \text{(once)}, \\ \beta_2 = k - 1 & (n - 1 \text{ times}). \end{cases}$$

Since $\beta = \alpha 2 + \alpha$, we can solve for $\alpha = \frac{1}{2}(-1 \pm \sqrt{1 + 4\beta})$. Should we take the plus or minus sign? Since $n = k2 + 1$, the value $\beta 1 = k2 + k + 1$ gives

$$\alpha = \frac{1}{2}(-1 \pm \sqrt{4k^2 + 4k + 1}) = \frac{1}{2}\{-1 \pm (2k + 1)\} = k \text{ or } -k - 1.$$

Now $\beta 1$ occurs only once as an eigenvalue, so we must choose only one of these. *G* is *k*-regular, so that the rows of *A* all sum to *k*. Thus, if x is the vector of all 1's, then $Ax = kx$, so that *k* is in fact an eigenvalue of *A*. Consider now $\beta 2$. The corresponding eigenvalues of *A* are

---

Page 56

$$\alpha = \begin{cases} \alpha_1 = \frac{1}{2}(-1 + \sqrt{4k - 3}) & (m_1 \text{ times}), \\ \alpha_2 = \frac{1}{2}(-1 - \sqrt{4k - 3}) & (m_2 \text{ times}). \end{cases}$$

The total multiplicity is $m1 + m2 = n - 1 = k2$. Since the trace of *A,* that is, the sum of its diagonal elements, also equals the sum of its eigenvalues, we can write

$$m1 + m2 = k2 \text{ (sum of multiplicities)}$$
$$a1m1 + a2m2 + k = 0 \text{ (sum of eigenvalues)}$$

Solving these equations for $m1$ and $m2$ gives

$$m_1 = \frac{-1}{2}\left\{\frac{-k^2 + 2k}{\sqrt{4k - 3}} - k^2\right\}$$

and

$$m_2 = \frac{1}{2}\left\{\frac{-k^2 + 2k}{\sqrt{4k - 3}} + k^2\right\}$$

The multiplicities *m*1 and *m*2 are integers. Consider the fraction

$$\frac{-k^2+2k}{\sqrt{4k-3}}.$$

If *k*=2, the numerator is 0. If *k*≠2, then $\sqrt{4k-3}$ must be an integer, so that 4*k*−3 is a perfect square, say 4*k*−3=*s*2. Then

$$k=\frac{1}{4}(s^2+3),$$

and

$$-k^2+2k=\frac{1}{16}(-s^4+2s^2+15).$$

This expression must be divisible by $\sqrt{4k-3}=s$. If *s* does not divide 15, it cannot be an integer, since the other 2 terms have no *s* in the denominator. Therefore *s*=1,3,5, or 15. The corresponding values of *k, m*1, *m*2, *a*1, and *a*2 are shown in the following table:

s	*k*	*n*	*m*1	*m*2	*a*1	*a*2
1	1	2	1	0	0	−1
3	3	10	4	5	1	−2
5	7	50	21	28	2	−3
15	57	3250	1520	1729	7	−8

Page 57

The value *k*=1 does not correspond to a graph. *k*=3 gives the Petersen graph. There is a unique Moore graph with *k*=7 and *n*=50, called the Hoffman-Singleton graph. It is not known whether a Moore graph with *k*=57 and *n*=3250 exists. The 5-cycle is a Moore graph with *k*=2. Its eigenvalues are

$$\alpha_1=\frac{1}{2}(-1+\sqrt{5})$$

and

$$\alpha_2=\frac{1}{2}(-1-\sqrt{5}),$$

with multiplicities *m*1=*m*2=2.

The *diameter* of a graph is the maximum distance between any two vertices,

$$\operatorname{diam}(G)=\max\{\textsc{Dist}(u,v):u,v\in V(G)\}.$$

Thus, Moore graphs have diameter two.

Exercises

3.3.1 Let *G* be a Moore graph of degree *k,* with *n*=*k*2+1 vertices. Let *u* be any vertex of *G*. Prove that there are exactly

$$\frac{k(k-1)^2}{2}$$

pentagons containing *u*. Conclude that *G* contains

$$\frac{k(k^2+1)(k-1)^2}{10}$$

pentagons, so that $k\not\equiv 4 \pmod 5$.

3.3.2 Let *G* be as above. Prove that every $v\in V(G)$ is contained in exactly

$$\frac{k(k-1)^2(k-2)}{2}$$

hexagons and in

$$\frac{k(k-1)^2(k-2)(k-3)}{2}$$

heptagons.

3.3.3 Show that in a *k*-regular graph of girth five, with *n*=*k*2+1 vertices, the distance DIST*(u, u)* between any two vertices is at most two. *Hint:* Show that DIST*(u, u)*= 3 implies the existence of a 4-cycle.

Page 58

3.4 Euler tours

Figure 3.12 shows a drawing of $K5$ illustrating a walk in which each edge is covered exactly once.

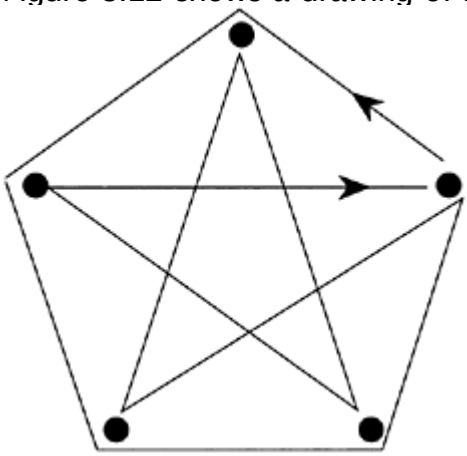


FIGURE 3.12
A traversal of $K5$

A walk which covers each edge of a graph G exactly once is called an *Euler trail* in G. A closed Euler trail is called an *Euler tour*. The example above shows that $K5$ has an Euler tour. Therefore we say that $K5$ is *Eulerian*. It is easy to prove that a graph is Eulerian when all its degrees are even.

THEOREM 3.7 A connected graph G has an Euler tour if and only if all degrees of G are even.

PROOF Let W be a closed Euler trail in G, beginning at vertex u. Each time that W enters a vertex u, it also must exit it. Therefore W uses an even number of edges at each vertex $u \neq u$. Since the trail is closed, the same is true of u. Since W covers every edge of G exactly once, all degrees must be even.

Conversly suppose that all degrees of G are even. The proof is by induction on the number of edges of G. The smallest connected graphs with even degrees are $K1$ and $K3$, and both of these are Eulerian (for $K1$, $W=\emptyset$ is an Euler trail). If the theorem is not true, let G be the smallest graph (i.e., smallest ε) with even degrees with no closed Euler trail. Clearly $\delta(G) \geq 2$, so that G contains a cycle, which is an Eulerian subgraph. Let C be the largest Eulerian subgraph which G contains. Then $\varepsilon(C) < \varepsilon(G)$. The complementary subgraph $G-C$ also has even

degrees, and since it is smaller than G, each component of it must be Eulerian. Furthermore, C intersects each component of $G-C$. We can now make an Euler trail in G from C, by inserting into C Euler trails of each component K of $G-C$, as the walk in C reaches each K in turn. Therefore G is Eulerian. By induction, all connected graphs with even degrees are Eulerian.

Notice that this theorem is true for multigraphs as well as simple graphs. If a connected graph G has exactly two vertices, u and u, of odd degree, then we can add an extra edge uu to G to get G', which will then have all even degrees. G' may now have multiple edges. If we now choose an Euler tour W in G' beginning at u, we can number the edges so that the new edge uu is the last edge traversed. Then $W-uu$ will be an Euler trail in G beginning at u and ending at u. This is illustrated in Figure 3.13.

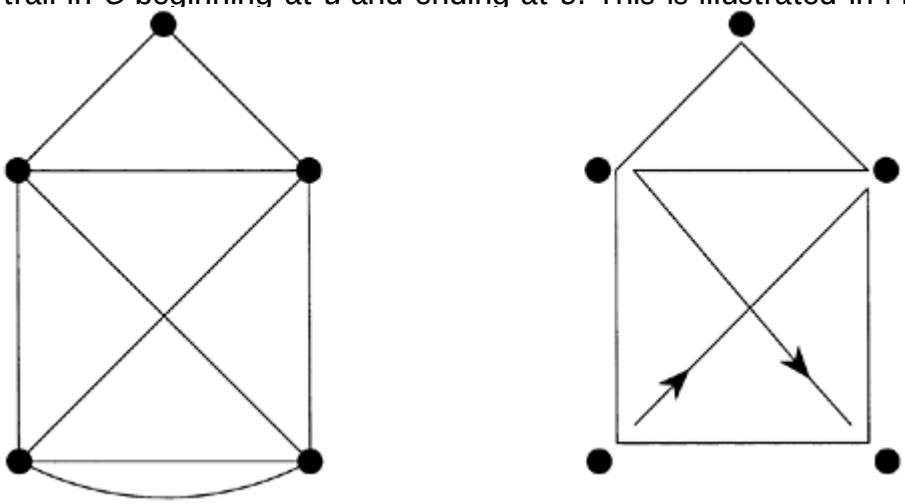


FIGURE 3.13

An Euler trail

If there are more than two vertices of odd degree, then it is clear that *G* cannot have an Euler trail.

3.4.1 An Euler tour algorithm

The proof of Theorem 3.7 is essentially an algorithm to find an Euler tour in a connected graph *G*. The algorithm works by building a walk from a starting vertex *u*. It takes the first edge *e*0=*uu* incident on *u* and follows it. Then it takes the first edge *e*1 incident on *u* and follows it, and so forth. Because the degrees are all even, it must eventually return to *u*. At this point, it will have found a closed walk in *G* that is a sub-tour of an Euler tour. All the vertices visited in the

Page 60

sub-tour are stored on an array called the *ScanQ*. It then repeats this process at *u,* finding another sub-tour. The sub-tours are then linked together to create one sub-tour. It continues like this until all the edges at *u* have been used up. It then moves to the next vertex on the *ScanQ* and builds a sub-tour there, always linking the sub-tours found into the existing tour. When the algorithm completes, all the vertices of *G* are in the *ScanQ* array, because *G* is connected. Therefore we have an Euler tour.

The Euler tour is stored as a linked list of edges. This makes it easy to insert a sub-tour into the list at any location. If *e*=*uu* is an edge of *G,* then we write $nextEdge\langle e\rangle$ and $prevEdge\langle e\rangle$ for the next and previous edges in a tour, respectively. The adjacency list for vertex *u* is denoted by *Graph[u]*. This is a linked list of incident edges.

When the algorithm begins building a sub-tour at vertex *u,* it needs to know an edge at *u* currently in the Euler tour, if there is one. This is stored as *EulerEdge[u]*. It allows the algorithm to insert a sub-tour into the existing tour at that location in the linked list.

Algorithm 3.4.1 is very efficient. For each vertex *u,* all incident edges are considered. Each edge is linked into the Euler tour. This takes DEG*(u)* steps. Several sub-tours at *u* may be linked into the Euler tour being constructed. There are at most DEG*(u)*/2 sub-tours at *u*. If follows that the complexity is determined by

$$\sum_u \textsc{Deg}(u) = 2\varepsilon(G) = O(\varepsilon).$$

Page 61

Algorithm 3.4.1: EULERTOUR*(G)*
comment: Construct an Euler tour in *G*
ScanQ[1]←1
QSize←1
k←1
while *k*≤*QSize*

```
u ← ScanQ[k]
while Graph[u] ≠ null
    e0 ← Graph[u]        "first edge at u"
    v ← other endpoint of e0
    remove edge e0 from the graph
    e1 ← e0
    while v ≠ u
        if v ∉ ScanQ
          then { QSize ← QSize + 1
                 ScanQ[QSize] ← v
        e2 ← Graph[v]        "first edge at v"
        nextEdge⟨e1⟩ ← e2
      do { prevEdge⟨e2⟩ ← e1
        if EulerEdge[v] = null
          then EulerEdge[v] ← e2
        e1 ← e2
    do  { v ← other endpoint of e1
        remove edge e1 from the graph
    comment: a sub-tour at u has just been completed
    prevEdge⟨e0⟩ ← e1
    nextEdge⟨e1⟩ ← e0
    if EulerEdge[u] = NULL
      then EulerEdge[u] ← e0
      else { comment: { insert the sub-tour at u into the
                        existing tour at u
           { e1 ← EulerEdge[u]
           { e2 ← prevEdge⟨e1⟩
           { e3 ← prevEdge⟨e0⟩
           { nextEdge⟨e2⟩ ← e0
           { nextEdge⟨e3⟩ ← e1
k ← k + 1
```


Page 62

Exercises

3.4.1 Program Algorithm 3.4.1 EULERTOUR*(G)*.

3.4.2 Let *G* be a connected graph in which 2*k* of the vertices are of odd degree. Show that there are *k* trails *W*1, *W*2,..., *Wk* such that, taken together, *W*1, *W*2,..., *Wk* cover each edge of *G* exactly once.

3.4.3 Let *W* be an Euler tour in *G*. To what subgraph of the line-graph *L(G)*, does *W* correspond?

3.4.4 Show that any Euler tour of a graph *G* can be written as a union of cycles.

3.4.5 What does the following algorithm do when input a connected graph *G*? What is its complexity?

procedure TESTGRAPH*(G)*

ScanQ[1]←1

QSize←1

Tag[1]←1

k←1

while *k*≤*QSize*

```
do { for all v ⟶ u
     do { if v ∉ ScanQ
          then { QSize ← QSize + 1
                 ScanQ[QSize] ← v
                 Tag[v] ← −Tag[u]
          else if Tag[v] = Tag[u]
          then return ( false )
   k ← k + 1
return (true)
```

3.5 Notes

The theorem on Moore graphs is due to HOFFMANN and SINGLETON [63]. The application of algebraic methods to graph theory is treated in BIGGS [9] and GODSIL and ROYLE [52]. The eigenvalues of graph adjacency matrices is a vast topic. See the surveys by HOFFMAN [62], SCHWENK and WILSON [107], or the book by CVETKOVIC, DOOB, and SACHS [30]. An excellent description of Euler tour algorithms can be found in GOULD [53].

Page 63

4

Trees and Cycles

4.1 Introduction

A *tree* is a connected graph that has no cycles. Figure 4.1 shows a number of trees.

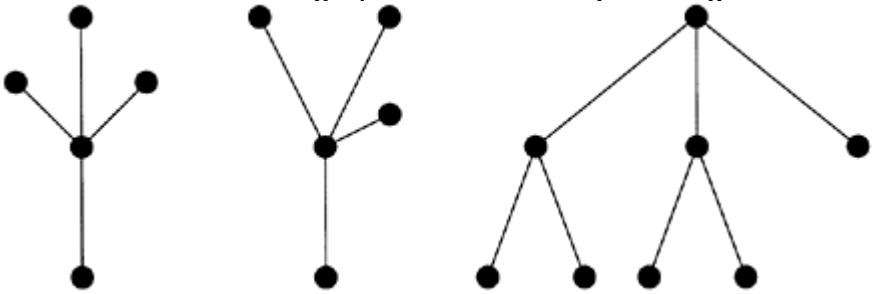


FIGURE 4.1
Several trees

Trees are the smallest connected graphs; remove any edge from a tree and it becomes disconnected. As well as being an important class of graphs, trees are important in computer science as data structures, and as objects constructed by search algorithms. A fundamental property of trees is that all trees on n vertices have the same number of edges.

THEOREM 4.1 If G is a tree, then $\varepsilon(G)=|G|-1$.

PROOF The proof is by induction on $|G|$. If $|G|=1$, then $G=K1$, which is a connected graph with no cycle, so that $\varepsilon=0$. Similarly, if $|G|=2$, then $G=K2$, which has $\varepsilon=1$. Assume that the result is true whenever $|G|\leq t$, and

Page 64

consider a tree G with $|G|=t+1$. Now G must have a vertex of degree one, or it would contain a cycle, so let $v \in V(G)$ have degree one. Then $G'=G-u$ is still connected, and has no cycle, so it is a tree on t vertices. Therefore $\varepsilon(G')=|G'|-1=|G|-2$. It follows that $\varepsilon(G)=|G|-1$, so that the result is true when $|G|=t+1$. By induction, it holds for all values of $|G|$.

We saw in this proof that a tree G with $\varepsilon>0$ must have a vertex of degree one. Consider a longest path P in G. The two endpoints of P can only be joined to vertices of P. Since G does not contain any cycles, we can conclude that the endpoints of a longest path have degree one. Therefore a tree has *at least two* vertices of degree one.

In a connected graph, any two vertices are connected by some path. A fundamental property of trees is that any two vertices are connected by a *unique* path. For if there were two uu-paths P and Q, where $P\neq Q$, then traveling from u to u on P we could find the first point of P which is not on Q. Continuing on P until we come to the first point which is again on both P and Q, we could now follow Q back toward u and so find a cycle, which, however, is not possible in a tree.

Every graph G has subgraphs that are trees. The most important of these are the *spanning trees,* that is, trees which span all the vertices of G.

LEMMA 4.2 Every connected graph has a spanning tree.
PROOF Let *G* be a connected graph. If *G* has no cycles, then *G* is a spanning tree. Otherwise choose a cycle *C,* and remove any edge $xy \in C$ from *G*. *G* is still connected, since any *uu*-path which uses *xy* can now be replaced by a path using *C*−*xy,* so that every *u* and *u* are still connected by some path after *xy* has been removed. We repeat this as many times as necessary until the resulting graph has no cycles. It is a spanning tree of the original *G*.
Exercises
4.1.1 Describe in pseudo-code an algorithm to find an edge on a cycle, if one exists.
4.1.2 Make a list of all isomorphism types of trees on 1, 2, 3, 4, 5, and 6 vertices.
4.1.3 Show that there is a tree on *n* vertices with degree sequence (*d*1,*d*2,..., *dn*) if and only if

$$\sum_{i=1}^{n} d_i = 2(n-1).$$

Page 65
4.2 Fundamental cycles
Figure 4.2 shows a spanning tree of the Petersen graph. If *T* is a spanning tree of *G,* let *G*−*T* stand for the graph whose edges are *E(G)*−*E(T)*. Notice that if any edge $xy \in E(G-T)$ is added to *T,* then *T*+*xy* contains a unique cycle *Cxy*. This is because *x* and *y* are connected by a unique path *Pxy* in *T*. *P*xy+*xy* creates a cycle, *C*xy, called the *fundamental cycle* of *xy* with respect to *T*.

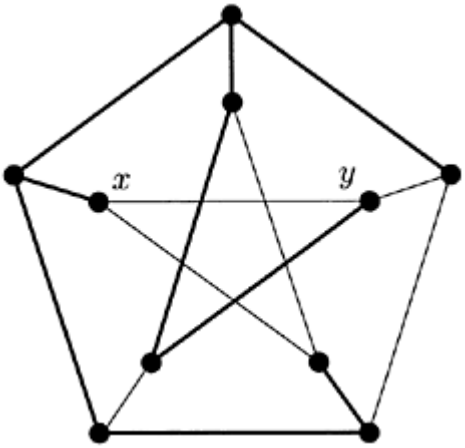


FIGURE 4.2
A spanning tree
Exercises
4.2.1 Show that *G* has *ε*−|*G*|+1 fundamental cycles with respect to any spanning tree *T*.
4.2.2 Let *T* be a spanning tree of *G,* and let *C* be a cycle of *G* containing exactly two edges *xy* and *uu* of *G*−*T*. Prove that $C = C_{xy} \oplus C_{uv}$, where $\oplus$ denotes the operation of *exclusive OR*.
Every cycle of *G* can be formed from the fundamental cycles of *G* with respect to any spanning tree *T*.
*THEOREM 4.3 Let T be a spanning tree of G. Let C be any cycle containing k edges u*1*u*1, *u*2*u*2,..., *ukuk of G*−*T, where k*≥1. *Then* $C = C_{u_1v_1} \oplus C_{u_2v_2} \oplus \cdots \oplus C_{u_kv_k}$.

Page 66
PROOF The proof is by induction on *k*. The result is certainly true when *k*=1. Suppose that the edges *uiui* occur on *C* in the order *i*=1,2,..., *k*. These edges divide the remaining edges of *C* into a number of paths *P*1, *P*2,..., *Pk,* where *Pi* connects *ui* to *ui*+1. This is shown in Figure 4.3.

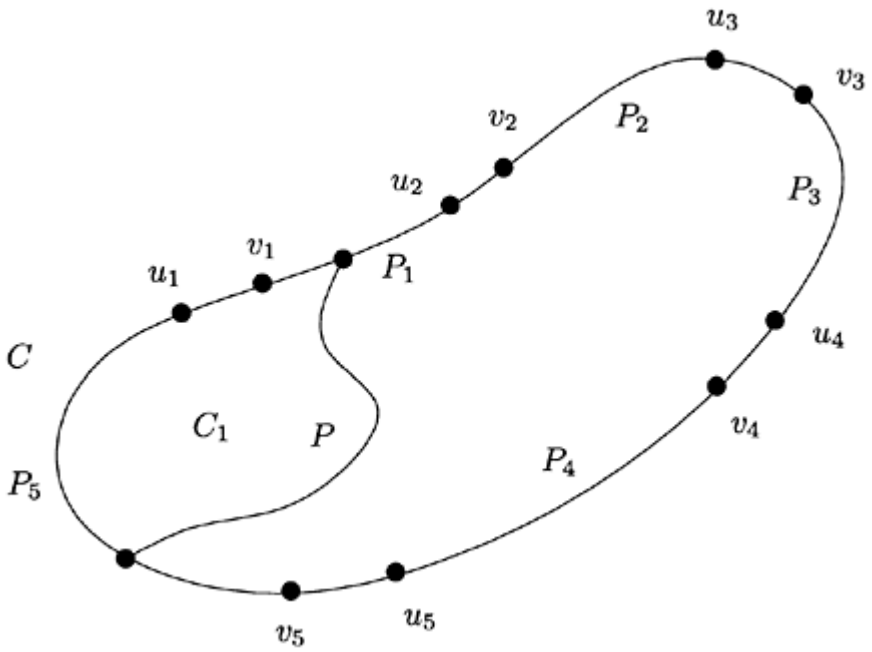


FIGURE 4.3
Decomposition into fundamental cycles

Let $C_i = C_{u_i v_i}$ denote the *ith* fundamental cycle. Consider *C*1. It consists of the edge *u*1*u*1 and the unique path *P* of *T* connecting *u*1 to *u*1. *P* and *P*1 both begin at vertex *u*1. As we travel on *P* from *u*1 toward *u*1, we eventually come to the first vertex of *P* which is not on *P*1. This is also the last vertex in common with *P*1, because *P* and *P*1 are both contained in *T,* which has no cycles. *P* may intersect several of the paths *Pi*. In each case the intersection must consist of a single segment, that is, a consecutive sequence of vertices of *Pi,* since *T* contains no cycles. The last path which *P* intersects is *Pk,* since both *P* and *Pk* end with *u*1. This is illustrated in Figures 4.3 and 4.4.

Consider now $H = C_1 \oplus C$. It is a subgraph of *G*. It consists of that part of *C* which is not contained in *C*1, plus that part of *P* which is not contained in *C*. Thus, the portions common to *P* and each *Pi* are discarded, but the new segments of *P* which are now added create one or more new cycles. Thus, *H* consists of one or more edge-disjoint cycles constructed from edges of *T,* plus the edges *u*2*u*2, *u*3*u*3,..., *ukuk*. Since each of these cycles contains fewer than *k*

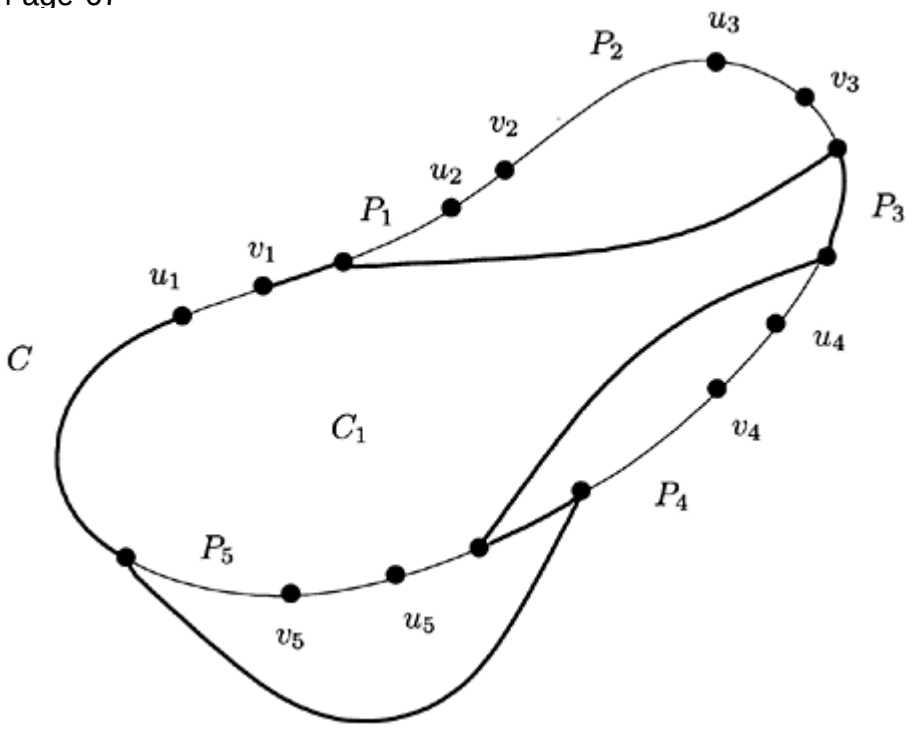


FIGURE 4.4
Decomposition into fundamental cycles

edges of *G−T,* we can say that $H = C_{u_2v_2} \oplus C_{u_3v_3} \oplus \cdots \oplus C_{u_kv_k}$. We then have $C_1 \oplus H = (C_1 \oplus C_1) \oplus C = C = C_{u_1v_1} \oplus C_{u_2v_2} \oplus \cdots \oplus C_{u_kv_k}$. Therefore the result is true when *C* contains *k* edges of *G−T*. By induction the result is true for all values of *k*.

Thus the fundamental cycles of *G* with respect to any spanning tree *T* generate all the cycles of *G*.

4.3 Co-trees and bonds

Let *G* be a graph. If $S \subset V(G)$, then $\overline{S}$ denotes *V(G)−S*. The edge-cut $[S, \overline{S}]$ consists of all edges of *G* with one endpoint in *S* and one endpoint in $\overline{S}$. Notice that $G - [S, \overline{S}]$ is a disconnected graph. See Figure 4.5.

If *T* is a spanning tree of *G,* then the complementary graph $\widehat{T} = G - T$ is called the *co-tree* corresponding to *T*. Now a co-tree $\widehat{T}$ cannot contain any edge-cut of

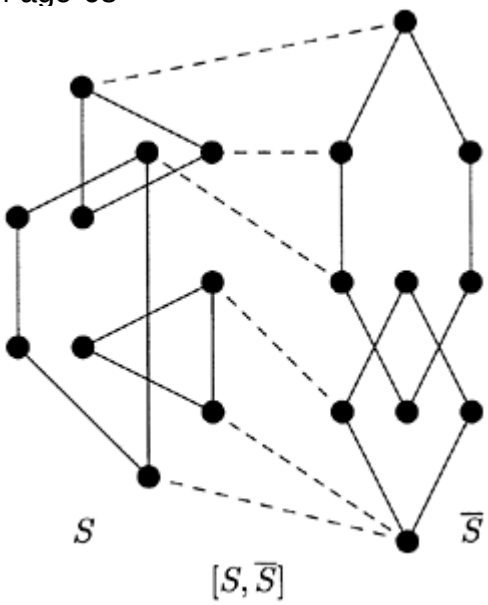


FIGURE 4.5
An edge-cut

G. This is because $G - \widehat{T} = T$ which is connected. If *uu* is any edge of *T,* then *T−uu* consists of two components, *Su,* those vertices connected to *u,* and *Su*, those vertices connected to *u*. *[Su, Su]* is an edge-cut of *G*. It is contained in $\widehat{T} + uv$. This is illustrated in Figure 4.6.

[Su, Su] is a *minimal edge-cut* of *G,* that is, it does not contain any smaller edge-cuts. For if *xy* is any edge, where $x \in S_u$ and $y \in S_v$, then *G−[Su,Su]+xy* is connected. Therefore:

1. A co-tree $\widehat{T}$ contains no edge-cut.

2. If *uu* is any edge of $G - \widehat{T}$, then $\widehat{T} + uv$ contains a unique minimal edge-cut *Buu=[Su, Su]*.

Compare this with trees:

1. A tree *T* contains no cycle.
2. If *uu* is any edge of *G−T,* then *T+uu* contains a unique fundamental cycle *Cuu*.

The unique edge-cut *Buu* contained in $\widehat{T} + uv$ is called the fundamental *edge-cut* of *uu* with respect to $\widehat{T}$. Any minimal edge-cut of *G* is called a *bond*. There is a duality between trees and co-trees, and cycles and bonds (bonds are sometimes called *co-cycles*). There is a linear algebra associated with every graph, in which cycles and bonds generate orthogonal vector spaces, called the *cycle space* and *bond space* of *G*. Theorem 4.3 above shows that the fundamental cycles with

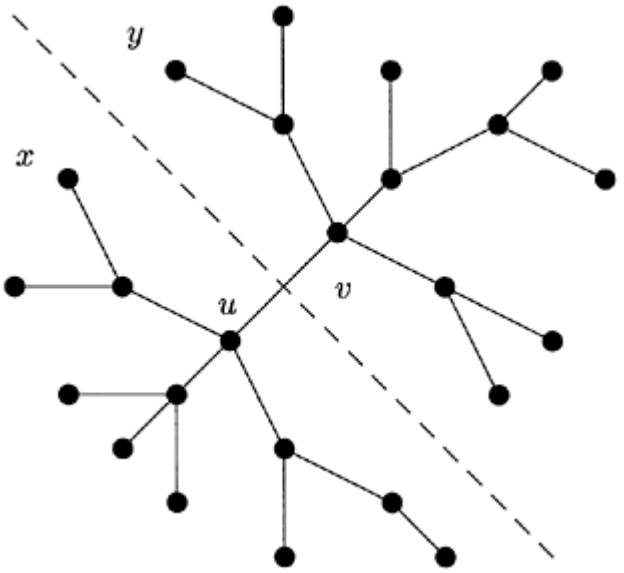


FIGURE 4.6
Co-trees and edge-cuts

respect to any spanning tree form a basis for the cycle space. Similarly, the fundamental edge-cuts form a basis for the bond space. See BONDY and MURTY [19] for more information.

Exercises

4.3.1 How may fundamental edge-cuts does *G* have with respect to a co-tree $\widehat{T}$? What are the dimensions of the cycle space and bond space of *G*?

4.3.2 Let *T* be a spanning tree of *G,* and let edge $uv \notin T$. Let *xy* be any edge of the fundamental cycle *Cuu,* such that *xy≠uu*. Then *T+uu−xy* is also a spanning tree of *G*. Thus, spanning trees of *G* are adjacent via fundamental cycles. The *tree graph* of *G* is *Tree(G)*. Its vertices are the spanning trees of *G,* and they are adjacent via fundamental cycles. Show that *Tree(G)* is a connected graph.

4.3.3 Show that trees (co-trees) are also adjacent via fundamental edge-cuts.

4.3.4 Let *T* be a spanning tree of *G,* and let *W* be a closed walk in *G* such that *W* uses edges of *T* and edges *u*1*u*1, *u*2*u*2,..., *ukuk* of *G−T*. Describe the subgraph $H = C_{u_1v_1} \oplus C_{u_2v_2} \oplus \cdots \oplus C_{u_kv_k}$. What is its relation to *W*?

4.3.5 Let $[S, \overline{S}]$ be an edge-cut of *G*. Prove that $[S, \overline{S}]$ is a bond if and only if *G[S]* and $G[\overline{S}]$ are connected graphs.

4.3.6 Let $B_1 = [S_1, \overline{S}_1]$ be an edge-cut of *G,* and let $B_2 = [S_2, \overline{S}_2]$ be a bond contained in $[S_1, \overline{S}_1]$; that is, $B_1 \subset B_2$. (Note: *S*2 will generally not be a subset of *S*1.) Prove

that $[S_1, \overline{S}_1] - [S_2, \overline{S}_2]$ is also an edge-cut.

4.3.7 Use the previous question to prove that every edge-cut can be decomposed into a disjoint union of bonds.

4.3.8 Find a decomposition of the edge-cut $[S, \overline{S}]$ in the graph shown in Figure 4.7 into bonds. The set *S* is marked by the shading. Is the decomposition unique? (*Hint:* Redraw the graph so that edges don't cross each other.)

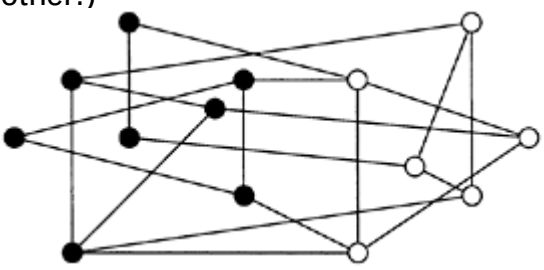


FIGURE 4.7
Find a decomposition into bonds

4.3.9 Let *T* be a spanning tree of *G*. Let *uu* and *xy* be edges of *T,* with corresponding bonds *Buu=[Su,Su]* and

Bxy=[Sx,Sy], where *Buu∩Bxy≠Ø*. Prove that $B_{uv} \oplus B_{xy}$ is a bond.
4.3.10 Prove that any cycle and any bond must intersect in an even number of edges.

4.4 Spanning tree algorithms

One of the easiest ways to construct a spanning tree of a graph *G* is to use a breadth-first search. The following code is adapted from Algorithm 2.4.1. The statements marked with $(\star)$ have been added.

Page 71

Algorithm 4.4.1: BFSEARCH*(G, u)*
comment: build a breadth-first spanning tree of *G*
for *u*←1 to |*G*| **do** *OnScanQ[u]*←**false**
ScanQ[1]←*u*
OnScanQ[u]←**true**
QSize←1
k←1
Parent[u]←0⋆
BFNum[u]←1⋆
Count←1⋆
Tree←empty list⋆
repeat
u←*ScanQ[k]*
for each $w \longrightarrow v$
do if not *OnScanQ[w]*

then
$QSize \leftarrow QSize + 1$
$ScanQ[QSize] \leftarrow w$
$OnScanQ[w] \leftarrow$ **true**
$Parent[w] \leftarrow v$ $(\star)$
$Count \leftarrow Count + 1$ $(\star)$
$BFNum[w] \leftarrow Count$ $(\star)$
add edge vw to *Tree* $(\star)$

k←*k*+1
until *k*>*QSize*

A number of arrays *ScanQ, OnScanQ, Parent,* and *BFNum* are used in this algorithm, as well as the counters *QSize* and *Count,* and the list of edges *Tree* of the spanning tree constructed.
BFSEARCH*(G,u)* visits each vertex of the connected graph *G,* beginning with *u*. The order in which the vertices are visited defines a numbering of the vertices, called the *breadth-first numbering*. It is saved in *BFNum*[·]. This is illustrated in Figure 4.8.
The search begins at node *u,* called the *root* of the spanning tree. In the example above, *u*=1. As each node *w* is placed on the *ScanQ,* its *parent* in the search tree is saved in *Parent[w]*. This is represented by the arrows on the tree in the diagram. Thus, beginning at any node in the graph, we can follow the *Parent*[·] values up to the root of the tree. The breadth-first numbering defines a traversal of the tree, which goes level by level, and from left to right in the drawing. A great many graph algorithms are built around the breadth-first search. The important property of breadth-first spanning trees is that the paths it constructs connecting

Page 72

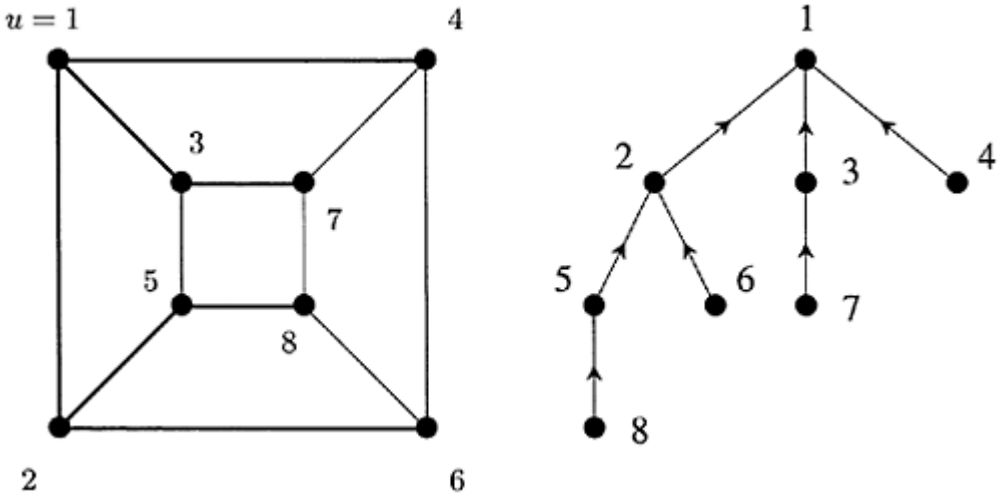


FIGURE 4.8
A breadth-first tree

any vertex *w* to the root of the tree are *shortest* paths.

In a weighted graph, different spanning trees will have different weight, where

$$\text{WT}(T) = \sum_{uv \in T} \text{WT}(uv).$$

We now want to find a spanning tree *T* of *minimum* weight. This is called the *minimum spanning tree problem*. There are many algorithms which solve it. We present some of them here.

4.4.1 Prim's algorithm

The idea here is to pick any $u \in V(G)$ and "grow" a tree on it; that is, at each iteration, we add one more edge to the current tree, until it spans all of *V(G)*. We must do this in such a way that the resulting tree is minimum.

Page 73

Algorithm 4.4.2: PRIM*(G)*

comment: *Tree* is a list of edges in a minimum spanning tree.
comment: VT are the vertices in the current tree being grown.

initialize *Tree* to contain no edges
$t \leftarrow 0$ "the number of edges in *Tree*"
choose any $u \in V(G)$
initialize VT to contain u
comment: the *Tree* now has 1 node and 0 edges
while $t < |G| - 1$
do
- choose an edge xy of *minimum* weight, with $x \in VT$ and $y \notin VT$
- add xy to *Tree*
- add y to VT
- $t \leftarrow t + 1$

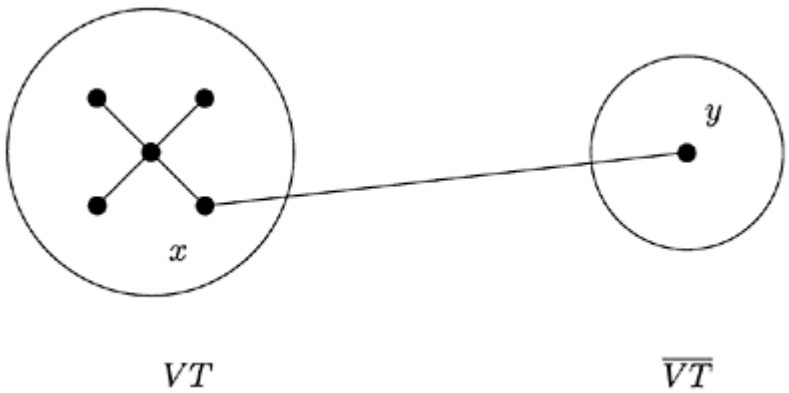


FIGURE 4.9

Growing a tree with Prim's algorithm

We first prove that Prim's algorithm does in fact produce a minimum spanning tree. Initially, *VT* contains one vertex, and Tree contains no edges. On each iteration an edge *xy* with $x \in VT$ and $y \notin VT$ is added to *Tree,* and *y* is added to *VT*. Therefore, the edges of *Tree* always form a tree which spans *VT*. After *n*−1 iterations, it is a spanning tree of *G*. Call the tree produced by Prim's algorithm *T,* and suppose that it consists of edges *e*1, *e*2,..., *e*n−1, chosen in that order. If it is not a minimum spanning tree, then choose a minimum tree *T** which agrees with *T* on the first *k* iterations, but not on iteration *k*+1, where *k* is as large as possible. Then *e*1, *e*2,..., $e_k \in T^*$, but $e_{k+1} \notin T^*$. Consider iteration

k+1, and let *ek*+1=*xy,* where $x \in VT$ and $y \in \overline{VT}$. Then *T**+*xy* contains a fundamental cycle *Cxy*.

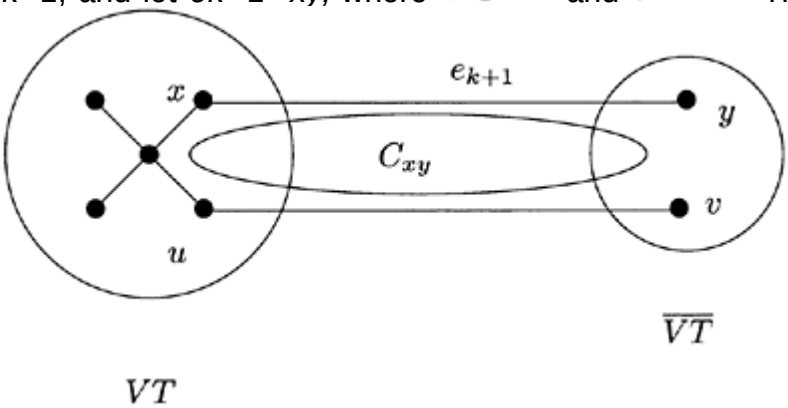


FIGURE 4.10
A fundamental cycle in Prim's algorithm

Cxy must contain another edge *uv* with $u \in VT$ and $v \in \overline{VT}$. Since Prim's algorithm chooses edges by weight, we know that WT*(xy)*≤WT*(uv)*. Now *T'*=*T**+*xy*-*uv* is also a spanning tree, and WT*(T')*≥WT*(T*)*, since *T** is a minimum tree. But WT*(T')*=WT*(T*)*+WT*(xy)*-WT*(uv)*=WT*(T*)*. Therefore, WT*(T')*=WT*(T*)* and WT*(xy)*=WT*(uv)*. It follows that *T'* is also a minimum tree, and that *T'* contains *e*1, *e*2,..., *ek*+1; that is, it agrees with *T* on *k*+1 iterations, a contradiction. Consequently, Prim's tree *T* is also a minimum spanning tree.

Data structures

The main operation performed in Prim's algorithm is to select the edge *xy,* of minimum weight, with $x \in VT$ and $y \in \overline{VT}$. One way to do this is to store two values for each vertex $y \in \overline{VT}$:
MinWt[y]: the minimum weight WT*(xy),* over all $x \longrightarrow y$, where $x \in VT$
MinPt[y]: that vertex $x \in VT$ with WT*(xy)*=*MinWt[y]*.

Then to select the minimum edge *xy,* we need only perform the following steps: select $y \in \overline{VT}$ with smallest *MinWt[y]* value *x*←*MinPt[y]*

$$\textbf{for each } w \longrightarrow y \textbf{ do } \begin{cases} \textbf{if } w \in \overline{VT} \\ \quad \textbf{then update } MinWt[w] \end{cases}$$

Let *n*=|*G*|. If we scan the set $\overline{VT}$ in order to select the minimum vertex *y* on each iteration, then the first iteration requires scanning *n*−1 vertices, the second iteration requires *n*−2 steps, etc., requiring $1 + 2 + \cdots + (n-1) = \binom{n}{2}$ in total. The total number of steps needed to update the *MinWt*[·] values is at most *Σy* DEG*(y)*=2*ε* steps, over all iterations. Thus, the complexity when Prim's algorithm is programmed like this is

$$O(2\varepsilon + \frac{n^2}{2}) = O(\varepsilon + n^2).$$

In order to remove the *O*(*n*2) term, we could store the vertices $\overline{VT}$ in a heap *H*. Selecting the minimum now requires approximately log *n* steps. For each $w \longrightarrow y$ we may also have to update *H,* requiring at most an additional DEG*(y)* log *n* steps per iteration. The total number of steps performed over all iterations is now at most

$$\sum_{k=1}^{n-1} \log n + \sum_{y} \text{DEG}(y) \log n \leq n \log n + 2\varepsilon \log n.$$

The complexity of Prim's algorithm using a heap is therefore $O(n \log n + \varepsilon \log n)$. If ε is small, this will be better than the previous method. But if ε is large, this can be worse, depending on how much time is actually spent updating the heap. Thus we can say that Prim's algorithm has complexity:
$O(\varepsilon + n2)$, if the minimum is found by scanning.
$O(n \log n + \varepsilon \log n)$, if a heap is used.

Exercises

4.4.1 Work Prim's algorithm by hand on the graph in Figure 4.11, starting at the shaded vertex.

4.4.2 Consider Dijkstra's shortest-path algorithm, which finds a shortest *uv*-path for all $v \in V(G)$. For each *u,* let *Pu* be the shortest path found. Show that the collection of paths, $\bigcup_v P_v$, defines a spanning tree of *G*. Is it a minimum spanning tree? (*Hint:* Use induction.)

4.4.3 Program Prim's algorithm, storing the vertices $\overline{VT}$ in a heap.

4.4.4 Modify the breadth-first search algorithm to find the fundamental cycles of *G* with respect to a BF-tree. Print out the edges on each fundamental cycle. What is the complexity of the algorithm?

4.4.5 Let *G* be a weighted graph in which all edge-weights are distinct. Prove that *G* has a unique minimum spanning tree.

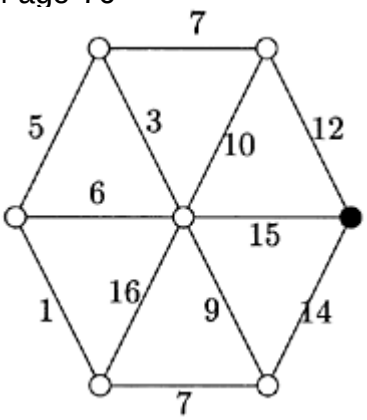


FIGURE 4.11
Find a spanning tree

4.4.2 Kruskal's algorithm

A *forest* is a graph which need not be connected, but whose every component is a tree. Prim's algorithm constructs a spanning tree by growing a tree from some initial vertex. Kruskal's algorithm is quite similar, but it begins with a spanning forest and adds edges until it becomes connected. Initially the forest has $n=|G|$ components and no edges. Each component is a single vertex. On each iteration, an edge which connects two distinct components is added, and the two components are merged. When the algorithm terminates the forest has become a tree.

Algorithm 4.4.3: KRUSKAL*(G)*

comment: $\begin{cases} \textit{Tree is a list of edges in a minimum spanning tree.} \\ T_u \text{ is the component of the forest which contains } u. \end{cases}$

initialize *Tree* to contain no edges
for each $u \in V(G)$ do initialize *Tu* to contain only *u*
t←0 "the number of edges in Tree"
comment: the forest currently has |*G*| nodes and 0 edges
while *t*<|*G*|−1

$$\textbf{do}\begin{cases}\text{Select the next edge } xy \text{ of } \textit{minimum} \text{ weight, and determine}\\ \text{which components } x \text{ and } y \text{ are in, say } x \in T_u \text{ and } y \in T_v\\ \textbf{if } T_u \neq T_v\\ \quad \textbf{then}\begin{cases}\text{merge } T_u \text{ and } T_v\\ \text{add } xy \text{ to } \textit{Tree}\\ t \leftarrow t+1\end{cases}\end{cases}$$

Page 77

Initially the forest has *n* components, and each one is a tree with no edges. Each edge that is added connects two distinct components, so that a cycle is never created. Whenever an edge is added the two components are merged, so that the number of components decreases by one. After *n*−1 iterations, there is only one component left, which must be a tree *T*. If *T* is not a minimum tree, then we can proceed as we did in Prim's algorithm. Let *T* consist of edges *e*1, *e*2,..., *e*n−1, chosen in that order. Select a minimum tree *T** which contains *e*1, *e*2,..., *ek*, but not *ek*+1, where *k* is as large as possible. Consider the iteration in which *ek*+1= *xy* was selected. *T**+*xy* contains a fundamental cycle *Cxy,* which must contain another edge *ab* incident on *Tx*. Since Kruskal's algorithm chooses edges in order of their weight, WT*(xy)*≤WT*(ab)*. Then *T′*=*T**+*xy*−*ab* is a spanning tree for which WT*(T′)*≤WT*(T*)*. But *T** is a minimum tree, so that WT*(T′)*= WT*(T*)*, and *T′* is also a minimum tree. *T′* contains edges *e*1, *e*2,..., *ek*+1, a contradiction. Therefore, Kruskal's tree *T* is a minimum spanning tree.

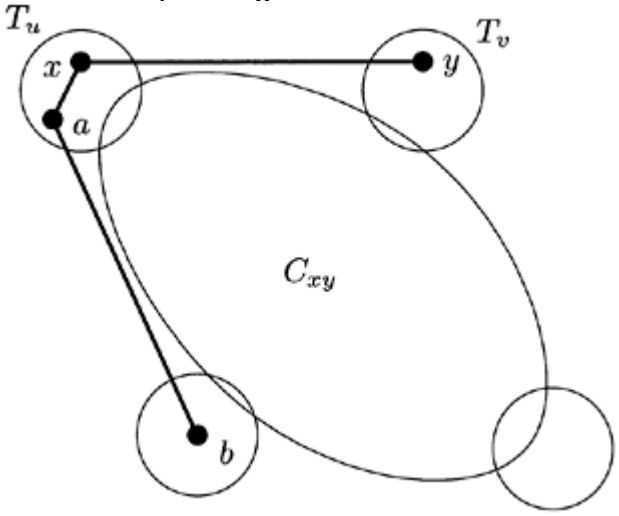


FIGURE 4.12
Growing a forest with Kruskal's algorithm

Data structures and complexity

The main operations in Kruskal's algorithm are:

1. Choose the next edge *xy* of minimum weight.

2. Determine that $x \in T_u$ and $y \in T_v$.

Page 78

3. Merge *Tu* and *Tu*.

The edges could either be completely sorted by weight, which can be done in *O*(*ε* log *ε*) steps, or they could be kept in a heap, which makes it easy to find the minimum edge. Since we may have a spanning tree *T* before all the edges have been considered, it is usually better to use a heap. The components *Tu* can easily be stored using the merge-find data structure described in Chapter 2.

Each time an edge *xy* is selected from the heap, it requires approximately log *ε* steps to update the heap. In the worse case we may need to consider every edge of *G,* giving a bound of *ε* log *ε* steps. Similarly, *O(εa(n))* steps are needed to build the components, where *n*=|*G*|. Thus, Kruskal's algorithm can be programmed with a complexity of *O*(*ε* log *n*+*εa(n)*), where we have used log *ε*<2 1og *n*. Notice that this can be slightly better than Prim's algorithm. This is because the term *a(n)* is essentially a constant, and because the heap does not need to be constantly updated as the *MinWt*[·] value changes.

4.4.3 The Cheriton-Tarjan algorithm

The Cheriton-Tarjan algorithm is a modification of Kruskal's algorithm designed to reduce the *O(ε* log *ε)* term.

It also grows a spanning forest, beginning with a forest of $n=|G|$ components each consisting of a single node. Now the term $O(\varepsilon \log \varepsilon)$ comes from selecting the minimum edge from a heap of ε edges. Since every component *Tu* must eventually be connected to another component, this algorithm keeps a separate heap *PQu* for each component *Tu,* so that initially *n* smaller heaps are used. Initially, *PQu* will contain only DEG*(u)* edges, since *Tu* consists only of vertex *u*. When *Tu* and *Tu* are merged, *PQu* and *PQu* must also be merged. This requires a modification of the data structures, since *heaps cannot be merged efficiently*. This is essentially because merging heaps reduces to building a new heap. Any data structure in which a minimum element can be found efficiently is called a *priority queue*. A heap is one form of priority queue, in which elements are stored as an array, but viewed as a binary tree. There are many other forms of priority queue. In this algorithm, *PQu* will stand for a priority queue which can be merged. The Cheriton-Tarjan algorithm can be described as follows.
It stores a list Tree of the edges of a minimum spanning tree. The components of the spanning forest are represented as *Tu* and the priority queue of edges incident on vertices of *Tu* is stored as *PQu*.

Page 79

Algorithm 4.4.4: CHERITONTARJAN*(G)*

initialize *Tree* to contain no edges
for each $u \in V(G)$
do { initialize T_u to contain only u
create PQ_u
for each $v \longrightarrow u$
  do add uv to PQ_u
comment: each edge will appear in two priority queue

comment: the forest currently has $|G|$ nodes and 0 edges
$t \leftarrow 0$
while $t<|G|-1$
do { select a component T_u
repeat
  select the minimum edge $xy \in PQ_u$ and determine
  which components x and y are in, say $x \in T_u$ and $y \in T_v$
until $T_u \neq T_v$
comment: xy connects two different components
merge T_u and T_v
merge PQ_u and PQ_v
add xy to *Tree*
$t \leftarrow t+1$

Exercises

4.4.1 Prove that the Cheriton-Tarjan algorithm constructs a minimum spanning tree.
4.4.2 Show that a heap is best stored as an array. What goes wrong when the attempt is made to store a heap with pointers?
4.4.3 Show that heaps cannot be merged efficiently. Describe an algorithm to merge two heaps, both with *n* nodes, and work out its complexity.
4.4.4 Program Kruskal's algorithm, using a heap to store the edges, and the merge-find data structure to store the components.

4.4.4 Leftist binary trees

A *leftist binary tree* (LB-tree) is a modification of a heap which allows efficient merging. A node *x* in an LB-tree has the following four fields:

1. $Value\langle x\rangle$: the value stored.

Page 80

2. $Left\langle x\rangle$: a pointer to the left subtree.
3. $Right\langle x\rangle$: a pointer to the right subtree.

4. $rPath\langle x\rangle$: the right-path distance.
An LB-tree satisfies the heap property; namely the entry stored in any node has value less than or equal to that of its two children:

$$Value\langle x\rangle \leq Value\langle Left\langle x\rangle\rangle$$

and

$$Value\langle x\rangle \leq Value\langle Right\langle x\rangle\rangle$$

Therefore the smallest entry in the tree occurs in the top node. Thus, a heap is a special case of an LB-tree. The distinctive feature of LB-trees is contained in field *rPath [x]*. If we begin at any node in an LB-tree and follow *Left* and *Right* pointers in any sequence, we eventually reach a nil pointer. In an LB-tree, the shortest such path is *always the rightmost path*. This is true for every node in the tree. The length of the path for a node *x* is the $rPath\langle x\rangle$ value. In the tree shown in Figure 4.13, the *rPath* values are shown beside each node.

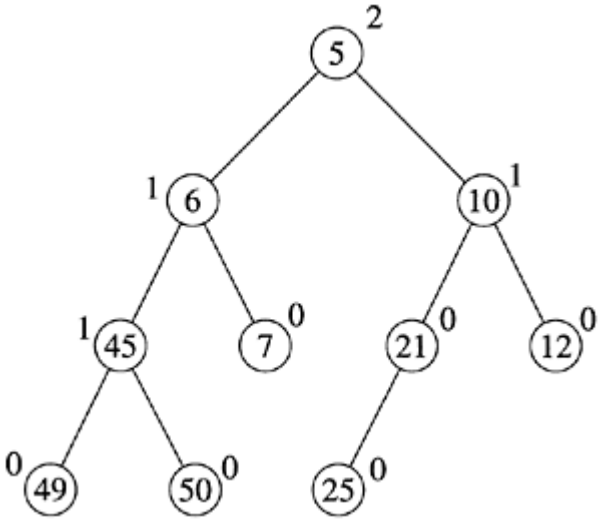


FIGURE 4.13
A leftist binary tree
In summary, an LB-tree is a binary tree which satisfies the heap property, and whose shortest path to a nil pointer from any node is always the rightmost path.

This means that LB-trees will tend to have more nodes on the left than the right; hence the name leftist binary trees.
The rightmost path property makes it possible to merge LB-trees efficiently. Consider the two trees *A* and *B* in Figure 4.14 which are to be merged into a tree *T*.

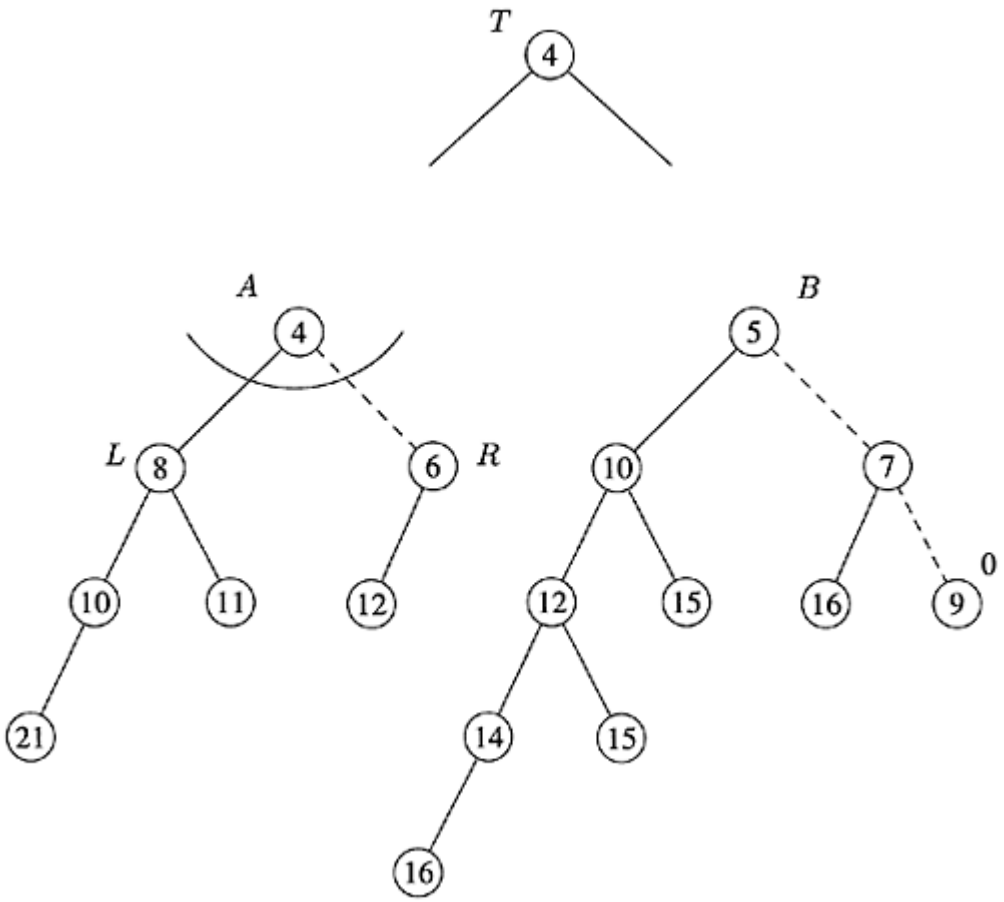


FIGURE 4.14
Merging two leftist binary trees
The top node of *T* is evidently taken from *A,* since it has the smaller minimum. This breaks *A* into two subtrees, *L* and *R*. The three trees *B, L*, and *R* are now to be made into two subtrees of *T*. The easiest way to do this is first to merge *R* and *B* into a single tree *P,* and then take *P* and *L* as the new right and left subtrees of *T,* placing the one with the smaller *rPath* value on the right. The recursive merge procedure is described in Algorithm 4.4.5, and the result of merging *A* and *B* of Figure 4.14 is shown in Figure 4.15.

Page 82

Algorithm 4.4.5: LBMERGE*(A, B)*
comment: Merge non-null LB-trees *A* and *B*
if $Value\langle A\rangle > Value\langle B\rangle$ **then** swap *A* and *B*
if $Right\langle A\rangle = \textbf{null}$ **then** *P*←*B*
else *P*←LBMERGE($Right\langle A\rangle$ B)
comment: choose the tree with the smaller *rPath* as right subtree

$$\textbf{if } Left\langle A\rangle = \textbf{null}$$
$$\textbf{then} \begin{cases} Right\langle A\rangle \leftarrow \textbf{null} \\ Left\langle A\rangle \leftarrow P \\ rPath\langle A\rangle \leftarrow 0 \end{cases}$$
$$\textbf{else} \begin{cases} \textbf{if } rPath\langle P\rangle \le rPath\langle Left\langle A\rangle\rangle \\ \quad \textbf{then } Right\langle A\rangle \leftarrow P \\ \quad \textbf{else} \begin{cases} Right\langle A\rangle \leftarrow Left\langle A\rangle \\ Left\langle A\rangle \leftarrow P \end{cases} \\ rPath\langle A\rangle \leftarrow rPath\langle Right\langle A\rangle\rangle + 1 \end{cases}$$

return *(A)*

Notice that when the top node of *A* is removed, thereby splitting *A* into two subtrees *L* and *R,* the left subtree *L* subsequently becomes one of the subtrees of *T*. That is, *L* is not decomposed in any way, it is simply

transferred to *T*. Furthermore, *L* is usually the larger of the two subtrees of *A*. Let us estimate the number of steps necessary to merge *A* and *B,* with *rPath* values *r*1 and *r*2, respectively. One step is needed to choose the smaller node *A,* say, as the new top node. The right subtree *R* will have rightmost path length of *r*1−1. When *R* and *B* are merged, one of them will be similarly decomposed into a left and right subtree. The left subtree is never broken down. At each step in the recursion, the smaller value is chosen as the new top node, and its *Right* becomes the next subtree to be considered; that is, LBMERGE() follows the rightmost paths of *A* and *B,* always choosing the smaller entry of the two paths. Thus, the rightmost paths of *A* and *B* are merged into a single path (see Figure 4.15). Therefore, the depth of the recursion is at most *r*1+*r*2. At the bottom of the recursion the *rPath* values may both equal zero. It then takes about five steps to merge the two trees. Returning up through the recursion, LBMERGE() compares the *rPath* values of *L* and *P,* and makes the smaller one into the new right subtree. Also, the new *rPath* value is assigned. All this takes about four steps per level of recursion, so that the total number of steps is at most 5(*r*1+*r*2+1).

What is the relation between the rightmost path length of an LB-tree and the number of nodes it contains? If the *rPath* value of an LB-tree *T* is *r,* then beginning at the top node, every path to a nil pointer has length at least *r*. Therefore, *T* contains at least a full binary tree of *r* levels; that is, *T* has at least 2(*r*+1)−1

Page 83

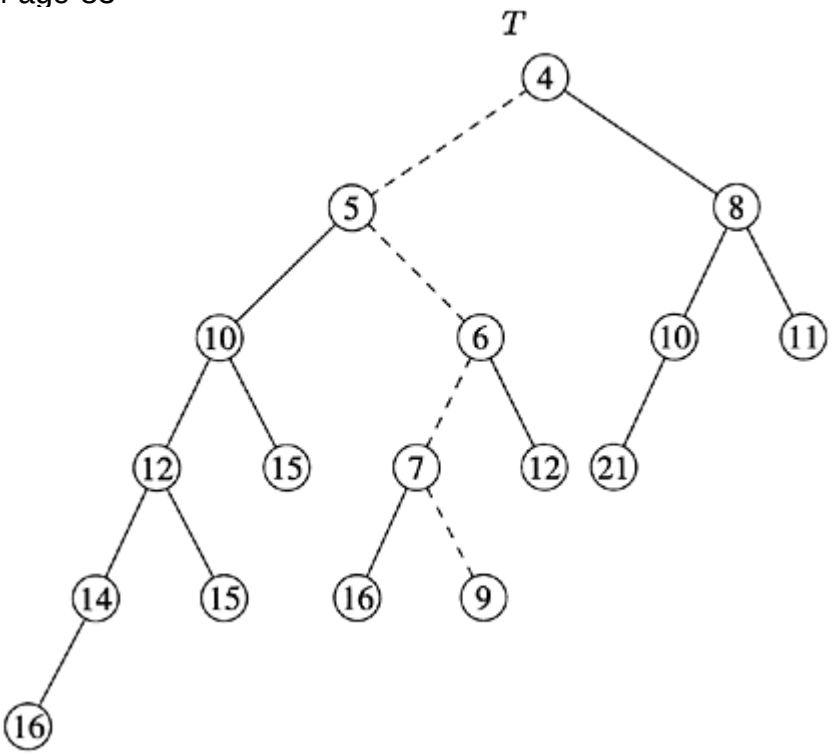


FIGURE 4.15
The merged LB-tree

nodes. The largest rightmost path length possible if *T* is to store *n* nodes is the smallest value of *r* such that 2(*r*+1)−1≥*n,* or $r \leq \lceil \log \frac{n+1}{2} \rceil$.

If *A* and *B* both contain at most *n* nodes, then $r_1, r_2 \leq \lceil \log \frac{n+1}{2} \rceil$, and LBMERGE(A, B) takes at most 5(*r*1+*r*2+1)≤10. $\lceil \log \frac{n+1}{2} \rceil + 5 = O(\log n)$ steps. Thus LB-trees can be merged quite efficiently.

We can use this same method to extract the minimum entry from an LB-tree *A,* using at most *O*(log *n*) steps: select minimum as $Value\langle A \rangle$

$A \leftarrow LBMERGE(Left\langle A \rangle, Right\langle A \rangle)$

Consider now how to construct an LB-tree. In Chapter 2 we found that there are two ways of building a heap, one much more efficient than the other. A similar situation holds for LB-trees. The most obvious way to build one is to merge successively each new node into an existing tree:

Page 84

initialize *A,* a new LB-tree with one node

repeat
get next value
create and initialize a new LB-tree, *B*
A←LBMERGE *(A,B)*
until all values have been inserted

However, this can easily create LB-trees that are really linear linked lists, as shown below. This algorithm then becomes an insertion sort, taking $O(n2)$ steps, where n is the number of nodes inserted.

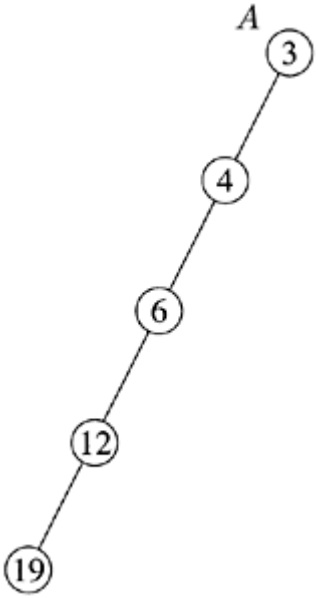


FIGURE 4.16
An LB-tree

A better method is to create n LB-trees, each containing only one node, and then merge them two at a time, until only one tree remains. The trees are kept on a queue, called the *MergeQ*.

Algorithm 4.4.6: BUILDLBTREE*(MergeQ)*
repeat
select *A* and *B,* the first two trees of *MergeQ*
A←LBMERGE*(A,B)*
put *A* at end of *MergeQ*
until *MergeQ* contains only one tree
return *(A)*

Page 85

How many steps are needed to build an LB-tree in this way, if we begin with n trees of one node each? There will be $\lfloor n/2 \rfloor$ merges of pairs 1-node trees, each of which takes at most 5 steps. This will leave $\lceil n/2 \rceil$ trees on the Merge*Q,* each with at most two nodes. These will be taken two at a time, giving $\lfloor n/4 \rfloor$ merges of up to 2-node trees. Similarly there will be $\lfloor n/8 \rfloor$ merges of up to 4-node trees, etc. This is summarized in the following table:

tree size	# pairs	max *rPath*	max *r*1+*r*2+1
1	$\lfloor n/2 \rfloor$	0	1
2	$\lfloor n/4 \rfloor$	0	1
4	$\lfloor n/8 \rfloor$	1	3
8	$\lfloor n/16 \rfloor$	2	4
⋮	⋮	⋮	⋮
2*k*	$\lfloor n/2^{k+1} \rfloor$	*k*−1	2*k*−1

The last step will merge two trees with roughly $n/2$ nodes each. The maximum *rPath* value for these trees will be $\leq \lceil \log \frac{n/2+1}{2} \rceil$, or approximately $\lceil \log n \rceil - 1$. The total number of steps taken to build the LB-tree is then at most

$$5\lfloor\frac{n}{2}\rfloor + \sum_{k=1}^{\lfloor\log n\rfloor-1} 5(2k-1)\lfloor\frac{n}{2^{k+1}}\rfloor \leq \frac{5n}{2} + 5n \sum_{k=1}^{\lfloor\log n\rfloor-1} \frac{2k-1}{2^{k+1}}.$$

We can sum this using the same technique as in the heap analysis of Chapter 2, giving a sum of $10n - \frac{5nr}{2^r}$, where $r = \lfloor\log n\rfloor - 1$. Thus, an LB-tree can be built in *O(n)* steps.

We can now fill in the details of the Cheriton-Tarjan spanning tree algorithm. There are three different kinds of trees involved in the algorithm:

1. A minimum spanning tree is being constructed.
2. The components *Tu* are merge-find trees.
3. The priority queues *PQu* are LB-trees.

At the beginning of each iteration, a component *Tu* is selected, and the minimum edge $xy \in PQ_u$ is chosen. How is *Tu* selected? There are several possible strategies. If we choose the same *Tu* on each iteration, then the algorithm grows a tree from *u;* that is, it reduces to Prim's algorithm. If we choose the component *Tu* incident on the minimum remaining edge, then the algorithm reduces to Kruskal's algorithm. We could choose the smallest component *Tu,* but this would add an extra level of complication, since we would now have to keep a heap of components in order to find the smallest component quickly. The method which Cheriton and Tarjan recommend is *uniform selection;* that is, we keep a queue, *TreeQ,* of components. Each entry on the *TreeQ* contains *Tu* and *PQu*. On each

iteration, the component *Tu* at the head of the queue is selected and the minimum $xy \in PQ_u$ is chosen, say $x \in T_u$ and $y \in T_v$, where *Tu≠Tu*. Once *Tu* and *T*u, *PQu* and *PQu* have been merged, they are moved to the end of the *TreeQ*. Thus, the smaller components will tend to be at the head of the queue. So the algorithm uses two queues, the *MergeQ* for constructing LB-trees and merging them, and the *TreeQ* for selecting components.

The complexity analysis of the Cheriton-Tarjan algorithm is beyond the scope of this book. If analyzed very carefully, it can be shown to be *O*(*ε* log log *ε*), if programmed in a very special way.

Minimum spanning tree algorithms are a good illustration of the process of algorithm development. We begin with a simple algorithm, like growing a spanning tree from an initial vertex, and find a complexity of *O*(*n*2). We then look for a data structure or programming technique that will allow the *n*2 term to be reduced, and obtain a new algorithm, with complexity *O*(*ε* log *n*), say. We then ask how the *ε* or log *n* term can be reduced, and with much more effort and more sophisticated data structures, obtain something like $O(\sqrt{\varepsilon}\log n)$ or *O*(*ε* log log *n*). Invariably, the more sophisticated algorithms have a higher constant of proportionality, so that improvements in running time are only possible when *n* and *ε* become very large. However, the sophisticated algorithms also indicate that there are theoretical limits of efficiency for the problem at hand.

Exercises

4.4.1 Prove that the result of LBMERGE(A,B) is always an LB-tree, where *A* and *B* are non-nil LB-trees.

4.4.2 Let *A* be an LB-tree with *n* nodes and let *B* be an arbitrary node in the tree. Show how to update *A* if:

(a) $\mathbf{Value}\langle B\rangle$ is increased.

(b) $\mathbf{Value}\langle B\rangle$ is decreased.

(c) Node *B* is removed.

4.4.3 **Delayed Merge.** When *(Tu,PQu)* is selected from the *TreeQ,* and merged with *(Tu, PQu),* the result is moved to the end of the queue. It may never come to the head of the *TreeQ* again. In that case, it would not really be necessary to perform the LBMERGE*(PQu,PQu)*. Cheriton and Tarjan delay the merging of the two by creating a new dummy node *D* and making *PQu* and *PQu* into its right and left subtrees. *D* can be marked as a dummy by setting $\mathbf{rPath}\langle D\rangle$ to −1. Several dummy nodes may accumulate at the top of the trees *PQu*. Should a tree with a dummy node come to the head of the queue, its dummy nodes must be removed before the minimum edge $xy \in PQ_u$ can be selected. Write a recursive tree traversal which removes the dummy nodes from an LB-tree, and places its non-dummy subtrees on the *MergeQ*. We can then use BUILDLBTREE() to combine all the subtrees on the

MergeQ into one.

4.4.4 Program the Cheriton-Tarjan algorithm, using leftist binary trees with delayed merge, to store the priority queues.

4.5 Notes

An excellent description of the cycle space and bond space can be found in BONDY and MURTY [19]. Kruskal's and Prim's algorithms are standard algorithms for minimum spanning trees. They are described in most books on algorithms and data structures. The Cheriton-Tarjan algorithm is from CHERITON and TARJAN [22]. Leftist binary trees are from KNUTH [75], and are also described in WEISS [122].

Page 88

This page intentionally left blank.

Page 89

5

The Structure of Trees

5.1 Introduction

The structure of trees is naturally recursive. When trees are used as data structures, they are typically processed by recursive procedures. Similarly, exhaustive search programs working by recursion also construct trees as they follow their search paths. These trees are always *rooted trees;* that is, they begin at a distinguished node, called the *root vertex,* and are usually built outwards from the root. Figure 5.1 shows several rooted trees, where the root vertex is shaded black.

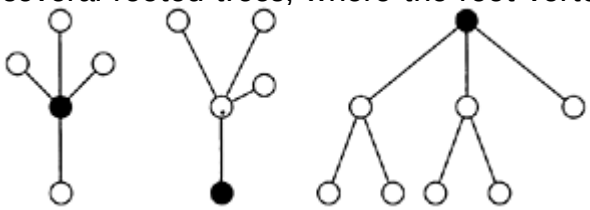


FIGURE 5.1
Several rooted trees

If T is a tree, then any vertex u can be chosen as the root, thereby making T into a rooted tree. A rooted tree can always be decomposed into *branches*. The tree T shown in Figure 5.2 has three branches $B1$, $B2$, and $B3$.

DEFINITION 5.1: Let T have root vertex u. The *branches* of T are the maximal subtrees in which u has degree one.

Thus, the root is in every branch, but the branches have no other vertices in common. The number of branches equals the degree of the root vertex. If we know the branches of some tree T, then we can easily recombine them to get T.

Page 90

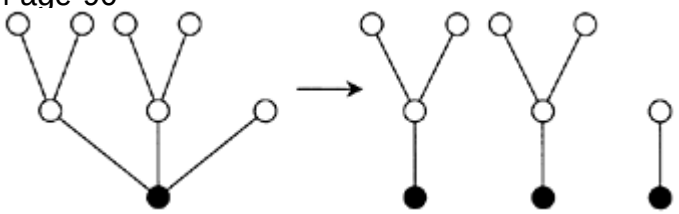


FIGURE 5.2
Decomposition into branches

Therefore, two rooted trees have the same structure; that is, they are isomorphic, if and only if they have the same number of branches, and their branches have the same structure.

Any vertex of a tree which has degree one is called a *leaf*. If the root is a leaf, then T is itself a branch. In this case, let u be the unique vertex adjacent to u, the root of T. Then $T' = T - u$ is a rooted tree, with root u. This is illustrated in Figure 5.3. T' can then be further broken down into branches, which can in turn be reduced to rooted trees, etc. This gives a recursive decomposition of rooted trees into branches, and branches into rooted trees.

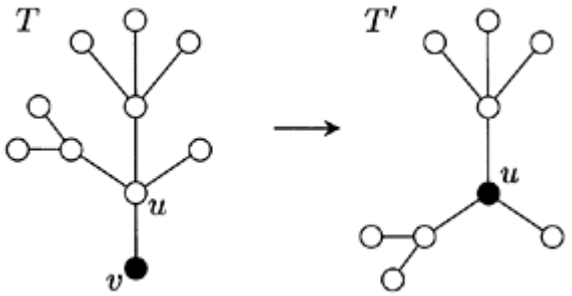


FIGURE 5.3
Reducing a branch to a rooted tree

This technique can be developed into a method for determining when two rooted trees have the same structure.

5.2 Non-rooted trees

All non-rooted trees on five and fewer vertices are displayed in Figure 5.4. Table 5.1 gives the number of trees up to 10 vertices.

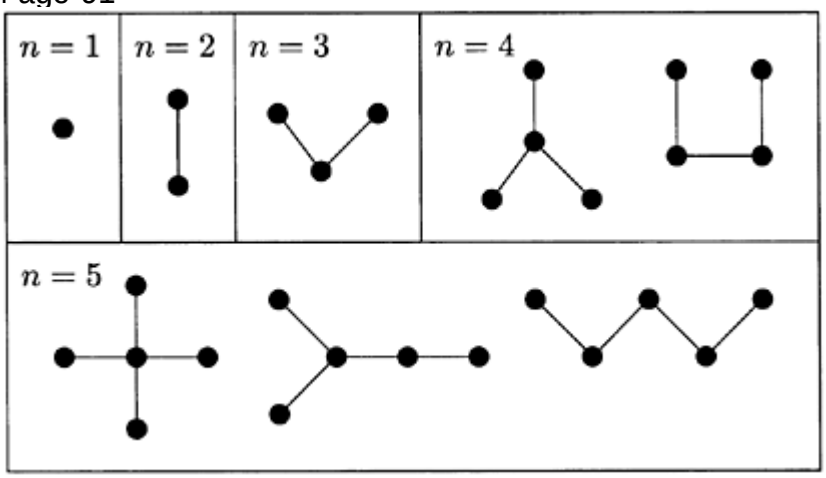


FIGURE 5.4
The trees on five and fewer vertices

If a leaf be removed from a tree on n vertices, a tree on $n-1$ vertices is obtained. Thus, one way to list all the trees on n vertices is to begin with a list of those on $n-1$ vertices, and add a leaf in all possible ways, discarding duplicates. How can we recognize when two trees have the same structure? We shall see that non-rooted trees can always be considered as rooted trees, by choosing a special vertex as root, in the *center* of *T,* denoted CTR*(T)*. The center is defined recursively.

DEFINITION 5.2: Let *T* be a tree on *n* vertices.

1. If $n=1$, say $V(T)=\{u\}$, then $\mathrm{CTR}(T)=u$.

2. If $n=2$, say $V(T)=\{u,u\}$, then $\mathrm{CTR}(T)=uu$.

3. If $n>2$, then T has at least two leaves. Delete all the leaves of *T* to get a tree *T′*. Then $\mathrm{CTR}(T)=\mathrm{CTR}(T')$.

Thus the center of a tree is either a vertex or an edge, since eventually case (1) or (2) of the definition is used in determining the center of *T*. Trees whose centre consists of a single vertex are called *central trees*. Trees with two vertices in the center (i.e., CTR*(T)* is an edge) are called *bicentral trees*. Figure 5.5 shows two trees, one central and one bicentral.

A central tree can always be considered a rooted tree, by taking the centre as the root. A bicentral tree can also be considered a rooted tree, but we must have a means of deciding which of two vertices to take as the root. Thus we can say that every tree is a rooted tree.

TABLE 5.1
The number of trees up to 10 vertices

n	# trees
2	1
3	1

4	2
5	3
6	6
7	11
8	23
9	47
10	106

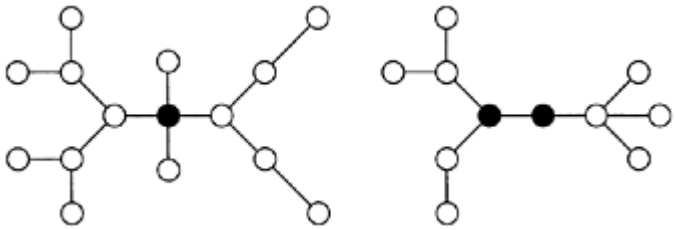


FIGURE 5.5
A central tree and a bicentral tree

Exercises

5.2.1 Find the centre of the trees shown in Figures 5.1 and 5.3.
5.2.2 Prove that any longest path in a tree *T* contains the center.
5.2.3 Prove that *T* is central if DIAM*(T)* is even, and bicentral if DIAM*(T)* is odd.
5.2.4 A *binary tree* is a rooted tree such that the root vertex has degree two, and all other vertices which are not leaves have degree three. Show that if *T* is a binary tree on *n* vertices, that *n* is odd, and that *T* has (*n*+1)/2 leaves.

5.3 Read's tree encoding algorithm

There are a number of interesting algorithms for encoding trees. Here we present one of Read's algorithms. It is basically an algorithm to find CTR*(T),* keeping certain information for each vertex as it progresses. When the centre is reached, a root node is uniquely chosen. Read's algorithm encodes a tree as an integer. Its

description reveals a number of interesting properties satisfied by trees.
Let *T* be a tree whose centre is to be found. Instead of actually deleting the leaves of *T,* let us simply draw a circle around each one. Draw the circle in such a way that it encloses all the circles of any adjacent nodes which have been previously circled. The last vertices to be circled form the centre.

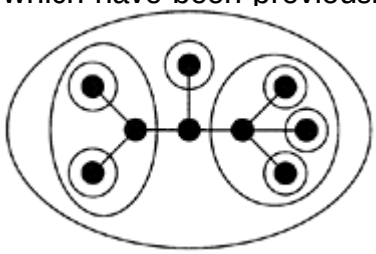


FIGURE 5.6
A circled tree

This system of nested circles can be redrawn in various ways.

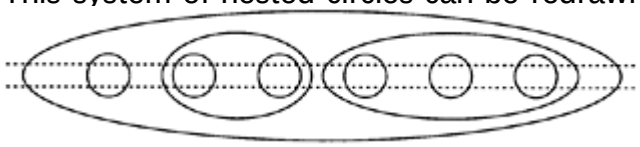


FIGURE 5.7
Nested circles

Each circle corresponds to a vertex of *T*. The largest circle which encloses the entire system corresponds to the centre of *T*. Two circles correspond to adjacent vertices if and only if one circle is nested inside the other. The circles not containing a nested circle are the leaves of *T*. If we cut off the top and bottom of each circle in Figure 5.7, we are left with a set of matched parentheses: (()(()())(()()())). By writing 0 for each left parenthesis and 1 for each right parenthesis, this can be considered a binary number, 001001011001010111, which represents an integer.
The internal circles in Figure 5.6 have been sorted and arranged in order of increasing complexity. For example, the first inner circle can be denoted 01. This is less than the second circle, which can be denoted 001011, which in turn is less that the third circle 00101011, considered as binary numbers. Thus, there is a

natural ordering associated with these systems of nested circles.
The binary number associated with each vertex *u* is called its *tag*, denoted *t(u)*. Initially each leaf has a tag of 01. The algorithm to find CTR*(T)* constructs the vertex tags as it proceeds.

Page 94

Algorithm 5.3.1: TREEENCODE*(T)*

comment: constructs an integer tree code to represent the tree *T*

repeat
construct *L(T)*, the set of leaves of *T*, stored on an array
for each leaf $v \in L(T)$
do {
comment: suppose that $v \longrightarrow \{u_1, u_2, \ldots, u_k\} \subseteq L(T)$
sort the tagged vertices adjacent to v by tag,
say $t(u_1) \leq t(u_2) \leq \cdots \leq t(u_k)$
$t(v) \leftarrow 0t(u_1)t(u_2)\cdots t(u_k)1$ "concatenate them"
T←*T*−*L(T)* "just *mark* the vertices deleted"
until all of *T* has been tagged
if *L(T)* contains one vertex *u*
then return *(t(u))*
else {
comment: $L(T)$ contains two vertices u and v, say $t(u) \leq t(v)$
return $(0t(u)t'(v))$

On the last iteration, when the centre was found, either one or two vertices will have been tagged. They form the centre of *T*. If *T* is a central tree, with CTR*(T)*=*u*, we choose *u* as the root of *T*. Then *t(u)*, the tag of the centre, represents the entire system of nested circles. It is chosen as the encoding of *T*.
If *T* is a bicentral tree, with centre *uu*, we must decide which vertex to choose as the root of *T*. We arbitrarily choose the one with the larger tag. Suppose that *t(u)*≤*t(u)*, so that *u* is chosen as the root. The code for the entire tree is formed by altering the enclosing circle of *u* so as to enclose the entire tree. This is illustrated in Figure 5.8.

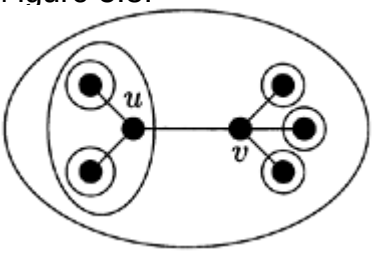


FIGURE 5.8
Choosing the root vertex

Thus, the tree code for *T* in the bicentral case is the concatenation 0*t(u)t′(u)*, where *t′(u)* is formed from *t(u)* by dropping one initial 0-bit.

Page 95

If *t(u)*=*t(u)* for some bicentral tree *T*, then we can obviously select either *u* or *u* as the root vertex.
The easiest way to store the tags *t(u)* is as an integer array *t[u]*. We also need to store the length *ℓ[u]*, of each tag, that is, its length in bits. Initially each leaf *u* has *t[u]*=1 and *ℓ[u]*=2. To concatenate the tags 0*t*(*u*1)*t*(*u*2)… *t(uk)*1 we use a loop.

t[u]←0
for *i*←1 **to** *k*
do {
shift $t[v]$ left $\ell[v]$ bits
$t[v] \leftarrow t[v] + t[u_i]$
$\ell[v] \leftarrow \ell[v] + \ell[u_i]$
t[u]←2*t[u]*+1
ℓ[u]←*ℓ[u]*+2

If a primitive left shift operation is not available, one can always store a table of powers of 2, and use

multiplication by $2^{\ell[u_i]}$ in order to shift *t[u]* left by *ℓ[ui]* bits.

5.3.1 The decoding algorithm

If the circles of Figure 5.6 are unnested, they can be redrawn so as to emphasize their relation to the structure of *T*.

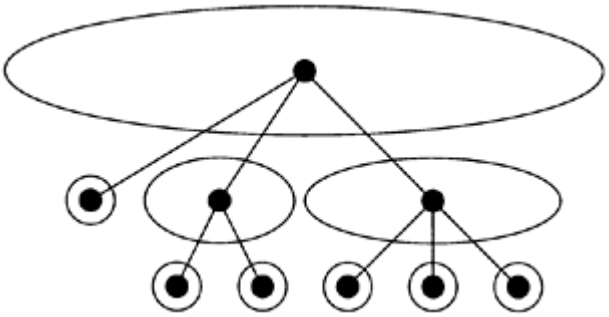


FIGURE 5.9
Unnested circles

The decoding algorithm scans across the system of nested circles. Each time a new circle is entered, a vertex is assigned to it. The first circle entered is that corresponding to the root vertex. The decoding algorithm uses a global vertex counter *k,* which is initially zero, and constructs a global tree *T*. It can be programmed to scan the tree code from right to left as follows:

Algorithm 5.3.2: TREEDECODE*(Tcode, ui)*
comment: a new circle has just been entered, from circle *ui*.
k←*k*+1 “create a new vertex”
join $v_k \longrightarrow v_i$
Tcode←*Tcode*/2 “shift right 1 bit”
while *Tcode* is odd

do {
comment: the rightmost bit = 1, a new circle is entered
TREEDECODE$(Tcode, v_k)$
$Tcode \leftarrow Tcode/2$ “shift right 1 bit”

The easiest way to use Algorithm 5.3.2 is to create a dummy vertex *u*0 which will only be joined to the root vertex, *u*1 and then delete *u*0 once the tree has been constructed.

k←0
TREEDECODE(*Tcode,u*0)
delete *u*0

Exercises

5.3.1 Encode the trees of Figure 5.5 into nested circles by hand. Write down their tree codes.
5.3.2 Work through Algorithm 5.3.2 by hand, for the tree codes 001011 and 0001011011.
5.3.3 Write Algorithm 5.3.2 so as to scan the tree code from left to right, using multiplication by 2 to shift *Tcode* to the right, and using the sign bit of the code to test each bit. Assume a word length of 32 bits.
5.3.4 Program the encoding and decoding algorithms.
5.3.5 If *T* has *n* vertices, what is the total length of its tree code, in bits? How many 1’s and 0’s does the code contain? How many leading 0’s does the code begin with? What is the maximum value that *n* can be if *T* is to be encoded in 32 bits?
5.3.6 Let *T* be a tree. Prove that the tree obtained by decoding TREEENCODE*(T)*, using the decoding algorithm, is isomorphic to *T*.
5.3.7 Let *T*1 and *T*2 be two trees. Prove that $T_1 \cong T_2$ if and only if TREEENCODE(*T*1)= TREEENCODE(*T*2).
5.3.8 Consider the expression *x*1*x*2…*xn*+1, where *x*1, *x*2,..., *xn*+1 are variables. If parentheses are inserted so as to take exactly two terms at a time, we obtain a valid bracketing of the expression, with *n* pairs of matched parentheses (e.g., ((*x*1(*x*2*x*3)) *x*4), where *n*=3). Each pair of matched parentheses contains exactly two terms. Describe the type of rooted tree on *n* vertices that corresponds to such

a valid bracketing. They are called *binary plane* trees. Each leaf of the tree corresponds to a variable *xi* and each internal node corresponds to a pair of matched parentheses, giving 2*n*+1 vertices in total.

5.3.9 Let *pn* denote the number of binary plane trees with *n* leaves (e.g., *p*0=0, *p*1= 1, *p*2=1, *p*3=2, etc.). We take *p*1=1 corresponding to the tree consisting of a single node. Let *p(x)=p*0+*p*1*x*+*p*2*x*2+...be the *generating function* for the numbers *pn,* where *x* is a variable. If *T*1 and *T*2 are two binary plane trees with n1 and n2 leaves, respectively, then they can be combined by adding a new root vertex, adjacent to the roots of *T*1 and *T*2. This gives a binary plane tree with *n*1+*n*2 leaves. There are $p_{n_1}p_{n_2}$ ways of constructing a tree in this way. This term arises from *p*2*(x)* as part of the coefficient of $x^{n_1+n_2}$. This holds for all values of *n*1 and *n*2. Therefore, we can write *p(x)=x+p*2*(x)*. Solve this identity for *p(x)* in terms of *x,* and then use the binomial theorem to write it as a power series in *x*. Finally, obtain a binomial expression for *pn* in terms of *n*.

The numbers *pn* are called the *Catalan numbers*. (The answer should be $p_n = \frac{1}{2n-1}\binom{2n-1}{n}$.

5.4 Generating rooted trees

One way to generate a list of all the trees on *n* vertices would be to add a new leaf to the trees on *n*−1 vertices in all possible ways, and to discard duplicates, using the tree codes to identify isomorphic trees. However, they can also be generated directly, one after the other, with no duplicates.

Let *T* be a central tree, rooted at its centre *u.* Decompose *T* into its branches *B*1, *B*2,..., *Bk*. Each branch *Bi* is also rooted at *u*. Write *T*=(*B*1, *B*2,...,*Bk*) to indicate the decomposition into branches. Since *u* is the centre of *T,* it is the middle vertex of every longest path in *T*. Therefore, the two "tallest" branches of *T* will have equal height, where we define the *height of a branch B* rooted at *u* as $h(B) = \text{MAX}\{\text{DIST}(v, w) \mid w \in B\}$. If the branches of the central tree *T* have been ordered by height, so that *h*(*B*1)≥*h*(*B*2)≥...≥*h*(*Bk*), then we know that *h*(*B*1)=*h*(*B*2). Any rooted tree for which the two highest branches have equal height is necessarily rooted at its centre. Therefore, when generating central trees, the branches must be ordered by height.

Generating the rooted trees on *n* vertices in a sequence implies a linear ordering on the set of of all rooted trees on *n* vertices. In order to construct a data structure representing a rooted tree *T* as a list of branches, we also require a linear order on the set of all branches. Then we can order the branches of *T* so that *B*1≥*B*2≥ ...≥*Bk*. This will uniquely identify *T,* as two trees with the same set branches will have the same ordered set of branches. The smallest possible branch will evidently be of height one, and have two vertices—it is *K*1 rooted at a vertex. The tree shown in Figure 5.10 has five branches of height one; call them *elementary branches*. The next smallest branch is of height two, and has three vertices—it is

the path *P*2 rooted at a leaf.

Associated with each branch *B* is a rooted tree, as shown in Figure 5.3, constructed by advancing the root to its unique adjacent vertex, and deleting the original root. We will use the ordering of rooted trees to define recursively an ordering of all branches, and the ordering of all branches to define recursively an ordering of all rooted trees. We know that all branches on at most three vertices have already been linearly ordered.

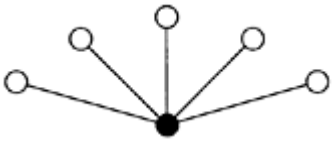


FIGURE 5.10
A tree with all branches of height one

DEFINITION 5.3: Suppose that all branches on at most *m*≥2 vertices have been linearly ordered. Let *T*=(*B*1, *B*2,..., *Bk*) and $T' = (B'_1, B'_2, \ldots, B'_\ell)$, where *k*+*ℓ*≥3, be any two distinct rooted trees with given branches, such that each branch has at most *m* vertices, ordered so that *B*1≥*B*2≥...≥*Bk* and $B'_1 \geq B'_2 \geq \cdots \geq B'_\ell$. Suppose that |*T*|≤|*T′*|. Then *T* and *T′* are compared as follows:

1. If |*T*|<|*T′*|, then *T*<*T′*.

2. Otherwise, compare (*B*1, *B*2,..., *B*k) and $(B'_1, B'_2, \ldots, B'_\ell)$ lexicographically. That is, find *i,* the first subscript such that $B_i \neq B'_i$; then *T* < *T′* if $B_i < B'_i$.

The first condition is to ensure that all rooted trees on *n* vertices precede all trees on *n*+1 vertices in the linear order. The second condition defines an ordering of trees based on the lexicographic ordering of branches. Notice that if *k*≠*ℓ* there must be an *i* such that $B_i \neq B'_i$; for if every $B_i = B'_i$, but *k*≠*ℓ*, then *T* and *T′* would have different numbers of vertices, so that condition (1) would apply. This defines a linear ordering on all rooted trees whose branches all have at most *m* vertices. This includes all trees on *m*+1 vertices with at least two branches. In fact, it includes all rooted trees on *m*+1 vertices, except for the path *Pm*, rooted at a leaf. As this is a branch on *m*+1 vertices, it is handled by Definition 5.4.

We now have an ordering of rooted trees with at least two branches, based on the ordering of branches. We

can use it in turn to extend the ordering of branches. Let B and B' be two distinct branches. In order to compare B and B', we advance their roots, as in Figure 5.3, to the unique adjacent vertex in each, and delete the

original root. Let the rooted trees obtained in this way be T and T'. respectively. Then $B<B'$ if $T<T'$. In summary, branches are compared as follows:

DEFINITION 5.4: Suppose that all rooted trees on $\leq m-1$ vertices have been linearly ordered, and that all rooted trees on m vertices with at least two branches have also been linearly ordered. Let B and B' be branches on m vertices, with corresponding rooted trees T and T'. Suppose that $|B| \leq |B'|$ and that if $|B| = |B'|$, then B is the branch of smaller height. Then B and B' are compared as follows:

1. If $|B|<|B'|$, then $B<B'$.

2. Otherwise, if $h(B)<h(B')$ then $B<B'$.

3. Otherwise, $B<B'$ if $T<T'$.

We have a recursive ordering which compares trees by the ordering of their branches, and branches by the ordering of their trees. We must prove that the definition is valid.

THEOREM 5.1 Definitions 5.3 and 5.4 determine a linear order on the set of all rooted trees.

PROOF Notice that rooted trees have a sub-ordering based on the number of vertices—all rooted trees on n vertices precede all rooted trees on $n+1$ vertices. Branches have an additional sub-ordering based on height—all branches of height h on n vertices precede all branches of height $h+1$ on n vertices. A branch is a special case of a rooted tree, in which the root vertex has degree one. If a branch B and tree T on n vertices are compared, where T has at least two branches, then that first branch of T has fewer than n vertices, so that $T<B$, by Definition 5.3. Therefore all trees whose root vertex has degree two or more precede all branches on n vertices.

The definitions are clearly valid for all rooted trees on ≤ 3 vertices. Suppose that the set of all rooted trees on $\leq n$ vertices is linearly ordered by these definitions, and consider two distinct rooted trees

$$T=(B1,B2,\ldots,Bk)$$

and

$$T' = (B'_1, B'_2, \ldots, B'_\ell)$$

on $n+1$ vertices. If $k=\ell=1$, then T and T' are both branches. The trees formed by advancing their root vertices have only n vertices, and so can be compared by Definition 5.3. Otherwise at least one of T and T' has two or more branches.

Therefore at least one of each pair Bi and B'_i of branches has $\leq n$ vertices. Therefore the branches Bi and B'_i can be compared by Definition 5.4. The conclusion follows by induction.

In this ordering of branches and trees, the first rooted tree on n vertices is a *star* consisting of the tree $K1,n-1$ rooted at its centre. The last rooted tree on n vertices is a path Pn, rooted at a leaf. The first branch on n vertices is $K1,n-1$, rooted at a leaf, and the last branch on n vertices is also the path Pn, rooted at a leaf. The first few rooted trees are shown in Figure 5.11.

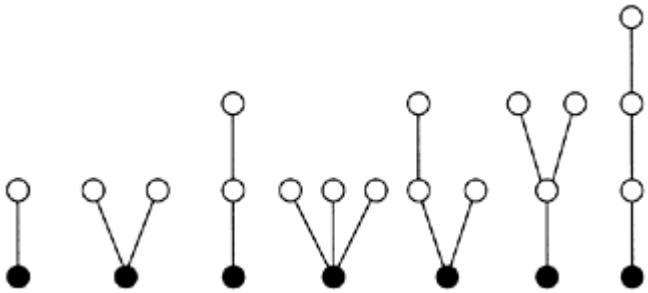


FIGURE 5.11
The beginning of the linear order of rooted trees

Let $T=(B1,B2,\ldots,Bk)$ be the list of branches of a rooted tree T, with root vertex u. The recursive data structure we use to represent T is a linked list of branches. Each branch Bi also has root u. It is in turn represented in terms of the rooted tree T', whose root vertex is the unique vertex adjacent to u. Thus, a record representing a tree T has four fields:

- $NodeNum\langle T\rangle$, the node number of the root, which is used for printing.

- $nNodes\langle T\rangle$, the number of vertices in the tree.
- $FirstBranch\langle T\rangle$, a pointer to the first branch.
- $LastBranch\langle T\rangle$, a pointer to the last branch.

Each branch *B* of *T* has a corresponding rooted tree *T′*. *B* is represented as a record having four fields:

- $Height\langle B\rangle$, the height of the branch.
- $NextRoot\langle B\rangle$, a pointer to the rooted tree *T′*.
- $NextBranch\langle B\rangle$, a pointer to the next branch of *T*.
- $PrevBranch\langle B\rangle$, a pointer to the previous branch of *T*.

Page 101

It is not necessary to store the number of vertices of a branch *B,* as it is given by $\langle NextRoot\langle B\rangle\rangle+1$. The functions which compare two trees and branches are given as follows. They return an integer, whose value is one of three constants, *LessThan, EqualTo,* or *GreaterThan*.

Algorithm 5.4.1: COMAPARETREES(*T*1, *T*2)
comment: *T*1 and *T*2 both have at least one branch
if |*T*1|<|*T*2**| then return** *(LessThan)*
if |*T*1|>|*T*2| **then return** *(GreaterThan)*
comment: otherwise |*T*1|=|*T*2|
$B_1 \leftarrow FirstBranch\langle T_1\rangle$
$B_2 \leftarrow FirstBranch\langle T_2\rangle$
Result←COMPAREBRANCHES (B1, *B*2)
while *Result*=*EqualTo*

$$\textbf{do}\ \begin{cases} \textbf{if } B_1 = LastBranch\langle T_1\rangle \textbf{ then return } (EqualTo) \\ B_1 \leftarrow NextBranch\langle B_1\rangle \\ B_2 \leftarrow NextBranch\langle B_2\rangle \\ Result \leftarrow \textsc{CompareBranches}(B_1, B_2) \end{cases}$$

return *(Result)*

Algorithm 5.4.2: COMPAREBRANCHES (*B*1, *B*2)
comment: *B*1 and *B*2 both have a unique vertex adjacent to the root
if |*B*1|<|*B*2|**then return** *(LessThan)*
if |*B*1|>|*B*2| **then return** *(GreaterThan)*
comment: otherwise |*B*1|=|*B*2|
$Height\langle B_1\rangle < Height\langle B_2\rangle$ **then return** *(LessThan)*
$Height\langle B_1\rangle > Height\langle B_2\rangle$ **then return** *(GreaterThan)*
comment: otherwise $Height\langle B_1\rangle = Height\langle B_2\rangle$
if $Height\langle B_1\rangle = 1$ **then return** *(EqualTo)*
$T_1 \leftarrow NextRoot\langle B_1\rangle$
$T_2 \leftarrow NextRoot\langle B_2\rangle$
return (COMPARETREES (*T*1, *T*2))

Using these functions we can generate all rooted trees on *n* vertices, one after the other, beginning with the first tree, which consists of a root vertex and *n*−1 elementary branches of height one, until the last tree is reached, which has only one branch, of height *n*−1. Written in pseudo-code, the technique is as follows,

Page 102

where NEXTTREE*(T)* is an procedure which replaces *T* with the next tree, and returns **true** unless *T* was the last tree (see Algorithm 5.4.3). FIRSTTREE*(n)* is a procedure that constructs the first tree on *n* vertices.

T←FIRSTTREE*(n)*
repeat
PRINTTREE*(T)*
until not NEXTTREE*(T)*

Suppose that *T* has branch decomposition (*B*1, *B*2,..., *Bk*), where *B*1≥ *B*2≥...≥*Bk*. The procedure NEXTTREE*(T)* works by finding the last branch *Bi* such that *Bi*≠*Bi*−1. Then *Bi, Bi*+1,..., *Bk* is a sequence of isomorphic branches. So long as *Bi* is not simply a path of length *h(Bi),* there is a larger branch with the same number of vertices. *Bi* is then replaced with the next larger branch and the subsequent branches *Bi*+1, *Bi*+2, ..., *Bk* are replaced with a number of elementary branches. This gives the lexicographically next largest tree. This is illustrated in Figure 5.12. Here, *B*2 was replaced with a branch of height three, and *B*3 was replaced with three elementary branches.

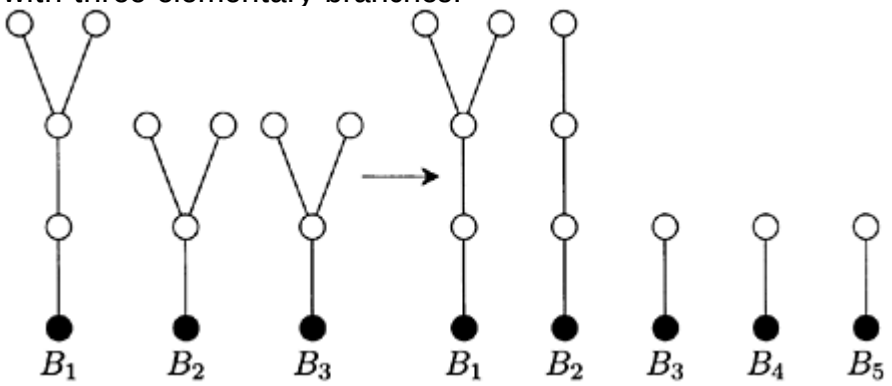


FIGURE 5.12
Constructing the next tree

But if *Bi* is simply a path, then it is the last branch with |*Bi*| vertices. In order to get the next branch we must add another vertex. *Bi* is then replaced with the first branch with one more vertex. This is the unique branch with |*Bi*|+1 vertices and height two. *T* is then filled in with as many elementary branches as needed. This is illustrated in Figure 5.13.

Page 103

Algorithm 5.4.3: NEXTTREE*(T)*

$B_1 \leftarrow LastBranch\langle T\rangle$
$B_1 = FirstBranch\langle T\rangle$
then { **comment:** only one branch – advance the root
if $nNodes\langle T\rangle = Height\langle B_1\rangle + 1$ **then return** (**false**)
return (NEXTTREE($NextRoot\langle B_1\rangle$)) }
comment: otherwise at least two branches
$B_2 \leftarrow PrevBranch\langle B_1\rangle$
while COMPAREBRANCHES(*B*1, *B*2)=*EqualTo*
do { $B_1 \leftarrow B_2$
if $B_1 = FirstBranch\langle T\rangle$ **then go to** 1
$B_2 \leftarrow PrevBranch\langle B_2\rangle$ }
1: **comment:** delete the branches of *T* following *B*1
N← DELETEBRANCHES(*T*, *B*1) "*N* nodes are deleted"
comment: replace *B*1 with next branch, if possible
if $nNodes\langle NextRoot\langle B_1\rangle\rangle > Height\langle B_1\rangle$
then { NEXTTREE($NextRoot\langle B_1\rangle$)
fill in T with N elementary branches
return (**true**) }
comment: otherwise construct the first branch with one more node
if *N*>0 **then** { $M \leftarrow$ DESTROYTREE($NextRoot\langle B_1\rangle$) "$M$ nodes are deleted"
$NextRoot\langle B_1\rangle \leftarrow$ FIRSTTREE($M+1$)
fill in T with $N-1$ elementary branches
return (**true**) }
comment: otherwise there's no branch following *B*1 to take a node from

repeat
*B*1←*B*2
if $B_1 = FirstBranch\langle T \rangle$ **then go to** 2 $B_2 \leftarrow PrevBranch\langle B_1 \rangle$
until COMPAREBRANCHES (*B*1, *B*2)≠*EqualTo*
2: **comment:** delete the branches of *T* following *B*1
N←*DEL*ETEBRANCHES(T, B1) "*N* nodes are deleted"
comment: replace *B*1 with next branch
if *nNodes* $\langle NextRoot\langle B_1 \rangle \rangle > Height\langle B_1 \rangle$
then $NextRoot\langle B_1 \rangle$
else $NextRoot\langle B_1 \rangle \leftarrow$ FIRSTTREE(nNodes$\langle NextRoot\langle B_1 \rangle \rangle + 1$+1) fill in *T* with elementary branches
return (true)

Page 104

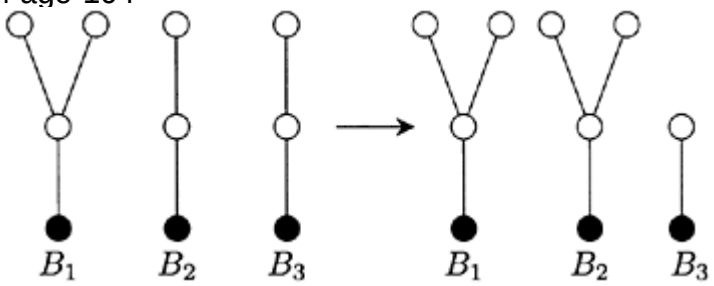


FIGURE 5.13
Constructing the next tree

The procedure DELETEBRANCHES(*T, B*1) destroys all branches of *T* following *B*1, and returns the number of nodes deleted. Similarly DESTROYTREE*(T)* is a procedure that destroys all branches of *T,* and returns the total number of nodes deleted.

THEOREM 5.2 Let T be a tree on n vertices. Algorithm NEXTTREE*(T) constructs the next tree on n vertices after T in the linear order of trees, if there is one*.

PROOF The proof is by induction on *n*. It is easy to check that it works for trees on *n*=2 and *n*=3 vertices. Suppose that it holds up to *n*−1 vertices, and let *T* have *n* vertices. Let *T*=(*B*1, *B*2,..., *Bk*) be the branches of *T,* where *B*1≥*B*2≥...≥*Bk*. The algorithm first checks whether there is only one branch. If so, and *T* is a branch of height *n*−1, it returns **false**. Otherwise let *T′* be the rooted tree corresponding to *B*1 by advancing the root. The algorithm calls NEXTTREE*(T′)*. Since *T′* has *n*−1 vertices, this gives the next branch following *B*1 in the linear order, as required.

Otherwise, *T* has at least two branches. It finds the branch *Bi* such that *Bi*= *Bi*+1=...=*Bk,* but *Bi*−1≠*Bi,* if there is one (possibly *i*=1). The first tree following *T* must differ from *T* in *Bi,* unless *i*=*k* and *Bi* is a path. In the first case, the algorithm replaces *Bi* with the next branch in the linear order, and fills in the remaining branches of *T* with elementary branches. This is the smallest tree on *n* vertices following *T*. Otherwise *i*=*k* and *Bi* is a path, so that there is no tree following *Bi*. Since there are at least two branches, the algorithm finds the branch *Bj* such that *Bj*=*Bj*+1=...=*Bk*−1>*Bk* (possibly *j*=1). It then replaces *Bj* with the first branch following it, and fills in *T* with elementary branches. In each case the result is the next tree after *T*.

Page 105

Exercises

5.4.1 Work through the NEXTTREE*(T)* algorithm by hand to construct all the rooted trees on 4, 5, 6, and 7 vertices.

5.4.2 Write the recursive procedure PRINTTREE*(T)* to print out a tree as shown in Figure 5.14, according to the distance of each vertex from the root.

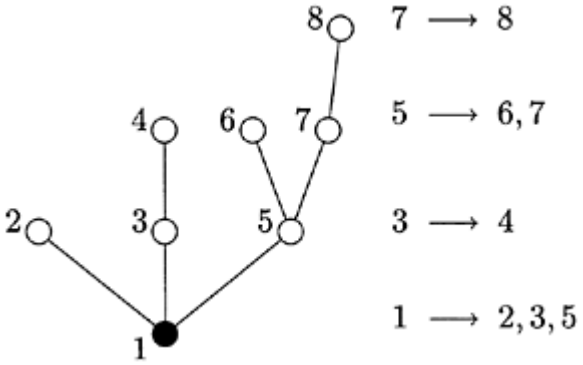


FIGURE 5.14
Printing a rooted tree

5.4.3 Write the recursive functions DESTROYTREE*(T)* and DELETEBRANCHES(*T,B*1), both of which return the number of vertices deleted.

5.4.4 Program the NEXTTREE*(T)* algorithm, and use it to find the number of rooted trees up to 10 vertices.

5.5 Generating non-rooted trees

The NEXTTREE*(T)* algorithm generates the rooted trees on *n* vertices in sequence, beginning with the first tree of height 1 and ending with the tree of height *n*−1. In order to generate non-rooted trees, we must root them in the centre. Since every non-rooted tree can be viewed as a rooted tree, all non-rooted trees also occur in the linear order of trees. Central trees can be generated by modifying NEXTTREE*(T)* so that the two highest branches are always required to have equal height. This can be done with another procedure, NEXTCENTRALTREE*(T),* which in turn calls NEXTTREE*(T)* when forming the next branch. Bicentral trees are slightly more difficult, since the highest branches *B*1 and *B*2 satisfy $h(B1)=h(B2)+1$. If we generate trees in which the heights of the two highest branches differ by one, then most bicentral trees will be constructed twice, once for each vertex in the centre. For example, Figure 5.15 shows two different branch

decompositions of the same bicentral tree. It would therefore be generated twice, since it has different branch decompositions with respect to the two possible roots.

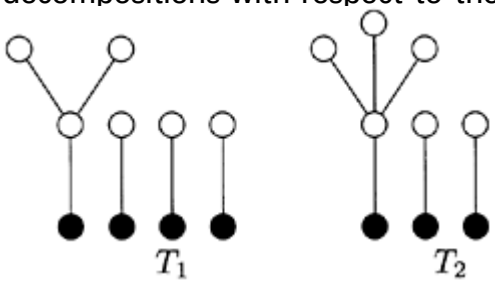


FIGURE 5.15
Two decompositions of a bicentral tree

The easiest solution to this is to subdivide the central edge with a new vertex, taking it as the root. Then each bicentral tree on *n* vertices corresponds to a unique central tree on *n*+1 vertices, with exactly two branches. We can construct these by generating rooted trees with only two branches, which have equal height, and then ignoring the extra root vertex.

Exercises

5.5.1 Write and program the procedures NEXTCENTRALTREE*(T)* and NEXTBICENTRALTREE*(T),* and use them to construct all the non-rooted trees on *n* vertices, up to *n*=15.

5.6 Prüfer sequences

Read's algorithm encodes a tree according to its isomorphism type, so that isomorphic trees have the same code. This can be used to list all the isomorphism types of trees on *n* vertices. A related question is to make a list all the trees on the *n* vertices *Vn*={1, 2,..., *n*}. These are sometimes referred to as *labeled trees*. For example, Figure 5.16 illustrates the three distinct, or labeled trees on three vertices, which are all isomorphic to each other.

Let *T* be a tree with *V(T)*={1, 2,..., *n*}. A *Prüfer sequence* for *T* is a special encoding of *T* as an integer sequence. For example, the tree of Figure 5.17 with *n*=9 has Prüfer sequence *t*=(6, 9, 1, 4, 4, 1, 6).

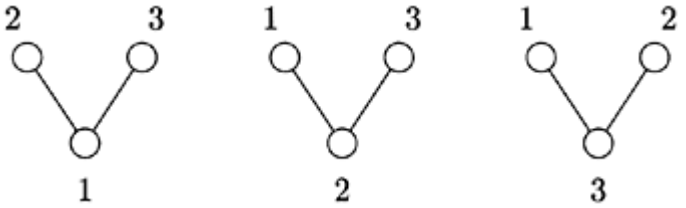


FIGURE 5.16
Three distinct trees on three vertices

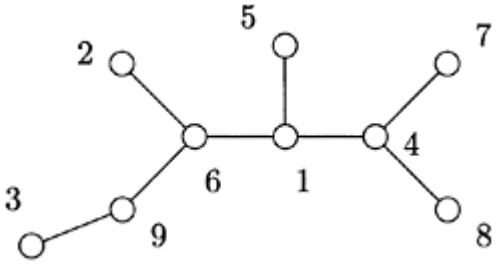


FIGURE 5.17
Finding the Prüfer sequence of a tree

This is constructed as follows. The leaves of *T* are *L(T)*={2, 3, 5, 7, 8}. The numerically smallest leaf is 2. Since 2→6, we take *t*1=6 as the first member of *t*. We now set *T:*=*T*−2, and find *L(T)*={3, 5, 7, 8}. We again choose the smallest leaf, 3, and since 3→9, we take *t*2=9 as the second member of the sequence, and set *T:*=*T*−3. Notice that when 3 is deleted, 9 becomes a leaf. Therefore, on the next iteration we will have *L(T)*={5, 7, 8, 9}. The general technique is the following:

for *k*←1 to *n*−2

$$\textbf{do}\begin{cases}\text{find } L(T)\\ \text{select } v \in L(T), \text{ the smallest leaf}\\ t_k \leftarrow \text{ the unique vertex adjacent to } v\\ T \leftarrow T - v\end{cases}$$

comment: *T* now has 2 vertices left

This always gives a sequence of *n*−2 integers *t*=(*t*1, *t*2,..., *t*n−2), where each $t_i \in V_n$. Notice that at each step, a leaf of *T* is deleted, so that *T* is always a tree throughout all the steps. Since *T* is a tree, we can always choose a leaf to delete. When *T* reduces to a single edge, the algorithm stops. Therefore no leaf of *T* is ever chosen as any *tk*. In fact, if *DEG(u)*≥2, then *u* will appear in *t* each time a leaf adjacent to *u* is deleted. When the degree drops to one, *u* itself

becomes a leaf, and will appear no more in *t*. Therefore, each vertex *u* appears in *t* exactly DEG*(u)*−1 times.

THEOREM 5.3 Let*t*=(*t*1, *t*2,..., *tn*−2) *be any sequence where each* $t_i \in V_n = \{1, 2, \ldots, n\}$. *Then t is the Prüfer sequence of a tree T on n vertices.*

PROOF The sequence *t* consists of *n*−2 integers of *Vn*. Therefore at least two members of *Vn* are not used in *t*. Let *L* be those numbers not used in *t*. If *t* were formed by encoding a graph using the above technique, then the smallest element $v \in L$ must have been a leaf adjacent to *t*1. So we can join *u*→*t*1, and discard *t*1 from *t*. Again we find *L*, the numbers not used in *t*, and pick the smallest one, etc. This is summarized in the following pseudo-code:

N←{1, 2,..., *n*}

for *k*←1 **to** *n*−2

$$\textbf{do}\begin{cases}\text{construct } L, \text{ those numbers of } N \text{ not used in } t\\ \text{select the smallest } v \in L\\ \text{join } v \longrightarrow t_k\\ \text{discard } t_k \text{ from } t\\ N \leftarrow N - v\end{cases}$$

comment: *T* now has 2 vertices left, *u* and *u* join $u \longrightarrow v$

This creates a graph *T* with *n*−1 edges. It is the *only* graph which could have produced the Prüfer sequence *t*, using the above encoding technique. Must *T* be a tree? If *T* were not a tree, then it could not be connected,

since T has only $n-1$ edges. In that case, some component of T would have a cycle C. Now the encoding technique only deletes leaves. No vertex on a cycle could ever be deleted by this method, for the degree of every $u \in C$ is always at least two. This means that a graph containing a cycle would not produce a Prüfer sequence of length $n-2$. Therefore T can have no cycle, which means that it must be a tree.
Thus we see that every tree with vertex set $\{1,2,\ldots,n\}$ corresponds to a unique Prüfer sequence, and that every sequence t can only be obtained from one tree. The corollary is that the number of trees equals the number of sequences. Now it is clear that there are n^{n-2} such sequences, since each of the $n-2$ elements t_k can be any of the numbers from 1 to n. This result is called Cayley's theorem.
THEOREM 5.4 **(Cayley's theorem)** *The number of distinct trees on n vertices is* n^{n-2}.

Page 109

5.7 Spanning trees

Consider the problem of making a list of all the spanning trees of a graph G. If $G \cong K_n$, then there are n^{n-2} spanning trees, and each one corresponds to a Prüfer sequence. If $G \ncong K_n$, then we can find all the spanning trees of G as follows. Choose any edge uv of G. First find all the spanning trees that use uv and then find all the trees that don't use uv. This gives all spanning trees of G. Write $\tau(G)$ for the number of spanning trees of G. The spanning trees T that don't use edge uv are also spanning trees of $G-uv$, and their number is $\tau(G-uv)$. If T is a spanning tree that does use uv, then we can *contract* the edge uv, identifying u and v so that they become a single vertex. Let $T \cdot uv$ denote the reduced tree. It is a spanning tree of $G \cdot\cdot uv$. Every spanning tree of $G \cdot uv$ is a contraction $T \cdot uv$ of some spanning tree T of G; for just expand the contracted edge back into uv to get T. This gives:
LEMMA 5.5 Let G be any graph. Then $\tau(G)=\tau(G-uv)+\tau(G-uv)$.
This applies equally well to simple graphs and multigraphs. It is illustrated in Figures 5.18 and 5.19.

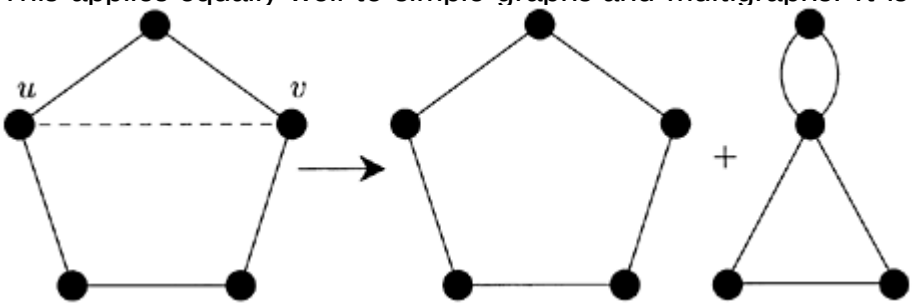


FIGURE 5.18
Deletion and contraction of edge uv

Notice that even when G is a simple graph, $G \cdot uv$ will often be a multigraph, or have loops. Now loops can be discarded, since they cannot be part of any spanning tree. However multiple edges must be kept, since they correspond to different spanning trees of G.
In the example above, the 5-cycle obviously has five spanning trees. The other graph is then decomposed, giving "trees" which contain some multiple edges (i.e., replacing the multiple edges with single edges gives a tree). The two such "trees" shown clearly have two and four spanning trees each, respectively. Therefore the original graph has 5+2+4=11 spanning trees.

Page 110

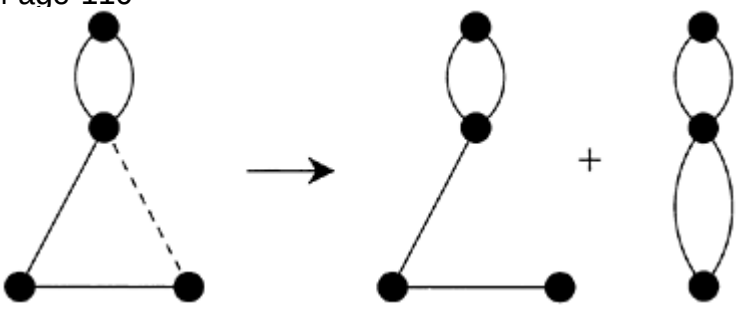


FIGURE 5.19
Finding the number of spanning trees

In general, if G has an edge of multiplicity k joining vertices u and v, then deleting any one of the equivalent k edges will give the same number of spanning trees. Contracting any one of them forces the rest to collapse

into loops, which are then discarded. This gives the following lemma:
LEMMA 5.6 Let edge uu have multiplicity k in G. Replace the multiple edges having endpoints u and u by a single edge uu to get a graph Guu. Then

$$\tau(G)=\tau(Guu-uu)+k\tau(Guu\cdot uu)$$

This gives a recursive technique for finding the number of spanning trees of a connected graph *G*. *G* is stored as a weighted simple graph, for which the weight of an edge represents its multiplicity.

Algorithm 5.7.1: SPTREES*(G)*
find an edge *uu* on a cycle
if there is no such edge
then { **comment:** G is a tree
return (product of edge weights)
else return (SPTREES(*G*−*uu)*+WT*(uu)**SPTREES(*G*·*uu*))

This can be expanded to make a list of all spanning trees of *G*. However, if only the *number* of spanning trees is needed, there is a much more efficient method.

5.8 The matrix-tree theorem

The number of spanning trees of *G* can be computed as the determinant of a matrix. Let *A(G)* denote the adjacency matrix of *G*. The *degree matrix* of *G* is *D(G),* all of whose entries are 0, except for the diagonal, which satisfies *[D]uu*=DEG*(u),* for vertex *u*. The *Kirchhoff matrix* of *G* is *K(G)*=*D*−*A*. This matrix is sometimes also called the *Laplacian matrix* of *G*. The number of spanning trees is found from the Kirchhoff matrix. First, notice that the row and column sums of *K* are all 0, since the row and column sums of *A* are the degrees of *G*. Therefore, det*(K)*=0. Consider the expansion of det*(K)* into cofactors along row *u*. Write

$$\det(K)=\sum_{v=1}^{n}(-1)^{u+v}k_{uv}\ \det(K_{uv}).$$

Here *kuu* denotes the entry in row *u* and column *u* of *K,* and *Kuu* denotes the submatrix formed by crossing out row *u* and column *u*. The *cofactor* of *kuu* is (−1)*u*+*u* det*(Kuu)*. There are *n* vertices.
THEOREM 5.7 **(Matrix-Tree Theorem)** *Let K be the Kirchhoff matrix of G. Then τ(G)*=(−1)u+u det*(Kuu), for any row index u and any column index u*.
PROOF Notice that the theorem says that all cofactors of *K* have the same value, namely, the number of spanning trees of *G*. The proof is by induction on the number of vertices and edges of *G*. Suppose first that *G* is a disconnected graph; let one of the components be *H*. Order the vertices so that vertices of *H* come before the rest of *G*. Then *K(G)* is a block matrix, as shown in Figure 5.20.

If row *u* and column *u,* where *u,* $v \in V(H)$, are crossed off, then the row and column sums of *G*−*H* will be all 0, so that the cofactor corresponding to *Kuu* will be zero. Similarly, if any other row and column are crossed off, the remaining cofactor will be zero. Therefore, if *G* is disconnected, the theorem is true, since *τ(G)*=0.
Suppose now that *G* is a tree. Choose a leaf *u* and let $v \longrightarrow u$. Without loss of generality, we can order the vertices so that *u* is the last vertex. Write *K*·*uu*=*K(G*·*uu)*. The two Kirchhoff matrices are shown in Figure 5.21, where *a*=*kuu*.

If *n*=2, there is only one tree, with Kirchhoff matrix $\begin{bmatrix} 1 & -1 \\ -1 & 1 \end{bmatrix}$. All the cofactors have value ±1, as desired. If *n*>2, we assume that the matrix-tree theorem holds for all trees with at most *n*−1 vertices, and form *K*−*uu*. Now det(*K*·*uu*)=0, since it is a Kirchhoff matrix. The submatrix *Kuu* differs from *K*·*uu* only in the single term *a* instead of *a*−1 in entry *uu*. When we expand det*(Kuu)* along row

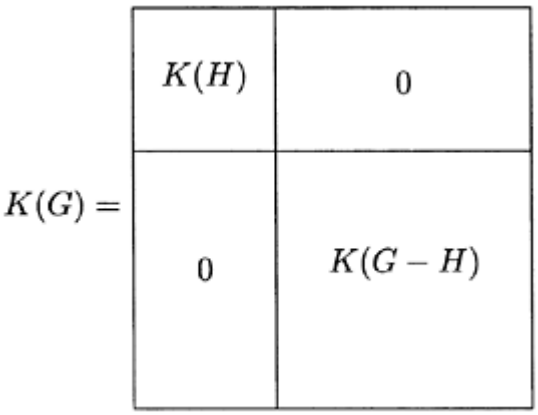


FIGURE 5.20
Kirchhoff matrix of a disconnected graph

u, all the terms are identical to expanding det($K \cdot uu$) along row *u*, except for this one. Therefore det(Kuv)−det($K \cdot uu$) equals the *uu*-cofactor in $K \cdot uu$. By the induction hypothesis, this is $\tau(G \cdot uu)=1$. Therefore det$(Kuv)=1$. Striking off row and column *u* from *K*, and expanding along row *v*, shows that det(Kuu) again equals the uu-cofactor in $K \cdot uu$, which is 1. Therefore, the uu-cofactor in *K* equals $\tau(G)$. Consider next the cofactor det(Kxy), where neither *x* nor *y* equals *u* or *v*. Strike off row *x* and column *y* of *K*. In order to evaluate $(-1)^{x+y}$ det(Kxy), first add row *v* to row *u*, and then expand the determinant along column *v*. The value clearly equals the *xy*-cofactor of $K \cdot uv$, which is $\tau(G)=1$. If $x=u$ but $y \neq u$ or *v*, a similar argument shows that the cofactor equals 1. If $x=u$ but $y \neq u$, then expand along column *v* to evaluate $(-1)^{x+y}$ det(Kxy). The result is $(-1)^{x+y}(-1)^{u+x-1}$ times thedeterminantof $(K \cdot uv)uy$. This reduces to $(-1)^{u+y}$ det$((K \cdot uv)uy)=\tau(G)$. Thus, in all cases, the cofactors equal $\tau(G)=1$. By induction, the matrix-tree theorem is true for all trees.

If *G* is a multigraph with $n=2$ vertices, then it consists of *m* parallel edges, for some $m \geq 1$, so that $\tau(G)=m$. It is easy to see that the theorem is true in this case, as the Kirchhoff matrix is $\begin{bmatrix} m & -m \\ -m & m \end{bmatrix}$. Suppose now that it holds for all multigraphs with fewer than *n* vertices, where $n>2$, and for all multigraphs on *n* vertices with less than ε edges, where $\varepsilon > n-1$, since we know that it holds for trees. Choose an edge *uv* of *G*, and write $\tau(G)=\tau(G-uv)+\tau(G \cdot uv)$. The corresponding Kirchhoff matrices are illustrated in Figure 5.22, where we write $K-uv$ for $K(G-uv)$.

The diagram is drawn as though the edge *uv* had multiplicity 1, but the proof is general, and holds for any multiplicity $m \geq 1$. Let *a* denote the entry *kuu* and *b* the entry *kuv*.

Consider the *uv*-cofactor of *K*. It is nearly identical to the *uv*-cofactor of

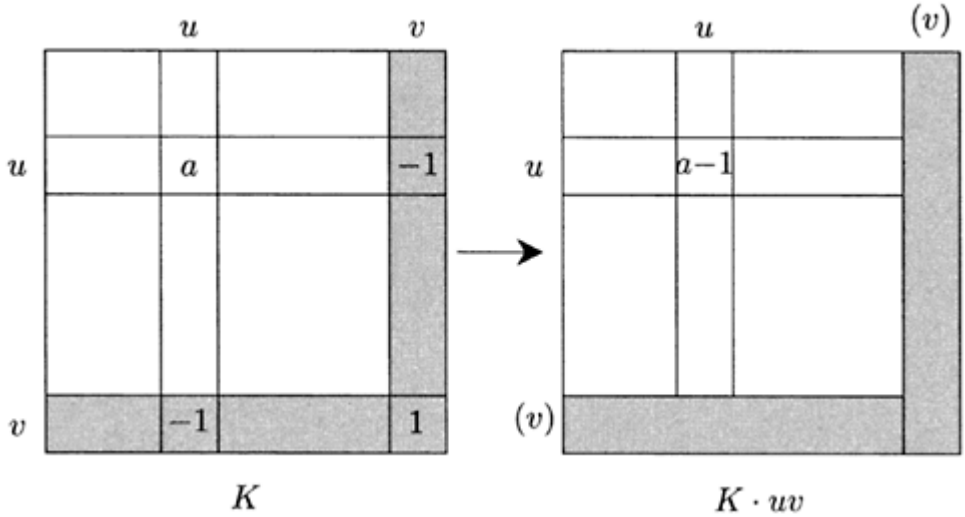


FIGURE 5.21
Kirchhoff matrices for a tree

$K-uv$, differing only in the *uu*-entry, which is *a* in *K* but $a-1$ in $K-uv$. Expanding det(Kuv) along row *u* shows that det(Kuv)−det$((K-uv)uv)$ equals the *uu*-cofactor of $K-uv$, with row and column *v* removed. Comparison with Figure 5.23 shows that this is identical to the uu-cofactor of $K \cdot uv$. Therefore

det*(Kuu)*−det((*K*−*uu*)*uu*)=det((*K*·*uu*)*uu*). By the induction hypothesis, this gives det*(Kuu)*=*τ*(*G*·*uu*)+*τ*(*G*−*uu*)=*τ*(*G*), as desired.
Consider now *Kuu*, formed by striking off row *u* and column *u* of *K*. This matrix is almost identical to that formed by striking off row *u* and column *u* from *K*−*uu*. The only difference is in the *uu*-entry. Expanding along row *u* shows that the difference of the determinants, det*(Kuu)*−det((*K*− *uu*)*uu*) is (−1)*u*+*u*−1(−1) det((*K*·*uu*)*uu*). Therefore (−1)*u*+*u* det*(Kuu)*= (−1)*u*+*u* det((*K*−*uu*)*uu*)+det((*K*·*uu*)*uu*)=*τ*(*K*−*uu* Thus, the *uu*-cofactor and the *uu*-cofactor both equal *τ*(*G*).
Finally, we show that the remaining entries in row *u* also have cofactors equal to *τ*(*G*). Consider any entry *kuw*, where *w*≠*u*,*u*. Strike off row *u* and column *w* of *K* and of *K*−*uu*. In order to evaluate det*(Kuw)* and det((*K*−*uu*)*uw*), first add the remaining part of column *u* to column *u* in both matrices. This is illustrated in Figure 5.24.
Kuw and (*K*−*uu*)*uw* are now identical, except for the *uu*-entry. Expand det*(Kuw)* along row *u*. All terms are equal to the corresponding terms in the expansion of det((*K*−*uu*)*uw*) along row *u*, except for the last term. The difference is (−1)*u*+*u*−1(−1) det((*K*·*uu*)*uw*). Therefore (−1)*u*+*w* det*(Kuw)*= (−1)*u*+*w* det((*K*−*uu*)*uw*)+(−1)*u*+*w* det((*K*·*uu*)*uw*). As before, we get (−1)*u*+*w* det*(Kuw)*=*τ*(*G*). Thus, all the cofactors of *K* from row *u* have equal

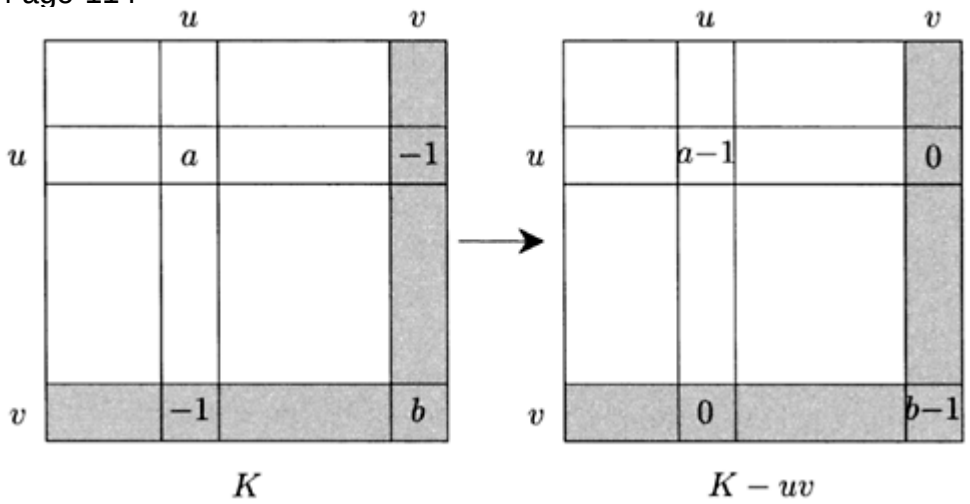


FIGURE 5.22
Kirchhoff matrices *K* **and** *K*−*uu*

value, namely, *τ*(*G*). Since *u* could be any vertex, all the cofactors of *K* have this value. This completes the proof of the matrix-tree theorem.
A nice illustration of the use of the matrix-tree theorem is to compute *τ*(*Kn*). The Kirchhoff matrix is

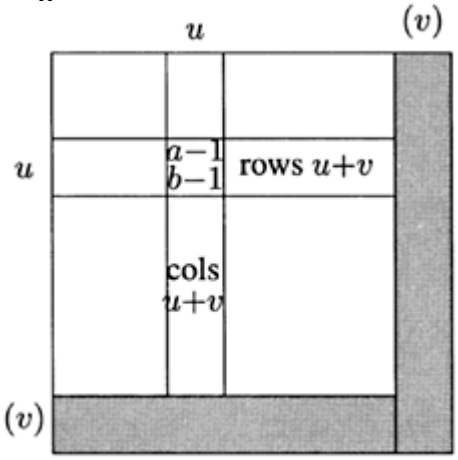
$$K(K_n) = \begin{bmatrix} n-1 & -1 & -1 & \cdots & -1 \\ -1 & n-1 & -1 & \cdots & -1 \\ -1 & -1 & n-1 & & \vdots \\ \vdots & \vdots & & \ddots & -1 \\ -1 & -1 & \cdots & -1 & n-1 \end{bmatrix}$$

Strike off the last row and column. In order to evaluate the determinant, add all the rows to the first row, to get

$$\begin{bmatrix} 1 & 1 & \cdots & \\ -1 & n-1 & & \\ \vdots & & \ddots & \\ & & & n-1 \end{bmatrix}_{(n-1)\times(n-1)}$$

Now add the first row to each row in turn, in order to get n's on the diagonal and 0's off the diagonal. Thus, the determinant is *nn*−2, as expected.
The Kirchhoff matrix was first used to solve electrical circuits. Consider a simple electrical network consisting of resistors and a battery producing voltage *V*. Let the nodes in the network be *u*1, *u*2,..., *un*, and suppose that the resistance

Page 115



$K \cdot uv$

FIGURE 5.23
Kirchhoff matrix *K·uu*

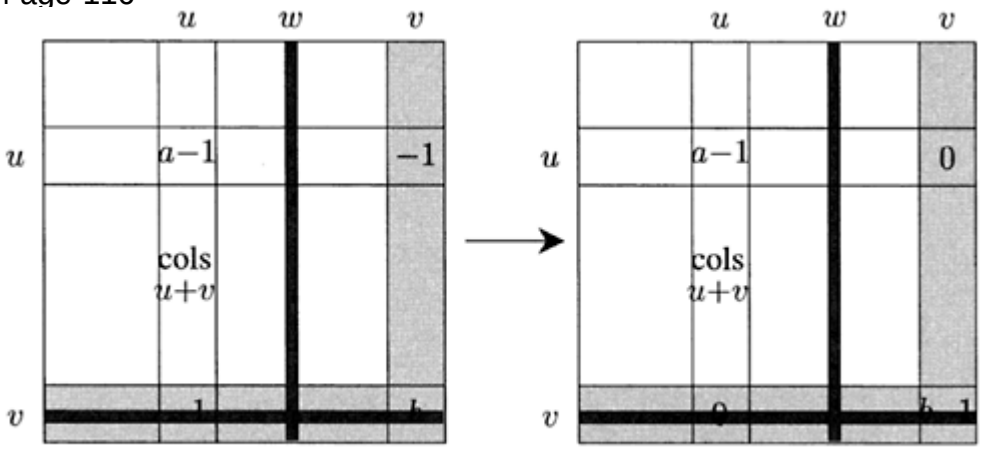
connecting *ui* to *uj* is *rij*. The battery causes current to flow in the network, and so sets up a voltage *Vi* at each node *ui*. The current from *ui* to *uj* is (*Vi*−*Vj*)/*rij*. This is illustrated below.

The law of conservation of charge say s that the total current flowing out of node *ui* must equal the total current flowing in, that is, not counting the current flowing through the battery,

$$\sum_{u_j \to u_i} \frac{(V_i - V_j)}{r_{ij}} = 0,$$

for all nodes *ui*. The battery maintains a constant voltage difference of *V* across nodes *u*1 and *un,* say. Let *I* denote the current through the battery. Then the *u*1-equation must be modified by setting the right-hand side to I instead of 0; and the *un*-equation requires the right-hand side to be—*I*. This gives a system of linear equations in the variables *Vi* and *I*. If we consider the network as a multigraph in which *ui* is joined to *uj* by 1/*rij* parallel edges, then the diagonal entries of the matrix corresponding to the equations are the degrees of the nodes. The offdiagonal entries form the negated adjacency matrix of the network. Thus, this is the Kirchhoff matrix of the network. Since the Kirchhoff matrix has determinant zero, there is no unique solution to the system. However, it is voltage *differences* that are important, and we know that the battery maintains a constant voltage difference of *V*. Therefore, we can arbitrarily set *Vn*=0 and *V*1=*V,* so that we can cross off the *nth* column from the matrix. The rows are linearly dependent, so that we can also discard any row. The system then has a unique solution, since striking off a row and column from the Kirchhoff matrix leaves the spanning tree matrix. Notice that once the current in each edge is known, each spanning tree

Page 116



K (with column v added to u) $K - uv$

FIGURE 5.24

of the network will determine the voltage distribution uniquely, since a spanning tree has a unique path connecting any two vertices.

Exercises

5.8.1 Find $\tau(K3,3)$ using the recursive method.

5.8.2 Find $\tau(Kn-uu)$, where uu is any edge of Kn, using the matrix-tree theorem.

5.8.3 Find $\tau(Cn)$, where Cn is the cycle of length n, using the matrix-tree theorem.

5.8.4 What is the complexity of finding the determinant of an $n \times n$ matrix, using Gaussian elimination? Accurately estimate an upper bound on the number of steps needed.

5.8.5 Let G be a graph with n vertices. Replace each edge of G with m multiple edges to get a graph Gm. Prove that $\tau(Gm) = mn-1\ \tau(G)$.

5.8.6 Program the recursive algorithm to find the number of spanning trees. Use a breadth-first search to find an edge on a cycle.

5.8.7 Solve the electrical circuit of Figure 5.25, taking all resistances equal to one. Solve for the voltage Vi at each node, the current in each edge, and the total current I, in terms of the battery voltage V.

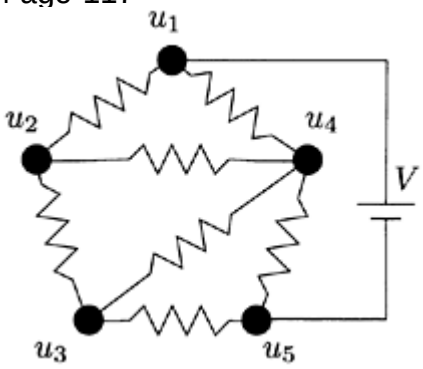


FIGURE 5.25
A simple network of resistors

5.9 Notes

Read's tree encoding algorithm is from READ [98]. Prüfer sequences date back to 1918—PRÜFER [95]. They are described in several books, including BONDY and MURTY [19]. The matrix-tree theorem is one of the most fundamental theorems in graph theory.

This page intentionally left blank.

6
Connectivity

6.1 Introduction

Trees are the smallest connected graphs. For deleting any edge will disconnect a tree. The following figure shows three graphs in order of increasing connectivity.

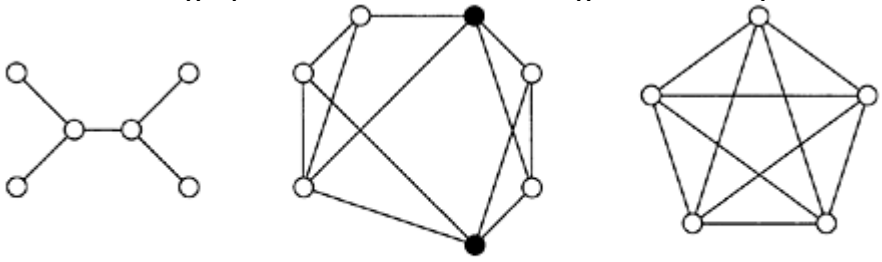


FIGURE 6.1
Three graphs with increasing connectivity

The second graph can be disconnected by deleting the two shaded vertices, but three edges must be deleted in order to disconnect it. The third graph is complete and cannot be disconnected by deleting any number of vertices. However, the deletion of four edges will do so. Thus, connectivity is measured by what must be deleted from a graph G in order to disconnect it. Because one can delete vertices or edges, there will be two measures of connectivity.

The *vertex-connectivity* of G is $\kappa(G)$, the minimum number of vertices whose deletion disconnects G. If G cannot be disconnected by deleting vertices, then $\kappa(G)=|G|-1$. A disconnected graph requires the deletion of 0 vertices, so it has $\kappa=0$. The complete graph has $\kappa(K_n)=n-1$. Hence, $\kappa(K1)=0$,

but all other connected graphs have $\kappa\geq 1$. Any set of vertices whose deletion disconnects G is called a *separating set* or *vertex cut* of G.

The *edge-connectivity* of G is $\kappa'(G)$, the minimum number of edges whose deletion disconnects G. If G has no edges, then $\kappa'(G)=0$. A disconnected graph does not need any edges to be deleted, and so it has $\kappa'=0$. $K1$ also has $\kappa'=0$ because it has no edges, but all other connected graphs have $\kappa'\geq 1$.

The edge-connectivity is always at most $\delta(G)$, since deleting the δ edges incident on a vertex of minimum degree will disconnect G. The following inequality always holds.

THEOREM 6.1 $\kappa\leq\kappa'\leq\delta$.

PROOF We know that $\kappa'\leq\delta$. We prove that $\kappa\leq\kappa'$ by induction on κ'. If $\kappa'=0$, then either G has no edges, or else it is disconnected. In either case, $\kappa=0$. Suppose that it is true whenever $\kappa'\leq m$, and consider $\kappa'=m+1$. If $\kappa'=|G|-1$, then $\delta=\kappa'$ and thus $\kappa\leq\kappa'$; so suppose that $\kappa'<|G|-1$. Let $[S,\overline{S}]$ be an edge-cut containing $m+1$ edges. Pick any edge $uv\in[S,\overline{S}]$ and form $H=G-uu$. Then $[S,\overline{S}]-uv$ is an edge-cut of H containing m edges, so $\kappa'(H)\leq m$. By the induction hypothesis, $\kappa(H)\leq m$. Let $U\subseteq V(H)$ be a minimum separating set of H. Then $|U|\leq m$, and $H-U$ consists of two or more components. We now want to put the edge uu back. Where does it go?

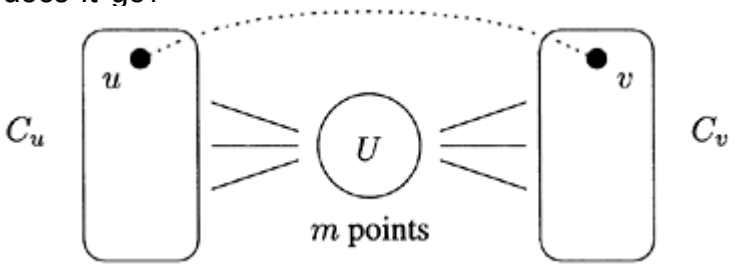


FIGURE 6.2
A minimum separating set of H

If $H-U$ had three or more components, then U would also be a separating set of G, in which case $\kappa(G)\leq|U|=m$. If $H-U$ has exactly two components, Cu and Cu, containing u and u, respectively, then U will not be a separating set of G, for the edge uu will keep it connected. However, $\kappa'(G)<|G|-1$, so that $m=\kappa'-1<|G|-2$. Therefore, one of Cu and Cu contains two or more vertices, say Cu does. Then $U'=U\cup\{u\}$ is a separating set of G with $m+1$ vertices, so that $\kappa(G)\leq\kappa'(G)$. By induction, the theorem is true for all values of κ'.

Except for this inequality, the parameters κ, κ', and δ are free to vary considerably, as shown by CHARTRAND and HARARY [23]. For example, the graph of Figure 6.3 has $\kappa=2$, $\kappa'=3$, and $\delta=4$.

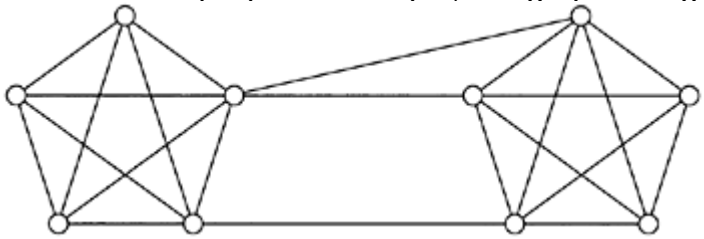


FIGURE 6.3
A graph with $\kappa=2$, $\kappa'=3$, **and** $\delta=4$

Given any three non-negative integers a, b, and c satisfying $a\leq b\leq c$, we can easily make a graph with $\kappa=a$, κ

′=*b*, and *δ*=*c*, as illustrated in Figure 6.3. Take two complete graphs *G′* and *G″*, isomorphic to *Kc*+1. They have minimum degree *δ*=*c*. Choose any a vertices $U' \subseteq V(G')$, and a corresponding vertices $U'' \subseteq V(G'')$. Join them up in pairs, using a edges. Then *U′* is a separating set of the graph, containing *a* vertices. Now add *b*−*a* edges connecting *G′* to *G″*, such that every edge added has one endpoint in *U′*. Clearly the graph constructed has *κ*=*a*, *κ′*=*b*, and *δ*=*c*.

Exercises

6.1.1 Let *G* be connected and let $uv \in E(G)$. Prove that *uu* is in every spanning tree of *G* if and only if *uu* is a cut-edge of *G*.

6.1.2 Show that a connected graph with exactly two vertices that are not cut-vertices is a tree. *Hint:* Consider a spanning tree of *G*.

6.1.3 Prove that if *G* is a *k*-regular bipartite graph with *k*>1 then *G* has no cut-edge.

6.1.4 Prove that if *G* is connected, with all even degrees, then $\omega(G-v) \leq \frac{1}{2}\text{DEG}(v)$, for any $v \in V(G)$, where *w(G)* is the number of connected components of *G*.

6.1.5 Let *G* be a 3-regular graph.
(a) If *κ*=1, show that *κ′*=1.
(b) If *κ*=2, show that *κ′*=2.
Conclude that *κ*=*κ′* for 3-regular graphs.

6.1.6 Let *G* be a 4-regular graph with *κ*=1. Prove that *κ′*=2.

6.1.7 Let (*d*1, *d*2,..., *dn*), where *d*1≤*d*2≤...≤*dn*, be the degree sequence of a graph *G*. Prove that if *dj*≥*j*, for *j*=1,2,...,*n*−1−*dn*, then *G* is connected.

6.1.8 Give another proof that *κ*≤*κ′*, as follows. Let $[S, \overline{S}]$ be a minimum edge-cut of *G*, containing *κ′* edges. Construct a set $U \subseteq S$ consisting of all vertices $u \in S$, such

that there is an edge $uv \in [S, \overline{S}]$. Then |*U*|≤*κ′*. If *U*≠*S*, then *U* is a separating set of *G* with ≤*κ′* vertices. Therefore *κ*≤*κ′*. Show how to complete the proof when *U*=*S*.

6.2 Blocks

Any graph *G* with *κ*≥1 is connected. Consequently *G* is said to be 1-connected. Similarly, if *κ*≥2, then at least two vertices must be deleted in order to disconnect *G*, so *G* is said to be 2-connected. It is usually easier to determine a lower bound, such as *κ*≥2 or *κ*≥3, than to compute the exact value of *κ*. In general, *G* is said to be *k-connected* if *κ*≥*k*, for some integer *k*.

If *G* is a disconnected graph, then its structure is determined by its components, that is, its maximal connected subgraphs. A component which is an isolated vertex will have *κ*=0, but all other components will be 1-connected.

If a connected graph *G* has a cut-vertex *u*, then it is said to be *separable*, since deleting *u* separates *G* into two or more components. A separable graph has *κ*=1, but it may have subgraphs which are 2-connected, just as a disconnected graph has connected subgraphs. We can then find the maximal non-separable subgraphs of *G*, just as we found the components of a disconnected graph. This is illustrated in Figure 6.4.

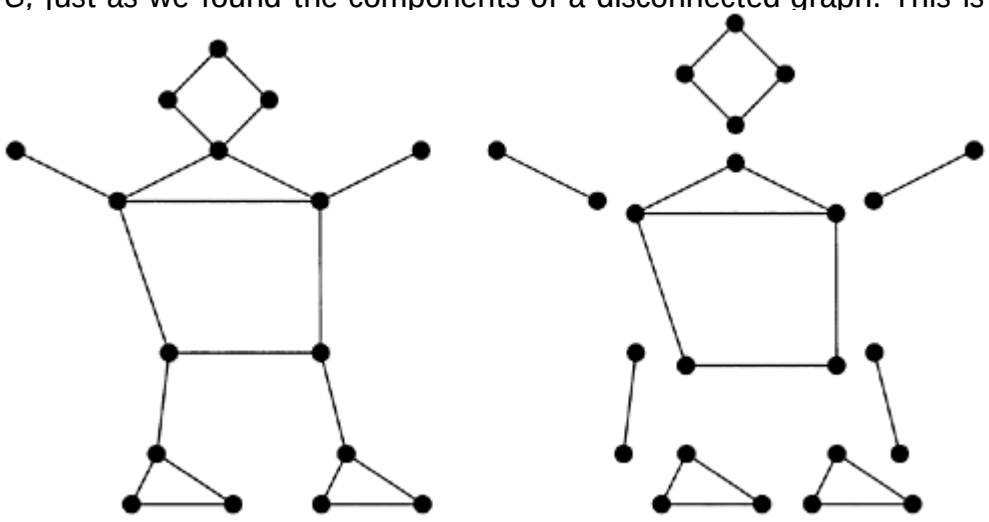


FIGURE 6.4

A graph (a) and its blocks (b)
The maximal non-separable subgraphs of *G* are called the *blocks* of *G*. The

graph illustrated above has eight blocks, held together by cut-vertices. Every separable graph will have two or more blocks. Any 2-connected graph is non-separable. However, *K*2, a graph which consists of a single edge, is also non-separable, since it has no cut-vertex. Therefore every edge of *G* is a non-separable subgraph, and so will be contained in some maximal non-separable subgraph. Can an edge be contained in two distinct blocks? We first describe some properties of 2-connected graphs.
Notice that cycles are the smallest 2-connected graphs, since a connected graph with no cycle is a tree, which is not 2-connected. Any two vertices *u* and *u* on a cycle *C* divide *C* into two distinct paths with only the endpoints *u* and *u* in common. Paths which have only their endpoints in common are said to be *internally disjoint;* see Figure 6.5.

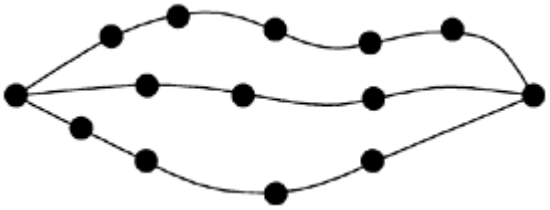


FIGURE 6.5
Three internally disjoint paths
THEOREM 6.2 A graph G with three or more vertices is 2-connected if and only if every pair of vertices is connected by at least two internally disjoint paths.
PROOF ⇐: Suppose that every pair of vertices of *G* is connected by at least two internally disjoint paths. If a vertex *w* is deleted, then every remaining pair of vertices is still connected by at least one path, so that *w* is not a cut-vertex. Therefore $\kappa > 2$.
⇒: Let *G* be 2-connected, and let $v \in V(G)$. We prove by induction on DIST*(u,u)* that *u* and *u* are connected by two internally disjoint paths. If DIST*(u,u)*=1, then *G*−*uu* is still connected, since $\kappa' \geq \kappa \geq 2$. Therefore *G*−*uu* contains a *uu*-path *P,* so that *G* has two *uu*-paths, *P* and *uu*. Suppose that the result holds when DIST*(u,u)*≤*m* and consider DIST*(u,u)*=*m*+1. Let *P* be a *uu*-path of length *m*+1 and let *w* be the last vertex before *u* on this path. Then DIST*(u,w)*=*m,* since *P* is a shortest path. By the induction hypothesis, *G* contains internally disjoint *uw*-paths *Pw* and *Qw*.
Since *G* is 2-connected, *G*−*w* is still connected, and so has a *uu*-path *R*. *R* has the endpoint *u* in common with both *Pw* and *Qw*. Let *x* be the last vertex common to *R* and either of *Pw* or *Qw,* say $x \in P_w$. Then *Pw[u,x]R[x,u]* and

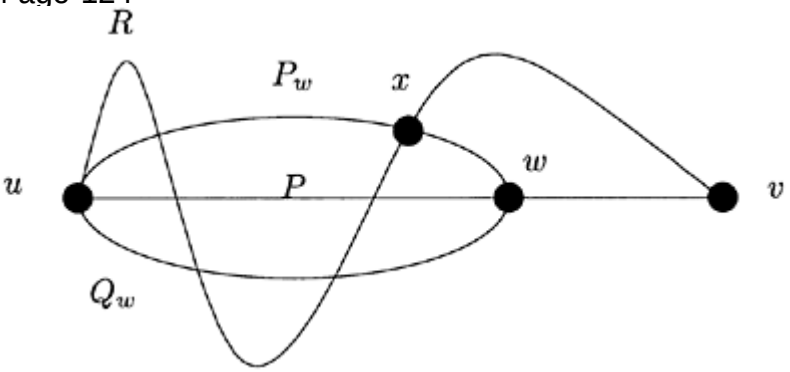


FIGURE 6.6
Internally disjoint paths *Pw* **and** *Qu*
Qwwu are two internally disjoint *uu*-paths. By induction, the result holds for all pairs *u, u* of vertices.
So in a 2-connected graph, every pair *u, u,* of vertices are connected by at least two internally disjoint paths *P* and *Q.* Since *P* and *Q* together form a cycle, we know that *every pair of vertices lies on a cycle*. Another consequence of this theorem is that every pair of *edges* also lies on a cycle.
COROLLARY 6.3 A graph G with three or more vertices is 2-connected if and only if every pair of edges lies on a cycle.

PROOF Let *G* be 2-connected and pick edges *uu,* $xy \in E(G)$. Subdivide *uu* with a new vertex *w,* and *xy* with a new vertex *z* to get a graph *G′*. Now *G* has no cut-vertex, so neither does *G′*. By the previous theorem, *w* and *z* lie on a cycle in *G′,* so that *uu* and *xy* lie on a cycle in *G*.
⇐: Suppose now that every pair of edges lies on a cycle. Then every vertex has degree at least two, since no cycle could pass through an edge incident on a vertex of degree one. Choose any two vertices *u* and *x*. Choose any vertex $v \longrightarrow u$ and a vertex $y \longrightarrow x$, such that *y≠u*. Then the edges *uu* and *xy* must lie on a cycle *C*. Clearly *C* contains *u* and *x,* so that every pair *u, x,* of vertices lies on a cycle. It follows that *G* is 2-connected.
LEMMA 6.4 Each edge uu of G is contained in a unique block.
PROOF Let *uu* be an edge in a graph *G,* and let *B* be a maximal 2-connected subgraph of *G* containing *uu*. If *B′* is another maximal 2-connected subgraph

Page 125

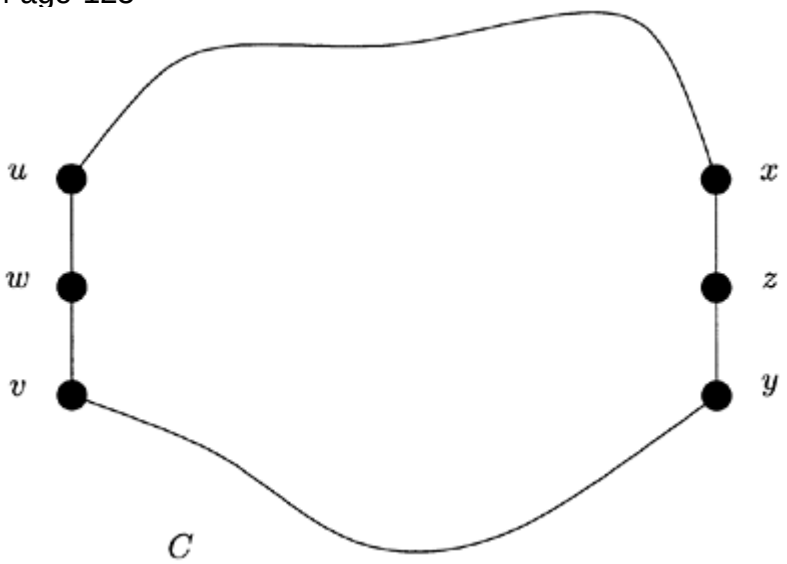


FIGURE 6.7
Two edges on a cycle

containing *uv*, where *B≠B′,* then choose any edge $xy \in B' - B$. *B′* contains a cycle *C* containing both *uu* and *xy,* since *B′* is 2-connected. The subgraph $B \cup C$ is 2-connected, and is larger than *B,* a contradiction. Therefore, each edge *uu* is contained in exactly one block of *G*.

6.3 Finding the blocks of a graph

The first algorithm to find the blocks of a graph was discovered by READ [99]. It uses the fundamental cycles with respect to a spanning tree *T*. Because each edge of *G* is contained in a unique block *Buu,* the algorithm begins by initializing *Buu* to contain only *uu* and uses the merge-find data structure to construct the full blocks *Buu*. For each edge $uv \notin T$, the fundamental cycle *Cuu* is found. Because *Cuu* is 2-connected, all its edges are in one block. So upon finding *Cuu,* we merge all the blocks *Bxy,* where $xy \in C_{uv}$, into one. Any spanning tree *T* can be used. If we choose a breadth-first tree, we have Algorithm 6.3.1.

Page 126

Algorithm 6.3.1: BLOCKS*(G)*
comment: *G* is a connected graph
for each $uv \in E(G)$
do initialize *Buu* to contain *uu*
pick any vertex $x \in V(G)$
place *x* on *ScanQ*
repeat
select *u* from head of *ScanQ*
for each $v \longrightarrow u$
do if $v \notin ScanQ$

then { **comment:** uv forms part of the spanning tree T
add edge uv to T
add v to *ScanQ*

else { **comment:** uv creates a fundamental cycle
construct C_{uv}
for each edge $xy \in C_{uv}$
do $B_{uv} \leftarrow B_{uv} \cup B_{xy}$

advance *ScanQ*

until all vertices on *ScanQ* are processed

comment: each *Buu* now consists of the unique block containing *uu*

LEMMA 6.5 At the beginning of each iteration of the repeat loop, each Buuis either a single edge, or else is 2-connected.

PROOF The proof is by induction on the number of iterations of the repeat loop. At the beginning of the first iteration it is certainly true. Suppose that it is true at the beginning of the *kth* iteration. If the edge *uu* chosen forms part of the spanning tree *T,* it will also be true for the (*k*+1)*st* iteration, so suppose that *uu* creates a fundamental cycle *Cuu*. Each *Bxy* for which $xy \in C_{uv}$ is either a single edge, or else 2-connected. The new *Buu* is formed by merging all the *Bxy* into one, say $B_{uv} = B_{x_1y_1} \cup B_{x_2y_2} \cup \cdots \cup B_{x_my_m}$. Pick any two edges *ab,*$cd \in B_{uv}$, say $ab \in B_{x_iy_i}$ and $cd \in B_{x_jy_j}$. Weshow that *Buu* contains a cycle containing both ab and *cd*. If *ab,* $cd \in C_{uv}$, it is certainly true. Otherwise, notice that each $B_{x_iy_i}$ contains some edge of *Cuu*. Without loss of generality, we can suppose that the edges $x_ly_l \in C_{uv}$, for *l*=1, 2,..., *m*. Since $B_{x_iy_i}$ is 2-connected, it contains a cycle *Ci* containing both *xiyi* and *ab,* and $B_{x_jy_j}$ contains a cycle *Cj* containing both *xiyj* and *cd*. This is illustrated in Figure 6.8. Then $C_{uv} \oplus C_i \oplus C_j$ is a cycle contained in *Buu* and containing *ab* and *cd*. By Corollary 6.3, the new

Buu is 2-connected. Therefore the result is true at the beginning of the (*k*+1)*st* iteration. By induction it is true for all iterations.

COROLLARY 6.6 Upon completion of Algorithm 6.3.1, each Buucontains the unique block containing uu.

PROOF By the previous lemma, each *Buu* will either be a single edge, or else 2-connected. If *Buu* is not the unique block *B* containing *uu,* then pick some edge $xy \in B - B_{uv}$. *B* contains a cycle *C* containing both *uu* and *xy*. By Theorem 4.3, *C* can be written in terms of fundamental cycles with respect to the spanning tree *T* constructed by Algorithm 6.3.1, $C = C_{u_1v_1} \oplus C_{u_2v_2} \oplus \cdots \oplus C_{u_mv_m}$. But each of the fundamental cycles $C_{u_iv_i}$ will have been processed by the algorithm, so that all edges of *C* are contained in one *Buu,* a contradiction. Therefore, each *Buu* consists of the unique block containing *uu*.

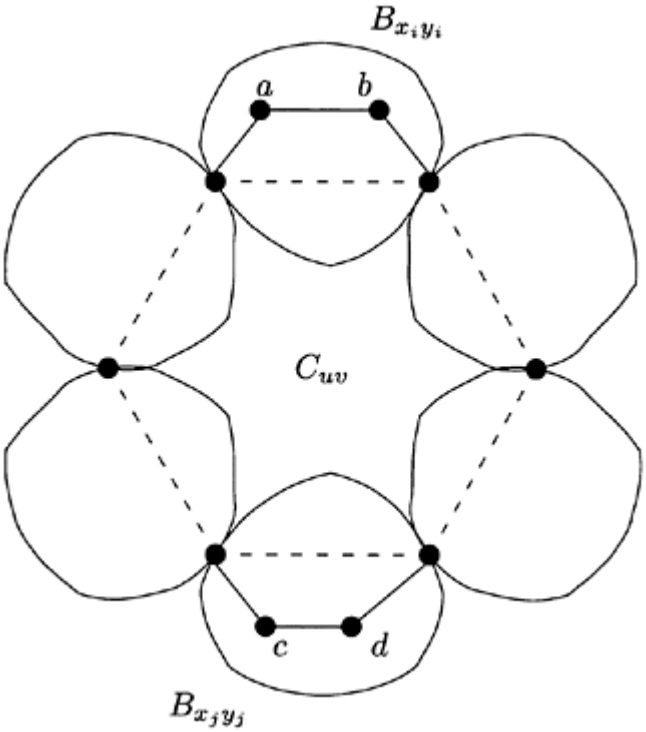


FIGURE 6.8
Merging the *Bxy*

Exercises

6.3.1 Given an edge *uu* which creates a fundamental cycle *Cuu,* describe how to find *Cuu* using the *Parent*[·] array created by the BFS.

6.3.2 Let (*d*1, *d*2,..., *dn*), where $d1 \le d2 \le ... \le dn$, be the degree sequence of a graph *G*. Prove that if $dj \ge j+1$, for $j=1, 2, ..., n-1-dn-1$, then *G* is 2-connected.

6.3.3 Program the BLOCKS() algorithm. One way to store the merge-find sets *Buu* is as an *n* by *n* matrix *BlockRep*[·,·]. Then the two values *x*←*BlockRep[u, u]* and *y*←*BlockRep[u, u]* together define the edge *xy* representing *uu*. Another way is to assign a numbering to the edges, and use a linear array.

6.3.4 Try to estimate the complexity of the algorithm BLOCKS(). It is difficult to obtain a close estimate because it depends on the sum of the lengths of all $\varepsilon - n + 1$ fundamental cycles of *G*, where $n = |G|$.

6.3.5 *The Block-Cut-Vertex Tree.* (See HARARY [59].) Let *G* be a connected separable graph. Let *B* denote the set of blocks of *G* and *C* denote the set of cut-vertices. Each cut-vertex is contained in two or more blocks, and each block contains one or more cut-vertices. We can form a bipartite graph *BC(G)* with vertex-set $\mathcal{B} \cup \mathcal{C}$ by joining each $B \in \mathcal{B}$ to the cut-vertices $v \in \mathcal{C}$ that it contains.

(a) Show that *BC(G)* has no cycles, and consequently is a tree.

(b) In the block-cut-vertex tree *BC(G),* the degree of each $v \in \mathcal{C}$ is the number of blocks of *G* containing *u*. Denote this value by *b(u),* for any vertex $v \in V(G)$. Show that

$$\sum_{v \in V(G)} b(v) - 1 = \sum_{v \in \mathcal{C}} b(v) - 1 = |\mathcal{B}| - 1,$$

so that the number of blocks of *G* is given by

$$|\mathcal{B}| = 1 + \sum_{v \in V(G)} b(v) - 1.$$

(c) Prove that every separable graph has at least two blocks which contain only one cut-vertex each.

6.4 The depth-first search

There is an easier, more efficient way of finding the blocks of a graph than using fundamental cycles. It was discovered by Hopcroft and Tarjan. It uses a *depth-first search (DFS)* to construct a spanning tree. With a depth-first search the fundamental cycles take a very simple form—essentially we find them for free, as they require no extra work. The basic form of the depth-first search follows. It is a recursive procedure, usually organized with several global variables initialized by the calling procedure. The example following uses a global counter *DFCount,* and two arrays *DFNum[u]* and *Parent[u]*. Each vertex *u* is assigned

Page 129
a number *DFNum[u],* being the order in which the DFS visits the vertices of *G*, and a value *Parent[u],* being the vertex u from which DFS*(u)* was called. It is the parent of *u* in the rooted spanning tree constructed. Initially all *DFNum*[·] values are set to 0.

Algorithm 6.4.1: DFS*(u)*

comment: extend a depth-first search from vertex *u*

DFCount←*DFCount*+1
DFNum[u]←*DFCount*
for each $v \longrightarrow u$
do if *DFNum[v]*=0

then {
- **comment:** v is not visited yet
- add edge uv to the spanning tree
- $Parent[v] \leftarrow u$
- DFS(v)

else {comment: *uv* creates a fundamental cycle

The calling procedure can be written as follows:

DFCount←0
for *u*←1 **to** *n* **do** *DFNum[u]*←0
select a staring vertex *u*
DFS*(u)*

Figure 6.9 shows a depth-first search in a graph. The numbering of the vertices shown is that of *DFNum*[·], the order in which the vertices are visited.

Notice that while visiting vertex *u,* DFS*(u)* is called immediately, for each $v \longrightarrow u$ discovered. This means that before returning to node *u,* all vertices that can be reached from *u* on paths that do not contain *u* will be visited; that is, all nodes of *G*−*u* that are reachable from *u* will be visited. We state this important property as a lemma. For any vertex *u,* let *A(u)* denote *u,* plus all ancestors of *u* in the depth-first tree, where an ancestor of *u* is either its parent, or any vertex on the unique spanning tree path from *u* to the root vertex.

LEMMA 6.7 Suppose that DFS *(u) is called while visiting node u. Then* DFS *(u) visits every vertex in V(G)*−*A(u) reachable from u before returning to node u.*

PROOF The statement
if *DFNum[u]*=0 **then**...

Page 130

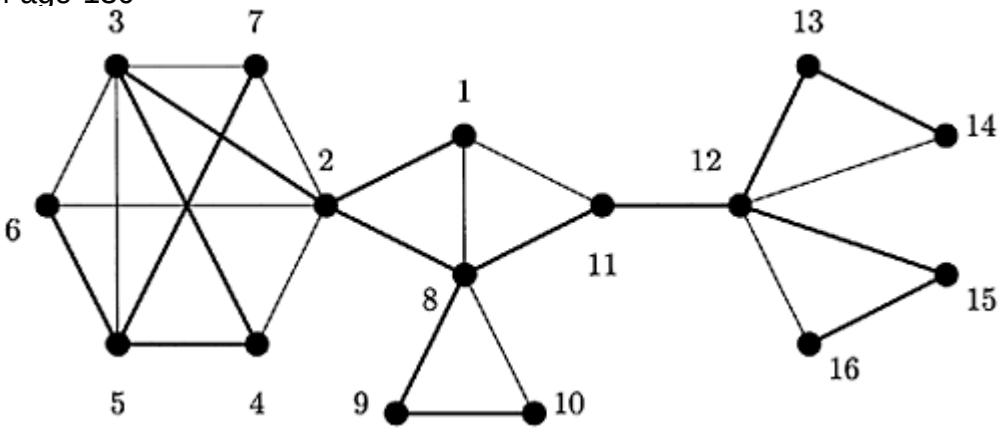


FIGURE 6.9
A depth-first search

ensures that no vertex of *A(u)* will be visited before returning to node *u*. To show that every vertex of *G*−*A(u)* connected to *u* is visited, let *w* be a vertex of *V(G)*−*A(u),* with DIST*(u,w)*=*k*. The proof is by induction on *k*. It is clear that all $w \longrightarrow v$ will be visited before returning to *u,* so that the statement is true when *k*=1. If *k*>1, let *P* be a *uw*-path of length *k,* and let $x \longrightarrow v$ be the first vertex of *P*. Now *x* will certainly be visited before returning to node *u*. When *x* is visited, either *w* will already have been visited, or else some DFS(y) called from node *x* will visit *w* before returning to *u,* since DIST*(x,w)*=*k*−1. Therefore all vertices of *G*−*u* connected to *u* will be visited before returning to *u*.

This makes it possible to detect when *u* is a cut-vertex. It also means that a spanning tree constructed by a depth-first search will tend to have few, but long, branches. The following diagram shows the DF-tree constructed by the DFS above.
Suppose that while visiting node *u*, a vertex $v \longrightarrow u$ with DFNum *[u]*≠0 is encountered. This means that *u* has already been visited, either previously to *u*, or as a descendant of *u*. While visiting *u,* the edge *uu* will have been encountered. Therefore, if *u* was visited previously, either DFS*(u)* was called from node *u,* so that *Parent[u]=u,* or else DFS*(u)* was called from some descendant *w* of *u*. So we can state the following fundamental property of depth-first searches:
LEMMA 6.8 Suppose that while visiting vertex u in a depth-first search, edge uu creating a fundamental cycle is encountered. Then either u is an ancestor of u, or else u is an ancestor of u.

Page 131

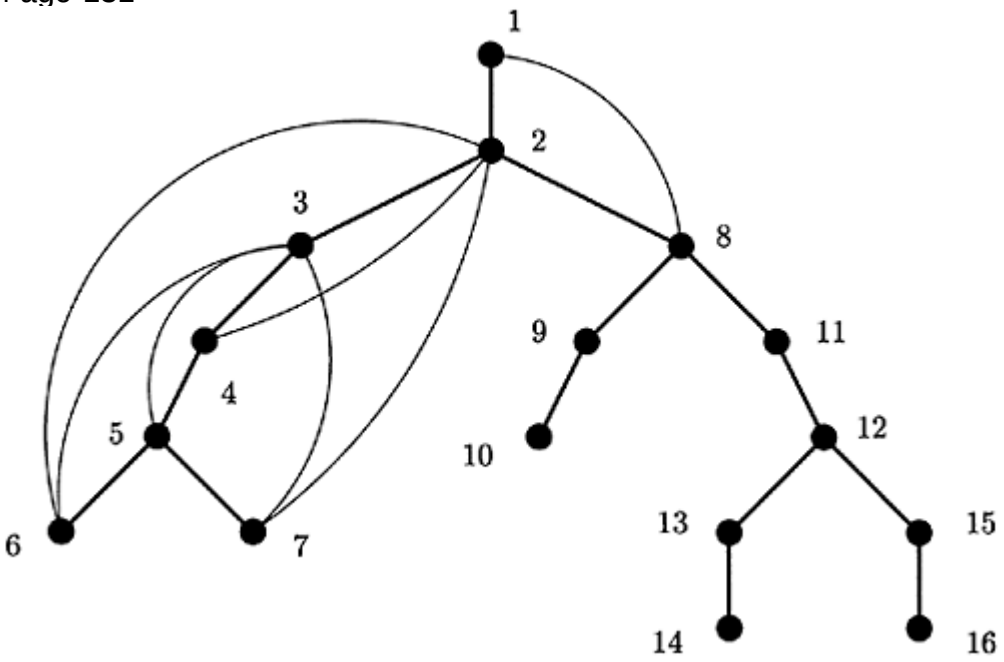


FIGURE 6.10
A depth-first tree with fronds
Edges which create fundamental cycles with respect to a depth-first spanning tree *T* are *called fronds* of *T*. Some of the fronds of the graph of Figure 6.9 are shown in Figure 6.10.
Now suppose that *G* is a separable graph with a cut-vertex *u*. *u* will occur somewhere in *T*, say that DFS*(u)* was called from node *u*, and that node *u* in turn calls DFS*(w)*, where *u* and *w* are in different blocks. Thus, edges *uu* and *uw* do not lie on any cycle of *G*. This is illustrated in Figure 6.11. Consider any descendant *x* of *w* and a frond *xy* discovered while visiting node *x*, where *y* is an ancestor of *x*. Now *y* cannot be an ancestor of *u,* for then the fundamental cycle *Cxy* would contain both edges *uu* and *uw,* which is impossible. Therefore either *y=u,* or else *y* is a descendant of *u*. *So, for every frond xy, where x is a descendant of w, either y=u, or else y is a descendant of u.* We can recognize cut-vertices during a DFS in this way. Ancestors of *u* will have smaller *DFNum*[·] values than *u,* and descendants will have larger values. For each vertex *u,* we need to consider the endpoints *y* of all fronds *xy* such that *x* is a descendant of *u*.
DEFINITION 6.1: Given a depth-first search in a graph *G*. The *low-point* of a vertex *u* is *LowPt[u],* the minimum value of *DFNum[y]*, for all edges *uy* and all fronds *xy*, where *x* is a descendant of *u*.

Page 132

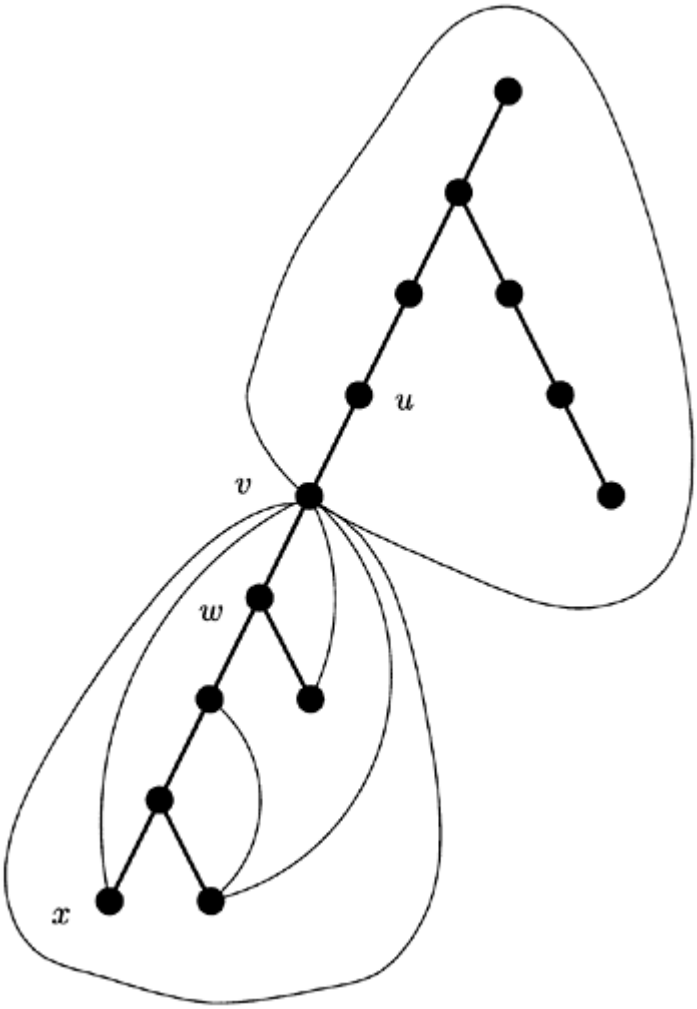


FIGURE 6.11
A cut-vertex *u* in a DF-tree

We can now easily modify DFS*(u)* to compute low-points and find cut-vertices. In addition to the global variables *DFCount, DFNum*[·], and *Parent*[·], the algorithm keeps a stack of edges. Every edge encountered by the algorithm is placed on the stack. When a cut-vertex is discovered, edges on the top of the stack will be the edges of a block of *G*.

The procedure DFSEARCH*(u)* considers all $v \longrightarrow u$. If *u* has been previously visited, there are two possibilities, either *u=Parent[u],* or else *uu* is a frond. When *uu* is a frond, there are also two possibilities, either *u* is an ancestor of *u,* or *u* is an ancestor of *u*. These two cases are shown in Figure 6.12. The algorithm only needs those fronds *uu* for which *u* is an ancestor of *u* in order to compute *LowPt[u]*.

Algorithm 6.4.2: DFBLOCKS*(G)*

comment: {DFS to find the blocks of a connected graph G, on n vertices.

procedure DFSEARCH*(u)*
comment: extend a depth-first search from *u*
DFCount←*DFCount*+1
DFNum[u]←*DFCount*
for each $v \longrightarrow u$ **do**

```
if DFNum[v] = 0
  then  comment: { v is not visited yet,
                 { add uv to the spanning tree
        Parent[v] ← u
        stack uv
        LowPt[v] ← DFNum[u]        "initial value"
        DFSEARCH(v)
        comment: LowPt[v] is now known
        if LowPt[v] = DFNum[u]
          then  { comment: u is a cut-vertex
                { unstack all edges up to, and including, uv
          else  { comment: otherwise LowPt[v] < DFNum[u]
                { if LowPt[v] < LowPt[u]
                  then LowPt[u] ← LowPt[v]
  else  comment: v has already been visited
        if v ≠ Parent[u]  then
          comment: uv is a frond, it creates a fund. cycle
          if DFNum[v] < DFNum[u]
            then  { comment: v is an ancestor of u
                  { stack uv
                  { if DFNum[v] < LowPt[u]
                    then LowPt[u] ← DFNum[v]
```

main
DFCount←0
for *u*←1 **to** *n* **do** *DFNum[u]*←0
select a vertex *u*
LowPt[u]←1
DFSEARCH *(u)*

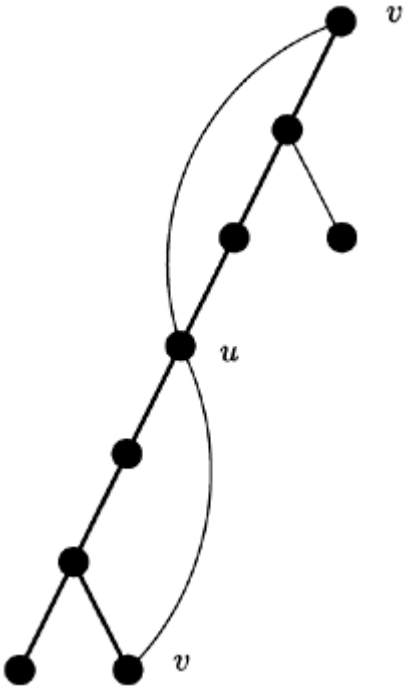


FIGURE 6.12
Two kinds of fronds

The next thing to notice is that the *LowPt[u]* is correctly computed. For if *u* is a leaf-node of the search tree, then all edges *uu* are fronds, and *u* is an ancestor of *u*. The algorithm computes *LowPt[u]* as the minimum *DFNum[u],* for all such *u*. Therefore, if *u* is a leaf-node, the low-point is correctly computed. We can now use induction on the depth of the recursion. If *u* is not a leaf-node, then some DFSEARCH*(u)* will be called from node *u*. The depth of the recursion from *u* will be less than that from *u,* so that we can assume that DFSEARCH*(u)* will correctly compute *LowPt[u]*. Upon returning from this recursive call, *LowPt[u]* is compared with the current value of *LowPt[u],* and the minimum is taken, for every unsearched $v \longrightarrow u$. Therefore, after visiting node *u, LowPt[u]* will always have the correct value.

So far the algorithm computes low-points and uses them to find the cut-vertices of *G*. We still need to find the edges in each block. While visiting node *u,* all new edges *uu* are stacked. If it is discovered that *LowPt[u]=DFNum[u];* so that *u* is a cut-vertex, then the edges in the block containing *uu* are all those edges on the stack up to, and including, *uu*.

THEOREM 6.9 Each time that

$$LowPt[u]=DFNum[u]$$

occurs in DFSEARCH*(u), the block containing uu consists of those edges on the stack up to and including uu*.

PROOF Let *Buu* denote the block containing *uu*. The proof is by induction on the number of times that *LowPt[u]=DFNum[u]* occurs. Consider the first time it occurs. DFSEARCH*(u)* has just been called, while visiting node *u*. Edge *uu* has been placed on the stack. DFSEARCH*(u)* constructs the branch of the search tree at *u* containing *u*. By Lemma 6.7, this contains all vertices of *G−u* connected to *u*. Call this set of vertices *B*. By the definition of the low-point, there are no fronds joining *u* or any de scendant of *u* to any ancestor of *u*. So *u* separates *B* from the rest of the graph. Therefore $B_{uv} \subseteq G[B \cup \{u\}]$.

Suppose now that *B* contained a cut-vertex *w*. No leaf-node of the DF-tree can be a cut-vertex, so some DFSEARCH*(x)* is called while visiting node *w*. It will visit all nodes of *G−w* connected to *x*. Upon returning to node *w,* it would find that *LowPt[x]=DFNum[w],* which is impossible, since this occurs for the first time at node *u*. Therefore *Buu* consists of exactly those edges encountered while performing DFSEARCH*(u);* that is, those on the stack.

Upon returning from DFS*(u)* and detecting that *LowPt[u]=DFNum[u],* all edges on the stack will be unstacked up to, and including, *uu*. This is equivalent to removing all edges of *Buu* from the graph. The remainder of the DFS now continues to work on *G−B*. Now *Buu* is an *end-block* of *G* (i.e., it has at most one cut-vertex) for *u* is the first vertex for which a block is detected. If *G* is 2-connected, then *G=Buu,* and the algorithm is finished. Otherwise, *G−B* is connected, and consists of the remaining blocks of *G*. It has one less block than *G,* so that each time *LowPt[u]=DFNum[u]* occurs in *G−B,* the edges on the stack will contain another block. By induction, the algorithm finds all blocks of *G*.

Each time the condition *LowPt[u]=DFNum[u]* occurs, the algorithm has found the edges of a block of *G*. In this case, *u* will usually be a cut-vertex of *G*. The exception is when *u* is the root of the DF-tree, since *u* has no ancestors in the tree. Exercise 6.4.3 shows how to deal with this situation.

6.4.1 Complexity

The complexity of DFBLOCKS() is very easy to work out. For every $u \in V(G)$, all $v \longrightarrow u$ are considered. This takes

$$\sum_u \text{DEG}(u) = 2\varepsilon$$

steps. Each edge is stacked and later unstacked, and a number of comparisons are performed in order to compute the low-points. So the complexity is *O(ε)*.

Exercises

6.4.1 Can the spanning tree shown in the graph illustrated in Figure 6.13 be a DF-tree, with the given root-node? If so, assign a DF-numbering to the vertices.

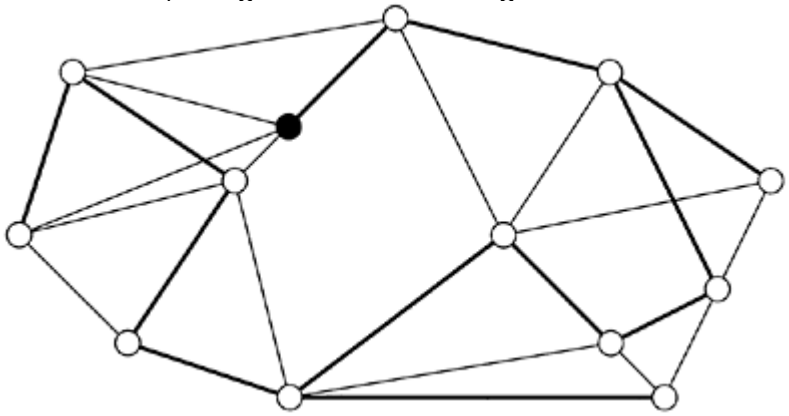


FIGURE 6.13
Is this a DF spanning tree?

6.4.2 Program the DFBLOCKS() algorithm to find all the blocks of a connected graph *G*. Print a list of the edges in each block. Choose the starting vertex to be sometimes a cut-vertex, sometimes not.

6.4.3 Modify your program to print also a list of the cut-vertices of *G,* by storing them on an array. A vertex *u* is a cut-vertex if *LowPt[u]=DFNum[u]* occurs while visiting edge *uu* at node *u*. However, if *u* is the root-node of the DF-tree, then it will also satisfy this condition, even when *G* is 2-connected. Find a way to modify the algorithm so that it correctly determines when the root-node is a cut-vertex.

6.4.4 A separable graph has κ=1, but can be decomposed into blocks, its maximal non-separable subgraphs. A tree is the only separable graph which does not have a 2-connected subgraph, so that every block of a tree is an edge. Suppose that *G* has κ=2. In general, *G* may have 3-connected subgraphs. Characterize the class of 2-connected graphs which do not have any 3-connected subgraph.

6.4.5 Let *G* be 3-connected. Prove that every pair of vertices is connected by at least three internally disjoint paths.

6.4.6 Let *G* have κ=2, and consider the problem of finding all separating pairs *{u,u}* of *G* using a DFS. Prove that for every separating pair *{u,u},* one of *u* and *u* is an ancestor of the other in any DF-tree. Refer to Figure 6.14.

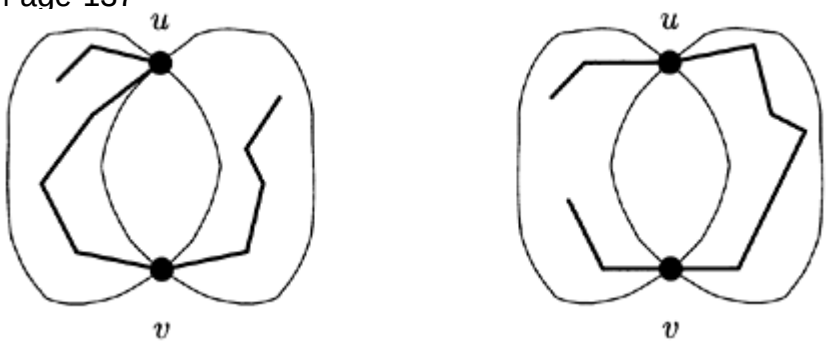


FIGURE 6.14
DFS with separating set *{u,u}*
6.4.7 Suppose that deleting *{u,u}* separates *G* into two or more components. Let *G*1 denote one component and *G*2 the rest of *G*. Show that there are two possible ways in which a DFS may visit *u* and *u,* as illustrated in Figure 6.14. Devise a DFS which will find all pairs *{u,u}* which are of the first type. (*Hint:* You will need *LowPt2 [u],* the second low-point of *u*. Define it and prove that it works.)

6.5 Notes

The example of Figure 6.3 is based on HARARY [59]. Read's algorithm to find the blocks of a graph is from READ [99]. The depth-first search algorithm is from HOPCROFT and TARJAN [67]. See also TARJAN [111]. Hopcroft and Tarjan's application of the depth-first search to find the blocks of a graph was a great breakthrough in algorithmic graph theory. The depth-first search has since been applied to solve a number of difficult problems, such as determining whether a graph is planar in linear time, and finding the 3-connected components of a graph.
Algorithms for finding the connectivity and edge-connectivity of a graph are described in Chapter 8. An excellent reference for connectivity is TUTTE [120], which includes a detailed description of the 3-connected components of a graph. A depth-first seach algorithm to find the 3-connected components of a graph can be found in HOPCROFT and TARJAN [68].

Page 138
This page intentionally left blank.

Page 139
7
Alternating Paths and Matchings

7.1 Introduction

Matchings arise in a variety of situations as assignment problems, in which pairs of items are to be matched together, for example, if people are to be assigned jobs, if sports teams are to matched in a tournament, if tasks are to be assigned to processors in a computer, whenever objects or people are to be matched on a one-to-one basis.
In a graph *G,* a *matching M* is a set of edges such that no two edges of *M* have a vertex in common. Figure 7.1 illustrates two matchings *M*1 and *M*2 in a graph *G*.

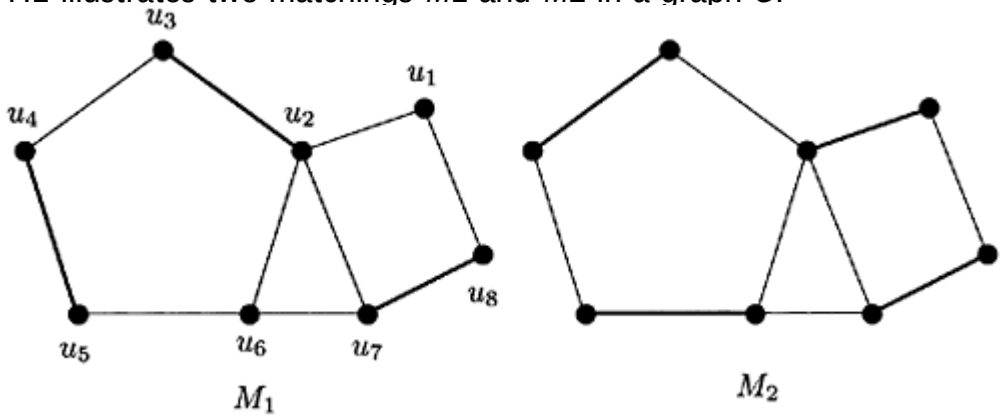


FIGURE 7.1
Matchings
Let *M* have *m* edges. Then 2 m vertices of *G* are matched by *M*. We also

Page 140
say that a vertex *u* is *saturated* by *M* if it is matched, and *unsaturated* if it is not matched. In general, we want *M* to have as many edges as possible.
DEFINITION 7.1: M is a *maximum matching* in *G* if no matching of *G* has more edges.
For example, in Figure 7.1, |*M*1|=3 and |*M*2|=4. Since |*G*|=8, *M*2 is a maximum matching. A matching which saturates every vertex is called *a perfect matching.* Obviously a perfect matching is always a maximum matching. *M*1 is not a maximum matching, but it is a *maximal matching;* namely *M*1 cannot be extended by the addition of any edge *uu* of *G*. However, there is a way to build a bigger matching out of *M*1. Let *P* denote the path (*u*1, *u*2,..., *u*6) in Figure 7.1.

DEFINITION 7.2: Let *G* have a matching *M*. An *alternating path P* with respect to *M* is any path whose edges are alternately in *M* and not in *M*. If the endpoints of *P* are unsaturated, then *P* is an *augmenting path*.
So *P*=(*u*1, *u*2,..., *u*6) is an augmenting path with respect to *M*1. Consider the subgraph formed by the *exclusive or* operation $M = M_1 \oplus E(P)$ (also called the *symmetric difference,* $(M_1 - E(P)) \cup (E(P) - M_1)$). *M* contains those edges of *P* which are not in *M*1, namely, *u*1*u*2, *u*3*u*4, and *u*5*u*6. *M* is a bigger matching than *M*1. Notice that *M*=*M*2.

LEMMA 7.1 Let G have a matching M. Let P be an augmenting path with respect to M. Then $M' = M \oplus E(P)$ *is a matching with one more edge than M.*
PROOF Let the endpoints of *P* be *u* and *u*. *M′* has one more edge than *M,* since *u* and *u* are unsaturated in M, but saturated in *M′*. All other vertices that were saturated in *M* are still saturated in *M′*. So *M′* is a matching with one more edge.
The key result in the theory of matchings is the following:
THEOREM 7.2 **(Berge's theorem)** *A matching M in G is maximum if and only if G contains no augmenting path with respect to M.*

PROOF ⇒**:** If *M* were a maximum matching and *P* an augmenting path, then $M \oplus E(P)$ would be a larger matching. So there can be no augmenting path if *M* is maximum.
⇐: Suppose that *G* has no augmenting path with respect to *M*. If *M* is not maximum, then pick a maximum matching *M′*. Clearly |*M′*|>|*M*|. Let *H*= $M \oplus M'$. Consider the subgraph of *G* that *H* defines. Each vertex *u* is incident on at most one *M*-edge and one *M′*-edge, so that in *H,* DEG*(u)*≤2. Every path in

Page 141
H alternates between *M*-edges and *M′*-edges. So *H* consists of alternating paths and cycles, as illustrated in Figure 7.2.

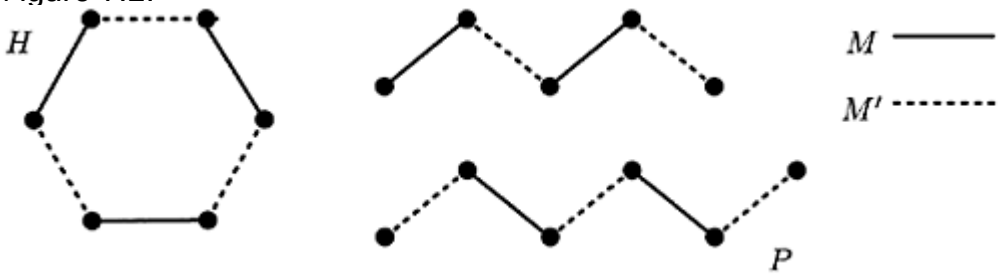


FIGURE 7.2
Alternating paths and cycles
Each cycle must clearly have even length, with an equal number of edges of *M* and *M′*. Since |*M′*|>|*M*|, some path *P* must have more *M′*-edges than *M*-edges. It can only begin and end with an *M′*-edge, so that *P* is augmenting with respect to *M*. But we began by assuming that *G* has no augmenting path for *M*. Consequently, *M* was initially a maximum matching.
This theorem tells us how to find a maximum matching in a graph. We begin with some matching *M*. If *M* is not maximum, there will be an unsaturated vertex *u*. We then follow alternating paths from *u*. If some unsaturated vertex *u* is reached on an alternating path *P,* then *P* is an augmenting *uu*-path. Set $M \leftarrow M \oplus E(P)$, and repeat. If the method that we have chosen to follow alternating paths is sure to find all such paths, then this technique is guaranteed to find a maximum matching in *G*.
In bipartite graphs it is slightly easier to follow alternating paths and therefore to find maximum matchings, because of their special properties. Let *G* have bipartition *(X,Y)*. If $S \subseteq X$, then the *neighbor set* of *S* is *N(S),* the set of *Y*-vertices adjacent to *S*. Sometimes *N(S)* is called the *shadow set* of *S*. If *G* has a perfect matching *M,* then every $x \in S$ will be matched to some $y \in Y$ so that |*N(S)*|≥|*S*|, for every $S \subseteq X$. HALL [58] proved that this necessary condition is also sufficient.
THEOREM 7.3 **(Hall's theorem)** *Let G have bipartition (X,Y). G has a matching saturating every* $x \in X$ *if and only if* |*N(S)*|≥|*S*|, *for all* $S \subseteq X$.
PROOF We have all ready discussed the necessity of the conditions. For the converse suppose that |*N(S)*| ≥|*S*|, for all $S \subseteq X$. If *M* does not saturate all of

Page 142

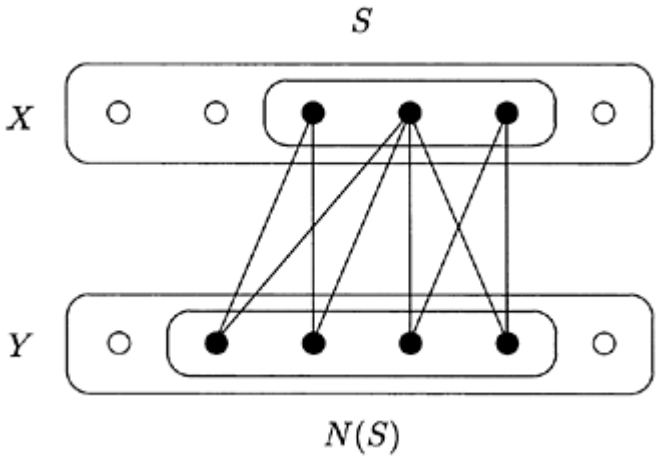


FIGURE 7.3
The neighbor set

X, pick an unsaturated $u \in X$, and follow all the alternating paths beginning at *u*. (See Figure 7.4.)

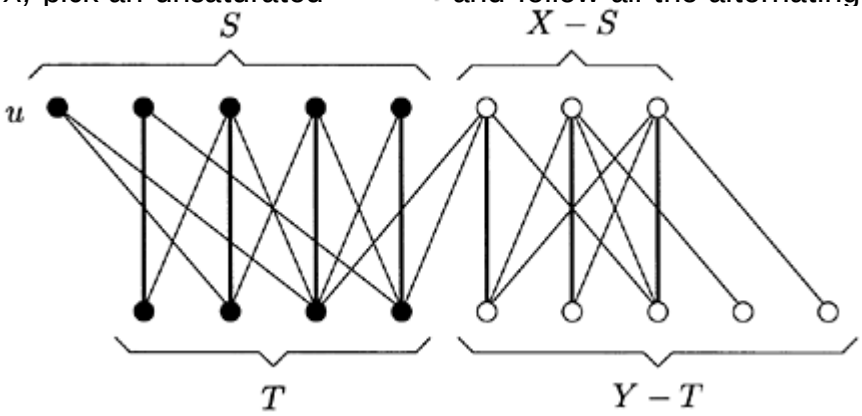


FIGURE 7.4
Follow alternating paths

Let $S \subseteq X$ be the set of *X*-vertices reachable from *u* on alternating paths, and let *T* be the set of *Y*-vertices reachable. With the exception of *u,* each vertex $x \in S$ is matched to some $y \in T$, for *S* was constructed by extending alternating paths from $y \in T$ to $x \in S$whenever *xy* is a matching edge. Therefore $|S|=|T|+1$. Now there may be other vertices $X-S$ and $Y-T$. However, there can be no edges $[S,Y-T]$, for such an edge would extend an alternating path to a vertex of $Y-T$, which is not reachable from *u* on an alternating path. So every $x \in S$ can

only be joined to vertices of *T;* that is, $T=N(S)$. It follows that $|S|>|N(S)|$, a contradiction. Therefore every vertex of *X* must be saturated by *M*.

COROLLARY 7.4 Every k-regular bipartite graph has a perfect matching, if $k>0$.

PROOF Let *G* have bipartition *(X,Y)*. Since *G* is *k*-regular, $\varepsilon=k\cdot|X|=k\cdot|Y|$, so that $|X|=|Y|$. Pick any $S \subseteq X$. How many edges have one end in *S*? Exactly $k\cdot|S|$. They all have their other end in *N(S)*. The number of edges with one endpoint in *N(S)* is $k\cdot|N(S)|$. So $k\cdot|S|\leq k-|N(S)|$, or $|S|\leq|N(S)|$, for all $S \subseteq X$. Therefore *G* has a perfect matching.

Exercises

7.1.1 Find a formula for the number of perfect matchings of *K*2*n* and *Kn,n*.

7.1.2 (Hall's theorem.) Let *A*1, *A*2,..., *An* be subsets of a set *S*. A *system of distinct representatives* for the family {*A*1, *A*2,..., *An*} is a subset {*a*1, *a*2,..., *an*} of *S* such that $a_1 \in A_1, a_2 \in A_2, \ldots, a_m \in A_m$, and $ai \neq aj$, for $i \neq j$. Example:

*A*1=students taking computer science 421
*A*2=students taking physics 374
*A*3=students taking botany 464
*A*4=students taking philosophy 221

The sets *A*1, *A*2, *A*3, *A*4 may have many students in common. Find four distinct students *a*1, *a*2, *a*3, *a*4, such

that $a_1 \in A_1, a_2 \in A_2, a_3 \in A_3,$ and $a_4 \in A_4$ to represent each of the four classes.
Show that {*A*1, *A*2,..., *An*} has a system of distinct representatives if and only if the union of every combination of *k* of the subsets *Ai* contains at least *k* elements, for all *k*=1, 2,..., *n. (Hint:* Make a bipartite graph *A*1, *A*2,..., *An* versus all $a_j \in S,$ and use Hall's theorem.)

7.2 The Hungarian algorithm

We are now in a position to construct an algorithm which finds a maximum matching in bipartite graphs, by following alternating paths from each unsaturated $u \in X.$ How can we best follow alternating paths? Let *n*=|*G*|. Suppose that we store the matching as an integer array *Match [x], x*=1, 2,..., *n,* where *Match [x]* is the vertex matched to *x* (so *Match [Match [x]*)=*x,* if *x* is saturated).

We use *Match[x]*=0 to indicate that *x* is unsaturated. We could use either a DFS or BFS to construct the alternating paths. A DFS is slightly easier to program, but a BF-tree tends to be shallower than a DF-tree, so that a BFS will likely find augmenting paths more quickly, and find shorter augmenting paths, too. Therefore the BFS is used for matching algorithms.
The array used to represent parents in the BF-tree can be used in combination with the *Match*[·] array to store the alternating paths. We write *PrevPt[u]* for the parent of *u* in a BF-tree. It is the previous point to *u* on an alternating path to the root. This is illustrated in Figure 7.5.

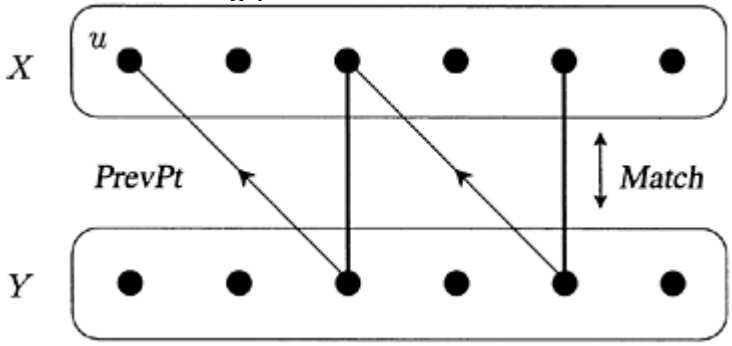


FIGURE 7.5
Storing the alternating paths

We also need to build the sets *S* and *N(S)* as queues, which we store as the arrays *ScanQ* and *NS,* respectively. The algorithm for finding a maximum matching in bipartite graphs is Algorithm 7.2.1. It is also called the *Hungarian algorithm* for maximum matchings in bipartite graphs.

Algorithm 7.2.1: MAXMATCHING*(G)*
comment: Hungarian algorithm. *G* has bipartition *(X,Y),* and *n* vertices.
for *i*←1 **to** *n* **do** *Match [i]*←0
for each $u \in X$

comment: u is currently unsaturated
$ScanQ[1] \leftarrow u$
$QSize \leftarrow 1$
comment: construct alternating paths from u using a BFS
for $i \leftarrow 1$ **to** n **do** $PrevPt[i] \leftarrow 0$
$k \leftarrow 1$
repeat
$x \leftarrow ScanQ[k]$
for each $y \longrightarrow x$ **do**
if $y \notin NS$
then
add y to NS
$PrevPt[y] \leftarrow x$
if y is unsaturated
then
comment: augmenting path found
AUGMENT(y)
go to 1 "u is now saturated"
add $Match[y]$ **to** $ScanQ$
$k \leftarrow k + 1$ "advance $ScanQ$"
until $k > QSize$
comment: $ScanQ$ now contains a set S, and NS contains the neighbor-set $N(S)$ such that $|S| = |N(S)| + 1$, no matching can saturate all of S
delete S and $N(S)$ from the graph
1 :
comment: $Match[\cdot]$ now contains a maximum matching

Notice that the algorithm needs to be able to determine whether $y \in NS$. This can be done by storing a boolean array. Another possibility is to use *PrevPt[u]*=0 to indicate that $v \notin N(S)$. We can test if *y* is unsaturated by checking whether *Match[y]*=0. AUGMENT *(y)* is a procedure that computes $M \leftarrow M \oplus E(P)$, where *P* is the augmenting path found. Beginning at vertex *y,* it alternately follows *PrevPt*[·] and *Match*[·] back to the initial unsaturated vertex, which is the root-node of the BF-tree being constructed. This is illustrated in Figure 7.6.

Algorithm 7.2.2: AUGMENT *(y)*

comment: follow the augmenting path, setting $M \leftarrow M \oplus E(P)$
repeat
w←PreuPt[y]
Match [y]←w
u←Match [w]
Match [w]←y
y←u
until *y*=0

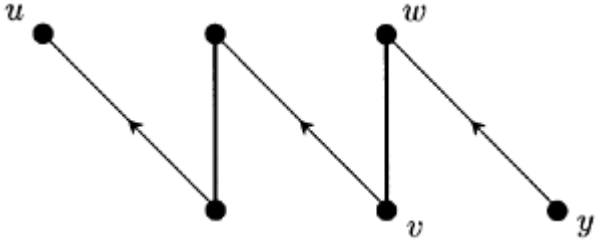


FIGURE 7.6
Augmenting the matching
The BFS constructs an alternating search tree. It contains all vertices reachable from the root-node *u* on alternating paths. Vertices at even distance from *u* in the tree form the set *S,* and those at odd distance form *N(S)*. The vertices of *S* are sometimes called *outer vertices,* and those of *N(S) inner vertices*. All the actual searching is done from the outer vertices.
THEOREM 7.5 The Hungarian algorithm constructs a maximum matching in a bipartite graph.
PROOF Let *G* have bipartition *(X,Y)*. If the algorithm saturates every vertex of *X,* then it is certainly a maximum matching. Otherwise some vertex *u* is not matched. If there is an augmenting path *P* from *u,* it must alternate between *X* and *Y,* since *G* is bipartite. The algorithm constructs the sets *S* and *N(S),* consisting of all vertices of *X* and *Y,* respectively, that can be reached on alternating paths. So *P* will be found if it exists. If *u* cannot be saturated, then we know that $|S|=|N(S)|+1$. Every vertex of *S* but *u* is matched. *S* and *N(S)* are then deleted from the graph. Does the deletion of these vertices affect the rest of the

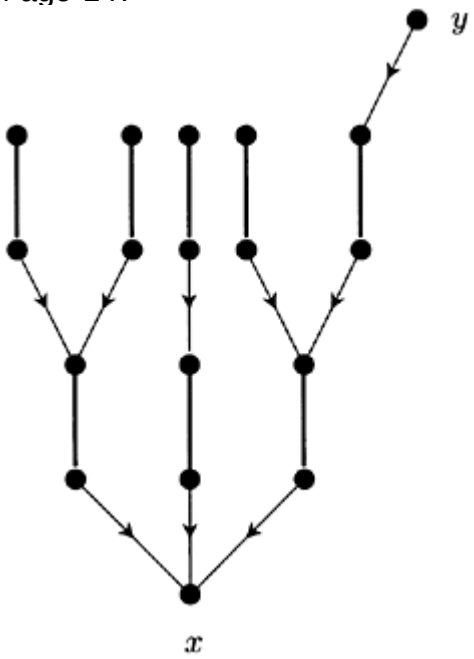


FIGURE 7.7
The alternating search tree
algorithm? As in Hall's theorem, there are no edges $[S,Y-N(S)]$. Suppose that alternating paths from a vertex $v \in X$ were being constructed. If such a path were to reach a vertex *y* in the deleted *N(S),* it could only extend to other vertices of *S* and *N(S)*. It could not extend to an augmenting path. Therefore these vertices can be deleted. Upon completion, the algorithm will have produced a matching *M* for which there are no augmenting paths in the graph. By Theorem 7.2, *M* is a maximum matching.

7.2.1 Complexity

Suppose that at the beginning of the for-loop, *M* has *m* edges. The largest possible size of *S* and *N(S)* is then $m+1$, and *m,* respectively. The number of edges *[S,N(S)]* is at most $m(m+1)$. In the worst possible case, *S* and *N(S)* will be built up to this size, and $m(m+1)$ edges between them will be encountered. If an augmenting path is now found, then *m* will increase by one to give a worst case again for the next iteration. The length of the augmenting path will be at most $2m+1$, in case all *m* matching edges are in the path. The number of steps performed in this iteration of the for-loop will then be at most $m(m+1)+(2m+1)$. Since $|X|+|Y|=n$, the number of vertices, one of $|X|$ and $|Y|$ is $\leq n/2$. We

can take *X* as the smaller side. Summing over all iterations then gives

$$\sum_{m=0}^{\frac{n}{2}-1} m(m+1)+(2m+1) = \sum 2\binom{m+1}{2} + (2m+1)$$

$$= 2\binom{n/2+1}{3} + 2\binom{n/2}{2} + \frac{n}{2}.$$

The leading term in the expansion is $n3/24$, so that the algorithm is of order $O(n3)$, with a small constant coefficient. It can be improved with a more careful choice of augmenting paths. HOPCROFT and KARP [65] maintain several augmenting paths, and augment simultaneously on all of them to give $O(n2.5)$. This can also be accomplished with network flow techniques.

Exercises

7.2.1 Program the Hungarian matching algorithm. The output should consist of a list of the edges in a maximum matching. If there is no matching saturating the set *X,* this should be indicated by printing out the sets $S \subseteq X$ found whose neighbor set *N(S)* is smaller than *S*. Use the graphs in Figure 7.8 for input. The set *X* is marked by shaded dots, and *Y* by open dots.

7.3 Perfect matchings and 1-factorizations

Given any graph *G* and positive integer *k,* a *k-factor* of *G* is a spanning subgraph that is *k*-regular. Thus a perfect matching is a 1-factor. A 2-factor is a union of cycles that covers *V(G),* as illustrated in Figure 7.9. The reason for this terminology is as follows. Associate indeterminates $x1, x2, \ldots, xn$ with the *n* vertices of a graph. An edge connecting vertex *i* to *j* can be represented by the expression $xi-xj$. Then the entire graph can be represented (up to sign) by the product $P(G) = \prod_{ij \in E(G)} (x_i - x_j)$. For example, if *G* is the 4-cycle, this product becomes $(x1-x2)\ (x2-x3)\ (x3-x4)\ (x4-x1)$. Since the number of terms in the product is $\varepsilon(G)$, when it is multiplied out, there will be ε x's in each term. A 1-factor of *P(G),* for example, $(x1-x2)\ (x3-x4)$, is a factor that contains each xi exactly once. This will always correspond to a perfect matching in *G,* and so on. With some graphs it is possible to decompose the edge set into perfect matchings. For example, if *G* is the cube, we can write $E(G) = M_1 \cup M_2 \cup M_3$, where

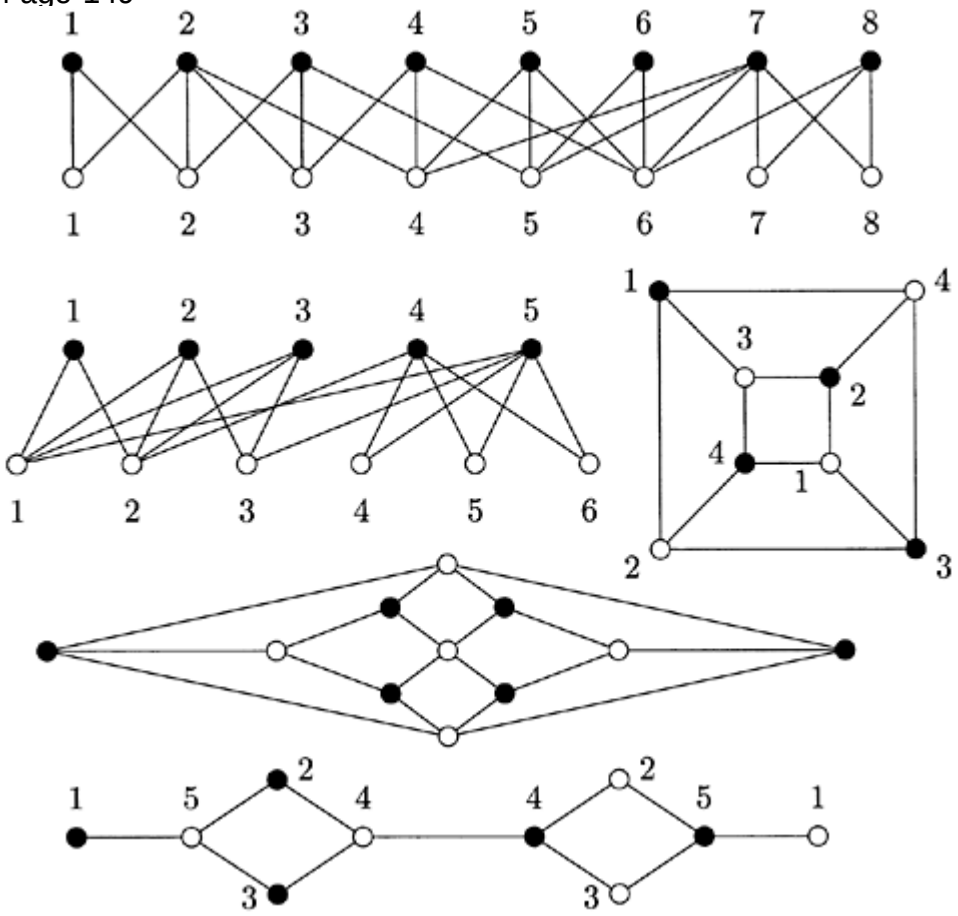


FIGURE 7.8
Sample graphs

*M*1={12,34,67,85}, *M*2={23,14,56,78}, and *M*3={15,26,37,48}, as shown in Figure 7.10. Each edge of *G* is in *exactly one* of *M*1, *M*2, or *M*3.

In general, a *k-factorization* of a graph *G* is a decomposition of *E(G)* into $H_1 \cup H_2 \cup \ldots \cup H_m$, where each *Hi* is a *k*-factor, and each *Hi* and *Hj* have no edges in common. The above decomposition of the cube is a 1-factorization. Therefore we say the cube is 1-factorable.

LEMMA 7.6 Kn,nis 1-factorable.

PROOF Let *(X,Y)* be the bipartition of *Kn,n,* where *X*={*x*0,*x*1,...,*x*n−1} and *Y*={*y*0,*y*1,...,*yn*−1}. Define *M*0={*xiyi*|*i*=0,1,...,*n*−1}, *M*1= {*xiyi*+1|*i*=0,1,...,*n*−1}, etc., where the addition is modulo *n*. In general *Mk*={*xiyi*+*k*|*i*=0,1,...,*n*−1}. Clearly *Mj* and *Mk* have no edges in common, for any *j* and *k,* and together *M*0, *M*1,..., *Mn*−1 contain all of *E(G)*.

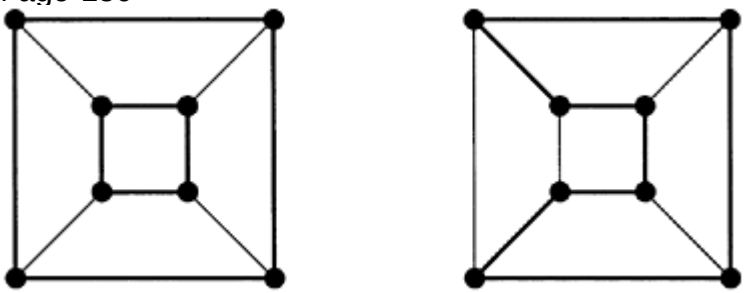


FIGURE 7.9
2-factors of the cube

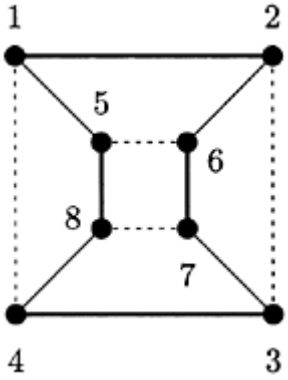


FIGURE 7.10
A 1-factorization of the cube

Thus we have a 1-factorization of *Kn,n*

*LEMMA 7.7 K*2*nis 1-factorable.*

PROOF Let $V(K_{2n}) = \{0, 1, 2, \ldots, 2n-2\} \cup \{\infty\}$. Draw *K*2*n* with the vertices 0, 1,..., 2*n*−2 in a circle, placing ∞ in the center of the circle. This is illustrated for *n*=4 in Figure 7.11. Take $M_0 = \{(0,\infty), (1,2n-2), (2,2n-3), \ldots, (n-1,n)\} = \{(0,\infty)\} \cup \{(i,-i) \mid i = 1,2,\ldots,n-1\}$, where the addition is modulo 2*n*−1. *M*0 is illustrated by the thicker lines in Figure 7.11.

We can then "rotate" *M*0 by adding one to each vertex, $M_1 = M_0 + 1 = \{(i+1, j+1) \mid (i,j) \in M_0\}$, where ∞+1=∞, and addition is modulo 2*n*−1. It is easy to see from the diagram that *M*0 and *M*1 have no edges in common. Continuing like this, we have

*M*0,*M*1,*M*2,...,*M*2*n*−2,

where *Mk*=*M*0+*k*. They form a 1-factorization of *K*2*n*.

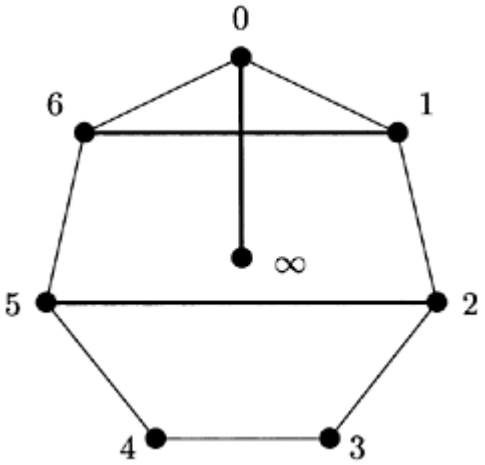


FIGURE 7.11
1-factorizing *K*2*n*, where *n*=4

We can use a similar technique to find a 2-factorization of *K*2*n*+1.

*LEMMA 7.8 K*2*n*+1 *is 2-factorable*.

PROOF Let $V(K_{2n+1}) = \{0, 1, 2, \ldots, 2n-1\} \cup \{\infty\}$. As in the previous lemma, draw the graph with the vertices in a circle, placing ∞ in the center. The first 2-factor is the cycle *H*0=(0, 1,−1, 2,−2,..., *n*−1, *n*+1, *n*, ∞), where the arithmetic is modulo 2*n*. This is illustrated in Figure 7.12, with *n*=3. We then rotate the cycle to get *H*1, *H*2,..., *Hn*−1, giving a 2-factorization of *K*2*n*+1.

Exercises

7.3.1 Find all perfect matchings of the cube. Find all of its 1-factorizations.
7.3.2 Find all perfect matchings and 1-factorizations of *K*4 and *K*6.
7.3.3 Prove that the Petersen graph has no 1-factorization.
7.3.4 Prove that for *k*>0 every *k*-regular bipartite graph is 1-factorable.
7.3.5 Describe another 1-factorization of *K*2*n*, when *n* is even, using the fact that *Kn,n* is a subgraph of *K*2*n*.

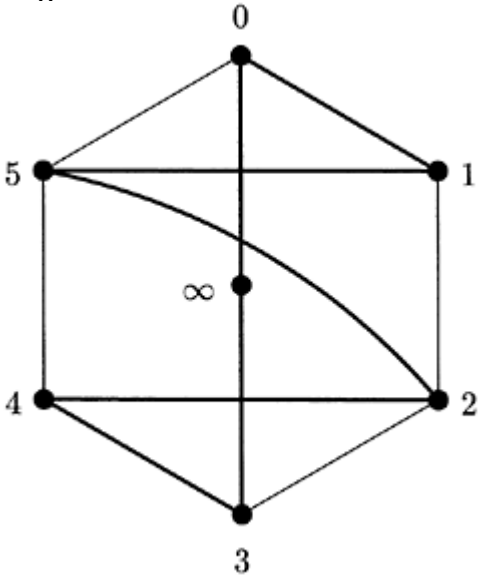


FIGURE 7.12
2-factorizing *K*2*n*+1, where *n*=3

7.3.6 Let *M*1, *M*2,..., *Mk* and $M_1', M_2', \ldots, M_k'$ be two 1-factorizations of a *k*-regular graph *G*. The two factorizations are isomorphic if there is an automorphism *θ* of *G* such that for each *i*, $\theta(M_i) = M_j'$, for some *j*; that is, *θ* induces a mapping of *M*1, *M*2,..., *Mk* onto $M_1', M_2', \ldots, M_k'$. How many non-isomorphic 1-factorizations are there of *K*4 and *K*6?
7.3.7 How many non-isomorphic 1-factorizations are there of the cube?

7.4 The subgraph problem

Let *G* be a graph and let *f: V(G)*→{0, 1, 2,...} be a function assigning a non-negative integer to each vertex

of *G*. An *f-factor* of *G* is a subgraph *H* of *G* such that deg*(u,H)=f(u),* for each $u \in V(G)$. So a 1-factor is an *f*-factor in which each *f(u)*=1.

Problem 7.1: Subgraph Problem
Instance: a graph *G* and a function *f: V(G)*→{0, 1, 2,...}.
Find: an *f*-factor in *G,* if one exists.

There is an ingenious construction by TUTTE [117], that transforms the subgraph problem into the problem of finding a perfect matching in a larger graph *G′*.

Page 153

Construct *G′* as follows. For each edge *e=uu* of *G, G′* has two vertices *eu* and *eu,* such that $e_u e_v \in E(G')$. For each vertex it of *G,* let *m(u)*=deg*(u)−f(u)*. Corresponding to $u \in V(G)$, G' has *m(u)* vertices *u*1, *u*2,..., *um(u)*. For each edge $e = uv \in E(G)$, *u*1, *u*2,..., *um(u)* are all adjacent to $e_u \in V(G')$. This is illustrated in Figure 7.13, where deg*(u)*=5 and *f(u)*=3.

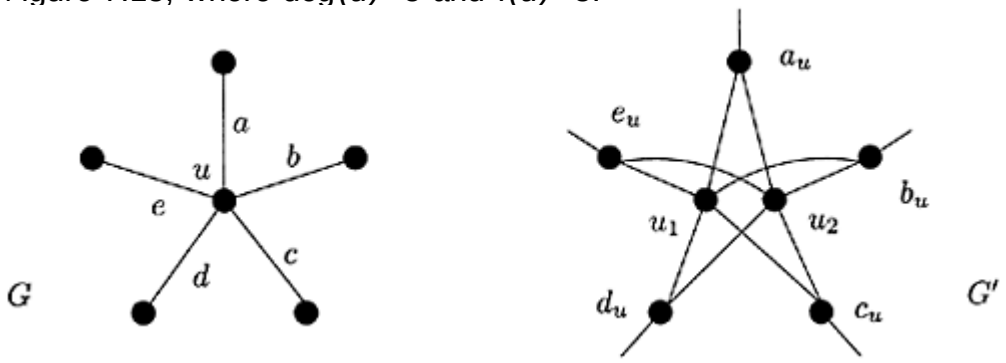


FIGURE 7.13
Tutte's transformation

THEOREM 7.9 G has an f-factor if and only if G′ has a perfect matching.

PROOF Suppose that *G* has an *f*-factor *H*. Form a perfect matching *M* in *G′* as follows. For each edge $uv \in H$, $e_u e_v \in M$. There are *m(u)*=deg*(u)−f(u)* remaining edges at vertex $u \in V(G)$. In *G′,* these can be matched to the vertices *u*1, *u*2,..., *um(u)* in any order.

Conversely, given a perfect matching $M \subseteq G'$, the vertices *u*1, *u*2,..., *um(u)* will be matched to *m(u)* vertices, leaving *f(u)* adjacent vertices of the form *eu* not matched to any *ui*. They can therefore only be matched to vertices of the form *eu* for some *u*. Thus *f(u)* edges *eueu* are selected corresponding to each vertex *u*. This gives an *f*-factor of *G*.

So finding an *f*-factor in *G* is equivalent to finding a perfect matching in *G′*. If *G* has *n* vertices and *ε* edges, then *G′* has

$$4\varepsilon - \sum f(u)$$

vertices and

$$\varepsilon + \sum (\deg^2(u) - \deg(u) f(u))$$

edges. Finding perfect matchings in non-bipartite graphs is considerably more complicated than in bipartite graphs, but is still very efficient. Edmonds' algorithm [38] will find a maximum matching in time *O*(*n*3). Thus, the subgraph

Page 154

problem can be solved using perfect matchings. However, it can be solved more efficiently by a direct algorithm than by constructing *G′* and then finding a maximum matching.

7.5 Coverings in bipartite graphs

A *covering* or *vertex cover* of a graph *G* is a subset $U \subseteq V(G)$ that covers every edge of *G;* that is, every edge has at least one endpoint in *U*.

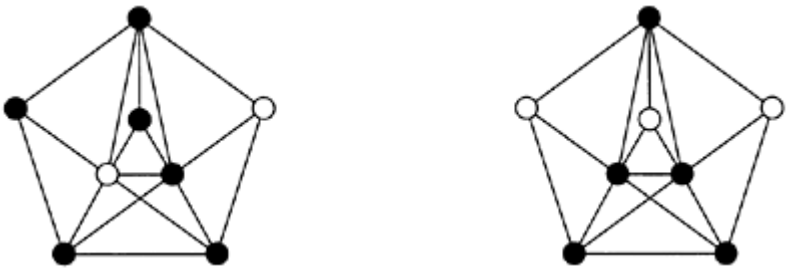


FIGURE 7.14
Coverings in a graph

In general, we want the smallest covering possible. This is called a *minimum covering*. Figure 7.14 shows two coverings, indicated by shaded vertices. The covering with six vertices is minimal; namely, it has no subset that is a smaller covering. The other is a minimum covering; namely, *G* has no smaller covering. This is because any covering must use at least three vertices of the outer 5-cycle, and at least two vertices of the inner triangle, giving a minimum of five vertices.

In bipartite graphs, there is a very close relation between minimum coverings and maximum matchings. In general, let *M* be a matching in a graph *G,* and let *U* be a covering. Then since *U* covers every edge of *M,* $|U| \geq |M|$. This is true even if *U* is minimum or if *M* is maximum. Therefore, we conclude that if $|U|=|M|$ for some *M* and *U,* then *U* is minimum and *M* is maximum. In bipartite graphs, equality can always be achieved.

THEOREM 7.10 **(König's theorem)** *If G is bipartite, then the number of edges in a maximum matching equals the number of vertices in a minimum covering.*

PROOF Let *M* be a maximum matching, and let *(X,Y)* be a bipartition of *G,* where $|X| \leq |Y|$. Let $W \subseteq X$ be the set of all *X*-vertices not saturated by *M*. If $W = \emptyset$, then $U=X$ is a covering with $|U|=|M|$. Otherwise construct the

set of all vertices reachable from *W* on alternating paths. Let *S* be the *X*-vertices reachable, and *T* the *Y*-vertices reachable. Take $U = T \cup (X - S)$. Then *U* is a covering with $|U|=|M|$, as illustrated in Figure 7.15.

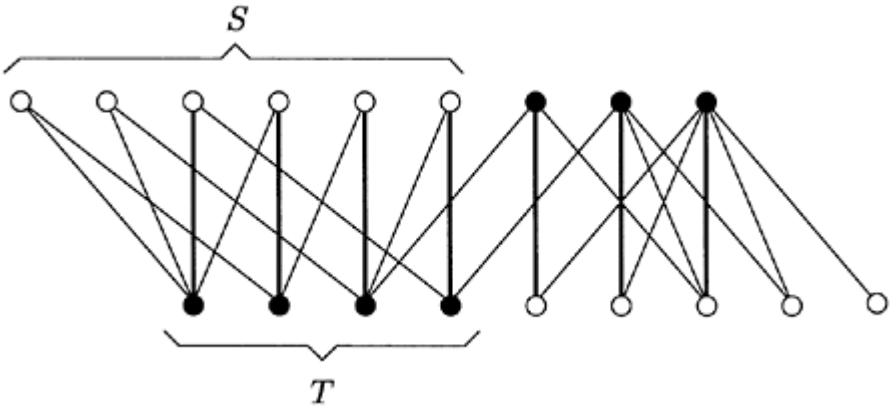


FIGURE 7.15
Minimum covering and maximum matching in a bipartite graph

7.6 Tutte's theorem

Tutte's theorem gives a necessary and sufficient condition for any graph to have a perfect matching.

Let $S \subseteq V(G)$. In general, $G-S$ may have several connected components. Write odd($G-S$) for the number of components with an odd number of vertices. The following proof of Tutte's theorem is due to LOVÁSZ [84].

THEOREM 7.11 **(Tutte's theorem)** *A graph G has a perfect matching if and only if odd(G−S)≤|S|, for every subset* $S \subseteq V(G)$.

PROOF ⇒**:** Suppose that *G* has a perfect matching *M* and pick any $S \subseteq V(G)$. Let $G1, G2, \ldots, Gm$ be the odd components of $G-S$. Each *Gi* contains at least one vertex matched by *M* to a vertex of *S*. Therefore $\text{odd}(G-S)=m \leq |S|$. See Figure 7.16.

⇐**:** Suppose that $\text{odd}(G-S)=m \leq |S|$, for every $S \subseteq V(G)$. Taking $S = \emptyset$ gives odd*(G)*=0, so $n=|G|$ is even. The proof is by reverse induction on *ε(G),* for any given *n*. If *G* is the complete graph, it is clear that *G* has a perfect

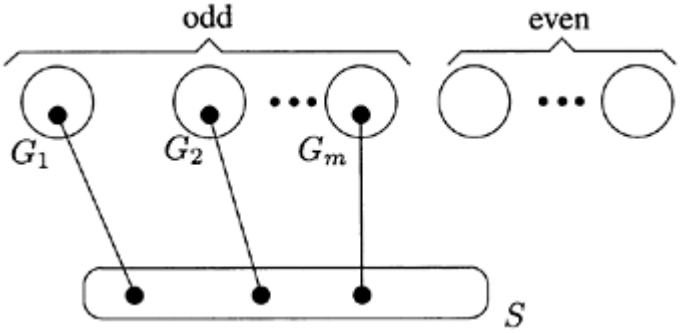


FIGURE 7.16
Odd and even components of $G-S$

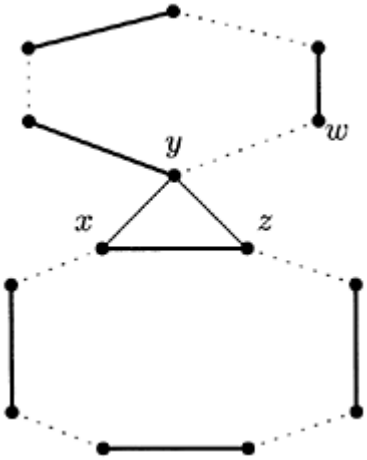


FIGURE 7.17
$H = M_1 \oplus M_2$**, case 1**

matching, so the result holds when $\varepsilon = \binom{n}{2}$. Let G be a graph with the largest ε such that G has no perfect matching. If $uv \notin E(G)$, then because $G+uu$ has more edges than G, it must be that $G+uu$ has a perfect matching. Let S be the set of all vertices of G of degree $n-1$, and let G' be any connected component of $G-S$. If G' is not a complete graph, then it contains three vertices x, y, z such that $x \longrightarrow y \longrightarrow z$, but $x \not\longrightarrow z$. Since $y \notin S$, deg(y)<$n-1$, so there is a vertex $w \not\longrightarrow y$. Let $M1$ be a perfect matching of $G+xz$ and let $M2$ be a perfect matching of $G+yw$, as shown in Figures 7.17 and 7.18. Then $xz \in M_1$ and $yw \in M_2$. Let $H = M_1 \oplus M_2$. H consists of one or more alternating cycles in G. Let Cxz be the cycle of H containing xz and let Cyw be the cycle containing yw.

Case 1. $Cxz \neq Cyw$.

Form a new matching M by taking $M2$-edges of Cxz, $M1$-edges of Cyw, and $M1$ edges elsewhere. Then M is a perfect matching of G, a contradiction.

Case 2. $Cxz = Cyw = C$.

C can be traversed in two possible directions. Beginning with the vertices y, w, we either come to x first or z first. Suppose it is z. Form a new matching M by taking $M1$-edges between w and z, $M2$-edges between x and y, and the edge yz. Then take $M1$ edges elsewhere. Again M is a perfect matching of G, a contradiction.

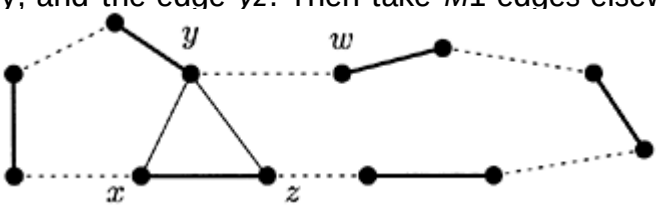


FIGURE 7.18
$H = M_1 \oplus M_2$**, case 2**

We conclude that every component G' of $G-S$ must be a complete graph. But then we can easily construct a

perfect matching of G as follows. Each even component of $G-S$ is a complete graph, so it has a perfect matching. Every odd component is also a complete graph, so is has a near perfect matching, namely, one vertex is not matched. This vertex can be matched to a vertex of S, since $\text{odd}(G-S) \leq |S|$. The remaining vertices of S form a complete subgraph, since they have degree $n-1$, so they also have a perfect matching. It follows that every G satisfying the condition of the theorem has a perfect matching.

Tutte's theorem is a powerful criterion for the existence of a perfect matching. For example, the following graph has no perfect matching, since $G-u$ has three odd components.

We can use Tutte's theorem to prove that every 3-regular graph G without cut-edges has a perfect matching. Let $S \subseteq V(G)$ be any subset of the vertices. Let $G1$, $G2$,..., Gk be the odd components of $G-S$. Let mi be the number of edges connecting Gi to S. Then $mi > 1$, since G has no cut-edge. Since $\sum_{v \in G_i} \text{DEG}(v) = 2\varepsilon(G_i) + m_i = 3|G_i|$=an odd number, we conclude that mi is odd. Therefore $mi \geq 3$, for each i. But $\sum_{v \in S} \text{DEG}(v) = 3|S| \geq \sum_i m_i$, since all of the mi edges have one endpoint in S. It follows that $3|S| \geq 3k$, or $|S| \geq \text{odd}(G-S)$, for all $S \subseteq V(G)$. Therefore G has a perfect matching M. G also has a 2-factor, since $G-M$ has degree two.

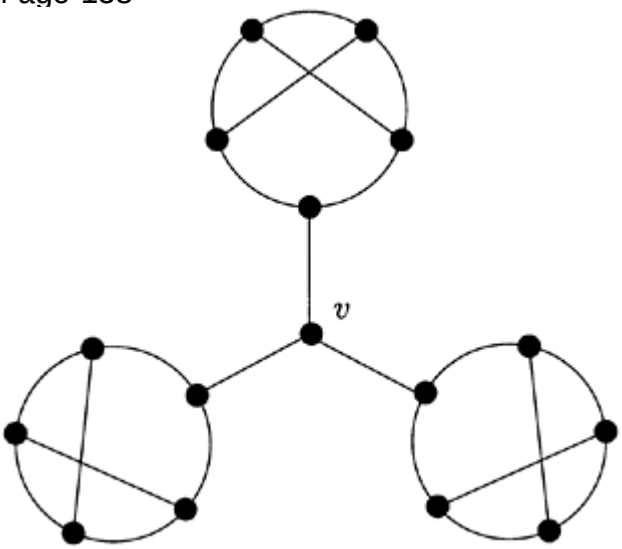


FIGURE 7.19
A 3-regular graph with no perfect matching

Exercises

7.6.1 For each integer $k > 1$, find a k-regular graph with no perfect matching.

7.6.2 A *near perfect matching* in a graph G is a matching which saturates all vertices of G but one. A *near 1-factorization* is a decomposition of $E(G)$ into near perfect matchings. Prove that K_{2n+1} has a near 1-factorization.

7.6.3 Find a condition similar to Tutte's theorem for a graph to have a near perfect matching.

7.7 Notes

Algorithms for maximum matchings in non-bipartite graphs are based on *blossoms,* discovered by EDMONDS [38]. A blossom is a generalization of an odd cycle in which all but one vertex is matched. An excellent description of Edmonds' algorithm appears in PAPADIMITRIOU and STEIGLITZ [94]. A good source book for the theory of matchings in graphs is LOVÁSZ and PLUMMER [85]. Exercise 7.1.2 is from BONDY and MURTY [19].

The proof of Tutte's theorem presented here is based on a proof by LOVÁSZ [84]. Tutte's transformation to reduce the subgraph problem to a perfect matching problem is from TUTTE [117]. His *Factor theorem,* TUTTE [118],

is a solution to the subgraph problem. It is one of the great theorems of graph theory. The theory of 1-factorizations has important applications to the theory of combinatorial designs.

Page 160
This page intentionally left blank.

Page 161
8
Network Flows
8.1 Introduction
A *network* is a directed graph used to model the distribution of goods, data, or commodities, etc., from their centers of production to their destinations. For example, Figure 8.1 shows a network in which goods are produced at node *s,* and shipped to node *t*. Each directed edge has a limited *capacity,* being the maximum number of goods that can be shipped through that channel per time period (e.g., 3 kilobytes per second or 3 truckloads per day). The diagram indicates the capacity as a positive integer associated with each edge. The actual number of goods shipped on each edge is shown in square brackets beside the capacity. This is called the *flow* on that edge. It is a non-negative integer less that or equal to the capacity. Goods cannot accumulate at any node; therefore the total in-flow at each node must equal the out-flow at that node. The problem is to find the distribution of goods that maximizes the net flow from *s* to *t*.
This can be modeled mathematically as follows. When the edges of a graph have a direction, the graph is called a *directed graph* or *digraph*. A *network N* is a directed graph with two special nodes *s* and *t; s* is called the *source* and *t* is called the *target*. All other vertices are called intermediate vertices. The edges of a directed graph are ordered pairs *(u,u)* of vertices, which we denote by $\overrightarrow{uv}$. We shall find it convenient to say that *u* is adjacent to *u* even when we do not know the direction of the edge. So the phrase u is adjacent to u means either $\overrightarrow{uv}$ or $\overrightarrow{vu}$ is an edge. Each edge $\overrightarrow{uv} \in E(N)$ has a capacity $(\overrightarrow{uv})$, being a positive integer, and a flow $f(\overrightarrow{uv})$, a non-negative integer, such that $f(\overrightarrow{uv}) \leq \text{CAP}(\overrightarrow{uv})$. If *u* is any vertex of *N,* the *out-flow* at *u* is

$$f^+(v) = \sum_{u,v \longrightarrow u} f(\overrightarrow{vu})$$

where the sum is over all vertices *u* to which *u* is joined. The *in-flow* is the sum

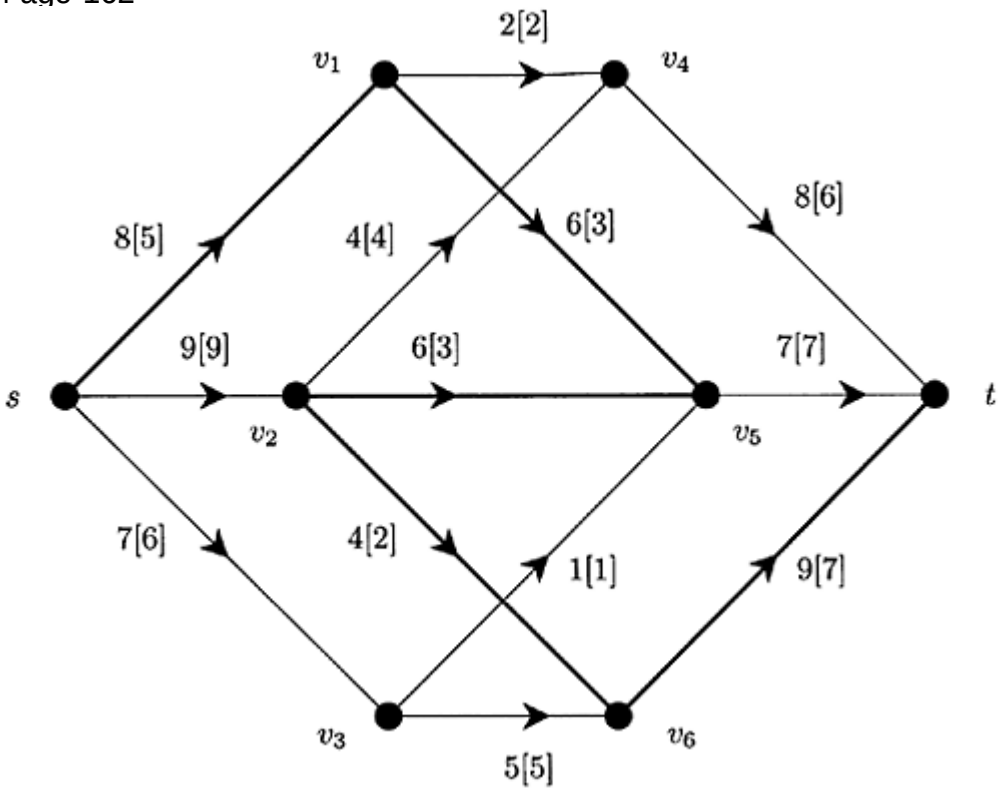
Page 162


FIGURE 8.1
A network

over all incoming edges at *u*

$$f^-(v) = \sum_{u, u \longrightarrow v} f(\overrightarrow{uv})$$

A valid flow *f* must satisfy two conditions.

1. Capacity constraint: $0 \leq f(\overrightarrow{uv}) \leq \text{CAP}(\overrightarrow{uv})$, for all $\overrightarrow{uv} \in E(N)$.
2. Conservation condition: $f^+(u) = f^-(u)$, for all $u \neq s,t$.
Notice that in Figure 8.1 both these conditions are satisfied. The value of the flow is the net out-flow at *s;* in this case, VAL(f)=20.
In general, there may be in-edges as well as out-edges at *s*. The net flow from *s* to *t* will then be the out-flow at the source minus the in-flow. This is called the *value* of the flow, VAL$(f) = f^+(s) - f^-(s)$. The max-flow problem is:

Page 163

Problem 8.1: Max-Flow

Instance: a network *N*.
Find: a flow *f* for *N* of maximum value.

Any flow *f* that has maximum value for the network *N* is called a *max-flow* of *N*. This problem was first formulated and solved by Ford and Fulkerson. In this chapter we shall present the Ford-Fulkerson algorithm, and study several applications of the max-flow-min-cut theorem.
It is possible that a network encountered in practice will have more than one source or target. If *s*1, *s*2,..., *sk* are all sources in a network *N,* and *t*1, *t*2,..., *tm* are all targets, we can replace *N* with a network *N′* with only one source and one target as follows. Add a vertex *s* to *N,* and join it to *s*1, *s*2,..., *s*k. Add a vertex *t* and join *t*1, *t*2,..., *tm* to *t*. Assign a capacity $\text{CAP}(\overrightarrow{ss_i})$ being the sum of the capacities of the out-edges at *si,* and a capacity $\text{CAP}(\overrightarrow{t_i t})$, being the sum of the capacities of all incoming edges to *ti*. Call the resulting network *N′*. For every flow in *N* there is a corresponding flow in *N′* with equal value, and vice-versa. Henceforth we shall always assume that all networks have just one source and target. The model we are using assumes that edges are one-way channels and that goods can only be shipped in the direction of the arrow. If a two-way channel from *u* to *u* is desired, this can easily be accommodated by two directed edges $\overrightarrow{uv}$ and $\overrightarrow{vu}$.
Let $S \subseteq V(N)$ be a subset of the vertices such that $s \in S, t \notin S$. Write $\overline{S} = V(N) - S$. Then $[S, \overline{S}]$ denotes the set of all edges of *N* directed from *S* to $\overline{S}$. See Figure 8.2. Consider the sum

$$\sum_{v \in S} (f^+(v) - f^-(v)). \tag{8.1}$$

Since $f^+(u) = f^-(u)$, if $u \neq s$, this sum equals VAL(f). On the other hand, $f^+(u)$ is the total out-flow at $v \in S$. Consider an out-edge $\overrightarrow{vu}$ at *u*. Its flow $f(\overrightarrow{vu})$ contributes to $f^+(u)$. It also contributes to $f^-(u)$. $u \in S$, then $f(\overrightarrow{vu})$ will appear twice in the sum 8.1, once for $f^+(u)$ and once for $f^-(u)$, and will therefore cancel. See Figure 8.2, where *S* is the set of shaded vertices. If $u \notin S$, then $f(\overrightarrow{vu})$ will appear in the summation as part of $f^+(u)$, but will not be canceled by $f^-(u)$. A similar argument holds if $v \in \overline{S}$ and $u \in S$. Therefore

$$\begin{aligned} \text{VAL}(f) &= \sum_{v \in S} (f^+(v) - f^-(v)) \\ &= \sum_{\overrightarrow{vu} \in [S, \overline{S}]} f(\overrightarrow{vu}) - \sum_{\overrightarrow{vu} \in [\overline{S}, S]} f(\overrightarrow{vu}) \end{aligned}$$

Page 164

This says that the value of the flow can be measured across any edge-cut $[S, \overline{S}]$, such that $s \in S$ and $t \in \overline{S}$. If we write

$$f^+(S) = \sum_{v \in S} f^+(v)$$

and

$$f^-(S) = \sum_{v \in S} f^-(v)$$

then

$$\text{VAL}(f) = f^+(S) - f^-(S).$$

If we write

$$f([S, \overline{S}]) = \sum_{\overrightarrow{vu} \in [S, \overline{S}]} f(\overrightarrow{vu})$$

and

$$f([\overline{S}, S]) = \sum_{\overrightarrow{vu} \in [\overline{S}, S]} f(\overrightarrow{vu})$$

then we can also express this as

$$\text{VAL}(f) = f([S, \overline{S}]) - f([\overline{S}, S]).$$

Let $K = [S, \overline{S}]$ be any edge-cut with $s \in S$ and $t \in \overline{S}$. The *capacity* of *K* is

$$\text{CAP}(K) = \sum_{\overrightarrow{uv} \in K} \text{CAP}(\overrightarrow{uv}).$$

This is the sum of the capacities of all edges out of *S*. The value of any flow in *N* is limited by the capacity of any edge-cut *K*. An edge-cut *K* is a *min-cut* if it has the minimum possible capacity of all edge-cuts in *N*.

LEMMA 8.1 Let $K = [S, \overline{S}]$ *be an edge-cut in a network N with* $s \in S$, $t \in \overline{S}$ *and flow f. Then* VAL*(f)*≤CAP*(K)*. *If* VAL*(f)*=CAP*(K), then f is a max-flow and K is a min-cut*.

PROOF Clearly the maximum possible flow out of *S* is bounded by CAP*(K);* that is, $f^+(S) \leq \text{CAP}(K)$. This holds even if *K* is a min-cut or *f* a max-flow. The flow into *S* is non-negative; that is, $f^-(S) \geq 0$. Therefore $\text{VAL}(f) = f^+(S) - f^-(S) \leq \text{CAP}(K)$. If VAL*(f)*=CAP*(K),* then it must be that *f* is maximum, for the value of no flow can exceed the capacity of any cut. Similarly *K* must be a min-cut. Note that in this situation $f^+(S) = \text{CAP}(K)$ and $f^-(S) = 0$. That is, every edge $\overrightarrow{uv}$ directed out of *S* satisfies $f(\overrightarrow{uv}) = \text{CAP}(\overrightarrow{uv})$. Every edge $\overrightarrow{uv}$ into *S* carries no flow, $f(\overrightarrow{uv}) = 0$.

Page 165

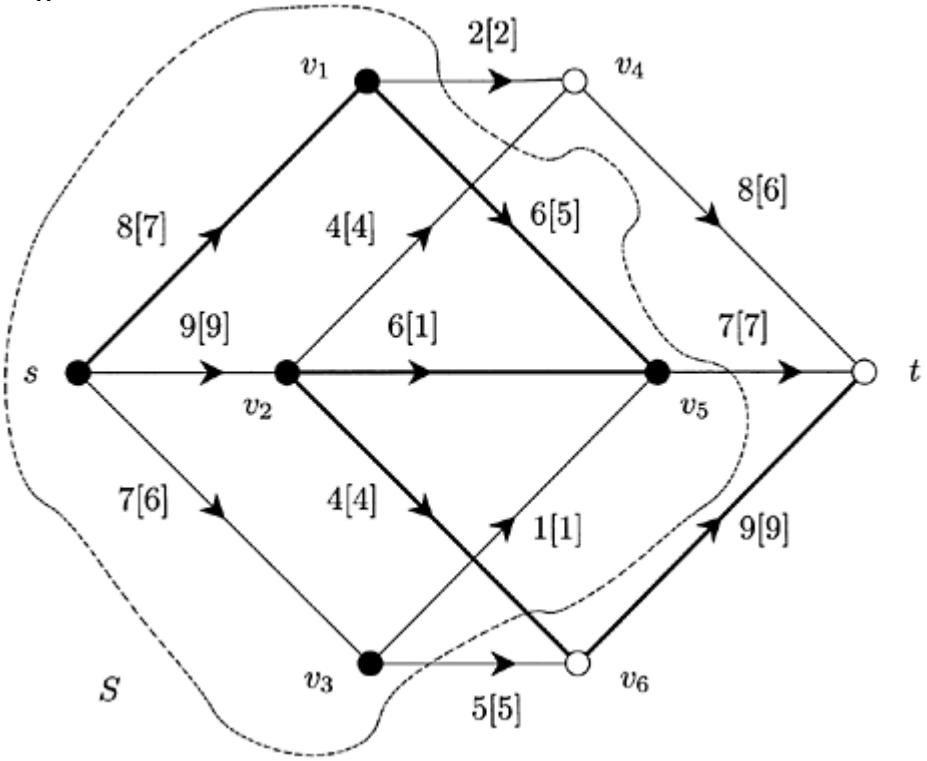


FIGURE 8.2

$s \in S, t \in \overline{S}$

A set *S* where
In the next section we shall prove the max-flow-min-cut theorem. This states that the value of a max-flow and the capacity of a min-cut are always equal, for any network *N*.

8.2 The Ford-Fulkerson algorithm

If we assign $f(\overrightarrow{uv}) = 0$, for all $\overrightarrow{uv} \in E(N)$, this defines a valid flow in *N,* the *zero flow*. The Ford-Fulkerson algorithm begins with the zero flow, and increments it through a number of iterations until a max-flow is obtained. The method uses augmenting paths. Consider the *st*-path *P*=*su*1*u*5*u*2*u*6*t* in Figure 8.1. (We ignore the direction of the edges when considering these paths.) Each edge of *P* carries a certain amount of flow. The traversal of *P* from *s* to *t* associates a direction with *P*. We can then distinguish two kinds of edges of *P, forward edges,* those like *su*1 whose direction is the same as that of *P,* and *backward edges,* those like *u*5*u*2 whose direction is opposite to that of *P*. Consider a forward

edge $\overrightarrow{uv}$ in an *st*-path *P*. If $f(\overrightarrow{uv}) < \text{CAP}(\overrightarrow{uv})$, then $\overrightarrow{uv}$ can carry more flow. Define the *residual capacity* of $uv \in E(P)$ to be

$$\text{RESCAP}(uv) = \begin{cases} \text{CAP}(\overrightarrow{uv}) - f(\overrightarrow{uv}), & \text{if } uv \text{ is a forward edge,} \\ f(\overrightarrow{vu}), & \text{if } uv \text{ is a backward edge.} \end{cases}$$

The residual capacity of a forward edge $uv \in E(P)$ is the maximum amount by which the flow on $\overrightarrow{uv}$ can be increased. The residual capacity of a backward edge $uv \in E(P)$ is the maximum amount by which the flow on $\overrightarrow{vu}$ can be decreased. For example, in the network of Figure 8.1, we increase the flow on all forward edges of *P* by 2, and decrease the flow on all backward edges of *P* also by 2. The result is shown in Figure 8.2. We have a new flow, with a larger value than in Figure 8.1.
In general, let *P* be an *st*-path in a network *N* with flow *f*. Define the *residual capacity* of *P* to be

$$\delta(P) = \text{MIN}\{\text{RESCAP}(\overrightarrow{uv}) : uv \in E(P)\}$$

Define a new flow *f** in *N* as follows:

$$f^*(\overrightarrow{uv}) = \begin{cases} f(\overrightarrow{uv}), & \text{if } uv \text{ is not an edge of } P, \\ f(\overrightarrow{uv}) + \delta(P), & \text{if } uv \text{ is a forward edge of } P \text{ and} \\ f(\overrightarrow{uv}) - \delta(P), & \text{if } uv \text{ is a backward edge of } P. \end{cases}$$

*LEMMA 8.2 f** is *a valid flow in N and* VAL*(f*)*=VAL*(f)*+*δ(P)*.
PROOF We must check that the capacity constraint and conservation conditions are both satisfied by *f**. It is clear that the capacity constraint is satisfied, because of the definition of the residual capacity of *P* as the *minimum* residual capacity of all edges in *P*. To verify the conservation condition, consider any intermediate vertex *u* of *P*. Let its adjacent vertices on *P* be *u* and *w,* so that *uu* and *uw* are consecutive edges of *P*. There are four cases, shown in Figure 8.3.
Case 1. *uu* and *uw* are both forward edges of *P*.
Because $f(\overrightarrow{uv})$ and $f(\overrightarrow{vw})$ both increase by *δ(P)* in *f**, it follows that *f+(u)* and *f−(u)* both increase *δ(P)*. The net result on *f+(u)−f−(u)* is zero.
Case 2. *uu* is a forward edge and *uw* is a backward edge.
In this case $f(\overrightarrow{uv})$ increases and $f(\overrightarrow{wv})$ decreases by *δ(P)* in *f**. It follows that *f+(u)* and *f−(u)* are both unchanged.
Case 3. *uu* is a backward edge and *uw* is a forward edge.
In this case $f(\overrightarrow{vu})$ decreases and $f(\overrightarrow{vw})$ increases by *δ(P)* in *f**. It follows that *f+(u)* and *f−(u)* are both unchanged.

Case 4. *uu* and *uw* are both backward edges of *P*.
Because $f(\overrightarrow{vu})$ *and* $f(\overrightarrow{wv})$ both decrease by *δ(P)* in *f**, it follows that *f+(u)* and *f−(u)* both decrease by *δ(P)*. The net result on *f+(u)−f−(u)* is zero.
The value of *f** is *f*+(s)−f*−(s)*. If the first edge of *P* is *su,* a forward edge, then it is clear that the value

increases by $\delta(P)$, since $f(\overrightarrow{su})$ increases. If su is a backward edge, then $f(\overrightarrow{su})$ decreases, so that $f^-(s)$ also decreases, thereby increasing the value of the flow. Therefore $\text{VAL}(f^*)=\text{VAL}(f)+\delta(P)$.

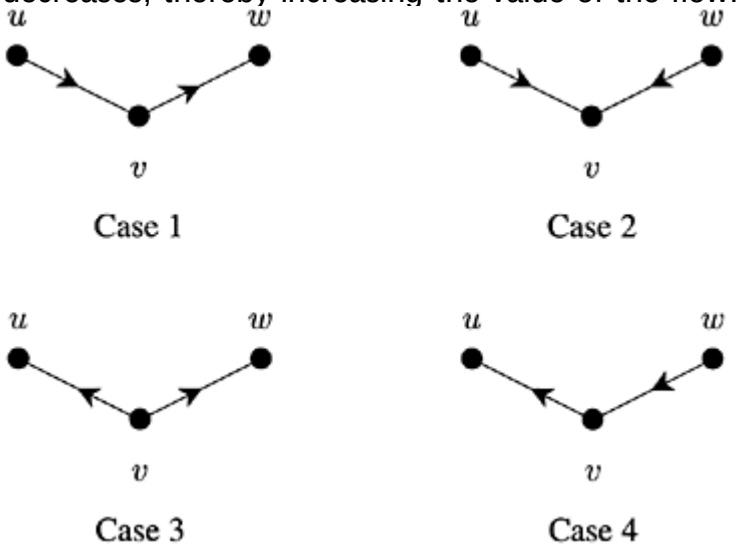


FIGURE 8.3
The four cases for edges *uu* and *uw* on path *P*

DEFINITION 8.1: An *st*-path *P* for which $\delta(P)>0$ is called an *augmenting path*.

This method of altering the flow on the edges of *P* is called *augmenting the flow*. If $\delta(P)>0$ it always results in a flow of larger value. We give an outline of the Ford-Fulkerson algorithm in Algorithm 8.2.1.

Algorithm 8.2.1: FF*(N,s,t)*

comment: N is a network with source s and target t.
f is the flow.
P is a path.

$f \leftarrow$ the zero flow
search for an augmenting path P
while a path P was found
do augment the flow on P
$\text{VAL}(f) \leftarrow \text{VAL}(f) + \delta(P)$
search for an augmenting path P
comment: the flow is now maximum

The algorithm stops when *N* does not contain an augmenting path. We show that in this situation the flow must be maximum. The outline given in Algorithm 8.2.1 does not specify how the augmenting paths are to be found. Among the possibilities are the breadth-first and depth-first searches. We shall see later that the breadth-first search is the better choice. As the algorithm searches for an augmenting path, it will construct paths from *s* to various intermediate vertices *u*. The paths must have positive residual capacity. An *su*-path with positive residual capacity is said to be *unsaturated*. A vertex *u* is *s-reachable* if *N* contains an unsaturated *su*-path. This means that *u* can be reached from *s* on an unsaturated path.

THEOREM 8.3 Let N be a network with a flow f. Then f is maximum if and only if N contains no augmenting path.

PROOF Suppose that *f* is a max-flow. There can be no augmenting path in *N,* for this would imply a flow of larger value. Conversely, suppose that *f* is a flow for which there is no augmenting path. We show that *f* is maximum. Let *S* denote the set of all *s*-reachable vertices of *N*. Clearly $s \in S$. Since there is no augmenting path, the target is not *s*-reachable. Therefore $t \in \overline{S}$. Consider the edge-cut $K = [S, \overline{S}]$. If $\overrightarrow{uv} \in K$ is an edge out of *S,* then $\text{RESCAP}(\overrightarrow{uv}) = 0$; for otherwise *u* would be *s*-reachable on the forward edge $\overrightarrow{uv}$ from $u \in S$. Therefore $f(\overrightarrow{uv}) = \text{CAP}(\overrightarrow{uv})$ for all $\overrightarrow{uv} \in K$ that are out edges of *S*. Thus $f^+(S) = \text{CAP}(K)$. If $\overrightarrow{uv} \in [\overline{S}, S]$ is any edge into *S,* then $f(\overrightarrow{uv}) = 0$; for otherwise *u* would be *s*-reachable on the backward edge $\overrightarrow{vu}$ from $v \in S$. Consequently $f^-(S)=0$. It follows that $\text{VAL}(f)=\text{CAP}(K)$, so that *f* is a max-flow and *K* a min-cut, by

Lemma 8.1.

Page 169

This is illustrated in Figure 8.2, in which all edges out of *S* are saturated. In this example there are no edges into *S*. If there were, they would carry no flow. So the flow in Figure 8.2 is maximum. Notice that a consequence of this theorem is that when *f* is a max-flow, the set *S* of *s*-reachable vertices defines a min-cut $K = [S, \overline{S}]$. This is summarized as follows:

THEOREM 8.4 **(Max-flow-min-cut theorem)** *In any network the value of a max-flow equals the capacity of a min-cut.*

We are now ready to present the Ford-Fulkerson algorithm as a breadth-first search for an augmenting path. The vertices will be stored on a queue, the *ScanQ,* an array of *s*-reachable vertices. *QSize* is the current number of vertices on the *ScanQ*. The unsaturated *su*-paths will be stored by an array *PrevPt*[·], where *PrevPt[u]* is the point previous to *u* on an *su*-path *Pu*. The residual capacity of the paths will be stored by an array *ResCap*[·], where *ResCap[u]* is the residual capacity *δ(Pu)* of *Pu* from *s* up to *u*. The algorithm is presented as a single procedure, but could be divided into smaller procedures for modularity and readability.

Page 170

Algorithm 8.2.2: MAXFLOW*(N,s,t)*
f←the zero flow
for all vertices *u* **do** *PrevPt[u]*←0
while true “search for an augmenting path”

```
do { ScanQ[1] ← s
     QSize ← 1
     k ← 1
     repeat
       u ← ScanQ[k]        "the k^th vertex on ScanQ"
       for all v adjacent to u
         do if v ∉ ScanQ
              then { if u ⟶ v
                       then { comment: a forward edge
                              if CAP(uv⃗) > f(uv⃗) then
                                { QSize ← QSize + 1
                                  ScanQ[QSize] ← v
                                  PrevPt[v] ← u
                                  if ResCap[u] < CAP(uv⃗) − f(uv⃗)
                                    then ResCap[v] ← ResCap[u]
                                    else ResCap[v] ← CAP(uv⃗) − f(uv⃗)
                                  if v = t then go to 1 }
                            }
                     else { comment: v ⟶ u, a backward edge
                            if f(vu⃗) > 0 then
                              { QSize ← QSize + 1
                                ScanQ[QSize] ← v
                                PrevPt[v] ← u
                                if ResCap[u] < f(vu⃗)
                                  then ResCap[v] ← ResCap[u]
                                  else ResCap[v] ← f(vu⃗)
                                if v = t then go to 1 }
                          }
       k ← k + 1        "advance ScanQ"
     until k > QSize
     comment: { Flow is now maximum.
                ScanQ contains the s-reachable vertices.
     output (ScanQ, and the flow on each edge)
     exit
     1 : comment: augmenting path found, re-initialize ScanQ
     AUGMENTFLOW(t)
     for k ← 1 to QSize do PrevPt[ScanQ[k]] ← 0 }
```


Page 171

The procedure which augments the flow starts at *t* and follows *PrevPt[u]* up to *s*. Given an edge $\overrightarrow{uv}$ on the augmenting path, where *u=PrevPt[u],* a means is needed of determining whether $\overrightarrow{uv}$ is a forward or backward edge. One way is to store *PrevPt[u]=u* for forward edges and *PrevPt[u]=−u* for backward edges. This is not indicated in Algorithm 8.2.2, but can easily be implemented.

Algorithm 8.2.3: AUGMENTFLOW*(t)*

u←*t*
u←*PrevPt[u]*
δ←*ResCap[t]*
while *u*≠**0**

$$\textbf{do}\ \begin{cases} \textbf{if}\ u \longrightarrow v \\ \quad \textbf{then}\ f(\overrightarrow{uv}) \leftarrow f(\overrightarrow{uv}) + \delta \qquad \text{“a forward edge”} \\ \quad \textbf{else}\ f(\overrightarrow{uv}) \leftarrow f(\overrightarrow{uv}) - \delta \qquad \text{“a backward edge”} \\ v \leftarrow u \\ u \leftarrow PrevPt[v] \end{cases}$$

VAL*(f)*←VAL*(f)*+δ

In programming the max-flow algorithm, the network *N* should be stored in adjacency lists. This allows the loop

for all *u* adjacent to *u* **do**

to be programmed efficiently. The out-edges and in-edges at *u* should all be stored in the same list. We need to be able to distinguish whether $u \longrightarrow v$ or $v \longrightarrow u$. This can be flagged in the record representing edge $\overrightarrow{uv}$. If $\overrightarrow{uv}$ appears as an out-edge in the list for node *u,* it will appear as an in-edge in the list for vertex *u*. When the flow on edge $\overrightarrow{uv}$ is augmented, it must be augmented from both endpoints. One way to augment from both endpoints simultaneously is to store not the flow $f(\overrightarrow{uv})$ itself, but a pointer to it. Then it is not necessary to find the other endpoint of the edge. Thus a node *x* in the adjacency list for vertex *u* contains following four fields:

- $AdjPt\langle x\rangle$, a vertex *u* that is adjacent to or from *u*.
- $OutEdge\langle x\rangle$, a boolean va**ri**able set to **true** if $u \longrightarrow v$, and **false** if $v \longrightarrow u$.
- $Flow\langle x\rangle$, a *pointer* to the flow on $\overrightarrow{uv}$.
- $Next\langle x\rangle$, the next node in the adjacency list for *u*.

This breadth-first search version of the Ford-Fulkerson algorithm is sometimes referred to as the “labeling” algorithm in some books. The values *ResCap[v]* and *PrevPt[u]* are considered the labels of vertex *u*.

The algorithm works by constructing all shortest unsaturated paths from *s*. If an augmenting path exists, it is sure to be found. This can easily be proved by induction on the length of the shortest augmenting path. The flow is then augmented and the algorithm exits from the inner repeat loop by branching to statement 1. If no augmenting path exists, then the inner repeat loop will terminate. The vertices on the *ScanQ* will contain the set *S* of all *s*-reachable vertices, such that $[S, \overline{S}]$ is a min-cut.

It is difficult to form an accurate estimate of the complexity of the BF-FF algorithm. We shall prove that it is polynomial. This depends on the fact that only *shortest* augmenting paths are used. If non-shortest paths are used, the FF algorithm is not always polynomial. Consider the network of Figure 8.4. We augment first on path *P=(s, a, b, t),* which has residual capacity one. We then augment on path *Q=(s, b, a, t),* also of residual capacity one, since *ba* is a backward edge, and $f(\overrightarrow{ab}) = 1.$ Augmenting on *Q* makes *δ(P)=1,* so we again augment on *P,* and then augment again on *Q,* etc. After 2000 iterations a max-flow is achieved—*the number of iterations can depend on the value of the max-flow*. This is not polynomial in the parameters of the network. However, if shortest augmenting paths are used, this problem does not occur.

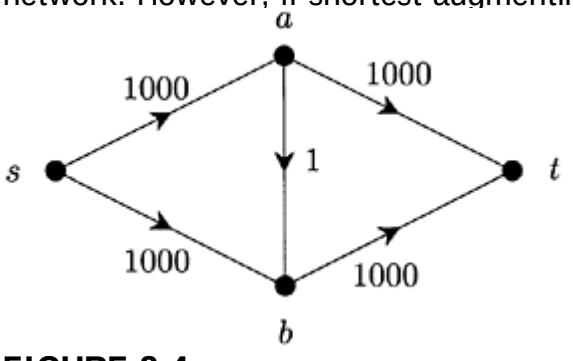


FIGURE 8.4
A max-flow in 2000 iterations

Consider an augmenting path *P* in a network *N*. *δ(P)* is the minimum residual capacity of all edges of *P*. Any edge $\overrightarrow{uv} \in P$ such that $\text{RESCAP}(\overrightarrow{uv}) = \delta(P)$ is called a *bottleneck*. Every augmenting path has at least one bottleneck, and may have several. Suppose that a max-flow in *N* is reached in *m* iterations, and let *Pj* be the

augmenting path on iteration *j*. Let *dj(s, u)* denote the length of a shortest unsaturated su-path in iteration *j*, for all vertices *u*.

LEMMA 8.5 $dj+1(s, u) \geq dj(s, u)$, for all $u \in V(N)$.

PROOF Let $Qj=Qj(s, u)$ be a shortest unsaturated *su*-path at the beginning of

Page 173

iteration *j*, and let $Qj+1=Qj+1(s, u)$ be a shortest unsaturated *su*-path at the beginning of iteration *j*+1. Then $\ell(Qj)=dj(s, u)$ and $\ell(Qj+1)=dj+1(s, u)$. If $\ell(Qj) \leq \ell(Qj+1)$, the lemma holds for vertex *u*, so suppose that $\ell(Qj) > \ell(Qj+1)$, for some *u*. Now $Qj+1$ is not unsaturated at the beginning of iteration *j*, so it must become unsaturated during iteration *j*. Therefore *Pj* and $Qj+1$ have at least one edge in common that becomes unsaturated during iteration *j*. The proof is by induction on the number of such edges. Suppose first that *xy* is the only such edge in common. Since *xy* becomes unsaturated during iteration *j*, it has opposite direction on *Pj* and $Qj+1$. See Figure 8.5. Without loss of generality, $Qj+1[s, x]$ and $Qj+1[y, u]$ are unsaturated on iteration *j*. Since *Pj* is a shortest path, $\ell(Pj[s, x]) \leq \ell(Qj+1[s, x])$. But then $Pj[s, y]Qj+1[y, u]$ is an unsaturated *su*-path on iteration *j*, and has length less than $Qj+1(s, u)$, which in turn has length less than $Qj(s, u)$, a contradiction. If $Pj[s, y]$ intersects $Qj+1[y, u]$ at a vertex *z*, then $Pj[s, z]Qj+1[z, u]$ is an even shorter unsaturated path.

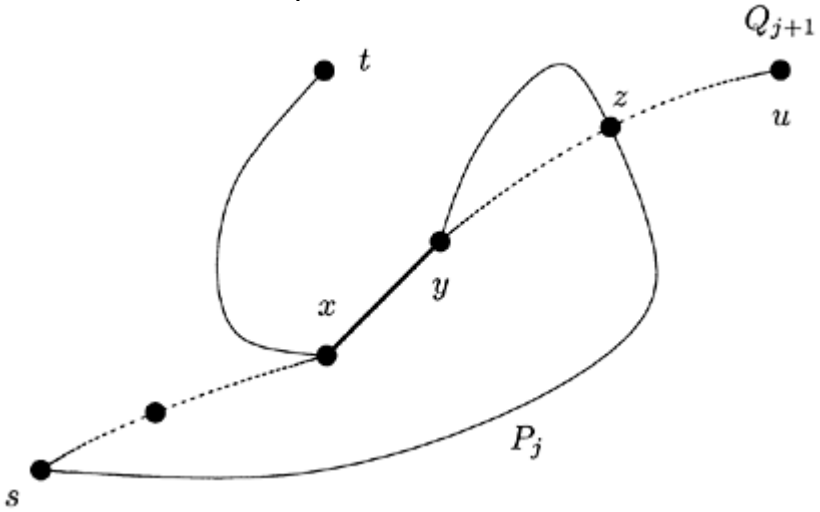


FIGURE 8.5
Paths *Pj* and *Qj*+1

Suppose now that *Pj* and $Qj+1$ have more than one edge in common that becomes unsaturated during iteration *j*. Let *xy* be the first such edge on $Qj+1$ traveling from *s* to *u*. Let *z* be the point on $Qj+1$ nearest to u that $Pj[s, x]$ contacts before reaching *x* (maybe $z=y$). Then $Qj+1[s, x]$ and $Pj[s, z]$ are unsaturated at the beginning of iteration *j*. Since *Pj* is a shortest unsaturated path, $\ell(Pj[s, z]) < \ell(Qj+1[s, x]) < \ell(Qj+1[s, z])$. Now either $Qj+1[z, u]$ is unsaturated on iteration *j*, or else it has another edge in common with *Pj*. If it is unsaturated, then $Pj[s, z]Qj+1[z, u]$ is an unsaturated *su*-path that contradicts the assumption that $dj(s, u) > dj+1(s, u)$. If there is another edge in common with *Pj*, then we can repeat this argument. Let *x′y′* be the first edge of $Qj+1[z, u]$ in

Page 174

common with *Pj*. Let *z′* be the point on $Qj+1[z, u]$ nearest to *u* that $Pj[s, x']$ contacts before reaching *x′*, etc. Proceeding in this way we eventually obtain an *su*-path that is shorter than *Qj* and unsaturated on iteration *j*, a contradiction.

It follows that $dj+1(s, t) \geq dj(s, t)$, for every iteration *j*. By constructing unsaturated paths from *t* in a backward direction we can similarly prove that $dj+1(u, t) \geq dj(u, t)$, for all vertices *u*. If we can now prove that $dj+1(s, t) > dj(s, t)$, then we can bound the number of iterations, since the maximum possible distance from *s* to *t* is $n-1$, where *n* is the number of vertices of *N*.

THEOREM 8.6 The breadth-first Ford-Fulkerson algorithm requires at most $\frac{1}{2}n\varepsilon + 1$ *iterations.*

PROOF On each iteration some edge is a bottleneck. After $\varepsilon+1$ iterations, some edge has been a bottleneck twice, since there are only ε edges. Consider an edge $\overrightarrow{uv}$ which is a bottleneck on iteration *i* and then later a bottleneck on iteration *j*. Refer to Figure 8.6.

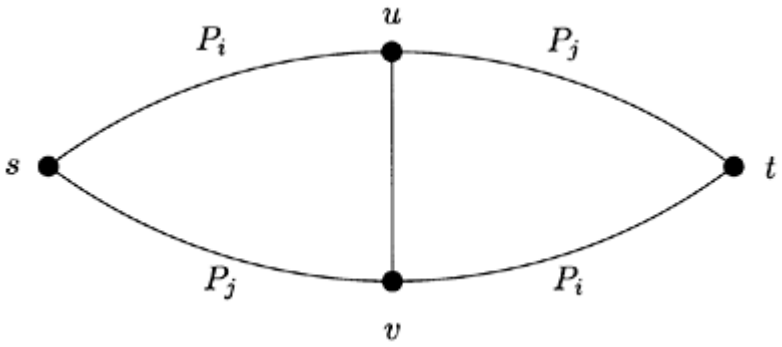


FIGURE 8.6
Paths *Pi* and *Pj*

Then $di(s, t)=di(s, u)+di(u, t)+1$ and $dj(s, t)=dj(s, u)+dj(u, t)+1$. But $di(s, u)\leq dj(s, u)=dj(s, u)+1$ and $di(u, t)\leq dj(u, t)=dj(u, t)+1$. Therefore $di(s, u)+di(u, t)\leq dj(s, u)+dj(u, t)+2$. It follows that $di(s, t)\leq dj(s, t)+2$. Each time an edge is repeated as a bottleneck, the distance from s to *t* increases by at least two. Originally $d1(s, t)\geq 1$. After $\varepsilon+1$ iterations, some edge has been a bottleneck twice. Therefore $d\varepsilon+1(s, t)\geq 3$. Similarly $d2\varepsilon+1(s, t)\geq 5$, and so on. In general $dk\varepsilon+1(s, t)\geq 2k+1$. Since the maximum distance from *s* to t is $n-1$, we have $2k+1\leq n-1$, so that $k\leq n/2$. The maximum number of iterations is then $k\varepsilon+1\leq \frac{1}{2}n\varepsilon+1$.

Page 175

Each iteration of the BF-FF algorithm is a breadth-first search for an augmenting path. A breadth-first search takes at most $O(\varepsilon)$ steps. Since the number of it-erations is at most $\frac{1}{2}n\varepsilon+1$, this gives a complexity of $O(n\varepsilon 2)$ for the breadth-first Ford-Fulkerson algorithm. This was first proved by EDMONDS and KARP [39].

Exercises

8.2.1 Find a max-flow in the network shown in Figure 8.7. Prove your flow is maximum by illustrating a min-cut *K* such that VAL(f)=CAP(K).

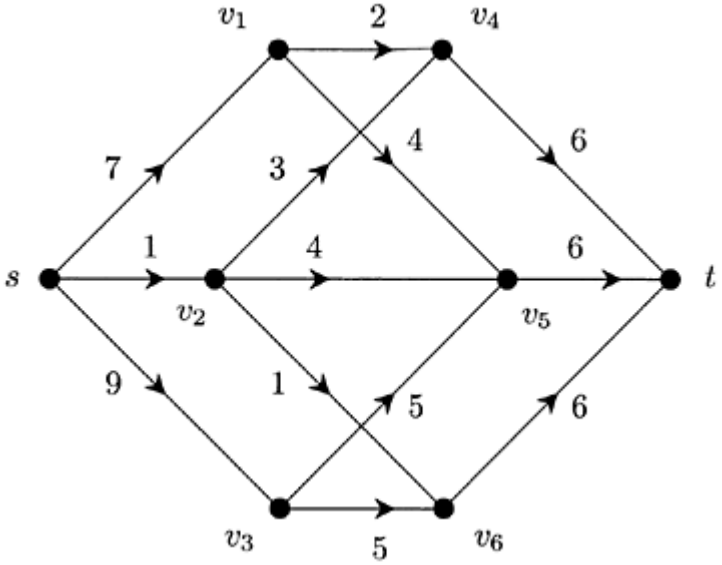


FIGURE 8.7
A network

8.2.2 Show that if there is no directed st-path in a network *N*, then the maximum flow in *N* has value zero. Can there be a flow whose value is negative? Explain.

8.2.3 Explain why $\sum_{v\in S} f^+(v)$ and $\sum_{\overrightarrow{vu}\in[S,\overline{S}]} f(\overrightarrow{vu})$ are in general, not equal.

8.2.4 Consider the network *N* of Figure 8.7 with flow *f* defined as follows: $f(su1)=6$, $f(su2)=0$, $f(su3)=2$, $f(u1u4)=2$, $f(u1u5)=4$, $f(u2u4)=0$, $f(u2u6)=0$, $f(u3u5)=2$, $f(u3u6)=0$, $f(u5u2)=0$, $f(u4t)=2$, $f(u5t)-6$, $f(u6t)=0$. A breadth-first search of *N* will construct the subnetwork of all shortest, unsaturated paths in *N*. This subnetwork is called the *auxiliary network,* Aux(N, f). A forward edge $\overrightarrow{uv}$ of *N* is replaced by a forward edge $\overrightarrow{uv}$ with capacity $\text{CAP}(\overrightarrow{uv})-f(\overrightarrow{uv})$ in the auxiliary network. A backward edge $\overrightarrow{vu}$ of *N* is replaced by a forward edge $\overrightarrow{uv}$ with capacity $f(\overrightarrow{uv})$ in Aux(N,f). Initially the flow in AUX(N,f) is the zero flow. Construct the auxiliary network for the graph shown. Find a max-flow in

Page 176
Aux*(N,f),* and modify *f* in *N* accordingly. Finally, construct the new auxiliary network for *N*.
8.2.5 Program the breadth-first Ford-Fulkerson algorithm. Test it on the networks of this chapter.
8.2.6 If $[S, \overline{S}]$ and $[T, \overline{T}]$ are min-cuts in a network *N,* show that $[S \cup T, \overline{S \cup T}]$ and $[S \cap T, \overline{S \cap T}]$ are also min-cuts. (*Hint:* Write $S = S_1 \cup (S \cap T)$ and $T = T_1 \cup (S \cap T)$ and use the fact that $[S, \overline{S}]$ and $[T, \overline{T}]$ are both min-cuts.)
8.2.7 Describe a maximum flow algorithm similar to the Ford-Fulkerson algorithm which begins at *t* and constructs unsaturated paths *P* until *s* is reached. Given that *P* is a *ts*-path, how should the residual capacity of an edge be defined in this case?
8.2.8 Describe an algorithm for finding an edge $\overrightarrow{uv}$ in a network *N* such that the value of a max-flow *f* in *N* can be increased if $(\overrightarrow{uv})$ is increased. Prove that your algorithm is correct and find its complexity. Does there always exist such an edge $\overrightarrow{uv}$? Explain.

8.3 Matchings and flows

There is a marked similarity between matching theory and flow theory:
Matchings: A matching *M* in a graph *G* is maximum if and only if *G* contains no augmenting path.
Flows: A flow *f* in a network *N* is maximum if and only if *N* contains no augmenting path.
Hungarian algorithm: Construct alternating paths until an augmenting path is found.
Ford-Fulkerson algorithm: Construct unsaturated paths until an augmenting path is found.
The reason for this is that matching problems can be transformed into flow problems. Consider a bipartite graph *G,* with bipartition *(X,Y)* for which a max-matching is desired. Direct all the edges of *G* from *X* to *Y,* and assign them a capacity of one. Add a source *s* and an edge *sx* for all $x \in X$, with $\text{CAP}(\overrightarrow{sx}) = 1.$ Add a target *t* and an edge *yt* for all $y \in Y$, with $\text{CAP}(\overrightarrow{yt}) = 1.$ Call the resulting network *N*. This is illustrated in Figure 8.8. Now find a max-flow in *N*. The flow-carrying edges of *[X,Y]* will determine a max-matching in *G*. Because $(\overrightarrow{sx}) = 1,$ there will be at most one flow-carrying edge out of each $x \in X$. Since $\text{CAP}(\overrightarrow{yt}) = 1,$ there will be at most one flow-carrying edge into *y,* for each $y \in Y$. The flow-carrying edges of *N* are called the *support of the flow*. An alternating path in *G* and an unsaturated path in *N* can be seen to be the same thing. If *G* is not bipartite there is no longer a direct correspondence between matchings and

Page 177
flows. However, it is possible to construct a special kind of *balanced network* such that a maximum balanced flow corresponds to a max-matching (see KOCAY and STONE [80] or FREMUTH-PAEGER and JUNGNICKEL [45]).

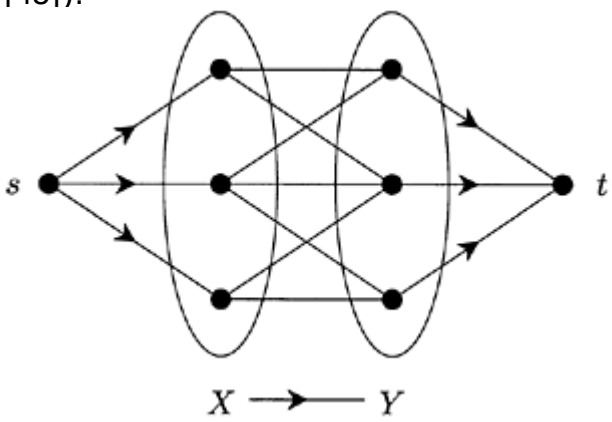


FIGURE 8.8
Matchings and flows

The basic BF-FF algorithm can be improved substantially. As it is presented here, it constructs a breadth-first network of all shortest unsaturated paths until *t* is reached. At this point, *f* is augmented, and the process is repeated. There may be many augmenting paths available at the point when *t* is reached, but only one augmenting path is used. The remaining unsaturated paths which have been built are discarded, and a new BFS is executed. In order to improve the BF-FF algorithm, one possibility is to construct the set of *all* shortest unsaturated paths. This is the auxiliary network of Exercise 8.2.4. We then augment on as many paths as

possible in the auxiliary network before executing a new BFS. This has the effect of making $dj+1(s,t) > dj(s,t)$ so that the number of iterations is at most n. Several algorithms are based on this strategy. They improve the complexity of the algorithm markedly. See the book by PAPADIMITRIOU and STEIGLITZ [94] for further information.

Exercises

8.3.1 Let G be a bipartite graph with bipartition (X,Y). We want to find a subgraph H of G such that in H, $\mathrm{DEG}(x)=b(x)$ and $\mathrm{DEG}(y)=b(y)$, where $b(u)$ is a given non-negative integer, for all $v \in V(G)$, if there exists such an H. For example, if $b(u)=1$ for all u, then H would be a perfect matching. If $b(u)=2$ for all u, then H would be a 2-factor. Show how to construct a network N such that a max-flow

in N solves this problem.

8.3.2 Let N be a network such that every vertex $v \in N$ has a maximum throughput $t(u)$ defined. This is the maximum amount of flow that is allowed to pass through u, that is, $f^-(u) \le t(u)$ must hold at all times. Show how to solve this problem by constructing a network N' such that a max-flow in N' defines a max-flow in N with maximum throughput as given.

8.4 Menger's theorems

Given any digraph, we can view it as a network N by assigning unit capacities to all edges. Given any two vertices $s, t \in V(N)$, we can compute a max-flow f from s to t. If $\mathrm{VAL}(f)=0$, then there are no directed paths from s to t, since a directed st-path would be an augmenting path. If $\mathrm{VAL}(f)=1$, then N contains a directed st-path P; however, there are no directed st-paths which are edgedisjoint from P, for such a path would be an augmenting path. In general, the value of a max-flow f in N is the maximum number of edge-disjoint directed st-paths in N. Suppose that $\mathrm{VAL}(f)=k \ge 1$. The support of f defines a subgraph of N that contains at least one directed st-path P. Delete the edges of P to get N' and let f' be obtained from f by ignoring the edges of P. Then $\mathrm{VAL}(f')=k-1$, and this must be a max-flow in N', since f is a max-flow in N. By induction, the number of edge-disjoint directed st-paths in N' is $k-1$, from which it follows that the number in N is k.

A min-cut in N can also be interpreted as a special subgraph of N. Let $K=[S,\overline{S}]$ be a min-cut in N, where $s \in S$ and $t \in \overline{S}$. If $\mathrm{CAP}(K)=0$, there are no edges out of S, so there are no directed st-paths in N. If $\mathrm{CAP}(K)=1$, there is only one edge out of S. The deletion of this edge will destroy all directed st-paths in N. We say that s is disconnected from t. In general, $\mathrm{CAP}(K)$ equals the minimum number of edges whose deletion destroys all directed st-paths in N. Suppose that $\mathrm{CAP}(K)=k \ge 1$. Delete the edges of K to get a network N'. Then in N', $\mathrm{CAP}([S,\overline{S}])=0$, so that N' contains no directed st-paths. Thus the deletion of the edges of K from N destroys all directed st-paths. Since N contains k edge-disjoint such paths, it is not possible to delete fewer than k edges in order to disconnect s from t. The max-flow-min-cut theorem now gives the first of Menger's theorems.

THEOREM 8.7 Let s and t be vertices of a directed graph N. Then the maximum number of edge-disjoint directed st-paths equals the minimum number of edges whose deletion disconnects s from t.

Recall that an undirected graph G is k-edge-connected if the deletion of fewer

than k edges will not disconnect G. In Chapter 6 we showed that a graph is 2-edge-connected if and only if every pair of vertices is connected by at least two edge-disjoint paths. We will use Theorem 8.7 to prove a similar result for k-edge-connected graphs. In order to convert Theorem 8.7 to undirected graphs, we can replace each edge $\overrightarrow{uv}$ of G by a "gadget", as shown in Figure 8.9, to get a directed graph N.

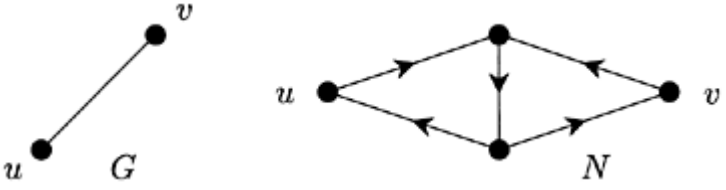


FIGURE 8.9
A gadget for edge-disjoint paths

The gadget contains a directed $\overrightarrow{uv}$-path and a directed $\overrightarrow{vu}$-path, but they both use the central edge of the gadget. Let $s, t \in V(G)$. Then edge-disjoint st-paths of G will define edge-disjoint directed st-paths in N. Conversely, edge-disjoint directed st-paths in N will define edge-disjoint st-paths in G. This gives another of

Menger's theorems.
THEOREM 8.8 Let s and t be vertices of an undirected graph G. Then the maximum number of edge-disjoint st-paths equals the minimum number of edges whose deletion disconnects s from t.
It follows that a graph *G* is *k*-edge-connected if and only if every pair *s, t* of vertices are connected by at least *k* edge-disjoint paths. This immediately gives an algorithm to compute *κ′(G),* the edge-connectivity of *G*. Number the vertices of *G* from 1 to n. Let the corresponding vertices of *N* also be numbered from 1 to *n*. The algorithm computes the minimum max-flow over all pairs *s, t* of vertices. This is the minimum number of edges whose deletion will disconnect *G*. Exactly $\binom{n}{2}$ max-flows are computed, so the algorithm has polynomial complexity.

Page 180
Algorithm 8.4.1: EDGE-CONNECTIVITY*(G)*

```
convert G to a directed graph N
κ′←n
for s←1 to n−1
do { for t ← s + 1 to n
     do { M ← MaxFlow(N, s, t)
          if M < κ′
            then κ′ ← M
return (κ′)
```

Exercises
8.4.1 Let *G* be an undirected graph. Replace each edge *uu* of *G* with a pair of directed edges $\overrightarrow{uv}$ and $\overrightarrow{vu}$ to get a directed graph *N*. Let *s,* $t \in V(G)$. Show that the maximum number of edge-disjoint *st*-paths in *G* equals the maximum number of edge-disjoint directed *st*-paths in *N.*
8.4.2 Program the edge-connectivity algorithm, using the transformation of Exercise 8.4.1.
8.5 Disjoint paths and separating sets
Recall that paths in a graph *G* are internally disjoint if they can intersect only at their endpoints. A graph is *k*-connected if the deletion of fewer than *k* vertices will not disconnect it. We proved in Chapter 6 that *G* is 2-connected if and only if every pair of vertices is connected by at least two internally disjoint paths. We prove a similar result for *k*-connected graphs by utilizing a relation between internally disjoint paths in *G* and directed paths in a network *N*. We first make two copies *u*1, *u*2 of each vertex *u* of *G*. $V(N) = \{u_1, u_2 \mid u \in V(G)\}$. Let $\overrightarrow{uv}$ be an edge of *G*. *N* will contain the edges (*u*1, *u*2), (*u*1, *u*2), (*u*2, *u*1), and (*u*2, *u*1). This is illustrated in Figure 8.10. Let $u \in V(G)$. Notice the following observations:
1. The only out-edge at *u*1 is *u*1*u*2.
2. The only in-edge at *u*2 is *u*1*u*2.
3. The edge $\overrightarrow{uv} \in E(G)$ corresponds to *u*2*u*1 and *u*2*u*1 in *N*.

Page 181

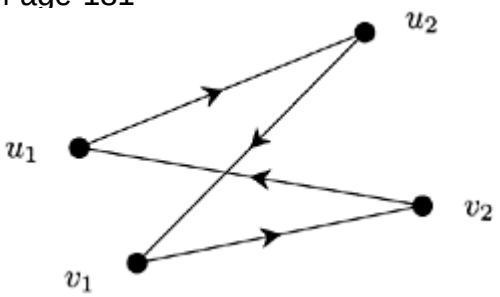


FIGURE 8.10
A gadget for internally disjoint paths
Consequently any *st*-path *suuw…t* in *G* corresponds to an *s*2*t*1-path *s*2*u*1*u*2*u*1*u*2*w*1*w*2…*t*1 in *N*. Internally disjoint *st*-paths in *G* give rise to internally disjoint *s*2*t*1-paths in *N*. On the other hand, edge-disjoint paths in *N* are in fact internally disjoint because of items 1 and 2. Therefore the maximum number of internally disjoint *st*-paths in *G* equals the maximum number of edge-disjoint directed *s*2*t*1-paths in *N*. This in turn equals the

minimum number of edges whose deletion will disconnect *s*2 from *t*1. If $s_2 \not\longrightarrow t_1$, then every *s*2*t*1-path in *G* will contain another vertex, say *u*1 or *u*2. By observations 1 and 2, it must contain both *u*1 and *u*2. Deleting *u*1*u*2 will destroy this path. If $K = [S, \overline{S}]$ is a min-cut in *N,* then the edges out of *S* must be of the form *u*1*u*2, since *u*2 can only be *s*2-reachable if *u*1 is, by observation 3 and Figure 8.10. Let $U = \{u \mid u_1u_2 \in K\}$. Then *U* is a set of vertices of *G* which separate *s* from *t*. This gives another of Menger's theorems.

THEOREM 8.9 Let G be a graph and let s, $t \in V(G)$, *where* $s \not\longrightarrow t$. *Then the maximum number of internally disjoint st-paths in G equals the minimum number of vertices whose deletion separates s from t.*
THEOREM 8.10 A graph G is k-connected if and only if every pair of vertices is connected by at least k internally disjoint paths.

PROOF Let *s,* $t \in V(G)$. If *s* and *t* are connected by at least *k* internally disjoint paths, then clearly *G* is *k*-connected; for at least *k* vertices must be deleted to disconnect *s* from *t*. Conversely suppose that *G* is *k*-connected. Then deleting fewer than *k* vertices will not disconnect *G*. If $s \not\longrightarrow t$, then by the Theorem 8.10, *G* must contain at least *k* internally disjoint *st*-paths. If $s \longrightarrow t$, then consider *G*−*st*. It is easy to see that *G*−*st* is (*k*−1)-connected. Therefore in *G*−*st,* there are at least *k*−1 internally disjoint *st*-paths. The edge *st* is another *st*-path, giving *k* paths in total.

We can also use this theorem to devise an algorithm which computes *κ(G),* the connectivity of *G*. We suppose that the vertices of *G* are numbered 1 to *n,* and that if $s \in V(G)$, then *s*1 and *s*2 are the corresponding vertices of *N*.

Algorithm 8.5.1: CONNECTIVITY*(G)*
convert *G* to a directed graph *N*
κ←*n*−1 "maximum possible connectivity"
s←0
while *s*<*κ*

do
- $s \leftarrow s + 1$ "vertex s"
- **for** $t \leftarrow s + 1$ **to** n
  - **do if** $s \not\longrightarrow t$
    - **then**
      - $M \leftarrow \text{MAXFLOW}(N, s_2, t_1)$
      - **if** $M < \kappa$
        - **then** $\kappa \leftarrow M$
      - **if** $s > \kappa$
        - **then return** (κ)

return *(κ)*

THEOREM 8.11 Algorithm 8.5.1 computes κ(G)

PROOF Suppose first that *G*=*Kn*. Then *κ(G)*=*n*−1. The algorithm will not call MAXFLOW() at all, since every *s* is adjacent to every *t*. The algorithm will terminate with *κ*=*n*−1. Otherwise *G* is not complete, so there exists a subset $U \subseteq V(G)$ such that *G*−*U* has at least two components, where |*U*|= *κ(G)*. The first *κ(G)* choices of vertex *s* may all be in *U*. However, by the (*κ(G)*+1)st choice of s we know that some $s \notin U$ has been selected. So *s* is in some component of *G*−*U*. The inner loop runs over all choices of *t*. One of these choices will be in a different component of *U*. For that particular *t,* the value of MAXFLO W(*N, s*2, *t*1) will equal |*U*|. After this, the value of *κ* in the algorithm will not decrease any more. Therefore we can conclude that some $s \notin U$ will be selected; that the value of *κ* after that point will equal *κ(G);* and that after this point the algorithm can stop. This is exactly what the algorithm executes.

The algorithm makes at most

$$\sum_{s=1}^{\kappa+1}(n-s)$$

calls to MAXFLOW(). Thus, it is a polynomial algorithm.

Exercises

8.5.1 Let *G* be *k*-connected. If $st \in E(G)$, prove that *G*−*st* is (*k*−1)-connected.

8.5.2 Program Algorithm 8.5.1, the CONNECTIVITY() algorithm.

8.5.3 Consider a network *N* where instead of specifying a capacity for each edge $\overrightarrow{uv}$, we specify a *lower bound* $b(\overrightarrow{uv}) \geq 0$ for the flow on edge $\overrightarrow{uv}$. Instead of the capacity constraint $f(\overrightarrow{uv}) \leq \text{CAP}(\overrightarrow{uv})$, we now have a lower bound constraint $f(\overrightarrow{uv}) \geq b(\overrightarrow{uv})$. The zero-flow is not a valid flow anymore. Show that there exists a valid flow in such a network if and only if for every edge $\overrightarrow{uv}$ such that $b(\overrightarrow{uv}) > 0$, $\overrightarrow{uv}$ is either: (i) on a directed *st*-path; or (ii) on a directed *ts*-path; or (iii) on a directed cycle. (*Hint:* If $\overrightarrow{uv}$ is not on such a path or cycle, follow directed paths forward from *u* and backward from *u* to get a contradiction.)

8.5.4 Consider the problem of finding a *minimum* flow in a network with lower bounds instead of capacities.

(a) How should an unsaturated path be defined?

(b) How should the capacity of an edge-cut be defined?

(c) Find a min-flow in the network of Figure 8.11, where the numbers are the lower bounds. Prove that your flow is minimum by illustrating an appropriate edge-cut.

(d) Is there a max-flow in the network given in Figure 8.11?

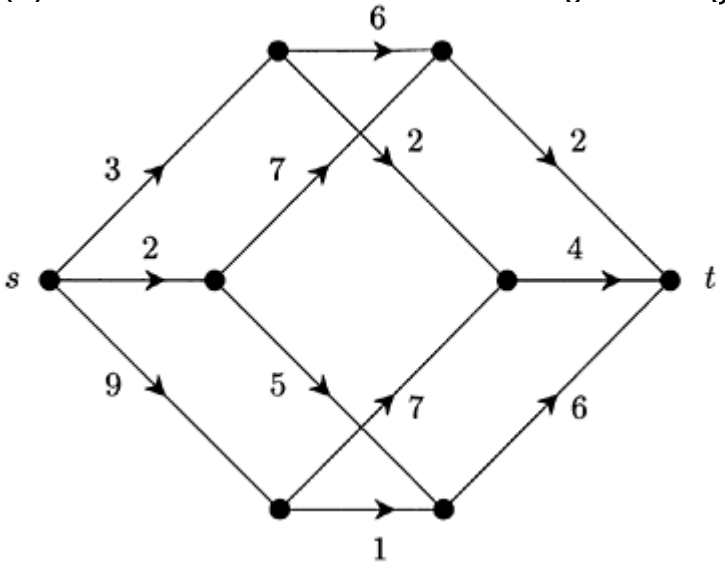


FIGURE 8.11
A network with lower bounds

8.5.5 Suppose that a network *N* has both lower bounds $b(\overrightarrow{uv})$ and capacities $\text{CAP}(\overrightarrow{uv})$ *on* its edges. We wish to find a max-flow *f* of *N*, where $b(\overrightarrow{uv}) \leq f(\overrightarrow{uv}) \leq \text{CAP}(\overrightarrow{uv})$. Notice that zero-flow may be no longer a valid flow. Before applying an augmenting path algorithm like the FF algorithm, we must first find a valid flow.

(a) Determine whether the networks of Figure 8.12 have a valid flow.

(b) How should residual capacity be defined?

(c) How should the capacity of an edge-cut be defined?

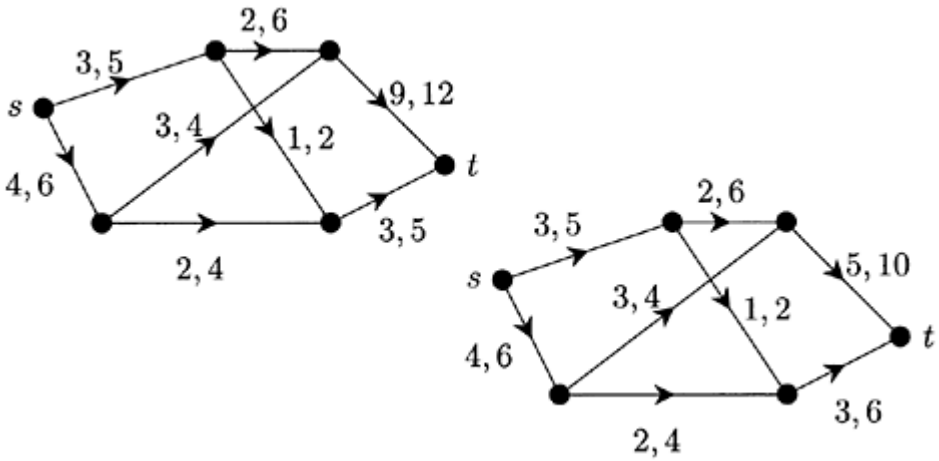


FIGURE 8.12
Networks with lower bounds and capacities

8.5.6 Let *N* be a network with lower bounds $b(\overrightarrow{uv})$ and capacities $\text{CAP}(\overrightarrow{uv})$ specified on its edges. Before finding a max-flow in *N* we need to find a valid flow. Construct a network *N′* as follows: Add a new source *s′* and target *t′*. Join *s′* to all vertices of *N*. Join every vertex of *N* to *t′*. Add edges $\overrightarrow{st}$ and $\overrightarrow{ts}$ to *N′*. The capacities in *N′* are defined as follows:

$$\text{CAP}'(\overrightarrow{s'u}) = \sum_v b(\overrightarrow{vu}), \text{ (sum over in-edges at } u \in V(N)).$$

$$\text{CAP}'(\overrightarrow{ut'}) = \sum_v b(\overrightarrow{uv}), \text{ (sum over out-edges at } u \in V(N)).$$

$$\text{CAP}'(\overrightarrow{uv}) = \text{CAP}(\overrightarrow{uv}) - b(\overrightarrow{uv}), (u, v \in V(N)).$$

$$\text{CAP}'(\overrightarrow{st}) = \text{CAP}'(\overrightarrow{ts}) = \infty.$$

Prove that there exists a valid flow in *N* if and only if there is a flow in *N′* that saturates all edges incident on *s′*.

8.5.7 Let *N* be a network such that there is a cost $c(\overrightarrow{uv})$ of using edge $\overrightarrow{uv}$, per unit of flow. Thus the cost of flow $f(\overrightarrow{uv})$ edge $\overrightarrow{uv}$ is $f(\overrightarrow{uv})c(\overrightarrow{uv})$. Devise an algorithm to find a max-flow of min-cost in *N*.

8.5.8 The circulation of money in the economy closely resembles a flow in a network. Each node in the economy represents a person or organization that takes part in economic activity. The main differences are that there may be no limit to the capacities of the edges, and that flow may accumulate at a node if assets are growing. Any transfer of funds is represented by a flow on some edge. Various nodes of the economic network can be organized into groups, such as banks, insurance companies, wage earners, shareholders, government, employers, etc.

(a) Develop a simplified model of the economy along these lines.

(b) A bank charges interest on its loans. If there is a fixed amount of money in the economy, what does this imply? What can you conclude about the money supply in the economy?

(c) When a new business is created, a new node is added to the network. Where does the flow through this node come from?

(d) Consider the node represented by government. Where does its in-flow come from? Where is its out-flow directed? Consider how government savings bonds operate in the model.

(e) Where does inflation fit into this model?

(f) How do shareholders and the stock market fit into the model?

8.6 Notes

The max-flow algorithm is one of the most important algorithms in graph theory, with a great many applications to other graph theory problems (such as connectivity and Menger's theorems), and to problems in discrete optimization. The original algorithm is from FORD and FULKERSON [43]. See also FULKERSON [47]. EDMONDS and KARP [39] proved that the use of shortest augmenting paths results in a polynomial time complexity of the Ford-Fulkerson algorithm.

Balanced flows were introduced by KOCAY and STONE [80], and then developed greatly in a series of papers by Fremuth-Paeger and Jungnickel. An excellent summary with many references can be found in FREMUTH-PAEGER and JUNGNICKEL [45].

A great many techniques have been developed to improve the complexity of the basic augmenting path algorithm. See PAPADIMITRIOU and STEIGLITZ [94] for further information. The algorithms to find the connectivity and edge-connectivity of a graph in Sections 8.4 and 8.5 are from EVEN [40].

Page 186
This page intentionally left blank.

Page 187
9
Hamilton Cycles
9.1 Introduction
A cycle that contains all vertices of a graph *G* is called a *hamilton cycle* (or *hamiltonian cycle*). *G* is *hamiltonian* if it contains a hamilton cycle. For example, Figure 9.1 shows a hamilton cycle in the graph called the *truncated tetrahedron*. It is easy to see that the graph of the *cube* is also hamiltonian (see Chapter 1).

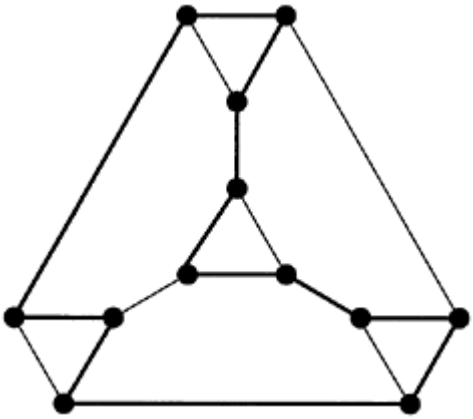


FIGURE 9.1
A hamiltonian graph
Figure 9.2 shows a non-hamiltonian graph *H*. It is easy to see that *H* is non-hamiltonian, since it is bipartite with an odd number of vertices. Clearly any bipartite graph that is hamiltonian must have an even number of vertices, since a hamilton cycle *C* must start and end on the same side of the bipartition. Although *H* is non-hamiltonian, it does have a *hamilton path,* that is, a path containing all its vertices.

Page 188

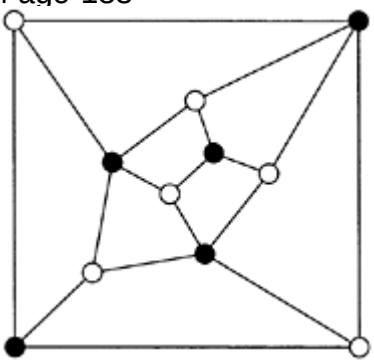


FIGURE 9.2
A non-hamiltonian graph
The problem of deciding whether a given graph is hamiltonian is only partly solved.
Problem 9.1: HamCycle
Instance: a graph *G*
Question: is *G* hamiltonian?
This is an example of an NP-*complete* problem. We will say more about NP-complete problems later. There is no known efficient algorithm for solving the HamCycle problem. Exhaustive search algorithms can take a very long time in general. Randomized algorithms can often find a cycle quickly if *G* is hamiltonian, but do not give a definite answer if no cycle is found.
The HamCycle problem is qualitatively different from most other problems in this book. For example, the

questions “is *G* bipartite, Eulerian, 2-connected, planar?”, and, “is a given flow *f* maximum?” can all be solved by efficient algorithms. In each case an algorithm and a theoretical solution are available. For the HamCycle problem, there is no efficient algorithm known, and only a partial theoretical solution. A great many graph theoretical problems are NP-complete.
A number of techniques do exist which can help to determine whether a given graph is hamiltonian. A graph with a cut-vertex *u* cannot possibly be hamiltonian, since a hamilton cycle *C* has no cut-vertex. This idea can be generalized into a helpful lemma.

LEMMA 9.1 If G is hamiltonian, and $S \subseteq V(G)$, then $w(G-S) \leq |S|$.
This lemma says that if we delete $k=|S|$ vertices from *G,* the number of

connected components remaining is at most *k*. Let *C* be a hamilton cycle in *G*. If we delete *k* vertices from *C,* the cycle *C* will be decomposed into at most *k* paths. Since *C* is a subgraph of *G,* it follows that $G-S$ will have at most *k* components.
For example, the graph of Figure 9.3 is non-hamiltonian, since the deletion of the three shaded vertices gives four components.

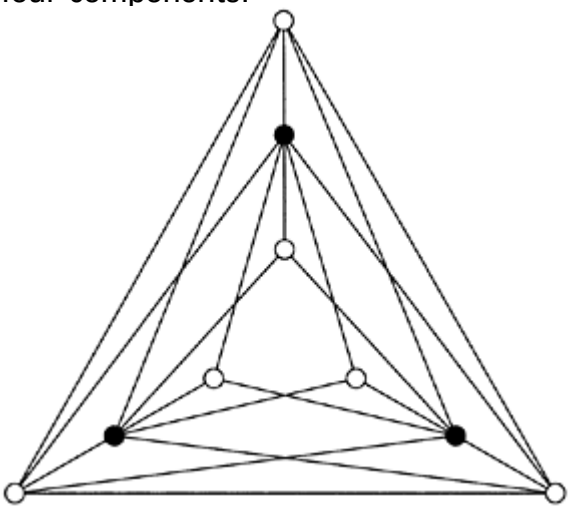


FIGURE 9.3
A non-hamiltonian graph
The Petersen graph is also non-hamiltonian, but this cannot be proved using Lemma 9.1. Instead we use an exhaustive search method called the *multi-path method,* see RUBIN [105]. Suppose that *C* were a hamilton cycle in the Petersen graph *G,* as shown in Figure 9.4. *G* is composed of an outer and inner pentagon, joined by a perfect matching. Since *C* uses exactly two edges at each vertex of *G,* it follows that *C* must use at least three edges of the outer pentagon, for otherwise some vertex on it would be missed by *C*. Consequently, *C* uses two adjacent edges of the outer pentagon. Without loss of generality, suppose that it uses the edges *uu* and *uw*. This means that *C* does not use the edge *uy,* so we can delete it from *G*. Deleting *uy* reduces the degree of *y* to two, so that now both remaining edges at *y* must be part of *C*. So the two paths *(u, u, w)* and *(x, y, z)* must be part of *C,* where a path is denoted by a sequence of vertices. This is illustrated in Figure 9.4. *C* must use two edges at *w,* so there are two cases. Either $wt \in C$ or $wr \in C$. Suppose first that $wt \in C$. Then since $wr \notin C$, *we* can delete wr from *G*. This reduces the degree of *r* to two, so that the remaining edges at *r* must be in *C*. Therefore $rz \in C$. This uses up two edges at *z,* so we delete *sz,* which in turn reduces the degree of *s* to two. Consequently the edge $us \in C$. But this now creates a cycle *(u, u, w, t, s)* in *C,* which is not possible. It follows that the choice

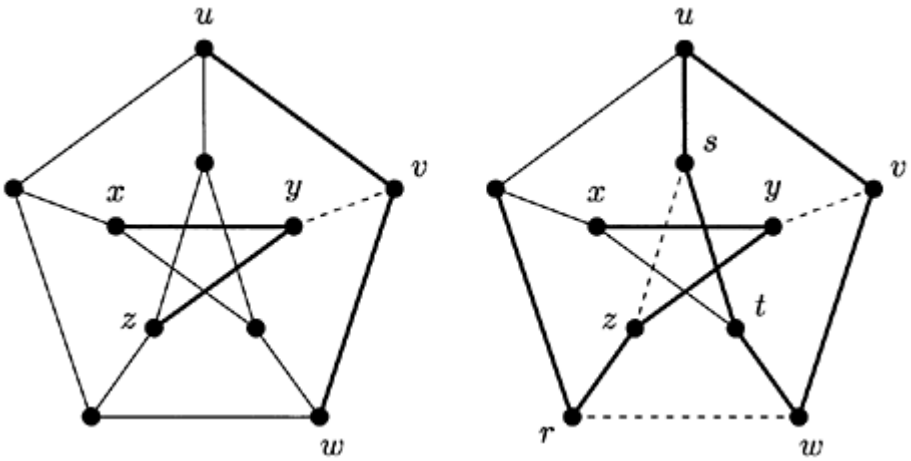


FIGURE 9.4
The multi-path method

$wt \in C$ was wrong. If we now try $wr \in C$ instead, a contradiction is again reached, thereby proving that the Petersen graph is non-hamiltonian.

This is called the multi-path method, since the cycle *C* is gradually built from a number of paths which are forced by two operations: the deletion of edges which are known not to be in *C;* and the requirement that both edges at a vertex of degree two be in *C*. The multi-path method is very effective in testing whether 3-regular graphs are hamiltonian, since each time an edge is deleted, the degree of two vertices reduces to two, which then forces some of the structure of *C*. Graphs of degree four or more are not so readily tested by it. We will say more about the multi-path method later on.

Exercises

9.1.1 Decide whether or not the graphs in Figure 9.5 are hamiltonian.

9.1.2 Prove that *Qn,* the n-cube, is hamiltonian for all $n \geq 2$.

9.1.3 Let *P* be a hamilton path in a graph *G,* with endpoints *u* and *u*. Show that $w(G-S) \leq |S|+1$, for all $S \subseteq V(G)$, in two ways:

a) By counting the components of $G-S$.

b) By counting the components of $(G+uu)-S$, and using Lemma 9.1.

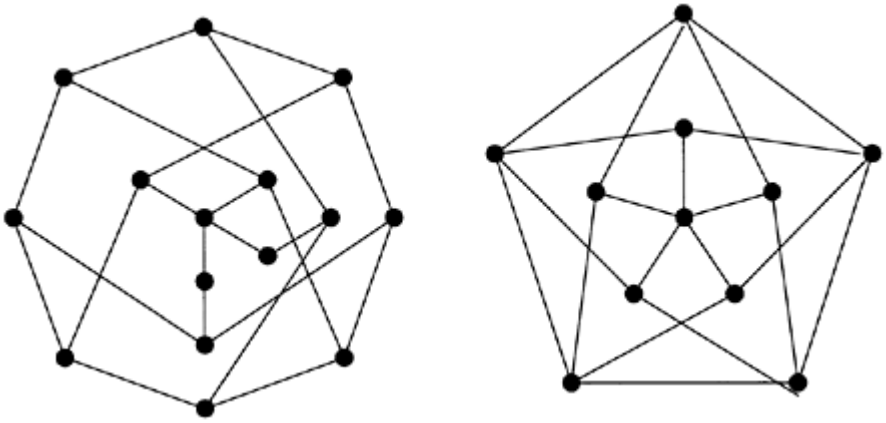


FIGURE 9.5
Are these hamiltonian?

9.2 The crossover algorithm

Suppose that we want to find a hamilton cycle in a connected graph *G*. Since every vertex of *G* must be part of *C,* we select any vertex *x*. We then try to build a long path *P* starting from *x*. Initially *P=(x)* is a path of length zero. Now execute the following steps:

u←*x; u*←*x* "*P* is a *uu*-path"

while $\exists w \longrightarrow u$ such that $w \notin P$

$$\textbf{do} \begin{cases} P \leftarrow P + uw \\ u \leftarrow w \end{cases}$$

while $\exists w \longrightarrow v$ such that $w \notin P$

$$\textbf{do}\ \begin{cases} P \leftarrow P + vw \\ v \leftarrow w \end{cases}$$

The first loop extends *P* from u and the second loop extends *P* from *u,* until it cannot be extended anymore. At this point we have a *uu*-path

$$P=(u,\ldots,\ x,\ldots,\ u)$$

such that the endpoints *u* and *u* are adjacent only to vertices of *P*. The length of *P* is *ℓ(P),* the number of edges in *P*. The vertices of *P* are ordered from *u* to *u*. If $w \in P$, then *w+* indicates the vertex following *w*(if *w≠u*). Similarly *w−* indicates the vertex preceding *w*(if *w≠u*).
If $u \longrightarrow v$, then we have a cycle *C=P+uu*. If *C* is a hamilton cycle, we are done. Otherwise, since *G* is connected, there is a vertex $w \in P$ such that

---

$w \longrightarrow y$, where $y \notin P$. Hence there exists a longer path

$$P^* = P - ww^+ + wy.$$

This is illustrated in Figure 9.6.

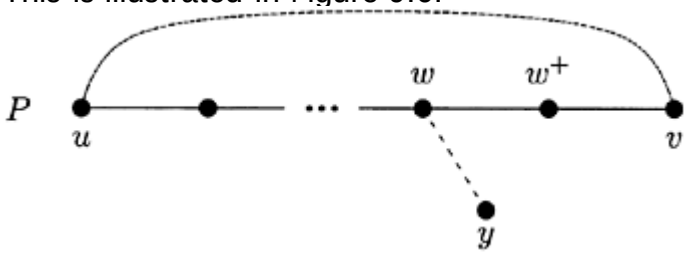


FIGURE 9.6
Finding a long path

If $u \not\longrightarrow v$, it may still be possible to find a cycle. Suppose that *P* contains a vertex *w* such that $v \longrightarrow w$ and $u \longrightarrow w^+$. This creates a pattern called a *crossover,* which is shown in Figure 9.7. When a crossover exists, there is a cycle

$$C=P+uw-ww^+ + uw^+$$

containing all the vertices of *P*.

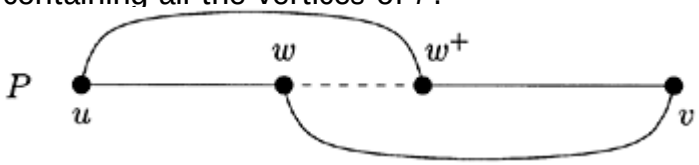


FIGURE 9.7
A crossover

Having converted the path *P* to a cycle *C* using the crossover, we are again in the situation where either *C* is a hamilton cycle, or else it contains a vertex $w \longrightarrow y \notin C$, which allows us to find a longer path *P**. We now extend *P** from both endpoints as far as possible, and then look for a crossover again, etc. The algorithm terminates either with a hamilton cycle, or with a long path that has no crossover. The crossover algorithm is summarized in Algorithm 9.2.1.

Algorithm 9.2.1: LONGPATH*(G,x)*

$$\textbf{comment:}\ \begin{cases} \text{Find a long path in } G \text{ containing } x\text{, using crossovers.} \\ P \text{ and } C \text{ are linked lists.} \end{cases}$$

u←x; u←x; P←(x) “a path of length 0”

repeat

comment: extend *P* from *u*

$$\textbf{while}\ \exists w \longrightarrow u \text{ such that } w \notin P$$

$$\textbf{do}\ \begin{cases} \text{add } w \text{ to } P \\ u \leftarrow w \end{cases}$$

comment: extend *P* from *u*.

while $\exists w \longrightarrow v$ such that $w \notin P$

do $\begin{cases} \text{add } w \text{ to } P \\ v \leftarrow w \end{cases}$

comment: search for a crossover

for all $w \longrightarrow v$ **do if** $u \longrightarrow w^+$

then $\begin{cases} \textbf{comment:} \text{ a crossover has been found} \\ C \leftarrow P + vw - ww^+ + uw^+ \\ \textbf{if } C \text{ is a hamilton cycle} \\ \quad \textbf{then go to } 1 \\ \text{find } z \in C \text{ such that } z \longrightarrow y \notin C \\ \text{convert } C + zy \text{ into a path } P \text{ from } y \text{ to } z^+ \\ u \leftarrow y; \quad v \leftarrow z^+ \end{cases}$

until no crossover was found

1: **comment:** *P* can be extended no more

9.2.1 Complexity

The main operations involved in the algorithm are extending *P* from *u* and *u,* converting *P* to a cycle *C,* and finding $z \in C$ such that $z \longrightarrow y \notin C$. We assume that the data structures are arranged so that the algorithm can check whether or not a vertex *w* is on *P* in constant time. This is easy to do with a boolean array. We also assume that the algorithm can test whether or not vertices *u* and *w* are adjacent in constant time.

• Extending *P* from *u* requires at most DEG*(u)* steps, for each *u*. Since *P* can extend at most once for each *u,* the total number of steps taken to extend *P* is at most $\sum_u \text{DEG}(u) = 2\varepsilon$, taken over all iterations of the algorithm.

Page 194

• Converting *P* to a cycle $C = P + uw - ww^+ + uw^+$ requires reversing a portion of *P*. This can take up to *ℓ(P)* steps. As *ℓ(P)* increases from 0 up to its maximum, this can require at most $O(n2)$ steps, taken over all iterations.

• Checking whether $z \in C$ is adjacent to some $y \notin C$ requires DEG*(z)* steps for each *z*. There are $\ell(C) = \ell(P) + 1$ vertices *z* to be considered. If at some point in the algorithm it is discovered that some *z* is not adjacent to any such *y,* we need never test that *z* again. We flag these vertices to avoid testing them twice. Thus the total number of steps spent looking for *z* and *y* is at most $O(n^2) + \sum_z \text{DEG}(z) = O(n^2 + \varepsilon)$.

So the total complexity of the algorithm is $O(n2+\varepsilon)$. More sophisticated data structures can reduce the $O(n2)$ term, but there is likely no reason to do so, since the algorithm is already fast, and it is not guaranteed to find a hamilton cycle in any case.

The crossover algorithm works very well on graphs which have a large number of edges compared to the number of vertices. In some cases we can prove that it will always find a hamilton cycle. On sparse graphs (e.g., 3-regular graphs), it does not perform very well when the number of vertices is more than 30 or so.

LEMMA 9.2 Let G be a graph on n vertices such that DEG(u)+DEG(u}≥n, for all non-adjacent vertices u and u. Then the crossover algorithm will always find a hamilton cycle in G.

PROOF If the crossover algorithm does not find a hamilton cycle, let *P* be the last path found. Since *P* can't be extended from its endpoints *u* and *u,* it follows that *u* and *u* are joined only to vertices of *P*. For each $w \longrightarrow v$, it must be that $u \not\longrightarrow w^+$, or a crossover would exist. Now *u* is joined to DEG*(u)* vertices of *P*. There are thus DEG*(u)* vertices that *u* is not joined to. Consequently *u* can be adjacent to at most *ℓ(P)*−DEG*(u)* vertices, where $\ell(P) \le n-1$ is the number of edges of *P*. So we have

$$\text{DEG}(u) + \text{DEG}(u) \le \ell(P) \le n-1,$$

a contradiction, since we assumed that DEG*(u)*+DEG*(u)*≥*n* for all non-adjacent *u* and *u*.

This lemma also shows that graphs which satisfy the condition DEG*(u)*+ DEG*(u)*≥*n* are always hamiltonian. Such graphs have many edges, as we shall see. However, the crossover algorithm will often find hamilton cycles or hamilton paths, even when a graph does not satisfy this condition.

The crossover algorithm can be improved enormously by searching for crossovers of higher order. The crossover of Figure 9.7 can be defined to be

the trail *Q=(u, w+, w, u)* which starts at *u,* intersects *P* in exactly one edge, and finishes at *u*. The cycle C is then given by $C = P \oplus Q$, where $\oplus$ indicates the operation of exclusive-OR, applied to the edges of *P* and *Q.* In general, higher order crossovers can be defined as follows.

DEFINITION 9.1: Let *P* be a *uu*-path. A *crossover Q* is a *uu*-trail such that $V(Q) \subseteq V(P)$ and $C = P \oplus Q$ is a cycle with *V(C)=V(P)*. The *order* of a crossover Q is the number *|P∩Q|* of edges common to *P* and *Q*. A *cross-edge* is any edge $xy \in E(Q) - E(P)$.

So a crossover of order 0 occurs when $u \longrightarrow v$. Then *Q=(u,u)* and *C= P+uu.* There is only one kind of crossover of order one, which is shown in Figure 9.7. A crossover of order two is illustrated in Figure 9.8. There are five different kinds of crossover of order two, as the reader can verify by constructing them. An algorithm employing crossovers of order higher than one requires a recursive search for crossovers up to a pre-selected maximum order M. It was found by KOCAY and LI [78] that choosing *M*=6 still gives a fast algorithm, and that it improves the performance of the basic algorithm enormously. This algorithm requires sophisticated data structures for an efficient implementation.

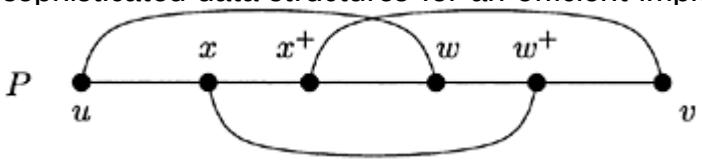


FIGURE 9.8
A crossover *Q=(u, w, w+, x, x+, u)* **of order two**

Suppose that a path *P* is the longest path found by the algorithm, and that it has no crossover. If there is a vertex $x \notin P$ such that *x→w, w+*, for some $w \in P$, then we can make a longer path by re-routing *P* through *x: P′=*(..., *w, x, w+,*...). Similarly, a configuration like Figure 9.9 can also be used to give a longer path. Once *P* has been re-routed to a longer path, we can again check for a crossover. When used in combination, crossovers and reroutings will very often find a hamilton cycle in *G,* if it is hamiltonian, even for sparse graphs *G.*

A re-routing is very much like a crossover. It is a closed trail *Q* whose endpoints are on the *uu*-path *P,* such that $P \oplus Q$ is a *uu*-path containing all vertices of *P*. It always results in a longer path. The algorithm that searches for higher order crossovers can be easily modified to search for re-routings as well.

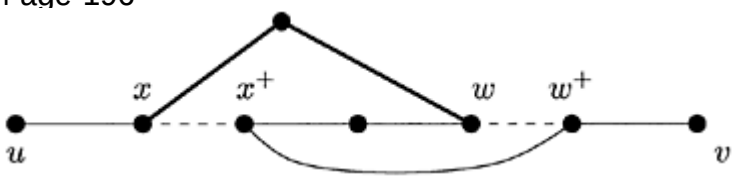


FIGURE 9.9
Re-routing *P*

Exercises

9.2.1 Show that if *G* is connected and *n*>2*δ,* where *δ* is the minimum degree of *G,* then *G* has a path of length at least 2*δ*. This is due to DIRAC [36]. (*Hint:* Consider a longest path.)

9.2.2 Program the crossover algorithm, and test it on the Petersen graph, on the graphs of Figure 9.5, and on the graph in Figure 9.10. Try it from several different starting vertices.

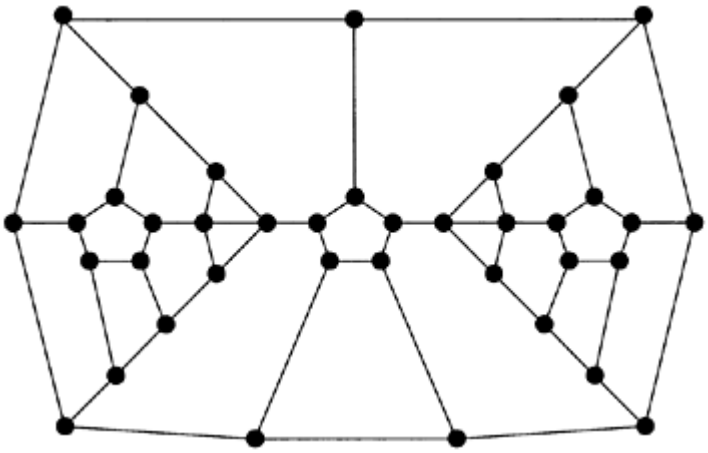


FIGURE 9.10
The Lederberg graph

9.2.3 Let *G* be a graph. Show how to create a graph *G′* from *G* by adding one vertex so that *G* has a hamilton path if and only if *G′* has a hamilton cycle.
9.2.4 Let *G* be a graph such that DEG*(u)*+DEG*(u)*≥*n*−1, for all non-adjacent vertices *u* and *u*. Show that *G* has a hamilton path.
9.2.5 Construct all five kinds of crossover of order two.
9.2.6 Construct the crossovers of order three.

Page 197

9.3 The Hamilton closure

Suppose that DEG*(u)*+DEG*(u)*≥*n* in a graph *G,* where *u* and *u* are non-adjacent vertices. Let *G′*=*G*+*uu*. If *G* is hamiltonian, then so is *G′*. Conversely, if *G′* is hamiltonian, let *C* be a hamilton cycle in *G′*. If $uv \in C$, then *P*=*C*−*uu* is a hamilton path in *G*. Since DEG*(u)*+DEG*(u)*≥*n,* we know that *P* has a crossover, so that *G* has a hamilton cycle, too. Thus we have proved:

LEMMA 9.3 Let DEG*(u)*+DEG*(u)*≥*n in a graph G, for non-adjacent vertices u and u. Let G′*=*G*+*uu. Then G is hamiltonian if and only if G′ is.*

This lemma says that we can add all edges *uu* to *G,* where DEG*(u)*+DEG*(u)*≥ *n,* without changing the hamiltonicity of *G*. We do this successively, for all non-adjacent vertices *u* and *u*.

DEFINITION 9.2: The *hamilton closure* of *G* is *cH(G),* the graph obtained by successively adding all edges *uu* to *G,* whenever DEG*(u)*+DEG*(u)*≥*n,* for non-adjacent vertices *u* and *u*.

For example, the hamilton closure of the graph of Figure 9.11 is the complete graph *K*7. It must be verified that this definition is valid, namely, no matter in what order the edges *uu* are added to *G,* the resulting closure is the same. We leave this to the reader.

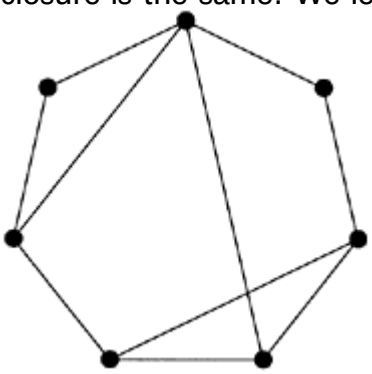


FIGURE 9.11
cH(G)=*K*7

Lemma 9.3 tells us that *cH(G)* is hamiltonian if and only if *G* is. In particular, if *cH(G)* is a complete graph, then *G* is hamiltonian. The hamilton closure can be used to obtain a condition on the degree sequence of *G* which will force *G* to be hamiltonian.

Page 198

THEOREM 9.4 **(Bondy-Chvátal theorem)** *Let G be a simple graph with degree sequence* (*d*1, *d*2,..., *dn*),

where d1≤d2≤...≤dn. If there is no m<n/2 such that dm≤m and dn−m<n−m, then cH(G) is complete.
PROOF Suppose that *cH(G)* is not complete. Let *u* and *u* be non-adjacent vertices such that DEG*(u)*+DEG*(u)* is as large as possible, where the degree is computed in the closure *cH(G)*. Then DEG*(u)*+DEG*(u)*<*n* by definition of the closure. Let *m*=DEG*(u)*≤DEG*(u)*. So *u* is joined to *m* vertices. There are *n*−DEG*(u)*−1 vertices that *u* is not adjacent to (not counting *u,* since $v \not\longrightarrow v$ Each of these has degree ≤*m*. So the number of vertices with degree ≤*m* is at least *n*−DEG*(u)*−1. But DEG*(u)*+DEG*(u)*<*n,* so that *m*=DEG*(u)*≤*n*−DEG*(u)*−1. That is, the number of vertices of the closure with degree ≤*m* is at least *m*. Since the degree sequence of *cH(G)* is at least as big as that of *G,* it follows that *dm*≤*m*.

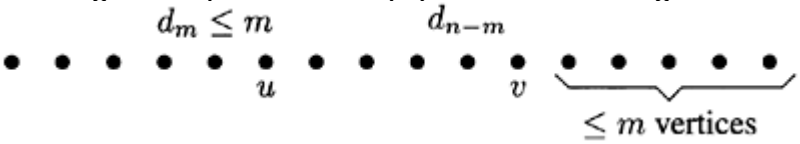


FIGURE 9.12
The degree sequence of *G*
How many vertices have degree >DEG*(u)*? We know that *u* is adjacent to all of them. Therefore, the number of them is at most *m,* so that there are at most *m* vertices after *u* in the degree sequence. It follows that DEG*(u)*≥*dn*−*m*. But since DEG*(u)*<*n*−*m,* it follows that *dn*−*m*<*n*−*m*. Thus, we have found a value *m* such that *dm*≤*m* and *dn*−*m*<*n*−*m*. Here *m*=DEG*(u)*≤DEG*(u)*<*n*−*m,* so that *m*<*n*/2. This contradicts the assumptions of the theorem. Therefore *cH(G)* must be complete under these conditions.
The degree sequence condition of the Bondy-Chvátal theorem is easy to apply. For example, any graph with the degree sequence (2,2,3,4,5,6,6,6) must be hamiltonian, since *d*1=2>1, *d*2=2≤2, but $d_{8-2} = 6 \not< 6,$ and *d*3=3≤3, but $d_{8-3} = 5 \not< 5.$ Thus there is no *m*<8/2 satisfying the condition that *dm*≤*m* and *dn*−*m*<*n*−*m*. This is the strongest degree sequence condition possible which forces an arbitrary graph *G* to be hamiltonian. Any stronger condition would have to place non-degree sequence restrictions on *G*. To see that this is so, let *G* be any non-hamiltonian graph. Let its degree sequence be (*d*1, *d*2,..., *dn*), where *d*1≤*d*2≤...≤*dn*. Since *G* is not hamiltonian, there is a value *m* such that *dm*≤*m* and *dn*−*m*<*n*−*m*. Construct a new degree sequence by increasing

each *di* until the sequence (*m,*..., *m,n*−*m*−1,..., *n*−*m*−1, *n*−1,...,*n*−1) is obtained, where the first *m* degrees are *m,* the last *m* degrees are *n*−1, and the middle *n*−2*m* degrees are *n*−*m*−1. We construct a non-hamiltonian graph *C(m, n)* with this degree sequence.

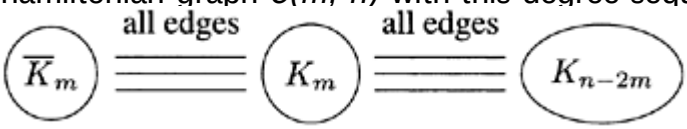


FIGURE 9.13
C(m, n)
C(m, n) is composed of three parts, a complete graph *Km,* a complete graph *Kn*−2*m,* and an empty graph $\overline{K}_m$. Every vertex of $\overline{K}_m$ is joined to every vertex of *Km,* and every vertex of *Km* is joined to every vertex of *Kn*−2*m*. This is illustrated in Figure 9.13. The vertices of $\overline{K}_m$ have degree *m,* those of *Kn*−2*m* have degree *n*−*m*−1, while those of *Km* have degree *n*−1. *C*(3, 9) is illustrated in Figure 9.3. It is easy to see that (*C(m, n)* is always non-hamiltonian, since the deletion of the vertices of *Km* leaves *m*+1 components. By Lemma 9.1, we conclude that *C(m, n)* is non-hamiltonian. Yet for every non-hamiltonian graph *G* on *n* vertices, there is some *C(m, n)* whose degree sequence is at least as large as that of *G,* in the lexicographic order.
Exercises
9.3.1 Prove that *cH(G)* is well-defined; that is, the order in which edges *uu* are added to *G* does not affect the result.
9.3.2 Prove that the crossover algorithm will find a hamilton cycle in *G* if *cH(G)* is complete or find a counterexample.
9.3.3 Use the Bondy-Chvátal theorem to show that any graph with the degree sequence (2, 3, 3, 4, 5, 6, 6, 6, 7) is hamiltonian. What about (3, 3, 4, 4, 4, 4, 4, 4)?
9.3.4 Define the hamilton-path closure to be $c'_H(G)$, obtained by adding all edges *uu* whenever DEG*(u)*+DEG*(u)*≥*n*−1. Prove that *G* has a hamilton path if and only $c'_H(G)$ does.
9.3.5 Obtain a condition like the Bondy-Chvátal theorem which will force $c'_H(G)$ to be complete.
9.3.6 Construct the graphs *C*(2, 8) and *C*(4, 12).

9.3.7 Work out $\varepsilon(C(m, n))$. Show that ε has its smallest value when

$$m = \frac{n}{3} - \frac{1}{6},$$

for which

$$\varepsilon = \frac{2}{3}\binom{n}{2} - \frac{1}{24}.$$

9.3.8 Show that if G is a graph on $n \geq 4$ vertices with $\varepsilon \geq \binom{n-1}{2} + 1$, then G is hamiltonian.
Notice that according to Exercise 9.3.7, $C(m,n)$ has approximately two-thirds of the number of edges of the complete graph K_n, at the minimum. This means that degree sequence conditions are not very strong. They apply only to graphs with very many edges.

9.4 The extended multi-path algorithm

The multi-path algorithm tries to build a hamilton cycle C using a recursive exhaustive search. At any stage of the algorithm, a number of disjoint paths $S1$, $S2$,..., Sk in G are given, which are to become part of C. Call them *segments* of C. Initially, we can take $k=1$, and the single segment $S1$ can consist of the starting vertex, that is, a path of length zero. On each iteration a vertex u is selected, an endpoint of some segment $P=Si$. Every $w \longrightarrow u$ is taken in turn, and P is extended to $P'=P+uw$. Vertex u may now have degree two in Si. In this case, the remaining edges ux of G are deleted. This reduces each DEG(x) by one. When DEG$(x)=2$, both remaining edges at x must become part of C. A new segment is created containing x. Thus, the choice of uw can force certain edges to be a part of C. It can also happen that when edges are forced in this way, that an edge connecting the endpoints of two segments is forced, and the two segments must be merged into one. This in turn forces other edges to be deleted, etc. The forcing of edges can be performed using a queue.
There are three possible outcomes of this operation:
1. An updated set of segments can be produced.
2. A hamilton cycle can be forced.
3. A small cycle can be forced.
By a small cycle, we mean any cycle smaller than a hamilton cycle. If a small cycle is forced, we know that the extension of P to $P+uw$ does not lead to a hamilton cycle. If a hamilton cycle is forced, the algorithm can quit. If a new set of segments is produced, the algorithm proceeds recursively. This can be summarized as follows. We assume a global graph G, and a global boolean variable *IsHamiltonian,* which is initially **false,** but is changed to **true** when a hamilton cycle is discovered.

Algorithm 9.4.1: MULTIPATH(S)

comment: Search for a ham cycle containing all segments of S
choose a vertex u, an endpoint of some path $P \in S$
for all $w \longrightarrow u$
do
- extend path P to $P + uw$
- **comment:** extending P to $P + uw$ may force some edges
- FORCEEDGES(uw)
- **if** a hamilton cycle was forced
  - **then**
    - *IsHamiltonian* $\leftarrow$ **true**
    - **return**
- **if** a small cycle was not forced
  - **then**
    - **comment:** the segments S have been updated
    - MULTIPATH(S)
    - **if** *IsHamiltonian*
      - **then return**
- restore G and S to their state before uw was chosen

comment: otherwise no hamilton cycle was found

Suppose that the multi-path algorithm were applied to a disconnected graph G. Although we know that G is

not hamiltonian, the algorithm could still take a very long time to discover this, for example, the connected components of *G* could be complete graphs. More generally, it is quite possible for the operation of forcing edges to delete enough edges so as to disconnect *G*. Thus the algorithm really is obliged to check that *G* is still connected before making a recursive call. This takes *O(ε)* steps. Now we know that a graph with a cut-vertex also has no hamilton cycle, and we can test for a cut-vertex at the same time as checking that *G* is connected. A depth-first search (DFS) can do both in *O(ε)* steps. Thus we add a DFS to the multi-path algorithm before the recursive call is made. But we can make a still greater improvement.
Suppose that the multi-path algorithm were applied to the graph of Figure 9.14. This graph is non-hamiltonian because the deletion of the two shaded vertices leaves three components. In certain cases the algorithm is able to detect this, using the DFS that tests for cut-vertices. Suppose that the segments of *G* are the bold edges. Notice that one of the segments contains the shaded vertex *u*. When the non-segment edges incident on *u* are deleted, *u* becomes a cut-vertex in the resulting graph. The DFS will detect that *u* is a cut-vertex, and the algorithm will report that adding the edge *uw* to the segment does not extend to a hamilton cycle.
Normally the algorithm would then try the next edge incident on *u,* etc. But it can do more. When the cut-vertex *u* is discovered, the DFS can *count the*

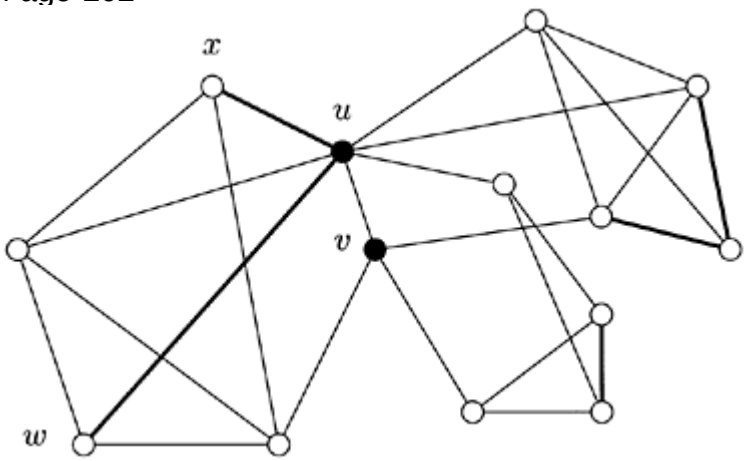


FIGURE 9.14
A non-hamiltonian graph

number of components of *G–u*. This will be one plus the number of descendants of *u* in the DF-tree. It requires almost no extra work for the DFS to calculate this. For vertex *u* in Figure 9.14, the count will be three components. But since this is the result of deleting only two vertices, namely, *u* and *u,* the algorithm can determine that the original *G* is non-hamiltonian, and stop the search at that point. More generally, a non-hamiltonian graph like Figure 9.14 can arise at some stage during the algorithm as a result of deleting edges, even though the original *G* is hamiltonian. The algorithm must be able to detect which graph in the search tree is found to be non-hamiltonian by this method. We leave it to the reader to work out the details.
It is helpful to view vertices like *u* in Figure 9.14 which have degree two in some segment as having been deleted from *G*. Each segment is then replaced by an equivalent single edge connecting its endpoints. The set of segments then becomes a matching in *G,* which is changing dynamically. For example, when the segments of Figure 9.14 are replaced by matching edges, the resulting graph appears as in Figure 9.15. The procedure which forces edges can keep a count of how many vertices internal to segments have been deleted in this way, at each level in the recursion. When the DFS discovers a cut-vertex, this count is used to find the size of a separating set in *G*. In cases like this, large portions of the search tree can be avoided.
A bipartite graph like the Herschel graph of Figure 9.2 is also non-hamiltonian, but the algorithm is not likely to delete enough vertices to notice that it has a large separating set. In general, suppose that at some stage in the algorithm *G–E(S)* is found to be bipartite, with bipartition *(X,Y),* where *S* is viewed as a matching in *G*. If there is a hamilton cycle *C* in *G* using the matching edges *S,* it must

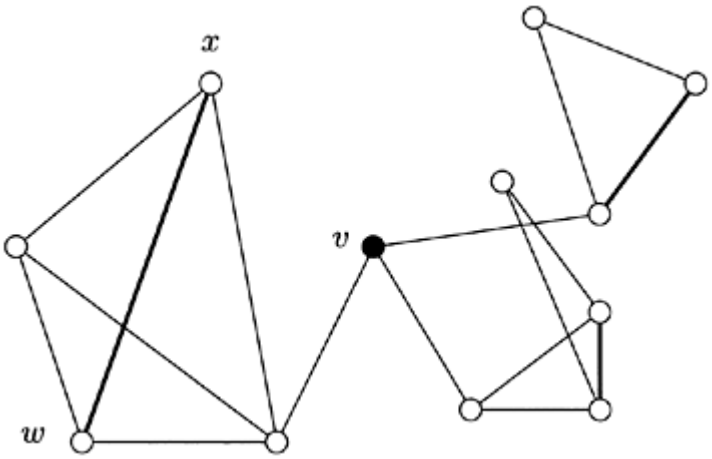


FIGURE 9.15
Segments viewed as a matching

look something like Figure 9.16, where the bipartition of $G-E(S)$ is shown by the shading of the nodes. There are now three kinds of segments: those contained within X, those contained within Y, and those connecting X to Y. Suppose that there are εX of the first type, and εY of the second type. The vertices of C must alternate between X and Y, except for the εX and εY edges, which must have endpoints of the same color. If we contract each of these edges to a single node, we obtain perfect alternation around the cycle. Therefore $|X|-\varepsilon X=|Y|-\varepsilon Y$ if G has a hamilton cycle. If this condition is not satisfied, we know that G is non-hamiltonian, and can break off the search. We again employ the DFS that tests for cut-vertices to simultaneously check whether $G-E(S)$ is bipartite, and to keep a count of the numbers $|X|-\varepsilon X$ and $|Y|-\varepsilon Y$. This requires very little extra work, and is still $O(\varepsilon)$. In this way, non-hamiltonian graphs derived from bipartite graphs or near-bipartite graphs can often be quickly found to be non-hamiltonian.

In summary, the extended multi-path algorithm adds a DFS before the recursive call. The DFS computes several things:

- Whether G is connected.
- $w(G-u)$, for each cut-vertex u.
- Whether $G-E(S)$ is bipartite.
- $|X|-\varepsilon X$ and $|Y|-\varepsilon Y$, if $G-E(S)$ is bipartite.

It may be possible to add other conditions to detect situations when G is non-hamiltonian. For example, every hamiltonian graph G with an even number of vertices n has two disjoint perfect matchings. If n is odd, every $G-u$ has a perfect matching.

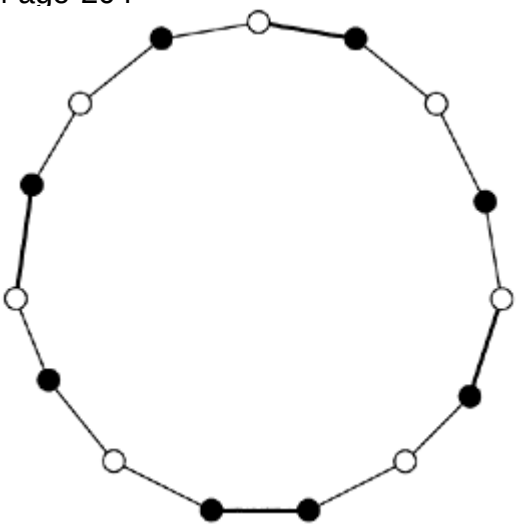


FIGURE 9.16
$G-E(S)$ **is bipartite**

9.4.1 Data structures for the segments

The extended multi-path algorithm still has exponential worst-case running time. Operations on the segments must be made as fast as possible. The operations that must be performed using segments are, given any

vertex *u,* to determine which segment contains *u,* and to find its endpoints; and to merge two segments when their endpoints are joined. One way to do this is with the merge-find data structure. An array *Segment[u]* is stored, which is an integer, pointing to the representative of the segment containing *u*. Each segment has two endpoints, which we arbitrarily designate as the right and left endpoints. The right endpoint *x* is the segment representative. It is indicated by a negative value of *Segment[x]*. Its value is $-y$, where *y* is the left endpoint. Thus we find the segment representative by following the pointers, using path compression (see Chapter 2). Segments are merged by adjusting the pointers of their endpoints.

Exercises

9.4.1 Program the multi-path algorithm. Use a DFS to test for the conditions mentioned above.

9.5 Decision problems, NP-completeness

The theory of NP-*completeness* is phrased in terms of *decision problems,* that is, problems with a yes or no answer, (e.g., "is *G* hamiltonian?"). This is so that an algorithm can be modeled as a Turing machine, a theoretical model of computation. Although Turing machines are very simple, they can be constructed to execute all the operations that characterize modern random access computers. Turing machines do not usually produce output, except for yes or no. Thus, a Turing machine can be constructed to read in a graph, and perform an exhaustive search for a hamilton cycle. If a cycle exists, it will be found, and the algorithm will report a yes answer. However, the exhaustive search will tend to take an exponential amount of time in general.

The class of all decision problems contains an important subclass called P, all those which can be solved in *polynomial* time; that is, the complexity of a problem is bounded by some polynomial in its parameters. For a graph, the parameters will be *n* and *ε,* the number of vertices and edges.

There is another class of decision problems for which polynomial algorithms are not always known, but which have an additional important property. Namely, if the answer to a problem is yes, then it is possible to write down a solution which can be verified in polynomial time. The HamCycle problem is one of these. If a graph *G* has a hamilton cycle *C,* and the order of vertices on the cycle is written down, it is easy to check in *n* steps that *C* is indeed a hamilton cycle. So if we are able to guess a solution, we can verify it in polynomial time. We say that we can write a *certificate* for the problem, if the answer is yes. A great many decision problems have this property that a certificate can be written for them if the answer is yes, and it can be checked in polynomial time. This forms the class NP of *non-deterministic polynomial* problems. The certificate can be checked in polynomial time, but we do not necessarily have a deterministic way of finding a certificate.

Now it is easy to see that $\mathrm{P} \subseteq \mathrm{NP}$, since every problem which can be solved in polynomial time has a certificate—we need only write down the steps which the algorithm executed in solving it. It is generally believed that HamCycle is in NP but not in P. There is further evidence to support this conjecture beyond the fact that no one has been able to construct a polynomial-time algorithm to solve HamCycle; namely, it can be shown that the HamCycle problem is one of the NP-complete problems.

To understand what NP-completeness means we need the concept of polynomial transformations. Suppose Π1 and Π2 are both decision problems. A *polynomial transformation* from Π1 to Π2 is a polynomial-time algorithm which when given any instance *I*1 of problem Π1 will generate an instance *I*2 of problem Π2 satisfying:

*I*1 *is a yes instance of Π*1 *if and only if I*2 *is a yes instance of Π*2

We use the notation $\Pi_1 \propto \Pi_2$ to indicate that there is a polynomial transformation from Π1 to Π2. We say that Π1*reduces* to Π2. This is because if we can find a polynomial algorithm *A* to solve I12, then we can transform Π1 into Π2, and then use *A* to solve Π2, thereby giving a solution to Π1.

DEFINITION 9.3: A decision problem is I1 is *NP-complete,* if

1. Π is in NP.

2. For any problem $\Pi' \in \mathrm{NP}, \Pi' \propto \Pi.$

It was Cook who first demonstrated the existence of NP-complete problems. He showed that Problem 9.2, *satisfiability of boolean expressions* (Sat) is NP-complete. Let *U* be a set of *n* boolean variables *u*1, *u*2,..., *un* with their complements $\bar{u}1$, $\bar{u}2$,..., $\bar{u}n$. These variables can only take on the values **true** and **false,** such that *ui* is **true** if and only if $\bar{u}i$ is **false,** and vice versa. If $x, y \in U$, then we denote by *x*+*y* the boolean **or** of *x* and *y* by *xy* the boolean **and** of *x* and *y*. A *clause* over *U* is a sum of variables in *U*. For example, $(u1+\bar{u}3+\bar{u}4+u6)$ is a clause. A boolean expression is a product of clauses. For example $(u1+\bar{u}3+\bar{u}4+u6)(u2+u5)(\bar{u}7)$ is a boolean expression. A truth assignment *t* is an assignment of values **true** and **false** to the variables in *U*. If *B* is a boolean expression, then *t(B)* is the evaluation of *B* with truth

assignment *t*. For example if

$$B=(u1+\bar{u}3+\bar{u}4+u6)(u2+u5)(\bar{u}7)$$

and

$$t=\begin{pmatrix} u_1 & u_2 & u_3 & u_4 & u_5 & u_6 & u_7 \\ \textbf{true} & \textbf{false} & \textbf{false} & \textbf{true} & \textbf{true} & \textbf{false} & \textbf{false} \end{pmatrix},$$

then

$$t(B)=(\textbf{true}+\textbf{true}+\textbf{false}+\textbf{false})(\textbf{false}+\textbf{true})(\textbf{true})=\textbf{true}.$$

Not every boolean expression *B* has a truth assignment *t* such that *t(B)*=**true**. For example there is no way to assign **true** and **false** to the variables in the expression $(\bar{u}1+\bar{u}2)(u1)(u2)$ so that it is **true**. if there is a truth assignment *t* such that *t(B)*=**true,** we say that *B* is *satisfiable*. The *satisfiability of boolean expressions* problem is

Problem 9.2: Sat
Instance: a set of boolean variables *U* and a boolean expression *B* over *U*.
Question: is *B* satisfiable?

Page 207
and was shown by COOK [28] to be NP-complete. See also KARP [70]. The proof of this is beyond the scope of this book; however, a very readable proof can be found in the book by PAPADIMITRIOU and STEIGLITZ [94]. Many problems have subsequently been proved NP-complete, by reducing them either to satisfiability, or to other problems already proved NP-complete.
The importance of the NP-complete problems is that, if a polynomial algorithm for any NP-complete problem II were discovered, then every problem in NP would have a polynomial algorithm; that is, P=NP would hold. Many people have come to the conclusion that this is not very likely, on account of the large number of NP-complete problems known, all of which are extremely difficult. We will now show that

$$\text{Sat} \propto \text{3-Sat} \propto \text{Vertex Cover} \propto \text{HamCycle}$$

and thus the HamCycle problem (as well as 3-Sat and Vertex Cover) is an NP-complete problem. Thus if P≠NP, then a polynomial algorithm for the HamCycle problem would not exist. This is why we say that the HamCycle problem is qualitatively different from most other problems in this book.
Among the most useful problems for establishing the NP-completeness of other problems is 3-Sat.

Problem 9.3: 3-Sat
Instance: a set of boolean variables *U* and a boolean expression *B* over *U,* in which each clause contains exactly three variables.
Question: is *B* satisfiable?

THEOREM 9.5 3-Sat is NP-*complete.*

PROOF It is easy to see that 3-Sat is in NP. Any truth assignment satisfying the boolean expression *B* can be checked in polynomial time by assigning the variables and then evaluating the expression.
We reduce Sat to 3-Sat as follows. Let *U* be a set of boolean variables and *B*=*C*1*C*2…*Cm* be an arbitrary boolean expression, so that *U* and *B* is an instance of Sat. We will extend the variable set set *U* to a set *U′* and replace each clause *Ci* in *B* by a boolean expression *Bi,* such that
(a) *Bi* is a product of clauses that use exactly three variables of *U′*.
(b) *Bi* is satisfiable if and only if *Ci* is.
Then *B′*=*B*1*B*2…*Bm* will be an instance of 3-Sat that is satisfiable over *U′* if and only if *B* is satisfiable over *U*.
Let *Ci*=(*x*1+*x*2+*x*3+…*xk*). There are three cases.

Page 208
Case 1: *k*=1.
In this case we introduce new variables *yi* and *zi* and replace *Ci* with

$$B_i=(x_1+y_i+z_i)(x_1+y_i+\overline{z}_i)(x_1+\overline{y}_i+z_i)(x_1+\overline{y}_i+\overline{z}_i).$$

Case 2: *k*=2.
In this case we introduce a new variable *yi* and replace *Ci* with

$$B_i=(x_1+x_2+y_i)(x_1+x_2+\overline{y}_i).$$

Case 3: *k*=3.
In this case we replace *Ci* with *Bi*=*Ci*. Thus we make no change.
Case 4: *k*>3.
In this case we introduce new variables $y_{i_1}, y_{i_2}, \ldots, y_{i_{k-3}}$ and replace *Ci* with

$$B_i = (x_1 + x_2 + y_{i_1})(\overline{y}_{i_1} + x_3 + y_{i_2})(\overline{y}_{i_2} + x_4 + y_{i_3}) \cdots (\overline{y}_{i_{k-3}} + x_{k-1} + x_k)$$

It is routine to verify for each of Cases 1, 2, 3, and 4, that *Bi* satisfies (a) and (b), see Exercise 9.5.2. We still must show that

$$B' = B1B2B3 \ldots Bm$$

can be constructed with a polynomial time algorithm. If *Ci*=(*x*1+*x*2+*x*3+ ...+*xk*) then *Bi* contains at most 4*k* clauses of three variables and at most *k*+1 new variables were introduced. Because *k*≤*n,* we conclude that to construct *B′,* at most 4*mn* new clauses of three variables are needed, and at most (*n*+1)*m* new variables are introduced. Both are polynomial in the size of the instance of Sat. Consequently we can construct *B′* in polynomial time.

Given a graph *G,* a *k*-element subset $K \subseteq V(G)$ of vertices is a called a *vertex cover* of size *k* if each edge of *G* has at least one end in *K*. The Vertex Cover decision problem is:

Problem 9.4: Vertex Cover

Instance: a graph *G* and positive integer *k*.

Question: does *G* have a vertex cover of size at most *k*?

THEOREM 9.6 Vertex Cover is NP-*complete.*

PROOF It is easy to see that Vertex Cover is in NP, for if *K* is a purported vertex cover of the graph *G* of size *k,* then we simply check each edge of *G* to see

that at least one endpoint is in *K*. There are *ε* edges to check so this takes time *O(ε)* and we can check in time |*K*|≤*n* whether or not |*K*|≤*k*.

We now give a polynomial transformation from 3-Sat to Vertex Cover. Let *B*=*C*1*C*2...*Cm* be a boolean expression over *U*={*u*1,*u*2,...,*un*} in which each clause is a sum of exactly three variables. Thus for *i*=1,2,...,*n,* *Ci*= (*xi*+*yi*+*zi*) for some $z_i \in U \cup \overline{U}$, where $\bar{U}$={$\bar{u}$1,$\bar{u}$2,...,$\bar{u}$n}. We construct a graph *G* on the vertex set

$$V = U \cup \overline{U} \cup W,$$

where $W = \cup_{i=1}^{m} \{a_i, b_i, c_i\}$. The edge set of *G* is the union of the edges of *m* subgraphs *Hi, i*=1,2,...,*m,* where *Hi* is the subgraph shown in Figure 9.17. It consists of a triangle *(ai,bi,ci),* edges from *ai, bi, ci* to the variables contained in the clause, and edges connecting the variables to their complements. *G* has 2*n*+3*m* vertices and *n*+6*m* edges and hence can be built in polynomial time. Choose *k*=*n*+2*m* to obtain an instance of the Vertex Cover problem for the graph *G* constructed.

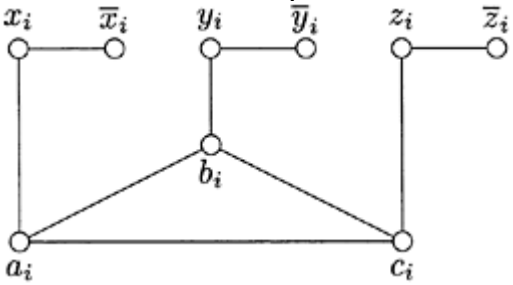


FIGURE 9.17
Subgraph *Hi* corresponding to clause (*xi*+*yi*+*zi*)

We show that *B* has a satisfying truth assignment if and only if *G* has a vertex cover *K* of size *k*=*n*+2*m*. If *t* is a truth assignment, such that

$$t(B) = \textbf{true},$$

then *t* must assign at least one variable *xi, yi* or *zi* to be **true** in clause *Ci*. Assume it is *xi*. As *xi* is adjacent to exactly one vertex, *ai,* in the triangle *{ai,bi,ci},* it follows that *{xi,bi,ci}* is a vertex cover of *Hi,* and hence

$$K = \cup_{i=1}^{m} \{x_i, b_i, c_i\}$$

is a vertex cover of size *k* for *G*. An example is given in Figure 9.18.

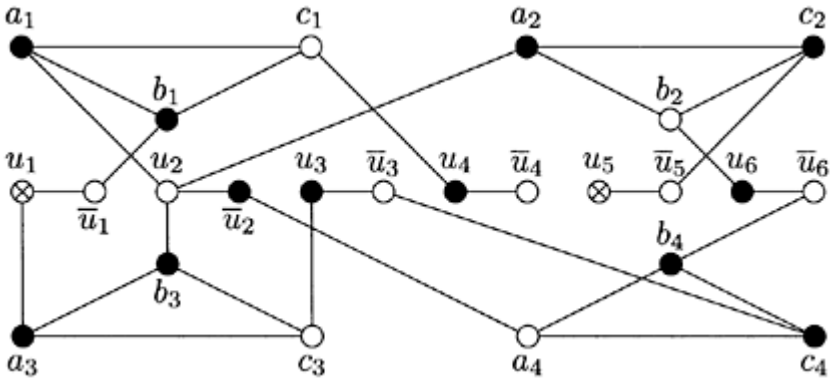


FIGURE 9.18
Graph *G* **corresponding to the boolean expression** $B=(u2+\bar{u}1+n4)(u2+u6+\bar{u}5)(u1+u2+u3)(\bar{u}2+\bar{u}6+\bar{u}3)$. **A vertex cover is** $K=\{u4,a1,b1,u6,a2,c2,u3,a3,b3,\bar{u}2,b4,c4i,u1,u5\}$ **and** $u3=u4=u6=$**true,** $u2=$**false,** *u***1,** *u***5, assigned arbitrarily is a truth assignment satisfying** *B.*

Conversely suppose that *K* is a vertex cover of size $k=n+2m$ of *G*. Then *K* must include at least one end of each of the *n* edges $\{ui,\bar{u}i\}$, $i=1,2,\ldots,n$, accounting for at least *n* vertices in *K*. Also *K* must cover the edges of each triangle *(aj,bj,cj),* and thus must contain at least two of *{aj,bj,cj},* for each $j= 1,2,\ldots,m$. This accounts for $2m$ more vertices, for a total of $n+2m=k$ vertices. Hence *K* must contain exactly one of the endpoints of each edge $\{ui,\bar{u}i\}$, for $i=1,2,\ldots, n$, and exactly two of *aj, bj, cj,* for each $j=1,2,\ldots, m$, corresponding to clause *Cj*. For each clause *Cj,* there is exactly one vertex *aj, bj* or *cj* of the triangle which is not in *K*. Call it *dj*.

Choose the unique variable of $U \cup \overline{U}$ adjacent to di, and assign it **true**. Then at least one variable in each clause *Cj* has been assigned the value **true,** giving a truth assignment that satisfies *B*. Any remaining unassigned variables in *U* can be assigned **true** or **false** arbitrarily.

THEOREM 9.7 HamCycle is NP*-complete.*

PROOF Let *G* be a graph. Given an ordering $u1,u2,\ldots, un$ of vertices of *G* we can check whether $(u1,u2,u3,\ldots,un)$ is a hamilton cycle in polynomial time. Thus HamCycle is in NP. To show that HamCycle is NP-complete we transform from Vertex Cover.

Let *G* and *k* be an instance of Vertex Cover, where *k* is a positive integer. We will construct a graph *G′* such that *G′* has a hamilton cycle if and only if *G* has a vertex cover $K=\{x1,x2,\ldots,xk\}$ of size *k*. The graph *G′* will have $k+12m$

vertices $V(G') = K \cup \{(u,e,i) : u \in V(G)$ is incident to $e \in E(G)$ and $i=1,2,\ldots,6\}$, where $m=|E(G)|$. The edges of *G′* are of three types.

Type 1 edges of *G′*

The type 1 edges are the $14m$ edges among the subgraphs *He,* $e \in E(G)$. We display *He,* where $e=uu$ in Figure 9.19.

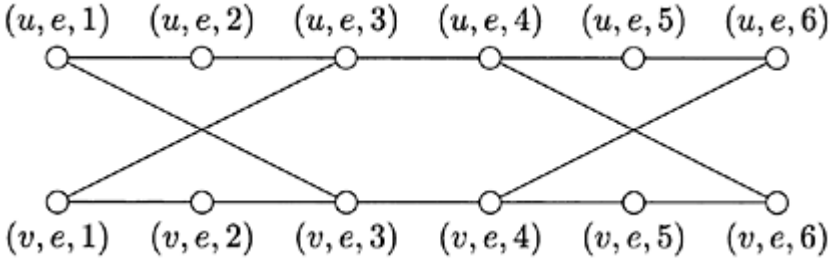


FIGURE 9.19
The subgraph *He,* **where** $e=uu$.

Type 2 edges of G′

For each vertex *u* of *G* choose a fixed but arbitrary ordering $e_{v_1}, e_{v_2}, \ldots, e_{v_3}$ of the $d=DEG(u)$ edges incident to *u*. The type 2 edges of *G′* corresponding to *u* are:

$$\{\{(v,e_{v_i},6),(v,e_{v_{i+1}},1)\} : i = 1,2,\ldots,d-1\}$$

Type 3 edges of *G′*

The type 3 edges of *G′* are:

$$\{\{x_i,(v,e_{v_1},j)\} : x_i \in K, v \in V(G), j \in \{1,6\}\}$$

The subgraph of *G′* corresponding to the edges incident to a vertex *u* in *G* is illustrated in Figure 9.20. Before proving that *G* has a vertex cover of size *k* if and only if *G′* has a hamilton cycle *C*=*u*1,*u*2,...,*un,* we make five observations.
1. C must enter and exit the subgraph *He, e*=*uu* from the four corners (*u,e,*1), (*u,e,*6), (*u,e,*1), (*u,e,*6).
2. If *C* enters *He* at (*u,e,*1), it must exit at (*u,e,*6) and either pass through all the vertices of *He* or only those vertices with first coordinate *u*. (In the first case as we shall see, *u* will be in the vertex cover of *G,* and in the latter case both *u* and *u* will be in the vertex cover of *G*.)
3. If *C* enters *He* at (*u,e,*1), it must exit at (*u,e,*6) and either pass through all the vertices of *He* or only those vertices with first coordinate *u*. (In the first case as we shall see, *u* will be in the vertex cover of *G,* and in the latter case both *u* and *u* will be in the vertex cover of *G*.)

Page 212

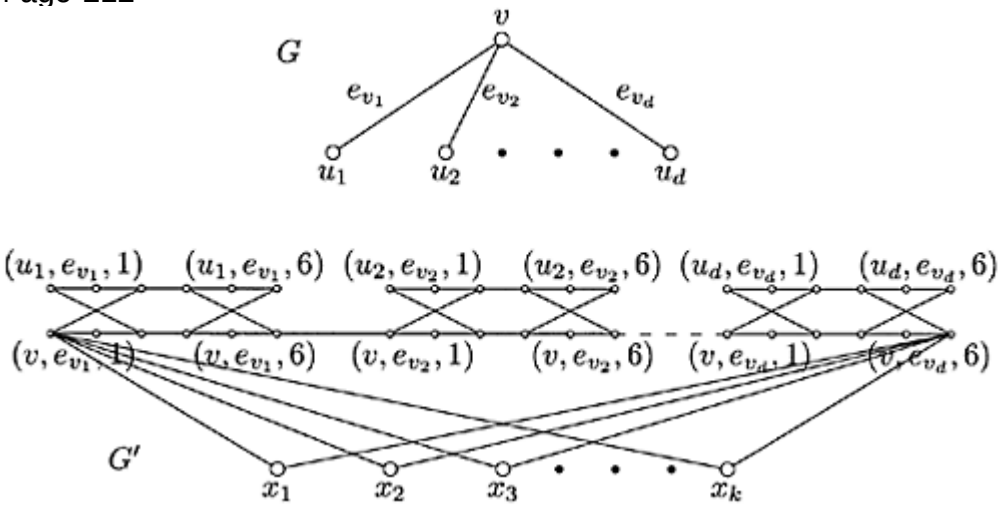


FIGURE 9.20
Subgraph of *G′* corresponding the the edges incident to *u* in *G*

4. The vertices {*x*1,*x*2,...,*xk*} divide *C* into paths. Thus we may assume, relabeling the vertices *x*1, *x*2,..., *xk* if necessary, that *C*=*P*1*P*2...*Pk* where *Pi* is an *xi* to *xi*+1 path, where *xk*+1=*xk*.
5. Let *ui* be such that *xi* is adjacent to *(ui,e,j)* in *Pi* where *j*=1 or 6. Then *Pi* contains every vertex *(ui,e′,h)* where e is incident to *u*.
We claim that the *k* vertices *u*1, *u*2,..., *uk* selected in observation 5 are a vertex cover of *G*. This is because the hamilton cycle *C* must contain all vertices of each of the subgraphs *He* for each $e \in G$; and when *He* is traversed by *C,* it is traversed by some *Pi* in *C* and that *Pi* selects an endpoint *ui* of *e*.
Conversely, suppose $K = \{v_1, v_2, \ldots, v_k\} \subseteq V(G)$ is a vertex cover of *G,* of size *k*. To construct a hamilton cycle *C* of *G′,* choose for each edge $e \in E(G)$ the edges of *He* specified in Figure 9.21 (a), (b), or (c) depending on whether $\{u, v\} \cup K$ equals *{u}, {u,u},* or *{u},* respectively. (One of these must occur, because *K* is a vertex cover.) Also include the edges

$$\{(v_i, e_{v_i}, 6), (v_i, e_{v_i}, 1)\}, i = 1, 2, \ldots, k,$$

the edges

$$\{x_i, (v_i, e_{v_i}, 1)\}, i = 1, 2, \ldots, k,$$

and the edges

$$\{x_{i+1}, (v_i, e_{v_i}, 1)\}, i = 1, 2, \ldots, k, \text{ where } v_{k+1} = v_1.$$

It is an easy exercise to verify that the included edges form a hamilton cycle in *G′;* see Exercise 9.5.5.

Page 213

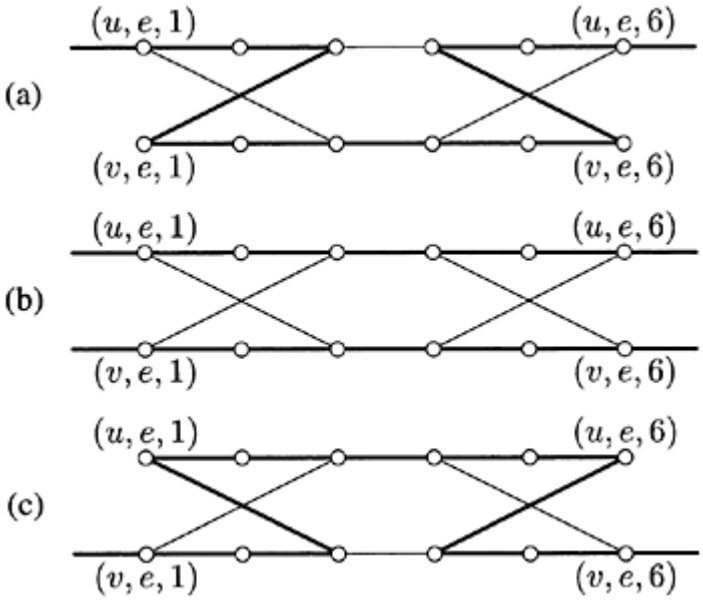


FIGURE 9.21
The three possible ways that a Hamilton cycle can traverse the subgraph *He*, **corresponding to the cases for** *e*=*{u,u}* **in which (a)** *e*∩*K*=*{u}*, **(b)** *e*∩*K*=*{u,u}*, **and (c)** *e*∩*K*=*{u}*.

Exercises

9.5.1 Consider the boolean expression

$$B = (x_1 + \overline{x}_2 + x_3 + \overline{x}_6)(x_2 + \overline{x}_4)(x_5)(x_2 + \overline{x}_4 + x_5)$$

Find a boolean expression equivalent to *B* in which each clause uses only three variables.

9.5.2 Show for each Case 1, 2, 3, and 4 in Theorem 9.5 that the pair *Bi, Ci* satisfies

(a) *Bi* is a product of clauses that use at most three variables in *U′*.

(b) *Bi* is satisfiable if and only if *Ci* is.

9.5.3 Consider the boolean expression

$$B = (x + y + z)(x + y + \overline{z})(w + \overline{x} + z)(w + \overline{x} + \overline{z})(\overline{w} + \overline{x} + z)$$
$$(\overline{w} + \overline{x} + \overline{z})(w + \overline{y} + z)(w + \overline{y} + \overline{z})(\overline{w} + \overline{y} + z)(\overline{w} + \overline{y} + \overline{z})$$

(a) Show that there is no truth assignment that satisfies *B*.

(b) Construct the graph *G* in Theorem 9.6 that corresponds to *B*.

(c) Show that *G* does not have a vertex cover of size 25.

9.5.4 Verify the five observations in Theorem 9.7.

9.5.5 Verify that the included edges in the converse part of Theorem 9.7, do indeed form a hamilton cycle.

9.6 The traveling salesman problem

The traveling salesman problem (TSP) is very closely related to the HamCycle problem. A salesman is to visit *n* cities *u*1, *u*2,..., *un*. The cost of traveling from *ui* to *uj* is *W(uiuj)*. Find the cheapest tour which brings him back to his starting point. Figure 9.22 shows an instance of the TSP problem. It is a complete graph *Kn* with positive integral weights on the edges. The problem asks for a hamilton cycle of minimum cost.

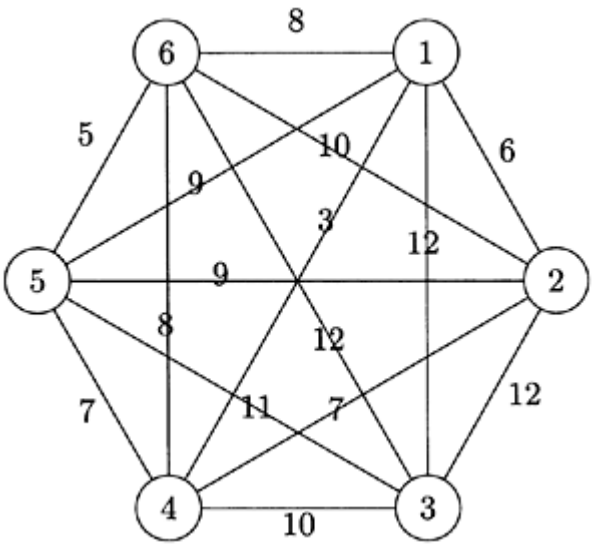


FIGURE 9.22
An instance of the TSP problem.

It is easy to show that the HamCycle problem can be reduced to the TSP problem. In order to do this, we first must phrase it as a decision problem.

Problem 9.5: TSP Decision

Instance: a weighted complete graph *Kn,* and an integer *M,*
Question: does *Kn* have a hamilton cycle of cost ≤*M*?

We can then find the actual minimum by doing a binary search on the range of values $n \leq M \leq nW\text{max}$, where *W*max is the maximum edge-weight. Suppose that we had an efficient algorithm for the TSP Decision problem. Let *G* be any graph on *n* vertices which we want to test for hamiltonicity. Embed *G* in a compete graph *Kn,* giving the edges of *G* weight 1, and the edges of $\overline{G}$ weight 2.

Now ask whether *G* has a TSP tour of cost ≤*n*. If the answer is yes, then *G* is hamiltonian. Otherwise *G* is non-hamiltonian.

Since HamCycle is NP-complete, we conclude that the TSP Decision problem is at least as hard as an NP-complete problem. In a certain sense, it is harder than the NP-complete problems, since the edge weights *W(uiuj)* are not bounded in size. So it may take many steps just to add two of the weights. However, if we limit the size of the weights to the range of numbers available on a computer with a fixed word length, then the TSP Decision problem is also NP-complete. It is easy to see that TSP Decision $\in \mathbf{NP}$, since we can write down the sequence of vertices on a cycle *C* of cost ≤*M* and verify it in *n* steps.

One way to approximate a solution is similar to the crossover technique. Choose a hamilton cycle *C* in *Kn* arbitrarily. For each edge $uv \in C$, search for an edge $wx \in C$ such that *W(uu)+W(wx)>W(uw)+W(ux)*. If such an edge exists, re-route *C* as shown in Figure 9.23. Repeat until no improvement can be made. Do this for several randomly chosen starting cycles, and take the best as an approximation to the optimum.

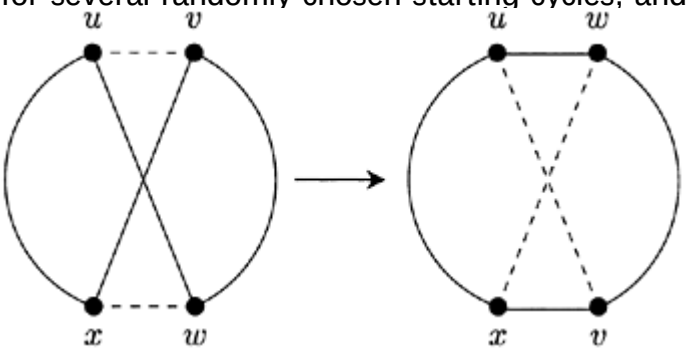


FIGURE 9.23
Re-routing a TSP tour

The cycle *Q=(u,u,x,w)* is similar to a crossover. In general, if *Q* is any cycle such that $C \oplus Q$ is a hamilton cycle, and *W(C∩Q)>W(Q−C)*, then $C \oplus Q$ will be a TSP tour of smaller cost than *C*. We can search for crossovers *Q* containing up to *M* edges, for some fixed value *M,* and this will provide a tour which may be

close to the optimum. How close does it come to the optimum?
It is possible to obtain a rough estimate of how good a tour *C* is, by using a minimum spanning tree algorithm. Let *C** be an optimum TSP tour. For any vertex *u, C*−u* is a spanning tree of *Kn−u*. Let *Tu* be a minimum spanning tree of *Kn−u*. Then *W*(*C*−u*) *W(Tu)*. Given the path *C*−u,* we must add back two edges incident on *u* to get *C**. If we add two edges incident on *u* to *Tu,* of minimum possible weight, we will get a graph T_v^*, such that $W(C^*) \geq W(T_v^*)$. For example, Figure 9.24 shows a minimum spanning tree *T*3, of *Kn*−3 for the

instance of TSP shown in Figure 9.22.

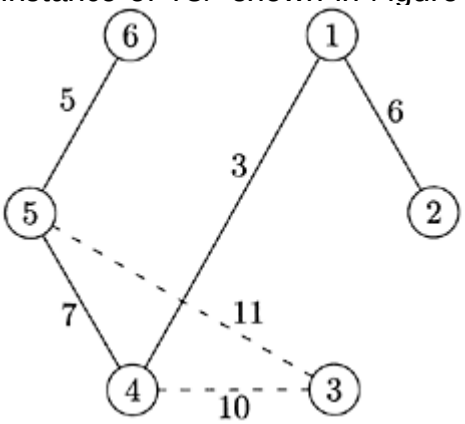


FIGURE 9.24
A minimum spanning tree *T*3, plus two edges
The two edges incident on vertex 3 that we add to *T*3 have weights 10 and 11 in this case. We thus obtain a bound $W(C^*) \geq W(T_3^*) = 42$. We do this for each vertex *u,* and choose the *maximum* of the bounds obtained. This is called the *spanning tree bound* for the TSP:

$$W(C^*) \geq \text{MAX}_v W(T_v^*).$$

Exercises
9.6.1 Work out the spanning tree bound for the TSP instance of Figure 9.22.
9.6.2 Find a TSP tour *C* in the graph of Figure 9.22 by re-routing any starting cycle, until no more improvement is possible. Compare the weight of *C* with the result of Exercise 9.6.1.
9.6.3 Construct all possible re-routing patterns (crossovers) containing three or four edges of *C*.

9.7 The ΔTSP

Distances measured on the earth satisfy the *triangle inequality,* namely, for any three points *X, Y,* and *Z,* DIST*(X,Y)*+DIST*(Y,Z)*≥DIST*(X,Z)*. The *triangle traveling salesman problem,* denoted ΔTSP, refers to instances of the TSP

satisfying this inequality. When the triangle inequality is known to hold, additional methods are possible.
THEOREM 9.8 Let Knbe an instance of the ΔTSP, and let G be any Eulerian spanning subgraph of Kn. If C is an optimum TSP tour, then W(C*)≤W(G)*.
PROOF Consider an Euler tour *H* in *G* starting at any vertex. The sequence of vertices traversed by *H* is $v_{i_0}, v_{i_1}, v_{i_2}, v_{i_3}, \cdots$ If *G* is a cycle, then *H* is a hamilton cycle, so that *W(C*)≤W(G),* and we are done. Otherwise, *H* repeats one or more vertices. Construct a cycle *C* from *H* by taking the vertices in the order that they appear in *H,* simply ignoring repeated vertices. Since *G* is a spanning subgraph of *Kn,* all vertices will be included in *C*. For example, if *G* is

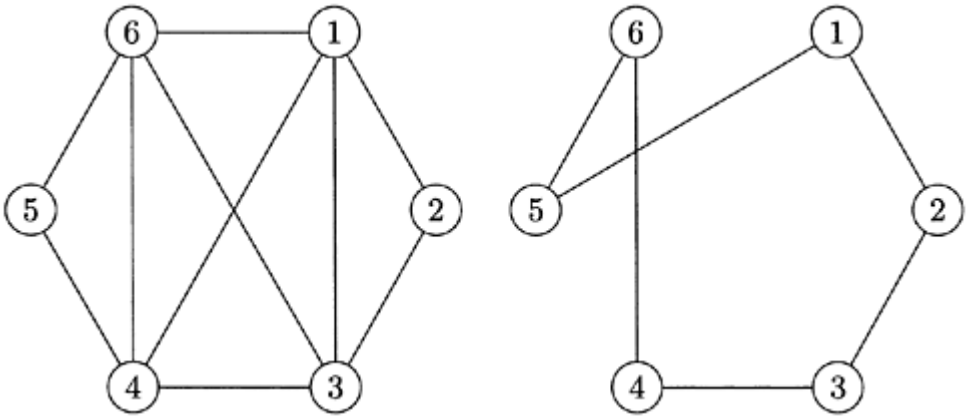


FIGURE 9.25
An Eulerian graph *G* **and TSP tour** *C*

the graph of Figure 9.25, and *H* is the Euler tour (1,2,3,4,6,1,3,6,5,4), then the cycle *C* obtained is (1,2,3,4,6,5). Because of the triangle inequality, it will turn out that $W(C) \leq W(G)$. Let the cycle obtained be $C=(u1,u2,\ldots,un)$, and suppose that the Euler tour *H* contains one or more vertices between *uk* and *uk*+1. Without loss of generality, suppose that there are just three vertices *x, y, z* between *uk* and *uk*+1. See Figure 9.26. Then because of the triangle inequality, we can write

$$W(ukx)+W(xy) \geq W(uky),$$
$$W(uky)+W(yz) \geq W(ukz),$$

and

$$W(ukz)+W(zuks+1) \geq W(ukuk+1).$$

Thus

$$W(ukuk+1) \leq W(ukx)+W(xy)+W(yz)+W(zuk+1).$$

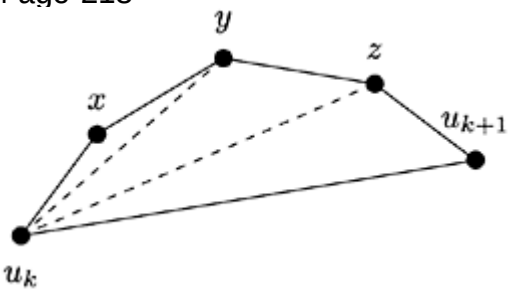


FIGURE 9.26
Applying the triangle inequality

The left side of the inequality contributes to *W(C)*. The right side contributes to *W(H)*. It follows that $W(C) \leq W(G)$, for any Eulerian *G*.

Notice that the particular cycle *C* obtained from *G* depends on the Euler tour *H* chosen, so that the graph *G* will give rise to a number of different hamilton cycles *C*. In particular, we could construct *G* from a minimum spanning tree *T,* by simply doubling each edge. This gives an Eulerian multigraph *G*. The method used in the theorem will also work with multigraphs, so we conclude that $W(C^*) \leq 2W(T)$. This is called the *tree algorithm* for the TSP.

LEMMA 9.9 The tree algorithm produces a cycle of cost at most twice the optimum.

PROOF Let *C* be the cycle obtained by the tree algorithm, let *C** be an optimum cycle, and let *T* be a minimum spanning tree of the instance for ΔTSP. Since *C** is a spanning subgraph of *Kn,* we conclude that $W(C^*) > W(T)$. But we know that $W(C) \leq 2W(T) < 2W(C^*)$.

9.8 Christofides' algorithm

Christofides found a way to construct an Eulerian subgraph of smaller weight than $2W(T)$. Let *Kn* be an instance of the ΔTSP, and let *T* be a minimum spanning tree. Let $X \subseteq V(K_n)$ be the vertices of *T* of odd degree. *X* contains an even number of vertices. The subgraph of *Kn* induced by *X* is a complete subgraph. Let *M* be a perfect matching in *X* of minimum weight. For example, Figure 9.27 shows a minimum spanning tree for the graph of Figure 9.22, together with a

minimum-weight matching *M,* shown as dashed lines.

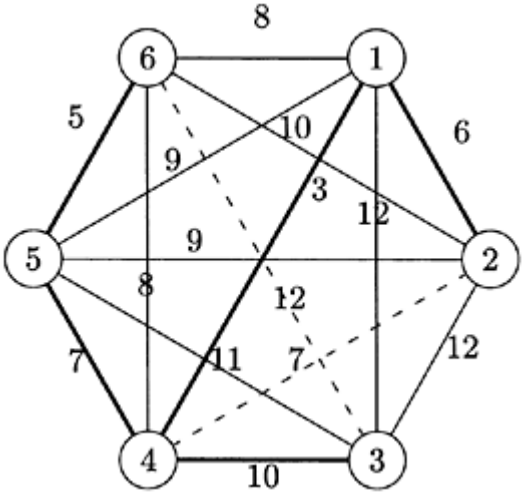


FIGURE 9.27
Christofides' algorithm

This gives a graph $G=T+M$ which is Eulerian. It is quite possible that G is a multigraph. We now find an Euler tour in G and use it to construct a TSP tour C of cost at most $W(T)+W(M)$. This is called *Christofides' algorithm*.

THEOREM 9.10 Let C be the TSP tour produced by Christofides' algorithm and let C be an optimum tour. Then*

$$W(C) \leq \frac{3}{2}W(C^*).$$

PROOF Let $u1, u2, \ldots, u2k$ be the vertices of odd degree, and suppose that they appear on C^* in that order. This defines two matchings,

$$M1=\{u1u2, u3u4, \ldots\}$$

and

$$M2=\{u2u3, u4u5, \ldots, u2ku1\}.$$

See Figure 9.28. If M is the minimum weight matching, we conclude that $W(M1), W(M2) \geq W(M)$. The portion of C^* between ui and $ui+1$ satisfies

$$W(C^*[ui, ui+1]) \geq W(uiui+1),$$

by the triangle inequality. Therefore

$$W(C^*) \geq W(M1)+W(M2) \geq 2W(M),$$

---

or

$$W(M) \leq \frac{1}{2}W(C^*).$$

The cycle C found by Christofides's algorithm satisfies

$$W(C) \leq W(T)+W(M) < W(C^*)+\frac{1}{2}W(C^*),$$

because $W(T)<W(C^*)$. It follows that

$$W(C) \leq \frac{3}{2}W(C^*).$$

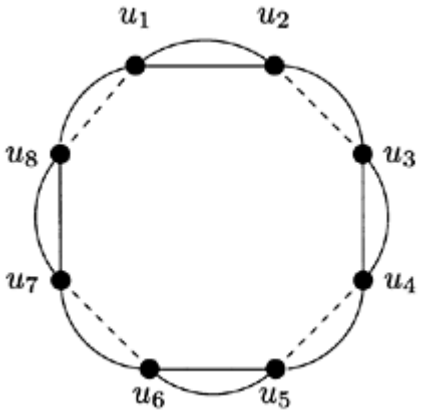


FIGURE 9.28
Two matchings *M*1 and *M*2

Thus, Christofides' algorithm always comes within 50% of the optimum.

Exercises

9.8.1 Use the tree algorithm to find a TSP tour for the graph of Figure 9.22.

9.8.2 Solve the same TSP instance using Christofides' algorithm. Compare the values found for *W(C), W(T),* and $W(T+M)$.

9.8.3 Solve the ΔTSP instance of Figure 9.29, using Christofides' algorithm. Compute the spanning tree bound as well.

Page 221

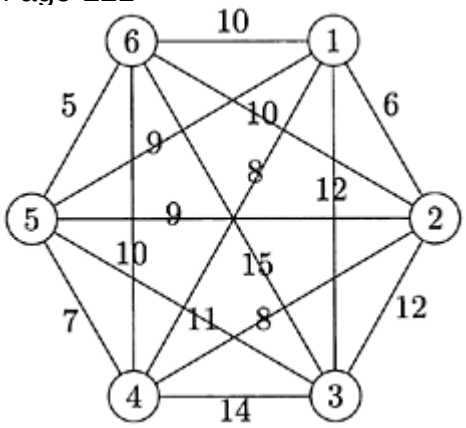


FIGURE 9.29
An instance of ΔTSP

9.9 Notes

An excellent survey of hamiltonian graphs appears in BERMOND [15]. The hamilton closure and the Bondy-Chvátal theorem are from BONDY and MURTY [19]. The extended multi-path algorithm is from KOCAY [77]. A classic book on the theory of NP-completeness is the text by GAREY and JOHNSON [50]. A very readable proof of Cook's theorem, that Satisfiability is NP-complete, appears in PAPADIMITRIOU and STEIGLITZ [94], which also contains an excellent section on Christofides' algorithm. The book by CHRISTOFIDES [26] has an extended chapter on the traveling salesman problem. The book LAWLER, LENSTRA, RINNOOY KAN and SHMOYS [83] is a collection of articles on the traveling salesman problem.

Page 222

This page intentionally left blank.

Page 223

10

Digraphs

10.1 Introduction

Directed graphs have already been introduced in the Chapter 8. If *G* is a digraph and *u,* $v \in V(G)$, we write $u \longrightarrow v$

to indicate that the edge *uu* is directed from *u* to *u*. The *in-edges* at *u* are the edges of the form *(u,u)*. The *in-degree* of *u* is *d−(u),* the number of in-edges. Similarly the *out-edges* at *u* are all edges of the form *(u,u)* and the *out-degree d+(u)* is the number of out-edges at *u*. The degree of *u* is

$$\mathrm{DEG}(u) = d^+(u) + d^-(u).$$

Given any undirected graph *G,* we can assign a direction to each of its edges, giving a digraph called an *oriented graph*. A digraph is *simple* if it is an orientation of a simple graph. A digraph is *strict* if it has no loops, and no two directed edges have the same endpoints. A strict digraph can have edges *(u,u)* and *(u,u),* whereas an oriented graph cannot.

Digraphs have extremely wide application, for the social sciences, economics, business management, operations research, operating systems, compiler design, scheduling problems, combinatorial problems, solving systems of linear equations, and many other areas. We shall describe only a few fundamental concepts in this chapter.

10.2 Activity graphs, critical paths

Suppose that a large project is broken down into smaller tasks. For example, building a house can be subdivided into many smaller tasks: dig the basement, install the sewer pipes, water pipes, electricity, pour the basement concrete, build

the frame, floor, roof, cover the roof and walls, install the wiring, plumbing, heat-ing, finish the walls, etc. Some of these tasks must be done in a certain order—the basement must be dug before the concrete can be poured, the wiring must be installed before the walls can be finished, etc. Other tasks can take place at the same time, (e.g., the wiring and plumbing can be installed simultaneously). We can construct a directed graph, called an *activity graph,* to represent such projects. It has a starting node s, where the project begins, and a completion node *t,* where it is finished. The subtasks are represented by directed edges. The nodes represent the beginning and end of tasks (the synchronization points between tasks). Figure 10.1 shows an example of an activity graph. Each task takes a certain estimated time to complete, and this is represented by assigning each edge *uu* a weight WT*(uu),* being the amount of time required for that task.

What is the *minimum amount of time* required for the entire project? It will be the length of the *longest* directed path from start to completion. Any longest directed path from *s* to *t* is called a *critical path*. Figure 10.1 shows a critical path in an activity graph.

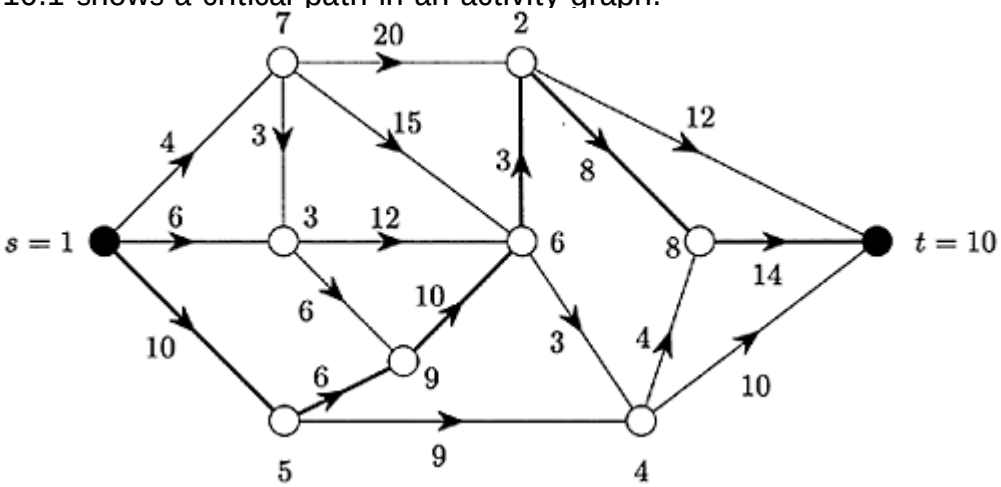


FIGURE 10.1
An activity graph

Notice that an activity graph must have no directed cycles. For if a directed cycle existed, it would be impossible to complete the project according to the constraints. Thus, activity graphs are *acyclic* digraphs. Activity graphs are applicable to any large project, such as building construction, business projects, or factory assembly lines.

The *critical path method* **(CPM)** is a technique for analyzing a project acccording to the longest paths in its activity graph. In order to find a longest path from *s*

to *t,* we proceed very much as in Dijkstra's algorithm (Chapter 2), which builds a spanning tree, rooted at s, of shortest paths. To find longest paths instead, the algorithm builds an out-directed spanning tree, rooted at *s,* of longest directed paths from *s* to each vertex *u*. We store a value *T[u]* for each *u,* being the earliest time

at which tasks starting from *u* can begin. *T[u]* is the length of a longest *su*-path. When the algorithm completes, the critical path is the unique path in the spanning tree to vertex *t*. Notice that in Figure 10.1, the edge from vertex 1 to 3 has length 6, but that the path (1,7,3) has the longer length of 7. In order for the algorithm to correctly choose the longer path, it must be sure to assign *T*(7) before *T*(3). Thus, the vertices must be taken in a certain order. For every edge *(u,u), T(u)* must be computed before *T(u)*.

10.3 Topological order

A *topological ordering* of an acyclic digraph *G* is a permutation *o* of

$$V(G)=\{1,2,\ldots,n\}$$

such that $o(u)<o(u)$ whenever $u \longrightarrow v$. Thus all edges are directed from smaller to higher vertex numbers. Notice that only acyclic digraphs have topological orderings, since a directed cycle cannot be ordered in this way. Topological orderings are easy to find. We present both a breadth-first and a depth-first algorithm. Algorithm 10.3.1 is the the breadth-first algorithm. The topological order is built on the queue. Algorithm 10.3.1 begins by placing all vertices with in-degree 0 on the queue. These are first in the topological order. *InDegree[u]* is then adjusted so that it counts the in-degree of *u* only from vertices not yet on the queue. This is done by decrementing *InDegree[u]* according to its in-edges from the queue. When *InDegree[u]*=0, *u* has no more in-edges, so it too, is added to the queue. When all *n* vertices are on the queue, the vertices are in topological order. Notice that if *G* has a directed cycle, none of the vertices of the cycle will ever be placed on the queue. In that case, the algorithm will terminate with fewer than *n* vertices on the queue. This is easy to detect. Computing *InDegree[u]* takes $\sum_v d^-(v) = \varepsilon$ steps. Each vertex is placed on the queue exactly once, and its $d^+(u)$ out-edges are taken in turn, taking $\sum_u d^+(u) = \varepsilon$ steps. Thus the complexity of the algorithm is $O(n+\varepsilon)$.

Algorithm 10.3.1: BFTOPSORT*(G,n)*

comment:
- Breadth-first topological sort of G
- $InDegree[v]$ is the in-degree of vertex v, an array
- $ScanQ[k]$ is the k^{th} vertex on a queue, an array
- $Qsize$ is the number of points on $ScanQ$

$Qsize \leftarrow 0$
$k \leftarrow 1$
for $u \leftarrow 1$ **to** n
do
- compute $InDegree[v]$
- **if** $InDegree[v] = 0$
  - **then**
    - $Qsize \leftarrow Qsize + 1$
    - $ScanQ[Qsize] \leftarrow v$

while $k \leq Qsize$
do
- $u \leftarrow ScanQ[k]$
- **for each** v such that $u \longrightarrow v$
  - **do**
    - $InDegree[v] \leftarrow InDegree[v] - 1$
    - **if** $InDegree[v] = 0$
      - **then**
        - $Qsize \leftarrow Qsize + 1$
        - $ScanQ[Qsize] \leftarrow v$
        - **if** $Qsize = n$
          - **then go to** 1
- $k \leftarrow k + 1$

if $Qsizie<n$ **then** *G* contains a directed cycle

Algorithm 10.3.2 is the depth-first topological sort algorithm and is easier to program, but somewhat subtler. It calls the recursive Procedure DFS(), and we assume that Procedure DFS() has access to the variables of Algorithm DFTOPSORT() as globals.

When Procedure DFS(u) is called from DFTOPSORT(), it builds a rooted tree, directed outward from the root *u*. *DFNum[u]* gives the order in which the vertices are visited. The depth-first search does a traversal of this tree, using a recursive call to visit all descendants of *u* before *u* itself is assigned a number *NUM[u],* its rank in the topological order. Thus, if *G* is acyclic, all vertices that can be reached on directed paths out of *u* will be

ranked before *u* itself is ranked. Thus, for every edge *(u,u),* the numbering will satisfy *NUM[u]*<*NUM[u]*. The first vertex numbered is assigned a *Rank* of *n*. The variable *Rank* is then decremented. So the vertices are numbered 1 to n, in topological order. It is obvious that the complexity of the algorithm is $O(n+\varepsilon)$.

Page 227

Algorithm 10.3.2: DFTOPSORT*(G, n)*

comment:
- Depth-first topological sort of G
- $DFNum[v]$ is the DF-numbering assigned to the vertex v
- $NUM[v]$ is the topological numbering of the vertex v
- $DFCount$, $Rank$ are counters

procedure DFS*(u)*
comment: extend the depth-first search to vertex *u*
DFCount←*DFCount*+1
DFNum[u]←*DFCount*
for each *w* such that $v \longrightarrow w$
do if *DFNum[w] =0*
then DFS*(w)*
NUM[u]←*Rank*
Rank←*Rank*−*1*

main
for *u*←1 **to** *n*
do $NUM[u] \leftarrow 0$; $DFNum[u] \leftarrow 0$
DFCount←0
Rank←*n*
for *u*←1 **to** *n*
do if *DFNum[u]*=0
then DFS*(u)*

The depth-first topological sort does not provide the vertices on a queue in sorted order. Instead it assigns a number to each vertex giving its rank in the topological order. If we need the vertices on a queue, as we likely will, we can construct one from the array *NUM* by executing a single loop.

for *u*←1 **to** *n* **do** *ScanQ[NUM[u]]*←*u*

This works because *NUM* is a permutation of the numbers 1 to *n*, and the loop computes the inverse of the permutation. Another method is to compute the inverse array during the DFS simultaneously with the *NUM* array.

What happens if the depth-first topological sort is given a digraph that is not acyclic? It will still produce a numbering, but it will not be a topological ordering. We will have more to say about this in Section 10.4.

Notice that DFS*(u)* may be called several times from Algorithm 10.3.2. Each time it is called, a rooted tree directed outward from the root is constructed. With undirected graphs, a DFS

Page 228

constructs a single rooted spanning tree of *G* (see Chapter 6). For directed graphs, a single out-directed tree may not be enough to span all of *G*. A spanning forest of rooted, out-directed trees is constructed.

We return now to the critical path method. Let *G* be an activity graph with *V(G)*={1, 2,..., *n*}, and suppose that the vertices have been numbered in topological order; that is, *u*<*u* whenever $u \longrightarrow v$. The start vertex is *s*=1. We set *T*(1)←0. We know that vertex 2 has an in-edge only from *s,* so *T*(2) is assigned the cost of the edge (1, 2). In general, *u* can have in-edges only from vertices 1,..., *u*−1, and we can take *T(u)* to be

$$T(v) \leftarrow \text{MAX}\{T(u) + \text{WT}(uv) : u \longrightarrow v\}.$$

We also store an array *PrevPt,* where *PrevPt[u]* is the point previous to *u* on a directed *su*-path. If *T(u)* is computed to be *T(u)*+WT*(uu)* for some *u,* we simultaneously assign *PrevPt[u]*←*u*. When the algorithm completes, we can find the critical path by executing *w*←*PrevPt[w]* until *w*=0, starting with *w*=*t*(=*n*). The number of steps required to compute the longest paths once the topological sort has been completed is proportional to $n+\sum_v d^-(v) = O(n+\varepsilon)$.

The minimum time required to complete the project is *T(n)*. This can be achieved only if all tasks along the

critical path begin and end on time. These tasks are critical. There may be some slack elsewhere in the system, though, which can be used to advantage. The earliest time at which a node *u* in the activity graph can be reached is *T(u),* the length of the longest *su*-path. We could also compute the latest time at which node *u* must be reached if the project is to finish on time. This is *T(n)* minus the length of the longest directed path from *u* to *t*. Let *T′(u)* be the length of the longest directed path from *u* to *t*. We can compute this in the same way that *T(u)* is computed, but beginning with *t* instead of *s,* and working backward. Thus for each node *u,* we can find the two values *T(u)* and *T(n)−T′(u),* being the earliest and latest times at which node *u* can be reached. This slack time can create some flexibility in project management.

Exercises

10.3.1 Find a topological ordering of the activity graphs of Figures 10.1 and 10.2. Apply the critical path method to find the longest *su*-paths and *vt*-paths, for each *u*. Work out the earliest and latest times for each node *u*.

10.3.2 Program the breadth-first and depth-first topological sort algorithms. Test them on the graph of Figure 10.1.

10.3.3 Consider the recursive procedure *DFS(u)* defined above, applied to a directed graph *G*. Suppose that *DFS(u)* has just been called, and that *A(u)* is the set of all vertices which are ancestors of *u* (the path from *u* to the root contains the ancestors of *u*).

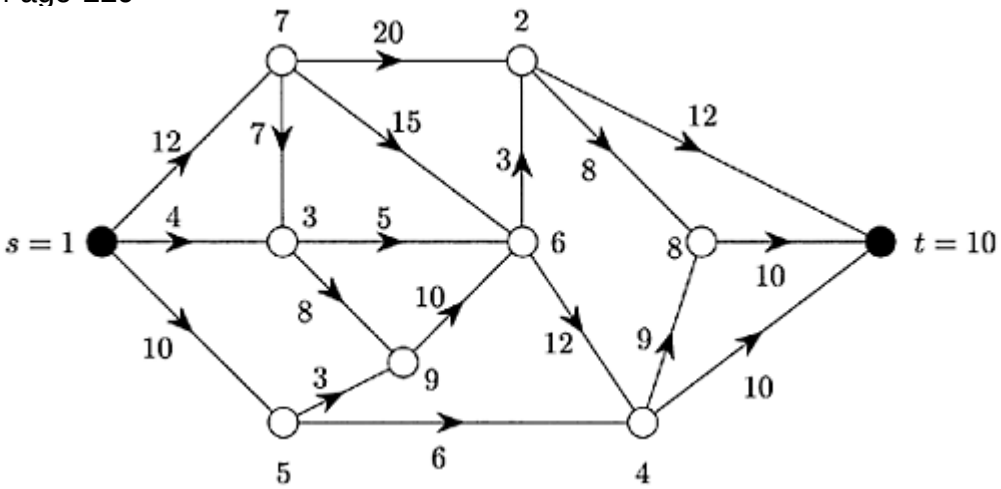


FIGURE 10.2
An activity graph

Suppose that *G−A(u)* contains a directed path from *u* to *w*. Prove that *w* will be visited before *DFS(u)* returns. Use induction on the length of the path from *u* to *w*.

10.4 Strong components

A digraph *G* is connected if every pair of vertices *u* and *u* is connected by a path. This need not be a directed path. The digraph *G* is *strongly connected* if every pair of vertices is connected by a *directed* path. Thus, if *G* is strongly connected, *G* contains both a *uu*-path and a *uu*-path, for every *u* and *u*. It follows that every vertex of *G* is contained in a directed cycle. A digraph which is strongly connected is said to be *strong*. Notice that a strong digraph does not have to be 2-connected. It may contain one or more cut-vertices.

By default, the complete digraph *K*1 is strong, since it does not have a pair of vertices. If *G* is acyclic, then the only strong subgraphs of *G* are the individual nodes. But if *G* contains any directed cycle, then *G* will contain one or more non-trivial strongly connected subgraphs. A subgraph *H* is a *strong component* of *G* if it is a maximal strongly connected subgraph; that is, *H* is strong, and *G* has no larger subgraph containing *H* which is also strong. Figure 10.3 shows a digraph *G* with four strong components. The edges of the strong components are indicated by thicker lines. Two of the strong components are single vertices, which are shaded black.

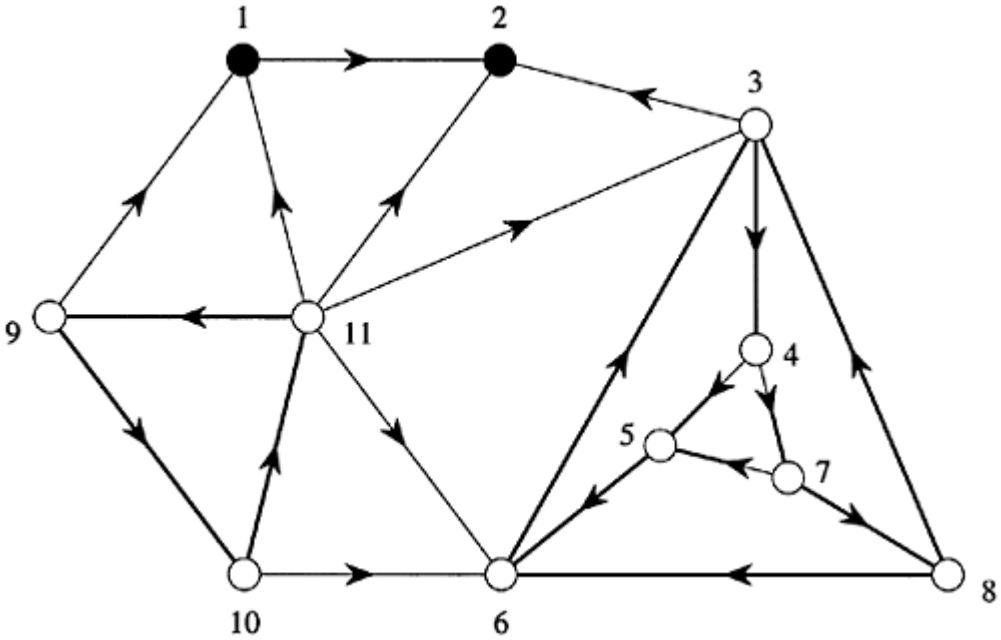


FIGURE 10.3
Strong components

Notice that every vertex of *G* is contained in exactly one strong component, but that some edges of *G* need not be contained in any strong component. Exercise 10.4.1. shows that this definition of strong components is well-defined.

If *G*1, *G*2,..., *Gm* are the strong components of *G,* we can construct a new digraph by contracting each strong component into a single vertex.

DEFINITION 10.1: Let *G*1, *G*2,..., *Gm* be the strong components of *G*. The *condensation* of *G* is the digraph whose vertices are *G*1, *G*2,..., *Gm,* and whose edges are all ordered pairs *(Gi,Gj)* such that *G* has at least one edge directed from a vertex of *Gi* to a vertex of *Gj*.

It is proved in Exercise 10.4.3 that the condensation is an acyclic digraph.

Exercises

10.4.1 Suppose that *H* is a strong subgraph of *G* such that *H* is contained in two larger strong subgraphs: $H \leq H1$ and $H \leq H2$, where *H*1 and *H*2 are both strong. Show that $H_1 \cup H_2$ is strong. Conclude that the strong components of *G* are well-defined.

10.4.2 Show that an edge *(u,u)* is contained in a strong component if and only if *(u,u)* is contained in a directed cycle.

10.4.3 Find the condensation of the digraph of Figure 10.3. Prove that the condensation of a digraph is always acyclic.

10.4.4 The *converse* of a digraph is obtained by reversing the direction of each edge. A digraph is *self-converse* if it is isomorphic to its converse. Find all self-converse simple digraphs on one, two, three, and four vertices.

10.4.5 Show that the condensation of the converse is the converse of the condensation.

10.4.6 Let *G* be a self-converse simple digraph, and let *G′* be the converse of *G*. Let θ be an isomorphism of *G* with *G′,* so that θ is a permutation of $V(G)=V(G')$. Prove that θ has at most one cycle of odd length. Find the possible cycle structures of θ when *G* has at most five vertices. Use this to find all the self-converse digraphs on five vertices.

In this section, we present an algorithm to find the strong components of a digraph *G*. It is based on a depth-first search. It is very similar to the DFS used to find the blocks of a graph in Chapter 6, and to the DFS used above to find a topological ordering in an acyclic digraph. When finding a topological ordering, we saw that in a digraph *G,* Algorithm 10.3.2 constructs a spanning forest of outdirected, rooted trees. Each time DFS*(u)* is called, a DF-tree rooted at *u* is built. The edges of *G* can be classified as either tree-edges or fronds. For example, a spanning forest for the graph of Figure 10.3 is shown in Figure 10.4 below. The fronds are shown as dashed edges. Not all the fronds are shown, as can be seen by comparing Figures 10.3 and 10.4. The numbering of the nodes is the DF-numbering.

Let the components of the spanning forest constructed by a depth-first search in a graph *G* be denoted T1, T2, ..., Tk, where the *Ti* were constructed in that order. Figure 10.4 has *k*=3. Each *Ti* is an out-directed,

rooted tree. Notice that each strong component of *G* is contained within some *Ti*, and that each *Ti* may contain more than one strong component. Fronds can be directed from a tree *Ti* to a previous tree *Tj*, where *j*<*i*, but not to a later tree, by nature of the depth-first search.
Given any vertex *u, u* is contained in some *Ti.* The set of ancestors of *u* is *A(u),* all vertices (except *u*) contained in the path in *Ti* from *u* to the root of *Ti*. When DFS(u) is called, it will in turn call DFS*(w)* for several vertices *w*. The *branch* of *Ti* at *u* containing *w* is the sub-tree built by the recursive call DFS (*w*). For example, in Figure 10.4, there are two branches at vertex 4, constructed by the recursive calls DFS(5) and DFS(7). If *x* is any vertex for which $v \in A(x)$, we write *Bu(x)* for the branch at *u* containing *x*. In Figure 10.4 above, we have *B*4(7)=*B*4(8) and *B*4(5)=*B*4(6).
LEMMA 10.1 Suppose that a depth-first search in G is visiting vertex u, and that G−A(u) contains a directed path from u to w. Then vertex w will be visited before the algorithm returns from visiting u.

Page 232

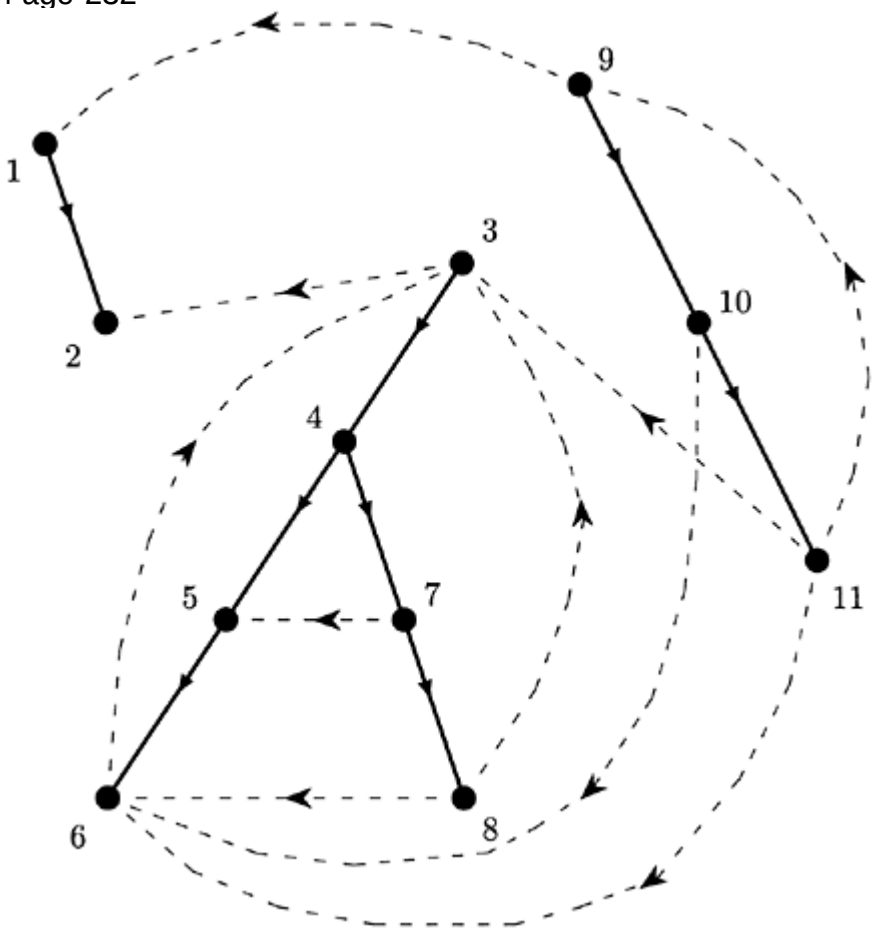


FIGURE 10.4
A depth-first, rooted, spanning forest
PROOF Exercise 10.4.1.
This lemma allows us to classify the fronds of *G* with respect to a depth-first forest.
*THEOREM 10.2 Let T*1, *T*2,..., *Tkbe the components of a depth-first spanning forest of G, where the Ti were constructed in that order. Let (x,y) be a frond, where* $x \in T_i$. *Then there are three possibilities:*
1. $y \in T_j$, *where j*<*i.*
2. $y \in T_i$, *and one of x and y is an ancestor of the other.*
3. $y \in T_i$, *and x and y are in different branches of a common ancestor u, where Bu(y) was searched before Bu(x).*

Page 233

PROOF Let *(x,y)* be a frond, where $x \in T_i$. If $y \in T_j$, where *j*≠*i*, then we must have *j*<*i*, for otherwise the previous lemma tells us that *y* would be visited before DFS(x) returns, so that *x* would be an ancestor of *y*.

Otherwise $x, y \in T_i$. If *x* is visited before *y,* then the previous lemma again tells us that *x* would be an ancestor of *y*. This gives the second case above. Otherwise *y* is visited before *x*. If *G* contains a directed *yx*-path, then we have y an ancestor of *x,* again the second case above. Otherwise there is no directed *yx*-path. The paths in *Ti* from *x* and *y* to the root of *Ti* first meet in some vertex *u*. Then *Bu(y)* was searched before *Bu(x),* giving Case 3.

We call a frond *(x,y)* type 1, 2, or 3 according to whether it falls in Case 1, 2, or 3 in Theorem 10.2. Fronds of type 1 cannot be part of any directed cycle since there are no edges from *Tj* to *Ti* when *j*<*i*. Therefore these fronds are not in any strong component. Consequently each strong component is contained within some *Ti*. A frond of type 2 creates a directed cycle, so that all edges of *Ti* on the path connecting *x* to *y* are in the same strong component. The low-point technique used to find blocks in Chapter 6 will work to find these cycles. A frond *(x,y)* of type 3 may or may not be part of a directed cycle. Consider the frond (7, 5) in Figure 10.4. Vertices 7 and 5 are in different branches at vertex 4. Since 4 is an ancestor of 7, we have a directed path (4, 7, 5). If we were to compute low-points, we would know that the low-point of 5 is vertex 3, an ancestor of 4. This would imply the existence of a directed cycle containing 3, 4, 7, and 5, namely, (3,4,7,5,6). So we find that 7 is in the same strong component as 5 and that the low-point of 7 is also 3.

We can build the strong components of *G* on a stack. Define the low-point of a vertex *u* to be *LowPt[u]*=the smallest *DFNum[w],*

where either *w*=*u* or $w \in A(v)$ and *G* contains a directed path from *u* to *w*. The main component of Algorithm 10.4.1 to compute the strong componets just initiallizes the variables and calls Proceedure DFS() to build each rooted tree of the spanning forest and to compute the low-points. We assume that Proceedure DFS() has access to the variables of the calling program as globals. The algorithm stores the vertices of each strong component on a stack, stored as an array. As before, we have the *DFNum*[·] and *LowPt*[·] arrays. We also store the *Stack*[·] as an array of vertices. *OnStack [u]* is **true** if *u* is on the stack. *DFCount* is a global counter. *Top* is a global variable giving the index of the current top of the stack.

The Procedure DFS() computes the low-points and builds the stack. The algorithm begins by stacking each vertex that it visits. The vertices on the stack will form the current strong component being constructed. *LowPt[u]* is initiallized to *DFNum[u]*. Each w such that $w \longrightarrow v$ is taken in turn. The statements at point (1) extend the DFS from vertex *u* to *w*. Upon returning from the recursive

---

Page 234

call, *LowPt[u]* is updated. Since $v \longrightarrow w$ and *G* contains a directed path from *w* to *LowPt[w],* we update *LowPt[u]* if *LowPt[w]* is smaller.

The statements at point (2) are executed if *uw* is a frond. If *DFNum[w]*> *DFNum[u],* it means that *u* is an ancestor of *w*. These fronds are ignored. Otherwise *w* was visited before *u*. If *w* is the parent of *u, uw* is a tree edge rather than a frond, and is ignored. If *w* is no longer on the stack, it means that *w* is in a strong component previously constructed. The edge *uw* cannot be part of a strong component in that case, so it is also ignored. If each of these tests is passed, *G* contains a directed path from *u* to *LowPt[w],* which is in the same strong component as *w*. Therefore *u* and *w* are in the same strong component. If this value is smaller than *LowPt[u],* then *LowPt[u]* is updated. Statement (3) is reached after all *w* adjacent to *u* have been considered. At this point the value of *LowPt[u]* is known. If *LowPt[u]=DFNum[u],* it means that there is no directed path from *u* to any ancestor of *u*. Every vertex of the strong component containing *u* has been visited, and so is on the stack. These vertices are then popped off the stack before returning.

The complexity of Algorithm 10.4.1 is easily seen to be $O(n+\varepsilon)$. For each vertex *u,* all out-edges *uw* are considered, giving

$$\sum_v d^+(v) = \varepsilon$$

steps. Some arrays of length *n* are maintained. Each node is stacked once, and removed once from the stack.

Page 235

Algorithm 10.4.1: STRONGCOMPONENTS *(G,n)*
comment: Find the strong components using a depth-first search.
procedure DFS*(u)*
comment: extend the depth-first search to vertex *u*
DFCount←*DFCount*+1
DFNum[u]←*DFCount*
LowPt[u]←*DFCount* "initial value"
Top←*Top*+1

```
        Stack[Top]←u "push u on Stack"
            OnStadc[u]←true
        for each w such that v ⟶ w
do { if DFNum[w] = 0
     then { DFS(w)
            if LowPt[w] < LowPt[v]
              then LowPt[v] ← LowPt[w]
     else { if DFNum[w] < DFNum[v]
              then if w ≠ parent of v
              then if OnStack[w]
              then if LowPt[w] < LowPt[v]
              then LowPt[v] ← LowPt[w]
   if LowPt[u]=DFNum[u] (3)
then { comment: { the points on the stack up to v form a
                { strong component – pop them off
       repeat
         w ← Stack[Top]
         Top ← Top − 1
         OnStack[w] ← false
       until w = v
                main
            for u←1 to n
do { DFNum[u] ← 0
   { OnStack[v] ← false
          DFCount←0
            Top←0
          for u←1 to n
       do if DFNum[u]=0
            then DFS(u)
```


Page 236

10.4.1 An application to fabrics

A fabric consists of two sets of strands at right angles to each other, called the *warp and weft,* woven together. The pattern in which the strands are woven can be represented by a rectangular matrix. Let the horizontal strands be *h*1, *h*2,..., *hm* and let the vertical strands be *u*1, *u*2,..., *un*. The matrix shown below contains an X wherever *hi* passes under *uj,* and a blank otherwise. The pattern can be repeated as often as desired.

	v_1	v_2	v_3	v_4	v_5	v_6
h_1	X		X	X		X
h_2		X		X	X	
h_3	X		X			X

Suppose that the strand *h*1 were lifted. Since it passes under *u*1, *u*3, *u*4, and *u*6, these vertical strands would also be lifted. But since *u*1 passes under *h*2, this would in turn lift *h*2. Similarly lifting *h*2 would cause *u*2 to be lifted, which in turn causes *h*3 to be lifted. So the fabric hangs together if any strand is lifted.

For some pattern matrices, it is quite possible that the fabric defined does not hang together. For example, in the simplest case, a strand *hi* could lie under or over every *uj,* allowing it to be lifted off the fabric, or the fabric could fall apart into two or more pieces. In general, we can form a bipartite directed graph whose vertices are the set of all strands. The edges are

{(*u, w*): strand *u* lies under strand *w*}.

Call this the *fabric graph*. It is an oriented complete bipartite graph. If the fabric graph is strongly connected, then it hangs together, since there is a directed path from any strand to another. If the fabric graph is not strongly connected, then it can be separated into its strong components. Some strong component will lie

completely over another strong component, and be capable of being lifted off.

Exercises

10.4.1 Prove Lemma 10.1.

10.4.2 Program the algorithm for strong components, and test it on the digraphs of Figures 10.1, 10.2, 10.3, 10.5, and 10.6.

10.4.3 Find all digraphs which can be obtained by orienting a cycle of length 5 or 6.

10.4.4 Determine whether the fabric defined by the pattern matrix in Figure 10.7 hangs together.

Page 237

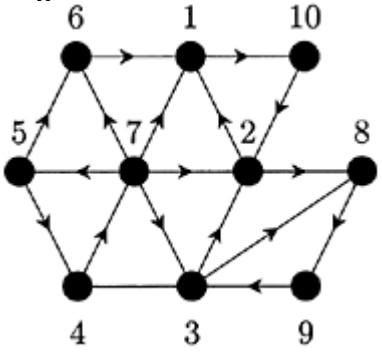


FIGURE 10.5
Find the strong components

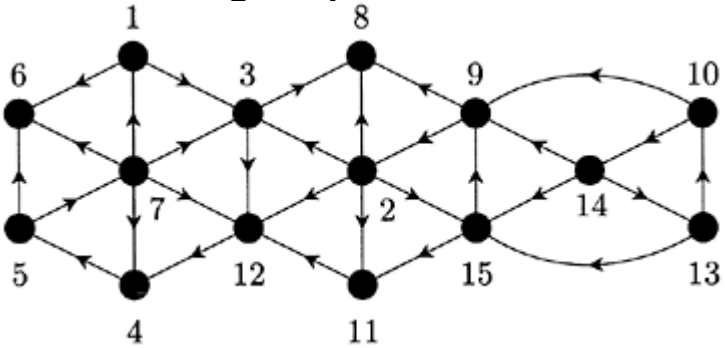


FIGURE 10.6
Find the strong components

			X		X		
X						X	
X	X		X	X	X	X	
X			X		X		
X		X	X	X	X	X	X
			X			X	
X	X		X		X	X	
X		X	X		X	X	X

FIGURE 10.7
A pattern matrix

Page 238

10.5 Tournaments

In a round-robin tournament with *n* teams, each team plays every other team. Assuming that ties are not allowed, we can represent a win for team *u* over team *u* by a directed edge *(u,u)*. When all games have been played we have a directed complete graph. We say that a *tournament* is any oriented complete graph. It is easy to see that there are exactly two possible tournaments on three vertices, as shown in Figure 10.8.

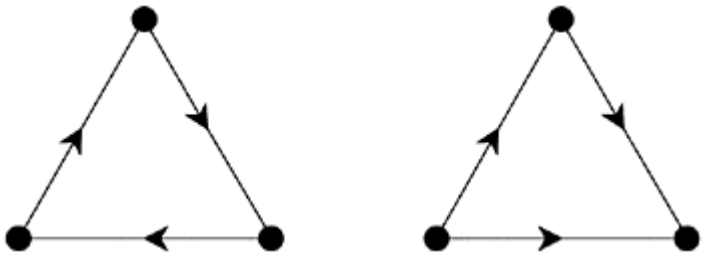


FIGURE 10.8
The tournaments on three vertices

The second of these tournaments has the property that if $u \longrightarrow v$ and $v \longrightarrow w$, then $u \longrightarrow w$. Any tournament which has this property for all vertices *u, u, w,* is called a *transitive tournament*. It is easy to see that there is a unique transitive tournament *Tn* for each *n*≥1. For if *Tn* is a transitive tournament, there must be a unique node *u* that is undefeated. If we delete it, we are left with a transitive tournament on *n*−1 vertices. We use induction to claim that this is the unique *Tn*−1 When *u* is restored, we have the uniqueness of *Tn*.

If *G* is any tournament on *n* vertices, it will have a number of strong components. Let *G** denote its condensation. Then since *G** is acyclic, it is a transitive tournament on *m*≤*n* vertices. We can find a topological ordering of *V(G*),* and this will define an ordering of the strong components of *G*. We can then make a list of the possible sizes of the strong components of *G,* ordered according to the topological ordering of *G**. We illustrate this for *n*=4. The possible sizes of the strong components are (1, 1, 1, 1), (1, 3), (3, 1), and (4), since a simple digraph cannot have a strong component with only two vertices. The first ordering corresponds to the transitive tournament *T*4. The orderings (1, 3) and (3, 1) correspond to the first two tournaments of Figure 10.9. It is easy to see that they are unique, since there is only one strong tournament on three vertices, namely the directed cycle.

The third tournament is strongly connected. We leave it to the reader to verify that it is the only strong tournament on four vertices. The following theorem will

Page 239

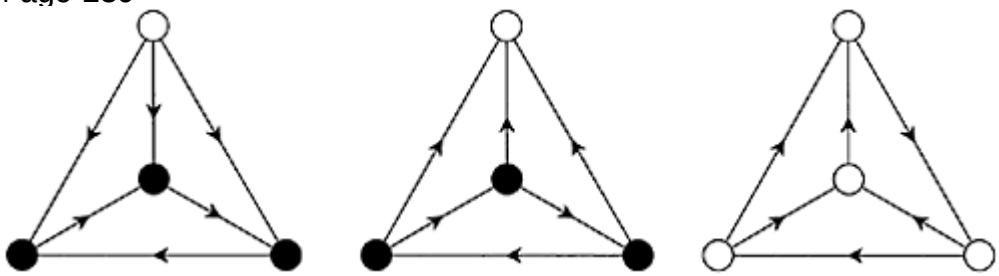


FIGURE 10.9
Non-transitive tournaments on 4 vertices

be helpful. A digraph is said to be hamiltonian if it contains a directed hamilton cycle.

THEOREM 10.3 Every strong tournament on n≥3 *vertices is hamiltonian*.

PROOF Let *G* be a strong tournament, and let *C*=(*u*1, *u*2, ..., *uk*) be the longest directed cycle in *G*. If *G* is non-hamiltonian then *ℓ(C)*<*n*. Pick any $v \notin C$. Because *G* is a tournament, either $v \longrightarrow u_1$ or else $u_1 \longrightarrow v$. Without loss, suppose that $u_1 \longrightarrow v$. If $v \longrightarrow u_2$, then we can produce a longer cycle by inserting *u* between *u*1and *u*2. Therefore $u_2 \longrightarrow v$. If $v \longrightarrow u_3$, then we can produce a longer cycle by inserting u between *u*2 and *u*3. Therefore $u_3 \longrightarrow v$, etc. Eventually we have $v \longrightarrow u_i$ for all *ui*. This is impossible, since *G* is strong. We finish this section with the following theorem **??**.

THEOREM 10.4 **(Robbins' Theorem)**

Every 2-connected graph has a strong orientation.

PROOF Let *G* be a 2-connected graph. Then *G* contains a cycle *C,* which has a strong orientation. Let *H* be a subgraph of *G* with the largest possible number of vertices, such that *H* has a strong orientation. If $u \notin H$, then since *G* is 2-connected, we can find two internally disjoint paths *P* and *Q* connecting *u* to *H*. Orient *P* from *u* to *H,* and *Q* from *H* to *u*. This gives a strong orientation of a larger subgraph than *H,* a contradiction.

Page 240

Exercises

10.5.1 Show that there is a unique strong tournament on four vertices.

10.5.2 Find all the tournaments on five vertices. Show that there are exactly 12 tournaments, of which 6 are strong.
10.5.3 Show that every tournament has a hamilton path.
10.5.4 Show that if an odd number of teams play in a round robin tournament, it is possible for all teams to tie for first place. Show that if an even number of teams play, it is not possible for all teams to tie for first place.
10.5.5 Prove the following theorem. Let *G* be a digraph on *n* vertices such that $d^+(u) + d^-(u) \geq n$ whenever $u \not\longrightarrow v$. Then *G* is strong.
10.5.6 Describe a $O(n+\varepsilon)$ algorithm to find a strong orientation of a 2-connected graph.
10.5.7 Show that every connected graph has an acyclic orientation.

10.6 2-Satisfiability

In Chapter 9 we saw that 3-Sat is NP-complete. The related problem 2-Sat $\in$ P. It has a number of practical applications.

Problem 10.1: 2-Sat

Instance: a set of boolean variables *U* and a boolean expression *B* over *U,* in which each clause contains exactly two variables.

Question: is *B* satisfiable?

Consider the following instance of 2-Sat:

$$(u1+u2)(u1+\bar{u}2)(u2+u3)(\bar{u}1+\bar{u}3)$$

We want a truth assignment satisfying this expression. If *u*1=**false,** the first clause tells us that *u*2=**true**. We could write this implication as $\bar{u}1 \rightarrow u2$. The second clause tells us that if *u*1=**false,** then *u*2=**false**. We could write this implication as $\bar{u}1 \rightarrow \bar{u}2$. As this gives a contradiction, we conclude that *u*1=**true** is necessary. Continuing in this line of reasoning quickly gives the solution.

This example shows that a clause (*x*+*y*) of an instance of 2-Sat, where $x, y \in U \cup \overline{U}$, corresponds to two implications $\overline{x} \rightarrow y$ and $\overline{y} \rightarrow x$. We can construct a digraph with edges based on these implications. Given an instance of 2-Sat with variables *U* and boolean expression *B,* construct a digraph *G* whose vertex set is $U \cup \overline{U}$. The edges of *G* consist of all ordered

pairs $(\overline{x}, y)$ and $(\overline{y}, x)$, where (*x*+*y*) is a clause of *B*. *G* is called the *implication digraph* of *B*. The implication digraph corresponding to the above expression is shown in Figure 10.10. A sequence of implications $\overline{x} \rightarrow y \rightarrow z$ corresponds to a directed path in *G*. Thus, directed paths in *G* are important. If any variable in a directed path is assigned the value **true,** then all subsequent variables in the path must also be **true**. Similarly, if any variable in a directed path is assigned the value **false,** then all previous variables must also be **false**. If *G* contains a directed cycle *C,* then all variables of *C* must be **true,** or all must be **false**. We are talking about the strong components of *G*.

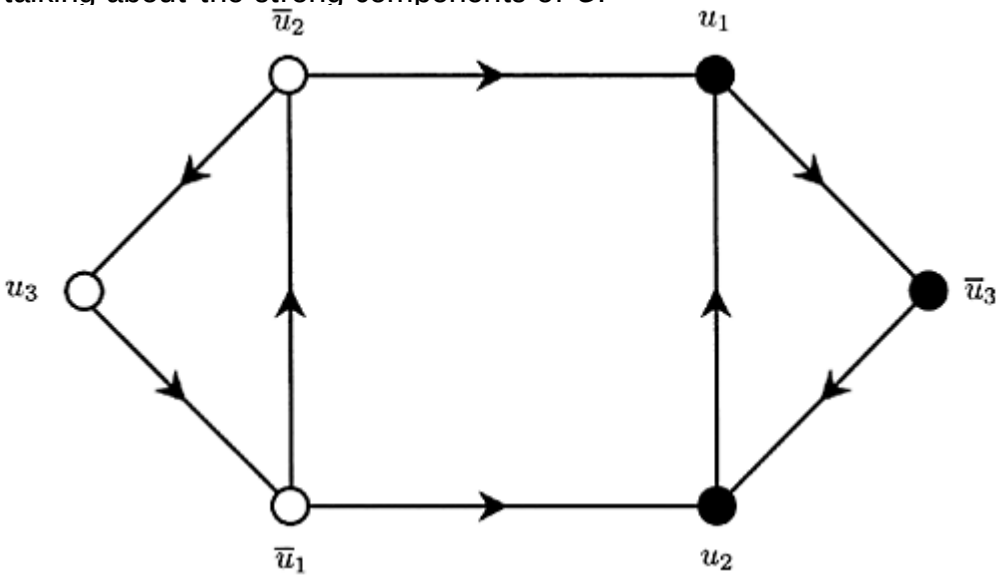


FIGURE 10.10
The implication digraph corresponding to an instance of 2-Sat

The graph *G* has an *antisymmetry*—if there is an edge *(x, y),* then there is also an edge $(\overline{y}, \overline{x})$, as can be seen

from the definition. Therefore the mapping of *V(G)* that interchanges every *ui* and *ūi* reverses the orientation of every edge.
Let *G*1, *G*2,..., *Gm* be the strong components of *G*. The variables in any strong component are either all true, or all false. If any strong component contains both *ui* and *ūi,* for any *i,* then the expression *B* is not satisfiable; for *ui* and *ūi* cannot be both true, or both false. So if *B* is satisfiable, *ui* and *ūi* are in different strong components.
The antisymmetry maps each strong component *Gj* to another strong component *G′j,* the *complementary strong component,* such that *x* is in *Gj* if and only if $\overline{x}$ is in *G′j*. If all variables of *Gj* are true, then all variables of *G′j* must be false, and conversely. This gives the following algorithm for 2-Sat:

Page 242

Algorithm 10.6.1: 2SAT*(B,U)*

comment: { Given an instance B of 2-Sat with variables U, construct a solution, if there is one.

construct the implication digraph *G* corresponding to *B*
construct the strong components of *G*
for each $u \in U$
do if *u* and *ū* are in the same strong component
then return *(NonSatisfiable)*
construct the *condensation* of *G,* and find a topological ordering of it
let *G*1, *G*2,..., *Gm* be the topological ordering of the strong components
for *i*←*m* **downto** 1
do { **if** the variables of G_i have not been assigned
then { assign all variables of G_i to be **true**
assign all variables of G_i' to be **false**

In the graph of Figure 10.10 there are two strong components—the shaded vertices and the unshaded vertices. The condensation is a digraph with one edge, directed from left to right. The algorithm will assign **true** to all variables in the shaded strong component, and **false** to all variables in the unshaded one. This is the unique solution for this instance of 2-Sat.
THEOREM 10.5 Given an instance B of 2-Sat with variables U. Algorithm 2SAT*(B,U)finds a solution if and only if a solution exists*.
PROOF Every clause (*x*+*y*) of *B* corresponds to two implications $\overline{x} \rightarrow y$ and $\overline{y} \rightarrow x$. The implication digraph *G* contains all these implications. Any assignment of truth values to the variables that satisfies all implications satisfies *B*. If some strong component of *G* contains both *ui* and *ūi* for some variable *ui,* then there is no solution. The algorithm will detect this. Otherwise, *ui* and *ūi* are always in complementary strong components. The algorithm assigns values to the variables such that complementary strong components always have opposite truth values. Therefore, for every *ui* and *ūi* exactly one will be true, and one will be false. Consider a variable $x \in U \cup \overline{U}$. Suppose that *x* is in a strong component *Gj*. Its complement $\overline{x}$ is in *G′j*. Without loss, suppose that *G′j* precedes *Gj* in the topological order. Then *x* will be assigned **true** and $\overline{x}$ will be assigned **false**. All clauses (*x*+*y*) containing *x* are thereby satisfied. All clauses $(\overline{x} + z)$ containing $\overline{x}$ correspond to implications *x*→*z* and $\overline{z} \rightarrow \overline{x}$. It follows that *z* is either in the same strong component as *x,* or else in a strong component following *Gj,* and $\overline{z}$ is in a strong component preceding *G′j*. In either case, the algorithm has already assigned *z*←**true,** so that the clause $(\overline{x} + z)$ is also satisfied. We conclude that

Page 243
the truth assignment constructed by the algorithm satisfies *B*.
This also gives the following theorem.
THEOREM 10.6 Given an instance B of 2-Sat with variables U. B is satisfiable if and only if, for every $u_i \in U$, *ui and ūi are contained in different strong components of the implication digraph.*
If *U* has *n* variables, and *B* has *k* clauses, the implication graph *G* will have 2*n* vertices and 2*k* edges. It takes *O*(*n*+*k*) steps to construct *G,* and *O*(*n*+*k*) steps to find its strong components, and to construct a topological order of them. It then takes *O(n)* steps to assign the truth values. Thus, we have a linear algorithm that solves 2-Sat.
Exercises

10.6.1 Construct the implication digraph for the following instance of 2-Sat.
$(u1+u2)(\bar{u}1+u3)(\bar{u}3+\bar{u}4)(u1+u4)(\bar{u}2+\bar{u}5)(u5+\bar{u}6)(u2+u6)(\bar{u}3+u4)$
10.6.2 Solve the previous instance of 2-Sat.
10.6.3 Given an instance of 2-Sat with the additional requirement that $u1=$ **true**. Show how to convert this into an instance of 2-Sat and solve it. Show also how to solve it if $u1$ is required to be **false**.
10.6.4 Consider an instance of Sat in which each clause has exactly two variables, except that one clause has three or more variables. Describe an algorithm to solve it in polynomial time.

10.7 Notes

An excellent reference for digraphs is BANG-JENSEN and GUTIN [8]. The algorithms for strong components is from AHO, HOPCROFT, and ULLMAN [1]. Strong components and 2-satisfiability are further examples of the importance and efficiency of the depth-first search. The subject of tournaments is a vast area. A survey can be found in REID and BEINEKE [102]. A good monograph on the topic is MOON [89].

Page 244
This page intentionally left blank.

Page 245
11
Graph Colorings

11.1 Introduction

A *coloring* of the vertices of a graph G is an assignment of colors to the vertices. A coloring is *proper* if adjacent vertices always have different colors. We shall usually be interested only in proper colorings. It is clear that the complete graph K_n requires n distinct colors for a proper coloring. Any bipartite graph can be colored in just two colors. More formally,
DEFINITION 11.1: An *m-coloring* of G is a mapping from $V(G)$ onto the set $\{1, 2, \ldots, m\}$ of m "colors". The *chromatic number* of G is $\chi(G)$, the minimum value m such that G has a proper m-coloring. If $\chi(B)=m$, G is then said to be *m-chromatic*.
If G is bipartite, we know that $\chi(G)=2$. Moreover, there is a $O(\varepsilon)$ algorithm to determine whether an arbitrary G is bipartite, and to construct a 2-coloring of it. When $\chi(G)\geq 3$, the problem becomes NP-complete. We show this in Section 11.7. (See Problem 11.1.) A consequence of this is that there is no complete theoretical characterization of colorability. As with the Hamilton cycle problem, there are many interesting techniques, but most problems only have partial solutions. We begin with a simple algorithm for coloring a graph, the *sequential algorithm*. We will indicate various colorings of G by the notation $\chi 1$, $\chi 2$,..., etc. While $\chi(G)$ represents the chromatic number of G, there may be many colorings of G that use this many colors. If $\chi 1$ is a coloring, then $\chi 1(u)$ represents the color assigned to vertex u under the coloring $\chi 1$.

Page 246

Algorithm 11.1.1: SEQUENTIALCOLORING(G,n)
comment: Construct a coloring $\chi 1$ of a graph G on n vertices
mark all vertices "uncolored"
order $V(G)$ in some sequence $u1, u2, \ldots, un$
for $i \leftarrow n$ **downto** 1
do $\chi 1(ui) \leftarrow$ the first color available for ui

To calculate the first available color for ui, we consider each $u \rightarrow ui$. If u is already colored, we mark its color "in use". We then take the first color not in use as $\chi 1(ui)$, and then reset the color flags before the next iteration. Thus, iteration i of the for-loop takes $O(\text{DEG}(ui))$ steps. Algorithm 11.1.1 is then easily seen to be $O(\varepsilon)$. The number of colors that it uses usually depends on the sequence in which the vertices are taken.
If we knew a proper $\chi(G)$-coloring before beginning the algorithm, we could order the vertices by color: all vertices of color 1 last, then color 2, etc. Algorithm 11.1.1 would then color G in exactly $\chi(G)$ colors. Since we don't know $\chi(G)$ beforehand, we investigate various orderings of $V(G)$.
Spanning trees often give useful orderings of the vertices. We choose a vertex $u1$ as a root vertex, and build a spanning tree from it. Two immediate possibilities are a breadth-first or depth-first tree. The ordering of the vertices would then be the order in which they are numbered in the spanning tree. This immediately gives the following lemma.
LEMMA 11.1 If G is a connected, non-regular graph, then $\chi(G)\leq\Delta(G)$. *If G is regular, then* $\chi(G)\leq\Delta(G)+1$.
PROOF Assume first that G is non-regular, and choose a vertex $u1$ of degree $<\Delta(G)$ as the root of a spanning tree. Apply Algorithm 11.1.1 to construct a coloring $\chi 1$. When each vertex $ui \neq u1$ comes to be

colored, the parent of *ui* has not yet been colored. Therefore at most DEG*(ui)*−1 adjacent vertices have already been colored. Hence $\chi 1(ui) \leq \Delta(G)$. When *u*1 comes to be colored, all adjacent vertices have already been colored. Since DEG(*u*1)<Δ*(G),* we conclude that $\chi 1(u1) \leq \Delta(G)$. Hence $\chi(G) \leq \Delta(G)$.
If *G* is regular, then the proof proceeds as above, except that $\chi 1(u1) \leq \Delta(G)+1$. The conclusion follows. Notice that if *G* is regular, only one vertex needs to use color Δ*(G)*+1.
If Algorithm 11.1.1, using a breadth-first or depth-first spanning tree ordering, is applied to a complete graph *Kn,* it is easy to see that it will use exactly *n* colors. If the algorithm is applied to a cycle *Gn,* it will always use two colors if *n* is even,

Page 247
and three colors, if *n* is odd. Complete graphs and odd cycles are special cases of Brooks' theorem.
THEOREM 11.2 **(Brooks' theorem)** *Let G be a connected graph that is not a complete graph, and not an odd cycle. Then* $\chi(G) \leq \Delta(G)$.
PROOF Suppose first that *G* is 3-connected. If *G* is non-regular, we know the result to be true. Hence we assume that *G* is a regular graph. Since *G* is not complete, we can choose vertices *u, u, w* such that $u \rightarrow u, w,$ but $v \nrightarrow w.$ Construct a spanning tree ordering of $G-\{u,w\}$, with *u* as root, so that $u1 = u$. Then set $un-1 = u$, $un = w$, and apply Algorithm 11.1.1 to construct a coloring $\chi 1$ of *G*. Vertices *u* and *w* will both be assigned color 1. Each $ui \neq u1$ will have $\chi 1(ui) \leq \Delta(G)$. When *u*1 comes to be colored, the two adjacent vertices *u* and *w* will have the same color. Hence $\chi 1(u1) \leq \Delta(G)$.
Otherwise *G* is not 3-connected. Suppose that *G* is 2-connected. Choose a pair of vertices $\{u,u\}$, such that $G-\{u,u\}$ is disconnected. If *H* is any connected component of $G-\{u,u\}$, let *Huu* be the subgraph of $G+uu$ induced by $V(H) \cup \{u, v\}.$ Now *G* is regular, of degree at least three (or *G* would be a cycle). Therefore we can choose *u* and *u* so that Huu is a non-regular graph. Therefore $\chi(Huu) \leq \Delta(Huu)$. But $\Delta(Huu) \leq \Delta(G)$. Color *Huu* in at most Δ*(G)* colors. Notice that *u* and *u* have different colors, because $uv \in E(H_{uv}).$ We can do the same for every subgraph *Kuu* so constructed from each connected component of $G-\{u,u\}$. Furthermore, we can require that *u* and *u* are colored identically in each *Kuu,* by permuting colors if necessary. These subgraph colorings determine a coloring of *G* with at most Δ*(G)* colors.
If *G* is not 2-connected, but has a cut-vertex *u,* we use an identical argument, deleting only *u* in place of $\{u,u\}$.

11.1.1 Intersecting lines in the plane

An interesting example of the use of Algorithm 11.1.1 is given by intersecting lines in the plane. Suppose that we are given a collection of *m* straight lines in the plane, with no three concurrent. Construct a graph *G* whose vertices are the points of intersection of the lines, and whose edges are the line segments connecting the vertices. An example is shown in Figure 11.1. We can use Algorithm 11.1.1 to show that $\chi(G) \leq 3$. Notice first that because at most two lines are concurrent at a point that we have $\Delta(G) \leq 4$. Also note that *G* can be enclosed by a disc in the plane. Choose a disc in the plane containing *G,* and choose a line ℓ in the plane that is outside the disc, such that ℓ is not parallel to any of the original *m* lines. The dotted line in Figure 11.1 represents ℓ. Assign an orientation to the edges of *G* by directing each edge toward ℓ. This converts *G* to an acyclic digraph. Notice

Page 248
that each vertex has at most two incident edges oriented outward, and at most two oriented inward. Let *u*1, *u*2,..., *un* be a topological ordering of *V(G)*. Apply Algorithm 11.1.1. The vertices of *G* have degree two, three, or four. When each *ui* comes to be colored, at most two adjacent vertices have already been colored—those incident on out-edges from *ui*. Therefore $\chi 1(ui) \leq 3$, for each *ui*. It follows that $\chi(G) \leq 3$.

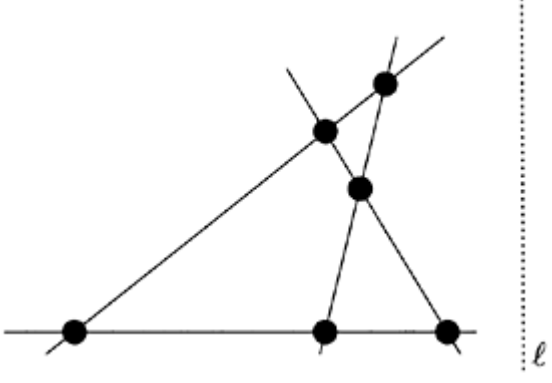


FIGURE 11.1
Intersecting lines in the plane

Exercises

11.1.1 Write computer programs to apply the sequential algorithm using breadth-first and depth-first searches, and compare the colorings obtained.

11.1.2 Show that if a breadth-first or depth-first sequential algorithm is applied to a bipartite graph, that exactly two colors will be used.

11.1.3 Construct a complete bipartite graph *Kn,n* with bipartition *X*={*x*1,*x*2,...,*xn*}, *Y*={*y*1, *y*2, ..., *yn*}, and remove the matching *M*={*x*1*y*1, ..., *xnyn*}, to get *G*=*Kn,n*−*M*. Order the vertices *x*1, *y*1, *x*2, *y*2,..., *xn, yn*. Show that Algorithm 11.1.1 with this ordering will use *n* colors.

11.1.4 Construct a complete tripartite graph *Kn,n,n* with tripartition *X*={*x*1,*x*2,..., *xn*}, *Y*={*y*1, *y*2,..., *yn*}, *Z*={*z*1, *z*2, ..., *zn*}, and remove the triangles *T*={*x*1*y*1*z*1,...,*xnynzn*}, to get *G*=*Kn,n,n*−*T*. Order the vertices *x*1, *y*1, *z*1, *x*2, *y*2, *z*2,..., *xn, yn, zn*. Show that Algorithm 11.1.1 with this ordering will use *n* colors. How many colors will be used by the breadth-first sequential algorithm?

11.1.5 Show that in the intersecting lines problem, if three or more lines are allowed to be concurrent, that $\chi(G)$ can be greater than three.

11.1.6 Let *G*1 and *G*2 be graphs with *m*-colorings $\chi 1$ and $\chi 2$, respectively. We say that *G*1 and *G*2 are *color-isomorphic* if there is an isomorphism θ from *G*1 to *G*2 that induces a permutation of the colors. More formally, if *u*1, $u_2 \in V(G_1)$ are mapped by θ to *u*1, $v_2 \in V(G_2)$, respectively, then $\chi 1(u1)=\chi 1$ (*u*2) if and only if $\chi 2(u1)=\chi 2(u2)$. Show how to construct graphs G_1' and G_2' such that *G*1 and *G*2 are color-isomorphic if and only if G_1' and G_2' are isomorphic.

11.1.7 Determine whether the graphs of Figure 11.2 are color-isomorphic, where the colors are indicated by the numbers.

11.1.8 Let the vertices of *G* be listed in the sequence *u*1, *u*2,..., *un* and apply Algorithm 11.1.1. Show that $\chi 1(ui)$ isatmost min{*n*−*i*+1, DEG(*ui*)+1}. Conclude that $\chi(G) \leq \max_{i=1}^{n} \min\{n-i+1, \text{DEG}(v_i)+1\}$.

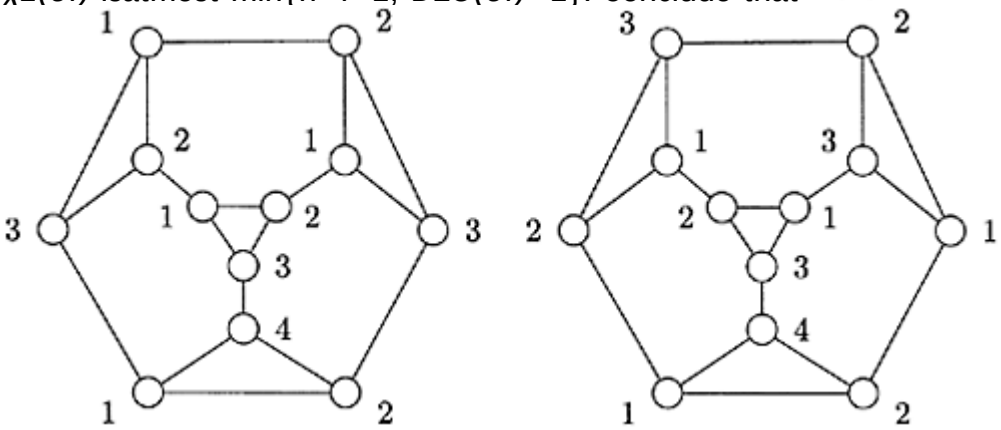


FIGURE 11.2
Are these color-isomorphic?

11.2 Cliques

Let *G* be a graph with a proper coloring. The subset of vertices of color *i* is said to be the *color class* with color *i*. These vertices induce a subgraph with no edges. In $\overline{G}$ they induce a complete subgraph.

DEFINITION 11.2: A *clique* in *G* is an induced complete subgraph. Thus a clique is a subset of the vertices that are pairwise adjacent. An *independent set* is

a subset of *V(G)* which induces a subgraph with no edges. A clique is a *maximum clique* if *G* does not contain any larger clique. Similarly, an independent set is a *maximum independent set* if *G* does not contain any larger independent set. An independent set is also called a *stable set*.

The problem of finding a maximum clique or maximum independent set in *G* is NP-complete, as shown in Section 11.7.

Write $\alpha(G)$ for the number of vertices in a maximum independent set in *G,* and $\overline{\alpha}(G)$ for the number of

$\overline{\alpha}(G)$,

vertices in a maximum clique. It is clear that $\chi(G) \geq$ since a clique in G of m vertices requires at least m colors. If we are given a coloring $\chi 1$ of G and $\chi 2$ of $\overline{G}$, let Vmax be a largest color class in G, and let $\overline{V}_{\max}$ be a largest color class in $\overline{G}$. Since $\overline{V}_{\max}$ induces a clique in G, we have $\overline{\alpha}(G) \geq |\overline{V}_{\max}|$. Since each color class is an independent set, $\alpha(G) \geq |V\max|$.
This gives the bounds

$$|\overline{V}_{\max}| \leq \overline{\alpha}(G) \leq \chi(G)$$

and

$$|V_{\max}| \leq \alpha(G) \leq \chi(\overline{G}).$$

Although we don't know $\chi(G)$ and $\chi(\overline{G})$, we can use the sequential Algorithm 11.1.1 to construct colorings $\chi 1$ and $\chi 2$ of G and $\overline{G}$, so as to obtain bounds on the clique number and independent set number of G and $\overline{G}$. We write $\chi 1(G)$ to denote the number of colors used by the coloring $\chi 1$.

LEMMA 11.3 If $\chi 1$ and $\chi 2$ satisfy $|V_{\max}| = \chi_2(\overline{G})$, then $\alpha(G) = \chi(\overline{G}) = \chi_2(\overline{G})$. If $\chi 1$ and $\chi 2$ satisfy $|\overline{V}_{\max}| = \chi_1(G)$, then $\overline{\alpha}(G) = \chi(G) = \chi_1(G)$.

PROOF The inequalities above hold for any proper colorings $\chi 1$ and $\chi 2$. If $|V_{\max}| = \chi_2(\overline{G})$, then the second inequity determines $\alpha(G)$. Therefore $\chi(\overline{G})$ is at least as big as this number. But since we have a coloring in $\chi_2(\overline{G})$ colors, this determines $\chi(\overline{G})$.

Thus, by using Algorithm 11.1.1 to color both G and $\overline{G}$, we can obtain bounds on $\overline{\alpha}(G)$, and $\chi(\overline{G})$. Sometimes this will give us exact values for some of these parameters. When it does not give exact values, it gives colorings $\chi 1$ and $\chi 2$, as well as a clique $\overline{V}_{\max}$ and independent set Vmax in G. In general, Algorithm 11.1.1 does not construct colorings that are optimal, or even nearly optimal. There are many variations of Algorithm 11.1.1. Improvements to Algorithm 11.1.1 will give improvements in the above bounds, by decreasing the number of colors used, and increasing the size of the maximum color classes found. One modification that is found to work well in practice is the *degree saturation method,* which orders the vertices by the "saturation" degree.
Consider a graph G for which a coloring $\chi 1$ is being constructed. Initially all vertices are marked uncolored. On each iteration of the algorithm, another vertex

is colored. For each vertex, let $c(u)$ denote the number of distinct colors adjacent to u. $c(u)$ is called the *saturation degree* of u. Initially $c(u)=0$. At any stage of the algorithm the vertices are partially ordered. A vertex u such that the pair $(c(u), \text{DEG}(u))$ is largest is chosen as the next vertex to color; that is, vertices are compared first by $c(u)$, then by $\text{DEG}(u)$.

Algorithm 11.2.1: DEGREESATURATION*(G, n)*
comment: Construct a coloring $\chi 1$ of a graph G on n vertices
mark all vertices "uncolored"
initialize $c(u) \leftarrow 0$, for all u
for $i \leftarrow 1$ **to** n
do
- select u as a vertex with largest $(c(u), \text{DEG}(u))$
- $\chi_1(u) \leftarrow$ the first color available for u
- **for each** v⟶u
  - **do if** v is uncolored, adjust $c(v)$

Algorithm 11.2.1 requires a priority queue in order to efficiently select the next vertex to color. In practice, it is found that it uses significantly fewer colors than Algorithm 11.1.1 with a fixed ordering of the vertices. However, algorithms based on the sequential algorithm are limited in their efficacy. JOHNSON [69] discusses the limitations of many coloring algorithms based on the sequential algorithm.
The first vertex, $u1$, that Algorithm 11.2.1 colors will be one of maximum degree. The second vertex, $u2$, will be a vertex adjacent to $u1$ of maximum possible degree. The third vertex, $u3$, will be adjacent to $u1$ and $u2$, if there is such a vertex, and so on. Thus, Algorithm 11.2.1 begins by constructing a clique of vertices of large degree. The algorithm could save this information, and use it as a lower bound on $\overline{\alpha}(G)$.
A recursive search algorithm MAXCLIQUE() to find a maximum clique can be based on a similar strategy. Algorithm 11.2.2 that follows constructs cliques C' The maximum clique found is stored in a variable C. C and

C′ are stored as global arrays. It also uses an array *S* of vertices eligible to be added to *C′*. The set of all neighbors of *u* is denoted by *N(u)*.
This algorithm builds a clique *C′* and tries to extend it from a set *S′* of eligible vertices. When $|C'|+|S'|$ is smaller than the largest clique found so far, it backtracks. The performance will depend on the ordering of the vertices used. As with Algorithm 11.2.1, the ordering need not be chosen in advance, but can be constructed as the algorithm progresses. For example, *ui* might be selected as a vertex of largest degree in the subgraph induced by the vertices not yet considered. If a maximum clique is found early in the algorithm, the remaining calls to EXTENDCLIQUE() will finish more quickly. We can estimate the complexity by

noticing that the loop in MAXCLIQUE() runs at most *n* times. After each choice of *ui,* the set *S* will contain at most $\Delta(G)$ vertices. There are $2\Delta(G)$ subsets of a set of size $\Delta(G)$. The algorithm might construct each subset at most once. Therefore the complexity is at most $O(n \cdot 2\Delta(G))$, an exponential value.

Algorithm 11.2.2: MAXCLIQUE*(G, n)*
comment: Construct a maximum clique *C* in *G*
procedure EXTENDCLIQUE*(S, u)*
comment: *u* has just been added to *C′*—adjust *S* and extend the clique
$S' \leftarrow S \cap N(u)$
if $|S'|=0$
then $\begin{cases} \textbf{if } |C'| > |C| \textbf{ then } C \leftarrow C' \\ \textbf{return} \end{cases}$
while $|S'|>0$ **and** $|C'|+|S'|>|C|$
do $\begin{cases} \text{select } u \in S' \\ C' \leftarrow C' \cup u \\ S' \leftarrow S' - u \\ \textsc{ExtendClique}(S', u) \\ C' \leftarrow C' - u \end{cases}$
main
choose an ordering *u*1, *u*2,..., *un* of the vertices
$C \leftarrow \emptyset$ "largest clique found so far"
$C' \leftarrow \emptyset$ "clique currently being constructed"
$S \leftarrow V(G)$ "vertices eligible for addition to *C′*"
$i \leftarrow 1$
while $|S|>|C|$
do $\begin{cases} C' \leftarrow \{v_i\} \\ S \leftarrow S - v_i \\ \textsc{ExtendClique}(S, v_i) \\ i \leftarrow i + 1 \end{cases}$
comment: *C* is now a maximum clique of size $\overline{\alpha}$
$\overline{\alpha} \leftarrow |C|$

11.3 Mycielski's construction
A graph *G* which contains a 4-clique necessarily has $\chi(G) \geq 4$. However, it is possible for a graph with no 4-clique to have $\chi(G) \geq 4$. Mycielski found a way to construct triangle-free graphs with arbitrarily large chromatic number.
We start with a triangle-free graph *G* with $\chi(G) \geq 3$. Any odd cycle with five or more vertices will do (e.g., $G = C5$). We now extend *G* to a graph *G′* as follows. For each $v \in V(G)$, we add a vertex *u′* to *G′* adjacent to the same vertices of *G* that *u* is adjacent to. We now add one more vertex *u*0 adjacent to each *u′*. Thus, if *G* has *n* vertices, *G′* will have $2n+1$ vertices. Refer to Figure 11.3.

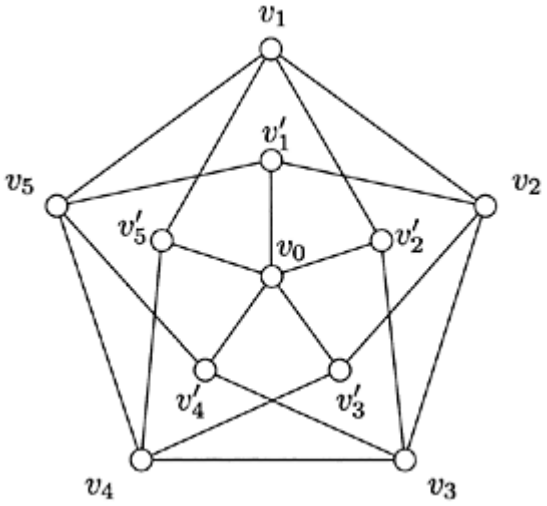


FIGURE 11.3
Mycielski's construction

LEMMA 11.4 $\chi(G')=\chi(G)+1$. *Furthermore, G′ has no triangles if G has none.*

PROOF Consider a coloring $\chi 1$ of G'. Since it induces a coloring of G, we conclude that $\chi(G')\geq\chi(G)$. Let $m=\chi(G)$. Some vertex of G with color number m is adjacent to $m-1$ other colors in G; for otherwise each vertex of color m could be recolored with a smaller color number. It follows that the vertices u' must be colored with at least $\chi(G)$ colors. In fact, we could assign each $\chi 1(u')=\chi 1(u)$. The vertex $u0$ is adjacent to $\chi(G)$ colors, so that $\chi 1(u0)= m+1$. It follows that $\chi(G')=\chi(G)+1$. It is easy to see that G' has no triangles, because G has none.

Page 254

By iterating Mycielski's construction, we can construct triangle-free graphs with large chromatic numbers. In fact, if we begin the construction with $G=K2$, the result is $C5$. Or we could start Mycielski's construction with a graph which contains triangles, but is $K4$-free, and construct a sequence of $K4$-free graphs with increasing chromatic numbers, and so forth.

11.4 Critical graphs

Let G be a graph with $\chi(G)=m$. If we remove an edge uu from G, there are two possibilities, either $\chi(G-uu)=m$ or $\chi(G-uu)=m-1$. In the latter case, we say that edge uu is *critical*.

DEFINITION 11.3: A graph G is *critical* if $\chi(G-uu)=\chi(G)-1$ for all edges $uv \in E(G)$. If $\chi(G)=m$, we say that G is *m-critical*.

It is easy to see that every graph contains a critical subgraph. If $\chi(G-uu)= \chi(G)$ for some edge uu, we can remove uu. Continue deleting edges like this until every edge is critical. The result is a critical subgaph. Critical graphs have some special properties.

LEMMA 11.5 If G is m-critical, then $\delta(G)\geq m-1$.

PROOF If $\mathrm{DEG}(u)<m-1$, choose an edge uu, and color $G-uu$ with $m-1$ colors. Since $\mathrm{DEG}(u)<m-1$, there are at most $m-2$ adjacent colors to u in G. So there is always a color in$\{1, 2,\ldots, m-1\}$ with which u can be colored to obtain an $(m-1)$-coloring of G, a contradiction.

If G is an m-critical graph, then G has at least m vertices, and each has degree at least $m-1$. Therefore every graph with $\chi(G)=m$ has at least m vertices of degree $\geq m-1$.

LEMMA 11.6 Every critical graph with at least three vertices is 2-connected.

PROOF Suppose that G is an m-critical graph with a cut-vertex u. Let H be a connected component of $G-u$, and let Hu be the subgraph induced by $V(H)\cup\{v\}$. Color Hu with $\leq m-1$ colors. Do the same for every such subgraph Hu. Ensure that u has the same color in each subgraph, by permuting colors if necessary. The result is a coloring of G in $\leq m-1$ colors, a contradiction.

The ideas of the Lemma 11.6 can be extended to separating sets in general.

Page 255

LEMMA 11.7 Let S be a separating set in an m-critical graph G. Then S does not induce a clique.

PROOF If S is a separating set, let H be a component of $G-S$, and consider the subgraph HS induced by $V(H)\cup S$. It can be colored in $m-1$ colors. The vertices of S are colored with $|S|$ distinct colors, which can be permuted in any desired way. Do the same for every component of $G-S$. The result is a coloring of G in

$m-1$ colors.

It follows from this lemma that if $\{u, v\}$ is a separating set in an m-critical graph, that $uv \notin E(G)$. Suppose that $\{u, v\}$ is a separating set. Let H be a component of $G-\{u, v\}$, and let K be the remaining components. Construct Huv induced by $V(H) \cup \{u,v\}$, and Kuv induced by $V(K) \cup \{u,v\}$. Huv and Kuv can both be colored in $m-1$ colors. If Huv and Kuv both have $(m-1)$-colorings in which u and v have different colors, then we can use these colorings to construct an $(m-1)$-coloring of G. Similarly, if Huv and Kuv both have $(m-1)$-colorings in which u and v have the same color, we can again construct an $(m-1)$-coloring of G. We conclude that in one of them, say Huv, u and v have the same color in every $(m-1)$-coloring; and that in Kuv, u and v have different colors in every $(m-1)$-coloring.

Now consider the graph $H' = Huv + uv$. It cannot be colored in $m-1$ colors, however $H' - uv$ can be. Let xy be any other edge of H'. Then $G-xy$ can be colored in $m-1$ colors. Now Kuv is a subgraph of $G-xy$. Therefore u and v have different colors in this coloring. It follows that $H'-xy$ can be colored in $m-1$ colors. Hence, $H' = Huv + uv$ is an m-critical graph.

Now consider the graph $K' = (Kuv + uv) \cdot uv$. It cannot be colored in $m-1$ colors, as this would determine an $(m-1)$-coloring of Kuv in which u and v have the same color. Let xy be any edge of K'. It corresponds to an edge $x'y'$ of G. $G-x'y'$ can be colored in $(m-1)$ colors. Since Huv is a subgraph of G, it follows that u and v have the same color. This then determines a coloring of $K'-xy$ in $(m-1)$ colors. Hence $K' = (Kuv + uv) \cdot uv$ is also an m-critical graph.

Exercises

11.4.1 Program Algorithm 11.2.1, and compare its performance with a breadth-first or depth-first sequential algorithm.

11.4.2 Program the MAXCLIQUE() algorithm.

11.4.3 Let G be an m-critical graph, and let $v \in V(G)$. Show that G has an m-coloring in which u is the only vertex of color number m.

11.4.4 Let G be an m-critical graph. Apply Mycielski's construction to obtain a graph G'. Either prove that G' is $(m+1)$-critical, or find a counterexample.

11.5 Chromatic polynomials

Suppose that we wish to properly color the complete graph Kn in at most λ colors, where $\lambda \geq n$. Choose any ordering of $V(Kn)$. The first vertex can be colored in A choices. The next vertex in $\lambda-1$ choices, and so on. Thus, the number of ways to color Kn is $\lambda(\lambda-1)(\lambda-2)\ldots(\lambda-n+1)$. This is a polynomial of degree n in λ.

DEFINITION 11.4: The *chromatic polynomial* of a graph G is $\pi(G,\lambda)$, the number of ways to color G in $\leq\lambda$ colors.

In order to show that $\pi(G, \lambda)$ is in fact a polynomial in λ, we use a method that is familiar from counting spanning trees. We first find $\pi(T, \lambda)$ for any tree T, and then give a recurrence for any graph G.

LEMMA 11.8 Let T be a tree on n vertices. Then $\pi(T, \lambda) = \lambda(\lambda-1)^{n-1}$.

PROOF By induction on n. It is certainly true if $n=1$ or $n=2$. Choose a leaf u of T, and let $T' = T-u$. T' is a tree on $n-1$ vertices. Therefore $\pi(T', \lambda) = \lambda(\lambda-1)^{n-2}$. In any coloring of T', the vertex adjacent to u in T has some color. There are $\lambda-1$ colors available for u. Every coloring of T arises in this way. Therefore $\pi(T, \lambda) = \lambda(\lambda-1)^{n-1}$.

Suppose now that G is any graph. Let $uv \in E(G)$. In practice we will want to choose uv so that it is an edge on a cycle.

THEOREM 11.9 $\pi(G, \lambda) = \pi(G-uv, \lambda) - \pi(G \cdot uv, \lambda)$.

PROOF In each coloring of $G-uv$ in $\leq\lambda$ colors, either u and v have different colors, or they have the same color. The number of colorings of $G-uv$ is the sum of these two. If u and v have different colors, then we have a coloring of G in $\leq\lambda$ colors. Conversely, every coloring of G in $\leq\lambda$ colors gives a coloring of $G-uv$ in which u and v have different colors. If u and v have the same color in $G-uv$, then this gives a coloring of $G \cdot uv$. Any coloring of $G \cdot uv$ in $\leq\lambda$ colors determines a coloring of $G-uv$ in which u and v have the same color. We conclude that $\pi(G-uv, \lambda) = \pi(G, \lambda) + \pi(G \cdot uv, \lambda)$.

One consequence of this theorem is that $\pi(G,\lambda)$ is in fact a polynomial of degree n in λ. Now if $n>0$, then $\lambda \mid \pi(G,\lambda)$, since G cannot be colored in $\lambda=0$ colors. Similarly, if $\varepsilon(G)\neq 0$, we conclude that $\lambda(\lambda-1) \mid \pi(G,\lambda)$, since G cannot be colored in $\lambda=1$ color. If G is not bipartite, then it cannot be colored in $\lambda=2$ colors. In this case $\lambda(\lambda-1)(\lambda-2) \mid \pi(G,\lambda)$. In general:

LEMMA 11.10 If χ(G)=m, then λ(λ−1)(λ−2)...(λ−m+1)|π(G,λ).
PROOF *G* cannot be colored in fewer than *m* colors. Therefore *π(G,λ)*=0, for *λ*=1,2,...,*m*−1.
Notice that if *G* contains an *m*-clique *S*, then *χ(G)*≥*m*, so that *λ*(*λ*−1)(*λ*− 2)...(*λ*−*m*+1) *π(G,λ)*. There are *λ*(*λ*−1)(*λ*−2)...(*λ*−*m*+1) ways to color *S* in ≤*λ* colors. The number of ways to complete a coloring of *G*, given a coloring of *S*, is therefore *π(G,λ)*/*λ*(*λ*−1)(*λ*−2)...(*λ*−*m*+1).
Suppose that *G* has a cut-vertex *u*. Let *H* be a connected component of *G*−*u,* and let *Hu* be the subgraph induced by $V(H) \cup \{v\}$. Let *Ku* be the subgraph *G*−*V(H)*. Every coloring of *G* induces colorings of *Hu* and *Ku*, such that *u* has the same color in both. Every coloring of *Hu* and *Ku* occurs in this way. Given any coloring of *Hu*, there are *π(Ku,λ)*/*λ* ways to complete the coloring of *Ku*. It follows that *π(G,λ)*=*π(Hu,λ)π(Ku,λ)*/*λ*.
More generally, suppose that *S* is a separating set of *G* which induces an *m*-clique. Let *H* be a component of *G*−*S*, and let *HS* be the subgraph induced by $V(H) \cup S$. Let *KS* be the subgraph *G*−*V(H)*. We have:
LEMMA 11.11 Let S be a separating set which induces an m-clique in G. Let HSand Ksbe defined as above. Then π(G,λ)=π(HS,λ)π(KS,λ)/λ(λ− 1)(λ−2)...(λ−m+1).
PROOF Every coloring of *G* induces a coloring of *HS* and *KS*. There are *π(HS,λ)* ways to color *HS*. There are *π(KS,λ)*/*λ*(*λ*−1)(*λ*−2)...(*λ*−*m*+1) ways to complete a coloring of *S* to a coloring of *KS*. This gives all colorings of *G*.
There are no efficient means known of computing chromatic polynomials. This is due to the fact that most coloring problems are NP-complete. If *π(G,λ)* could be efficiently computed, we would only need to evaluate it for *λ*=3 to determine if *G* can be 3-colored.

Page 258
Exercises
11.5.1 Find *π*(*C*2*n*, *λ*) and *π*(*C*2*n*+1, *λ*).
11.5.2 Find *π(G, λ),* where *G* is the graph of the cube and the graph of the octahedron.
11.5.3 Let *G′* be constructed from *G* by adding a new vertex joined to every vertex of *G*. Determine *π(G′, λ)* in terms of *π(G, λ)*.
11.5.4 The *wheel Wn* is obtained from the cycle *Cn* by adding a new vertex joined to every vertex of *Cn*. Find *π(Wn, λ)*.
11.5.5 A *unicyclic* graph *G* is a graph formed from a tree *T* by adding a single edge connecting two vertices of *T*. *G* has exactly one cycle. Let *G* be a unicyclic graph on *n* vertices, such that the unique cycle of *G* has length *m*. Find *π(G, λ)*.
11.5.6 Let *G′* be constructed from *G* by adding two new adjacent vertices joined to every vertex of *G*. Determine *π(G′, λ)* in terms of *π(G, λ)*.
11.5.7 Let *G′* be constructed from *G* by adding *k* new mutually adjacent vertices joined to every vertex of *G*. Determine *π(G′, λ)* in terms of *π(G, λ)*.
11.5.8 Let *G′* be constructed from *G* by adding two new non-adjacent vertices joined to every vertex of *G*. Determine *π(G′, λ)* in terms of *π(G, λ)*.
11.5.9 Find *π(Km, m, λ)*.
11.5.10 Let *G* be a graph, and suppose that $uv \notin E(G)$. Show that *π(G,λ)*=*π*(*G*+ *uu, λ*)+*π*((*G*+*uu*)·*uu, λ*). When *G* has many edges, this is a faster way to compute *π(G, λ, λ)* than the method of Theorem 11.9.
11.5.11 Calculate *π*(*Kn*−*uu, λ*), *π*(*Kn*−*uu*−*uw, λ*), and *π*(*Kn*−*uu*−*wx, λ*), where *u, u, w, x* are distinct vertices of *Kn*.
11.5.12 If *G* is a connected graph on *n* vertices, show that *π(Kn, λ)*≤*π(G,λ)*≤ *λ*(*λ*−1)*n*−1, for all *λ*≥0.
11.5.13 Prove that the coefficient of *λn* in *π(G, λ)* is 1, and that the coefficients alternate in sign.

11.6 Edge colorings

A coloring of the edges of a graph *G* is an assignment of colors to the edges. More formally, an *m-edge-coloring* is a mapping from *E(G)* onto a set of *m* colors {1, 2,..., *m*}. The coloring is *proper* if adjacent edges always have different colors. The *edge-chromatic number* or *chromatic index* of *G* is *χ′(G),* the minimum value of *m* such that *G* has a proper m-edge coloring. Notice that in any proper edge-coloring of *G*, the edges of each color define a matching in *G*. Thus, *χ′(G)* can be viewed as the minimum number of matchings into which *E(G)* can be partitioned.
When the edges of a multi-graph are colored, all edges with the same endpoints must have different colors.

Page 259
Consider a vertex *u* of maximum degree in *G*. There must be DEG*(u)* colors incident on *u*. Therefore *χ′ (G)*≥Δ*(G)*. There is a remarkable theorem by Vizing (Theorem 11.14) that states *χ′(G)*≤Δ*(G)*+1 for simple graphs.

Before we come to the proof of Vizing's theorem, first consider the case when *G* is bipartite. A matching *Mi* saturating all vertices of degree Δ*(G)* can be found with Algorithm 7.2.1 (the Hungarian algorithm). Alternating paths can be used to ensure that each vertex of degree Δ*(G)* is saturated. Thus we know that the bipartite graph *G* has a maximum matching saturating every vertex of degree Δ*(G)*. This gives an algorithm for edge-coloring a bipartite graph.

Algorithm 11.6.1: BIPARTITECOLORING*(G)*
comment: Edge-color a graph *G* on *n* vertices
find the degrees of all vertices
i←1
repeat

$$\begin{cases} \text{find a maximum matching } M_i \text{ in } G \\ \quad \text{saturating every vertex of degree } \Delta(G) \\ \text{assign color } i \text{ to the edges of } M_i \\ G \leftarrow G - M_i \\ i \leftarrow i+1 \end{cases}$$

until Δ*(G)*=0

It follows from this algorithm, that when *G* is bipartite, *χ′(G)*=Δ*(G)*.

Suppose that *G* is a *k*-regular graph, edge-colored in *k* colors. Then every color occurs at every vertex of *G*. If *i* and *j* are any two colors, then the *(i, j)*-subgraph is the subgraph of *G* that contains only the edges of color *i* and *j*. Because the *(i, j)*-subgraph is the union of two matchings, it is the disjoint union of alternating cycles. Let $U \subseteq V(G)$. Let *ni* and *nj* denote the number of edges of colors *i* and *j,* respectively, in the edge-cut [*U, V−U*]. Each cycle of the *(i, j)*-subgraph intersects [*U, V−U*] in an even number of edges. Thus we conclude that *ni*+*nj* is even. Therefore *ni*≡*nj* (mod 2). This gives the following parity lemma:

LEMMA 11.12 **(Parity lemma)** *Let G be a k-regular graph, edge-colored in colors* {1, 2,..., *k*}. *Let* $U \subseteq V(G)$. *Let ni denote the number* of edges of color *i in* [*U, V−U*]. *Then n*1≡*n*2≡...≡*nk*(*mod* 2).

Vizing's theorem (Theorem 11.14) is based on an algorithm to edge-color a graph in ≤Δ*(G)*+1 colors. The proof presented is based on that of

FOURNIER [42]. It begins with an arbitrary coloring of *G* in Δ*(G)*+1 colors, and then gradually improves it until it becomes a proper coloring. Given a coloring, let *c(u)* denote the number of colors occurring at vertex $v \in V(G)$. If *c(u)*=DEG*(u),* for all *u,* then the coloring is proper. Otherwise *c(u)*<DEG*(u),* for some *u*. The sum Σ*uc(u)* in an indication of how close an arbitrary coloring is to being a proper coloring.

Suppose first that *G* is arbitrarily colored in two colors.

LEMMA 11.13 If G is a graph that is not an odd cycle, then G has a 2-edge-coloring in which c(u)≥2, *for all vertices u, with DEG(u)*≥2.

PROOF If *G* is Eulerian, choose an Euler tour, and color the edges alternately blue and red along the tour. If *G* is not Eulerian, add a new vertex *u*0 adjacent to every odd degree vertex of *G*. The result is an Eulerian graph. Color it in the same way.

We can use Lemma 11.13 on subgraphs of *G*. Given a proper coloring of *G,* the edges of colors *i* and *j* each define a matching in *G*. Consider the *(i, j)-subgraph*. Each connected component is a path or an even cycle whose colors alternate. If, however, we begin with an arbitrary coloring of *G,* then we want to maximize Σ*uc(u)*. If some component of the *(i, j)*-subgraph is not an odd cycle then by Lemma 11.13, it can be 2-colored so that *c(u)*≥2, for all vertices *u,* with *DEG(u)*≥2.

THEOREM 11.14 **(Vizing's theorem)** *If G is simple, then χ′(G)*≤Δ*(G)*+1.

PROOF We begin by arbitrarily coloring the edges in Δ*(G)*+1 colors. We show that when Σ*uc(u)* is as large as possible, the coloring must be proper. Suppose that the coloring is not proper, and choose a vertex *u* with *c(u)*<DEG*(u)*. Some color *i*0 is missing at *u,* and some color *i*1 occurs at least twice. Let edges *uu*0 and *uu*1 have color *i*1. If color *i*0 is missing at either *u*0 or *u*1, we can recolor one of these edges with color *i*0, thereby increasing Σ*uc(u)*. Hence, we can assume that color *i*0 occurs at both *u*0 and *u*1. Some color is missing at *u*1; call it *i*2. If *i*2 is also missing at *u,* we can recolor *uu*1 with color *i*2, thereby increasing Σu*c(u)*. Hence, let *uu*2 be an edge of color *i*2. Some color is missing at *u*2; call it *i*3. If *i*3 is also missing at *u,* we can recolor *uu*1 with color *i*2, and *uu*2 with color is, thereby increasing Σ*uc(u)*. It follows that *i*3≠*i*0, so that *i*0 occurs at *u*2. We continue in this way, constructing a sequence of edges *uu*1, *uu*2,..., *uuk* of distinct colors *i*1, *i*2..., *ik,* such that color *i*0 occurs at each of *u*1,..., *uk,* and color *ij*+1 does not occur at *uj*. Refer to Figure 11.4. We continue in this fashion generating a sequence *i*0, *i*1,..., of distinct colors until we find a color *ik*+1

that

Page 261

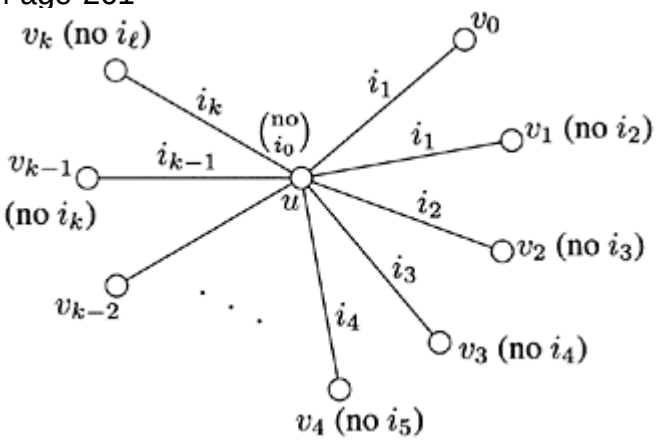


FIGURE 11.4
Vizing's theorem

is not missing at *u*. Thus *ik*+1 has previously occurred in the sequence. Suppose that *ik*+1=*iℓ,* where 1≤*ℓ*≤*k*. Recolor the edges *uu*1, *uu*2,..., *uvℓ*−1 with the colors *ℓ*2, *ℓ*3,..., *iℓ,* respectively. This does not change Σ*uc(u),* because each *c(uj)* is unchanged. Notice that *uuℓ*−1 and *uuℓ* are now both colored *iℓ*. Consider the (*i*0, *iℓ*)-subgraph containing *u,* It contains *uℓ*−1 and *uℓ*. If it is not an odd cycle, it can be recolored so that colors *i*0 and *iℓ* both occur at each of the vertices that have degree exceeding one. This would increase *c(u)* and therefore Σ*uc(u)*. Hence this subgraph can only be an odd cycle, so that *G* contains an (*i*0,*iℓ*)-path from *uℓ*−1 to *uℓ*.

Now recolor the edges *uuℓ, uuℓ*+1, ..., *uuk* with the colors *iℓ*+1,..., *ik, ik*+1=*iℓ,* respectively. Once again Σ*uc(u)* is unchanged. The (*i*0, *iℓ*)-subgraph containing *u* now contains *uℓ*−1 and *uk,* and must be an odd cycle. Hence *G* contains an (*i*0, *iℓ*)-path from *uℓ*−1 to *uk*. This contradicts the previous path found. It follows that the coloring can only be proper, and that *χ′*≤Δ*(G)*+1.

The proof of Vizing's theorem makes use of a *color rotation* at vertex *u,* namely, given a sequence of incident edges *uu*1, *uu*2,..., *uuk* of colors *i*1, *i*2,..., *ik,* respectively, such that *uj* is missing color *ij*+1, for *j*=1,2,...,*k*− 1, and *uk* is missing color *iℓ* where $\ell \in \{1, 2, \ldots, k-1\}$. We then recolor *uu*1, *uu*2,..., *uuℓ*−1 with colors *i*2, *i*3, ..., *iℓ,* respectively. This technique will be used in the edge-coloring algorithm given as Algorithm 11.6.2. The algorithm will build a sequence of vertices *u*1, *u*2, *u*3,... adjacent to *u,* searching for a color rotation.

Graphs *G* for which *χ′(G)*=Δ*(G)* are said to be *Class I graphs*. If *χ′(G)*= Δ*(G)*+1, then *G* is a *Class II graph*. Although there is an efficient algorithm

Page 262

to color a graph in at most Δ*(G)*+1 colors, it is an NP-complete problem to determine whether an arbitrary graph is of Class I or II. A proof of this remarkable result is presented at the end of the chapter.

If *G* is a multigraph with no loops, then the general form of Vizing's theorem states that *χ′(G)*≤Δ*(G)*+*μ(G),* where *μ(G)* is the maximum edge-multiplicity. It can often happen that *μ(G)* is not known. *Shannon's theorem* gives an alternative bound $\chi'(G) \leq \lfloor 3\Delta(G)/2 \rfloor$. The proof presented is due to ORE [93]. It requires two lemmas. Given a proper edge-coloring of *G,* we write *C(u)* for the set of colors present at *u*.

LEMMA 11.15 **(Uncolored edge lemma)** *Let G be a multigraph without loops. Let uu be any edge of G, and let G*−*uu be edge-colored with k colors, and suppose that χ′(G)=k*+1. *Then:*

$$|C(u) \cup C(v)| = k$$

$$|C(u) \cap C(v)| = \text{DEG}(u) + \text{DEG}(v) - k + 2$$

$$|C(u) - C(v)| = k - \text{DEG}(v) + 1$$

$$|C(v) - C(u)| = k - \text{DEG}(u) + 1$$

PROOF Every color missing at *u* is present at *u,* or there would be a color available for *uu,* thereby making *χ′(G)*=*k*. Therefore all colors are present at one of *u* or *u*. This gives the first equation. The colors present at *u* can be counted as

$$|C(u)-C(v)|+|C(u)\cap C(u)|=\text{DEG}(u)-1.$$

Similarly, those present at u are given by

$$|C(u)-C(u)|+|C(u)\cap C(v)|=\text{DEG}(u)-1.$$

Now

$$|C(u) \cup C(v)| = |C(u) - C(v)| + |C(v) - C(u)| + |C(u) \cap C(v)|,$$

so that we can solve these equations for $|C(u)\cap C(u)|$, giving the second equation. The third and fourth equations then result from combining this with the previous two equations.

LEMMA 11.16 **(Ore's lemma)** *Let G be a multigraph without loops. Then:*

$$\chi'(G) \leq \text{MAX}\{\Delta(G), \frac{1}{2}\text{MAX}_{\{u,v,w\}}\{\text{DEG}(u) + \text{DEG}(v) + \text{DEG}(w)\}\},$$

where the second maximum is over all triples of vertices u, u, w such that $v \longrightarrow u \longrightarrow w$.

Page 263

PROOF The proof is by induction on $\varepsilon(G)$. It is clearly true if $\varepsilon(G)\leq 3$. Suppose it holds for all graph with $\varepsilon(G)\leq m$ and consider G with $\varepsilon(G)=m+1$. Let uu be any edge. Delete uu to obtain $G-uu$, for which the result holds. Let $\chi'(G-uu)=k$, and consider a proper k-edge coloring of $G-uu$. If there is a color available for uu, then $\chi'(G)=k$, and the result holds. Otherwise the uncolored edge lemma (Lemma 11.15) applies to G. Pick a color $i \in C(u) - C(v)$ and an edge uw of color i. We first show that $C(v)-C(u) \subseteq C(w)$. Let j $j \in C(v)-C(u)$. If color j is missing at w, then since j is also missing at u, we can recolor edge uw with color j, and assign color i to uu. This results in a k-edge-coloring of G, a contradiction. Therefore $j \in C(w)$, so that $C(v) - C(u) \subseteq C(w)$.

We also show that $C(u) - C(v) \subseteq C(w)$. We know that $i \in C(u) - C(v)$. If there is no other color in $C(u)-C(u)$, we are done. Otherwise, pick also $\ell \in C(u) - C(v)$. If $\ell \notin C(w)$, consider the (ℓ,j)-subgraph H containing u. If H does not contain u, we can interchange colors in H, and assign color ℓ to uu, a contradiction. Therefore H consists of an alternating path from u to u. Interchange colors in H, recolor edge uw with color ℓ, and assign color i to edge uu, again a contradiction. We conclude that $C(u) - C(v) \subseteq C(w)$. We have $|C(u)-C(u)|+|C(u)-C(u)|\leq|C(w)|$. By the uncolored edge lemma, this means that $\text{DEG}(u)\geq 2k-\text{DEG}(u)-\text{DEG}(u)+2$. Therefore $\text{DEG}(u)+\text{DEG}(u)+\text{DEG}(w)\geq 2k+2$, so that $k+1 \leq \frac{1}{2}(\text{DEG}(u)+\text{DEG}(v)+\text{DEG}(w))$. The result then holds for G, as required.

THEOREM 11.17 **(Shannon's theorem)** *Let G be a multigraph without loops. Then* $\chi'(G) \leq \lfloor 3\Delta(G)/2 \rfloor$.

PROOF By Ore's lemma:

$$\chi'(G) \leq \text{MAX}\{\Delta(G), \frac{1}{2}\{\Delta(G) + \Delta(G) + \Delta(G)\}\} = 3\Delta(G)/2.$$

Vizing's theorem results in an algorithm to color the edges of G in at most $\Delta(G)+1$ colors. Let the vertices of G be numbered 1, 2,..., n, and let the colors be {0, 1,..., $\Delta(G)$}. There are two somewhat different ways in which this can be programmed. In both methods, we begin by assigning color 0 to every edge. We then take each vertex u in turn and gradually improve the coloring local to u. Both methods involve constructing color rotations and alternating paths, as can be expected from the proof of Vizing's theorem.

One method uses alternating paths to improve the coloring local to u. When this does not succeed, a color rotation is used. The second method uses color

---

Page 264

rotations to improve the coloring local to u, and builds alternating paths when the color rotation does not succeed. We present the first method here. The algorithm keeps the adjacency lists in partially sorted order—all edges of any color i are consecutive, and any colors which occur only once at vertex u are stored at the end of the adjacency list for u. This is certainly true initially when all edges have color 0. This allows the algorithm to easily determine whether there is a repeated color at u by simply taking the first two adjacent vertices.

Algorithm 11.6.2: EDGECOLOR(G, n)

comment: { Initially the edges all have color 0.
Convert the coloring to a proper coloring in
at most $\Delta(G) + 1$ colors.

```
                    for u←1 to n
                    ⎧ v ← 1st vertex joined to u
                    ⎪ w ← 2nd vertex joined to u
                    ⎪ while Color[uv] = Color[uw]
                 do ⎨      ⎧ comment: u has a repeated color
                    ⎪   do ⎨ ColorBFS(u)
                    ⎪      ⎪ v ← 1st vertex joined to u
                    ⎩      ⎩ w ← 2nd vertex joined to u
```

Algorithm 11.6.2 takes the vertices in order 1,2,..., *n*. When vertex *u* is considered, it assumes that the subgraph induced by {1, 2,..., *u*−1} has been properly colored in at most Δ*(G)*+1 colors. This is certainly true when *u*=1. If the first two incident edges have the same color *k,* it attempts to recolor an incident edge of color *k* as follows. Let *i* be a color missing at vertex *u*.

1. If any edge *uu* of color *k* is such that vertex *u* is also missing color *i,* then *uu* is assigned color *i*.

2. Or if any edge *uu* of color *k* is such that *u*>*u,* then *uu* is assigned color *i*. This improves the coloring at *u,* and does not affect the subgraph induced by {1, 2,..., *u*−1}.

3. Otherwise all incident edges *uu* of color *k* are such that *u*<*u,* and *u* has exactly one incident edge of color *i*. The algorithm constructs paths from *u* whose edges alternate colors *k* and *i,* using a breadth-first search.

4. If a path *P* to a vertex *u* is found, such that *u* is missing either color *i* or *k,* then the colors on the path are reversed.

5. Or if a path *P* to a vertex *u* is found, such that *u*>*u,* then the colors on the path are reversed. In each of these cases, the coloring is improved at

vertex *u,* and a valid coloring of the subgraph induced by {1, 2,..., *u*−1} is maintained.

6. Otherwise, each alternating path of colors *i* and *k* induces an odd cycle containing *u*. There must be an even number of incident edges of color *k*. If there are more than two such edges, let *uu* be one of them. Some color *j* is missing at *u*. We recolor edge *uu* with color *j,* and reverse the colors on the remainder of the odd cycle containing *uu*. This improves the coloring at *u,* and maintains a valid coloring in the subgraph induced by {1, 2,..., *u*−1}.

7. Otherwise, there are exactly two incident edges of color *k*. The algorithm searches for a color rotation—it constructs a sequence of incident edges *uu*1, *uu*2,..., *uum* of colors *i*1=*k, i*2,..., *im,* respectively, such that *uj* is missing color *ij*+1, for *j*=1, 2,..., *m*−1, and *um* is missing color *iℓ* where $\ell \in \{1, 2, \ldots, m-1\}$. If color *iℓ* occurs an even number of times at *u,* then the edges *uu*1, *uu*2,..., *uuℓ*−1 are recolored *i*2, *i*3,..., *iℓ*. This creates an odd number of edges of color *iℓ* incident at *u*. The edges of color *iℓ* are moved to the front of the adjacency list of *u*. The next iteration of the algorithm is certain to improve the coloring at *u*.

8. Otherwise, color *iℓ* occurs an odd number of times. If it occurs at least three times, the edges of color *iℓ* are moved to the front of the adjacency list of *u*. The next iteration of the algorithm is certain to improve the coloring at *u*.

9. Otherwise, color *iℓ* occurs exactly once at *u*. The algorithm builds an alternating path *P* of colors *i* and *iℓ* from *um*. If *P* leads to *uℓ,* we recolor the edges *uu*1, *uu*2,..., *uuℓ*−1 with colors *i*2, *i*3,..., *iℓ* and move the edges of color *iℓ* to the head of the adjacency list of *u*. If *P* does not lead to *uℓ,* we recolor the edges *uu*1, *uu*2,..., *uum* with colors *i*2, *i*3,..., *im*, *iℓ,* and move the edges of color *iℓ* to the head of the adjacency list of *u*. In either case, the next call to *ColorBFS* is guaranteed to improve the coloring at *u*.

Thus, it is always possible to improve the coloring at *u*. Each time that an edge *uw* is recolored, it is necessary to adjust the order of the adjacency lists of *u* and *w* to ensure that they remain partially sorted. If *u*<*u,* we know that every color incident on *u* occurs exactly once at *u*. But if *u*>*u,* it will be necessary to change some links in the linked list. After the coloring has been completed for vertex *u*=*n,* the entire graph has a valid coloring.

The algorithm uses a global array *ScanQ* for breadth-first searches. The number of vertices currently on the *ScanQ* is denoted by *QSize*. The *ScanQ* accumulates vertices that form alternating paths. The previous vertex to *u* in such a path is denoted *PrevPt[u]*. We set *PrevPt[u]*←0 initially. When it is necessary to reinitialize the *ScanQ,* the entire array is not reset—only the vertices *u* currently on the *ScanQ* have *PrevPt[u]* reset to 0, and *QSize* reset to 0. The color of edge *uu* is indicated by $\mathrm{Color}\langle uv \rangle$ in the pseudo-code, but in an actual program it would

Page 266
be stored as a field in the adjacency list structure of the graph. The frequency of color *i* at a given vertex is stored in an array as *Freq[i]*.

Algorithm 11.6.3: COLORBFS*(u)*

comment: 2 or more incident edges at *u* have the same color—recolor one

i←a missing color at *u*

u←first vertex joined to *u*

$k \leftarrow Color\langle uv \rangle$

comment: initialize the *ScanQ* with all adjacent vertices of color *k*

QSize←0

while $Color\langle uv \rangle = k$

do
- **if** (color i is missing at v) **or** $(v > u)$
  - **then**
    - RECOLOREDGE(u, v, i)
    - re-initialize the *ScanQ* and *PrevPt* arrays
    - **return**
- $QSize \leftarrow QSize + 1$
- $ScanQ[QSize] \leftarrow v$
- $PrevPt[v] \leftarrow u$
- $v \leftarrow$ next vertex adjacent to u

comment:
- each v in *ScanQ* satisfies $Color\langle uv \rangle = k$ and $v < u$
- build paths whose edges are alternately colored i and k.

w←ALTERNATINGPATH*(u, i, k)*

if *w≠0*

then
- RECOLORPATH(w, i, k)
- **return**

comment:
- it was not possible to re-color any alternating path.
- the vertices on *ScanQ* induce odd cycles of colors i and k.

u←*ScanQ*[1]

if *Freq[k]*>2

then
- $j \leftarrow$ a color missing at v
- RECOLOREDGE(u, v, j)
- RECOLORCYCLE(v, i, k)
- **return**

comment:
- if this point is reached, it is necessary
- to search for a color rotation

COLORROTATION*(u, i)*

The algorithm uses two procedures RECOLOREDGE() and RECOLORPATH() to recolor an edge or alternating path. They are described as follows:

Page 267

procedure RECOLOREDGE*(u,u,j)*

comment: Recolor edge *uu* with color *j*.

Color[uu]←*j*

adjust the order of the adjacency lists of *u* and *u* as necessary

procedure RECOLORPATH(*w*, *i*1, *i*2)

comment:
- Recolor the edges of an alternating path beginning at w
- The edges currently alternate colors i_1 and i_2.

u←*PrevPt[w]*

while *w*≠0

```
do { j ← i1 + i2 − Color[vw]
     RECOLOREDGE(v, w, j)
     w ← v
     v ← PrevPt[w]
```

re-initialize the *ScanQ* and *PrevPt* arrays

ALTERNATINGPATH(*u,i*1,*i*2) is a procedure which builds alternating paths of colors *i*1 and *i*2 beginning from the vertices on the *ScanQ*. When it is called, the vertices *u* currently on the *ScanQ* are all known to have an incident edge of color *i*1. If *uw* has color *i*1, and *w* has no incident edge of color *i*2, or if *w*>*u,* then vertex *w* is returned, so that the colors on the path from *w* to *u* can be reversed. This improves the coloring at *u,* and maintains the coloring on the subgraph induced by {1, 2,..., *u*−1}.

When a suitable alternating path to a vertex *w* is found, it can be traced by following the *PrevPt*[·] values. When an edge *uw* is recolored, it may be necessary to adjust the ordering of the adjacency lists of *u* and *w* so that the partial order is maintained.

COLORROTATION *(u,i)* searches for a color rotation from a vertex *u* which is missing color *i* by building a sequence of vertices *u*1, *u*2,..., *um* on the *ScanQ* such that each edge *uuj* has color *ij* and *uj* is missing color *ij*+1. The pseudocode for it is left to the reader.

Algorithm 11.6.4: ALTERNATINGPATH(*u, i*1, *i*2)

```
comment: { Build alternating paths of colors i1 and i2,
           beginning from vertices currently on the ScanQ.
           Return the first w encountered which is missing color i1 or i2
           or which satisfies w > u, if there is such a w.

NewQSize←QSize
j←1
while j≤QSize
do { comment: extend alternating paths using edges of color i1
     repeat
       { v ← ScanQ[j]      "j^th vertex on ScanQ"
         w ← a vertex of color i1 adjacent to v
         if (color i2 is missing at w) or w < u
           then return (w)
         if w ∉ ScanQ
           then { NewQSize ← NewQSize + 1
                  ScanQ[NewQSize] ← w
                  PrevPt[w] ← v
         j ← j + 1
     until j > QSize
     QSize ← NewQSize
     swap(i1, i2)
return (0) "no suitable alternating path was found"
```

11.6.1 Complexity

It is somewhat difficult to organize the data structures to allow an efficient implementation of this algorithm. The reason is that the program often needs to find an edge *uu* incident on *u* of a given color *i,* or to determine whether *u* is missing color *i*. This can be done in constant time if an *n*×*n* array is stored for which entry *[u, i]* is a pointer to a linked list of all incident edges of color *i*. This array will be quite sparse for most graphs. With this data structure, the algorithm will be quite efficient.

The main program takes each vertex *u* in turn. If there is a repeated color at *u,* COLORBFS(u) is called. It will usually succeed in recoloring one of the incident edges at u, taking only a constant number of steps, so that *O*(DEG*(u)*) steps are usually needed to make *c(u)*=DEG*(u)*. But as more and more of the graph is colored, it becomes necessary to construct alternating paths to improve the coloring at *u*. Let *G(u)* denote the subgraph induced by {1, 2,..., *u*}. The

Page 269
alternating paths constructed are contained in $G(u)$. A breadth-first search to construct the alternating paths takes $O(\varepsilon(G(u)))$ steps, provided that the edge of color i incident on a vertex u can be found in a constant number of steps.
Since $\varepsilon(G(u)) \leq \varepsilon(G)$, we determine that if a color rotation is not required, that COLORBFS(u) will take at most $O(\varepsilon)$ steps. If a color rotation is constructed, it requires at most $O(\text{DEG}(u))$ steps to build the sequence $uu1$, $uu2$,..., uum of colors $i1=k$, $i2$,..., im. The maximum number of steps are executed when an alternating path must be constructed from um, taking $O(\varepsilon(G(u)))$ steps. If COLORBFS(u) does not succeed in improving the coloring at u on a given call, it is guaranteed to succeed on the following call. We conclude that it takes at most $O(\text{DEG}(u) \cdot \varepsilon)$ steps to make $c(u)=\text{DEG}(u)$. The total complexity is therefore at most $O(\varepsilon 2)$.

Exercises

11.6.1 Describe an algorithm using alternating paths in a bipartite graph which finds a maximum matching saturating all vertices of degree $\Delta(G)$.
11.6.2 Work out the complexity of the bipartite coloring algorithm.
11.6.3 Program the bipartite edge-coloring algorithm.
11.6.4 Show that an edge coloring of G gives a vertex coloring of $L(G)$.
11.6.5 Determine χ' for the Petersen graph.
11.6.6 Show that when the inverter shown in Figure 11.6 is edge-colored in three colors, one of the two pairs of edges $\{a, b\}, \{c, d\}$ has the same color, and the other pair has different colors.

11.7 NP-completeness

In this section we show that several coloring-related problems are NP-complete.

Problem 11.1: 3-Colorability
Instance: a graph G.
Question: is $\chi(G) \leq 3$?

We transform from 3-Sat. Consider an instance of 3-Sat containing variables $u1$, $u2$,..., un, with complements $\bar{u}1$, $\bar{u}2$,..., $\bar{u}n$. Suppose that there are m clauses, denoted $c1$, $c2$,..., cm. Construct a graph G as follows. G will have vertices $u1$, $u2$,..., un, $\bar{u}1$, $\bar{u}2$,..., $\bar{u}n$, $x1$, $x2$,..., xn, $\overline{x}_1, \overline{x}_2, \ldots, \overline{x}_n$, and

Page 270
$c1$, $c2$,..., cm. Each ui is joined to $\bar{u}i$ and to xi. Each $\bar{u}i$ is joined to $\overline{x}_i$. There will also be triangles $T1$, $T2$, ..., Tm where Ti corresponds to clause ci, and vertices denoted B and W (blue and white). Each ui and $\bar{u}i$ is joined to W. Each ci is joined to B. A clause like $c1=(u1+\bar{u}2+\bar{u}3)$ has the corresponding vertex $c1$ joined to $x1$, $\overline{x}_2$, and $\overline{x}_3$. The triangle $T1$ corresponding to $c1$ has its three corners joined to the same vertices as $c1$. The construction is illustrated in Figure 11.5. We will color G in colors red, white, and blue.

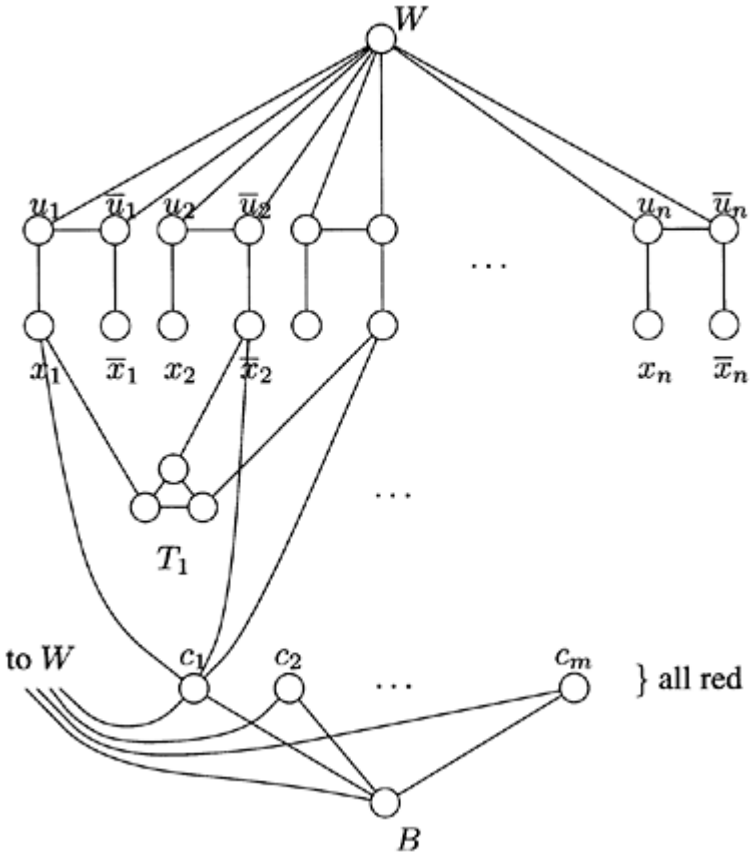


FIGURE 11.5
Graph 3-Colorability is NP-complete

Suppose first that *G* has a proper 3-coloring. Without loss of generality, we can assume that vertex *B* is colored blue, and that vertex *W* is colored white. It follows that all vertices *ci* are colored red. Each triangle *Ti* must use all three colors, so that *Ti* has a vertex colored white, which is adjacent to some *xj* or $\overline{x}_j$, which must be colored blue (since *ci* is also adjacent, and it is red). The adjacent *uj* (or *ūj*) can then only be colored red. Only one of *uj* and *ūj* can be red—the other must be blue, with corresponding *xj* or $\overline{x}_j$ white. It follows that every clause *ci* contains a literal *uj* or *ūj* that is colored red. Take these as the variables assigned the value true. The result is an assignment of values satisfying every clause.

Conversely, if there is an assignment of values satisfying every clause, color each *uj* or *ūj* red if its value is true, and color its complement blue. If *uj* is red, color the corresponding *xj* blue; otherwise color it white. The corner of *Ti* corresponding to *uj* or *ūj* is colored white. The other corners of *Ti* can be red or blue. The result is a 3-coloring of *G*.

We conclude that a polynomial algorithm which solves 3-Colorability could also be used to solve 3-Sat. Since 3-Sat is NP-complete, so is 3-Colorability.

Problem 11.2: Clique

Instance: a graph *G* and an integer *k*.

Question: does *G* have a clique with $\geq k$ vertices (i.e., is $\overline{\alpha}(G) \geq k$)?

We transform from Problem 9.5.9.4 Vertex Cover. Recall that a vertex cover in a graph *G* is a subset $U \subseteq V(G)$ such that every edge has at least one endpoint in *U*. Given an integer *k* the Vertex Cover problem asks: does *G* have a vertex cover with $< k$ vertices?

If *U* is a vertex cover, the set $\overline{U} = V(G) - U$ is an independent set in *G*. Hence $\overline{U}$ induces a clique in $\overline{G}$. If *G* has *n* vertices, and *U* has $\geq m$ vertices, then *U* is a vertex cover with $\leq n-m$ vertices. Thus, given an instance of the Vertex Cover problem, we construct $\overline{G}$, and ask whether it has a clique with at least $n-m$ vertices. If the anwer is yes, then *G* has a vertex cover with at most *m* vertices. We conclude that a polynomial algorithm which solves Clique could also be used to solve Vertex Cover. It follows that Clique is NP-complete.

Problem 11.3: Chromatic Index

Instance: a graph *G*.

Question: is $\chi'(G)=\Delta(G)$?
There is an ingenious construction of Holyer proving that it is NP-complete to determine whether a 3-regular graph is of Class I or II. Holyer's construction is based on the graph shown in Figure 11.6, which he calls an *inverting component,* or *inverter*. The inverter was originally discovered by Loupekine. It consists of two 5-cycles sharing two common consecutive edges. Edges *a, b, c, d, e* are attached to five of the vertices.
A 5-cycle requires at least three colors. Consider any 3-edge-coloring of the inverter. Apply the parity condition (Lemma 11.13) to the set of seven vertices shown in the diagram. We determine that in any 3-edge coloring, that three of the five edges *a, b, c, d, e* must have the same color, and that the other two have distinct colors, since 5 can only be written as 3+1+1, as a sum of three odd numbers. We then find that *a, e,* and *c* cannot all have the same color, so that

Page 272

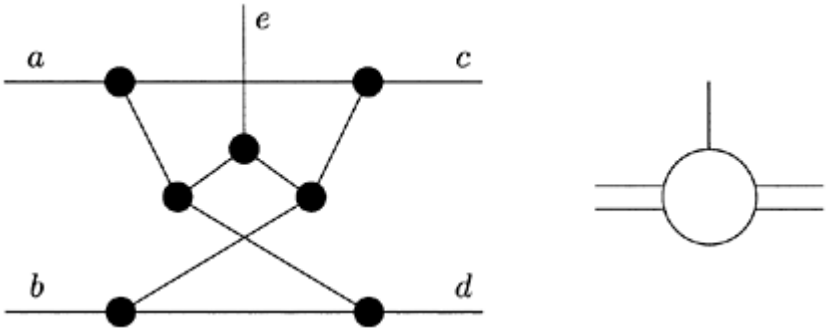


FIGURE 11.6
The inverter and its schematic representation (Used with permission of the SIAM Journal of Computing)

in any 3-edge-coloring, either *a* and *b* have the same color, and *c, d, e* have three distinct colors; or by symmetry, *c* and *d* have the same color, and *a, b, e* have three distinct colors. The inverter is represented schematically in Figure 11.6, as a circle with two pairs of inputs (or outputs) and a fifth input (or output).
Holyer transforms from 3-Sat to Chromatic Index. Consider an instance of 3-Sat with clauses $c1, c2, \ldots, cm$ involving variables $u1, u2, \ldots, un$ and their complements $\bar{u}i$ Each variable is either true or false. A value of true is represented in the edge coloring of an inverter, as an input pair of the same color. A value of false is represented as an input pair of different colors. In every 3-edge-coloring of the inverter, a value of true is inverted to a value of false, or vice versa.
A clause of three variables, for example, $(ui+uj+\bar{u}k)$, is represented by three inverters tied together by a 7-cycle, as shown in Figure 11.7. Note the edges marked *a, b, x, y*. It is easy to see that *a* and *b* have the same color if and only if *x* and *y* have the same color. Because of the 7-cycle, at least one of the three inverters must have two outside inputs the same color. This will be used to indicate that at least one of the variables *ui, uj,* and $\bar{u}k$ in the above clause must have a value of true (i.e., every clause will be satisfied).
Now a given variable *ui,* and/or its complement $\bar{u}i$ will appear in several clauses. It must have the same value in all clauses. In order to ensure this, the 7-cycles corresponding to clauses containing *ui* or $\bar{u}i$ are also tied together. Holyer constructs a cycle of inverters, with two inverters for each occurrence of *ui* or $\bar{u}i$ in the clauses. For example, if *ui* and $\bar{u}i$ occur a total of four time in clauses $c1, c2, c3, c4$, the cycle of Figure 11.8 is constructed. Each pair of inverters corresponds to a clause containing *ui* or $\bar{u}i$. Notice that there are a total of six inputs connecting a pair of inverters to the rest of the graph. By the parity lemma, if this graph is 3-edge-colored, the possible color combinations for the six inputs are 6+0+0, 4+2+0, or 2+2+2. If one pair of external inputs represents

Page 273

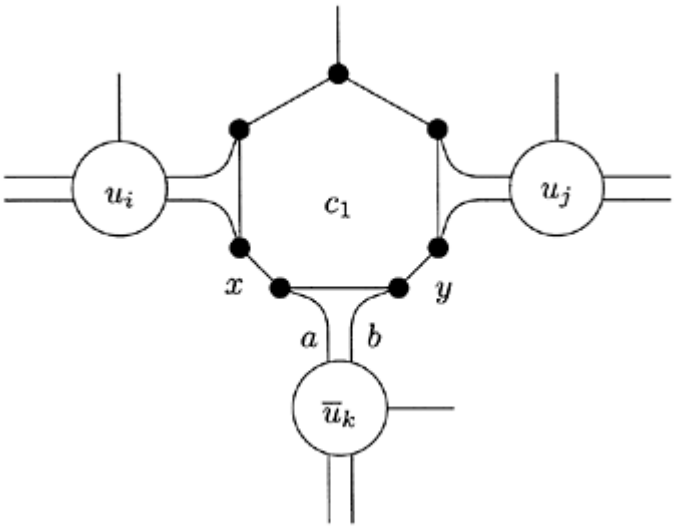


FIGURE 11.7
The representation of a clause *c*1 = (*ui* + *uj* + *ūk*) of 3-Sat (Used with permission of the SIAM Journal of Computing)

true, then by the properties of an inverter, the opposite pair also represents true. Then the parity lemma implies that the remaining two inputs also represent true. It follows that all pairs of inverters in the diagram have all external input pairs representing true. Consequently, if any one pair represents false, they all represent false. This mechanism is used to guarantee that *ui* has the same value in all clauses.

We now put these ideas together. We are given an instance of 3-Sat. For each clause *cj,* three inverters are tied together using a 7-cycle, as in Figure 11.7. For each variable *ui,* a cycle of pairs of inverters is constructed as in Figure 11.8. The input corresponding to *cj* in this structure is connected to the input corresponding to *ui* in the 7-cycle component corresponding to *cj*. If the clause contains *ūi* rather than *ui,* then another inverter is placed between the two before connecting. The result is a graph containing a subgraph for each *ui* and for each *Cj,* tied together through their inputs. There are still a number of inputs not connected to anything. In order to complete the construction and have a 3-regular graph, a second copy of this graph is constructed. The corresponding unused inputs of the two copies are connected together. The result is a 3-regular graph *G*. In any 3-edge-coloring of *G,* every clause is guaranteed to have at least one variable representing true. All occurrences of each variable are guaranteed to have the same value. The result is an assignment of values to the variables solving the 3-Sat instance. We conclude that if we could determine whether $\chi'(G)$=3, we could solve 3-Sat. Since

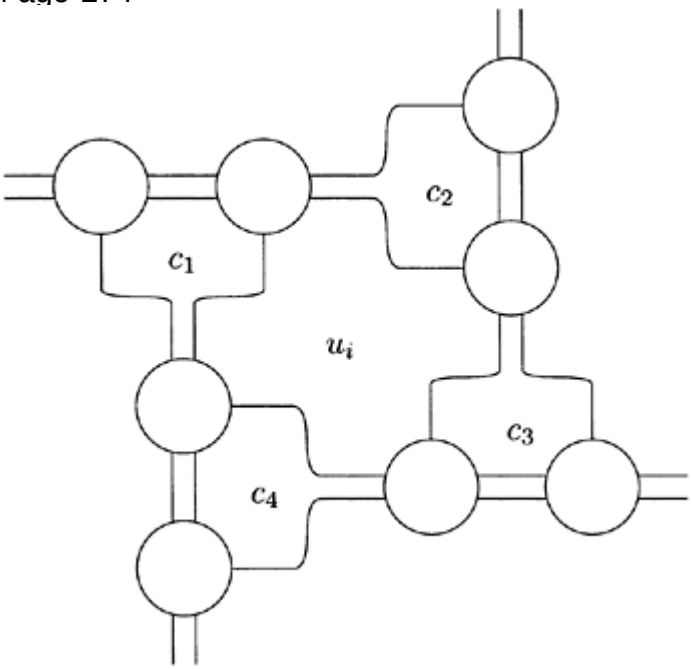


FIGURE 11.8

A pair of inverters for each clause containing u_i or $\bar{u}_i$ (Used with permission of the SIAM Journal of Computing)

3-Sat is NP-complete, we conclude that Chromatic Index is also NP-complete.

11.8 Notes

The DEGREESATURATION() algorithm is from BRELAZ [20]. A very good paper on the limitations of the sequential algorithm is JOHNSON [69]. A very readable survey of chromatic polynomials appears in READ and TUTTE [101]. See also TUTTE [119], where the word *chromial* is coined for chromatic polynomial. The proofs of the uncolored edge lemma and Ore's lemma (Lemmas 11.15 and 11.16) are based on those of BERGE [14]. A very efficient edge-coloring algorithm based on Vizing's theorem was developed by ARJOMANDI [7]. The proof of the NP-completeness of Chromatic Index is based on HOLYER [64]. Figures 11.6, 11.7 and 11.8 are modified diagrams based on those appearing in Holyer's paper. They are used with the permission of the SIAM Journal of Computing.

Page 275

12

Planar Graphs

12.1 Introduction

A graph *G* is *planar* if it can be drawn in the plane such that no two edges intersect, except at a common endpoint. The vertices of *G* are represented as points of the plane, and each edge of *G* is drawn as a continuous curve connecting the endpoints of the edge. For example, Figure 12.1 shows a planar graph (the cube), and a planar drawing of the same graph. Although the cube is planar, the drawing on the left is not a planar drawing. These drawings both have straight lines representing the edges, but any continuous curve can be used to represent the edges. We shall often find it more convenient to represent the vertices in planar drawings as circles rather than points, but this is just a drawing convenience.

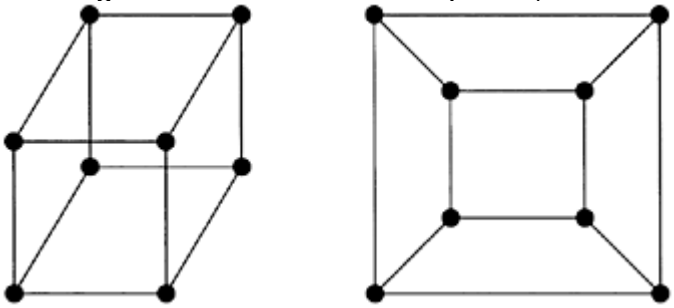


FIGURE 12.1
Two drawings of the cube

Page 276

12.2 Jordan curves

Any closed, continuous, non-self-intersecting curve *C* drawn in the plane divides the plane into three regions: the points inside *C,* the points outside *C,* and the points of *C* itself. This is illustrated in Figure 12.2. We are relying on an intuitive understanding of the words "plane", "curve", "continuous", "region", etc. Exact mathematical definitions of these ideas would require a lengthy excursion into topology. An intuitive understanding should suffice for this chapter. Notice that the interior of *C,* denoted INT*(C),* is bounded, since it is enclosed by *C,* and that the exterior, denoted EXT*(C),* is unbounded, since the plane is unbounded. If *u* is any point in INT*(C),* and $v \in \text{EXT}(C)$, then *any continuous curve with endpoints u and u must intersect C in some point*. This fact is known as the *Jordan curve theorem*. It is fundamental to an understanding of planarity.

DEFINITION 12.1: A closed, continuous, non-self-intersecting curve *C* in a surface is called a *Jordan curve*.

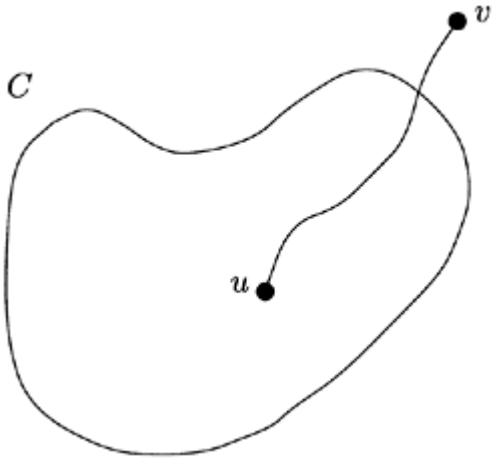


FIGURE 12.2
The Jordan curve theorem

Let *G* be any graph. We would like to construct a planar embedding of *G,* if possible. That is, we want to map the vertices of *G* into distinct points in the plane, and the edges of *G* into continuous curves that intersect only at their endpoints. Let *ψ* denote such a mapping. We write *Gψ* to indicate the image of *G* under the mapping *ψ*. Let *C* be any cycle in *G*. If *ψ* is a planar embedding of *G,* then *ψ* maps *C* onto *Cψ,* a Jordan curve in the plane. For example, consider *G*=*K*5. Let *V*(*K*5)=*{u, w, x, y, z}*. If *K*5 were planar, the cycle *C*= *(x, y, z)* must embed as a Jordan curve *Cψ* in the plane. Vertex *u* is either in

INT*(Cψ)* or EXT*(Cψ)*. Without loss of generality, we can place *u* in INT*(Cψ),* as in Figure 12.3. The paths *ux, uy,* and *uz* then divide INT*(Cψ)* into three smaller regions, each bounded by a Jordan curve. We cannot place *u* in EXT*(Cψ),* as we then cannot embed the path *uu* without crossing *Cψ*. We cannot place *u* in any of the smaller regions in INT*(Cψ),* for in each case there is a vertex outside the Jordan curve bounding the region that cannot be reached. We conclude that *K*5 cannot be embedded in the plane. *K*5 is a *non-planar* graph. We state this as a lemma.

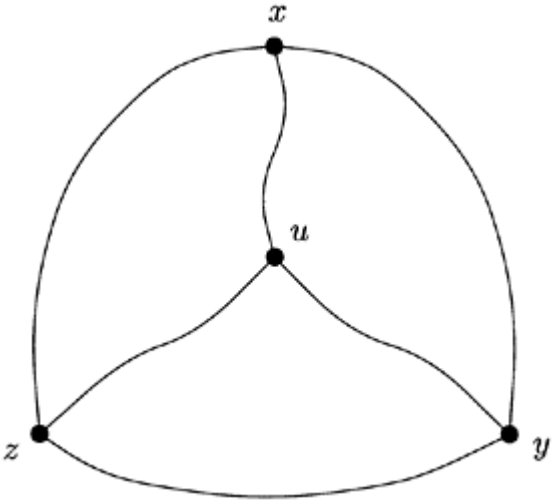


FIGURE 12.3
Embedding *K*5

*LEMMA 12.1 K*5 *and K*3 ,3 *are non-planar graphs*.

PROOF The proof for *K*5 appears above. The proof for *K*3 ,3 is in Exercise 12.3.1.

12.3 Graph minors, subdivisions

The graphs *K*5 and *K*3, 3 are special graphs for planarity. If we construct a graph from *K*5 by replacing one or more edges with a path of length ≥2, we obtain a *subdivision* of *K*5. We say that the edges of *K*5 have been *subdivided*.

DEFINITION 12.2: Given a graph *G,* a *subdivision* of *G* is any graph obtained from *G* by replacing one or more edges by paths of length two or more.

It is clear that any subdivision of *K*5 or *K*3, 3 is non-planar, because *K*5 and *K*3,3 are non-planar. It is apparent that vertices of degree two do not affect the planarity of a graph. The inverse operation to subdividing an edge is to *contract* an edge with an endpoint of degree two.
DEFINITION 12.3: Graphs *G*1 and *G*2 are *topologically equivalent* or *homeomorphic,* if *G*1 can be transformed into *G*2 by the operations of subdividing edges and/or contracting edges with an endpoint of degree two.
We will denote by *TK*5 any graph that is topologically equivalent to *K*5. Similarly, *TK*3, 3 denotes any graph that is topologically equivalent to *K*3, 3. In general, *TK* denotes a graph topologically equivalent to *K,* for any graph *K*. If *G* is a graph containing a subgraph *TK*5 or *TK*3, 3, then *G* must be non-planar. Kuratowski's theorem states that this is a necessary and sufficient condition for a graph to be non-planar. We will come to it later.
If *G* is a planar graph, and we delete any vertex *u* from *G,* then *G*−*u* is still planar. Similarly, if we delete any edge *uu,* then *G*−*uu* is still planar. Also, if we contract any edge *uu* of *G,* then *G*−*uu* is still planar. Contracting an edge can create parallel edges or loops. Since parallel edges and loops do not affect the planarity of a graph, loops can be deleted, and parallel edges can be replaced by a single edge, if desired.
DEFINITION 12.4: Let *H* be a graph obtained from *G* by any sequence of deleting vertices and/or edges, and/or contracting edges. *H* is said to be a *minor* of *G*.
Notice that if *G* contains a subgraph *TK*5, *K*5 is a minor of *G,* even though *K*5 need not be a subgraph of *G*. For we can delete all vertices and edges which do not belong to the subgraph *TK*5, and then contract edges to obtain *K*5. Similarly, if *G* has a subgraph *TK*3, 3, then *K*3, 3 is a minor of *G,* but need not be a subgraph. Any graph having *K*5 or *K*3, 3 as a minor is non-planar. A special case of minors is when a graph *K* is subdivided to obtain *G*.
DEFINITION 12.5: Let *G* contain a subgraph that is a subdivision of a graph *K,* where $\delta(K) \geq 3$. Then *K* is said to be a *topological minor* of *G*.
LEMMA 12.2 If H is a minor of K, and K is a minor of G, then H is a minor of G.
PROOF This follows from the definition.
A consequence of this lemma is that the relation of being a graph minor is a partial order on the set of all graphs.

The inverse operation to contracting an edge whose endpoints have degree three or more is *splitting* a vertex.
DEFINITION 12.6: Let *G* be any graph with a vertex *u* of degree at least three. Let *u* be adjacent to vertices {*u*1, *u*2,..., *uk*}. Construct a graph G_v^+ by *splitting* vertex *u:* replace *u* with two new vertices *u*1 and *u*2. Join *u*1 to $\ell 1 \geq 2$ of {*u*1, *u*2,..., *uk*}, and join *u*2 to $\ell 2 \geq 2$ of them, such that together, *u*1 and *u*2 are adjacent to all of these vertices. Then join *u*1 to *u*2.
In any graph G_v^+ resulting from splitting vertex *u,* *u*1 and *u*2 both have degree at least three, and $G_v^+ \cdot v_1 v_2 = G,$ that is, splitting vertex *u* is an inverse operation to contracting the edge *u*1*u*2. Notice that *G* is a minor of G_v^+. Splitting a vertex is illustrated for *G*=*K*5 in Figure 12.4. The following lemma shows that *K*5 and *K*3, 3 are very closely related graphs.

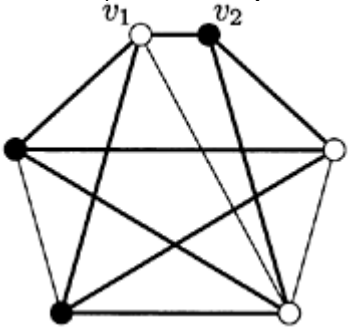


FIGURE 12.4
Splitting a vertex of *K*5
*LEMMA 12.3 Let G be any graph obtained by splitting a vertex of K*5. *Then G contains a subgraph TK*3 ,3.
PROOF Let *u*1 and *u*2 be the two vertices resulting from splitting a vertex of *K*5. Each has at least degree three. Consider *u*1. It is joined to *u*2. Together, *u*1 and *u*2 are joined to the remaining four vertices of *G,* and each is joined to at least two of these vertices. Therefore we can choose a partition of these four vertices into *x, y* and *w, z* such that $v_1 \longrightarrow x, y$ and $v_2 \longrightarrow w, z.$ Then *G* contains a *K*3,3 with bipartition *u*1, *w, z* and *u*2, *x, y,* as illustrated in Figure 12.4.
In the previous example, it was convenient to form a minor *K*5 of *G* by first deleting a subset of vertices

and/or edges, and then contracting a sequence of edges to obtain $K5$. All minors of G can be obtained in this way, as shown by the following lemma:

Page 280

LEMMA 12.4 Suppose that G has a minor H. Then H can be obtained by first deleting a subset of vertices and/or edges of G, and then contracting a sequence of edges.

PROOF Let $G0, G1,\ldots, Gk$ be a sequence of graphs obtained from G, where $G0=G$ and $Gk=H$, such that each Gi, where $i\geq 1$, is obtained from $Gi-1$ by deleting a vertex, deleting an edge, or contracting an edge. If all deletions occur before contractions, there is nothing to prove. So let Gi be the first graph obtained from $Gi-1$ by the deletion of an edge or vertex. Without loss of generality we can assume that $i\geq 2$, and that $G1,\ldots, Gi-1$ were obtained by contracting edges $e1,\ldots, ei-1$, where ej is an edge of $Gj-1$. Let $ei-1=u1u2$.

Suppose first that $Gi=Gi-1-u$, for some vertex u. If u is the result of identifying $u1$ and $u2$ when $ei-1$ is contracted, then we can replace $Gi-1$, Gi in the sequence of graphs by G'_{i-1}, G'_i, where G'_{i-1} and G'_i are obtained by deleting $u1$, and then $u2$ from $Gi-2$. If u is not the result of identifying $u1$ and $u2$, we can interchange the order of Gi and $Gi-1$ by deleting u before contracting $ei-1$. In each case, we obtain an equivalent sequence of graphs with only $i-1$ edge contractions preceding a deletion. The number of edges contracted does not increase, and the final result is still H.

Suppose now that $Gi=Gi-1-uu$, for some edge uu. We know that $Gi-1=Gi-2\cdot ei-1$. We can reverse the order of the two operations, and delete uu before contracting $ei-1$, thereby replacing $Gi-1$, Gi with graphs G'_{i-1}, G'_i. Again we obtain an equivalent sequence of graphs with only $i-1$ edge contractions preceding a deletion. We repeat this as many times as required until all deletions precede all contractions.

It follows that when constructing a minor H of graph G, we can start with a subgraph of G and apply a sequence of edge-contractions only to obtain H. Often this is used as the definition of graph minor.

Consider the situation when a vertex u of degree three in G is split into $u1u2$. If u is adjacent to vertices x, y, z in G, then in G^+_v, $u1$ is adjacent to at least two of x, y, z, and via $u2$ there is always a path from $u1$ to the third vertex. This is used in the following theorem, and also in Exercise 12.3.5.

THEOREM 12.5 If G has a minor K3, 3, then G contains a subgraph TK3,3. If G has a minor K5, then G contains a subgraph TK5 or TK3, 3.

PROOF Suppose that G has a minor $K5$ or $K3, 3$. If no edges were contracted to obtain this minor, then it is also a subgraph of G. Otherwise let $G0, G1,\ldots, Gk$ be a sequence of graphs obtained from G, where $G0$ is a subgraph of G, edge ei of $Gi-1$ is contracted to obtain Gi, and Gk is either $K5$ or $K3, 3$.

Page 281

If each ei has an endpoint of degree two, then we can reverse the contractions by subdividing edges, resulting in a $TK5$ or $TK3, 3$ in G, as required. Otherwise let ei be the edge with largest i, with both endpoints of at least degree three. All edges contracted subsequent to Gi have an endpoint of degree two, so that Gi has a subgraph $TK5$ or $TK3, 3$. $Gi-1$ can be obtained by splitting a vertex u of Gi. If u is a vertex of $TK5$, then by Lemma 12.3, $Gi-1$ contains $TK3, 3$. If u is a vertex of $TK3, 3$, then $Gi-1$ also contains $TK3, 3$. If u is a vertex of neither $TK5$ nor $TK3, 3$, then $Gi-1$ still contains $TK5$ or $TK3, 3$. In each case we find that $G0$ must have a subgraph $TK5$ or $TK3, 3$.

DEFINITION 12.7: Given a subgraph TK of G, equal to $TK5$ or $TK3,3$. The vertices of TK which correspond to vertices of $K5$ or $K3, 3$ are called the *corners* of TK. The other vertices of TK are called *inner verftices* of TK.

Suppose that G is a non-planar graph with a subgraph $TK5$. Let $u1, u2,\ldots, u5$ be the corners of $TK5$. Each ui has degree four in $TK5$; the inner vertices of $TK5$ have degree two. Let Pij be the path in $TK5$ connecting ui to uj. Consider the situation where G contains a path P from $x \in P_{ij}$ to $y \in P_{k\ell}$, where x and y are inner vertices. This is illustrated in Figure 12.5, where a vertex $x \in P_{45}$ is connected by a path to $y \in P_{23}$. We see that in this case, G contains a $TK3,3$.

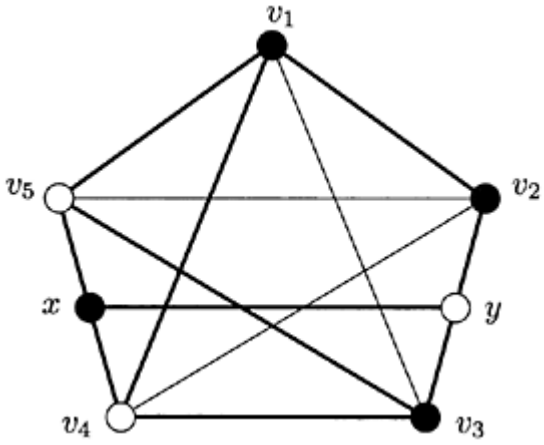


FIGURE 12.5
*TK***5and** *TK***3, 3**

*THEOREM 12.6 Let G contain a subgraph TK*5, *with corners u*1, *u*2,..., *u*5*connected by paths Pij. If G has vertices x and y such that x is an inner vertex of Pij, and* $y \in P_{k\ell}$ *but* $y \notin P_{ij}$, *where Pij≠Pkℓ, then G contains a TK*3,3.

Page 282

PROOF One case of the proof is illustrated in Figure 12.5. The remaining cases are done in Exercise 12.3.2. A consequence of Theorem 12.6 is that nearly any graph that has a subgraph *TK*5 also has a subgraph *TK*3, 3. This theorem will be useful in embedding algorithms for non-planar graphs in Chapter 13. It also permits a recursive characterization of graphs which contain *TK*5 but not *TK*3, 3.

Exercises

12.3.1 Show that *K*3, 3 is non-planar, using the Jordan curve theorem.
12.3.2 Complete the proof of Theorem 12.6.
12.3.3 Characterize the class of 2-connected graphs which contain *TK*5 but not *TK*3,3.
12.3.4 Construct a *O(ε)* algorithm which accepts as input a graph *G* and a subgraph *TK*5 with corners *u*1, *u*2, ..., *u*5, and finds a *TK*3, 3 containing *u*1, *u*2,..., *u*5 if one exists.
12.3.5 Let *K* be a graph such that $\Delta(K) \leq 3$. Show that *K* is a minor of *G* if and only if *G* has a subgraph *TK*.

12.4 Euler's formula

Let *G* be a connected planar graph with *n* vertices and *ε* edges, embedded in the plane by a mapping *ψ*. If we remove all the points of the image *Gψ* from the plane, the plane falls apart into several connected regions. This is equivalent to *cutting* the plane along the edges of *Gψ*. For example, if we cut the plane along the edges of the planar embedding of the cube in Figure 12.1, there are six regions, one of which is unbounded.

DEFINITION 12.8: The *faces* of an embedding *Gψ* are the connected regions that remain when the plane is cut along the edges of *Gψ*. The unbounded region is called the *outer face*.

Notice that if *uu* is an edge of *G* contained in a cycle *C,* then *(uu)ψ* is a portion of a Jordan curve *Cψ*. The face on one side of *(uu)ψ* is in INT*(Cψ)* and the face on the other side is in EXT*(Cψ)*. These faces are therefore distinct. But if *uu* is an edge not contained in any cycle, then it is a cut-edge, and the same face appears on each side of *(uu)ψ*. For example, in a tree, every edge is a cut-edge, and there is only one face, the outer face.

Page 283

We view the plane as an oriented surface, which can be viewed from "above" or "below". Given an embedding *Gψ* in the plane, we will view it consistently from one side, which we can assume to be "above" the plane. If we then view an embedding *Gψ* from "below" the surface, it will appear to have been reversed. Therefore we choose one orientation ("above") for all embeddings.

The boundary of a face *F* of an embedding *Gψ* is a closed curve in the plane. It is the image under *ψ* of a closed walk *C* in *G*. We can walk along *Cψ* so that the interior of *Cψ* is to our right-hand side. We will call this a *clockwise* direction and thereby assign an *orientation* to *C*. *We shall always choose a clockwise orientation for traversing the boundaries of faces,* so that the face *F* will be to our right-hand side.

DEFINITION 12.9: An oriented closed walk *C* in *G* bounding a face of *Gψ* is called a *facial walk* of *Gψ* (or *facial cycle* if *C* is a cycle).

Notice that if *C* contains a cut-edge *uu,* then *uu* will appear twice on *C*. The two occurrences of *uu* on *C* will have opposite orientations. All other edges appear at most once on *C*. As we will mostly be interested in 2-

connected graphs *G,* facial walks will almost always be cycles.
DEFINITION 12.10: The *degree* of a face *F* is DEG*(F),* the length of its facial walk.
Notice that a cut-edge appearing on the facial walk of *F* will contribute two to its degree.
THEOREM 12.7 **(Euler's formula)** *Let G be a connected planar graph with n vertices and ε edges. Let Gψ have f faces, where ψ is a planar embedding of G. Then*

$$n+f-\varepsilon=2$$

PROOF The proof is by induction on $\varepsilon-n$. Every connected graph has a spanning tree. If $\varepsilon-n=-1$, then *G* is a tree. It is clear that every tree has a planar embedding. Since a tree has no cycles, there is only one face, the outer face, so $f=1$. Euler's formula is then seen to hold for all embeddings of *G*.
Now suppose that $\varepsilon-n=k\geq 0$. Choose any cycle *C* in *G,* and any edge $uv \in C$. Let the faces on the two sides of $(uv)\psi$ be $F1$ and $F2$. Consider $G'=G-uv,$ with n', ε' and f' vertices, edges, and faces, respectively. Clearly $n'=n$ and $\varepsilon'=\varepsilon-1$. G' is connected and ψ is a planar embedding of it. One of the faces of $G'\psi$ is $F_1 \cup F_2$. The other faces of $G'\psi$ are those of $G\psi$. Therefore $f'=f-1$ Euler's formula follows by induction.
It follows from Euler's formula that all planar embeddings of a connected graph *G* have the same number of faces. Hence we will refer to *f(G)* as the number of

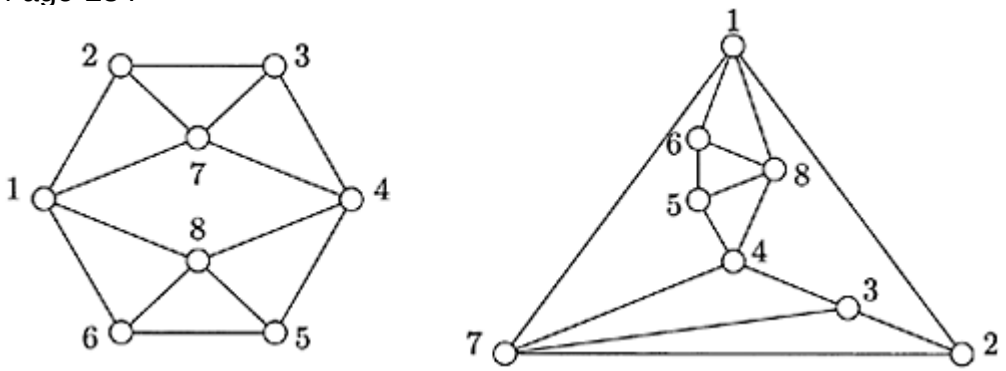


FIGURE 12.6
Two embeddings of a graph

faces of *G,* without specifying an embedding. In Figure 12.6 there is an example of a graph with two distinct embeddings. The embeddings have the same number of faces, but the actual faces and their boundaries are different.

12.5 Rotation systems

Once we have an embedding $G\psi$, we can choose any vertex $v \in V(G)$, and walk around $v\psi$ in a small, clockwise circle. We encounter the incident edges in a certain cyclic order. For example, in the embedding on the left of Figure 12.6, the edges incident on vertex 1 have the clockwise cyclic order (12, 17, 18, 16). Those incident on vertex 2 have the order (23, 27, 21). Those incident vertex 3 have the order (34, 37, 32), etc.
In Figure 12.6, the edges are drawn as straight lines. It is conceivable that the embedding ψ could assign wildly behaved functions, like $\sin(1/x)$ to the curves representing the edges of *G*. Each edge may then be encountered many times when walking around $v\psi$ in a small circle, no matter how small the circle is chosen. We will assume that this does not occur, and that ψ assigns well behaved functions (like straight lines or gradual curves) to the edges. In fact, for graphs embedded on the plane, we shall see that it is always possible to draw the edges as straight lines. For a more complete treatment, the reader is referred to the books of GROSS and TUCKER [54] or MOHAR and THOMASSEN [88].
DEFINITION 12.11: Let $G\psi$ be an embedding in the plane of a loopless connected graph *G*. A *rotation system p* for *G* is a mapping from *V(G)* to the set of permutations of *E(G),* such that for each $v \in V(G)$, *p(u)* is the cyclic per-

mutation of edges incident on *u,* obtained by walking around $v\psi$ in a clockwise direction.
Notice that if *G* has a loop *uu,* then as we walk around $v\psi$, we will cross the loop twice. Therefore *p(u)* will contain *uu* twice. In order to extend the definition to be correct for graphs with loops, we must ensure that for each loop *uu,* that *p(u)* contains two corresponding "loops" $(uu)1$ and $(uu)2$.
Suppose we are given the rotation system *p* determined by an embedding $G\psi$. We can then easily find the facial cycles of $G\psi$. The following fundamental algorithm shows how to do this. Let $u \in V(G)$, and let *e* be

any edge incident on *u*. We are assuming that given an edge *e′=uu* in *p(u),* we can find the corresponding edge *e″=uu* in *p(u)*. A data structure of linked lists can do this easily in constant time. If *G* is a simple graph, then the rotation system is completely specified by the cyclic adjacency lists. Given a planar embedding *Gψ,* we will assume that the adjacency lists are always given in cyclic order, so that the rotation system *p* corresponding to *ψ* is available.

Algorithm 12.5.1: FACIALCYCLE*(Gψ, u, e)*

comment: Given an embedding G^{ψ} with corresponding rotation system p and vertex u with incident edge e, find the facial cycle containing e.

$e' \leftarrow e$

repeat

comment: e' currently equals uv, for some v

$v \leftarrow$ other end of e'

$e'' \leftarrow$ edge of $p(v)$ corresponding to e'

comment: e'' currently equals vu

$e' \leftarrow$ edge preceding e'' in $p(v)$

$u \leftarrow v$

until *e′=e*

LEMMA 12.8 The sequence of edges traversed by FACIALCYCLE*(Gψ, u, e) forms the facial cycle of the face to the right of eψ.*

PROOF Let *F* be the face to the right of *eψ*. Let *e=uu*. As we walk along *eψ* from *uψ* to *uψ,* the face *F* is to our right-hand side. When we reach *uψ,* we are on the image of an edge *uu* in *p(u)*. Since *p(u)* has a clockwise cyclic order, the next edge in the facial cycle is the one preceding *uu* in *p(u)*. This is the one chosen by the algorithm. The algorithm repeats this process until it arrives back at the starting edge *e*.

Algorithm FACIALCYCLE() is very simple, but it is tremendously important. It is used in nearly all algorithms dealing with graph embeddings. Notice that its running time is *O(ε)* and that it can be used to find all the facial cycles of *Gψ* in *O(ε)* time.

COROLLARY 12.9 The facial cycles of an embedding Gψ are completely determined by its rotation system.

PROOF All facial cycles can be determined by executing Algorithm FACIALCYCLE*(Gψ, u, e),* such that each edge *e* is visited at most twice, once for the face on each side of *e*. The rotation system is the only information about ψ that is needed.

Thus, it turns out that planar embeddings are essentially combinatorial, as the facial cycles of an embedding are completely determined by the set of cyclic permutations of incident edges of the vertices. Later we will see that for 3-connected planar graphs, the rotation system is unique, up to orientation. If *p* is the rotation system corresponding to an embedding *ψ,* we will often write *Gp* instead of *Gψ*. We call *Gψ* a *topological embedding,* and *Gp* a *combinatorial embedding*. The combinatorial embedding determines the facial cycles of the embedding, but it does not give an actual drawing of *G* in the plane.

12.6 Dual graphs

Consider an embedding *Gψ,* illustrated by the cube in Figure 12.7. Let its faces be listed as {*F*1, *F*2,..., *Ff*}. Two faces *Fi* and *Fj* are adjacent if they share a common edge *(uυ)ψ* on their boundaries. We can construct a planar graph *Gψ** by placing a new vertex *fi* in each region *Fi,* for *i*=1, 2,..., *f*. Whenever two faces *Fi* and *Fj* share an edge *(uυ)ψ* on their boundaries, we draw a continuous curve from *fi* to *fj,* passing through *(uυ)ψ* in exactly one interior point, and intersecting *Gψ* in only this point. This is illustrated for an embedding of the cube in Figure 12.7. We call *Gψ* a planar dual of Gψ*.

LEMMA 12.10 Let Gψ be a planar embedding with a planar dual Gψ. Let Gψ**be any planar dual of Gψ*.* Then $G^{\psi**} \cong G^{\psi}$.

PROOF Let the faces of Gψ be *F*1, *F*2,..., *Ff*, and let *fi* be the vertex of *Gψ** corresponding to *Fi*. Consider any vertex *u* of *G* and its cyclic permutation *p(u)*=(*uυ*1, *uυ*2,..., *uυk*) of incident edges. Each edge *(uυℓ)ψ* separates two

faces *Fi* and *Fj,* and so is crossed by a curve connecting *fi* to *fj*. As we successively take the edges *uυℓ* of *p(u),* we traverse these curves, thereby constructing a facial boundary of *Gψ**. Vertex *uψ* is contained in the

region interior to this facial boundary. We conclude that each face of $G\psi^*$ contains exactly one $u\psi$, and that the edges $(uv\ell)\psi$ are curves connecting the vertices $u\psi$ and v_ℓ^ψ located inside the faces of $G\psi^*$. That is, the planar dual construction applied to $G\psi^*$ gives back $G\psi$. Equivalently, $G^{\psi^{**}} \cong G^\psi$.

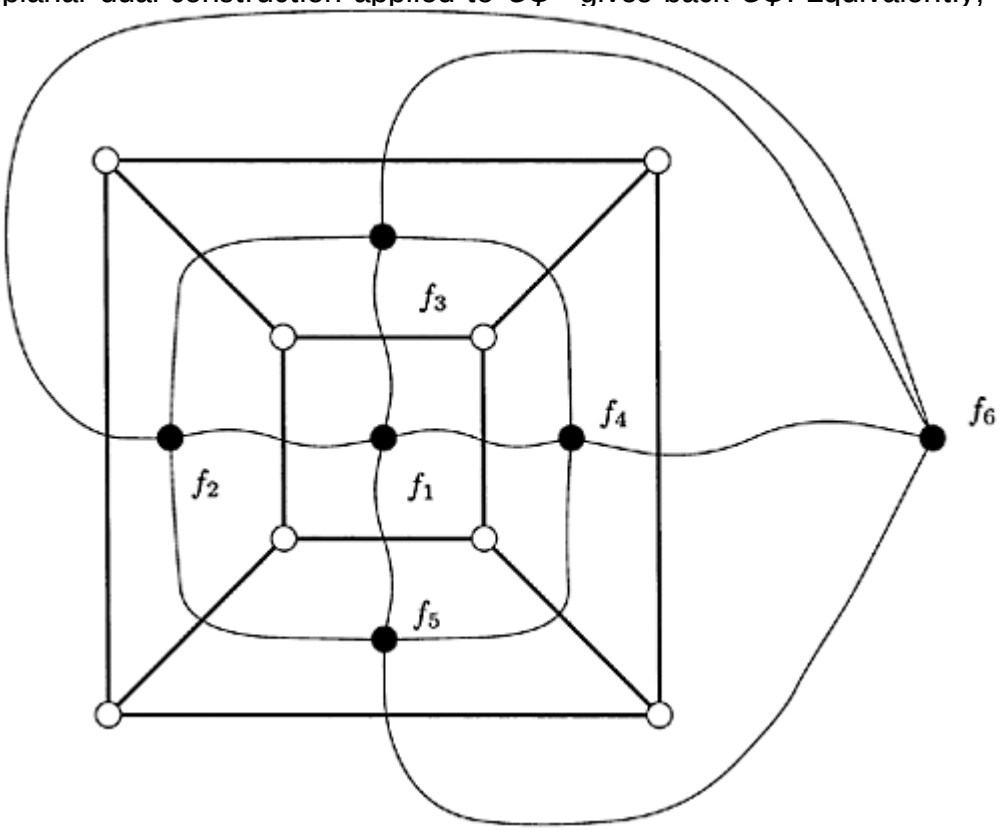


FIGURE 12.7
Constructing a dual graph

Now the adjacencies of the planar dual are determined completely by common edges of the facial cycles of $G\psi$, and these are determined by the rotation sy stem p of $G\psi$. Therefore we define the combinatorial planar dual in terms of the rotation system.

Page 288

DEFINITION 12.12: Let p be the rotation system for G corresponding to a planar embedding ψ. Let the facial cycles of Gp be $F=\{F1, F2,..., Ff\}$. The *combinatorial planar dual* of Gp is denoted Gp^*. The vertex set of Gp^* is $\{F1, F2,..., Ff\}$. The edges of Gp^* are defined by a rotation system, also denoted p, and given as follows. Consider a facial cycle

$$Fi=(u1, u2,..., uk),$$

traversed in a clockwise direction. Each edge $u\ell u\ell+1$ is contained in exactly two facial cycles, which are adjacent in Gp^*. As we walk along the facial cycle, the face corresponding to Fi appears on the right-hand side of $(u\ell u\ell+1)\psi$. On the left-hand side is the face corresponding to $F\ell'$, where $F\ell'$ is the unique facial cycle containing edge $u\ell+1u\ell$. We then take

$$p(F_i) = (F_{1'}, F_{2'}, \ldots, F_{k'}).$$

It is easy to see that $FiFj$ occurs in $p(Fi)$ if and only if $FjFi$ occurs in $p(Fj)$. Thus the definition is valid. If Fi contains a cut-edge uu, then the same face occurs on both sides of $(uu)\psi$. Gp^* will then contain a loop $FiFi$. Since uu appears twice on the facial cycle, $FiFi$ will occur twice in $p(Fi)$.

The graph Gp^* constructed by this definition is always isomorphic to the planar dual $G\psi^*$ constructed above, because of the correspondence between faces and facial cycles. Therefore the rotation system constructed for Gp^* always corresponds to a planar embedding $G\psi^*$. It follows from the above lemma that $G^{p^{**}} \cong G^p$.

Now let Gp denote a combinatorial planar embedding of G, and let $\{F1, F2,..., Ff\}$ be the facial cycles of Gp. The degree of Fi is $DEG(Fi)$, the length of the walk. Let Gp^* be the dual of Gp, and write $n^*, \varepsilon^*,$ and f^* for the numbers of vertices, edges, and faces, respectively, of Gp^*.

$$\sum_{i=1}^{f} \text{DEG}(F_i) = 2\varepsilon(G).$$

LEMMA 12.11

PROOF Each edge *uu* of *G* is incident on two faces of *Gp*.

LEMMA 12.12 n=f, f*=n, and ε*=ε.*

PROOF *n*=f* follows from the definition of *Gp**. Since $G^{p**} \cong G^{p}$, we have *f*=n*. Each edge of *Gp** corresponds to exactly one edge of *Gp,* and every edge of *Gp* corresponds to exactly one edge of *Gp**. Therefore *ε*=ε*.

Page 289

Algorithm 12.6.1: CONSTRUCTDUAL*(Gp)*

comment: Given a graph G with a planar rotation system p, construct the combinatorial dual G^{p*}.

nFaces←0

for all edges *uu*

do $FaceNumber\langle uv\rangle \leftarrow 0$

for all vertices *u*

do **for each** uv in $p(u)$

do **if** $FaceNumber\langle uv\rangle = 0$

then $nFaces \leftarrow nFaces + 1$
traverse the facial cycle F to the right of uv
using FACIALCYCLE(G^p, u, uv) and store
nFaces in the *FaceNumber* of each edge of F

comment: we have numbered all the faces, now construct the edges of the dual

for all vertices *u*

do **for each** uv in $p(u)$

do $i \leftarrow FaceNumber\langle uv\rangle$
if face i has not been traversed yet

then traverse the facial cycle F to the right of uv
using FACIALCYCLE(G^p, u, uv),
and for each edge xy of F
let $j = FaceNumber\langle yx\rangle$ in $p(y)$
append ij to $p(i)$ in G^{p*}

Algorithm 12.6.1 is a simple algorithm to construct the dual in *O(ε)* time. We assume that the faces of *Gp* are numbered 1, 2,..., *f;* that the rotation system *p* is represented as cyclic linked lists; and that the linked list node corresponding to *uu* in *p(u)* contains a field called the *FaceNumber,* used to indicate which face of *Gp* is on the right-hand side of *uu* as it is traversed from *u* to *u*. We will denote this by $FaceNumber\langle uv\rangle$, although it is not stored as an array. We will also use a variable *nFaces* to count the faces of *Gp*.

Algorithm 12.6.1 uses FACIALCYCLE() (Algorithm 12.5.1) to walk around the facial cycles of *Gp* and number them. Notice that FACIALCYCLE() only requires the rotation system *p* rather than the topological embedding *ψ*. Each edge of *G* is traversed exactly twice, taking a total of *O(ε)* steps. It then traverses the facial cycles again, constructing the rotation system of *Gp*,* using the face numbers which were previously stored. This again takes *O(ε)* steps. Thus, the dual graph is completely determined by the combinatorial embedding *Gp*.

Page 290

12.7 Platonic solids, polyhedra

The cube is a 3-regular planar graph whose faces all have degree four. So its dual is 4-regular. It can be seen from Figure 12.7 that the dual of the cube is the octahedron. Let *G* be a connected *k*-regular planar graph whose dual is *ℓ*-regular, where *k, ℓ*≥3. These graphs are called graphs of the *Platonic solids*. Then *kn*=2*ε* and *ℓf*=2*ε*. Substituting this into Euler's formula and dividing by *ε* gives

$$\frac{1}{k}+\frac{1}{\ell}=\frac{1}{2}+\frac{2}{\varepsilon}$$

If we consider graphs with $\varepsilon \geq 4$ edges, we have

$$\frac{1}{2}<\frac{1}{k}+\frac{1}{\ell}\leq 1$$

As the number of integers satisfying this inequality is limited, this can be used to find all such graphs. They are the graphs of the regular polyhedra—the tetrahedron, cube, octahedron, dodecahedron, and icosahedron.
In general, a *polyhedron* is a geometric solid whose faces are *polygons,* that is, regions of a plane bounded by a finite sequence of line segments. Each edge of a polyhedron is common to exactly two polygonal faces. So if we consider an edge *uu*1 incident on a vertex *u,* there are two polygons, *P*1 and *P*2 incident on *uu*1. Now *P*2 has two edges incident on *u*. Let the other be *uu*2. But this edge is also incident on two polygons, *P*2 and *P*3. Since *P*3 has two edges incident on *u,* we obtain another edge *uu*3, etc. Continuing in this way, we get a sequence *u*1, *u*2,... of vertices adjacent to *u,* until we return to *P*1.
DEFINITION 12.13: A *polyhedron* is a connected collection of polygons such that
1. Each edge is contained in exactly two polygons.
2. Polygons do not intersect, except on a common edge.
3. Any two polygons intersect in at most one edge.
4. The polygons incident on a vertex form a single cycle.
The fourth condition is to prevent a polyhedron created by identifying two vertices of otherwise disjoint polyhedra.
DEFINITION 12.14: The *skeleton* of a polyhedron is the graph whose vertices are the vertices of the polyhedron, such that vertices *u* and *u* are adjacent in the graph if and only if *uu* is an edge of the polyhedron.
DEFINITION 12.15: A *regular polygon,* denoted *{p},* is a planar polygon with *p* sides all of equal length. A *regular polyhedron,* denoted *{p, q},* is a poly-

hedron whose faces are polygons *{p},* such that exactly *q* polygons are incident on each vertex.
The symbols *{p}* and *{p, q}* are called the *Schläfli symbols* for the polygon and polyhedron, respectively. Notice that given a vertex *u* of a regular polyhedron *{p, q},* the midpoints of the edges incident on *u* form a regular polygon *{q}*.
A polyhedron is *convex* if its interior is a convex region; that is, given any two points *P* and *Q* in the interior, the line segment connecting *P* to *Q* is completely contained inside the polyhedron. There is a remarkable theorem of Steinitz characterizing the skeletons of convex polyhedra.
THEOREM 12.13 **(Steinitz's theorem)** *A graph G is the skeleton of a convex polyhedron if and only if G is planar and 3-connected.*
A proof of Steinitz's theorem can be found in the book by GRÜNBAUM [56] or ZIEGLER [129]. It is too lengthy to include here.

Exercises

12.7.1 Find the planar dual of the line graph of *K*4. Find the line graph of the cube, and find its planar dual.
12.7.2 Find all k-regular planar graphs whose duals are ℓ-regular, for all possible values of *k* and ℓ.
12.7.3 Find the dual of the multigraph constructed from *K*4 by doubling each edge. Refer to Figure 12.8.

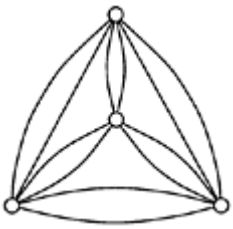


FIGURE 12.8
Find the dual graph

12.7.4 A planar graph *G* is *self-dual* if it is isomorphic to its planar dual. Find a self-dual planar graph on *n* vertices, for all $n \geq 4$.
12.7.5 Program the algorithm CONSTRUCTDUAL(), and test it on the graphs ofExer cises 12.7.1 and 12.7.2.
12.7.6 Show that a planar graph is bipartite if and only if its dual is Eulerian.

12.7.7 Find the Schläfli symbols of the tetrahedron, cube, octahedron, dodecahedron, and icosahedron.
12.7.8 Given a Schläfli symbol *{p, q}* for a polyhedron, find a formula for the number of vertices, edges, and faces of the polyhedron in terms of *p* and *q*.

12.8 Triangulations

A planar embedding *Gp* whose faces all have degree three is called a *triangulation*. If *Gp* is a triangulation, then $3f=2\varepsilon$. Substituting into Euler's formula gives:

LEMMA 12.14 A triangulation Gpsatisfies $\varepsilon=3n-6$ *and* $f=2n-4$.

If *Gp* is not a triangulation, and has no multiple edges or loops, then every face has at least degree three. We can convert *Gp* to a triangulation by adding some diagonal edges to faces of degree four or more. This gives the following:

LEMMA 12.15 A simple planar graph G has $\varepsilon\leq 3n-6$.

For example, since $K5$ has $\varepsilon=10>3n-6$, we can conclude that $K5$ is non-planar.

One consequence of the above lemma is that *O(ε)* algorithms on planar graphs are also *O(n)* algorithms. For example, a DFS or BFS in a planar graph takes *O(n)* steps. This will be useful in algorithms for testing planarity, or for drawing or coloring a planar graph, or constructing dual graphs.

Given a graph *G,* we can subdivide any edges of *G* without affecting the planarity of *G*. Therefore, we will assume that *G* has no vertices of degree two. We will also take *G* to be 2-edge-connected, so that there are no vertices of degree one. Let *G* be a simple planar graph, and let *ni* be the number of vertices of degree *i,* for $i=3, 4,\ldots$. Counting the edges of *G* gives

$$3n3+4n4+5n5+\ldots=2\varepsilon<6n-12$$

Counting the vertices of *G* gives

$$n3+n4+n5+\ldots=n$$

Multiply the second equation by 6 and subtract the two equations to obtain:

LEMMA 12.16 A simple planar graph with no vertices of degree one or two satisfies

$$3n3+2n4+n5\geq 12+n7+2n8+3n9+\ldots$$

COROLLARY 12.17 A simple planar graph with no vertices of degree one or two has a vertex of degree three, four, or five.

PROOF The values *ni* are non-negative integers.

The above corollary results in a technique for reducing a planar triangulation on *n* vertices to one on *n*−1 vertices that is fundamental for understanding the structure of planar graphs, and for handling them algorithmically.

Algorithm 12.8.1: REDUCEGRAPH*(Gp)*

comment: Given a simple planar triangulation G on $n > 4$ vertices with rotation system p. Construct a planar triangulation G' on $n-1$ vertices.

if there is a vertex *u* with DEG*(u)*=3

then let $p(u) = (ux, uy, uz)$; $G' \leftarrow G - u$; **return** (G')

if there is a vertex *u* with DEG*(u)*=4

then let $p(u) = (uw, ux, uy, uz)$

if $w \not\longrightarrow y$

then $G' \leftarrow G - u + wy$; wy replaces wu in $p(w)$ and yw replaces yu in $p(y)$

else $G' \leftarrow G - u + xz$; xz replaces xu in $p(x)$ and zx replaces zu in $p(z)$

return (G')

comment: otherwise, DEG*(u)*=5

let *p(u)=(uu,uw,ux,uy,uz)*

if $v \not\longrightarrow x$ and $v \not\longrightarrow y$

$$\textbf{then}\ \begin{cases} G' \leftarrow G - u + vx + vy \\ vx, vy \text{ replace } vu \text{ in } p(v) \\ xv \text{ replaces } xu \text{ in } p(x) \text{ and } yv \text{ replaces } yu \text{ in } p(y) \end{cases}$$

else if $v \longrightarrow x$

$$\textbf{then}\ \begin{cases} G' \leftarrow G - u + wy + wz \\ wy, wz \text{ replace } yu \text{ in } p(w) \\ yw \text{ replaces } yu \text{ in } p(y) \text{ and } zw \text{ replaces } zu \text{ in } p(z) \end{cases}$$

else if $v \longrightarrow y$

$$\textbf{then}\ \begin{cases} G' \leftarrow G - u + zw + zx \\ zw, zx \text{ replace } zu \text{ in } p(z) \\ wz \text{ replaces } wu \text{ in } p(w) \text{ and } zw \text{ replaces } zu \text{ in } p(z) \end{cases}$$

return *(G′)*

Page 294

Here *Gn* is a triangulation on *n*≥4 vertices. If *n*=4, then *K*4 is the only possibility. So we assume that *n*>4. We know that *Gn* always has a vertex of degree three, four, or five.

THEOREM 12.18 Given a simple planar triangulation Gnon n>4 *vertices, Algorithm* REDUCEGRAPH() *constructs a simple planar triangulation Gn*−1 *on n*−1 *vertices*.

PROOF We know that *Gn* has a vertex of degree three, four, or five. If there is a vertex *u* of degree three, let *p(u)=(ux, uy, uz),* as illustrated in Figure 12.9. Since *Gn* is a triangulation, we know that *xy, yz,* and *zx* are edges of *Gn*. Consequently *Gn−u* is also a triangulation.

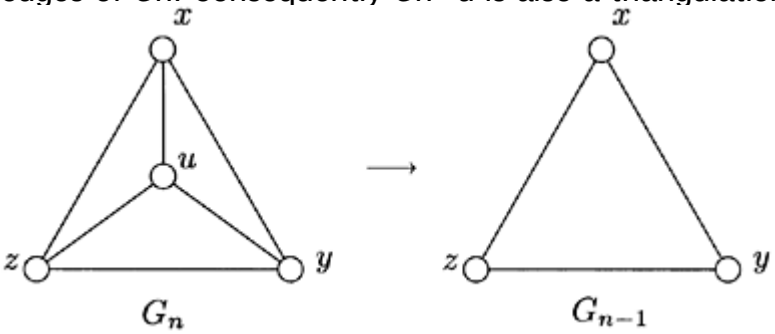


FIGURE 12.9

DEG*(u)*=3

If there is a vertex *u* of degree four, let *p(u)=(uw, ux, uy, uz),* as illustrated in Figure 12.10. Since *Gn* is a triangulation, we know that *wx, xy, yz,* and *zw* are edges of *Gn*. When *u* is deleted, one face of *Gn−u* is a quadrilateral. If $w \not\longrightarrow y$, we can add the edge wy to get a planar triangulation *Gn−u+wy*. If $w \longrightarrow y$, then edge *wy* is exterior to the quadrilateral face. Consequently $x \not\longrightarrow z$, so that we can add the edge *xz* to get a planar triangulation *Gn−u+xz*. In either case we get a planar triangulation *Gn*−1.

If there is a vertex *u* of degree five, let *p(u)=(uu, uw, ux, uy, uz),* as illustrated in Figure 12.11. Since *Gn* is a triangulation, we know that *uw, wx, xy, yz,* and *zu* are edges of *Gn*. When *u* is deleted, one face of *Gn−u* is a pentagon. If $v \not\longrightarrow x$ and $v \not\longrightarrow y$, we can add the edges *ux* and *uy* to get a planar triangulation *Gn−u+ux+uy*. Otherwise, if $v \longrightarrow x$, then edge *ux* is exterior to the pentagonal face. Consequently, $w \not\longrightarrow y$ and $w \not\longrightarrow z$, so that we can add the edges *wy* and *wz* to get a planar triangulation *Gn−u+wy+wz*. Otherwise, if $v \longrightarrow y$, then edge *uy* is exterior to the pentagonal face, and we can proceed as above to get a triangulation *Gn−u+zw+zx*. The proof is complete.

Page 295

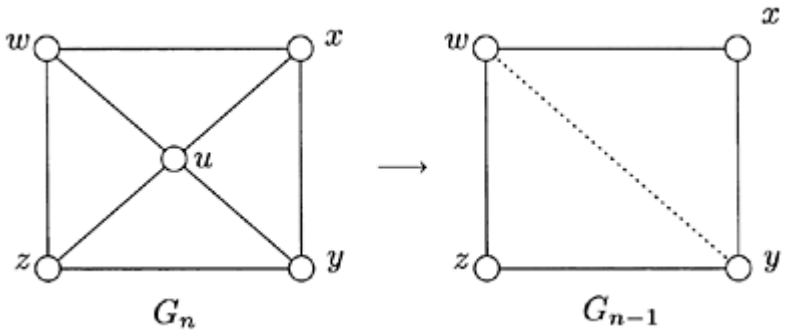


FIGURE 12.10
DEG*(u)*=**4**

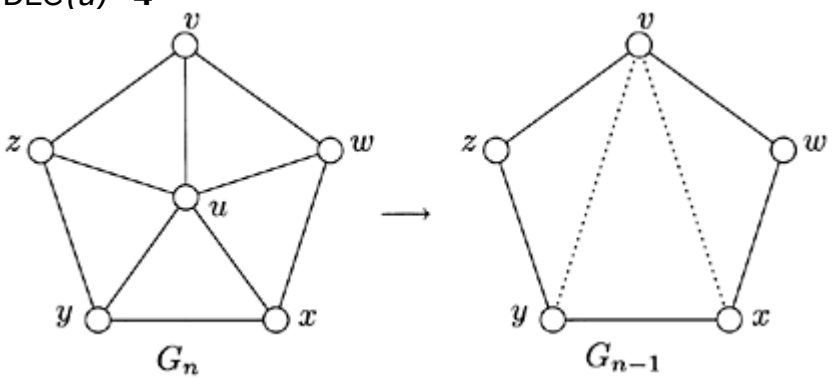


FIGURE 12.11
DEG*(u)*=**5**

Note that Algorithm REDUCEGRAPH() requires the degrees of the vertices. These can be computed in *O(n)* time and stored in an array. Once the degrees are known, the reduction from *Gn* to *Gn*−1 takes constant time. Usually, this algorithm will be applied recursively to reduce a planar triangulation *Gn* to *G*4, which must equal *K*4. If the algorithm is being used to find a planar embedding of *Gn,* or to color it, the graph will then be rebuilt in reverse order.

12.9 The sphere

The plane can be mapped onto the surface of the sphere by a simple transformation called *stereographic projection*. Place a sphere on the surface of the plane, so that it is tangent at the south pole. See Figure 12.12. Now from any point *P* on the plane, construct a straight-line *L* to the north pole of the sphere. *L* will intersect the surface of the sphere in some point. Call it *P′*. This transformation maps any

point *P* in the plane to a point *P′* on the surface of the sphere. The mapping is clearly invertible and continuous. The only point on the sphere to which no point of the plane is mapped is the north pole.

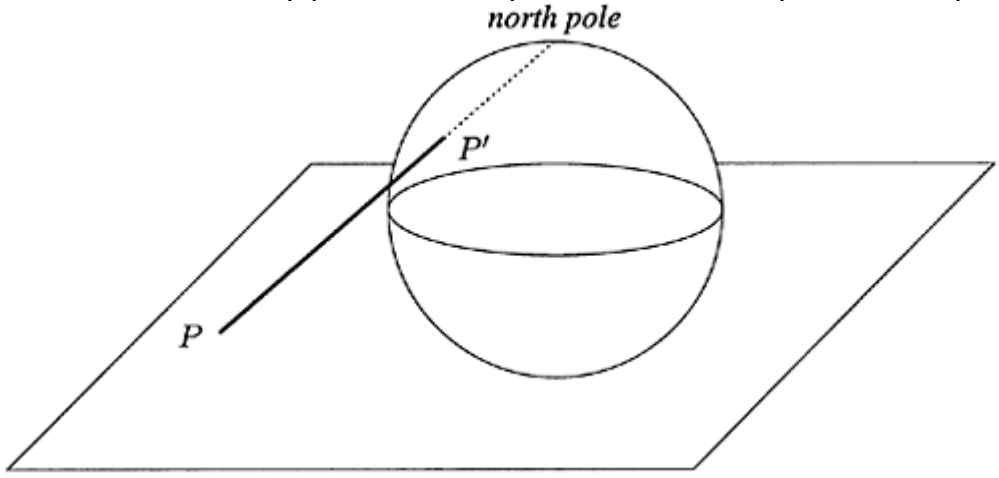


FIGURE 12.12
Mapping the plane to the sphere

If *Gψ* is an embedding of a graph on the plane, then stereographic projection will map the points of *Gψ* onto an embedding of *G* on the surface of the sphere. Conversely, if we are given an embedding of *G* on the

surface of the sphere, we can roll the sphere to ensure that the north pole is not a point of the embedding. Then use stereographic projection to map the surface of the sphere onto the plane, thereby obtaining an embedding of *G* on the plane. Consequently, embedding graphs on the plane is equivalent to embedding them on the sphere.

When a graph is embedded on the surface of the sphere, the faces are the regions that remain when the sphere is cut along the edges of *G*. There is no outer face. Every face is bounded. However, the face that contains the north pole will become the outer face when the embedding is projected onto the plane. By rolling the sphere to place any desired face at the top, we can make any face the outer face. We state this as a lemma.

LEMMA 12.19 A planar graph can be drawn so that any facial cycle, any edge, or any vertex appears on the boundary of the outer face.

Page 297

12.10 Whitney's theorem

The plane and sphere are oriented surfaces. Consider a graph *G* embedded on the plane as *Gp,* where *p* gives the clockwise orientation of the edges incident on each vertex. We are assuming that the plane is viewed from above. If we now view the plane from below, the clockwise orientation of each *p(u)* will appear counter clockwise. If an embedding *Gp* is projected onto the sphere, then each *p(u)* will appear clockwise if viewed from inside the sphere, but counter clockwise if viewed from outside the sphere. Given a rotation system *p,* we write $\overline{p}$ for the rotation system obtained by reversing the cyclic order of each *p(u)*. So $\overline{p}(u) = p(u)^{-1}$. The embeddings *Gp* and $G^{\overline{p}}$ are usually considered equivalent.

DEFINITION 12.16: Let G^{p_1} and G^{p_2} be two embeddings of a graph *G,* with rotation systems *p*1 and *p*2, respectively. G^{p_1} and G^{p_2} are *isomorphic embeddings* if there is an automorphism of *G* which transforms *p*1 into *p*2. G^{p_1} and G^{p_2} are *equivalent embeddings* if there is an automorphism of *G* which transforms *p*1 into either *p*2 or $\overline{p}_2$.

An automorphism of *G* will permute the vertices of *G,* and consequently alter the edges in the cyclic permutations of a rotation system. If *θ* is an automorphism of *G,* then *p*1*(u)*=(*e*1, *e*2,..., *ek)* is transformed by *θ* into *θ*(*p*1*(u)*)= (*θ*(*e*1), *θ*(*e*2),..., *θ(ek)*). If this equals *p2(θ(u))*, for all vertices *u,* then G^{p_1} and G^{p_2} are isomorphic. Isomorphic rotation systems are equivalent.

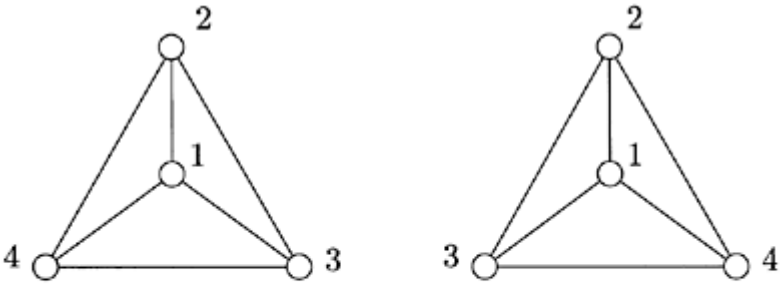
Two isomorphic rotation systems for *K*4 are illustrated in Figure 12.13. Here $p_2 = \overline{p_1}$. it is easy to see that if we take *θ*=(3, 4), then *θ*(*p*1*(u)*)=*p*2*(θ(u)),* for all *u*=1, 2, 3, 4.

A triangulation *G* on 7 points is shown in Figure 12.14. Two rotation systems for it, *p*1 and *p*2 are also given below. Here we also have $p_2 = \overline{p_1}$. However, there is no automorphism of *G* that will transform *p*1 into *p*2. This can be verified using the degrees of the vertices. Vertex 2 is the only vertex of degree six. Hence any automorphism *θ* must fix 2. So *θ*(*p*1(2)) must equal *p*2(2). The only vertices of degree four are vertices 1 and 5. Therefore either *θ*(1)=1 or *θ*(1)=5. If *θ*(1)=1, then from *p*2(2) we see that *θ*(3)=6. This is impossible as vertices 3 and 6 have different degrees. If *θ*(1)=5, then from *p*2(2) we have *θ*(3)=4, which is also impossible, as vertices 3 and 4 have different degrees.

So G^{p_1} and G^{p_2} are equivalent embeddings that are non-isomorphic. This can only occur if there is an automorphism of *G* mapping *p*1 to $\overline{p}_2$, but no automorphism mapping *p*1 to *p*2; that is, G^{p_2} is obtained by "flipping" G^{p_1} upside down. The example of Figure 12.13 shows that this is not possible with *K*4, but is possible with the triangulation of Figure 12.14.

DEFINITION 12.17: A planar graph *G* is *orientable* if there exists a planar rotation system *p* such that $G^p \ncong G^{\overline{p}}$. Otherwise *G* is *non-orientable*.

Page 298



$p_1(1) = (12, 13, 14)$ $p_2(1) = (12, 14, 13)$
$p_1(2) = (21, 24, 23)$ $p_2(2) = (21, 23, 24)$
$p_1(3) = (31, 32, 34)$ $p_2(3) = (31, 34, 32)$
$p_1(4) = (41, 43, 42)$ $p_2(4) = (41, 42, 43)$

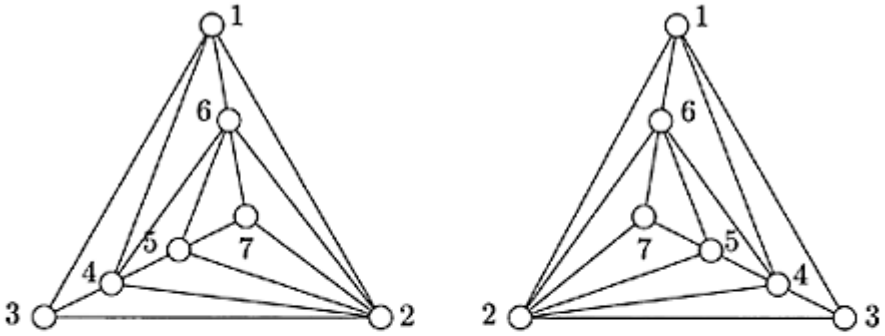
FIGURE 12.13
Two isomorphic rotation systems for *K*4

So *K*4 is non-orientable, but the graph of Figure 12.14 is orientable. An example of a 2-connected graph with two inequivalent planar embeddings is shown in Figure 12.6. Whitney's theorem states that if *G* is 3-connected, this cannot happen. Let *C* be a cycle in a connected graph *G*. *C* is a *separating cycle* if *G−V(C)* is disconnected.

THEOREM 12.20 **(Whitney's theorem)** *Let G be a 3-connected planar graph. Let p be any planar rotation system for G. The facial cycles of Gp are the induced, non-separating cycles of G.*

PROOF Let *C* be an induced, non-separating cycle of *G*. In any planar embedding of *G, C* corresponds to a Jordan curve in the plane. There can be vertices of *G* in the interior or exterior of the Jordan curve, but not both, because *C* is non-separating. Without loss of generality, assume that any vertices of *G−C* are in the exterior of the Jordan curve. It follows that the interior of the Jordan curve is a face, so that *C* is a facial cycle of *G*.

Conversely, let *C* be a facial cycle. Without loss of generality, we can assume that *C* corresponds to a Jordan curve whose interior is a face. If *G* contains an edge *uu* which is a chord of *C,* then *u* and *u* divide *C* into two paths *C[u, u]* and *C[u, u]*. Since *uu* must be embedded exterior to *C,* there can be no path from an is disconnected, a contradiction, as *G* is 3-connected. Consequently, *C* is an interior vertex of *C[u, u]* to an interior vertex of *C[u, u]*. Therefore *G−{u, u}* induced cycle. Let *x* be any vertex of *G−C*. If *x* is the only vertex of *G−C,* then *G−C=x,* so that *C* is a non-separating cycle, and we are done. Otherwise



$p_1(1) = (12, 16, 14, 13)$ $p_2(1) = (12, 13, 14, 16)$
$p_1(2) = (21, 23, 24, 25, 27, 26)$ $p_2(2) = (21, 26, 27, 25, 24, 23)$
$p_1(3) = (31, 34, 32)$ $p_2(3) = (31, 32, 34)$
$p_1(4) = (41, 46, 45, 42, 43)$ $p_2(4) = (41, 43, 42, 45, 46)$
$p_1(5) = (52, 54, 56, 57)$ $p_2(5) = (52, 57, 56, 54)$
$p_1(6) = (61, 62, 67, 65, 64)$ $p_2(6) = (61, 64, 65, 67, 62)$
$p_1(7) = (72, 75, 76)$ $p_2(7) = (72, 76, 75)$

FIGURE 12.14
Two equivalent, non-isomorphic rotation systems
let *y* be another vertex of *G−C*. Since *G* is 3-connected, *G* contains at least three internally disjoint *xy*-paths. At most two of these paths can intersect *C*. See Figure 12.15. Therefore *G−C* contains an *xy*-path, for all *x, y*. It follows that *C* is a non-separating cycle of *G*.
One consequence of Whitney's theorem is that, if *G* is a 3-connected planar graph, an embedding can be found purely from the cycles of *G*. If we can identify an induced, non-separating cycle *C,* we can then assign an orientation to *C*. Each edge of *C* will be contained in another induced, non-separating cycle, so that the orientation of adjacent cycles will thereby also be determined. Continuing in this way, a complete rotation system for *G* can be constructed. This rotation system can then be used to construct a dual graph.
DEFINITION 12.18: Let *G* be a 3-connected planar graph. The *abstract dual* is *G*,* a dual graph determined by the induced non-separating cycles of *G*.
The abstract dual is the same as *Gp*,* where *p* is a rotation system determined by the induced, non-separating cycles, but it can be constructed without reference to a rotation system.

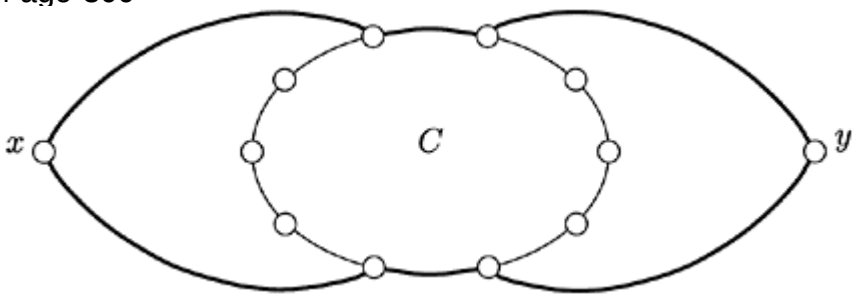


FIGURE 12.15
Whitney's theorem
Up to equivalence of embeddings, a 3-connected planar graph has just one rotation system, so that the abstract dual is uniquely defined. Orientable 3-connected planar graphs have just two rotation systems (which are inverses of each other). Non-orientable 3-connected planar graphs have just one rotation system.
Whitney's theorem does not provide an algorithm for determining whether a graph is planar, as the characterization of planarity in terms of induced, non-separating cycles does not lead to an efficient algorithm. There are too many cycles in a graph to effectively find them all and determine whether they are induced, non-separating cycles.

12.11 Medial digraphs

Let *G* be a loopless graph with a planar rotation system *p*. Note that *G* is allowed to be a multigraph. We construct a digraph representing *Gp*.
DEFINITION 12.19: The *medial digraph M(G)* is obtained from *G* by subdividing every edge *uu* of *G* with a vertex *xuu*. The edges of *M(G)* consist of all arcs of the form *(u, xuu), (xuu, u),* and *(xuu, xuw),* where *uu* and *uw* are consecutive edges in *p(u),* with *uw* following *uu*.
An example of a medial digraph is shown in Figure 12.16. Since *G* is planar, *M(G)* will also be planar.
Notice that $G^{\overline{p}}$ is the digraph converse of *Gp,* obtained by reversing all directed edges. If *G* is non-orientable, then $G^{\overline{p}} \cong G^{p}$. Otherwise $G^{\overline{p}} \ncong G^{p}$. This provides a means of determining whether an embedding is orientable. It also gives information about the automorphisms of the planar embedding.

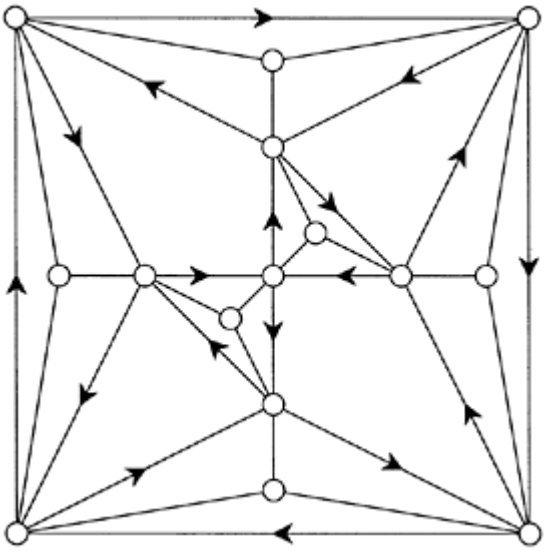


FIGURE 12.16
A medial digraph

Exercises

12.11.1 Find all triangulations on 4, 5, 6, and 7 vertices.

12.11.2 Prove that G^{p_1} and $G^{p_2}$are *iso*morphic embeddings if and only if $M(G^{p_1})$ and $M(G^{p_2})$ are isomorphic digraphs.

12.11.3 Determine whether the platonic solids are orientable.

12.11.4 Prove that a planar embedding *Gp* is orientable if and only if *Gp** is orientable.

12.11.5 Determine which of the triangulations on 5, 6, and 7 vertices are orientable.

12.11.6 Determine the graph *G* for which *M(G)* is shown in Figure 12.16.

12.12 The 4-color problem

Given a geographic map drawn in the plane, how many colors are needed such that the map can be colored so that any two regions sharing a common border have different colors? In 1852, it was conjectured by Francis Guthrie that four colors suffice. This simple problem turned out to be very difficult to solve. Several flawed "proofs" were presented. Much of the development of graph theory originated in attempts to solve this conjecture. See AIGNER [2] for a development of graph theory based on the 4-color problem. In 1976, Appel and Haken announced a proof of the conjecture. Their proof was based on the results of a computer program that had to be guaranteed bug-free. A second computer proof

by ALLAIRE [3] appeared in 1977. Each of these approaches relied on showing that any planar graph contains one of a number of configurations, and that for each configuration, a proper coloring of a smaller (reduced) graph can be extended to a proper coloring of the initial graph. The computer programs generated all irreducible configurations, and colored them. In the Appel-Haken proof, there were approximately 1800 irreducible configurations. The uncertainty was whether all irreducible configurations had indeed been correctly generated. In 1995, ROBERTSON, SANDERS, SEYMOUR, and THOMAS [104] presented another proof, also based on a computer program, but considerably simpler than the original, requiring only 633 irreducible configurations.

In this section, we present the main ideas of Kempe's 1879 "proof" of the 4-color theorem.

Given a geographic map drawn in the plane, one can construct a dual graph, by placing a vertex in the interior of each region, and joining vertices by edges if they correspond to adjacent regions. Coloring the regions of the map is then equivalent to coloring the vertices of the dual, so that adjacent vertices are of different colors. Consequently, we shall be concerned with coloring the vertices of a planar graph.

THEOREM 12.21 **(4-Color theorem)** *Every planar graph can be properly colored with four colors*.

If *G* is any simple planar graph, then it is always possible to extend *G* to a simple triangulation, by adding diagonal edges in non-triangular faces. Therefore, if we can prove that all simple planar triangulations are 4-colorable, the result will be true for all planar graphs. Hence we assume that we are given a planar triangulation *Gn* on *n* vertices. We attempt to prove the 4-color theorem (Theorem 12.21) by induction on *n*. The colors can be chosen as the numbers {1,2,3,4}. Given a coloring of *G,* then the subgraph induced by any two colors *i* and *j* is bipartite. We denote it by *Kij*.

DEFINITION 12.20: Given any 4-coloring of a planar graph *G,* each connected component of *Kij* is called a *Kempe* component. The component containing a vertex *x* is denoted *Kij(x)*. A path in *Kij* between vertices *u*

and *u* is called a *Kempe chain*.
Notice that if we interchange the colors *i* and *j* in any Kempe component, we obtain another coloring of *G*.
Now let *Gn* be a simple triangulation on *n* vertices. If *n*=4, then *Gn*=*K*4. It is clear that Theorem 12.21 is true in this case. Assume that *n*>4. By Corollary 12.17, we know that *Gn* has a vertex of degree three, four, or five. Let *u* be such a vertex. Using Algorithm 12.8.1, we reduce *Gn* to a simple planar triangulation *Gn*−1 by deleting *u* and adding up to two diagonals in the resulting

Page 303
face. We assume as an induction hypothesis, that *Gn*−1 has a 4-coloring. There are three cases.
Case 1. DEG*(u)*=3.
Let the three adjacent vertices to *u* be *(x,y,z)*. They all have different colors. Therefore there is a fourth color available for *u,* giving a coloring of *Gn*.
Case 2. DEG*(u)*=4.
Let the four vertices adjacent to *u* in *Gn* be *(w, x, y, z),* with a diagonal wy in *Gn*−1. It is clear that *w, x,* and *y* have different colors. If *x* and *z* have the same color, then a fourth color is available for *u*. Otherwise, let *w, x, y, z* be colored 1, 2, 3, 4, respectively. There may be a Kempe chain from *x* to *z*. If there is no Kempe chain, interchange colors in the Kempe component *K*24*(x),* so that *x* and *z* now both have color 4. If there is a Kempe chain from *x* to *z,* there can be no Kempe chain from *w* to *y,* for it would have to intersect the *xz*-Kempe chain. Interchange colors in *K*13*(w),* so that *w* and *z* now both have color 3. In each case there is a fourth color available for *u*.

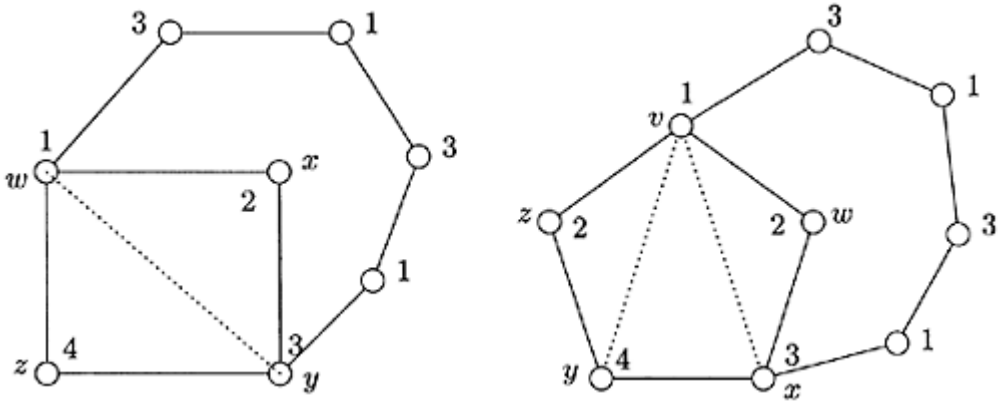


FIGURE 12.17
Kempe chains
Case 3. DEG*(u)*=5.
Let the five vertices adjacent to *u* in *Gn* be *(u, w, x, y, z),* with diagonals *ux* and *uy* in *Gn*−1. It is clear that *u, x,* and *y* have different colors. Since we have a 4-coloring of *Gn*−1, the pentagon *(u, w, x, y, z)* is colored in either 3 or 4 colors. If it is colored in three colors, there is a fourth color available for *u*. If it is colored in four colors, then without loss of generality, we can take these colors to be (1, 2, 3, 4, 2), respectively. If *K*13*(u)* contains no *ux*-Kempe chain, then we can interchange colors in *K*13*(u),* so that *u* and *x* are now both colored 3. Color 1 is then available for *u*. If *K*14*(u)* contains

Page 304
no *uy*-Kempe chain, then we can interchange colors in *K* 14*(u),* so that *u* and *y* are now both colored 4. Color 1 is again available for *u*. Otherwise there is a Kempe chain *Pux* connecting *u* to *x* and a Kempe chain *Puy* connecting *u* to *y*. It follows that *K*24*(w)* contains no *wy*-Kempe chain, as it would have to intersect *Pux* in *K*l3*(u)*. Similarly, *K*23*(z)* contains no *uz*-Kempe chain, as it would have to intersect *Puy* in *K*14*(u)*. If *Pux* and *Puy* intersect only in vertex *u,* then we can interchange colors in both *K*24*(w)* and *K*23*(z),* thereby giving *w* color 4 and *z* color 3. This makes color 2 available for *u*. The difficulty is that *Pux* and Puy can intersect in several vertices. Interchanging colors in *K*24*(w)* can affect the other Kempe chains, as shown in Figure 12.18, where the pentagon *(u, w, x, y, z)* is drawn as the outer face.

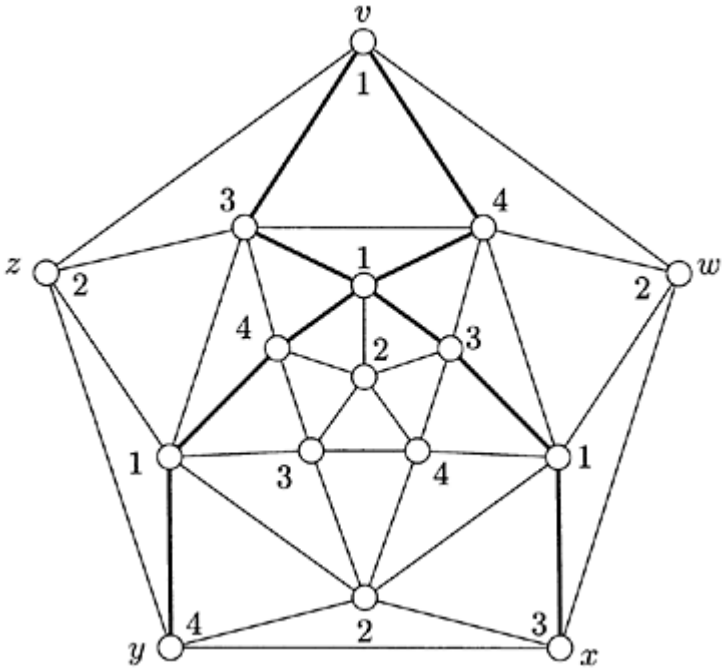


FIGURE 12.18
Intersecting Kempe chains

Although this attempted proof of Theorem 12.21 fails at this point, we can use these same ideas to prove the following.

THEOREM 12.22 **(5-Color theorem)** *Any planar graph can be colored in five colors*.

PROOF See Exercise 12.12.1.

Appel and Haken's proof of the 4-color theorem is based on the important concept of *reducibility*. Given a graph *G,* a *reducible configuration H* is a subgraph of *G* with the property that *H* can be reduced to a smaller subgraph *H′,* such that a 4-coloring of *H′* can be extended to all of *H* and *G*. If every planar graph contained a reducible configuration, then every planar graph could be 4-colored. Appel and Haken's proof was essentially a computer program to construct all irreducible configurations, and to show that they could be 4-colored. The difficulty with this approach is being certain that the computer program is correctly constructing all irreducible configurations. The reader is referred to SAATY and KAINEN [106] or WOODALL and WILSON [128] for more information on reducibility.

Exercises

12.12.1 Prove Theorem 12.22, the 5-color theorem.

12.12.2 Let *G* be a planar triangulation with a separating 3-cycle *(u, u, w)*. Let *H* and *K* be the two connected subgraphs of *G* that intersect in exactly *(u, u, w),* such that $G = H \cup K$. Show how to construct a 4-coloring of *G* from 4-colorings of *H* and *K*.

12.12.3 Let *G* be a planar triangulation with a separating 4-cycle *(u,u,w,x)*. Let *H* and *K* be the two connected subgraphs of *G* that intersect in exactly *(u, u, w, x),* such that $G = H \cup K$. Show how to construct a 4-coloring of *G* from 4-colorings of the triangulations *H+uw* and *K+uw*. *Hint: u, u,* and *w* can be assumed to have the same colors in *H* and *K*. If *x* is colored differently in *H* and *K,* look for an *xu*-Kempe chain, try interchanging colors in *Kij(x),* or try coloring *H+ux* and *K+ux*.

12.12.4 All lakes are blue. Usually all bodies of water are colored blue on a map. Construct a planar graph with two non-adjacent vertices that must be blue, such that the graph cannot be colored in four colors subject to this requirement.

12.13 Straight-line drawings

Every simple planar graph can be drawn in the plane with no edges crossing, so that each edge is a straight line. Read's Algorithm is a linear-time algorithm for doing this. It is based on the triangulation reduction. Suppose that *Gn* is a triangulation on *n* vertices that has been reduced to a triangulation *Gn*−1 on *n*−1 vertices, by deleting a vertex *u* as in Algorithm 12.8.1, and adding up to two edges *e* and *e′*. Suppose that a straight-line embedding of *Gn*−1 has already been computed. If DEG*(u)*=3, let *x, y, z* be the adjacent vertices. We can place *u* inside the triangle *(x, y, z)* to obtain a straight-line embedding of

Page 306
Gn. If DEG*(u)*=4, the edge *e* is a diagonal of a quadrilateral in *Gn*−1. We can place *u* on the line representing *e* to obtain a straight-line embedding of *Gn*.
Suppose now that DEG*(u)*=5. The edges *e*=*ux* and *e′*=*uy* are diagonals of a pentagon in *Gn*−1. This pentagon may have several possible polygonal shapes, which are illustrated in Figure 12.19. The triangle *(u, x, y)* is completely contained inside the pentagon. Inside *(u, x, y),* there is a "visible" region, shown shaded gray. The visible region can be calculated, by extending the lines of the adjacent triangles with sides *ux* and *uy,* and intersecting the half-planes with the triangle *(u, x, y)*. Vertex *u* can then be placed inside the visible region to obtain a straight-line embedding of *Gn*. Thus in each case, a straight-line embedding of *Gn*−1 can be extended to *Gn*.
This gives:
THEOREM 12.23 **(Fáry's theorem)** *Every planar graph has a straight-line embedding*.

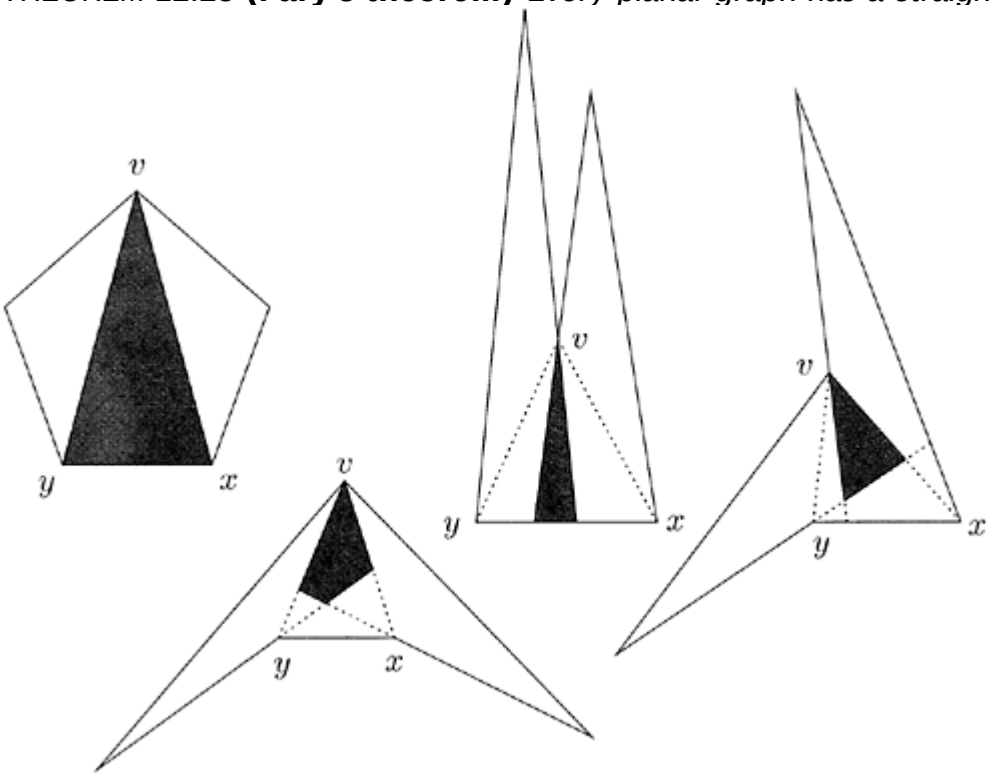


FIGURE 12.19
The "visible" region
Read's algorithm begins by triangulating *G* if *ε*<3*n*−6. It then deletes a

Page 307
sequence of vertices *u*1, *u*2,..., *un*−4 to reduce *G* to *K*4. It next assigns a planar coordinatization to the vertices of *K*4, and then restores the deleted vertices in reverse order. For each vertex *ui* deleted, it is necessary to store *ui* and its degree, so that it can later be correctly restored to the graph. Finally, the triangulating edges are removed. The result is a straight-line embedding of *G*.

Algorithm 12.13.1: READSALGORITHM*(GP)*

comment: Given a simple planar graph G on $n \geq 4$ vertices with rotation system p, construct a straight-line drawing of G in the plane.

triangulate *G* without creating multiple edges or loops
mark all triangulating edges as "virtual" edges
i←1
while *n*>4
do
$G \leftarrow$ REDUCEGRAPH(G)
$u_i \leftarrow$ the vertex that was deleted
$i \leftarrow i + 1$
$n \leftarrow n - 1$

comment: *G* is now *K*4
assign pre-chosen coordinates to the vertices of *K*4
for *i*=*n*−4 **downto** 1
do { calculate the visible region for u_i ; restore u_i to G }
remove all virtual edges from *G*

It is easy to see that Read's algorithm is *O(n)*. It takes *O(n)* steps to compute the degrees of *G,* and to triangulate *G*. It takes *O(n)* steps to reduce *G* to *K*4, and then *O(n)* steps to rebuild *G*. Read's algorithm can be modified by initially choosing any facial cycle *F* of *G,* and assigning coordinates to the vertices of *F* so that they form a regular convex polygon. The reduction to *K*4 is then modified so that vertices of the outer facial cycle *F* are never deleted. The result is a planar embedding with the given facial cycle as the outer face.

The embeddings produced by Read's algorithm are usually not convex embeddings. Tutte has shown how to produce a straight-line embedding of a graph such that all faces are convex regions, by solving linear equations. Consider any face of *G,* with facial cycle (*u*1, *u*2,..., *uk*). We begin by assigning coordinates to *u*1, *u*2,..., *uk* such that they form a convex polygon in the plane. This will be the outer face of a planar embedding of *G*.

Tutte then looks for a coordinatization of the remaining vertices with the special property: the coordinates of *ui,* where *i*>*k,* are the *average* of the coordinates of

all adjacent vertices. A coordinatization with this property is called a *barycentric coordinatization*. We can express it in terms of matrices as follows.

Let *A* be the adjacency matrix of *G* such that the first *k* rows and columns correspond to vertices *u*1, *u*2,..., *uk*. Let *D* be the *n*×*n* diagonal matrix such that entry *Dii* equals 1, if *i*≤*k*. If *i*>*k,* entry *Dii* equals DEG*(ui)*. Let *X* be the vector of *x*-coordinates of the vertices, and let *Y* be the vector of *y*-coordinates. Construct a matrix *B* from *A* by replacing the first *k* rows with zeroes. Then the first *k* entries of *BX* are zero. But if *i*>*k,* the *ith* entry is the sum of the *x*-coordinates of vertices adjacent to *ui*. Let *Xk* denote the vector whose first *n* entries are the *x*-coordinates of *u*1,..., *uk,* and whose remaining entries are zero. *Yk* is similarly defined. Then the barycentric condition can be written as

$$DX = Xk + BX,\ DY = Yk + BY.$$

These equations can be written as $(D-B)X = Xk$ and $(D-B)Y = Yk$. Consider the matrix $D-B$.

LEMMA 12.24 The matrix D−B is invertible.

PROOF Consider the determinant det(*D*−*B*). The first *k* rows of *D*−*B* look like an identity matrix. Expanding the determinant along the first *k* rows gives det(*D*−*B*)=det*(K)* where *K* is the matrix formed by the last *n*−*k* rows and columns. *K* looks very much like a Kirchhoff matrix, except that the degrees are not quite right. In fact, if we construct a graph *G′* from *G* by identifying *u*1, *u*2,..., *uk* into a single vertex *u*0, and deleting the loops created, then *K* is formed from the Kirchhoff matrix *K(G′)* by deleting the row and column corresponding to *u*0. It follows that $(K) = \pm\tau(G')$, by the matrix-tree theorem. Since *G′* is a connected graph, det*(K)*≠0, so that *D*−*B* is invertible.

It follows that the barycentric equations have a unique solution, for any assignment of the coordinates *Xk* and *Yk*. Tutte has shown that if *G* is a 3-connected planar graph, this solution has remarkable properties: if we begin with a convex polygon for the outer face, the solution is a planar coordinatization with straight lines, such that no edges cross. All faces, except the outer face, are convex regions. No three vertices of any facial cycle are collinear.

It is fairly easy to see that a planar barycentric coordinatization of a 3-connected graph must have convex faces. For consider a non-convex face, as in Figure 12.20. Vertex *u* is a corner at which a polygonal face is non-convex. Clearly vertex *u* is not on the outer face. Let the two adjacent vertices of the polygon be *u* and *w*. Since *G* is 3-connected, *u* has at least another adjacent vertex. All other vertices adjacent to *u* must be in the angle between the lines *uu* and *uw* as indicated in the diagram, since *G* is 3-connected, and there are no crossing edges. But then all

vertices adjacent to *u* are to the right of the dotted line, which is impossible in a barycentric coordinatization.

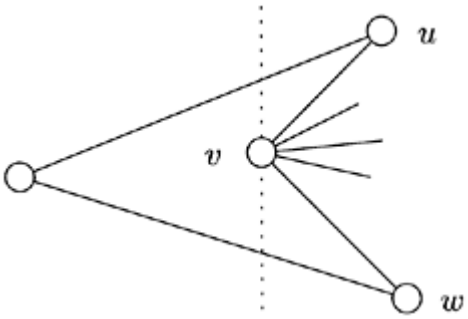


FIGURE 12.20
A non-convex face

12.14 Kuratowski's theorem

In this section we will prove Kuratowski's theorem. The proof presented is based on a proof by KLOTZ [73]. It uses induction on *ε(G)*.

If *G* is a disconnected graph, then *G* is planar if and only if each connected component of *G* is planar. Therefore we assume that *G* is connected. If *G* is a separable graph that is planar, let *H* be a block of *G* containing a cut-vertex *u*. *H* is also planar, since *G* is. We can delete *H*−*u* from *G,* and find a planar embedding of the result. We then choose a planar embedding of *H* with *u* on the outer face, and embed *H* into a face of *G* having *u* on its boundary. This gives:

LEMMA 12.25 A separable graph is planar if and only if all its blocks are planar.

So there is no loss in generality in starting with a 2-connected graph *G*.

THEOREM 12.26 **(Kuratowski's theorem)** *A graph G is planar if and only if it contains no subgraph TK*3, 3 *or TK*5.

PROOF It is clear that if *G* is planar, then it contains no subgraph *TK*3 , 3 or *TK*5. To prove the converse, we show that if *G* is non-planar, then it must contain *TK*3, 3 or *TK*5. We assume that *G* is a simple, 2-connected graph with *ε* edges. To start the induction, notice that if *ε*≤6, the result is true, as all graphs with *ε*≤6 are planar. Suppose that the theorem is true for all graphs with at most *ε*−1 edges. Let *G* be non-planar, and let $ab \in E(G)$ be any edge of *G*. Let *G′*=*G*−*ab*. If *G′* is non-planar, then by the induction hypothesis, it contains a *TK*3 ,3 or *TK*5, which is also a subgraph of *G*. Therefore we assume that *G′* is planar. Let *κ(a, b)* denote the number of internally disjoint *ab*-paths in *G′*. Since *G* is 2-connected, we know that *κ(a, b)*≥1.

Case 1. *κ(a, b)*=1.

G′ has a cut-vertex *u* contained in every *ab*-path. Add the edges *au* and *bu* to *G′,* if they are not already present, to get a graph *H,* with cut-vertex *u*. Let *Ha* and *Hb* be the blocks of *H* containing *a* and *b,* respectively. If one of *Ha* or *Hb* is non-planar, say *Ha,* then by the induction hypothesis, it contains a *TK*3, 3 or *TK*5. This subgraph must use the edge *au,* as *G′* is planar. Replace the edge *au* by a path consisting of the edge *ab* plus a *bu*-path in *Hb*. The result is a *TK*3, 3 or *TK*5 in *G*. If *Ha* and *Hb* are both planar, choose planar embeddings of them with edges *au* and *bu* on the outer face. Glue them together at vertex *u,* remove the edges *au* and *bu* that were added, and restore *ab* to obtain a planar embedding of *G,* a contradiction.

Case 2. *κ(a, b)*=2.

Let *P*1, and *P*2 be two internally disjoint *ab*-paths in *G′*. Since *κ(a, b)*=2, there is a vertex $u \in P_1$ and $v \in P_2$ such that all *ab*-paths contain at least one of *{u, u}*, and *G′*−*{u, u}* is disconnected. If *Ka* and *Kb* denote the connected components of *G′*−*{u, u}* containing *a* and *b,* respectively, let G'_a be the subgraph of *G′* induced by $K_a \cup \{u, v\}$, let G'_b be the subgraph induced by $K_b \cup \{u, v\}$. Now add a vertex *x* to G'_a, adjacent to *u, u,* and *a* to obtain a graph *Ha*. Similarly, add *y* to G'_b adjacent to *u, u,* and *b* to obtain a graph *Hb*. Suppose first that *Ha* and *Hb* are both planar. As vertex *x* has degree three in *Ha,* there are three faces incident on *x*. Embed *Ha* in the plane so that the face with edges *ux* and *xu* on the boundary is the outer face. Embed *Hb* so that edges *uy* and *yu* are on the boundary of the outer face. Now glue *Ha* and *Hb* together at vertices *u* and *u,* delete vertices *x* and *y,* and add the edge *ab* to obtain a planar embedding of *G*. Since *G* is non-planar, we conclude that at least one of *Ha* and *Hb* must be non-planar. Suppose that *Ha* is non-planar. It must contain a subgraph *TK*5 or *TK*3, 3. If the *TK*5 or *TK*3, 3 does not contain *x,* then it is also contained in *G,* and we are done. Otherwise the *TK*5 or *TK*3, 3 contains *x*. Now *Hb* is 2-connected (since *G* is), so that it contains internally disjoint

Page 311
paths *Pbu* and *Pbu* connecting *b* to *u* and *u,* respectively. These paths, plus the edge *ab,* can be used to replace the edges *ux,ux,* and *ax* in *Ha* to obtain a *TK*5 or *TK*3, 3 in *G*.
Case 3. $\kappa(a, b) \geq 3$.
Let *P*1, *P*2, and *P*3 be three internally disjoint *ab*-paths in *G′*. Consider a planar embedding of *G′*. Each pair of paths $P_1 \cup P_2$, $P_1 \cup P_3$, and $P_2 \cup P_3$ creates a cycle, which embeds as a Jordan curve in the plane. Without loss of generality, assume that the path *P*2 is contained in the interior of the cycle $P_1 \cup P_3$, as in Figure 12.21. The edge *ab* could be placed either in the interior of $P_1 \cup P_2$ or $P_2 \cup P_3$, or else in the exterior of $P_1 \cup P_3$. As *G* is non-planar, each of these regions must contain a path from an interior vertex of *Pi* to an interior vertex of *Pj*. Let *P*12 be a path from *u*1 on *P*1 to *u*2 on *P*2. Let *P*13 be a path from *u*1 on *P*1 to *u*3 on *P*3. Let *P*23 be a path from *u*2 on *P*2 to *u*3 on *P*3. If $u1 \neq u1$, contract the edges of *P*1 between them. Do the same for *u*2, *u*2 on *P*2 and *u*3, *u*3 on *P*3. Adding the edge *ab* to the resultant graph then results in a *TK*5 minor. By Theorem 12.5, *G* contains either a *TK*5 or *TK*3, 3.

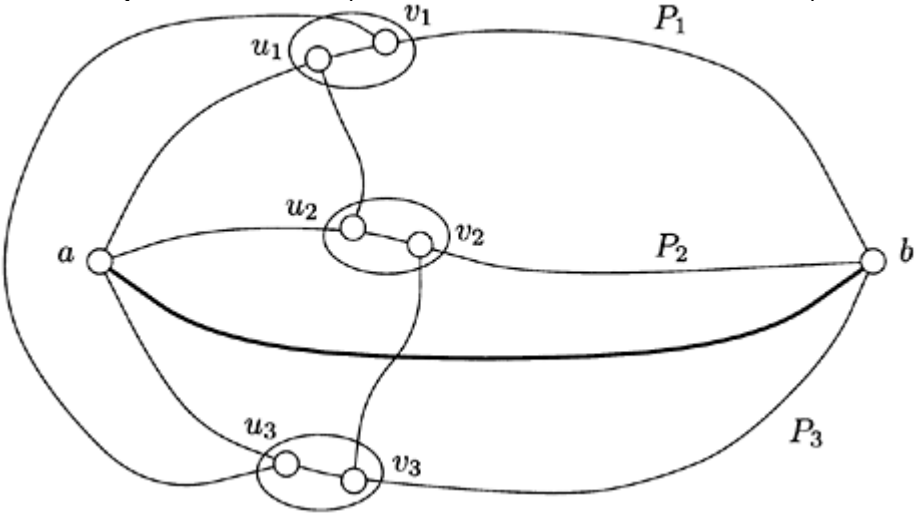


FIGURE 12.21
A *K*5 minor
We can also state Kuratowski's theorem in terms of minors.
THEOREM 12.27 **(Wagner's theorem)** *A graph G is planar if and only if it does not have K*3, 3 *or K*5 *as a minor*.

Page 312
PROOF It is clear that if *G* is planar, then it does not have *K*3 ,3 or *K*5 as a minor. Conversely, if *G* does not have *K*3, 3 or *K*5 as a minor, then it cannot have a subgraph *TK*3, 3 or *TK*5. By Kuratowski's theorem, *G* is planar.
The graphs *K*5 and *K*3, 3 are called *Kuratowski graphs*.
Exercises
12.14.1 Find a *TK*3, 3 or *TK*5 in the Petersen graph.
12.14.2 Find a *TK*3, 3 or *TK*5 in the graph of Figure 12.22.
12.14.3 Show that if *G* is a non-planar 3-connected graph, then either $G = K5$, or else *G* contains a *TK*3, 3.
12.14.4 Let *G* be a graph with a separating set *{u, u}*. Let *H′* be a connected component of $G - \{u, u\}$, and let *H* denote the graph induced by $V(H') \cup \{u, v\}$. Let *K* be the graph induced by $V(G) - V(H')$, so that $G = H \cup K$, and *H* and *K* intersect only in *{u, u}*. Let $H+ = H+uu$ and $K+ = K+uu$. Show that *G* is planar if and only if *H+* and *K+* are planar.
12.14.5 Let *G* be a 3-connected graph with at least five vertices. Show that *G* contains an edge *xy* such that $G \cdot xy$ is 3-connected. *Hint:* If $G \cdot xy$ is not 3-connected, choose *xy* so that the subgraph *K* of the preceding exercise is as large as possible, and find a contradiction. This was proved by THOMASSEN [114].
12.14.6 Suppose that a coordinatization in the plane of an arbitrary 3-connected graph *G* on *n* vertices is obtained by solving the barycentric equations $(D - B)X = Xk$ and $(D - B)Y = Yk$. Describe an $O(n3)$ algorithm which determines whether *G* is planar, and constructs a rotation system, using the coordinates *X* and *Y*.
12.15 The Hopcroft-Tarjan algorithm
A number of different algorithms for planarity-testing have been developed. The first linear-time algorithm was the Hopcroft-Tarjan planarity algorithm. Given a 2-connected graph *G* on *n* vertices, it determines whether *G* is planar in *O(n)* time. If *G* is found to be planar, it can be extended to construct a rotation

system, too. If *G* is found to be non-planar, it can also be modified to construct a *TK*5 or *TK*3, 3 subgraph, although this is somewhat more difficult. We present a simplified version of the Hopcroft-Tarjan algorithm here.

There is no loss in generality in starting with a 2-connected graph *G*. Suppose first that *G* is hamiltonian, and that we are given a hamilton cycle *C,* and number the vertices of *C* consecutively as 1, 2,..., *n*. The remaining edges of *G* are

Page 313

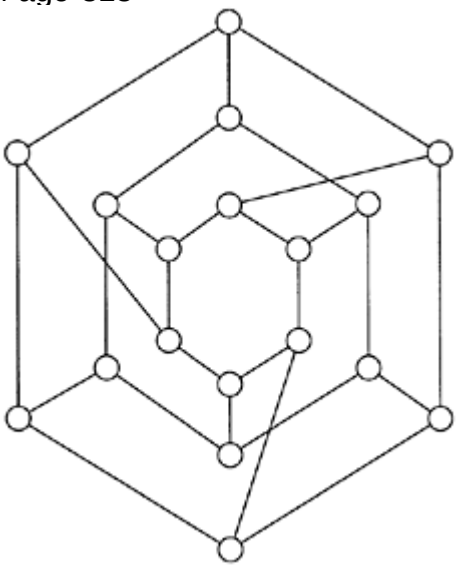


FIGURE 12.22
*Find a TK***3, 3**

chords of *C*. For each vertex *u,* order the adjacent vertices (*u*1, *u*2,..., *uk*) so that *u*1<*u*2<...<*uk*. We then start at vertex *u*=*n,* and follow the cycle *C* back to vertex 1. As we proceed, we will place each chord *uu* either inside *C* or outside *C*. When we have returned to vertex 1, we will have constructed a planar embedding of *G*. We draw the cycle *C* as in Figure 12.23, with the path from 1 to n starting near the top, and moving down the page.

Consider the example of Figure 12.23. The algorithm stores two linked lists of chords, one for the inside of *C,* and one for the outside of *C*. We denote these as *Li* and *Lo,* respectively. Each linked list defines a sequence of chords [*u*1*u*1, *u*2*u*2, *u*3*u*3,...] as they are added to the embedding. The inside of *C* appears in the diagram to the left of the path from 1 to *n*. The outside of *C* appears to the right of the path. In the example, the algorithm begins at vertex *n*=7 with adjacent vertices (1, 3, 4, 6). It first places the "chord" (7, 1), which really completes the cycle, on the inside linked list. It then places chords (7, 3) and (7, 4) also on *Li*. The inside linked list is now [(7, 1), (7, 3), (7, 4)]. The chord (7, 4) is called the *leading* chord in the linked list. The next chord to be inserted is to be placed after it. To determine whether the next chord *(u, u)* fits on the inside, it need only compare its endpoint *u* with the upper endpoint of the current leading chord of *Li*. After placing the chords incident on vertex 7, the algorithm moves to vertex 6, where it sees the chord (6, 1). This will not fit on the inside (because 1<4), but is easily placed on the outside linked list. It then moves to vertex 5,

Page 314

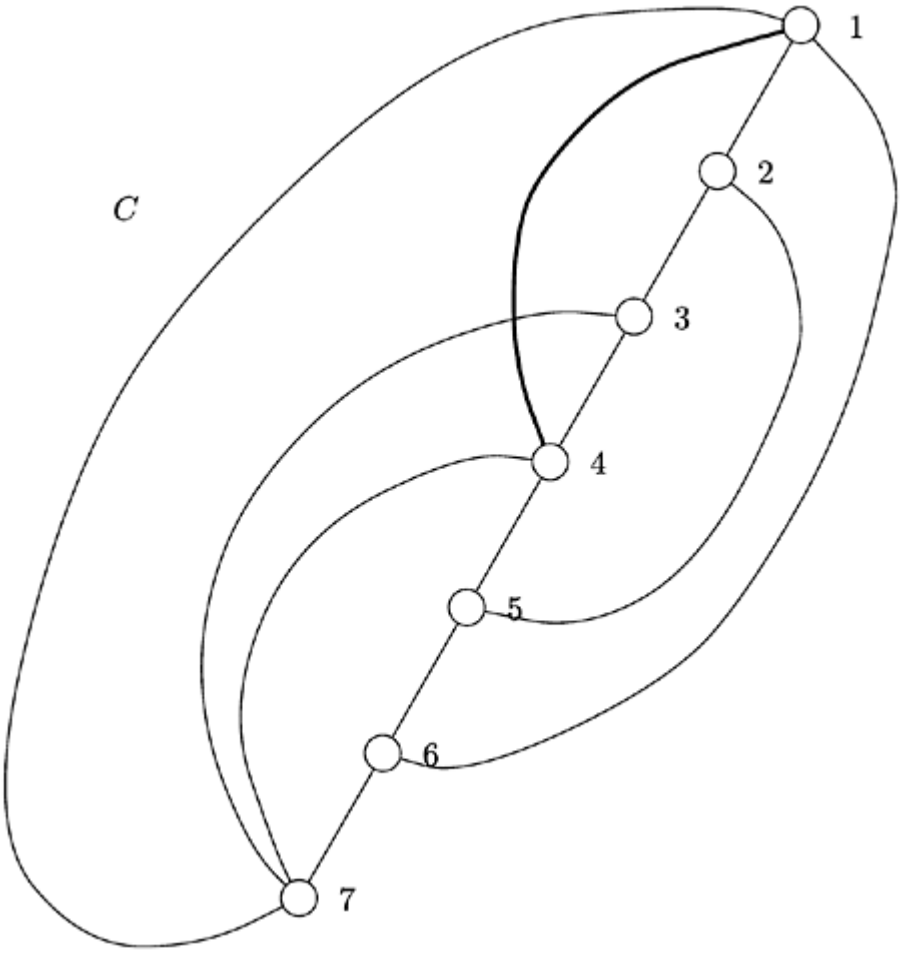


FIGURE 12.23
The Hopcroft-Tarjan algorithm

and places (5, 2) also on *Lo*. The outside linked list is then [(6, 1), (5, 2)], where (5, 2) is the leading chord. It then moves to vertex 4, where it sees the chord (4, 1). When the algorithm moves up the cycle to vertex 4, the leading chord of *Li* is moved past (7, 4) to (7, 3), because the chord (4, 1) is above (7, 4). It then determines that (4, 1) will not fit on the inside (because 1<3, where (7,3) is the leading chord of *Li*); and that (4,1) will not fit on the outside (because 1<2, where (5, 2) is the leading chord of *Lo*). Therefore *G* is non-planar. In fact the cycle *C,* together with the three chords (7, 3), (5, 2), and (4, 1) form a subgraph *TK*3,3, which we know to be non-planar.

Page 315

Algorithm 12.15.1: BASICHT*(G, C)*

comment: {Given a 2-connected graph G with a hamilton cycle $C = (1, 2, \ldots, n)$. Determine whether G is planar.

for *u*←*n* **downto** 1

```
suppose that u is adjacent to (v_1, v_2, ..., v_k)
for j ← 1 to k
    do  if v_j ≥ u − 1 go to L1
        comment: uv_j is a chord with v_j above u
        if uv_j fits inside C, place it in L_i
          else if uv_j fits outside C, place it in L_o
        do  m = SWITCHSIDES(u, v_j)
            if m = 0 return (NonPlanar)
            if m = 1
              then  comment: uv_j now fits inside C
                    place uv_j inside C
              else if m = −1
              then  comment: uv_j now fits outside C
                    place uv_j outside C
        L1 :
```

We must still describe what SWITCHSIDES*(u, u)* does. Its purpose is to determine whether some chords of *Li* and *Lo* can be interchanged to allow *(u, u)* to be embedded. Its description will follow.

This simplified version of the Hopcroft-Tarjan algorithm contains the main features of the complete algorithm, but is much easier to understand. The algorithm stores two linked lists, *Li* and *Lo*. Each list has a *leading chord*—the chord after which the next chord is to be placed when inserted in the list as the sequence of chords is extended. Initially the leading chord will be the last chord in the linked list, but this will change as the algorithm progresses. It is convenient to initialize both linked lists with dummy chords, so that each list has at least one chord. Each chord stored in a linked list will be represented by a pair, denoted *(LowerPt, UpperPt)*, where *UpperPt* is the endpoint with the smaller value—it is above the *LowerPt* in the diagram. As the chords are placed in the linked lists, a linear order is thereby defined on the chords in each list. Given a chord *(u, u),* where *u*>*u,* the next chord after *(u, u)* is the chord *nested immediately inside (u, u),* if there is such a chord. If there is no such chord, then the next chord after *(u, u)* is the *first chord below (u, u)* if there is one; that is, the first chord whose *UpperPt* ≥*u*. To determine whether a chord *(u, u)* fits either inside or outside *C,* we need only compare *u* with the leading chord's *UpperPt*. It fits if *u*≥*UpperPt*. As *u* moves up the cycle, we must adjust the pointer to the leading chord on both sides

to ensure that the leading chord always satisfies *UpperPt*<*u*.

So we know how to store the chords, and how to determine whether a chord *(u, u)* fits in either side. If it fits, we insert it in the appropriate linked list and continue with the next chord. What if *(u, u)* will not fit in either side?

12.15.1 Bundles

Two chords (*u*1, *u*1) and (*u*2, *u*2) are said to be in *conflict* if either *u*1>*u*2> *u*1>*u*2 or *u*2>*u*1>*u*2>*u*1. Conflicting chords cannot both be placed on the same side of *C*. We define a *conflict graph K* whose vertices are the set of all chords currently in the linked lists. Two chords are adjacent in *K* if they are in conflict. A set of chords corresponding to a connected component of the conflict graph is called a *bundle* (originally called a "block" in HT's paper; however, the term "block" has another graph theoretical meaning). The conflict graph must alway s be bipartite, as two chords in the same linked list must never be in conflict. Therefore each bundle *B* is also bipartite—the bipartition of a bundle consists of the chords inside the cycle, denoted *Bi,* and the chords outside the cycle, denoted *Bo*. A typical bundle is shown as the shaded area in Figure 12.24. The two shaded zones represent *Bi* and *Bo* for one bundle. Any chord inside a shaded zone conflicts with all chords in the opposite shaded zone. Notice that if the conflict graph is not bipartite, that it is impossible to assign the chords to the inside and outside of *C*. Consequently, *G* must be non-planar in such a case.

Now the conflict graph *K* is changing dynamically as the algorithm progresses. Therefore the bundles are also changing. However, they change in a simple way. Each chord is initially a bundle by itself, until it is found to conflict with another chord. In a 1-chord bundle, one of *Bi* and *Bo* will be empty. The algorithm will store the bundles in a stack.

Consider the situation where no conflicts have yet been discovered, so that all chords so far have been placed on *Li*. The algorithm is visiting vertex *u* on *C,* attempting to place a chord *(u, u)*. The leading inside chord satisfies *LowerPt*≥*u*>*UpperPt*. It belongs to a 1-chord bundle, the *current* bundle. The algorithm first attempts

to nest *(u, u)* inside the leading inside chord. If it fits, then *(u, u)* becomes the new leading inside chord, and a new 1-chord bundle is created containing only *uu,* which becomes the current bundle—*the bundles are nested*. The innermost bundle is always the *current bundle,* and is stored at the top of the bundle stack.
If *(u, u)* will not fit on the inside, it conflicts with the leading inside chord. *(u, u)* may conflict with several chords of the inside linked list. They will be *consecutive chords* of *Li,* preceding the leading chord. These chords initially belong to different bundles, but will become merged into the current bundle *B,* thereby forming *Bi,* when *(u, u)* is placed in the outside list. *Bo* will consist of *(u, u)*. If *B* denotes the current bundle, we will call *Bi* and *Bo* the current inside

Page 317

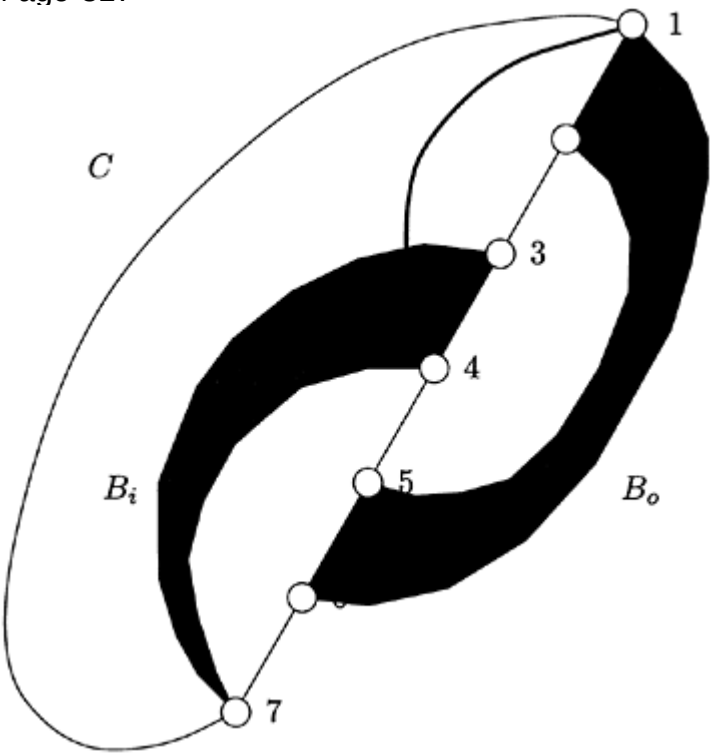


FIGURE 12.24
Bundles of chords

and outside bundles, although they are part of the same bundle.
At this point we can say that *the current inside and outside bundles consist of one or more consecutive chords in the two linked lists*. This will be true at each point of the algorithm.
So in order to represent a bundle *B,* we require two pointers into each of *Li* and *Lo,* being the first and last chords of *Li* that belong to *Bi* and the first and last chords of *Lo* that belong to *Bo*. We could switch the chords of *Bi* to *Lo* and the chords of *Bo* to Li by reassigning four pointers. Because the bundles are nested, we store them as a stack.
When a chord *(u, u)* is placed in one of the linked lists, and it does not conflict with the leading chord on either side, a new current bundle *B* containing *(u, u)* is created, nested inside the previous current bundle. The current bundle is the one at the top of the stack. As vertex *u* moves up the cycle, we eventually have $u \leq UpperPt$ for the uppermost chord in the current bundle. The current bundle *B* is then no longer relevant, as no more chords can conflict with it. Therefore *B*

Page 318
is removed from the stack and deleted. The next bundle *B′* on the stack becomes the current bundle. When this happens, the chords belonging to *Bi* or *Bo* often occur in *Li* or *Lo,* respectively, as a consecutive subsequence of chords contained within B_i' or B_o'. When *B* is deleted, the effect is to merge the chords of *B* into *B′*. Then *B′* becomes the new current bundle. Its chords are again consecutive chords of *Li* and *Lo*.

12.15.2 Switching bundles

We have a chord *(u, u)* that will not fit on the inside or outside of *C*. What do we do? There are several possible ways in which this can arise. Two of them are illustrated in Figure 12.25. The current bundle is shown shaded in gray.

In the left diagram of Figure 12.25, *(u, u)* conflicts with one or more chords in *Bi* and in *Bo*. If we form a subgraph consisting of the cycle *C*, the edge *(u, u),* and a conflicting chord from each of *Bi* and *Bo*, we have a subgraph *TK*3, 3 which we know is non-planar. In the right diagram of Figure 12.25, *(u, u)* conflicts with one or more chords in *Bo*. It does not conflict with any chords in *Bi* but it does conflict with the leading chord *(a, b)* of *Li*, which is in a previous inside bundle. If we interchange the the chords of *Bi* and *Bo,* there will be room to place *(u, u)* in *Lo*. This can be done in constant time, because the chords in each bundle are consecutive chords in the linked list. We only need to change a constant number of pointers to transfer a sequence of chords from *Li* to *L*o, and vice versa.
A third situation also exists which is nearly the mirror image of the right diagram of Figure 12.25, in which *(u, u)* conflicts with one or more chords in *Bi,* but does not conflict with any chords in *Bo,* and does conflict with the leading chord *(a, b)* of *Lo,* which is in a previous outside bundle. It can be handled in a similar way.
Suppose that a situation similar to the right diagram of Figure 12.25 exists. *B* is the current bundle, and chord *(u, u)* conflicts with a chord of *B*o, but not with *Bi*. The leading chord of *Lo* is in *Bo*. The leading chord of *Li* is *(a, b)*, which is not in *Bi*. Since every chord is contained in some bundle, *(a, b)* is in a previous bundle on the stack. The bundles are nested, so that *B* is nested within a bundle *B′,* which may in turn be nested within in a bundle *B″,* etc. Without loss of generality, suppose that there are at least three bundles on the stack, which begins *B, B′, B″,* and that *(a, b)* is in *B″*.
Now *B* is nested within *B′,* which is nested within *B″*. Since the leading chord of *Li* is in *B″,* it follows that *(u, u)* does not conflict with any chord of B_i', and that *(u, u)* does conflict with some chord of B_o'. So *(u, u)* conflicts with a chord of both *Bo* and B_o'. If *(u, u)* can be embedded, *Bo* and B_o' must be merged into one bundle, and they must both be on the same side of *C*. They will be in the same part of the bipartition of *K*. Therefore the algorithm merges *B* and *B′*. Call the result *B*. It then discovers that interchanging the chords of *Bi* and *Bo* allows

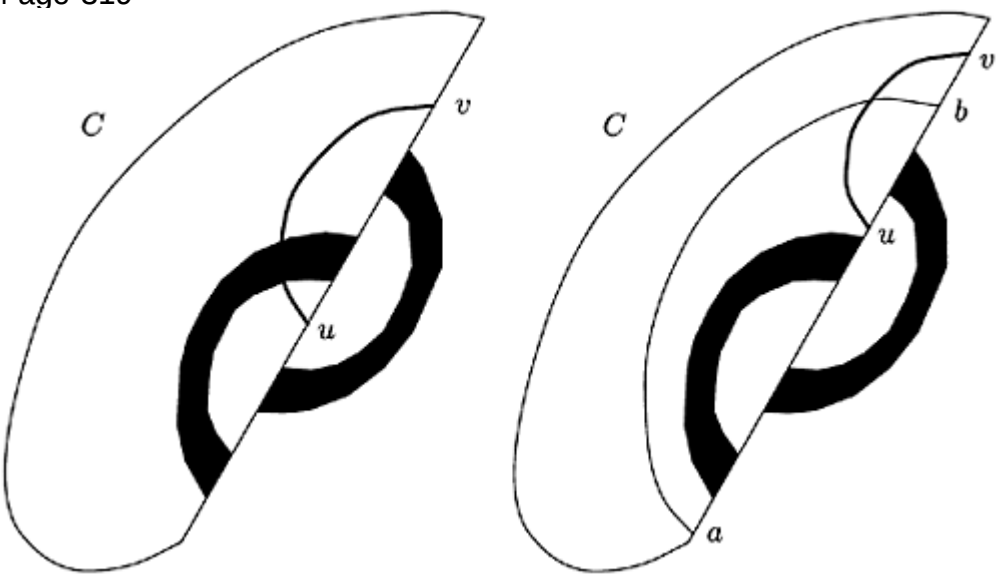


FIGURE 12.25
Switching bundles

(u, u) to be placed on *Lo*. Since *(u, u)* conflicts with *(a, b)* in *Li*, the bundles *B* and *B″* are then also merged. The properties of bundles are summarized as follows:
1. The bundles correspond to connected components of the conflict graph.
2. The bundles are nested inside each other, and consequently stored on a stack.
3. The current bundle is the one on the top of the stack.
4. The chords of each *Bi* and *Bo* form a contiguous subsequence of *Li* and *Lo*, respectively.
The description of SWITCHSIDES*(u, u)* Algorithm 12.15.2 can now be given. It is called when chord *(u, u)* conflicts with both the leading chord of *Li* and the leading chord of *Lo*. The algorithm needs to know whether the leading chord of *Li* is within *Bi*, and whether the leading chord of *Lo* is within *Bo*. This can be done by comparing the endpoints of the leading chords with the first and last chords of *Bi* and *Bo*. One of the leading chords is always in the current bundle. The other must be in a previous bundle, or the conflict graph will be non-bipartite, and *G* will be non-planar.
When *Bi* and *Bo* are interchanged, the leading chords in *Li* and *Lo* also change. Suppose that the leading

chord of *Lo* is within *Bo,* but that the leading chord of

Page 320

Li is in a previous bundle, as in Figure 12.25. The new leading chord of *Lo* can easily be found by taking the first chord of *Lo* following *Bo*. The new leading chord of *Li* is the former leading chord of *Lo*.

This procedure merges the bundles on the stack until either a non-bipartite conflict graph is found, in which case it returns zero, or until it becomes possible to place *(u, υ)* in one of *Li* or *Lo*. Notice that swapping the chords of *Bi* and *Bo* takes a constant number of steps, and that merging the current bundle with the previous bundle on the stack also takes a constant number of steps. The total number of bundles is at most the number of edges of *G,* so that the total number of steps required by SWITCHSIDES*(u, υ)* is *O(n),* summed over all iterations.

If SWITCHSIDES*(u, υ)* returns either 1 or −1, then it is possible to place *(u, υ)* inside or outside *C*. If SWITCHSIDES*(u, υ)* returns 0, then the current bundle contains chords of *Li* and *Lo* that conflict with *(u, υ),* so that the conflict graph is not bipartite. In this case *G* is non-planar. However, it is not particularly easy to find a *TK*5 or *TK*3, 3 in *G* in this situation. If the algorithm succeeds in placing all the chords, then a rotation system can be found from the order in which the chords occur in *Li* and *Lo*.

So the three main components of this simplified version of the Hopcroft-Tarjan algorithm are:

1. Construct the linked lists *Li* and *Lo* in the right order.

2. Construct the connected components of the conflict graph as a stack of bundles.

3. Keep the bundles up to date. Each time that a chord is added to one of *Li* or *Lo,* the current bundle must be updated.

These steps can all be performed in linear time.

Page 321

Algorithm 12.15.2: SWITCHSIDES*(u, υ)*

comment: *(u, υ)* conflicts with the leading chord of *Li* and *Lo*

while (true)

```
do { let B denote the current bundle
     let B' denote the previous bundle on the stack
     if the leading chord of L_o is within B_o
       then { if the leading chord of L_i is within B_i
                then return (0)        "non-planar"
              comment: { otherwise the leading chord of L_i
                       { is in a bundle previous to B
              if the leading chord of L_i is within B'_i
                then { interchange the chords of B_i and B_o
                       merge B and B'
                     then { find the new leading chords of L_i and L_o
                            if (u, v) does not conflict with L_o
                              then return (-1)
                                                "(u, v) now fits in L_o"
              else merge B and B' }
       else { comment: { the leading chord of L_i is within B_i
                       { the leading chord of L_o is
                       { in a bundle previous to B
              if the leading chord of L_o is within B'_o
                then { interchange the chords of B_i and B_o
                       merge B and B'
                     then { find the new leading chords of L_i and L_o
                            if (u, v) does not conflict with L_i
                              then return (1)
                                                "(u, v) now fits in L_i"
              else merge B and B' }
   }
```

12.15.3 The general Hopcroft-Tarjan algorithm

Up to now, we have assumed that we are given a hamilton cycle *C* in a 2-connected graph *G* which we are testing for planarity. If we are not given a starting hamilton cycle, the algorithm is the recursive extension of the hamilton cycle case. We give a brief sketch only. The first step is to perform a depth-first search in *G* starting from vertex 1, to assign a DF-numbering to the vertices, and to calculate the low-points of all vertices. The DFS will construct a DF-spanning tree *T* rooted at 1. Number the vertices of *G* according to their DF-numbers. It

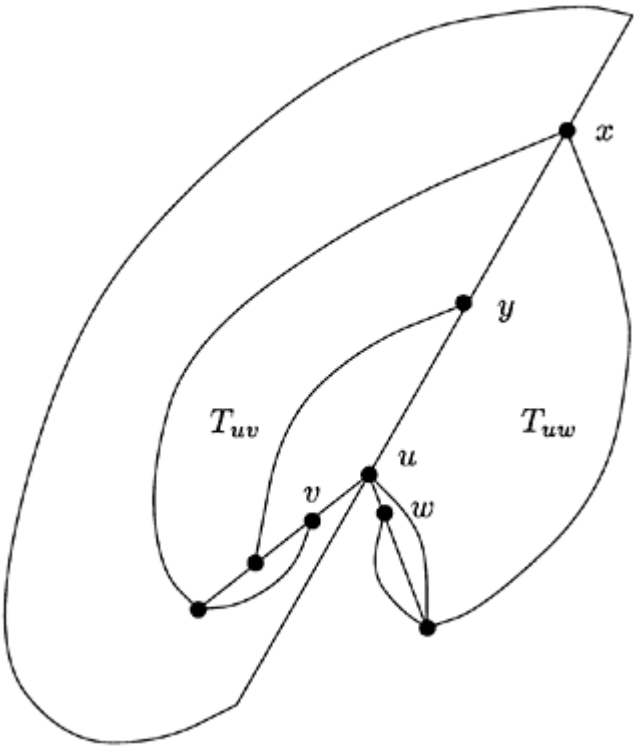


FIGURE 12.26
Subtrees *Tuu* **and** *Tuw*

can happen that the tree *T* is a hamilton path, in which case we have the proceeding situation exactly—the vertices are numbered consecutively along *T,* and as the DFS returns from recursion, it visits the vertices in the order *n, n*−1,..., 1. If we sort the chords incident on each vertex in order of increasing DF-number of the other endpoint, the algorithm is identical.

If the tree *T* is not a hamilton path, consider the situation where the DFS is visiting vertex *u,* and a recursive call

While visiting vertex *u,* the recursive call DFS*(u)* constructs a subtree *Tuu,* where *u* is adjacent to *u*. Refer to Figure 12.26. When *Tuu* is constructed, *LowPt[u]* is calculated. Since *G* is 2-connected, this is a vertex somewhere above *u* in *T*. The entire subtree *Tuu* behaves very much like a single chord *(u, LowPt[u])*. Therefore the vertices adjacent to *u* must be sorted according to *LowPt*[·] values, just as in the simplified algorithm (where *u* is used rather than *LowPt[u]*).

There are two kinds of possible subtree, and these are illustrated as *Tuu* and *Tuw* in Figure 12.26. Both *Tuu* and *Tuw* have the same *LowPt,* equal to *x*. Notice that *Tuu* has a frond with endpoint *y* between *u* and *x,* but that *Tuw* has no such

frond. We will call a subtree like *Tuw* a type I subtree, and a subtree like *Tuu* a type II subtree. It is easy to distinguish type I and II subtrees. The DFS can compute the second low-point as well as the low-point. If the second low-point is between *u* and *LowPt[u],* then the subtree *Tuu* is of type II; otherwise it is of type I. Now a type I subtree *Tuw* behaves exactly like a chord *(u, LowPt[w])*. There can be any number of them, and they can be nested inside each other in any order. However, they cannot be nested inside a type II subtree. Therefore we must embed all type I subtrees at *u* with *LowPt=x* before any type II subtrees at *u* with *LowPt=x*. This can be accomplished by ordering the adjacent vertices at *u* so that fronds *ux* and type I subtrees *Tuw* with *LowPt[w]= x* precede type II subtrees *Tuu* with *LowPt[u]=x*. Hopcroft and Tarjan assign a weight to all edges incident on *u*. A frond *ux* has weight 2*x*. A type I subtree *Tuu* has weight 2·*LowPt[u]*. A type II subtree *Tuu* has weight 2·*LowPt[u]*+1. The adjacent vertices are then sorted by weight, which can be done by a bucket sort in linear time, because the weights are all in the range 1,..., 2*n*+1.

The general algorithm takes place in two stages. The first stage is LOWPTDFS() which constructs a DF-tree, calculates the low-points and second low-points, and sorts the adjacency lists. The second stage is another DFS which we call EMBEDDINGDFS(). It is a DFS using the re-ordered adjacency lists and is given as Algorithm 12.15.3

Algorithm 12.15.3 constructs an embedding by placing the edges into two linked lists *Li* and *Lo,* as in the simplified algorithm. The list *Li* which originally corresponded to the chords placed inside *C,* now corresponds to the edges placed to the left of the DF-tree, since the drawings were made with the interior of *C* to the left

of the path. Similarly, the list *Lo* now corresponds to edges placed to the right of the DF-tree, since the exterior of *C* was drawn to the right of the path. EMBEDDINGDFS() first descends the DF-tree. The first leaf it encounters will have a frond back to vertex 1. This is a consequence of the ordering of the adjacency lists. This creates a cycle *C,* which we draw to the left of the path. The remaining fronds and subtrees will be placed either in *Li* or *Lo,* exactly as in the simplified algorithm.
A subtree *Tuu* is similar to a chord *(u, LowPt[u])*·*Tuu* fits in *Li* if and only if a chord *(u, LowPt[u])* does. In this case, we place a dummy chord *(u, LowPt[u])* in *Li* and a dummy chord *(u, u)* in *Lo*. A dummy chord is a chord that has an associated flag indicating that it is a placeholder for a subtree. If the dummy chord *(u, LowPt[u])* is assigned to *Li,* the subtree*Tuu* is to be embedded to the left of the path of the DF-tree containing *u*. The algorithm then calls EMBEDDINGDFS*(u)* recursively. The fronds placed in *Li* by the recursion are placed to the left of the tree *Tuu*. The fronds placed in *Lo* are to the right of *Tuu*. The dummy chord *(u, u)* in *Lo* has the purpose of ensuring that any fronds placed in *Lo* by the recursive call must have *UpperPt*≥*u*.

Page 324

Algorithm 12.15.3: EMBEDDINGDFS*(u)*

```
comment: extend the embedding DFS from u
for all u adjacent to u
do if uv is a frond and v is above u
     then if (u, v) fits in L_i
            then place (u, v) in L_i
            else if (u, v) fits in L_o
            then place (u, v) in L_o
            else m = SWITCHSIDES(u, v)
                 if m = 0
                   then NonPlanar = true
                        exit
                 else if m = 1
                   then place (u, v) in L_i
                 else if m = -1
                   then place (u, v) in L_o
     else comment: uv is a tree edge
          if u = Parent[v] then
            w = LowPt[v]
            if (u, w) fits in L_i
              then place (u, w) in L_i and (u, u) in L_o
              else if (u, w) fits in L_o
              then place (u, w) in L_o and (u, u) in L_i
              else m = SWITCHSIDES(u, w)
                   if m = 0
                     then NonPlanar = true
                          exit
                   else if m = 1
                     then place (u, w) in L_i and (u, u) in L_o
                   else if m = -1
                     then place (u, w) in L_o and (u, u) in L_i
            EMBEDDINGDFS(v)
            if NonPlanar then exit
```


Page 325

12.16 Notes

The important area of graph minors was developed in a series of over 20 papers by Robertson and Seymour. Some of their early work on graph minors is surveyed in their paper ROBERTSON and SEYMOUR [103]. They have proved a theorem of far-reaching significance, that in any infinite collection of graphs, there are always two graphs such that one is a minor of the other; or in other words, any set of graphs in which no graph is a minor of another, is finite. The books by DIESTEL [35] and ZIEGLER [129] contain excellent chapters on graph minors.
Rotation systems were developed by HEFFTER [60] and EDMONDS [37].
Read's algorithm to draw a planar graph by reducing a triangulation is from READ [100]. It was modified to use a regular polygon as the outer face by KOCAY and PANTEL [79]. Tutte's method of using barycentric coordinates to construct convex drawings of graphs appeared in TUTTE [116].
Whitney's theorem appeared in WHITNEY [125].
Good source books for polytopes are the books by GRÜNBAUM [56] and ZIEGLER [129].
The original proof of the four-color theorem appeared in APPEL and HAKEN [4] and [5]. An excellent survey article of the Appel-Haken proof is WOODALL and WILSON [128]. There are a number of excellent books on the 4-color problem, including SAATY and KAINEN [106] and ORE [93]. A very readable history of the 4-color problem can be found in WILSON [127]. A shorter proof was accomplished by ROBERTSON, SANDERS, SEYMOUR, and THOMAS in [104]. Much of the development of graph theory arose out of attempts to solve the 4-color problem. AIGNER [2] develops the theory of graphs from this perspective.
Kuratowski's theorem is a famous theorem of graph theory. It originally appeared in KURATOWSKI [82]. The proof presented here is based on a proof of KLOTZ [73]. See also THOMASSEN [112].
The Hopcroft-Tarjan planarity algorithm is from HOPCROFT and TARJAN [66]. See also WILLIAMSON [126]. It is usually presented as a "path-addition" algorithm; that is, an algorithm that embeds one path at a time across a cycle. It is presented here as an equivalent algorithm that recursively embeds a branch of the DF-tree.

Page 326
This page intentionally left blank.

Page 327
13
Graphs and Surfaces
13.1 Introduction
The plane and the sphere are the simplest topological surfaces. The structure of planar graphs, and algorithms for embedding graphs on the plane are well understood. Much less is known about graph embeddings on other surfaces, and the structure of these graphs. We begin with the torus, the doughnut-shaped surface shown in Figure 13.1. We imagine this surface made out of rubber, and using scissors, cut it along the two circumferences shown in the diagram. The surface of the torus then unfolds into a rectangle, which is indicated on the right. The opposite sides of the rectangle labeled *a* must be glued together with the arrows aligned, as must the sides labeled *b,* in order to reconstruct the torus. We could glue the edges in the order *a,* then *b;* or else *b,* then *a*. Both represent the same torus.

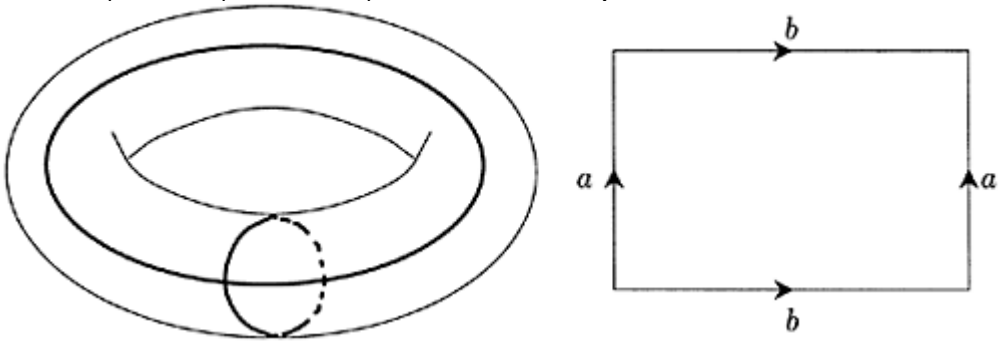


FIGURE 13.1
The torus

Page 328
When a graph is drawn on the rectangle representing the torus, we must remember that the two sides labeled *a* (and the two sides *b*) are really the same, so that graph edges can "wrap around" the diagram.

Notice that the four corners of the rectangle all represent the same point. Figure 13.2 shows two embeddings of $K3, 3$ on the torus.

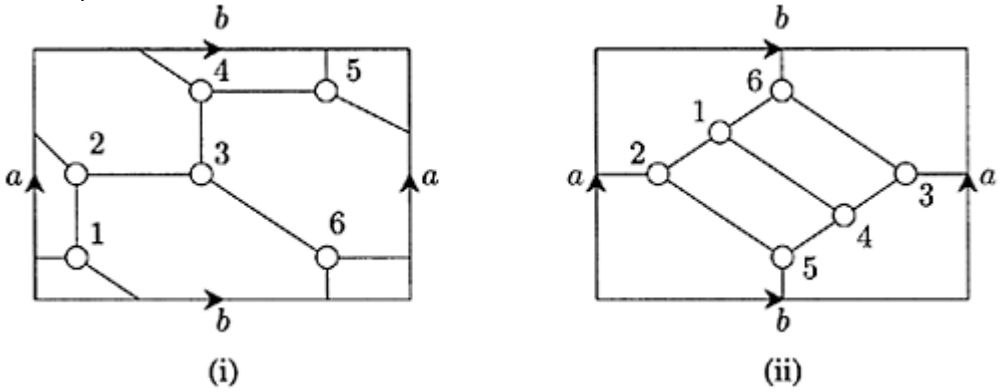


FIGURE 13.2
Two embeddings of $K3, 3$ on the torus

These embeddings of $K3, 3$ are very different from each other. Unlike the plane in which a 3-connected graph has a unique embedding (up to orientation), some graphs have very many distinct embeddings in the torus, or other surfaces.

DEFINITION 13.1: An *embedding* of a graph G in a surface Σ is a function ψ that maps the vertices of G into points of Σ, and the edges of G into continuous curves in Σ, such that the curves representing two edges intersect only at a common endpoint. We write $G\psi$ for the image of G under the embedding ψ.

In the embeddings of Figure 13.2, we can assign a "coordinate system" to the rectangle representing the torus, and then construct ψ by assigning coordinates to the vertices, and then draw the edges as straight lines. This is how the diagram was constructed.

Definition 12.1 uses an intuitive notion of a *surface,* and an intuitive notion of *continuous*. Currently we have the plane, sphere, or torus in mind. We will later make the definition of a surface more precise. Because we have coordinate systems for the above surfaces, by "continuous" we mean continuous mappings of the coordinates. However, topological continuity does not require coordinates.

If we cut the torus along the edges of the embeddings of $K3, 3$, the torus surface falls apart into several connected regions. As in the case of the plane, we call these regions the *faces* of the embedding. A *facial cycle* is an oriented cycle of the graph which bounds a face. The embedding of $K3, 3$ in Figure 13.2(i) has

three faces, each bounded by a hexagon. The embedding in Figure 13.2(ii) also has three faces, two bounded by quadrilaterals, and one bounded by a 10-cycle.

An *open disc* in the plane is the region interior to a circle. We will use an intuitive understanding of the notion of *homeomorphic* subsets of surfaces. Two regions $R1$ and $R2$ are said to be homeomorphic if one can be transformed into the other by a one-to-one continuous deformation, whose inverse is also continuous. For example, an open disc is homeomorphic to a face bounded by a hexagon, or by any other polygon. A *homeomorphism* of $R1$ and $R2$ is any continuous one-to-one mapping from $R1$ onto $R2$, whose inverse is also continuous.

DEFINITION 13.2: A *2-cell* is a region homeomorphic to an open disc. An embedding $G\psi$ is a *2-cell embedding* if all faces are 2-cells.

Corresponding to the idea of a 2-cell are 0-cells (a single point), 1-cells (an open line segment), and 3-cells (the interior of a sphere in 3-space), etc.

It is necessary to restrict embeddings of a graph G to 2-cell embeddings. For example, we could draw a planar embedding of a planar graph G, such as the cube, in the rectangle representing the torus. The result would not be a 2-cell embedding of G, for the outer face of the embedding would be homeomorphic to a torus with a hole in it, which is not a 2-cell. Embeddings which are not 2-cell embeddings really belong in a different surface. For the cube, there are five distinct 2-cell embeddings on the torus.

DEFINITION 13.3: Two embeddings $G\psi 1$ and $G\psi 2$ in a surface Σ are *homeomorphic* if there is a homeomorphism of Σ which maps $G\psi 1$ to $G\psi 2$. Otherwise $G\psi 1$ and $G\psi 2$ are *distinct embeddings*.

It is clear that a homeomorphism of $G\psi 1$ and $G\psi 2$ induces an automorphism of G, and that the faces of $G\psi 1$ map to the faces of $G\psi 2$. The embeddings of Figure 13.2 are then easily seen to be distinct, because their facial cycles have different lengths. In general, there is no easy method of determining all the embeddings of a graph on a given surface, or even to determine whether a graph is embeddable.

13.2 Surfaces

We can use the method of representing the torus as a rectangle, as in Figure 13.1, to represent a *cylinder,*

and a *Möbius band,* shown in Figures 13.3 and 13.4.
The cylinder is glued along only one edge of the rectangle. We call it an *open surface* because it has a boundary (the edges which are not glued). If we project a cylinder onto a plane, one possible result is an annulus, that is, a disc with a hole in it. This immediately gives the following theorem:

Page 330

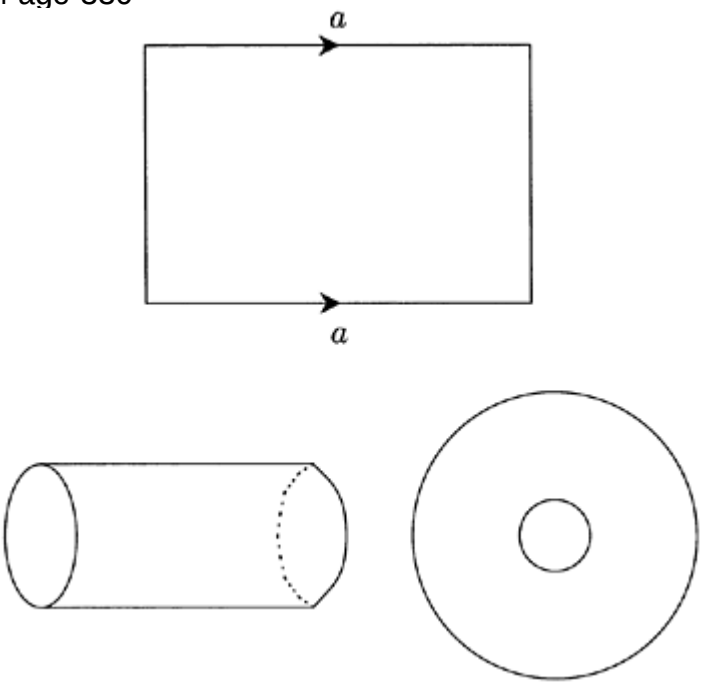
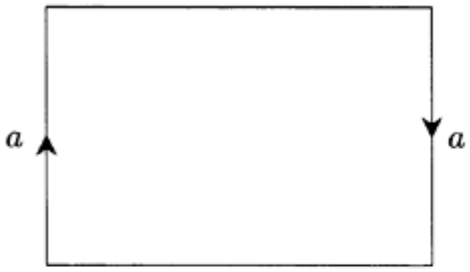


FIGURE 13.3
Three representations of the cylinder

THEOREM 13.1 A graph can be embedded on the cylinder if and only if it can be embedded on the plane.

PROOF Given an embedding of a graph *G* on the plane, choose any face and cut a hole in it. The result is an embedding on the cylinder, and vice versa.
Notice that an embedding of *G* on the cylinder corresponds to an embedding of *G* on the plane with two distinguished faces. They can be any two faces of a planar embedding of *G*.
The Möbius band is constructed by giving one end of the rectangle a twist of 180 degrees before aligning and gluing the opposite edges. This is indicated by the opposite orientation of the arrows. Notice that if we follow the boundary of the Möbius band, it is a single closed curve, unlike the boundary of the cylinder.
The sphere, torus, and cylinder can all be considered as *two-sided* surfaces—they have an inside and outside. One way to define this is to imagine a small clockwise-oriented circle drawn in the surface. If we reflect this circle in the

Page 331



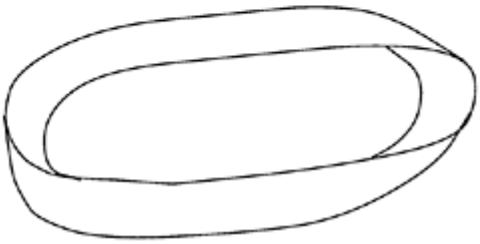


FIGURE 13.4
The Möbius band

surface, we obtain a circle of the opposite orientation. On the sphere, torus, and cylinder it is not possible to walk along the surface, taking the oriented circle with us, until it coincides with its opposite orientation (we are not allowed to walk over the boundary of the cylinder or Möbius band). On the Möbius band, it is possible to make these two circles coincide. We therefore say that the Möbius band is a *one-sided* surface. A two-sided surface is said to be *orientable*. We can assign an orientation to the surface, by partitioning the set of oriented circles defined at every point of the surface. One orientation is called the inside, and the other the outside. A one-sided surface is *non-orientable,* as the set of oriented circles does not have this partition into two subsets. For surfaces like the sphere and torus which can be constructed in euclidean 3-space, we could alternatively use a normal vector to the surface, and its reflexion in the surface in place of an oriented circle and its reflexion.

Aside from the cylinder and Möbius band, the surfaces we will be interested in are closed surfaces.

A *closed surface* is a generalized notion of a *polyhedron*. A *polyhedron* is a

---

three-dimensional object consisting of a set of polygons, of three or more sides each. Each polygon is bounded by a sequence of p straight-line segments connecting p vertices in cyclic order, for some $p \geq 3$. The line segments are the edges of the polyhedron. Each edge is shared by exactly two polygons. Any two polygons may intersect only on a single common edge. There are at least three polygons meeting at each vertex, and the polygons meeting at any vertex form a single cycle.

This idea is generalized by allowing the polygons composing a polyhedron to be infinitely stretchable and interpenetrable. They are then called *curvilinear polygons*.

DEFINITION 13.4: A *closed surface* is a set of points homeomorphic to a polyhedron made of curvilinear polygons.

Thus a closed surface is an object that is capable of being represented as a collection of curvilinear polygons glued together along common edges. However, the surface is not any single one of these representations, since many different polygonal representations of the same closed surface are possible. For example, the surface of the sphere may be partitioned into many different polygonal forms. Similarly, the embeddings of $K3$, 3 of Figure 13.2 partition the torus into two different polyhedra.

When we identify opposite edges of a rectangle, we are identifying two edges of a single polygon. In order to conform to the definition of polyhedron, in which only distinct polygons share a common edge, we can subdivide the rectangle into two or more polygons, as necessary. There are three more surfaces that can be made from a rectangle by identifying its edges in pairs. They are the sphere, projective plane, and Klein bottle, illustrated in Figures 13.5, 13.6, and 13.8. The sphere can also be represented as a *digon* (a polygon of two sides), as shown, where $c = ab$.

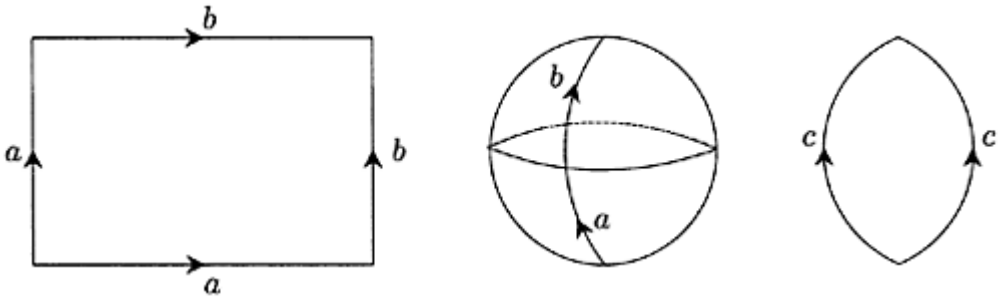


FIGURE 13.5
The sphere as a rectangle and as a digon

The projective plane is constructed from a rectangle by first using a twist to

create a Möbius band, and then by gluing the boundary of the Möbius band to itself, to create a closed surface. This is illustrated in Figure 13.6, where the edges labeled *b* have been glued. This cannot be done in euclidean space, for the polygon must cross itself without intersecting itself. But mathematically, we can consider the surface to be constructed in this way. The projective plane can also be viewed as a digon, as illustrated in Figure 13.7, by combining the *a* and *b* sides of the rectangle into a single side, labeled *c*. Because of the orientation of the arrows, the two "corners" of the digon represent the same point. We can identify the corners, creating a "figure eight", and then identify the two lobes of the figure eight to complete the projective plane. If we then remove a disc from the projective plane by cutting along the circle containing the dotted line, the result is called a *crosscap*. It is a projective plane with a hole in it. In Exercise 13.2.1, it is proved that a crosscap is homeomorphic to a Möbius band.

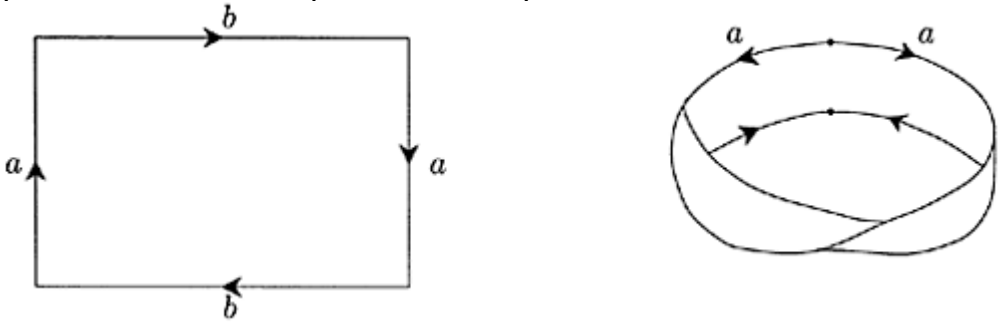


FIGURE 13.6
The projective plane as a rectangle, and as a Möbius band

The Klein bottle can be constructed by gluing two edges of a rectangle to create a cylinder, and then by gluing the ends of the cylinder together, according to the orientation of the arrows. This also cannot be done in euclidean space, as the cylinder must cross through itself without intersecting itself.

These surfaces have been constructed from a rectangle by gluing edges together. By subdividing the rectangle into curvilinear polygons, closed surfaces represented as curvilinear polyhedra are obtained. Polygons with more sides than a rectangle could also be used. By the classification theorem of closed surfaces (Theorem 13.2), every closed surface can be constructed by gluing together edges of a single curvilinear polygon.

The rectangle representing the torus in Figure 13.1 can be written symbolically as $a+b+a-b-$. This means that we choose a clockwise orientation of the rectangle and write $a+$ for the edge labeled *a* when the direction of the arrow is clockwise, and $a-$ when the direction of the arrow is counterclockwise. The boundary of the rectangle is then determined by the above symbol. Similarly the rectangle

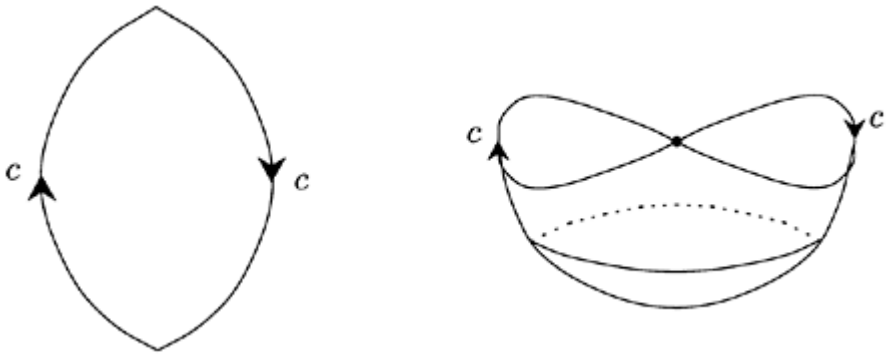


FIGURE 13.7
The projective plane as a digon, and as a crosscap

representing the sphere (Figure 13.5) can be characterized as $a+b+b-a-$, that of the projective plane (Figure 13.6) as $a+b+a+b+$, and that of the Klein bottle (Figure 13.8) as $a+b+a+b-$.

In general, we have a polygon with an even number of sides, with labels $a1$, $a2$,..., ap, such that every label ai appears on exactly two sides. We place an arrow on each edge of the polygon in some direction, and choose a clockwise orientation of the polygon. This defines a symbolic representation in terms of the a_i^+ and a_i^-. Every closed surface can be represented symbolically by a *normal form* of this type, as shown by the following theorem:

THEOREM 13.2 **(Dehn and Heegard—Normal forms for closed surfaces)** *Every closed surface can be represented symbolically by one of the following normal forms. Two closed surfaces are homeomorphic if and only if they have the same normal form.*

1. $a+a-$
2. $a_1^+b_1^+a_1^-b_1^-a_2^+b_2^+a_2^-b_2^- \dots a_p^+b_p^+a_p^-b_p^-$
3. $a_1^+a_1^+a_2^+a_2^+ \dots a_q^+a_q^+$

The proof of this theorem can be found in FRÉCHET and FAN [44] or STILLWELL [109]. It is too lengthy to include here. It involves cutting and pasting a curvilinear polygon until the normal form is achieved. This is done in stages. The torus is represented in Figure 13.1 by $a+b+a-b-$, which is already in normal form. The sphere is represented by $a+b+b-a-$. It is clear from the diagram that the adjacent $b+b-$ corresponds to edges that can be glued, so that they cancel from the formula, leaving $a+a-$ as the normal form for the sphere. The projective

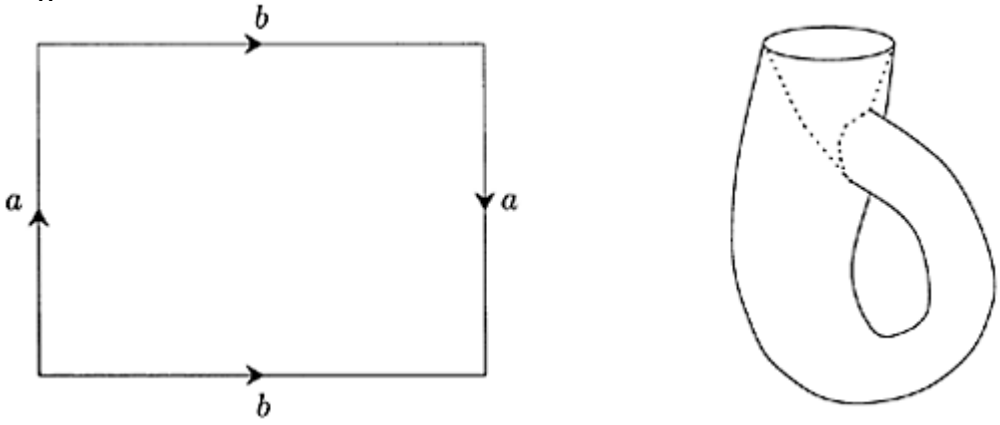


FIGURE 13.8
The Klein bottle as a rectangle

plane is represented in Figure 13.6 as $a+b+a+b+$, which is not in normal form. The proof of Theorem 13.2 would convert it to normal form by the following sequence of operations, illustrated in Figure 13.9.

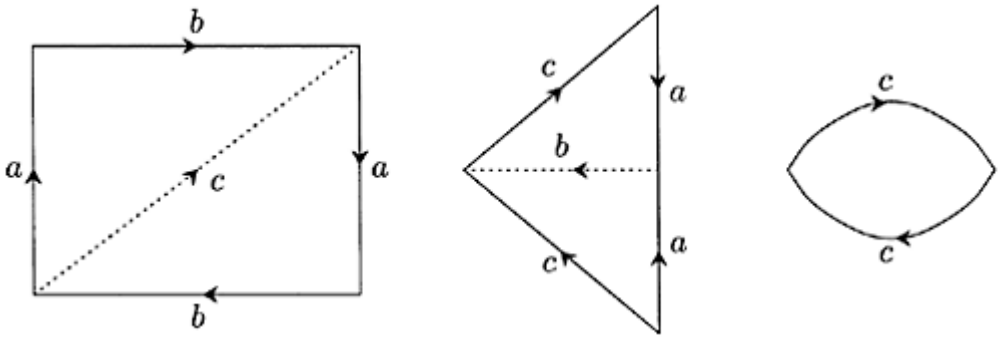


FIGURE 13.9
Transforming the projective plane to normal form
We make a diagonal cut across the rectangle, and label it *c*. We thereby obtain two triangles which we glue together along the edge *b,* which then disappears. The symbolic form is now $c+a+a-c+$. The edges $a+a-$ are then glued to produce the digon in Figure 13.9, giving the normal form $c+c+$ for the projective plane.

13.2.1 Handles and crosscaps

Consider a surface with normal form $a+b+a-b-c+d+c-d-$, shown in Figure 13.10. We have an octagon with edges that are to be identified in pairs. Both endpoints of the upper edge marked *b* represent the same point, as they are the same endpoint of the arrows marked *a*. Therefore gluing the edges labeled *a* will make this *b*-edge the boundary of a hole in the surface. Consequently, the other edge labeled *b* also represents the boundary of a hole in the surface, and these two boundaries must be identified. One way to identify them is to attach the two ends of a cylindrical tube to each of these holes. The result is a *handle* attached to the surface.

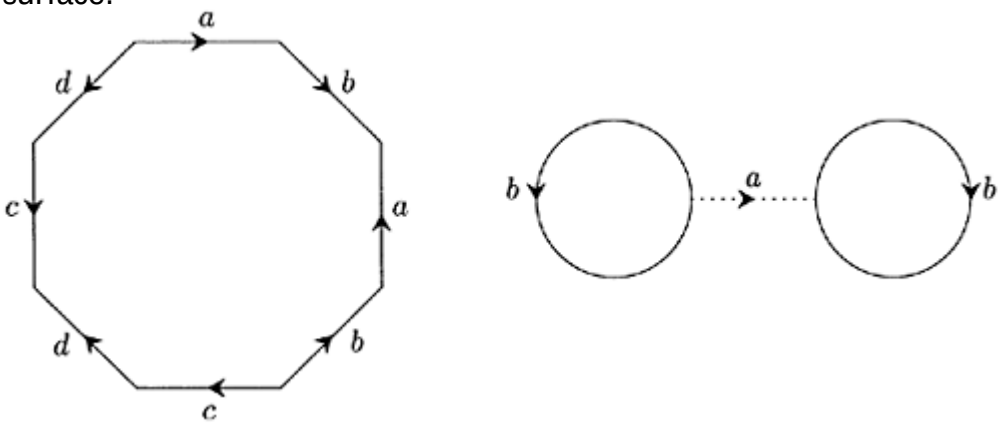


FIGURE 13.10
A sphere with two handles
Now the same can be done for the edges marked *c* in the diagram—we attach another handle. This same argument holds for any normal form of this type. This gives:
THEOREM 13.3 A surface with normal form

$$a_1^+ b_1^+ a_1^- b_1^- a_2^+ b_2^+ a_2^- b_2^- \dots a_p^+ b_p^+ a_p^- b_p^-$$

is homeomorphic to a sphere with p handles.
Because a sphere is an orientable surface, so is a sphere with *p* handles.
Consider now a surface with normal form $a+a+b+b+c+c+$, illustrated in Figure 13.11. We have a hexagon in which consecutive edges are to be identified in pairs. The vertex common to the two sides marked *a* is both the head and tail of the a-arrow. Therefore indentifying the endpoints of the edges marked *a*

makes two holes in the surface, bounded by the *a*-edges. We must identify the boundaries of these two lobes.

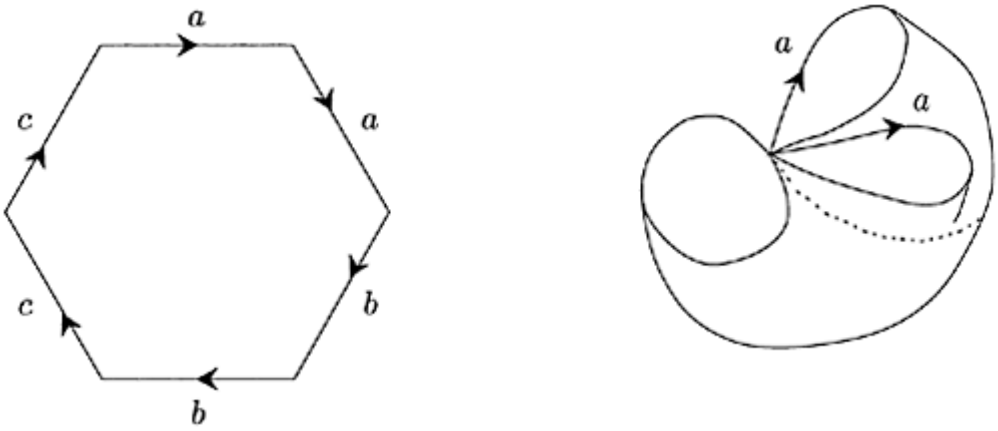


FIGURE 13.11
A sphere with three crosscaps

This is nearly identical to the construction of the projective plane from a digon, illustrated in Figure 13.7. If we draw a circle around the two *a*-lobes in Figure 13.11, and cut along the dotted line, we see that what we have is in fact a *crosscap* glued into a hole cut in the surface. We can then do the same for the remaining pairs $b+b+$ and $c+c+$ to get a sphere with three crosscaps. In general, this gives:

THEOREM 13.4 A surface with normal form

$$a_1^+ a_1^+ a_2^+ a_2^+ \dots a_q^+ a_q^+$$

is homeomorphic to a sphere with q crosscaps.

Because a crosscap is an unorientable surface, so is a sphere with q crosscaps. Gluing a crosscap to a hole in a sphere is equivalent to gluing the boundary of a Möbius band to a hole in the sphere.

13.2.2 The Euler characteristic and genus of a surface

We know from Chapter 12 that a connected planar graph with n vertices, ε edges, and f faces satisfies Euler's formula $n-\varepsilon+f=2$. Furthermore, the skeleton of a polyhedron is a planar graph, so that any polyhedral division of the sphere also satisfies this formula. We say that the *Euler characteristic* of the sphere is 2.

DEFINITION 13.5: Let Σ be a closed surface represented by a curvilinear polyhedron with n vertices, f polygons, and ε edges. The *Euler characteristic* of the surface is the value $n-\varepsilon+f$. It is denoted $\chi(\Sigma)$.

It has been proved by Kerékjártó that the value $n-\varepsilon+f$ for a surface is invariant, no matter what polygonal subdivision is used to represent it. This is difficult to prove because of the vast numbers of polygonal subdivisions that are possible. However, we can get an understanding of it as follows. If we add a diagonal across a face, n does not change, but ε and f both increase by one. Thus $n-\varepsilon+f$ does not change. Similarly, if we subdivide an edge with a new vertex, n and ε increase by one, but f does not change. Modifying a polygonal subdivision by these operations does not change the value $n-\varepsilon+f$. Suppose that we now add a handle or crosscap to a given polygonal division of a surface. Consider a polygonal division with n vertices, f polygons, and ε edges. Choose two polygons on the surface, cut a disc from the interior of each polygon, and attach a handle connecting them. Let the polygons be $P1$ and $P2$. Refer to Figure 13.12. We draw two curves along the handle connecting $P1$ to $P2$. The result is a polygonal division of the surface with an additional handle. Let it have n' vertices, f' polygons, and ε' edges. The effect of drawing the curves connecting $P1$ to $P2$ is to add two vertices and two edges to each of $P1$ and $P2$, plus two additional edges represented by the curves. The number of polygons does not change. We therefore have $n'=n+4$, $f'=f$, and $\varepsilon'=\varepsilon+6$, so that $n'-\varepsilon'+f'=(n-\varepsilon+f)-2$. Thus, when a handle is added to a surface, the Euler characteristic decreases by two. It does not matter to which faces the handle attaches. It follows that a sphere with p handles has Euler characteristic $2-2p$.

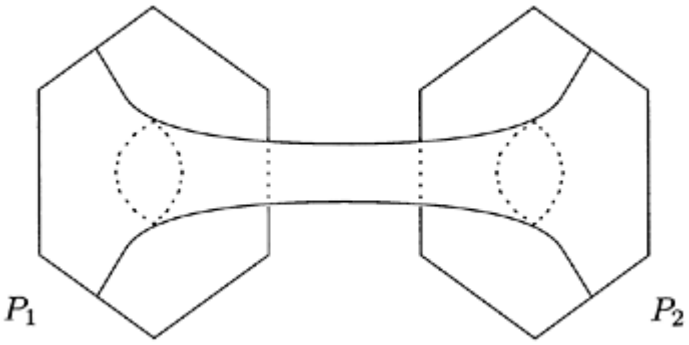


FIGURE 13.12
Attaching a handle to a surface

Page 339
Suppose that we now add a crosscap to a polygonal division of a surface (e.g., the sphere). We choose a polygon $P1$, cut a disc from its interior, and attach a crosscap. Let C be the boundary of the disc. Now a crosscap is a Möbius band, and the boundary of the disc becomes the boundary of the Möbius band. We draw a curve connecting $P1$ to C, continue across the Möbius band to the opposite side of C, and continue to an opposite point of $P1$. The result is a polygonal division of the surface with an additional crosscap. Let it have n' vertices, f' polygons, and ε' edges. The effect of drawing the curve across $P1$ is to add two vertices and two edges to $P1$, plus an additional edge represented by the curve. When the Möbius band is cut in half, it remains connected. Therefore the number of polygons does not change. We therefore have $n'=n+2$, $f'=f$, and $\varepsilon'=\varepsilon+3$, so that $n'-\varepsilon'+f'=(n-\varepsilon+f)-1$. Thus, when a crosscap is added to a surface, the Euler characteristic decreases by one. It does not matter to which face the crosscap attaches. It follows that a sphere with q crosscaps has Euler characteristic $2-q$.
Consequently surfaces can be classified according to whether they are orientable or non-orientable, and their Euler characteristic. A related parameter of a surface is its *genus*.
DEFINITION 13.6: A Jordan curve in a surface Σ is *contractible* or *null-homotopic* if it can be continuously shrunk to a point within Σ.
Cutting a surface along a contractible Jordan curve always separates it into two pieces.
DEFINITION 13.7: The *genus* of a surface is the maximum number of Jordan curves that can be drawn on the surface such that cutting along the curves does not separate it into two or more pieces.
It is easy to see that the sphere has genus zero, and that the torus has genus one. In general, a sphere with p handles has genus p, as exactly one Jordan curve can be drawn around each handle without separating the surface. Because a sphere with handles is an orientable surface, we say it has *orientable genus p*.
The projective plane has genus one, as shown in Exercise 13.2.2. It then follows from Exercise 13.2.3 that a sphere with q crosscaps has genus q. We say it has *unorientable genus q*. Some texts use the term *crosscap number* in place of genus for a non-orientable surface. The relation with Euler characteristic can now be stated.
THEOREM 13.5 An orientable surface of genus p has Euler characteristic $2-2p$. *A non-orientable surface of genus q has Euler characteristic* $2-q$.
When a graph G is embedded in a surface Σ, cycles of G map to Jordan curves in Σ. We will be interested in cycles which are embedded as non-contractible curves.

Page 340
DEFINITION 13.8: Let $G\psi$ be an embedding of a graph G in a surface Σ. A cycle C in G is an *essential* cycle, or *non-contractible* cycle of the embedding if $G\psi$ is not contractible.
For example, in the embedding of $K3, 3$ on the left in Figure 13.2, the cycles (1, 2, 3, 4) and (2, 3, 4, 5) are essential cycles, while (1, 2, 3, 6, 5, 4) is not.
DEFINITION 13.9: The *genus* of a graph G is $g(G)$, the smallest genus of an orientable surface Σ such that G has a 2-cell embedding in Σ. The *unorientable genus* or *crosscap number* of G is $\overline{g}(G)$, the smallest genus of a non-orientable surface Σ such that G has a 2-cell embedding in Σ.
Suppose that $G\psi$ is a 2-cell embedding of G in an orientable surface Σ of genus p. Let $G\psi$ have n vertices, ε edges, and f faces. Then since the faces of $G\psi$ determine a polygonal division of the surface, we have $n-\varepsilon+f=\chi(\Sigma)=2-2p$. This is called the *Euler-Poincaré formula*. If G is embedded on an unorientable surface of genus q, then $n-\varepsilon+f=\chi(\Sigma)=2-q$. This gives the following relations for graphs embedded on the:
plane: $n-\varepsilon+f=2$

torus: $n-\varepsilon+f=0$
projective plane: $n-\varepsilon+f=1$.
LEMMA 13.6 A triangulation of an orientable surface of genus p, with n vertices satisfies $\varepsilon=3n+6(p-1)$. *A triangulation of a non-orientable surface of genus q satisfies* $\varepsilon=3n+3(q-2)$.
PROOF A triangulation satisfies $3f=2\varepsilon$. Combining this with $n-\varepsilon+f=\chi(\Sigma)$ gives the result.
Exercises
13.2.1 Show that if a disc is removed from a projective plane, the result is a Möbius band.
13.2.2 Show that if a the projective plane is cut along a non-separating Jordan curve, the result is a disc.
13.2.3 Show that exactly one Jordan curve can be drawn on the Möbius band without separating it. What is the result of cutting the Möbius band along a non-separating Jordan curve?
13.2.4 Use cutting and pasting to convert the representation $a+b+a+b-$ of the Klein bottle to normal form $c+c+d+d+$.
13.2.5 A sphere with one handle and one crosscap can be represented symbolically by $a+b+a-b-c+c+$. Find the normal form for this surface.

13.2.6 Find the facial cycles of the embeddings of $K3, 3$ on the torus shown in Figure 13.2.
13.2.7 Use Lemma 13.6 to obtain a lower bound on $g(G)$ and $\overline{g}(G)$ for an arbitrary graph G.
13.2.8 Show that $g(Kn)\geq(n-3)(n-4)/12$ and that $\overline{g}(K_n) \geq (n-3)(n-4)/6$.
13.2.9 Let G be a graph with no triangles embedded on a surface of genus g. Find an upper bound on the number of edges of G.
13.3 Graph embeddings, obstructions
Three of the main algorithmic problems of graph embeddings are:
Problem 13.1: Graph Embeddability
Instance: a graph G and a surface Σ.
Question: is G embeddable in Σ?
Problem 13.2: Graph Embeddings
Instance: a graph G and a surface Σ.
Question: find all embeddings of G in Σ.
Problem 13.3: Graph Genus
Instance: a graph G and an integer k.
Question: is $g(G)\leq k$?
It was proved by THOMASSEN [113] that Graph Genus is NP-complete.
The first two problems are solved for the plane, but only partially solved for other surfaces. Several efficient algorithms are known for Graph Embeddability on the projective plane. For the plane, Kuratowski's theorem tells us that G is embeddable if and only if it has no subgraph $TK5$ or $TK3, 3$. We call these graphs *obstructions* to planarity.
DEFINITION 13.10: Given a surface Σ, a *topological obstruction* for Σ is a minimal graph K with $\delta(K)\geq 3$ such that any graph containing a subdivision TK cannot be embedded in Σ.
There are two topological obstructions for the plane, $K5$ and $K3, 3$. The definition requires that K be *minimal,* namely, that no proper subgraph of K has this

property (otherwise $K6$, $K7$, etc., would all be considered as topological obstructions).
DEFINITION 13.11: Given a surface Σ, a *minor-order obstruction* (or *minimal forbidden minor* or *excluded minor*) for Σ is a graph K, such that any graph having K as a minor cannot be embedded in Σ, but no proper minor of K has this property.
Recall that the graph relation "H is a minor of G" forms a partial order on the set of all graphs. If we restrict the set to all graphs which are not embeddable in the surface Σ, then the minimal graphs of this partial order are the minor-order obstructions. If K is a minor-order obstruction, then it is also a topological obstruction. $K5$ and $K3, 3$ are both minor-order obstructions and topological obstructions for the plane, since any graph which has $K5$ or $K3, 3$ as a minor necessarily contains either a $TK5$ or $TK3, 3$. For other surfaces, there is a distinction between the two concepts of topological obstruction and minor-order obstruction.
Robertson and Seymour have proved that there are a finite number of obstructions for any surface, as a consequence of the *graph minor theorem,* which we state without proof.
THEOREM 13.7 **(Graph minor theorem)** *In any infinite collection of graphs, there are always two graphs such that one is a minor of the other*.

It is known that there are 103 topological obstructions for the projective plane, of which 35 are minor-order obstructions, as found by ARCHDEACON [6]. These are often called *Kuratowski subgraphs* for the projective plane. A list of them can be found in MOHAR and THOMASSEN [88]. For the torus, the number of obstructions is in the hundreds of thousands, as shown by MYRVOLD [90]. From an algorithmic point of view, this is not an effective characterization, as there are too many obstructions.

13.4 Graphs on the torus

Given a 2-cell embedding ψ of a 2-connected graph G on the torus, there must be an essential cycle C in G. Cutting the torus along $C\psi$ results in a cylinder. Since the cylinder is not a 2-cell, but the embedding is a 2-cell embedding, there must be another essential cycle C' in G, cutting the cylinder along an axis. Consequently C and C' must intersect, either in a path or a vertex.

DEFINITION 13.12: A *theta-graph* is a graph consisting of two vertices of degree three, connected by three paths of one or more edges each.

A theta-graph is illustrated in Figure 13.13.

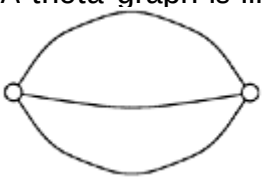


FIGURE 13.13
A theta-graph in schematic form

Thus, $C \cup C'$ must be either a theta-subgraph of G, or two cycles with a vertex in common, and ψ is a 2-cell embedding of it. The simplest form of theta-subgraph is a *multigraph* consisting of two vertices connected by three parallel edges. A 2-cell embedding of it is shown in Figure 13.14. It is often necessary to consider embeddings of graphs with multiple edges and/or loops, as the duals of many graph embeddings (e.g., $K4$, $K5$, $K3$, 3) often have multiple edges, and sometimes loops. We shall always insist that in any embedding of a multigraph:

1. The cycle induced by any loop is an essential cycle (no face is a loop).
2. The cycle induced by any digon is an essential cycle (no face is a digon).

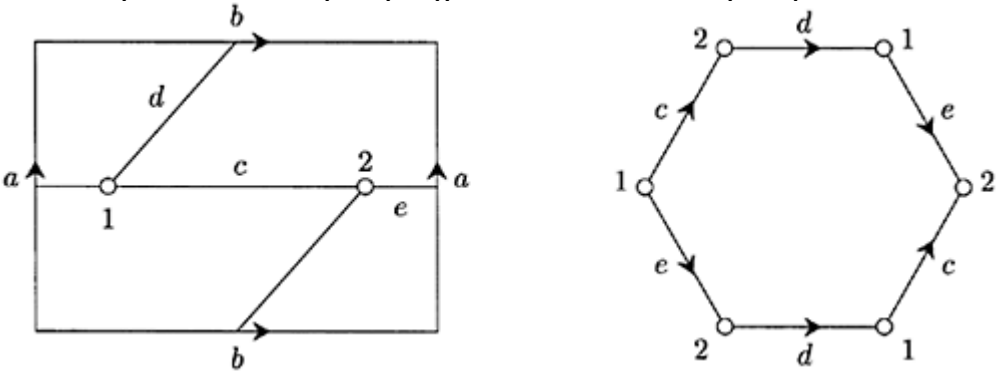


FIGURE 13.14
A 2-cell embedding of a theta-graph

If we cut the torus along the edges of this theta-graph, we find there is one

face, a *hexagon,* shown in Figure 13.14 as $c+d+e+c-d-e-$. Thus we see that *the torus can also be represented as a hexagon*. The hexagonal form corresponds to an embedding of a theta-subgraph. The rectangular form corresponds to an embedding of two cycles with a common vertex.

Given an embedding $G\psi$ on the torus, we choose an orientation of the torus, and walk around a vertex $u\psi$ in a small clockwise circle in the surface, and construct the cyclic adjacency list, just as for embeddings in the plane (Section 12.5). This determines a *rotation system* for G, exactly as in the planar case. We will denote a rotation system for a graph embedded on the torus by t. The faces of the embedding are completely determined by t, since Algorithm 12.5.1, FACIALCYCLE(), to find the facial cycles of a planar graph from its rotation system also applies to toroidal rotation systems, or to rotation systems for any orientable surface. Similarly, algorithm CONSTRUCTDUAL() applies equally to toroidal rotation systems as well as other orientable surfaces. Hence we denote a combinatorial toroidal embedding by Gt and its dual by Gt^*. Now the rotation

system determines the faces of the embedding. Hence, we can determine from *t* whether or not *Gt* has any faces which are digons or loops, but it cannot determine whether any digons or loops are embedded as essential cycles.
It is convenient to refer to a graph embedded on the torus as a torus *map*.
DEFINITION 13.13: A *torus map* is a combinatorial 2-cell embedding *Gt,* where *G* is a 2-connected graph.
We begin by embedding planar graphs on the torus. Let *G* be a 2-connected planar graph that is not a cycle, with a planar rotation system *p*. By Exercise 13.4.3, *G* has a theta-subgraph *H*. Let *u* and *u* be the two vertices with degree three in *H,* and let *P*1, *P*2, *P*3 be the three uu-paths of *H*. Let *w*1, *w*2, and *w*3 be the first vertices of *P*1, *P*2, *P*3, respectively, adjacent to *u*. Refer to Figure 13.15.
THEOREM 13.8 Let p be a planar rotation system for G, and let H be a thetasubgraph of G, as described above. Let t be a rotation system constructed from p by interchanging w2 and w3 in the cyclic adjacency list of u, and leaving all other vertices the same. Then t is a toroidal rotation system for G.
PROOF In the embedding *Gp* in the plane, the three paths *P*1, *P*2, *P*3 divide the plane into three regions. Without loss of generality, let the paths occur in the order illustrated in Figure 13.15. Denote the subgraphs of *G* contained within the three regions as *G*12 (between paths *P*1 and *P*2), *G*23, and *G*31. Subgraph *Gij* may have edges connecting it only to *Pi* and *Pj,* and to *u* and *u*. Construct the hexagonal representation of the torus with *P*1, *P*2, and *P*3 on the boundary of the hexagon, and embed *G*12, *G*23, and *G*31 inside the hexagon as planar embeddings, as shown, resulting in a toroidal embedding of *G*. It is easy to verify from the diagram that any vertex in *G*12, *G*23, and *G*31 has the same cyclic adjacency list

in the toroidal embedding as in the planar embedding. Similarly any vertex other than *u* or *u* of any *Pi* has the same cyclic adjacencies in both embeddings. The same is also true for *u*. The only vertex whose cyclic adjacencies differ is *u*. The adjacency list of u has been arranged so that $w_1 \in P_1$is followed by the edges to *G*12, followed by $w_3 \in P_3$, followed by the edges to *G*23, followed by $w_2 \in P_2$, followed by the edges to *G*31. The only difference to the planar adjacencies is that *w*2 and *w*3 have been interchanged for vertex *u*.
It is evident from Figure 13.15 that there are several other ways to convert the planar rotation system to a toroidal rotation system.

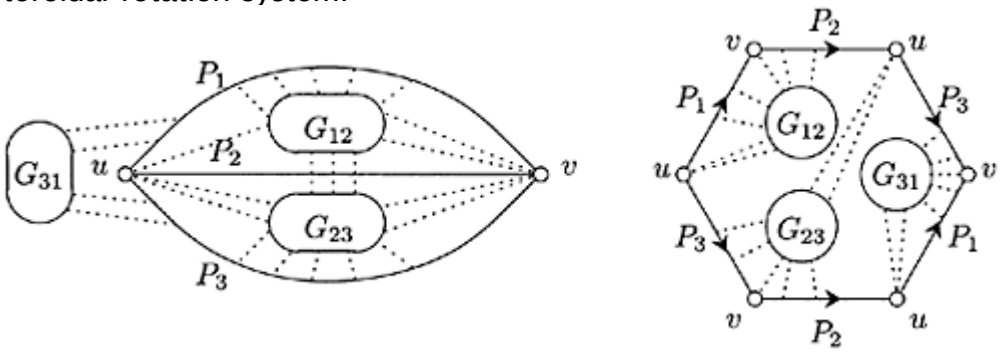


FIGURE 13.15
Constructing a toroidal rotation system
A planar graph also has non-2-cell embeddings on the torus. If a planar graph *G* is embedded in a disc on the surface of the torus, we will call this a *disc embedding* of *G*. If a planar graph is embedded such that one of the faces is homeomorphic to a cylinder, we will call this a *cylindrical embedding* of *G*. A cylindrical embedding on the torus of any graph *G* determines an embedding on the cylinder, so that *G* must be planar.
LEMMA 13.9 Every embedding of a non-planar graph on the torus is a 2-cell embedding.
PROOF A non-2-cell embedding would necessarily be either a disc or cylindrical embedding. But a non-planar graph has no disc or cylindrical embedding.
Currently, there is no satisfactory algorithm known to determine whether an arbitrary graph can be embedded on the torus, or to find all embeddings, or to

characterize all possible embeddings. Whitney's theorem (12.20) on induced non-separating cycles does not apply to embeddings on the torus. There are several simple techniques that are useful in an exhaustive search to find the embeddings. Given two combinatorial embeddings with toroidal rotation systems *t*1 and *t*2, we need to distinguish whether G^{t_1} and G^{t_2} are equivalent embeddings. In general, two embeddings are considered equivalent if they have the same facial cycles, as the faces can be glued together along the facial

boundaries in a unique way to construct the torus. Since the facial cycles are completely determined by the rotation system, we define equivalence in terms of rotation systems. Definitions 12.16 and 12.17 of equivalent embeddings and graph orientability apply equally well to toroidal graphs as to planar graphs, using a toroidal rotation system *t* in place of a planar rotation system *p*. We summarize the definitions as follows:

DEFINITION 13.14:

1. Embeddings G^{ψ_1} and G^{ψ_2} are *homeomorphic embeddings* if there is a homeomorphism of the torus mapping G^{ψ_1} to G^{ψ_2}. Otherwise they are distinct.

2. Embeddings G^{t_1} and G^{t_2} are *isomorphic* if there is an automorphism of *G* which induces a mapping of *t*1 to *t*2.

3. Embeddings G^{t_1} and G^{t_2} are *equivalent embeddings* if there is an automorphism of *G* which induces a mapping of *t*1 to *t*2 or $\bar{t}_2$, where $\bar{t}_2$ is obtained by reversing the cycles of *t*2.

4. Embedding *Gt* is a *non-orientable embedding* if there is an automorphism of *G* inducing a mapping of *t* to $\bar{t}$. Otherwise it is an *orientable embedding*.

Now an embedding G^{ψ_1} determines a rotation system *t*1. Homeomorphic embeddings G^{ψ_1} and G^{ψ_2}determine equivalent combinatorial embeddings G^{t_1} and G^{t_2}, since a homeomorphism can be either orientation preserving or orientation reversing. Conversely, if G^{t_1} and G^{t_2} are equivalent combinatorial embeddings of *G,* then they have the same facial cycles (up to orientation). The facial cycles can be glued together to construct a curvilinear polyhedron representing the torus. Therefore, topological embeddings G^{ψ_1} and G^{ψ_2} can be constructed from G^{t_1} and G^{t_2}, so that G^{ψ_1} and G^{ψ_2} are homeomorphic. This gives:

THEOREM 13.10 Topological embeddings $G^{\psi_1}$and $G^{\psi_2}$are homeomorphic if and only if the corresponding combinatorial embeddings $G^{t_1}$and $G^{t_2}$are equivalent.

Now the homeomorphism between G^{ψ_1} and G^{ψ_2} was constructed by gluing curvilinear polygons (the faces of the embeddings) together along common edges; that is, it involves cutting and pasting the torus. For example, the two embeddings

G^{t_1} and G^{t_2} shown in Figure 13.16 are equivalent, which can easily be verified from the rotation systems. However, they are homeomorphic only by cutting the torus along a non-contractible cycle to create a cylinder, then twisting one end of the cylinder by 360 degrees, and then re-gluing the cylinder to create a torus. It is not possible to transform one into the other without cutting and pasting.

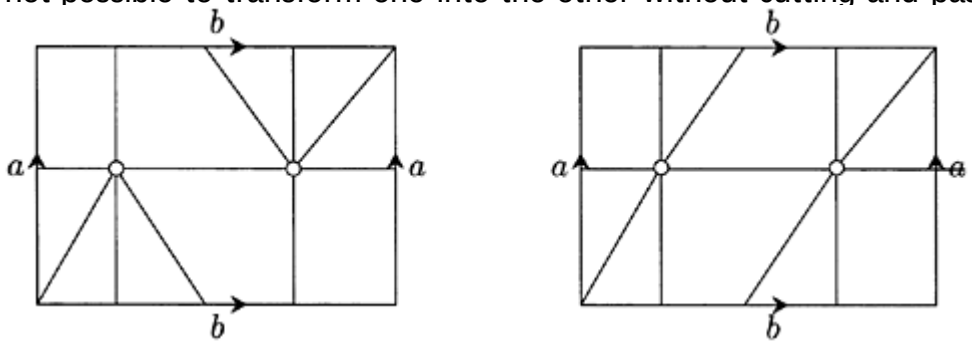


FIGURE 13.16
Two equivalent embeddings

For graphs on a small number of vertices, it is possible to distinguish inequivalent embeddings by inspection. However, even for *K*5 and *K*6, it is reasonably difficult to determine the inequivalent embeddings by hand. One technique that helps is the dual graph—if G^{t_1} and G^{t_2} have non-isomorphic dual graphs, then the embeddings are distinct. More generally, we can use the *medial digraph* (Definition 12.19) to distinguish embeddings and orientations. It can also be used to determine the symmetries (automorphisms) of an embedding. The medial digraph was defined for planar rotation systems, but the definition is also valid for toroidal rotation systems. We use *M(Gt)* to denote the medial digraph of *G* with a toroidal rotation system *t*. The medial digraph was defined for multigraphs. If we want to allow for loops as well, then the definition must be modified slightly (Exercise 13.4.7). Usually graph isomorphism software is necessary to make effective use of the medial digraph.

Suppose that *G* is a 2-connected planar graph. Choose a theta-subgraph *H* of *G*. We would like *H* to have as many edges as reasonably possible. We can do this by hand for small graphs. With larger graphs, a depth-

first search can be used to find a theta-subgraph with a large number of edges. Every embedding of *G* on the torus induces an embedding of *H*. It will be either a 2-cell embedding, a cylindrical embedding, or a disc embedding. We start with a 2-cell embedding of *H,* and proceed as in Theorem 13.8 to find all ways of extending the embedding of *H* to *G*. This usually gives a number of embeddings. We then proceed to the cylindrical and disc embeddings of *H*. In each case, all possible ways of extending *H* to a 2-cell embedding must be exhaustively considered. For each

embedding *t,* we construct *M(Gt),* and compare the medial digraphs found for isomorphism, using graph isomorphism software.

If *G* is a 3-connected non-planar graph, we proceed recursively. We choose a vertex *u* and find all embeddings of *G*−*u*. Let the adjacent vertices to *u* be *u*1, *u*2,..., *uk*. If *G*−*u* is non-planar, then every embedding of it is a 2-cell embedding. If *u*1, *u*2,..., *uk* are all on the same facial cycle in some embedding of it, we can add vertex *u* to get an embedding of *G,* possibly in several ways. If *G*−*u* is planar, instead we first find a *TK*3, 3 in *G* with as many edges as reasonably possible (assuming a *TK*3, 3 exists). For each embedding of *TK*3, 3 in the torus, we exhaustively consider all possible ways of extending it to an embedding of *G,* and then use medial digraphs to compare the results.

For example, consider the graph *K*4. If *uu* is an edge of *K*4, then *K*4−*uu* is a theta-graph. We easily find that there are exactly two 2-cell embeddings of *K*4 on the torus. We then consider *K*5, one of the few non-planar graphs that does not contain *TK*3, 3. If u is a vertex of *K*5, then $K_5 - v \cong K_4$. For each embedding of *K*4 on the torus, including cylindrical and disc embeddings, we find all ways of adding *u* to the embedding. The result is six embeddings of *K*5, of which three are orientable and three non-orientable. We proceed to *K*6 by looking for a face of *K*5 containing all five vertices. We find there are four embeddings of *K*6, of which two are orientable and two non-orientable. Exactly one of these has all six vertices on a common face. This gives one embedding of *K*7, shown in Figure 13.17. It is an orientable embedding. Its dual is also shown. The dual is known as the *Heawood graph*.

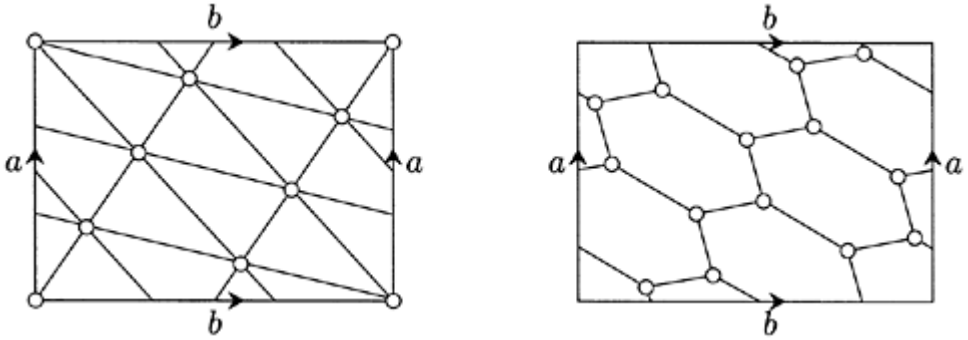


FIGURE 13.17
***K*7 and its dual, the Heawood graph, on the torus**

Exercises

13.4.1 Use cutting and pasting to convert the representation *c*+*d*+*e*+*c*−*d*−*e*− of the torus to normal form.

13.4.2 Construct the duals of the embeddings of *K*3, 3 on the torus of Figure 13.2.

13.4.3 Show that every 2-connected graph that is not a cycle has a theta-subgraph.

13.4.4 Describe *O(ε)* depth-first and breadth-first search algorithms to find a thetasubgraph of a 2-connected graph *G*.

13.4.5 Construct the two distinct embeddings of *K*3, 3 on the hexagonal form of the torus.

13.4.6 Verify that the two embeddings shown in Figure 13.16 are equivalent, and that the torus must be cut and pasted to construct a homeomorphism.

13.4.7 Show how to modify the definition of a medial digraph to allow for embeddings with loops (subdivide a loop with two vertices), and prove that it works.

13.4.8 Construct all distinct embeddings of *K*4, *K*5, and *K*6 on the torus.

13.4.9 Construct all distinct embeddings of the 3-prism on the torus. Begin with a thetagraph containing all six vertices.

13.4.10 Construct all distinct embeddings of the Petersen graph on the torus. Begin with a theta graph containing all 10 vertices.

13.4.11 Determine which graphs are shown in the toroidal embeddings of Figure 13.18. Determine the dual

graphs. (Note: None of the graph edges follow the boundary of the rectangles.)
13.4.12 Verify that the graphs in Figure 13.19 are distinct embeddings of the same graph. Do you recognize this graph? Are these embeddings orientable? Find the duals of both embeddings, and determine what graphs they are. (Note: None of the graph edges follow the boundary of the rectangles.)

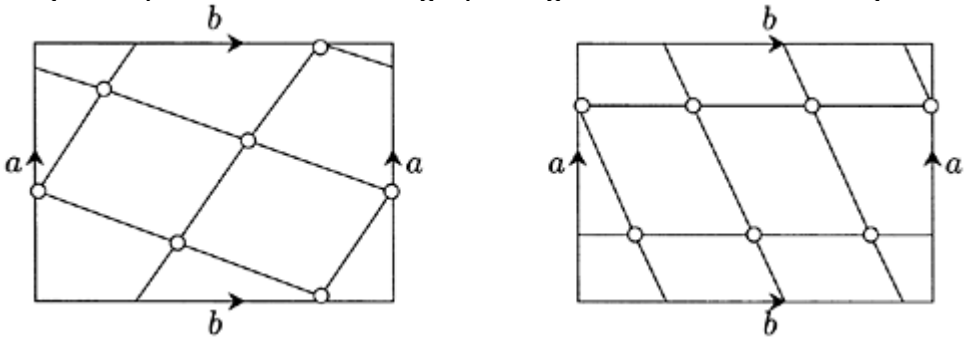


FIGURE 13.18
Two torus maps

13.4.1 Platonic maps on the torus

The embedding of *K*7 in Figure 13.17 shows a triangulation of the torus in which each vertex has degree six. By translating it repeatedly horizontally and vertically,

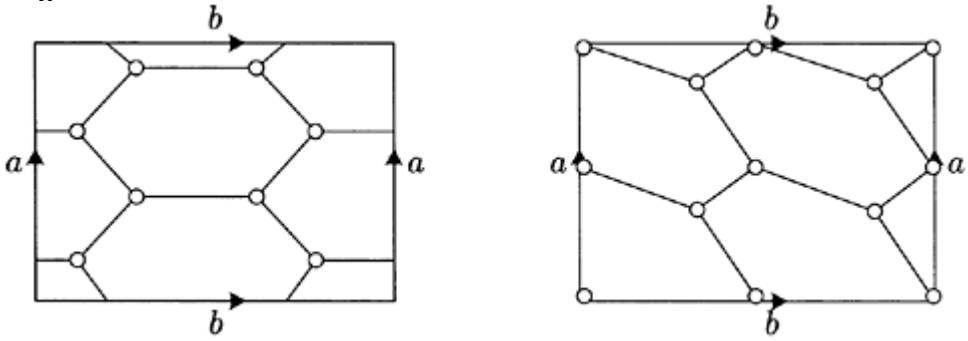


FIGURE 13.19
Two torus maps

we obtain a symmetric tiling of the plane by triangles. Its dual gives a symmetric hexagonal cover of the plane in which three hexagons meet at each vertex. The graphs in Figure 13.19 also give hexagonal coverings of the plane. Their duals will be 6-regular triangulations. The graphs of Figure 13.18 give symmetric tilings by parallelograms. These embeddings all belong to families of graphs with these properties. We will call them *Platonic maps*.

Let G be a k-regular torus map on n vertices whose dual map is ℓ-regular. For the plane, such graphs are the graphs of the Platonic solids. Then $nk=2\varepsilon=\ell f$. Using Euler's formula for the torus $n+f-\varepsilon=0$, we obtain

$$\frac{1}{k}+\frac{1}{\ell}=\frac{1}{2}$$

The only integral solutions are $(k,\ell)=(4, 4)$, (3, 6), and (6, 3). Clearly the last two are duals of each other. The graphs of Figure 13.18 are examples of the (4, 4)-pattern.

Consider a torus map Gt in which each vertex has even degree. Choose any edge uu. The incident edges at u are cyclically ordered by t. Let $DEG(u)= 2i$. The *diagonally opposite* edge to uu is uw, the ith edge following uu in $t(u)$. Given a vertex $u0$, with adjacent vertex $u1$, we construct a *diagonal path* $u0$, $u1$, $u2$,... by always choosing $uiui+1$ as the diagonally opposite edge to $ui-1ui$. Eventually a vertex must repeat, creating a cycle. Let the cycle be $C=(u0, u1, u2,..., um)$. C is a *diagonal cycle* if every edge is the diagonally opposite edge to its previous edge.

In the torus maps of *K*7 (Figure 13.17) and Figure 13.18 there are many diagonal cycles. They are drawn as straight lines diagonally across the rectangle. As can be seen in *K*7, a single diagonal cycle may wind around the torus several times.

Suppose now that Gt is a Platonic graph on the torus, with parameters (6,3)

or (4, 4). Consider a cycle *C*=(*u*0, *u*1, *u*2,..., *um*) constructed by following a diagonal path.
LEMMA 13.11 If C=(*u*0, *u*1, *u*2,..., *um*) *is a diagonal cycle in Gt, then C is an essential cycle*.
PROOF Suppose first that *G* has parameters (4,4), and suppose that *C* is contractible, with interior INT*(C)*. For each $v_i \in C$, there is one adjacent vertex in INT*(C)*. This is illustrated in Figure 13.20. Since each face of *Gt* has degree four, the interior adjacent vertices to *u*0, *u*1, *u*2,..., *um* form another diagonal cycle *C′* in INT*(C)*. Interior to *C′* is another diagonal" cycle, etc., leading to an infinite sequence of diagonal cycles, a contradiction. Therefore *C* must be an essential cycle. If *G* has parameters (6,3), the argument is nearly identical, except that each $v_i \in C$ has two adjacent vertices in INT*(C)*. Since *Gt* is a triangulation, we again find *C′* in INT*(C),* and so forth.

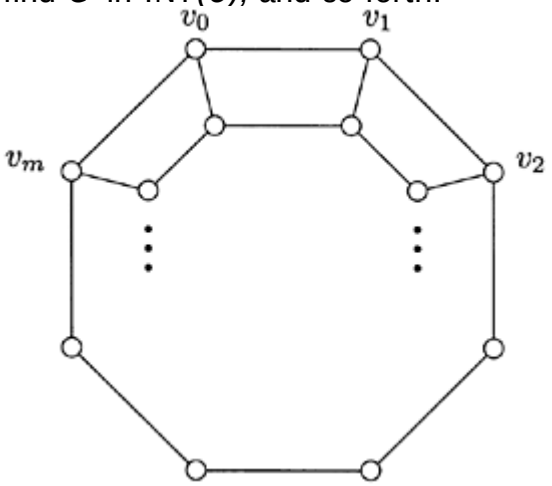


FIGURE 13.20
A diagonal cycle with *(k, ℓ)*=**(4, 4)**
The proof of Lemma 13.11 also shows how to draw *Gt*. Given a diagonal cycle *C*0, a sequence of "parallel" adjacent diagonal cycles is determined, *C*0, *C*1, *C*2,.... For any $v_0 \in C_0$, an edge not on *C*0 can then be selected, and a diagonal cycle containing it can be constructed. We find that the edges of *G* can be partitioned into "orthogonal" diagonal cycles $C'_0, C'_1, \ldots$ Each *Ci* winds

around the torus one or more times, intersecting each C'_j in a regular pattern, as can be seen from Figures 13.17 and 13.18.
If *C*=(*u*0, *u*1, *u*2,..., *um*) is any cycle constructed by following a diagonal path in a Platonic map, then the argument of Lemma 13.11 can be used to show that *C* must be a diagonal cycle. The only way in which *C* may fail to be a diagonal cycle is if one pair of edges, say *umu*0 and *u*0*u*1 are not diagonal edges. Suppose that *G* has parameters (4,4). We then find that *u*0 has either 0 or 2 adjacent vertices to the right of *C*. Since every face has degree four, the parallel cycle *C′* is either shorter than *C* or longer than *C,* by two edges. If it is longer than *C,* then the parallel cycle *C″* is again longer than *C′* by two edges, and so forth. As this leads to a contradiction, suppose that *C′* is two edges shorter than *C*. Then *C″* is again two edges shorter than *C′,* etc. Eventually we find a cycle of length four or five for which no parallel cycle can exist. If *G* has parameters (6,3), the argument is similar.

13.4.2 Drawing torus maps, triangulations

Read's algorithm for drawing a planar graph, given a rotation system, can be extended to torus maps. Let *Gt* be a 2-connected combinatorial embedding, with no vertices of degree two. Suppose that *G* has no loops, and that if there are multiple edges, no face is a digon. If Gt is not a triangulation, we can triangulate it by adding diagonal edges across non-triangular faces, so that no loops or digon faces are created. The smallest possible triangulation of the torus is shown in Figure 13.21. We denote it by *T*3. It can be constructed from the theta-graph shown in Figure 13.14 by adding one vertex *w,* adjacent to each vertex *u* and *u* three times. Notice that *T*3 is a 6-regular graph, whose faces are all triangles. It is the unique triangulation of the torus on three vertices (Exercise 13.4.4).
There is a triangulation on four points, denoted *T*4, which can be constructed from the rectangular form of the torus. A 3-vertex graph consisting of two digon cycles *(u, u)* and *(u, w)* with a common vertex *u* is 2-cell embedded on the torus. There is a single face, of degree eight. A fourth vertex *x* is placed in this face, and joined to each vertex on the boundary of the face. It is illustrated in Figure 13.22. In this diagram, the sides *a* and *b* of the rectangle are also graph edges. Notice that *T*4 has two vertices (*u* and *x*) of degree eight, and

two (*u* and *w*) of degree four.
Suppose that *G* has *n* vertices, with *n*3 of degree three, *n*4 of degree four, etc. Then since *Gt* is a triangulation, we have $3f=2\varepsilon$. Euler's formula then gives $\varepsilon=3n$, and:

$$3n3+2n4+n5=n7+2n8+3n9+...$$

LEMMA 13.12 Either there is a vertex of degree three, four, or five, or else all vertices have degree six.

Page 353

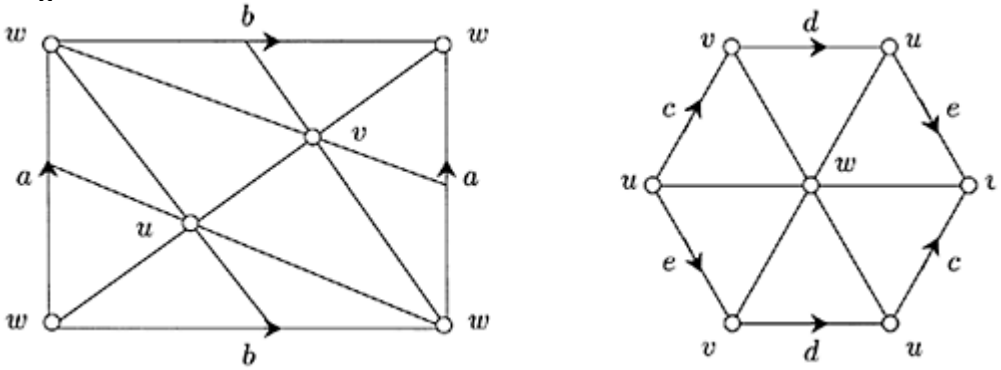


FIGURE 13.21
The triangulation *T*3

Now any triangulation in which all vertices have degree six is a Platonic map of type (6,3), and we know how to draw it as a tiling of the plane. Otherwise, there is a vertex of degree three, four, or five. We can use a modification of the algorithm REDUCEGRAPH() of Section 13.4.2 to reduce the triangulation *Gt* on *n* vertices to a triangulation on *n*−1 vertices, until either *T*3 or *T*4 results, or a 6-regular triangulation results. We must ensure that the reduction does not create any loops or digon faces.
Suppose that vertex *u* of *G* has degree three, four, or five, and suppose that *n*≥4. If DEG*(u)*=3, then *Gt*−*u* is a triangulation of the torus on *n*−1 vertices. If DEG*(u)*=4, suppose that *u* is adjacent to *u, w, x, y,* in cyclic order. If at least three of these vertices are distinct, then at least one of the diagonals of the 4-cycle *(u, w, x, y)* has distinct endpoints. Suppose it is *ux*. Then *Gt*−*u*+*ux* will be a triangulation on *n*−1 vertices, without loops or digon faces. Otherwise there are only two distinct vertices in the 4-cycle, which is then *(u, w, u, w);* that is, there are four parallel edges connecting *u* and *w*. Three of these parallel edges form a theta-graph, whose only embedding has a single face, a hexagon, shown in Figure 13.23. The fourth parallel edge cuts the hexagon into two quadrilaterals, one of which contains *u*.
The remaining vertices of *G* are located in the other quadrilateral. If *n*=4, then the map can only be the triangulation *T*4, with *u* and *w* as the two vertices of degree eight. If *n*≥5, there are at least two other vertices in the other quadrilateral. This quadrilateral and the vertices it contains determine a planar graph, which must have several vertices of degree three, four, or five. We choose one of these, and delete it instead of *u*.

Page 354

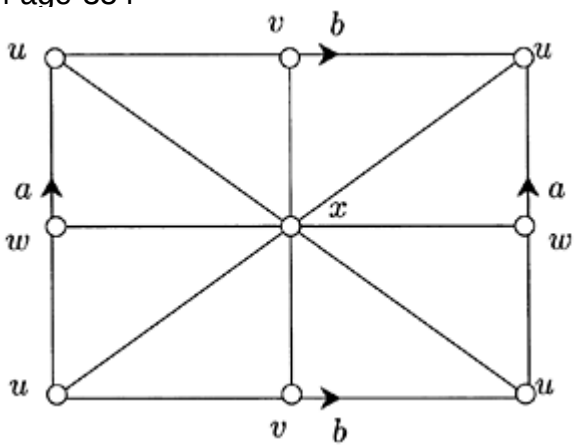


FIGURE 13.22
The triangulation *T*4

If DEG*(u)*=5, let *u* be adjacent to *u, w, x, y, z,* in cyclic order. If *u, x,* and *y* are distinct, we proceed as in the planar case, deleting *u* and adding two diagonals across the pentagon. Otherwise, we can assume that *u*=*x,*

since *G* has no loops.
If *w, y,* and *z* are distinct, then we can add the diagonals *wy* and *wz* to get a triangulation. Otherwise we can assume that *w=y*. But then *z, w,* and *x* are distinct, so that we can add the diagonals *zw* and *zx* to obtain a triangulation with no loops or digon faces. There are always at least three distinct vertices on the boundary of the pentagon. This gives the following theorem:
THEOREM 13.13 Let Gt be a torus map on n≥4 vertices which is not 6*-regular, with no loops or digon faces. Then Gt can be reduced to one of the following:*
*1. The triangulation T*3
*2. The triangulation T*4
3. A 6-regular triangulation

Page 355

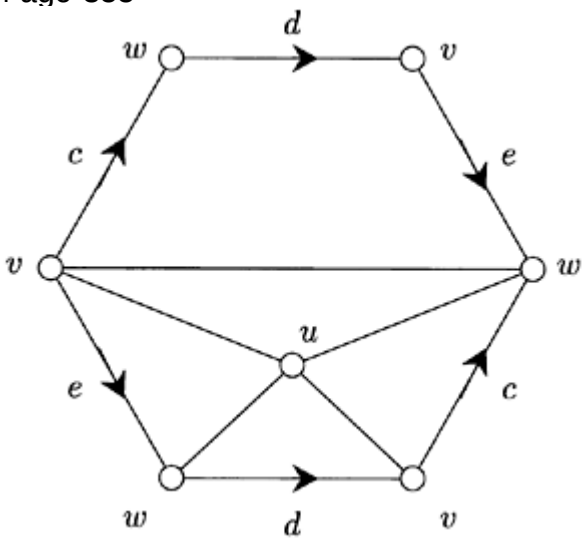


FIGURE 13.23
Reducing a triangulation, DEG*(u)*=4

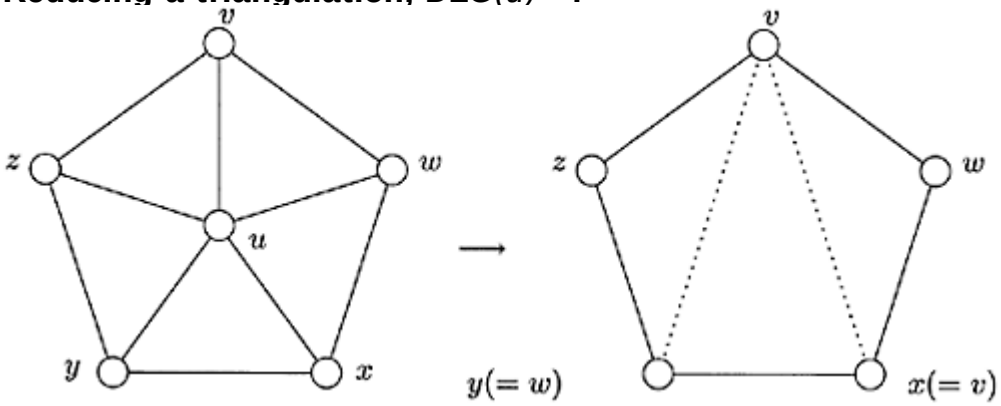


FIGURE 13.24
Reducing a triangulation, DEG*(u)*=5

Page 356

Algorithm 13.4.1: REDUCETORUSMAP*(G,t)*
if *G*=*T*3**or** *G*=*T*4**or** *G* is 6-regular **return (null)**
if there is a vertex *u* with DEG*(u)*=3

$$\textbf{then} \begin{cases} \text{let } t(u) = (uv, uw, ux) \\ G' \leftarrow G - u \\ \textbf{return } (G') \end{cases}$$

if there is a vertex *u* with DEG*(u)*=4

$$\textbf{then}\ \begin{cases}\text{let } t(u)=(uv,uw,ux,uy)\\ \textbf{if } v=x \textbf{ and } w=y\\ \quad \textbf{then pick a new } u \text{ from } V(G)-\{u,v,w\} \text{ of degree 4 or 5}\end{cases}$$

if DEG*(u)*=4

$$\textbf{then}\ \begin{cases}\text{let } t(u)=(uv,uw,ux,uy)\\ \textbf{if } v\neq x\\ \quad \textbf{then}\ \begin{cases}G'\leftarrow G'-u+vx\\ vx \text{ replaces } vu \text{ in } t(v) \text{ and } xv \text{ replaces } xu \text{ in } t(x)\end{cases}\\ \quad \textbf{else}\ \begin{cases}G'\leftarrow G-u+wy\\ wy \text{ replaces } wu \text{ in } t(w) \text{ and } yw \text{ replaces } yu \text{ in } t(y)\end{cases}\\ \textbf{return } (G')\end{cases}$$

pick *u* of degree 5, let *t(u)=(uu, uw, ux, uy, uz)*

if *u, x* and *y* are distinct

$$\textbf{then}\ \begin{cases}G'\leftarrow G-u+vx+vy\\ vx, vy \text{ replace } vu \text{ in } t(v)\\ xv \text{ replaces } xu \text{ in } t(x) \text{ and } yv \text{ replaces } yu \text{ in } t(y)\end{cases}$$

else if *w, y* and *z* are distinct

$$\textbf{then}\ \begin{cases}G'\leftarrow G-u+wy+wz\\ wy, wz \text{ replace } wu \text{ in } t(w)\\ yz \text{ replaces } yu \text{ in } t(y) \text{ and } zw \text{ replaces } zu \text{ in } t(z)\end{cases}$$

else if *x, z* and *u* are distinct

$$\textbf{then}\ \begin{cases}G'\leftarrow G-u+xz+xv\\ xz, xv \text{ replace } xu \text{ in } t(x)\\ zv \text{ replaces } zu \text{ in } t(z) \text{ and } vx \text{ replaces } vu \text{ in } t(v)\end{cases}$$

else if *y, u* and *w* are distinct

$$\textbf{then}\ \begin{cases}G'\leftarrow G-u+yv+yw\\ yv, yw \text{ replace } yu \text{ in } t(y)\\ vw \text{ replaces } vu \text{ in } t(v) \text{ and } wy \text{ replaces } wu \text{ in } t(w)\end{cases}$$

else if *z, w* and *x* are distinct

$$\textbf{then}\ \begin{cases}G'\leftarrow G-u+zw+zx\\ zw, zx \text{ replace } zu \text{ in } t(z)\\ wx \text{ replaces } wu \text{ in } t(w) \text{ and } xz \text{ replaces } xu \text{ in } t(x)\end{cases}$$

return *(G′)*

Algorithm 13.4.1 is input a triangulation of the torus, *G,* on $n \geq 4$ vertices with rotation system *t,* with no loops or digon faces. It constructs a triangulation *G′* on $n-1$ vertices, whenever possible. It can be used to successively reduce *Gt* to one of *T*3, *T*4, or a 6-regular triangulation. Drawings of *T*3 and *T*4 on the torus are shown in Figures 13.21 and 13.22. We can use these coordinatizations in a rectangle as topological embeddings. If a 6-regular triangulation is obtained, we can use diagonal cycles to obtain a coordinatization of it. These embeddings have no loops, and no digon faces. Every digon is embedded as an essential cycle. We then replace the deleted vertices in reverse order, exactly as in the planar case of READSALGORITHM(), using the visible region to assign coordinates to the deleted vertex. The result is a straight-line drawing of *Gt* on the torus; that is, a topological embedding *Gψ*.

We summarize this as follows:

THEOREM 13.14 Every torus map has a straight-line embedding in the rectangle and hexagon models of the torus.

Exercises

13.4.1 Find the duals of the embeddings of the triangulations *T*3 and *T*4 shown in Figures 13.21 and 13.22. What graphs are they?

13.4.2 Find the two tilings of the plane determined by a Platonic map of $K4,4$.
13.4.3 Find the tiling of the plane determined by a Platonic map of $C3 \times C3$.
13.4.4 Show that $T3$ is the unique triangulation of the torus on three vertices.
13.4.5 Show that there are two distinct triangulations of the torus on four vertices.
13.4.6 Show that there are five distinct embeddings of the cube on the torus.

13.5 Graphs on the projective plane

The projective plane is most conveniently represented as a disc with antipodal points identified. This is equivalent to the digon form $c+c+$ of the projective plane shown in Figures 13.7 and 13.9. An embedding of $K6$ and its dual are shown in Figure 13.25. It is easy to verify that the dual of $K6$ on the projective plane is the *Petersen graph*. As before, we shall only be concerned with *2-cell embeddings of 2-connected graphs*. Now it can be a little tricky to visualize the faces of an embedding on the projective plane, because the projective plane is non-orientable. Each point on the circumference of the disc is identified with its antipodally opposite point. When

an edge of the graph meets the disc boundary, it continues from the antipodal point. But the region immediately to the right of the edge as it meets the boundary is identified with the region immediately to the *left* of the antipodal point. A consequence is that rotation systems must be defined somewhat differently for non-orientable surfaces, and the algorithm FACIALCYCLE() which constructs the faces of an embedding must be modified.

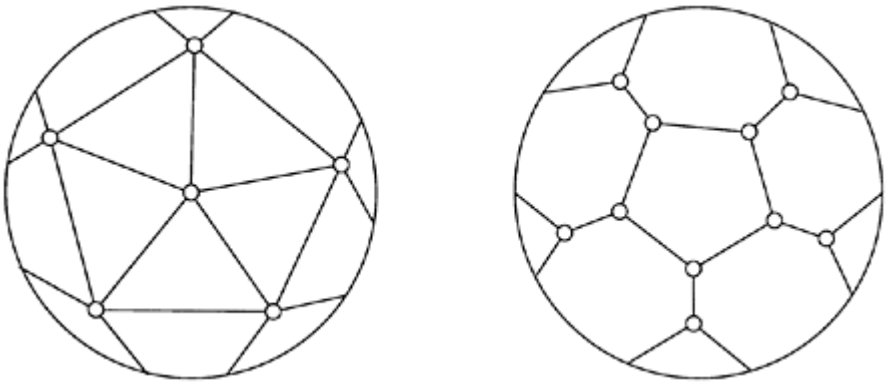


FIGURE 13.25
***K*6 and its dual, the Petersen graph, on the projective plane**

Let $G\psi$ be an embedding of a 2-connected graph G in the projective plane, and let u be a vertex of G. If we walk around $u\psi$ in a small clockwise circle, we encounter the incident edges in a certain cyclic order, say $uv1$, $uv2$,..., uvk. If we walk along the edges of $G\psi$, always staying within the disc, then the embedding appears exactly like a planar embedding. If we traverse an edge that crosses the boundary of the disc, and continue on until we reach $u\psi$ again, we find that the cyclic order of the incident edges at $u\psi$ has been reversed. Consequently a rotation system must be defined differently for a non-orientable surface. The projective plane is represented as a disc. We choose an orientation for this disc. Then given any vertex u, we walk around $u\psi$ in a small clockwise circle, and construct a cyclic list of incident edges. We assign a *signature* to each edge uv, denoted $\mathrm{SGN}(uv)$. If an edge $(uv)\psi$ crosses the boundary of the disc, then $\mathrm{SGN}(uv)=-1$. Otherwise it is $+1$. The signature does not depend on the direction in which the edge is traversed. In this way, the embedding ψ determines a signed rotation system.

We will use π to denote a rotation system for an embedding on the projective plane. For each vertex u, $\pi(u)$ denotes a cyclic list of *signed* incident edges. Two embeddings of $K3$, 3 are shown in Figure 13.26. The rotation systems corresponding to them are shown in Figure 13.27. Although the rotation systems are

different, the embeddings are equivalent.

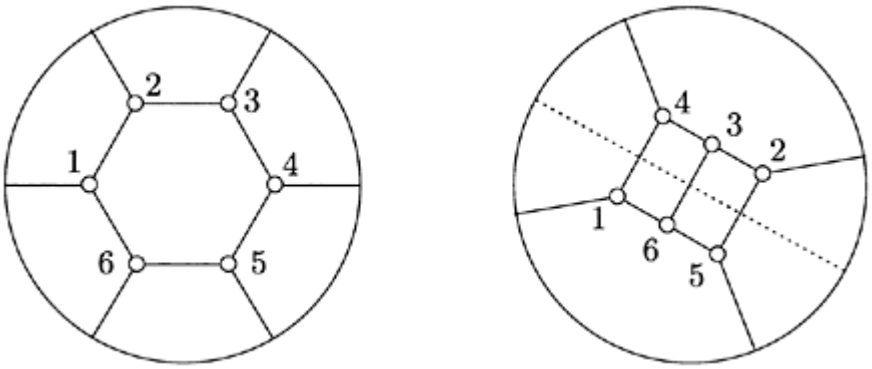


FIGURE 13.26
Two equivalent embeddings of *K*3, 3on the projective plane

$\pi(1) = (12, 16, -14)$ $\pi(1) = (14, 16, -12)$

$\pi(2) = (21, -25, 23)$ $\pi(2) = (23, -21, 25)$

$\pi(3) = (32, -36, 34)$ $\pi(3) = (34, 32, 36)$

$\pi(4) = (43, -41, 45)$ $\pi(4) = (41, -45, 43)$

$\pi(5) = (54, -52, 56)$ $\pi(5) = (52, -54, 56)$

$\pi(6) = (61, 65, -63)$ $\pi(6) = (61, 63, 65)$

FIGURE 13.27
Two rotation systems for *K*3, 3 on the projective plane

Given a topological embedding *Gψ,* this method of defining a rotation system for *G* is not unique, as it depends on the disc chosen to represent the projective plane. With an orientable surface, this situation does not arise. We must show that a signed rotation system uniquely determines the faces of an embedding, and that all rotation systems corresponding to *ψ* determine the same faces. In the embedding on the right of Figure 13.27, we can cut the projective plane along the non-contractible Jordan curve indicated by the dotted line, call it *C*. We then flip one-half of the disc over, and glue the two pieces along the common boundary.

We obtain another disc representing the projective plane, with antipodal points identified. This gives another rotation system *π′* for *G*. In Figure 13.27, the edges crossed by *C* will now have a signature of −1; edges which previously had a signature of −1 will now have +1. It is easy to see that with respect to the new disc, the embedding will have a face which is a 6-cycle (1, 2, 3, 4, 5, 6) with "spokes" to the boundary of the disc, exactly like the embedding on the left. Thus the embeddings are equivalent.

If *π* is a signed rotation system determined by an embedding *Gψ* and a disc representation of the projective plane, we call *Gπ* a combinatorial embedding. As we shall see, the faces of *Gψ* are completely determined by *Gπ*.

DEFINITION 13.15: A *projective map* is a combinatorial embedding *Gπ* of a 2-connected graph *G* on the projective plane, where π is a signed rotation system corresponding to a disc representation of the projective plane.

In order to show that all signed rotation systems arising from *Gψ* have the same facial cycles, we begin by rewriting the algorithm FACIALCYCLE() for a non-orientable surface. Notice that when traversing the facial cycles of a graph embedded on a disc representing the projective plane, the clockwise cyclic order of the incident edges viewed from above the disc appears counterclockwise when viewed from below, and vice versa. The algorithm uses a boolean variable *onTop* to indicate whether it is currently viewing the disc from above. Initially *onTop* has the value true. Each time an edge *uv* with SGN*(uv)*=−1 is encountered, it reverses the value of *onTop*. Any vertices visited while *onTop* is false will see a counterclockwise orientation for their incident edges. Those with *onTop* true will see a clockwise orientation.

When a graph is embedded on an orientable surface, the facial cycles are oriented cycles. We can assign a clockwise orientation to one cycle and the orientation of all adjacent cycles is determined, and so forth, so that an orientation can be assigned to the entire embedding. Reversing the cycles gives an equivalent, but reversed, embedding.

Graphs embedded on the projective plane do not have this property. If we try to assign an orientation to the facial cycles of an embedding *Gψ,* we can then choose different signed rotation systems corresponding to *Gψ,*

and different orientations of the cycles will be obtained. However, if we are given the (unoriented) facial cycles, they can be glued together uniquely along their common edges to construct a polygonal representation of the projective plane. Therefore if we are given the facial cycles, they will determine a topological embedding *Gψ*.
Algorithm 13.5.1 when given a signed rotation system π of an embedding *Gψ* on a non-orientable surface and a vertex *u* with an incident edge *e,* will find the facial cycle containing *e*.

Page 361

Algorithm 13.5.1: FACIALCYCLESGN*(Gπ, u, e)*
onTop←**true** "initially view the disc from above"
e′←*e*
repeat

comment: e' currently equals uv, for some v
if SGN$(e') = -1$ **then** *onTop* ← **not** *onTop*
$v \leftarrow$ other end of e'
$e'' \leftarrow$ edge of $\pi(v)$ corresponding to e'
comment: e'' currently equals vu
if *onTop*
  then $e' \leftarrow$ edge preceding e'' in $\pi(v)$
  else $e' \leftarrow$ edge following e'' in $\pi(v)$
$u \leftarrow v$

until *e′*=*e* **and** *onTop*

If this algorithm is applied to the combinatorial embeddings of *K*3, 3 given in Figure 13.27, identical faces will be constructed. Since the projective plane is constructed by gluing together the faces, it follows that the topological embeddings they represent are equivalent.
In Definition 13.14 combinatorial embeddings G^{t_1} and G^{t_1} on the torus were defined to be equivalent if there exists an autmorphism of *G* that induces a mapping of *t*1 to *t*2 or $\overline{t_2}$. This is inappropriate for signed rotation systems, as it cannot take the signatures into consideration. Therefore we define equivalence of signed rotation systems in terms of facial cycles.
*DEFINITION 13.16***:** Projective maps G^{π_1} and G^{π_2} are *equivalent* if they have the same facial cycles.
In general, let *Gψ* be a topological embedding, and consider two different representations of the projective plane as digons, *a*+*a*+ and *b*+*b*+. Let *πa* be the signed rotation system corresponding to *a*+*a*+ and let *πb* correspond to *b*+*b*+. The boundary of the disc given by *b*+*b*+ is a non-contractible Jordan curve *C* in the *a*+*a*+ representation. It will intersect the graph *Gψ* in one or more points. Refer to Figure 13.28. When the disc is cut along the curve *C,* it may cut some edges of *Gψ* more than once. If so, we subdivide those edges which are cut more than once. Hence we can assume that every edge of *Gψ* is cut at most once by *C*. If the curve cuts through a vertex, we can move it slightly so that it misses the vertex. We can do this because the graph has a finite number of vertices.
Suppose first that *C* cuts the boundary of the *a*+*a*+ disc in exactly two points (which are antipodal points), as in Figure 13.28. To transform the *a*+*a*+ representation into the *b*+*b*+ representation, we cut the disc along the dotted curve,

Page 362

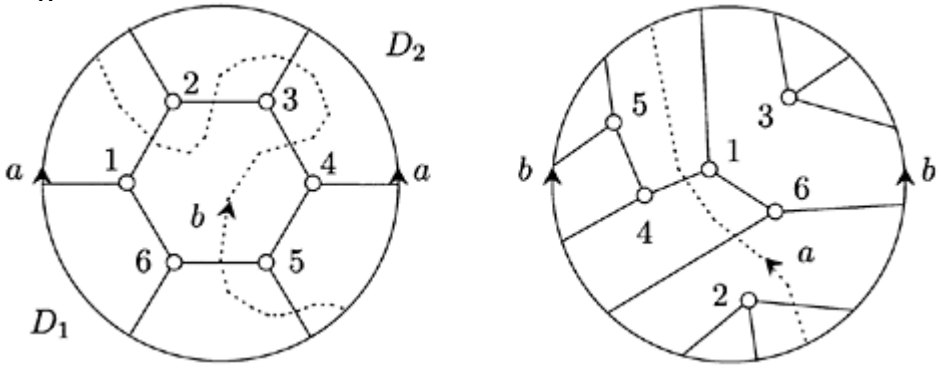


FIGURE 13.28
Representations *a*+*a*+ and *b*+*b*+ of the projective plane

then flip one-half of the disc over, and glue the two equivalent parts of the *a*+*a*+ boundary together. We can flip either half over. Denote the two parts of the disc obtained as *D*1 and *D*2, where *D*2 is the part that is flipped over. The result is a disc whose boundary is *b*+*b*+, shown as the disc on the right in Figure 13.28. We obtain *πb* from *πa* by *reversing* the cyclic adjacencies for all vertices in *D*2, and by assigning −1 to those edges that are cut by *C*. We now compare the facial cycles of G^{π_a} and G^{π_b} constructed by the algorithm FACIALCYCLESGN(). Let (*u*1, *u*2,..., *uk*) be a facial cycle constructed for G^{π_a}. Without loss of generality, suppose that FACIALCYCLESGN (G^{π_b}) begins to trace out this face from *u*1 which is in *D*1. If the entire face is within *D*1, the result will be the same as the face obtained using *πa*. If the facial cycle crosses from *D*1 to *D*2 via the *a*+*a*+ boundary, then since *D*2 was flipped upside down, and the cyclic adjacencies of πa were reversed to obtain *πb*, the same facial boundary will be constructed using *πb* or *πa*. If the facial cycle crosses from *D*1 to *D*2 via *C*, then since *πb* attaches a signature of −1 to these edges, the cyclic adjacencies will be reversed by FACIALCYCLESGN(). But the cyclic adjacencies of *πa* in *D*2 were also reversed in *πb*. The net effect is that the same facial boundary is constructed using *πb* or *πa* at each step of the algorithm. It follows that the two embeddings G^{π_a} and G^{π_b} have the same 2-cells as faces. Now we may have subdivided some edges of *G* before cutting and pasting the disc. Vertices of degree two do not affect the faces, and a cyclic order of two edges is invariant when reversed. Therefore, when traversing a facial cycle along a path created by subdividing an edge, the important factor is the number of −1's encountered. Hence we can contract any subdivided edges and assign a signature of −1 if the number of edges in the subdivided path was odd. The result will be the same facial cycle. We conclude that the faces of G^{π_a} and G^{π_b} are identical, so that the embeddings are equivalent.

Page 363

Suppose now that *C* cuts the boundary of the *a*+*a*+ disc in more than one pair of antipodal points, where *C* is the boundary of the *b*+*b*+ disc. There are an infinite (in fact, uncountable) number of possible non-contractible Jordan curves *C*. But there are only a finite number of possible signed rotation systems for *Gψ*, since *G* is finite. Therefore we will only consider non-contractible Jordan curves which meet the boundary *a*+*a*+ in a finite number of points. A more complete treatment can be found in the book of MOHAR and THOMASSEN [88].

If *C* cuts the boundary of the *a*+*a*+ disc in more than one pair of antipodal points, we proceed by induction. We are given a graph *G* embedded on the disc, with a rotation system πa. As before we assume that *C* does not cut any vertices of *Gψ*. Choose two consecutive points *P* and *Q* on *C* at which the disc boundary is cut such that one of the intervals *[P, Q]* and *[Q, P]* on the boundary is not cut by any other points of *C*. This is illustrated in Figure 13.29, where *C* is the dotted curve. Let the antipodal points of *P* and *Q* be *P′* and *Q′*. Let *C[P, Q]* denote the portion of *C* from *P* to *Q*. Make a cut in the disc very close to *C[P, Q]*, cutting off a portion *D*1 of the disc, so that *C[P, Q]* is completely contained within *D*1, but so that the only part of *G* that is affected by the cut are the edges of *G* that cross *C[P, Q]*. This is possible because the graph is finite. Let the remainder of the disc be *D*2. We now flip *D*1 over, and glue the matching boundaries of *D*1 and *D*2 near *P′* and *Q′*. The result is a disc representation of the projective plane such that *C* cuts its boundary in four fewer points. Let the boundary of the new disc be *c*+*c*+, and let the signed rotation system corresponding to it be *πc*.

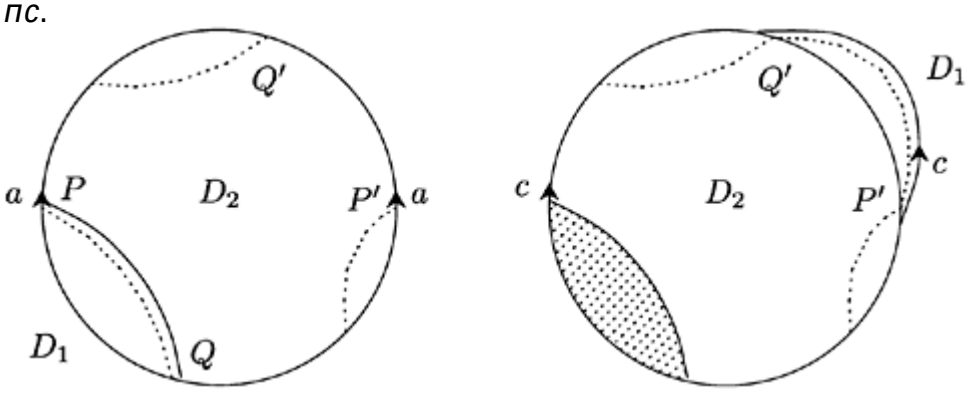


FIGURE 13.29
Transforming a disc representation of the projective plane

Consider the faces of G^{π_a} and G^{π_c}. The rotation systems *πa* and *πc* differ only in edges which are within *D*1, or which cross from *D*1 to *D*2. Vertices within *D*1 have adjacency lists of opposite orientation in *πa* and *πc*. FACIALCYCLESGN() will construct the same faces for both *πa* and *πc*. With respect to the π*c* disc,

Page 364
C has fewer intersections with the boundary. We use induction to conclude that FACIALCYCLESGN() will construct the same faces for both *πc* and *πb*. It follows that G^{π_a} and G^{π_b} always have the same faces. Because the projective plane is constructed by gluing the faces together, this gives:

THEOREM 13.15 Let Gψ be an embedding on the projective plane. Given any two signed rotation systems πa and πb for Gψ, corresponding to different disc representations of the projective plane, G^{π_a} *and* G^{π_b} *are equivalent embeddings.*

THEOREM 13.16 Let G^{ψ_1} *and* G^{ψ_2} *be topological embeddings of G on the projective plane with corresponding combinatorial embeddings* G^{π_a} *and* G^{π_b}*, with respect to two disc representations of the projective plane. Then* G^{ψ_1} *and* G^{ψ_2} *are homeomorphic if and only* G^{π_a} *and* G^{π_b} *are equivalent.*

PROOF If G^{π_a} and G^{π_b} are equivalent, they have the same facial cycles. The faces are homeomorphic to 2-cells bounded by the facial cycles. It follows that the faces of G^{π_a} and G^{π_b} determine a homeomorphism of the embeddings G^{ψ_1} and G^{ψ_2}. Therefore G^{ψ_1} and G^{ψ_2} are homeomorphic if and only if G^{π_a} and G^{π_b} are equivalent.

If *Gp* is a planar embedding of a 2-connected graph, it is very easy to convert *p* to a projective rotation system *π*. When *G* is drawn in the plane, one face is always the outer face. We draw *G* in a disc representing the projective plane. We then choose any edge *e* on the outer face, and reroute it so that it crosses the boundary of the disc. The result is a projective map. The two faces on either side of *e* in the planar map become one face in the projective map. The cyclic order of adjacent vertices is unchanged, for all vertices of *G*. Thus, *p* can be converted to a projective rotation system, by assigning a signature of −1 to any one edge of *G*. However, the embeddings constructed in this way are somewhat unsatisfactory, as there is a non-contractible Jordan curve in the surface which cuts the embedding in only one point.

13.5.1 The facewidth

The projective plane has unorientable genus one. The torus has orientable genus one. Although they both have genus one, these surfaces behave very differently. *K*7 can be embedded on the torus. Yet it is easy to see that it cannot be embedded on the projective plane, as the unique embedding of *K*6 shown in Figure 13.25 cannot be extended to *K*7. Alternatively, Euler's formula can be used to show that *K*7 has too many edges to embed on the projective plane. However, there are infinite families of graphs that can be embedded on the projective plane, but not on the torus.

Page 365
We begin with two families of graphs called the *Möbius ladder* and *Möbius lattice,* which can be embedded on both the projective plane and torus.

DEFINITION 13.17: The *Möbius ladder L*2*n* is the graph with 2*n* vertices {*u*1, *u*2,..., *u*2*n*} such that $v_i \longrightarrow v_{i+1}$, and $v_i \longrightarrow v_{i+n}$, where subscripts larger than 2*n* are reduced modulo 2*n*.

The Möbius ladder *L*6 is just *K*3, 3, shown in Figure 13.26. *L*8 is shown in Figure 13.30. Notice that *L*2*n* is always a 3-regular graph.

DEFINITION 13.18: The *Möbius lattice L*2*n*−1 is the graph with 2*n*−1 vertices {*u*1, *u*2,..., *u*2*n*−1} such that $v_i \longrightarrow v_{i+1}$, $v_i \longrightarrow v_{i+n-1}$, and $v_i \longrightarrow v_{i+n}$ where subscripts larger than 2*n*−1 are reduced modulo 2*n*−1.

The Möbius lattice *L*5 is just *K*5. *L*7 is shown in Figure 13.30. Notice that *L*2*n*−1 is always a 4-regular graph.

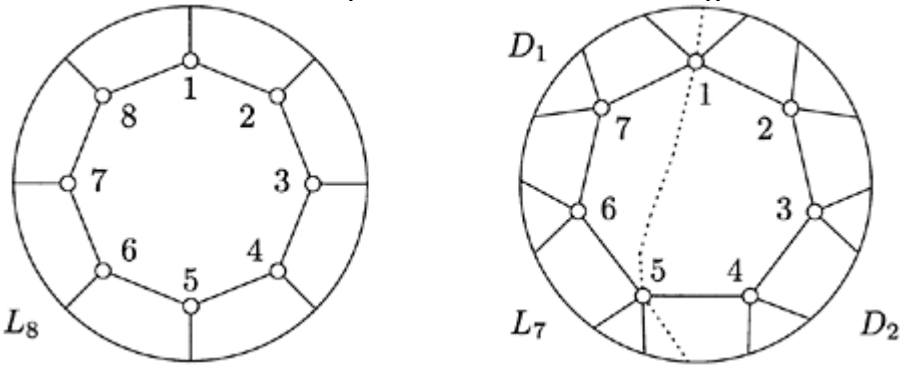


FIGURE 13.30
The Möbius ladder *L*8 and Möbius lattice *L*7

There is a clever trick that can be used to convert these projective embeddings of *L*2*n* and *L*2*n*−1 to toroidal embeddings. Draw an essential cycle *C* across the disc as shown by the dotted line in Figure 13.30, dividing it

into two parts, *D*1 and *D*2. Notice that *C* intersects the graph embedding in only two points, representing the vertices 1 and 5. Vertices 1 and 5 are both joined to several vertices located in *D*1 and *D*2. Now cut the disc along *C,* flip *D*2 over, and glue *D*1 and *D*2 along the common boundary to get a new disc representation of the projective plane, as shown in Figure 13.31. Vertices 1 and 5 are on the boundary of the new disc. These antipodal points are the only points of the embedding *Gψ* on the disc boundary. Therefore we can convert the disc into the hexagonal form of the torus, obtaining an embedding on the torus.

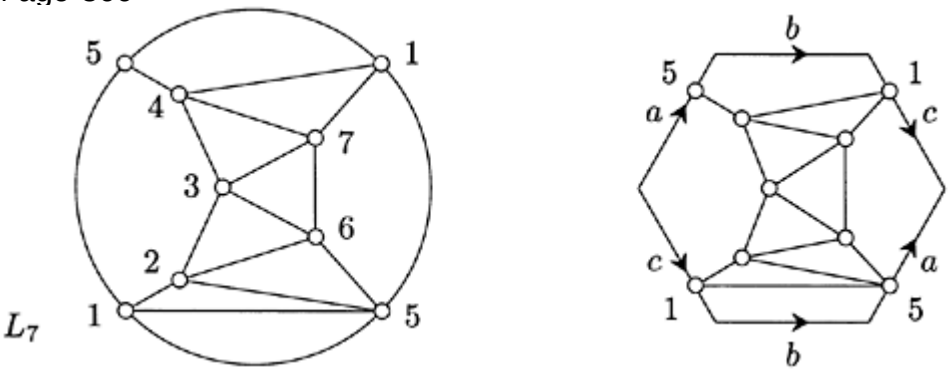


FIGURE 13.31
Converting a projective embedding to a toroidal embedding

DEFINITION 13.19: Let *Gψ* be a graph embedding in a surface Σ. Let *C* be a non-contractible Jordan curve in Σ. The *facewidth* of *C* is fw*(C),* the number of points of *Gψ* common to *C*. The *facewidth* of *Gψ* is fw*(Gψ),* the minimum fw*(C),* where *C* is any non-contractible Jordan curve in Σ.

The facewidth is sometimes known as the *representativity* of an embedding. Let *C* be a non-contractible Jordan curve of minimum possible facewidth, for an embedding *Gψ*. If *C* intersects a face *F,* then it also intersects the boundary of *F*. If an intersection point is not at a vertex, then it is at an interior point of an edge *e*. Then *C* also intersects the face on the other side of *e*. In such a case, we can alter *C* slightly so that it passes through *e* at an endpoint of the edge. The result is another non-contractible Jordan curve, also of minimum facewidth. This gives the following:

LEMMA 13.17 Given any embedding Gψ, there is a non-contractible Jordan curve C of facewidth fw*(Gψ) such that C intersects Gψ only at images of vertices*.

Now the faces of an embedding *Gψ* are determined completely by its rotation system. If *C* intersects *Gψ* only at images of vertices, then *C* determines a cyclic sequence of vertices (*u*1, *u*2,..., *uk*), where *k*=fw*(C),* such that consecutive vertices are on the same facial boundary. It follows that fw*(Gψ)* depends only on the rotation system, so that we can also write fw*(Gπ)*. We show that the method used above to convert a projective embedding of *L*7 to a toroidal embedding works in general, whenever fw*(Gπ)* is at most three.

THEOREM 13.18 Let Gπ be a projective embedding with facewidth at most three. Then G can be embedded on the torus.

PROOF The proof proceeds as in the previous example. Let *C* be a non-contractible Jordan curve with fw*(C)*≤3. Use the hexagonal form of the torus, $a+b+c+a-b-c-$, as in Figure 13.32. We cut the disc of the projective plane along *C* obtaining *D*1 and *D*2, which we glue together to obtain a new disc. Without loss of generality, assume that fw*(C)*=3, so that there are three pairs of antipodal points of *Gπ* on the boundary of the new disc, call them *p, q,* and *r,* and suppose that they occur in this order along the boundary of the disc. We convert the disc into the hexagonal form of the torus, as in Figure 13.32. We place *p* on the side of the hexagon labeled *a, q* on the side labeled *b,* and *r* on the side labeled *c*. In the hexagon, the identified pairs of sides are not antipodally reversed as in the disc model of the projective plane. However, there is only one point *p, q,* or *r* on each side, so that the sequence of points on the boundary is the same. The result is a toroidal embedding of *G*.

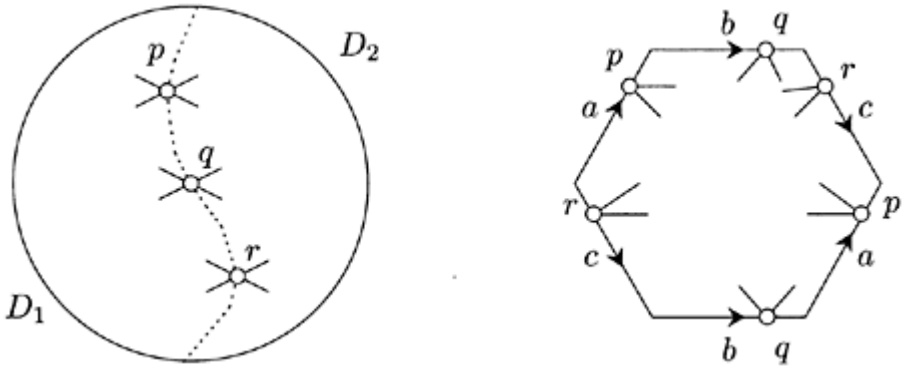


FIGURE 13.32
Converting a projective embedding to a toroidal embedding

This transformation will only work when fw*(C)*≤3. It has been shown by FIEDLER, HUNEKE, RICHTER, and ROBERTSON [41] that the converse of this theorem is also true; so that if a toroidal graph can be embedded on the projective plane, then the facewidth on the projective plane is at most three. We can use this theorem to construct projective graphs which are not toroidal. The construction uses Möbius ladders or lattices. Suppose that we start with a Möbius ladder *L*2*n* with vertices {*u*1, *u*2,..., *u*2*n*}. Add 2*n* more vertices *u*1, *u*2, ..., *u*2*n* forming a cycle in which $u_i \longrightarrow u_{i+1}$, and add the edges $u_i \longrightarrow v_i$, for all *i*. The result is a graph as shown in Figure 13.33. The facewidth of this graph is four, if *n*≥4,

so that it cannot be embedded in the torus. The facewidth can be made arbitrarily high by increasing *n* and adding more cycles in this fashion.

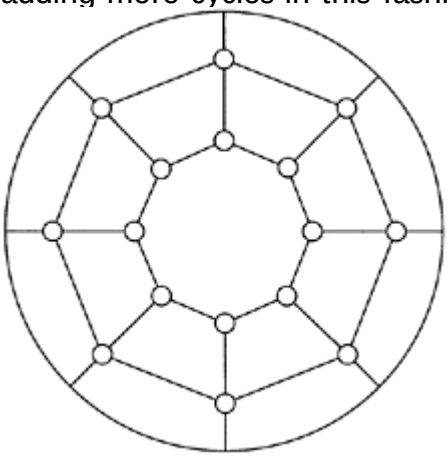


FIGURE 13.33
An embedding with facewidth four

13.5.2 Double covers

The projective plane can be viewed as exactly one-half of a sphere, by considering the disc representing the projective plane as the upper hemisphere of a sphere. If we choose a great circle as the equator of a sphere, and identify antipodal points on the sphere, the result is a two-to-one mapping of the sphere onto the projective plane. Open discs on the sphere are mapped to open discs on the projective plane. The equator becomes the boundary of the disc representing the projective plane. Thus, we say that the sphere is a *double cover* of the projective plane.

It is also possible for one graph to be a double cover of another graph.

DEFINITION 13.20: Let *G* and *H* be simple graphs such that there is a two-to-one mapping $\theta: V(H) \rightarrow V(G)$ with the property that θ induces a two-to-one mapping of edges, $\theta: E(H) \rightarrow E(G)$. Then *H* is said to be a *double cover* of *G*.

Since the sphere is a double cover of the projective plane, given any graph *G* embedded on the projective plane, there is a corresponding planar graph *H* embedded on the sphere, such that *H* is a double cover of *G*. We begin with an embedding *Gп* with respect to a disc representation of the projective plane. We make a copy of the disc, and transform it by reflecting each of its points in the center of the disc (an antipodal reflection). We then glue the two discs together

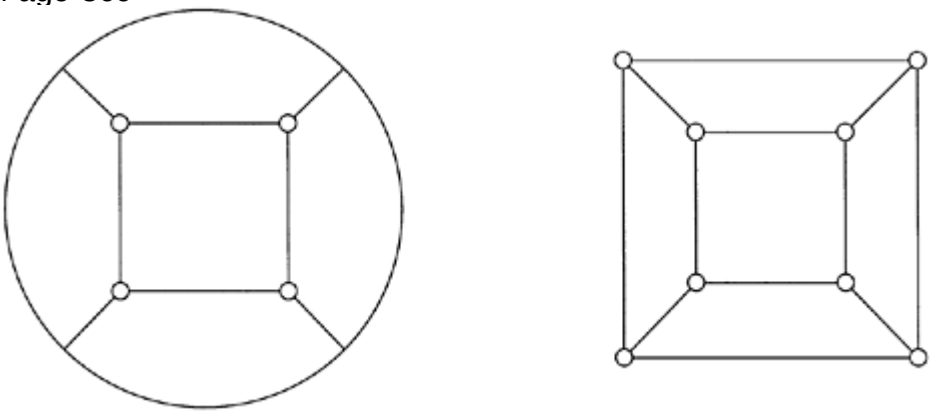


FIGURE 13.34
The cube is a double cover of *K*4

along their common boundary. The common boundary of the discs becomes the equator of the sphere. The result is an embedding on the sphere of a double cover *H* of *G*. The edges of *H* which cross the equator are those with signature −1 in *π*. We denote this double cover by *DC(Gπ)*. An example is illustrated in Figure 13.34, showing that the graph of the cube is a double cover of *K*4. We also find that the dodecahedron is a double cover of the Petersen graph, as can be seen from Figure 13.25.

The *medial digraph* of an embedding as expressed in Definition 12.19 is inappropriate for unorientable surfaces. Although it encapsulates the cyclic adjacencies of each vertex, it does not take into account the signature of the edges. In order to distinguish inequivalent projective embeddings of a graph, and to determine the symmetries (automorphisms) of an embedding, we can use the double cover.

THEOREM 13.19 Let G^{π_1} *and* G^{π_2} *be projective embeddings with corresponding spherical embeddings* $DC(G^{\pi_1})^{p_1}$ *and* $DC(G^{\pi_2})^{p_2}$. *Then* G^{π_1} *and* G^{π_2} *are equivalent if and only if* $DC(G^{\pi_1})^{p_1}$ *and* $DC(G^{\pi_2})^{p_2}$ *are equivalent.*

PROOF Consider a projective embedding *Gπ*. Let *H=DC(Gπ),* and let *Hp* be the spherical embedding determined by *Gπ*. Let *V(G)*={*u*1, *u*2,..., *un*}, *V(H)*={*u*1, *u*2,..., *un, w*1, *w*2,..., *wn*}, and let *θ: V(H)→V(G)* be the natural double cover mapping; namely, *θ(ui)=θ(wi)=ui*. The mapping *θ* maps the cyclic adjacency lists of *ui* and *wi* to the cyclic adjacency list of *ui,* but the orientations are opposite to each other. Notice that *H* has an *antipodal automorphism;* namely, the unique permutation of *V(H)* which interchanges *ui*

and *wi* is an automorphism of *H*. Corresponding to each face of *Gπ,* there are two faces of *Hp,* and these faces have opposite orientation. Each face of *Hp* corresponds uniquely to a face of *Gπ*. We see that *H* is an unorientable planar graph.

Suppose that G^{π_1} and G^{π_2} are equivalent embeddings. As the topological embeddings they represent are homeomorphic, they correspond to different disc representations of a projective embedding of *G*. Let $H = DC(G^{\pi_1})$, with corresponding spherical embedding *Hp*. The boundary of the disc corresponding to G^{π_2} is a non-contractible Jordan curve *C* in the projective plane. There is a unique antipodally symmetric Jordan curve *C′* on the sphere corresponding to *C*. As in Figures 13.28 and 13.29, we cut the disc of G^{π_1} along *C* to get *D*1 and *D*2, flip *D*2 over, and glue the two parts along the common boundary to obtain the disc of G^{π_2}. The corresponding transformations on the sphere are as follows. The equator of the sphere corresponds to the boundary of the *π*1-disc. The curve *C′* corresponds to the boundary of the *π*2-disc. The equator and *C′* intersect in a pair of antipodal points, dividing the surface of the sphere into four crescents, which correspond alternately to *D*1 and *D*2. The transformation of the *π*1-disc into the *π*2-disc is equivalent to taking *C′* as the new equator, leaving *H* unchanged. It follows that $DC(G^{\pi_2})^{p_2}$ and $DC(G^{\pi_1})^{p_1}$ are equivalent.

Conversely, suppose that $DC(G^{\pi_1})^{p_1}$ and $DC(G^{\pi_2})^{p_2}$ are equivalent. Since a double cover is an unorientable planar graph, the embeddings of $DC(G^{\pi_1})$ and $DC(G^{\pi_2})$ on the sphere can only differ in their equators. It follows that G^{π_1} and G^{π_2} are equivalent disc representations of homeomorphic projective embeddings.

Notice that if $DC(G^{\pi_1})$ is a 3-connected graph, by Whitney's theorem, it has a unique embedding on the sphere. Consequently G^{π_1} and G^{π_2} will be equivalent if and only if $DC(G^{\pi_1}) \cong DC(G^{\pi_2})$ in this case.

Exercises

13.5.1 Find all embeddings of *K*4 and *K*5 on the projective plane.

13.5.2 Find a projective embedding of $K3, 4$ and find its projective dual. What graph is it?
13.5.3 Let $G\psi$ be a projective embedding of G, and let π be an associated rotation system. Let C be any cycle of G. Show that C is an essential cycle of $G\psi$ if and only if the number of edges of C with signature -1 is congruent to 2 (mod 4).
13.5.4 Find the dual maps of the embeddings shown in Figure 13.30.
13.5.5 Show that the Möbius ladder $L2n$ contains a topological subgraph $TL2n-2$, when $n \geq 3$.
13.5.6 Show that the Möbius lattice $L2n-1$ is a minor of $L2n+1$, if $n \geq 3$.

Page 371
13.5.7 Show that the Möbius ladder $L2n$ has an embedding on the torus in which all faces are hexagons. Construct the embedding for $n=4$, 5, and find the dual map. (*Hint:* In the rectangular representation of the torus, draw a cycle of length $2n$ which wraps around the torus twice. Show how to complete this to a hexagonal embedding of $L2n$.)
13.5.8 Show that the Möbius lattice $L2n-1$ has an embedding on the torus in which all faces are quadrilaterals. Construct the embedding for $n=4,5$, and find the dual map. (*Hint:* In the rectangular representation of the torus, draw a cycle of length $2n-1$ which wraps around the torus twice. Show how to complete this to a quadrilateral embedding of $L2n-1$.)
13.5.9 Show that there is a unique triangulation of the projective plane with three vertices and six edges.
13.5.10 Show that Read's algorithm for drawing a planar graph can be adapted to the projective plane. Show that there is a unique triangulation of the projective plane on three vertices, and that any triangulation can be reduced to it by deleting vertices of degrees three, four, or five, and adding diagonals to the faces obtained. Conclude that every projective graph has a straight-line drawing in the disc model of the projective plane.
13.5.11 Show that the graph of the $2n$-prism is a double cover of the Möbius ladder $L2n$.
13.5.12 The cube is a double cover of $K4$. Find another double cover of $K4$.
13.5.13 Find a double cover of $K6$, as illustrated in Figure 13.25.
13.5.14 Find a projective embedding of the graph of the cube, and find its double cover.
13.5.15 The *Desargues* graph is shown in Figure 13.35. The Desargues graph is non-planar, non-projective, and non-toroidal. Show that it is a double cover of the Petersen graph.

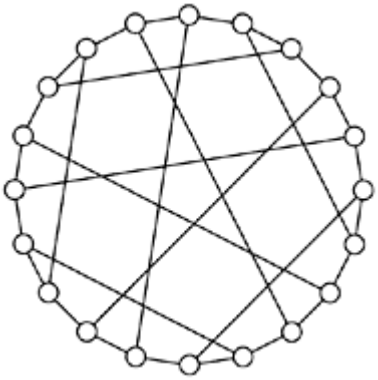


FIGURE 13.35
The Desargues graph

Page 372
13.6 Embedding algorithms
In this section, we outline an algorithm to determine whether a 2-connected graph G can be embedded on the projective plane, and to find an embedding $G\pi$. It is modeled on algorithms of Gagarin, Mohar, and Myrvold and Roth. If G is planar, we know how to convert a planar embedding to a projective embedding. Hence we can assume that G is non-planar, so that it contains a Kuratowski subgraph.

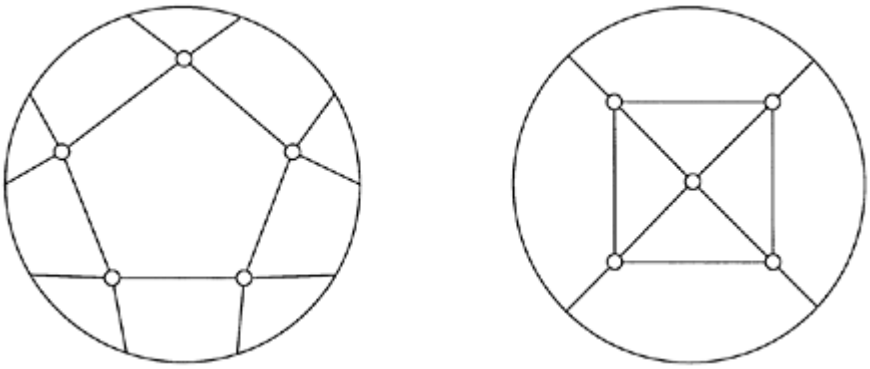


FIGURE 13.36
The embeddings of *K*5on the projective plane

There is exactly one embedding of *K*3, 3 on the projective plane, shown in Figure 13.26, and two embeddings of *K*5, shown in Figure 13.36. These embeddings all have the property that there are no repeated vertices on any facial cycle. In Figure 13.26, the hamilton cycle (1, 2, 5, 6, 3, 4) is an essential cycle. Since *K*3, 3 has a unique embedding on the projective plane, this gives:

*LEMMA 13.20 In any embedding of K*3, 3 *in the projective plane, some hamilton cycle is an essential cycle.*

If we now cut the projective plane along this cycle, the result is a disc in which each vertex 1, 2,..., 6 appears twice on the boundary. The resulting diagram, shown in Figure 13.37, is a very convenient representation of the projective plane.

Consider a subgraph *TK*3, 3 in *G*. We want to determine whether *G* can be embedded in the projective plane. We will begin by embedding the subgraph *TK*3, 3. There are six hamilton cycles of *K*3, 3. Each corresponds to a cycle of *TK*3, 3. One of them must be essential. Exercise 13.6.2 describes an easy way to enumerate the hamilton cycles of *K*3, 3. We take each of the six cycles in turn, and construct an embedding of *TK*3, 3, as in Figure 13.37, and try to extend it

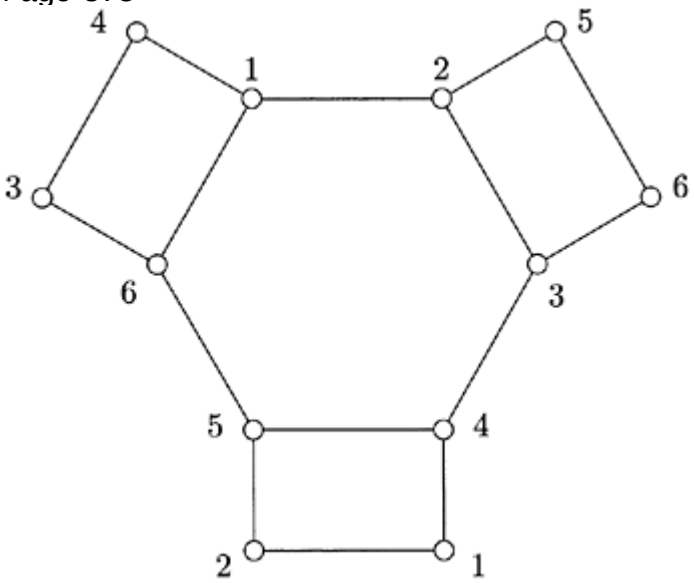


FIGURE 13.37
A representation of the projective plane

to an embedding of *G*. If any one succeeds, then *G* is projective. Otherwise we conclude that *G* is non-projective.

The embedding of *K*3,3 divides the projective plane into four faces—a hexagon, and four quadragons. The remaining vertices and edges of *G* must be placed in one of these faces. If we delete *V*(*TK*3, 3) from *G,* the result is a subgraph consisting of a number of connected components. If *H* is such a connected component, then since *G* is 2-connected, there must be at least two edges with one endpoint in *H* and the other in *TK*3, 3.

DEFINITION 13.21: A *bridge* of *G* with respect to *TK*3, 3 is either:

*1.*An edge *uu,* where *u,* $v \in V(TK_{3,3})$ but $uv \notin E(TK_{3,3})$, or
*2.*A connected component *H* of *G*−*V*(*TK*3, 3) together with all edges connecting *H* to *TK*3, 3

If *B* is a bridge, then a *vertex of attachment* of *B* is any vertex *u* of *B* such that $u \in V(TK_{3,3})$.

We can use a breadth-first (BFS) or depth-first search (DFS) to find the bridges of *G* with respect to an embedding of *TK*3, 3. Each bridge has at least two vertices of attachment. Since each face is a 2-cell, and each bridge must be embedded in a face of *TK*3, 3, each bridge must be planar. Into which faces of *TK*3, 3 can the bridges be placed?
The embedding of *K*3, 3 in Figure 13.37 determines a classification of the vertices and edges of *K*3, 3. Edges on the boundary of the hexagon are called *hexagon*

Page 374
edges. Edges which are on the boundary of a quadragon, but not on the hexagon are called *quadragon edges*. Hexagon edges like {1, 2} and {4, 5} are called *opposite* edges. Vertices like 1 and 4 are called *diagonally opposite* vertices, because they are diagonally opposite on the hexagon. By a *path* of *TK*3, 3 we mean a path connecting two corner vertices. The paths of *TK*3, 3 corresponding to hexagon edges of *K*3, 3 are called *hexagon paths,* those corresponding to opposite edges of the hexagon are called *opposite paths,* and so forth. In general, a path of *TK*3, 3 will be either a hexagon or a quadragon path of *TK*3, 3. The following lemmas on bridges can be proved by considering all possibilities of placing a bridge in Figure 13.37.
*LEMMA 13.21 A bridge B can be placed in three faces of TK*3, 3 *if and only if B has exactly two vertices of attachment, which are diagonally opposite vertices*.
*LEMMA 13.22 A bridge B can be placed in two faces of TK*3, 3 *if and only if all vertices of attachment of B are on the same path, or on opposite paths of TK*3, 3.
It follows from these lemmas that a bridge *B* can be placed in at most three faces, and that bridges for which these lemmas do not apply, either cannot be placed in any face, or can be placed in at most one face. A bridge is *compatible* with a face if it can be placed in the face. A bridge *B* is a *k-face* bridge if it can be placed in exactly *k* faces of *TK*3, 3. Thus, we have 3-face, 2-face, 1-face, and 0-face bridges with respect to an embedding of *TK*3, 3. We can determine which faces a bridge may embed in by using its vertices of attachment and the previous lemmas.
Two bridges *B*1 and *B*2 *conflict* in face *F* if they can both be placed in face *F,* but cannot be simultaneously placed in face *F*. Suppose that *B*1 can be embedded in face *F* and that it has *k* vertices of attachment *u*1, *u*2, ..., *uk,* where *k*≥2*,* and where the vertices occur in that order on the facial cycle of *F*. The vertices divide the facial cycle into *k* intervals [*u*1, *u*2], [*u*2, *u*3],..., [*uk*−1, *uk*], [*uk, u*1], where each interval is a path from *ui* to *ui*+1. If *B*2 is another bridge that can also be embedded in face *F,* then *B*1 and *B*2 do not conflict if and only if all vertices of attachment of *B*2 lie in one interval of *B*1, and vice versa.
Suppose that *B* is a 3-face bridge, with vertices of attachment *u* and *u*. All 3-face bridges with these vertices of attachment can be combined into one bridge, as they can always all be embedded in the same face if any embedding is possible. Thus, we can assume that there are at most three 3-face bridges, one for each pair of diagonally opposite vertices. Furthermore, any two distinct 3-face bridges conflict in the hexagon, so that at most one 3-face bridge can be embedded in the hexagon. The algorithm looks for embeddings with no 3-face bridges in the hexagon, or with exactly one 3-face bridge in the hexagon. Thus there are four subproblems to consider.

Page 375
If we choose a DFS to find the bridges, it can be organized as follows. The procedure uses a variable *nBridges* to count the number of bridges found so far. It is initially zero. We take each vertex $u \in V(TK_{3,3})$ in turn, and consider all incident edges $uv \notin E(TK_{3,3})$. Edge *uu* belongs to exactly one bridge. We store a value *B(uu)* for each edge, indicating the bridge to which *uu* belongs. If *B(uu)*=0, then *B(uu)* has not yet been assigned. If $v \in V(TK_{3,3})$, then edge *uu* is a bridge, and we assign *B(uu)*. Otherwise we call a procedure BRIDGEDFS*(u)* to build the bridge *B*. Because of the nature of a DFS, it will visit all vertices of *B* before returning, and will explore only edges of *B*. For each edge $xy \in B$, *B(xy)* is assigned to be the current value of *nBridges*. Each time it encounters a vertex of *TK*3,3, it has found a vertex of attachment. For each bridge, a list of vertices of attachment is saved. If two bridges *B* and *B′* are both found to have exactly two vertices of attachment, and they are the same two vertices, then *B* and *B′* are combined into a single bridge. The vertices of attachment are later used to determine the compatible faces, using the previous lemmas, and to sort the adjacency list of each *u*.

Algorithm 13.6.1: CONSTRUCTBRIDGES (*G,TK*3, 3)
comment: Construct all bridges of *G* with respect to *TK*3, 3
nBridges←0
for each $u \in V(TK_{3,3})$

```
do { for each edge uv such that uv ∉ E(TK_{3,3})
     do { if B(uv) = 0 then
          { comment: the bridge of uv has not been visited
            nBridges ← nBridges + 1
            if v ∈ V(TK_{3,3})
              then B(uv) ← nBridges
              else BRIDGEDFS(v) }
   comment: all bridges incident on u have now been constructed
```

We can now present an algorithm for determining the conflicts of bridges in a face *F*. Let (*u*1, *u*2,..., *uk*) denote the facial cycle of *F*. We assign a numbering to the facial cycle, such that *ui* is numbered *i*. This defines an ordering *u*1 < *u*2 < ... < *uk*. For each bridge *B* that is compatible with *F,* we sort the vertices of attachment according to this ordering. The purpose of this is to determine whether the vertices of attachment of each bridge lie completely within an interval of all other bridges. Let *b*min denote the smallest vertex of attachment of bridge *B,* and let *b*max denote the largest. The algorithm then walks along the facial cycle from *uk* to *u*1 and sorts the adjacency list of each *ui*. The edges incident on *ui* can be divided into the following three classes:

1. Edges *uiu* belonging to a bridge *B* such that *ui* = *b*min
2. Edges *uiu* belonging to a bridge *B* such that *b*min < *ui* < *b*max
3. Edges *uiu* belonging to a bridge *B* such that *ui* = *b*max

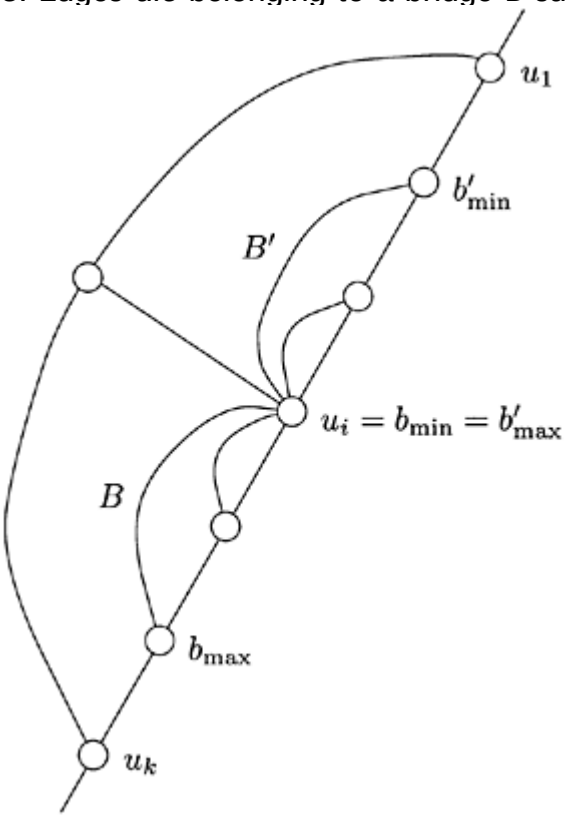


FIGURE 13.38
Determining whether bridges conflict

Refer to Figure 13.38. The adjacency list is ordered so that edges in the first class precede those in the second class, which precede those in the third class, and so that edges of each bridge are contiguous in each of the three classes. The edges in the first class are further sorted so that if *uiu* and *uiu′* belong to bridges *B* and *B′,* respectively, where $b_{\max} < b'_{\max}$, then *uiu* precedes *uiu′*. If $b_{\max} = b'_{\max}$, then *uiu* precedes *uiu′* if *B* has more vertices of attachment. The edges in the third class are further sorted so that if *uiu* and *uiu′* belong to bridges *B* and *B′,* respectively, where $b_{\min} < b'_{\min}$, then *uiu* precedes *uiu′*. If $b_{\min} = b'_{\min}$, then *uiu* precedes *uiu′* if *B* has fewer vertices of attachment.

In Figure 13.38 the *u*1*uk*-path of the facial cycle is drawn approximately vertically, and the bridges are placed to the left of the path. With this representation, the ordering of the adjacency lists appears as a clockwise circle drawn at each

Page 377

ui. If *ui*=*b*min for some bridge *B,* there can be several edges *uiu* belonging to bridge *B*. The last such edge is saved as *B*min. Similarly, if *ui*=*b*max for some bridge *B,* the first edge *uiu* of *B* is saved as *B*max.

The algorithm then walks along the facial cycle from *uk* to *u*1. Every edge *uiu* such that *B(uiu)* is compatible with *F* is placed on a stack. When *B*min, the last edge of bridge *B* is encountered, all edges of *B* are removed from the stack, and conflicts with *B* are determined. The algorithm stores a linked list of conflicting bridges for each bridge *B*.

Algorithm 13.6.2: BRIDGECONFLICTS*(F)*

comment:
- Determine the conflicts among bridges incident on face F
- The facial cycle of F is $(u_1, u_2, \ldots, u_k)$
- The adjacency list of each u_i has been sorted

for *ui*←*uk***downto** *u*1

do
- **for each** edge u_iv such that $u_iv \notin E(TK_{3,3})$
  - **do**
    - $B \leftarrow B(u_iv)$
    - **if** B is not compatible with face F **go to** $L1$
    - place u_iv on *Stack*
    - **if** $u_iv = B_{\min}$
      - **then**
        - let u_j be the vertex of attachment of $B_{\max}$
        - $u_\ell w \leftarrow$ top of *Stack*
        - **while** $u_\ell w \neq B_{\max}$ **do**
          - **if** $B(u_\ell w) = B$
            - **then** remove $u_\ell w$ from *Stack*
            - **else** **if** $u_\ell \neq u_i$ **and** $u_\ell \neq u_j$ **then** B and $B(u_\ell w)$ conflict
          - $u_\ell w \leftarrow$ next edge on *Stack*
        - remove $u_\ell w$ from *Stack*
    - $L1:$

We prove that the algorithm works. Suppose that *B* is a bridge with *b*max=*ui* and *b*min=*uj,* with extreme edges *B*max=*uiu* and *B*min=*ujw*. All vertices of attachment of *B* lie between *ui* and *uj*. If no other bridge has an attachment *uℓ* here, such that *ui*≠*uℓ*≠*uj,* then *B* does not conflict with other bridges. Consider the point in the algorithm when *B*min is reached. The algorithm will have stacked each edge of *B* incident on the facial cycle, including *B*min. It then removes all these edges. If there is no *uℓ* between *ui* and *uj,* no conflicts are discovered. But if *B′* is a bridge with a vertex of attachment *uℓ* in this range, then an edge *uℓx* of *B′* will have been stacked after *B*max and before *B*min. Since the

Page 378

edges of *B′* are still on the stack while edges of *B* are being removed, $b'_{\min} \leq b_{\min}$. If $b'_{\min} < b_{\min}$, then *B* and *B′* are in conflict, and this is discovered. If $b'_{\min} = b_{\min}$, then since *B*min precedes $B'_{\min}$ in the adjacency lists, we know that $b'_{\max} \geq b_{\max}$. If $b'_{\max} > b_{\max}$, then *B* and *B′ a*re in conflict, and this is discovered. Otherwise $b'_{\max} = b_{\max}$ and $b'_{\min} = b_{\min}$, and *uℓ* is strictly between these limits. The ordering of the adjacency list tells us that *B* has at least as many vertices of attachment as *B′*. Therefore *B* and *B′* both have a vertex of attachment strictly between *ui* and *uj*. We conclude that the bridges conflict.

Once all bridges have been constructed and all conflicts have been determined, we construct an instance of the 2-Sat problem to represent this embedding problem. The 2-Sat problem will have boolean variables corresponding to the placement of bridges, and boolean expressions to characterize the conflicts of bridges. Let the bridges be *B*1, *B*2,..., *Bm*. If there are any 0-face bridges, the embedding of *TK*3,3 cannot be extended. If *Bi* is a 1-face bridge, embeddable in face F, create a boolean variable *xi* for it. We require

xi=**true,** and consider this to mean that *Bi* is assigned to face *F*. If *Bi* is a 2-face bridge embeddable in faces *F* and *F′,* create boolean variables *xi* and *yi* for it. We consider *xi*=**true** to mean that *Bi* is assigned to *F* and *yi*=**true** to mean that *Bi* is assigned to *F′*. Since we do not want *xi* and *yi* both to be true, or both to be false, we construct the clauses

$$(x_i + y_i)(\overline{x}_i + \overline{y}_i)$$

This ensures that exactly one of *xi* and *yi* will be true.
If *Bi* is a 3-face bridge, create boolean variables *xi, yi,* and *zi* for it as above, where *zi*=**true** means that *Bi* is embedded in the hexagon. The 3-face bridges require special treatment. We take *zi*=**false** and *zi*=**true** as separate cases. If *zi*=**false,** *Bi* becomes a 2-face bridge, and we construct the clauses

$$(x_i + y_i)(\overline{x}_i + \overline{y}_i)$$

to represent this bridge. If *zi*=**true,** *Bi* becomes a 1-face bridge, and we require *xi*=*yi*=**false**.
If *Bi* and *Bj* are bridges that conflict in a face *F,* suppose without loss of generality that *xi* represents the assignment of *Bi* to face *F,* and that *wj* represents the assignment of *Bj* to face *F,* where *wj* is one of *xj, yj,* or *zj*. We then construct the clause

$$(\overline{x}_i + \overline{w}_j)$$

to ensure that at most one of *Bi* and *Bj* can be placed in face *F*.
When a variable is required to have a certain value (e.g., *xi*=**true**), we construct clauses

$$(x_i + x_0)(x_i + \overline{x}_0)$$

Page 379
where *x*0 is an additional boolean variable. Notice that this can only be satisfied if *xi*=**true**. If *xi*=**false** is required, we construct the clauses

$$(\overline{x}_i + x_0)(\overline{x}_i + \overline{x}_0)$$

which can only be satisfied if *xi*=**false**.
Thus, we can represent all conflicts and all placements of bridges by instances of 2-Sat. Suppose that an algorithm for 2-Sat finds a solution satisfying the constraints. If *B*1, *B*2,..., *Bk* are the bridges assigned to face *F,* they are all mutually non-conflicting bridges. Consequently, all vertices of attachment of any *Bi* lie in an interval determined by two vertices of attachment of every *Bj*. If each *Bi* has a planar embedding in *F,* then they all have a common planar embedding in *F,* and conversely. If there is no common planar embedding of the bridges in *F,* then some bridge *Bi* has no planar embedding in *F*. It then follows that *Bi* cannot be embedded in any other face *F′*. Thus, we can complete the projective embedding of *G* by determining whether there is a planar embedding of the bridges in *F*. We construct a graph *G(F)* which is the union of the facial cycle of *F,* and all bridges *B*1,*B*2,..., *Bk* assigned by 2-Sat to *F*. We add one additional vertex *u*0, joined to every vertex of the facial cycle, in order to distinguish an “inside” and “outside” for *F*. We have:
*LEMMA 13.23 G(F) is planar if and only if bridges B*1, *B*2,..., *Bk have a common embedding in F*.
We can now present the algorithm for projective planarity.

Page 380
Algorithm 13.6.3: PROJECTIVEPLANARITY(*G,TK*3,3)
comment: { Given a graph G with a subgraph $TK_{3,3}$, determine whether G is projective.
let *n* and *ε* denote the number of vertices and edges of *G*
if *ε*>3*n*−3
then return *(NonProjective)*
construct the bridges *B*1, *B*2,..., *Bm* with respect to *TK*3,3
for each embedding of *TK*3,3

```
classify the bridges as 0-face, 1-face, 2-face, or 3-face
if there is a 0-face bridge go to L1
determine all conflicts of bridges
construct the clauses representing all conflicts of bridges
wlog, assume that B1, B2 and B3 are 3-face bridges
z1, z2, z3 ← false
for i ← 0 to 3
do { construct the clauses representing B1, B2 and B3
     solve the resulting 2-Sat problem
     if there is no solution go to L2
     for each face F of TK3,3 do
     do { take all bridges assigned to F, construct graph G(F)
          if G(F) is non-planar go to L2
     comment: we now have a projective embedding
     return (Projective)
     L2 :
     zi ← false
     zi+1 ← true
L1 :
return (NonProjective)
```

The bridges *B*1, *B*2,..., *Bm* are constructed using a BFS or DFS. This takes *O(n)* steps, since $\varepsilon \leq 3n-3$. There are six embeddings of *TK*3,3 that are considered. For each embedding, the bridges are classified according to the faces they can be assigned to. The conflicts between bridges are then calculated. As the number of bridges *m* is bounded by *n,* the number of conflicts is at most $O(n2)$. An instance of 2-Sat is then constructed with at most $3m+1$ variables and at most $4m+m(m-1)$ clauses. This can be solved in $O(n2)$ time. If a solution is found, a planar embedding algorithm must be used for each face to find the actual embedding. If a linear or quadratic planarity algorithm is used, the result is at most $O(n2)$ steps to complete the projective embedding. The result is a $O(n2)$

algorithm for finding a projective embedding of *G,* if one exists, when we are given a *TK*3,3in *G*.

Now it would be possible to consider the embeddings of *TK*5 in a similar way, and construct the bridges with respect to each embedding of *TK*5, etc. There are 27 embeddings of *TK*5 in the projective plane. However, there is an easier way. In Section 12.3 we found that most graphs containing a subgraph *TK*5 also contain *TK*3,3 and that a simple breadth-first search algorithm can find a *TK*3,3, given a *TK*5. Thus, if we are given a *TK*5 in *G,* we first try to find a *TK*3,3 in its place, and then use Algorithm PROJECTIVEPLANARITY() to extend it to *G*.

If *TK*5 cannot be extended to *TK*3,3, then the structure of *G* is limited. Let {*u*1,*u*2,*u*3,*u*4,*u*5} be the corners of *TK*5. Then *G*−{*u*1,*u*2,*u*3,*u*4,*u*5} is a disconnected graph. Each component is adjacent to exactly two of the corner vertices of *TK*5. Let Gij denote the subgraph induced by all components adjacent to *ui* and *uj,* together with all edges connecting them to *ui* or *uj*. *Gij* is called a *K*5-*component* of *G*. Notice that ui and uj are vertices of *Gij,* and that $E(G) = \cup_{i,j} E(G_{ij})$. An augmented *K*5-*component* is the graph G_{ij}^{+} with the additional edge *uiuj;* namely, $G_{ij}^{+} = G_{ij} + v_i v_j$. We have the following theorem:

*THEOREM 13.24 Suppose that G has a subgraph TK*5 *which cannot be extended to TK*3,3. *Then G is projective if and only if all augmented K*5*-components* G_{ij}^{+} *are planar*.

The proof of this theorem is left as an exercise. A consequence of it is that algorithms for projective planarity can focus on *TK*3,3 subgraphs, which have fewer embeddings. A similar, but more complicated result holds for toroidal graphs containing a *TK*5 which cannot be extended to *TK*3,3.

Exercises

13.6.1 Show that a graph can be embedded on the projective plane if and only if it can be embedded on the Möbius band.

13.6.2 Show that *K*3,3 has six hamilton cycles. If *C*=(1,2,5,6,3,4) is a hamilton cycle, show that all hamilton

cycles can be obtained by successively applying the permutation (1)(3,5)(2,4,6) to *C*.
13.6.3 Show how to find a projective rotation system for a graph *G* containing *TK*3,3, When the algorithm PROJECTIVEPLANARITY() determines that *G* is projective. *Hint:* Use the projective embedding of *K*3,3 in Figure 13.39.
13.6.4 Prove Theorem 13.24.

Page 382

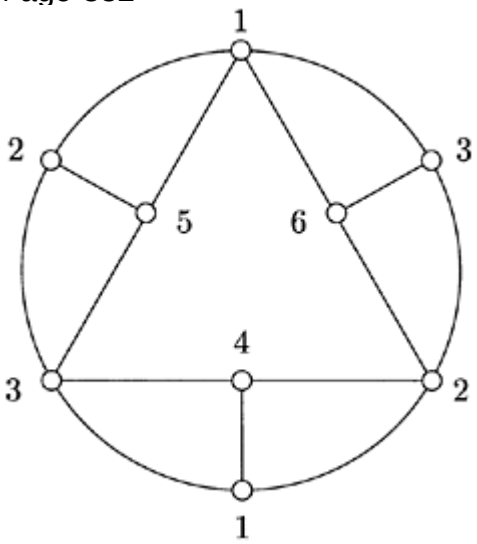


FIGURE 13.39
A projective embedding of *K*3,3

13.7 Heawood's map coloring theorem

We conclude this chapter with a discussion of Heawood's map coloring theorem. The 4-color theorem states that $\chi(G) \leq 4$, for graphs of genus zero. Heawood's map coloring theorem gives an analogous result for graphs of genus one or more.

LEMMA 13.25 Let $n \geq 3$. *Then* $g(K_n) \geq \lceil (n-3)(n-4)/12 \rceil$.

PROOF By Lemma 13.6, $\varepsilon(Kn) = n(n-1)/2 \leq 3n+6(g-1)$. Solving for g gives the result.

THEOREM 13.26 **(Heawood's theorem)** *Let G be a graph on n vertices with genus* $g \geq 1$. *Then* $\chi(G) \leq \lfloor \frac{1}{2}(7+\sqrt{1+48g}) \rfloor$.

PROOF Let $\chi(G) = k$. If G is not a critical graph, then it contains a critical subgraph. Since a k-critical graph has minimum degree at least $k-1$, we conclude that the sum of degrees of G is at least $(k-1)n$, so that $\varepsilon \geq (k-1)n/2$. Lemma 13.6 gives $\varepsilon \leq 3n+6(g-1)$. These two inequalities together give

$$k \leq 7 + \frac{12(g-1)}{n},$$

with equality only if both inequalities above are equalities. Now $g \geq 1$ so that for fixed g, this is a non-increasing function of n, so that $\chi(G)$ will be bounded.

Page 383
This arises because the number of edges in a k-critical graph increases as $kn/2$, whereas the maximum number of edges in a graph embedded in a surface of genus g increases as $3n$. We also know that $k \leq n$, an increasing function. Therefore the largest possible value of k is when $k = n = 7+12(g-1)/n$. The equation then becomes $n^2 - 7n - 12(g-1) = 0$, which gives the solution $n = \frac{1}{2}(7+\sqrt{1+48g})$. Since $k \leq n$ and k must be an integer, the result follows.
If $\frac{1}{2}(7+\sqrt{1+48g})$ is an integer, the inequalities used in the proof of Heawood's theorem must be equalities. This requires that $\varepsilon = (n-1)n/2$, so that $G = Kn$. The quantity $\frac{1}{2}(7+\sqrt{1+48g})$ represents the largest number of vertices that a graph can have, and still satisfy $n(n-1)/2 \leq 3n+6(g-1)$. If it is not an integer, this means that $n(n-1)/2 < 3n+6(g-1)$, so that a complete graph on $n = \lfloor \frac{1}{2}(7+\sqrt{1+48g}) \rfloor$ vertices will not be a triangulation. In general, we have:
THEOREM 13.27 Let G be a graph on n vertices with genus $g \geq 1$ *and chromatic number*

$\chi(G) = \lfloor \frac{1}{2}(7+\sqrt{1+48g}) \rfloor$. *Then G contains a spanning complete graph.*

PROOF Let $h = \lfloor \frac{1}{2}(7+\sqrt{1+48g}) \rfloor$. Let *G* have *n* vertices. Since $\chi(G)=h$, we know that $n \geq h$, and $\varepsilon(G) \geq n(h-1)/2 \geq h(h-1)/2$. But h is the largest integer such that $h(h-1)/2 \leq 3h+6(g-1)$. Therefore $n=h$ and *G* contains a spanning complete graph. Note that a complete graph may not triangulate the surface, so that the number of edges in a triangulation, $3h+6(g-1)$, may be larger than $h(h-1)/2$.

We conclude that the extreme value of χ will be achieved only if a complete graph with this many vertices can be embedded in the surface.

This theorem gives $\chi(G) \leq 7$ for the torus. An embedding of $K7$ in the torus is shown in Figure 13.17, so that seven colors are necessary for the torus. We say that the *chromatic number* of the torus is seven, because all toroidal graphs can be colored in at most seven colors, and seven colors are necessary for some graphs. The dual of the embedding of $K7$ on the torus is the Heawood graph. Heawood was coloring the faces of an embedding rather than the vertices of a graph, and discovered this graph. The 4-color theorem tells us that the chromatic number of the plane is four. The formula of Heawood's theorem gives the bound $\chi(G) \leq 4$ for the plane. However, the proof is invalid when $g=0$, and there are many planar graphs other than $K4$ which require four colors. Lemma 13.25 gives $g(K_n) \geq \lceil (n-3)(n-4)/12 \rceil$. A proof that $g(Kn)$ equals this bounds would mean that Kn can always be embedded in a surface of this genus. Since $\chi(Kn)=n$, the inequality of Heawood's theorem could then be replaced by an equality. Calculating the genus of Kn was accomplished by a number of people

Page 384

(Heffter, Gustin, Ringel, Youngs, and Mayer) over many years. We state the result, without proof, as the following theorem.

THEOREM 13.28 **(Ringel-Youngs)** *Let* $n \geq 3$. *Then* $g(K_n) = \lceil (n-3)(n-4)/12 \rceil$.

A complete proof of this result can be found in the survey paper of WHITE [124]. A consequence is the following:

THEOREM 13.29 **(Heawood map coloring theorem)** *The chromatic number of an orientable surface of genus* $g \geq 1$ is $\lfloor \frac{1}{2}(7+\sqrt{1+48g}) \rfloor$.

The corresponding results for non-orientable surfaces of genus $\overline{g} \geq 1$ are as follows. Corresponding to Lemma 13.25 is the bound $\overline{g}(K_n) \geq \lceil (n-3)(n-4)/6 \rceil$. Corresponding to Heawood's theorem is the bound $\chi(G) \leq \lfloor \frac{1}{2}(7+\sqrt{1+24\overline{g}}) \rfloor$, which is proved in an analogous way. Again, the graphs which meet the bound are the complete graphs. The non-orientable version of the Ringel-Youngs theorem is the following:

THEOREM 13.30 Let $n \geq 5$. *Then* $\overline{g}(K_n) = \lceil (n-3)(n-4)/6 \rceil$, *except that* $\overline{g}(K_7) = 3$.

The formula $(n-3)(n-4)/6$ gives $\overline{g}(K_7) \geq 2$. However, $\overline{g}(K_7) = 3$, as $K7$ does not embed on the Klein bottle. The map coloring theorem for non-orientable surfaces is then:

THEOREM 13.31 The chromatic number of a non-orientable surface of genus $\overline{g} \geq 1$ *is* $\lfloor \frac{1}{2}(7+\sqrt{1+24\overline{g}}) \rfloor$, *except that the chromatic number of the Klein bottle is 6.*

Exercises

13.7.1 Let t be a positive integer, and let $tK3$ denote the graph with three vertices, and t parallel edges connecting each pair of vertices, so that $\varepsilon(tK3)=3t$. Consider embeddings of $tK3$ in which there are no digon faces. Show that $g(tK3) \geq (t-1)/2$ and that $\overline{g}(tK_3) \geq t-1$.

13.7.2 Show that $g(tK3)=(t-1)/2$ and $\overline{g}(tK_3) = t-1$ by constructing embeddings of $tK3$ on the appropriate surfaces.

Page 385

13.7.3 Let *G* be a graph with *n* vertices and genus *g*, and let n_k denote the number of vertices of degree *k*. Suppose that $n_1=n_2=0$. Construct an inequality satisfied by n_3, n_4,...in terms of *g*, using the number of edges in a triangulation. Do the same for $\overline{g}(G)$.

13.7.4 The *maximum genus* of a graph *G* is the largest value *g* such that *G* has a 2-cell embedding on a surface of genus *g*. If g' is the maximum genus of a graph *G* on *n* vertices, use the Euler-Poincaré formula to show that $g' \leq (\varepsilon-n+1)/2$. Find the maximum orientable genus of $K4$.

13.7.5 Show that $K7$ does not embed on the Klein bottle.

13.8 Notes

An excellent source book related to topology and geometry is HILBERT and COHN-VOSSEN [61]. It is perhaps one of the best and most readable mathematics books ever written. Proofs of the Dehn-Heegard theorem can be found in FRÉCHET and FAN [44] and in STILLWELL [109]. Both contain very readable accounts of combinatorial topology. Fréchet and Fan call the Euler-Poincaré formula *Descartes' formula*.
There are excellent chapters in DIESTEL [35] and ZIEGLER [129] on the graph minor theorem. The minor order obstructions for the projective plane were found by ARCHDEACON [6]. MYRVOLD [90] has found over 200,000 topological obstructions for the torus, and the list may not be complete.
An excellent source for graphs and surfaces is the book by MOHAR and THOMASSEN [88], or the book on topological graph theory by GROSS and TUCKER [54].
THOMASSEN [113] has proved that Graph Genus is NP-complete.
The algorithm for drawing graphs on the torus, given a toroidal rotation system is from KOCAY, NEILSON, and SZYPOWSKI [76]. It is adapted from Read's algorithm READ [100] for planar graphs.
Theorem 13.18 relating projective embeddings to toroidal embeddings is from FIEDLER, HUNEKE, RICHTER, and ROBERTSON [41].
The algorithm for embedding a graph on the projective plane is based on algorithms by GAGARIN [48], MOHAR [87], and MYRVOLD and ROTH [91]. Theorem 13.24 is from GAGARIN and KOCAY [49].
The survey article by WHITE [124] contains a complete proof of the RingelYoungs theorem and the Heawood map coloring theorem.

Page 386
This page intentionally left blank.

Page 387
14
Linear Programming
14.1 Introduction
Problems which seek to find a "best" configuration to achieve a certain goal are called *optimization problems*. *Programming problems* deal with determining optimal allocations of limited resources to meet given objectives. They deal with situations in which a number of resources such as manpower, materials, and land are available to be combined to yield one or more products. There are, however, restrictions imposed, such as the total number of resources available and the quantity of each product to be made. *Linear programming* deals with the class of programming problems in which these restrictions and any other relations among the variables are *linear*. In particular, the constraints imposed when searching graphs and networks for say shortest paths, matchings, and maximum flow can be expressed as linear equations.
14.1.1 A simple example
Let us consider a furniture shop that produces tables, chairs, cabinets, and stools. There are three type of machines used: table saw, band saw, and drill press. We assume that the production is continuous and each product first uses the table saw, then the band saw, and finally the drill press. We also assume that setup time for each machine is negligible. Table 14.1 shows
1. The hours required on each machine for each product
2. The profit realized on the sale of each product
We wish to determine the weekly output for each product in order to maximize profit. Let $x1$, $x2$, $x3$, and $x4$ denote the number of tables, chairs, cabinets, and stools produced per week, respectively. We want to find the values of $x1$, $x2$, $x3$, $x4$ which maximizes the profit. The available machine time is lim-

Page 388
TABLE 14.1
Data for the simple example

Machine type	Table	Chair	Cabinet	Stool	Time available
table saw	1.5	1	2.4	1	2000
band saw	1	5	1	3.5	8000
drill press	1.5	3	3.5	1	5000
profit	5.24	7.30	8.34	4.18	

ited so we cannot arbitrarily increase the output of any one product. Thus we must allocate machine hours among the products without exceeding the maximum number of machine hours available.
Consider the restriction imposed by the table saw. According to Table 14.1 it will be used a total of

$$1.5x1+x2+2.4x3+x4$$

hours per week, but can only be used for at most 2000 hours. This yields the linear inequality

$$1.5x1+x2+2.4x3+x4\leq 2000.$$

Similarly, the band saw and drill press yield the following restrictions:

$$x1+5x2+x3+3.5x4\leq 8000$$
$$1.5x1+3x2+3.5x3+x4\leq 5000$$

Furthermore, we can't produce a negative amount of a product, so we also have

$$x1\geq 0,\ x2\geq 0,\ x3\geq 0, \text{ and } x4\geq 0.$$

Now we have all the restrictions. The profit is

$$z=5.24x1+7.30x2+8.34x3+3.4.18x4.$$

Thus our goal is to solve the following linear program.

$$\text{Maximize: } z=5.24x1+7.30x2+8.34x3+3.4.18x4$$
$$\text{Subject to: } 1.5x1+x2+2.4x3+x4\leq 2000$$
$$x1+5x2+x3+3.5x4\leq 8000$$
$$1.5x1+3x2+3.5x3+x4\leq 5000$$
$$x1\geq 0,\ x2\geq 0,\ x3\geq 0,\ x4\geq 0$$

page_388

Page 389

14.1.2 Simple graphical example

We can graphically solve linear programs that involve only two variables. For example, consider the linear program

$$\text{Maximize: } z=5x1+3x2$$
$$\text{Subject to: } 3x1+5x2\leq 18$$
$$5x1+2x2\leq 11$$
$$x1\geq 0,\ x2\geq 0$$

Each of the constraints determines a half plane consisting of the set of points $(x1,x2)$ that satisfy the inequality. For example, the inequality $x1\geq 0$ determines the half plane of points lying to the right of the $x2$-axis. The intersection of these four half planes is the set of *feasible solutions* to the problem. In this example the feasible solutions form a bounded polygonal region. It is not difficult to show that such a region must be convex. (It is possible to have an unbounded region.) The four vertices of this region are easily determined by pairwise solving the four bounding equations.

1. Solving $x1=0$ and $3x1+5x2=18$ gives the vertex (0,3.6).
2. Solving $3x1+5x2=18$ and $5x1+2x2=11$ gives the vertex (1,3).
3. Solving $5x1+2x2=11$ and $x2=0$ gives the vertex (2.2,0).
4. Solving $x1=0$ and $x2=0$ gives the vertex (0,0).

The set of feasible solutions is depicted in Figure 14.1. We have also drawn the objective function $z=5x1+3x2$, when $z=3.5$, 11, and 38.5.

Consider any line segment joining points $P=(p1,p2)$ and $Q=(q1,q2)$. Let

$$z0=5p1+3p2$$

be the value of the objective function at P and let

$$z1=5q1+3q2$$

be the value of the objective function at Q. Assume without loss that $z0\leq z1$. The coordinates of any point on the line segment between P and Q is given by

$$((1-t)p1+tq1,(1-t)p2+tq2$$

for $0\leq t\leq 1$. The value of the objective function at this point is

$$zt-5((1-t)p1+tq1)+3((1-t)p2+tq2)$$
$$=(1-t)(5p1+3p2)+t(5q1+3q2)$$
$$=(1-t)z0+tz1$$

page_389

Page 390

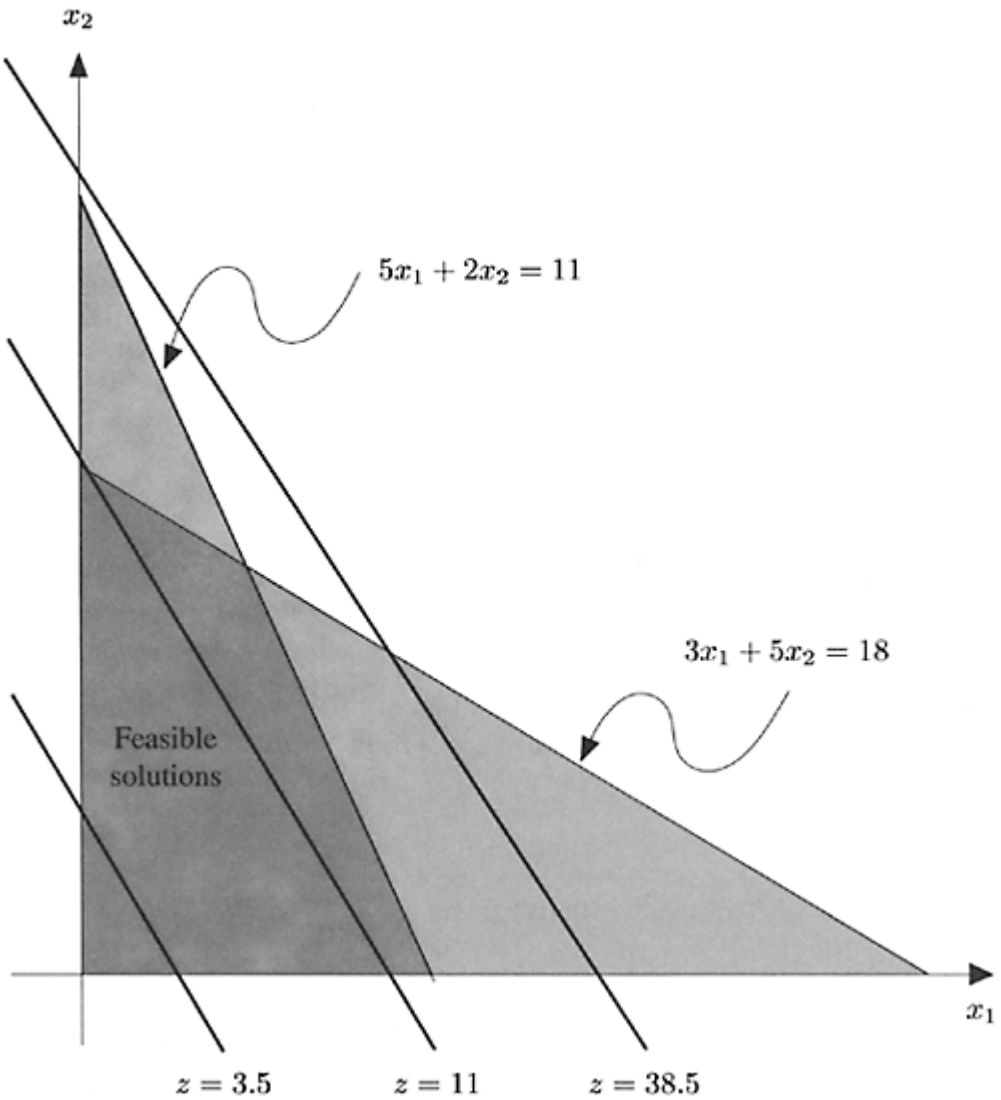


FIGURE 14.1
Graphical example

Page 391
Observe that

$$z0=(1-t)z0+tz0\leq(1-t)zt+tz1=zt$$

and

$$zt=(1-t)z0+tz1\leq(1-t)z1+tz1=z1.$$

Thus $z0\leq zt\leq z1$ for any $0\leq t\leq 1$. Therefore the maximum value of the the objective function among the points on any line segment occurs at one of the endpoints. *It follows that the maximum value of any (linear) objective function among the points of a compact region occurs at some point on the boundary.* Also, *if the boundary is a polygon, then the maximum value of the objective will occur at one of the vertices*. Hence for our example the maximum value of the objective function

$$z=5x1+3x2$$

occurs at least one of (0, 3.6), (1, 3), (2.2, 0), or (0, 0). The value at these points is 10.8, 14, 11, and 0. Thus the maximum occurs at $x1=1$, $x2=3$.
If the region of feasible solutions is unbounded, then there may be no point in the region for which the objective function achieves its absolute maximum. Of course there is also no point for which the objective function achieves its maximum, if there are no feasible solutions.
If the objective function achieves its maximum at two adjacent vertices *P* and *Q,* and the region of feasible solutions is connected and polygonally bounded, then it will achieve its maximum at infinitely many points: namely, those lying on the line segment joining *P* and *Q*.
In a general linear program the region of feasible solutions is the intersection of the half hyper-planes determined by the linear constrains. Thus the region of feasible solutions to the general linear program is a compact convex region, bounded by facets (hyper-plane segments). That is, it is a polyhedron.1 Consequently

it follows that:
THEOREM 14.1 In a general linear program the objective function achieves its maximum either at exactly one point, at infinitely many points, or at no point. Furthermore if it does achieve a maximum, it does so on one of the vertices of the polyhedral boundary of the feasible solutions.

14.1.3 Slack and surplus variables

It is easier to work with equalities, than with inequalities because we can take advantage of linear algebra. We can convert linear inequalities into equalities by introducing surplus and slack variables.
1O.K. it is only a polyhedron if it is bounded. It could have some open sides, but the closed sides are bounded by hyper-planes.

For example, consider an inequality of the form

$$a1x1+a2x2+\ldots+atxt\leq b \qquad (14.1)$$

Given fixed assignments to the variables $x1$, $x2$,..., xt that satisfy this inequality, there will be *slack* or "room for improvement" amounting to

$$xj=b-a1x1+a2x2+\ldots+atxt\geq 0.$$

Thus introducing a new variable xj and requiring $xj\geq 0$ we obtain the equality

$$a1x1+a2x2+\ldots+atxt+xj=b$$

which is equivalent to Inequality 14.1.
The variable xj is called a *slack variable*. Similarly an inequality of the form

$$a1x1+a2x2+\ldots+atxt\geq b \qquad (14.2)$$

represents a *surplus* of

$$s'=a1x1+a2x2+\ldots+atxt-b$$

for a fixed assignment to the variables $x1$, $x2$,..., xt. Thus introducing xj' as a new variable and requiring $xj'\geq 0$ we obtain the equality

$$a1x1+a2x2+\ldots+atxt-xj'=b$$

which is equivalent to Inequality 14.2. The variable xj' is called a *surplus variable*. Adding slack and surplus variables in this way will reduce the the system of inequalities to a system of equalities and variables xi that are either unbounded or satisfy $xi\geq 0$. If xi is unbounded, we find an inequality that xi satisfies, solve for xi, and substitute to eliminate xi from the set of equations. A linear program in which all the variables are required to be non-negative and the remaining constraints are equality constraints is said to be in *standard form* or a *standard linear program*.
For example, to convert the linear program

$$\begin{gathered}\text{Maximize: } z=5x1+3x2+3x3+x4\\ \text{Subject to: } 2x2+x4=2\\ x1+x2+x4\leq 3\\ -x1-2x2+x3\geq 1\\ x1\leq 0,\ x2,\ x3\geq 0\end{gathered}$$

into standard form we introduce slack and surplus variables $x5$, $x6$, and $x7$ obtaining

$$\begin{gathered}\text{Maximize: } z=5x1+3x2+3x3+x4\\ \text{Subject to: } 2x2+x4=2\\ x1+x2+x4+x5=3\\ -x1-2x2+x3-x6=1\\ x1+x7=0\\ x2,\ x3,\ x5,\ x6,\ x7\geq 0\end{gathered}$$

Now variables $x1$ and $x4$ are unbounded, so we solve for them

$$x1=-x7$$
$$x4=2-2x2$$

Substituting, we obtain

$$\begin{gathered}\text{Maximize: } z=x2+3x3-35x7+2\\ \text{Subject to: } -x2+x5-x7=1\\ -2x2+x3-x6=x7=1\\ x2,x3,x5,x6,x7\geq 0.\end{gathered}$$

Finally, to convert to a problem of minimization, we set

$$Z=2-z$$

and we have

Minimize: $Z=-x2-3x3+35x7$
Subject to: $-x2+x5-x7=1$
$-2x2+x3-x6+x7=1$
$x2,x3,x5,x6,x7\geq 0.$

a linear program in standard form. In matrix form we set

$$X=[x2,x3,x5,x6,x7]T$$
$$c=[-1,-3,0,0,35]T$$
$$b=[1,1]T$$
$$A=\begin{bmatrix}-1&0&1&0&-1\\-2&1&0&-1&1\end{bmatrix}$$

and we see that the original linear program is equivalent to

Minimize: $Z=cTX$
Subject to: $AX=b$
$X\geq 0$

Exercises

14.1.1 Solve the following graphically and shade the region representing the feasible solutions:

Maximize: $z=x1+1.5x2$
Subject to: $2x1+3x2\leq 6$
$x1+4x2<4$
$x1,x2\geq 0$

14.1.2 Solve the following graphically and shade the region representing the feasible solutions:

Minimize: $Z=6x1+4x2$
Subject to: $2x1+x2\geq 1$
$3x1+4x2\geq 1.5$
$x1,x2\geq 0$

14.1.3 Carefully examine Exercises 14.1.1 and 14.1.2. How are the solutions related? They form what is called a pair of dual problems. Note that they involve the same constants, but in a rearranged order.

14.1.4 Put the following linear program into standard form:

Maximize: $z=2x1+3x2+5x3$
Subject to: $3x1+10x2+5x3\leq 15$
$33x1-10x2+9x3<33$
$x1+2x2+x3\geq 4$
$x1,x2\geq 0$

14.2 The simplex algorithm

14.2.1 Overview

The simplex algorithm can be used to solve linear programs of the form

Minimize: $Z=cTX$ (14.3)
Subject to: $AX=b$
$X\geq 0$

There are three phases to the algorithm.

Phase 0: Find a basis solution or show that the linear program is infeasible.
Phase 1: Find a basis feasible solution or show that the linear program is infeasible.
Phase 2: Improve the basis feasible solution until
1. it's optimal, or
2. the linear program can be shown to be unbounded.

14.2.2 Some notation

In order to study the linear program in Equation 14.3 we first introduce some notation. Denote the columns of the m by n matrix A by

$$A=[A1,A2,A3,...,An].$$

Without loss, assume that the rank of A is m. Suppose

$$\mathcal{B}=\{A_{j_1},A_{j_2},A_{j_3},\ldots A_{j_m}\}$$

is a set of m linearly independent columns of A. Then the m by m matrix

$$B = [A_{j_1}, A_{j_2}, A_{j_3}, \ldots A_{j_m}]$$

is nonsingular. Thus $B-1$ exists. Consequently, $B-1A$ contains the identity matrix on columns $j1$, $j2$, $j3$,..., jm of A. Let $XB=B-1b$. Then X given by

$$X[j] = \begin{cases} X_B[i] & \text{if } j = j_i \\ 0 & \text{if not} \end{cases}$$

satisfies

$$AX=b$$

and is called the *basis solution* given by B. If $XB \geq 0$, then X also satisfies

$$X \geq 0$$

Page 396

with the remaining constraints of the linear program given in Equation 14.3. Thus in this case we call X a *basis feasible solution*. Also corresponding to B we define cB by

$$cB[i]=c[ji], \text{ for } i=1,2,\ldots,m$$

The value Z of the objective function at X is

$$Z = c^T X = c_B^T X_B$$

It is also convenient to set

$$z_j = c_B^T Y_j;$$

where $Yj=B-1Aj$. Using this notation we can safely identify XB with X and refer to XB as the basis solution.

14.2.3 Phase 0: finding a basis solution

A basis solution if one exists can be found by *pivoting*. Pivoting is also known as Gaussian elimination or row reduction. To pivot on non-zero entry aij of the m by n matrix A we replace row k by

$$a_{ij}[a_{k1}, a_{k2}, \ldots, a_{kn}] - a_{kj}[a_{i1}, a_{i2}, \ldots, a_{in}]$$

for $k \neq i$ and we replace row i by

$$\frac{1}{a_{ij}}[a_{i1}, a_{i2}, \ldots, a_{in}].$$

The result is that column j becomes

$$[0, 0, \ldots, 0, \underbrace{1}_{j^{th}}, 0, 0, \ldots, 0]^T.$$

We record this procedure as Algorithm 14.2.1.

Algorithm 14.2.1: PIVOT*(i,j)*

for $k \leftarrow 1$ **to** m

$$\textbf{do} \begin{cases} \textbf{if } k \neq i \\ \quad \textbf{then} \begin{cases} \textbf{for } h \leftarrow 0 \textbf{ to } n \\ \quad \textbf{do } A[k,h] \leftarrow A[i,j] \cdot A[k,h] - A[k,j] \cdot A[i,h] \end{cases} \\ \quad \textbf{else} \begin{cases} \textbf{for } h \leftarrow 0 \textbf{ to } n \\ \quad \textbf{do } A[k,h] \leftarrow A[k,h]/A[i,j] \end{cases} \end{cases}$$

Page 397

Thus to find a basis solution we iteratively select a non-zero entry a_{ij_i} in row i and pivot on a_{ij_i} for i=1,2, ...,m. That is, we have selected the linearly independent columns

$$\{A_{j_1}, A_{j_2}, A_{j_3}, \ldots, A_{j_m}\}$$

of A and have determined the basis solution XB; where

$$B = [A_{j_1}, A_{j_2}, A_{j_3}, \ldots A_{j_m}].$$

14.2.4 Obtaining a basis feasible solution

Suppose that $A=[A1,A2,...,An]$ has rank m and that X is a feasible solution to the linear program in Equation 14.3. Let $p \leq n$ be the number of positive variables in X. Without loss we may assume that the first p variables are positive. Then

$$X = [x_1, x_2, \ldots, x_p, \underbrace{0, 0, \ldots, 0}_{n-p}]^T$$

and so

$$\sum_{j=1}^{p} x_j A_j = b. \tag{14.4}$$

If the *p* columns *A*1, *A*2,..., *Ap* are linearly independent, then $p \leq m$, and there exists an additional $m-p$ columns of *A* whose inclusion with the first *p* are a linear independent set. Thus we can form a basis solution with $m-p$ zeros.

If the *p* columns are linearly dependent, then there exists *aj* not all zero such that

$$\sum_{j=1}^{p} \alpha_j A_j = 0$$

Let *Ar* be any column for which $ar \neq 0$, $j=1,2,\ldots,p$. Then

$$A_r = - \sum_{\substack{j=1 \\ j \neq r}}^{p} \frac{\alpha_j}{\alpha_r} A_j \tag{14.5}$$

Substituting Equation 14.5 into Equation 14.5, we obtain

$$\sum_{\substack{j=1 \\ j \neq r}}^{p} (x_j - x_r \frac{\alpha_j}{\alpha_r}) A_j = b \tag{14.6}$$

Thus we have a solution with at most $p-1$ non-zero entries. However, we are not certain that they are non-negative. We need

$$x_j - x_r \frac{\alpha_j}{\alpha_r} \geq 0$$

for $j=1,\ldots,p$, $j \neq r$. If $aj=0$, then this is automatically satisfied. If $aj \neq 0$, then dividing by *aj* gives

$$\frac{x_j}{\alpha_j} - \frac{x_r}{\alpha_r} \geq 0 \qquad \text{if } \alpha_j > 0$$

and

$$\frac{x_j}{\alpha_j} - \frac{x_r}{\alpha_r} \leq 0 \qquad \text{if } \alpha_j < 0$$

Thus we may select *Ar* such that

$$\frac{x_r}{\alpha_r} = \text{MIN}_j \{ \frac{x_j}{\alpha_j} : \alpha_j > 0 \}$$

or such that

$$\frac{x_r}{\alpha_r} = \text{MAX}_j \{ \frac{x_j}{\alpha_j} : \alpha_j < 0 \}.$$

Then each entry in Equation 14.6 will be non-negative and a feasible solution with no more than $p-1$ non-zero variables has been found. We can continue this process of selecting columns *Aj* until $p \leq m$ in which case a basis feasible solution has been found.

14.2.5 The tableau

The *initial tableau* for the linear program in Equation 14.3 is the array

$$\left[\begin{array}{c|c} A & b \\ \hline c^T & 0 \end{array}\right]. \tag{14.7}$$

Note that other authors use a more elaborate tableau, but this is sufficient. Suppose *B* is the submatrix of *A* corresponding to the basis *B* as in Section 14.2.2. For the purpose of explanation assume *B* is the first *m* columns of *A;* then the tableau has the form

$$\left[\begin{array}{c|ccc|c} B & \cdots & A_j & \cdots & b \\ \hline c_B^T & \cdots & c_j & \cdots & 0 \end{array}\right];$$

where *Aj* is the *jth* column of *A*. Multiplying the tableau with

$$\left[\begin{array}{c|c} B^{-1} & 0 \\ \hline -c_B^T B^{-1} & 1 \end{array}\right]$$

Page 399
we obtain the new tableau

$$\left[\begin{array}{c|ccc|c} I & \cdots & Y_j & \cdots & X_B \\ \hline 0 & \cdots & c_j - z_j & \cdots & -z \end{array}\right]. \qquad (14.8)$$

This is because *Yj*=*B*−1*Aj*, *XB*=*B*−1*b*, and $z_j = c_B^T Y_j$ as defined in Section 14.2.2. Thus selecting a new basis solution is equivalent to pivoting on entries of the first *m* rows of the tableau 14.7 to obtain a tableau similar to the tableau 14.8. (The identity matrix need not be among the first columns.) Thus in the process of selection of a basis the values of *cj*−*zj*, *z*, and *XB* are also easily determined.
An algorithm for the Phase 0 portion of the simplex algorithm that takes as input a tableau for a linear program whose adjacency matrix has *m* rows and *n* columns is provided in Algorithm 14.2.2.

Algorithm 14.2.2: PHASE0 *(Tableau, m,n)*

```
Infeasible←false
for r←1 to m
do { c ← 1
     while Tableau[r, c] = 0 do c ← c + 1
     if c > n
       then { if Tableau[n + 1] ≠ 0
                then { comment: the linear program is infeasible
                       Infeasible ← true
                       exit
              else { comment: { the linear program has
                              { rank < M – shift up the rows
                     for i ← r + 1 to m
                       do for j ← 1 to n + 1
                       do Tableau[i − 1, j] ← Tableau[i, j]
                     m ← m − 1
                     r ← r − 1 }
       else { PIVOT(r, c)
              pivots[r] ← c }
```

14.2.6 Phase 2: improving a basis feasible solution

Let *XB*=*B*−1*b* be a basis feasible solution of the linear program given in Equation 14.3. The value of the objective function at *XB* is

$$Z = c_B^T X_B$$

Page 400
Some questions naturally arise:
Can we find another basis feasible solution with better Z?
Furthermore, can we do this by changing only one column of B?
Can we remove one column Brof B and replace it with a column Aj of A and get a smaller value Z of the objective function Z=*cTX*?
Column *Aj* is a linear combination of the columns in *B* because *B* is a nonsingular *m* by *m* submatrix of *A*. Thus

$$A_j = \sum_{i=1}^{m} y_{ij} B_i. \tag{14.9}$$

Then *Aj* can replace any *Br* for which

$$yrj \neq 0.$$

because the new set of vectors

$$\{B1, B2, \ldots, Br-1, Aj, Br+1, Br+2, \ldots, Bm\}$$

will be linearly independent. Let

$$B^* = [B_1, B_2, \ldots, B_{r-1}, A_j, B_{r+1}, B_{r+2}, \ldots, B_m].$$

Then X_{B^*} is a basis solution, but it may not be feasible. Observe from Equation 14.9 that

$$B_r = \frac{1}{y_{rj}} A_j - \sum_{\substack{i=1 \\ i \neq r}}^{m} \frac{y_{ij}}{y_{rj}} B_i$$

Also

$$b = BX_B = \sum_{i=1}^{m} X_B[i] B_i = \sum_{\substack{i=1 \\ i \neq r}}^{m} X_B[i] B_i + X_B[r] B_r$$

So, substituting we have:

$$b = \sum_{\substack{i=1 \\ i \neq r}}^{m} X_B[i] B_i + X_B[r] \left(\frac{1}{y_{rj}} A_j - \sum_{\substack{i=1 \\ i \neq r}}^{m} \frac{y_{ij}}{y_{rj}} B_i \right)$$

$$= \sum_{\substack{i=1 \\ i \neq r}}^{m} \left(X_B[i] - X_B[r] \frac{y_{ij}}{y_{rj}} \right) B_i + \frac{X_B[r]}{y_{rj}} A_j$$

Thus

$$X_{B^*}[i] = \begin{cases} X_B[i] - X_B[r] \frac{y_{ij}}{y_{rj}} & \text{if } i \neq r \\ \frac{X_B[r]}{y_{rj}} & \text{if } i = r \end{cases} \tag{14.10}$$

is feasible if and only if

$$X_B[i] - X_B[r] \frac{y_{ij}}{y_{rj}} \geq 0 \tag{14.11}$$

and

$$\frac{X_B[r]}{y_{rj}} \geq 0. \tag{14.12}$$

Thus if *XB[r]*≠0, we see from Equation 14.12 that we must have

$$yrj \geq 0.$$

If *yij*≤0, then Equation 14.11 automatically holds. So, we need to only be concerned with coordinates *i* for which *yij*>0. When *yij*>0, the condition given by Equation 14.11 can be rewritten as

$$\frac{X_B[r]}{y_{rj}} \leq \frac{X_B[i]}{y_{ij}}.$$

Thus we need to choose that column *r* of *B* such that

$$\frac{X_B[r]}{y_{rj}} = \text{MIN}_i \left\{ \frac{X_B[i]}{y_{ij}} : y_{ij} > 0 \right\} = \theta. \tag{14.13}$$

To summarize:
We began with a nonsingular submatrix *B* of *A*

$$B = [A_{i_1}, A_{i_2}, A_{i_3}, \ldots A_{i_m}].$$

If *XB*=*B*−1*b* is a basis feasible solution to the linear program given in Equation 14.3, then we selected an arbitrary column *Aj* of *A,* not in *B* and wrote

Aj=*BYj*,

a linear combination of the columns of *B,* where

Yj=[*y*1*j*,*y*2*j*,...,*ymj*].

Page 402

If some *yij*>0, there is a column *Br* of *B,* which we can replace with *Aj* to get a basis feasible solution $X_{B^\star}$. Equation 14.13 shows how to select *Br* so that this is possible.
Now what about the value of the objective function—is it better? Let

$$B^\star = [B_1^\star, B_2^\star, \ldots, B_m^\star]$$

be the new matrix obtained by replacing *Br* with *Aj*. That is $B_i^\star = B_i$, for *i*≠*r* and $B_r^\star = A_j$. The new basis feasible solution is $X_{B^\star}$ given in Equation 14.10. The objective function corresponding to $B^\star$ is

$$Z^\star = c_{B^\star}^T X_{B^\star}$$

where

$$c_{B^\star}[i] = \begin{cases} c_B[i] & \text{if } i \neq r \\ c_j & \text{if } i = r \end{cases}$$

Therefore

$$Z^\star = \sum_{\substack{i=1 \\ i \neq r}}^{m} \left(X_B[i] - X_B[r]\frac{y_{ij}}{y_{rj}} \right) c_B[i] + \frac{X_B[r]}{y_{rj}} c_j$$

$$= \sum_{i=1}^{m} \left(X_B[i] - X_B[r]\frac{y_{ij}}{y_{rj}} \right) c_B[i] + \frac{X_B[r]}{y_{rj}} c_j,$$

$$\text{because } X_B[i] - X_B[r]\frac{y_{ij}}{y_{rj}} = 0 \text{ when } i = r$$

$$= Z - \frac{X_B[r]}{y_{rj}} \sum_{i=0}^{m} y_{ij} c_B[i] + \frac{X_B[r]}{y_{rj}} c_j$$

$$= Z + \frac{X_B[r]}{y_{rj}} (c_j - c_B^T Y_j)$$

Setting $z_j = c_B^T Y_j = c_B^T B^{-1} A_j$ we have

$$Z^\star = Z + \frac{X_B[r]}{y_{rj}} (c_j - z_j) = Z + \theta(c_j - z_j)$$

Therefore if we can find *Aj* such that

cj−*zj*<0

and at least one *yij*>0, then it is possible to replace one of the columns of the columns of *B* by *Aj* and obtain a new value $Z^\star$ of the objective function satisfying

$$Z^\star \leq Z.$$

Page 403

If the given basis solution is not degenerate, then

$$Z^\star < Z.$$

In terms of the tableau given in Equation 14.8, this means we can find a column *j* with a negative entry in last row and a positive entry *yij* in row *i*. For each positive entry *yij* compute the ratio

$$\theta_i = \frac{X_B[i]}{y_{ij}}$$

and chose *i* so that *θi* is smallest. Recall that

$$[XB[1], XB[2], \ldots, XB[m], -Z]T$$

is the last column of the tableau. Then pivoting on *yij* produces a new tableau with smaller *Z*. Note that $-Z$ is the entry in the last row and column of the tableau.

14.2.7 Unbounded solutions

In Section 14.2.6, given a basis feasible solution $XB=B-1b$ we found a column *Aj* that had at least one $yij>0$, $i=1,2,\ldots,m$ where

$$Yj=B-1Aj.$$

For this column, we found a column *Br* in *B* which when replaced by *Aj*, resulted in a basis feasible solution $X_{B^\star}$. The value of the objective function for $X_{B^\star}$ was

$$Z^\star = Z + \theta(c_j - z_j)$$

Let us consider a column *Aj* for which $yij \leq 0$ for each $i=1,2,\ldots,m$. We have

$$b = BX_B = \sum_{i=1}^{m} X_B[i]B_i$$

with value of the objective function equal to $Z = c_B^T X_B$.

Adding and subtracting *θAj* for any theta yields

$$b = BX_B - \theta A_j + \theta A_j$$

$$= \sum_{i=1}^{m} X_B[i]B_i - \theta A_j + \theta A_j$$

Page 404

but

$$A_j = BY_j = \sum_{i=1}^{m} y_{ij}B_j$$

So substituting we obtain:

$$b = \sum_{i=1}^{m} (X_B[i] - \theta y_{ij})B_i + \theta A_j \qquad (14.14)$$

When $\theta>0$, then $(XB[i]Bi-\theta yij)\geq 0$ because we have assumed that $yij \leq 0$, for $i=1,2,\ldots,m$. Thus Equation 14.14 is feasible. The value of the objective function is again

$$Z^\star = \sum_{i=1}^{m} c_B[i](X_B[i] - \theta y_{ij}) + c_j\theta$$

$$= Z + \theta(c_j - z_j)$$

Thus choosing *θ* arbitrarily large we can make $Z^\star$ arbitrarily small if

$$cj-zj<0$$

Page 405

To summarize:

Given any basis feasible solution to a linear program, if there is a column *Aj* not in the basis for which

$$cj-zj<0$$

and

$$Yj=B-1Aj\leq 0,$$

then the linear program in Equation 14.3 has an unbounded solution.

In terms of the tableau given in Equation 14.8, this means that if we can find a column *j* with *every* entry less than or equal to zero with a negative entry in last row, then the linear program in Equation 14.3 has an unbounded solution.

14.2.8 Conditions for optimality

Assume that $XB = B-1b$ is a basis feasible solution of the linear program given in Equation 14.3 and that the value of the objective function at *XB* is

$$Z_0 = c_B^T X_B.$$

In addition, suppose that

$$cj - zj \geq 0$$

for every column *Aj* of *A* not in *B*. Thus the value of the objective function cannot be improved by replacing a column of *B* with a column of *A*. We will show that *Z*0 is the minimal value of the linear program and hence that *XB* is an optimal solution. Set $\overrightarrow{z} = [z_1, z_2, \ldots, z_n]^T$. So,

$$\overrightarrow{z} \leq c$$

Let *X* be any feasible solution of linear program given in Equation 14.3. Then

$$x1A1 + x2A2 + \ldots + xnAn = b \qquad (14.15)$$

and $X \geq 0$. Let $Z^\star = C^T X$ be the value of the objective function at *X*.
Every column *Aj* of *A* can be written as a linear combination of the columns of *B:*

$$Aj = BYj$$

Setting $Y = [Y1, Y2, \ldots, Yn]$ we have $A = BY$, and $\overrightarrow{z} = c_B^T Y$. Then

$$BXB = b = AX = BY\ X$$

Therefore,

$$XB = YX$$

because *B* is nonsingular. Hence

$$Z_0 = c_B^T X_B = c_B^T Y X = \overrightarrow{z}\ X \leq c^T X = Z^\star,$$

which proves that *Z*0 is the optimal value of the objective function and hence *XB* is an optimal solution to the linear program in Equation 14.3.
In terms of the tableau given in Equation 14.8, this means if there is no negative entry in the last row, then we have found an optimal solution. The optimal value is *Z* the negative of the entry in the last row and column. To find the optimal solution, first discover the columns $\{j1, j2, \ldots, jm\}$ that contain the identity matrix on the first *m* rows of the tableau. That is, column *ji* is

$$[0, 0, \ldots, 0, \underbrace{1}_{i\text{th}}, 0, 0, \ldots, 0]^T.$$

Then the optimal solution is *X* where

$$X[j] = \begin{cases} X_B[i] & \text{if } j = j_i \\ 0 & \text{if not} \end{cases}$$

The Phase 2 portion of the simplex algorithm is given in Algorithm 14.2.3.

Algorithm 14.2.3: PHASE2 *(Tableau m, n)*

$c \leftarrow 1$

while $c < n$

$$
\begin{array}{l}
\textbf{if } Tableau[m+1,c] < 0 \\
\quad \left\{ \begin{array}{l}
i \leftarrow 0 \\
\textbf{while } Tableau[i,c] \leq 0 \textbf{ and } i \leq m \textbf{ do } i \leftarrow i+1 \\
\textbf{if } i \geq m \textbf{ then } \left\{ \begin{array}{l} Unbounded \leftarrow \textbf{true} \\ \textbf{return} \end{array} \right. \\
M \leftarrow \frac{Tableau[i,n+1]}{Tableau[i,c]} \\
r \leftarrow i \\
i \leftarrow i+1 \\
\textbf{while } i \leq m \textbf{ do} \\
\quad \left\{ \begin{array}{l}
\textbf{if } (Tableau[i,c] > 0) \textbf{ and } \frac{Tableau[i,n+1]}{Tableau[i,c]} < M \\
\quad \textbf{then } \left\{ \begin{array}{l} M \leftarrow \frac{Tableau[i,n+1]}{Tableau[i,c]} \\ r \leftarrow i \end{array} \right. \\
i \leftarrow i+1
\end{array} \right. \\
\textsc{Pivot}(r,c) \\
pivots[r] \leftarrow c \\
c \leftarrow 1
\end{array} \right. \\
\textbf{do} \quad \textbf{then} \\
c \leftarrow c+1
\end{array}
$$

14.2.9 Phase 1: initial basis feasible solution

Suppose that after **Phase 0** we have obtained a basis solution *XB* to the linear program

$$\text{Minimize: } Z = cTX \qquad (14.16)$$
$$\text{Subject to: } AX = b$$
$$X \geq 0$$

where *A* is a *m* by *n* matrix of rank *m*. Then

$$B = [B1, B2, \ldots, Bm]$$

is a nonsingular submatrix of *A* and $XB = B-1b$. The result of pivoting on the columns in *B* is the tableau

$$\left[\begin{array}{c|c} \cdots \quad Y_j \quad \cdots & X_B \\ \hline \cdots \; c_j - z_j \; \cdots & -z \end{array}\right].$$

If *XB[i]*<0 for some row *i,* then the basis solution is infeasible and we cannot proceed to **Phase 2**. Let E be the m-dimensional vector defined by

$$E[i] = \begin{cases} -1, \text{ if } X_B[i] < 0 \\ 0, \text{ if } X_B[i] \geq 0 \end{cases}$$

Let *x*0 be a new artificial variable and define the **Phase 1** *linear program* as follows:

$$\text{Minimize: } w = [1,0,0,\ldots,0] \begin{bmatrix} x_0 \\ X \end{bmatrix} \qquad (14.17)$$
$$\text{Subject to: } [E,Y] \begin{bmatrix} x_0 \\ X \end{bmatrix} = X_B$$
$$x_0, X \geq 0$$

The **Phase 1** *tableau* is:

$$\left[\begin{array}{c|ccc|c} E & \cdots & Y_j & \cdots & X_B \\ \hline 0 & \cdots & c_j - z_j & \cdots & -Z \\ \hline 1 & 0,0,\cdots & 0 & \cdots 0,0 & 0 \end{array}\right].$$

It has columns 0, 1,..., *n*. The second to last row corresponds to the cost equation to the linear program 14.16 and can almost be ignored for this discussion. Let *w*(*x*0,*X*) denote the value of the objective function for the **Phase 1** linear program at a given *x*0 and *X*.

LEMMA 14.2 The **Phase 1** *linear program is always feasible and has a non-negative optimal objective value*.

PROOF Let *i* be the row for which *XB[i]* is smallest (i.e., most negative), and pivot on row *i* column 0. The result is a tableau with a feasible solution to the **Phase 1** linear program. We can apply the techniques of Section 14.2.6 to obtain an optimal feasible solution or show that it is unbounded below. It cannot be unbounded below, however, because

$$w = w(x_0, X) = [1, \overrightarrow{0}] \begin{bmatrix} x_0 \\ X \end{bmatrix} = x_0 \geq 0.$$

Hence *w* is non-negative.

THEOREM 14.3 A linear program is feasible if and only if the artificial objective function has minimum value wmin=0. *Moreover, if w*=0 *for x*0≥0 *and X*≥0, *then X is a feasible solution of the original linear program*.

Page 409

PROOF By Lemma 14.2, the **Phase 1** linear program has an optimal solution for which the minimum value *wmin*≥0. First suppose *wmin*=0. Then there exist *X* and *x*0 such that *w*(*x*0,*X*)≥0, *x*0=0, *X*≥0, and

$$[E, Y] \begin{bmatrix} x_0 \\ X \end{bmatrix} = x_0 E + YX = X_B.$$

Then *B*−1*AX*=*YX*=*B*−1*b*=*XB* and hence *AX*=*b*. Thus the linear program 14.16 is feasible.

Conversely, *suppose* there exists a feasible solution $\hat{X}$ of *AX*=*b*. Then $Y\hat{X} = B^{-1}A\hat{X} = B^{-1}b = \hat{X}_B$ and thus

$$[E, Y] \begin{bmatrix} x_0 \\ X \end{bmatrix} = X_B$$

is solvable in non-negative variables, by choosing *x*0=0, and $X = \hat{X}$. For these values of *x*0 and *X*, $w(0, \hat{X}) = 0.$ Because *w*≥0 must hold for every feasible solution of the **Phase 1** linear program, we have *wmin*=0.

Page 410

Algorithm 14.2.4: PHASE1 *(Tableau, m, n)*

Unbounded←**false**

Infeasible←**false**

comment: Use column 0 and row *m*+2 to create the Phase 1 tableau

$$\textbf{for } i \leftarrow 1 \textbf{ to } m \textbf{ do} \begin{cases} \textbf{if } Tableau[i, n+1] < 0 \\ \quad \textbf{then } Tableau[i, 0] \leftarrow -1 \\ \quad \textbf{else } Tableau[i, 0] \leftarrow 0 \end{cases}$$

Tableau [m+2,0]←1

for *j*←1 **to** *n* **do** *Tableau*[*m*+2,*j*]←0

Tableau [m+2,*n*+1]←0

comment: find the Phase 1 first pivot

r=0

M←0

$$\textbf{for } i \leftarrow 1 \textbf{ to } m \textbf{ do} \begin{cases} \textbf{if } Tableau[i, n+1] < M \\ \quad \textbf{then} \begin{cases} M \leftarrow Tableau[i, n-1] \\ r \leftarrow i \end{cases} \end{cases}$$

if *r*=0

$$\textbf{then} \begin{cases} \textbf{comment:} \text{ no phase 1 is necessary} \\ \textbf{return} \end{cases}$$

PIVOT (*r*,0)

c←1

while *c*≤*n*

```
if Tableau[m+2, c] < 0
do then { i ← 1
          while Tableau[i, c] ≤ 0 do i ← i + 1
          M ← Tableau[i,n+1] / Tableau[i,c]
          r ← i
          i ← i + 1
          while i ≤ m do
          { if (Tableau[i, c] > 0) and Tableau[i,n+1] / Tableau[i,c] < M
            then { M ← Tableau[i,n+1] / Tableau[i,c]
                   r ← i + 1
            i ← i + 1
          PIVOT(r, c)
          pivots[r] ← c
          c ← 1
c ← c + 1
```

if *Tableau* [*m*+2,*n*+1]≠0
then *Infeasible*←**true**

Page 411

We can now give the complete simplex algorithm.

Algorithm 14.2.5: SIMPLEX *(Tableau, m, n)*

Unbounded←**false**
Infeasible←**false**
PHASE0 *(Tableau, m, n)*
if *Infeasible*
then { **output** ("The Linear Program is infeasible.")
return

PHASE 1 *(Tableau, m, n)*
if Infeasible
then { **output** ("The Linear Program is infeasible.")
return

PHASE2 *(Tableu m, n)*
if *Unbounded*
then { **output** ("The Linear Program is unbounded.")
return

Z← −*Tabl*eau[*m*+1, *n*+1]
for *j*←1 **to** *n* **do** *X[j]*←0
for i←1 to *m* **do** *X [pivots[r]]= Tableau* [*i, n*+1]
return *(X,Z)*

14.2.10 An example

Minimize: $Z=7x1+2x2$
Subject to: $-x1+2x2+x3=4$
$4x1+3x2+x3+x4=24$
$-2x1-2x2+x5=-7$
$x1,x2,x3,x4\geq 0$

The initial tableau is

$$\left[\begin{array}{rrrrr|r} -1 & 2 & 1 & 0 & 0 & 4 \\ 4 & 3 & 1 & 1 & 0 & 24 \\ -2 & -2 & 0 & 0 & 1 & -7 \\ \hline 7 & 2 & 0 & 0 & 0 & 0 \end{array}\right]$$

Page 412

and we start **Phase 0.** Pivoting on the [1,3] entry obtains

$$\left[\begin{array}{rrrrr|r} -1 & 2 & 1 & 0 & 0 & 4 \\ 5 & 1 & 0 & 1 & 0 & 20 \\ -2 & -2 & 0 & 0 & 1 & -7 \\ \hline 7 & 2 & 0 & 0 & 0 & 0 \end{array}\right]$$

Thus we have the basis solution

$$X=[0,0,4,20,-7]T$$

corresponding to basis consisting of columns 3, 4 and 5. This ends **Phase 0**. The basis solution found was not feasible so we start **Phase 1**. In this phase we have columns 0, 1, 2, 3, 4, and 5.

$$\left[\begin{array}{r|rrrrrr|r} 0 & -1 & 2 & 1 & 0 & 0 & 4 \\ 0 & 5 & 1 & 0 & 1 & 0 & 20 \\ -1 & -2 & -2 & 0 & 0 & 1 & -7 \\ \hline 0 & 7 & 2 & 0 & 0 & 0 & 0 \\ \hline 1 & 0 & 0 & 0 & 0 & 0 & 0 \end{array}\right]$$

First we price out column 0.

$$\left[\begin{array}{r|rrrrrr|r} 0 & -1 & 2 & 1 & 0 & 0 & 4 \\ 0 & 5 & 1 & 0 & 1 & 0 & 20 \\ 1 & 2 & 2 & 0 & 0 & -1 & 7 \\ \hline 0 & 7 & 2 & 0 & 0 & 0 & 0 \\ \hline 0 & -2 & -2 & 0 & 0 & 1 & -7 \end{array}\right]$$

We now have a feasible solution, but the **Phase 1** objective value is 7. We proceed to reduce this value to 0. In the last row we find a negative entry in column 1. The smallest ratio of the last column entries with positive entries in column 1 is θ=7/2, so we pivot on the [3,1] entry to obtain the tableau below.

$$\left[\begin{array}{r|rrrrrr|r} \frac{1}{2} & 0 & 3 & 1 & 0 & -\frac{1}{2} & 7\frac{1}{2} \\ -2\frac{1}{2} & 0 & -4 & 0 & 1 & 2\frac{1}{2} & 2\frac{1}{2} \\ \frac{1}{2} & 1 & 1 & 0 & 0 & -\frac{1}{2} & 3\frac{1}{2} \\ \hline -3\frac{1}{2} & 0 & -5 & 0 & 0 & 3\frac{1}{2} & -24\frac{1}{2} \\ \hline 1 & 0 & 0 & 0 & 0 & 0 & 0 \end{array}\right]$$

The **Phase 1** objective value is now zero and we have a feasible solution. We proceed to **Phase 2** by dropping column 0 and the last row.

Page 413

$$\left[\begin{array}{rrrrr|r} 0 & 3 & 1 & 0 & -\frac{1}{2} & 7\frac{1}{2} \\ 0 & -4 & 0 & 1 & 2\frac{1}{2} & 2\frac{1}{2} \\ 1 & 1 & 0 & 0 & -\frac{1}{2} & 3\frac{1}{2} \\ \hline 0 & -5 & 0 & 0 & 3\frac{1}{2} & -24\frac{1}{2} \end{array}\right]$$

There is a negative entry in the last row in column 2, so the objective value can be reduced. The smallest ratio is $\theta = (7\frac{1}{2})/3,$ so we pivot on the [1,2]-entry.

$$\left[\begin{array}{ccccc|c} 0 & 1 & \frac{1}{3} & 0 & -\frac{1}{6} & 2\frac{1}{2} \\ 0 & 0 & 1\frac{1}{3} & 1 & 1\frac{5}{6} & 12\frac{1}{2} \\ 1 & 0 & -\frac{1}{3} & 0 & -\frac{1}{3} & 1 \\ \hline 0 & 0 & 1\frac{2}{3} & 0 & 2\frac{2}{3} & -12 \end{array}\right]$$

There are no negative entries left in the last row so an optimal solution has been found. This solution is

$$X = [1, 2\frac{1}{2}, 0, 12\frac{1}{2}, 0]^T$$

and has objective value Z=12.

14.3 Cycling

It is prudent to ask whether it is possible for the simplex algorithm to go through an endless sequence of iterations without terminating. Consider the following linear program:

Minimize: $Z=-10x1+57x2+9x3+24x4$
Subject to: $0.5x1-5.5x2-2.5x3+9x4+x5=0$
$0.5x1-1.5x2-0.5x3+x4+x6=0$
$x1+x7=1$
$x1,x2,x3,x4,x5,x6,x7\geq=0$

The initial tableau is

$$\left[\begin{array}{ccccccc|c} 0.5 & -5.5 & -2.5 & 9 & 1 & 0 & 0 & 0 \\ 0.5 & -1.5 & -0.5 & 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ \hline -10 & 57 & 9 & 24 & 0 & 0 & 0 & 0 \end{array}\right].$$

If we adopt the following rule:

Always pivot on the column with the smallest (most negative) entry in the bottom row, choosing the first row that achieves the smallest ratio. If there are two such columns always choose the first one, (i.e., the one with smallest index).

Then the sequence of iterations is:

1. Pivot on (1, 1).

$$\left[\begin{array}{ccccccc|c} 1 & -11 & -5 & 18 & 2 & 0 & 0 & 0 \\ 0 & 4 & 2 & -8 & -1 & 1 & 0 & 0 \\ 0 & 11 & 5 & -18 & -2 & 0 & 1 & 1 \\ \hline 0 & -53 & -41 & 204 & 20 & 0 & 0 & 0 \end{array}\right]$$

2. Pivot on (2, 2).

$$\left[\begin{array}{ccccccc|c} 1 & 0 & 0.5 & -4 & -0.75 & 2.75 & 0 & 0 \\ 0 & 1 & 0.5 & -2 & -0.25 & 0.25 & 0 & 0 \\ 0 & 0 & -0.5 & 4 & 0.75 & -2.75 & 1 & 1 \\ \hline 0 & 0 & -14.5 & 98 & 6.75 & 13.25 & 0 & 0 \end{array}\right]$$

3. Pivot on (1, 3).

$$\left[\begin{array}{ccccccc|c} 2 & 0 & 1 & -8 & -1.5 & 5.5 & 0 & 0 \\ -1 & 1 & 0 & 2 & 0.5 & -2.5 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ \hline 29 & 0 & 0 & -18 & -15 & 93 & 0 & 0 \end{array}\right]$$

4. Pivot on (2, 4).

$$\left[\begin{array}{ccccccc|c} -2 & 4 & 1 & 0 & 0.5 & -4.5 & 0 & 0 \\ -0.5 & 0.5 & 0 & 1 & 0.25 & -1.25 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ \hline 20 & 9 & 0 & 0 & -10.5 & 70.5 & 0 & 0 \end{array}\right]$$

5. Pivot on(1, 5).

$$\left[\begin{array}{rrrrrrr|r} -4 & 8 & 2 & 0 & 1 & -9 & 0 & 0 \\ 0.5 & -1.5 & -0.5 & 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ \hline -22 & 93 & 21 & 0 & 0 & -24 & 0 & 0 \end{array}\right]$$

6. Pivot on (2, 6).

$$\left[\begin{array}{rrrrrrr|r} 0.5 & -5.5 & -2.5 & 9 & 1 & 0 & 0 & 0 \\ 0.5 & -1.5 & -0.5 & 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ \hline -10 & 57 & 9 & 24 & 0 & 0 & 0 & 0 \end{array}\right]$$

The tableau obtained after iteration 6 is identical to the initial tableau and so this cycle of iterations would repeat and the simplex algorithm would never terminate. It is easy to see that this is the only way that the simplex algorithm can fail to terminate. That is

THEOREM 14.4 If the simplex algorithm fails to terminate, then it must cycle.

Several methods have been proposed to avoid this cycling phenomenon. The easiest is to adopt the *smallest index rule*.

Always pivot on the column with the first negative entry in the bottom row (i.e., the one with the smallest index), and choose the first row in that column that achieves the smallest ratio.

We leave it as an exercise to prove the following result:

THEOREM 14.5 **(R.G. Bland,** 1977) *The simplex method always terminates if the smallest index rule is adopted.*

PROOF Exercise 14.3.2.

Exercises

14.3.1 Solve the following linear programs using the simplex algorithm.

(a)

$$\begin{gathered} \text{Maximize: } z=3x1+2x2+x3+4x4 \\ \text{Subject to: } 4x1+5x2+x3-3x4=5, \\ 2x1-3x2-4x3+5x4=7, \\ x1+4x2+2.5x3-4x4=6, \\ x1,\ x2,\ x3,\ x4\geq 0 \end{gathered}$$

(b)

$$\begin{gathered} \text{Maximize: } z=3x1+4x2+x3+7x4 \\ \text{Subject to: } 8x1+3x2+4x3+x4\leq 5, \\ 2x1+6x2+x3+5x4\leq 7, \\ x1+4x2+5x3+2x4\leq 6, \\ x1,\ x2,\ x3,\ x4\geq 0 \end{gathered}$$

(c)

$$\begin{gathered} \text{Maximize: } z=2x1-3x2+4x3+x4 \\ \text{Subject to: } x1+5x2+9x3-6x4\geq -2, \\ 3x1-1x2+x3+3x4\leq 10, \\ -2x1-3x2+7x3-8x4\geq 0, \\ x1,x2,x3,x4\geq 0 \end{gathered}$$

14.3.2 Prove Theorem 14.5.

14.4 Notes

The topic of linear programming appears in a variety of different subjects, for example operations research, mathematical programming, and combinatorial optimization. There are thus numerous books in which it is discussed and among them are CHVÁTAL [27], HADLEY [57], NEMHAUSER and WOLSEY [92], PAPADIMITRIOU and STEIGLITZ [94], TAHA [110], and WALKER [121].

In this chapter we have only discussed the simplex algorithm which was invented in the late 1940s by DANZIG [31] (see also DANTZIG [32]). A thorough discussion of the history of linear programming can be found in DANTZIG'S celebrated work [33].

The example of cycling in the simplex method found in Section 14.3, is from the book by CHVÁTAL [27]. Theorem 14.5 appears in BLAND [16].
It can be shown that in the worst case the running time of simplex algorithm is not polynomial bounded (see KLEE and MINTY [72]) and hence the simplex algorithm is theoretically not satisfactory. In practice it is eminently useful and except for very contrived problems exceedingly fast. In 1979, KHACHIAN [71] provided a method called the *ellipsoid method* that solves linear programs in polynomial time. This is a marvelous, elegant, and simple jewel of pure mathematics. However, we believe that it is unlikely that the ellipsoid method will ever be a serious challenger to the simplex method.

15

The Primal-Dual Algorithm

15.1 Introduction

In Chapter 14 we found it convenient to convert every linear program consisting of constraints that are a mix of inequalities and equalities:

$$\sum a_{ij}x_i\{\leq=\geq\}b_i, \quad i=1,2,\ldots,m$$

to a system of equations $Ax=b$, $b\geq 0$. A slightly different formulation of constraints will prove useful here. We convert every equality to the equivalent pair of inequalities, so that

$$\sum a_{ij}x_i = b_i$$

becomes the two inequalities

$$\sum a_{ij}x_i \geq b_i$$

$$\sum a_{ij}x_i \leq b_i$$

We then multiply all inequalities with the relation $\geq$ through by a -1 so that each has the form

$$\sum a_{ij}x_i \leq b_i.$$

Now we have a linear program of the form

Maximize $z=cTX$ (Primal)
subject to: $DX\leq d$
$X\geq 0$

which we call the *primal linear program*. Corresponding to the primal linear program is another linear program which we call the *dual linear program*.

Minimize $Z=dTW$ (Dual)
subject to: $DTW\geq c$
$W\geq 0$

LEMMA 15.1 The dual of the dual is the primal:

PROOF The dual linear program:

Minimize $Z=dTW$
subject to: $DTW\geq c$
$W\geq 0$

is equivalent to :

$$\begin{array}{lrcl} \text{Maximize} & z^\star = (-d)^T W & & \\ \text{subject to:} & (-D)^T W & \leq & -c\ . \\ & W & \geq & 0 \end{array}$$

This is in the form of a primal linear program. The dual linear program corresponding to it is:

$$\begin{array}{lrcl} \text{Minimize} & Z^\star = (-c)^T X & & \\ \text{subject to:} & -DX & \geq & -d\ . \\ & X & \geq & 0 \end{array}$$

This linear program is equivalent to

Maximize $z=cTX$
subject to: $DX\leq d$.
$X\geq 0$

LEMMA 15.2 If X is a feasible solution to the primal and W is a feasible solution to the dual, then cTX≤dTW (*implying z≤Z*).
PROOF Suppose *X* is a feasible solution to the primal. Then *DX≤d*. If *W* is a feasible solution to the dual, then *W*≥0, so we see

$$WTDX<WTd=dTW$$

Similarly, because *X*≥0 and *W* is a feasible solution,

$$DTW>c$$
$$XTDTW\geq XTc$$
$$WTDX\geq cTX$$

Page 419
Therefore

$$cTX\leq WTDX\leq dTW. \qquad (15.1)$$

LEMMA 15.3 If $\widehat{X}$ *is a feasible solution to the primal and* $\widehat{W}$ *is a feasible solution to the dual such that* $c^T\widehat{X} = d^T\widehat{W}$, *then* $\widehat{X}$ *and* $\widehat{W}$ *are optimal solutions to the primal and dual, respectively.*
PROOF By assumption, $c^T\widehat{X} = d^T\widehat{W}$, but for any feasible solution *X* to the primal,

$$c^T X \leq d^T\widehat{W} = c^T\widehat{X}.$$

(with the inequality from Lemma 15.2). Therefore, $\widehat{X}$ is an optimal solution to the primal.
By the same logic, for any feasible solution *W* of the dual,

$$d^T W \geq c^T\widehat{X} = d^T\widehat{W}.$$

Thus, $\widehat{W}$ is an optimal solution to the dual.
LEMMA 15.4 If the dual or the primal has an optimal solution, then the other also must have an optimal solution and their optimal values are the same.
PROOF Because of Lemma 15.1, we need only show that if the primal linear program has an optimal solution, then so does the dual linear program. Recall that the primal linear program is:

Maximize *z=cTX*
subject to: *DX≤d*
X≥0.

Add slack variables *Xs* and convert it to a minimization problem to get the standard form linear program:

Minimize *(−z)=(−c)TX* (15.2)
subject to: *DX+IXs=d,*
X, Xs≥0

where *I*=[*E*1,*E*2,...,*Em*] is the *m* by *m* identity matrix. Let *D*= [*D*1,...,*Dn*]. The tableau for the simplex algorithm is

$$\left[\begin{array}{cccccccc|c} D_1 & \cdots & D_j & \cdots & D_n & E_1 & \cdots & E_{j'} \cdots E_m & d \\ \hline -c_1 & \cdots & -c_j & \cdots & -c_n & 0 & \cdots & 0 \quad \cdots 0 & 0 \end{array}\right]$$

Page 420
If the linear program 15.2 has an optimal solution *XB,* with optimal value *(−z)=(−CB)TXB,* then there is a rank *m* submatrix *B* of the columns *D*1, *D*2,..., *Dn, E*1, *E*2,..., *Em* such that *XB=B−1d* and after multiplying the tableau by

$$\left[\begin{array}{c|c} B^{-1} & 0 \\ \hline c_B^T B^{-1} & 1 \end{array}\right]$$

we obtain the tableau

$$\left[\begin{array}{cccccccc|c} \cdots & \cdots & Y_j & \cdots & \cdots & \cdots & Y_{j'} & \cdots & \cdots & X_B \\ \hline \cdots & \cdots & z_j - c_j & \cdots & \cdots & \cdots & z_{j'} & \cdots & \cdots & z \end{array}\right]$$

where,

$$z = c_B^T B^{-1} d,$$
$$Y_j = B^{-1} D_j,$$
$$Y_{j'} = B^{-1} E_{j'},$$

and because of optimality,

$$0 \leq z_j - c_j = c_B^T B^{-1} D_j - c_j$$
$$0 \leq z_{j'} = c_B^T E_{j'}.$$

So these equations show that

$$c_B^T B^{-1} D - c^T \geq 0$$
$$c_B^T B^{-1} I \geq 0, \text{ and}$$
$$z = c_B^T B^{-1} d.$$

If we let $W^T = c_B^T B^{-1}$, then these equations show that *W* satisfies

$DTW \geq c$
$W \geq 0$, and
$z = dTW$.

Thus *W* is a an optimal feasible solution to the dual linear program. Optimality follows from the last equation and Lemma 15.3.

Observe the similarity between Lemma 15.4 and the *max-flow-min-cut* Theorem (Theorem 8.4). Indeed the Ford-Fulkerson algorithm for solving the maximum network flow problem that was presented in Section 8.2 is a primal-dual algorithm. We re-examine this algorithm in Section 15.6.3.

In the proof of Lemma 15.4 we discovered how to construct an optimal solution of the dual linear program given an optimal solution to the primal. We record this useful fact as the following corollary:

COROLLARY 15.5 If X is an optimal solution to

Maximize $z = cTX$
subject to: $DX \leq d$
$X \geq 0$

with basis B, then $W = B - TcB$ *is an optimal solution to the dual linear program*

Minimize $Z = dTW$
subject to: $DTW \geq c$
$W \geq 0$

THEOREM 15.6 Given a primal-dual pair, exactly one of the following can occur:
a. Both the primal and the dual have a finite optimum.
b. The primal is unbounded and the dual is infeasible.
c. The primal is infeasible but the dual is unbounded.
d. Both the primal and the dual are infeasible.

PROOF We saw in Chapter 14 that every linear program either (i) has a finite optimum, (ii) is unbounded, or (iii) is unfeasible. Thus for a primal-dual pair there are nine possibilities. Namely:

1. Both the primal and the dual have a finite optimum.
2. The primal has a finite optimum but the dual is unbounded.
3. The primal has a finite optimum but the dual is infeasible.
4. The primal is unbounded but the dual has a finite optimum.
5. Both the primal and the dual are unbounded.
6. The primal is unbounded and the dual is infeasible.
7. The primal is infeasible but the dual has a finite optimum.
8. The primal is infeasible but the dual is unbounded.
9. Both the primal and the dual are infeasible.

Lemma 15.4 shows that possibilities 2, 3, 4, and 7 cannot occur. Equation 15.1 tells us that if either the primal or the dual is unbounded, then the other cannot have a feasible solution and thus possibility 5 is eliminated. It is easy to construct examples of the remaining four possibilities, 1, 6, 8, and 9.

1. A primal-dual pair in which both the primal and the dual have a finite opti-

mum:

Maximize $z=x1$ (Primal)
subject to: $x1\leq 1$
$x1\geq 0$

Minimize $Z=w1$ (Dual)
subject to: $w1\geq 1$
$w1\geq 0$

6. A primal-dual pair in which the primal is unbounded and the dual is infeasible:

Maximize $Z=-x1+2x2$ (Primal)
subject to: $-x1+x2\leq 1$
$x1, x2\geq 0$

Minimize $z=w1$ (Dual)
subject to: $-w1\geq -1$
$w1\geq 2$
$w1\geq 0$

8. A primal-dual pair in which the primal is infeasible and the dual is unbounded:

Maximize $z=x1$ (Primal)
subject to: $x1\leq 1$
$-x1\leq -2$
$x1\geq 0$

Mimmize $Z=w1-2w2$ (Dual)
subject to: $w1-w2\geq 1$
$w1, w2\geq 0$

9. A primal-dual pair in which both the primal and the dual are infeasible:

Maximize $z=-x1+2x2$ (Primal)
subject to: $x1-x2\leq 1$
$-x1+x2\leq -2$
$x1, x2\geq 0$

Minimize $Z=w1-2w2$ subject to: $w1-w2\geq -1$ (Dual)
$-w1+w2>2$
$w1, w2\geq 0$

15.2 Alternate form of the primal and its dual

It is often more convent to write the primal linear program as:

Maximize $z=cTX$ (Primal: equality form)
subject to: $AX=b$
$X\geq 0$.

This is equivalent to

Maximize $z=cTX$
subject to: $AX\leq b$
$-AX\leq -b$
$X\geq 0$,

and this has the dual linear program

$$\begin{aligned} \text{Minimize} \quad & Z = [b^T, -b^T] \begin{bmatrix} W_1 \\ W_2 \end{bmatrix} \\ \text{subject to:} \quad & [A^T, -A^T] \begin{bmatrix} W_1 \\ W_2 \end{bmatrix} \geq c \\ & W_1, W_2 \geq 0. \end{aligned}$$

This dual is equivalent to:

Minimize $Z=bT(W1-W2)$
subject to: $AT(W1-W2)\geq c$
$W1, W2\geq 0$.

If we let $W=W1-W2$, then the entries of W are unrestricted in sign and the dual linear program is equivalent to

Minimize $Z=bTW$

subject to: $ATW \geq c$
W unrestricted.

Similarly, if we take the dual linear program to be

Minimize $Z = bTW$ (Dual: equality form)
subject to: $ATW = c$
$W > 0$.

Then its corresponding primal linear program is

Maximize $z = cTX$
subject to: $AX \leq b$
X unrestricted.

A similar proof of Corollary 15.5 found in Lemma 15.4 establishes Corollary 15.7.

COROLLARY 15.7 If W is an optimal solution to

Minimize Z=bTW (Dual: equality form)
subject to: ATW=c
W≥0.

with basis B, then X=B−TbBis an optimal solution to the dual linar program

Maximize $z = cTX$
subject to: $AX \leq b$
X unrestricted.

15.3 Geometric interpretation

A geometric interpretation can be give to the dual linear program.
Let $A = [A1, A2, \ldots, An]$ and write the primal linear program as:

Maximize $z = cTX$ (15.3)
subject to: $x1A1 + x2A2 + \ldots + xnAn = b$
$X \geq 0$.

Then the dual is

$$\begin{array}{lll} \text{Minimize} & Z = b^T W & \\ \text{subject to:} & A_1^T W & \geq c_1 \\ & A_2^T W & \geq c_2 \\ & \vdots & \\ & A_n^T W & \geq c_n \\ & W \text{ unrestricted.} & \end{array} \quad (15.4)$$

The vectors Aj in the primal linear program 15.3 are the normals to the the half-spaces that represent the constraints in the dual linear program 15.4. Furthermore the requirement vector of the primal is normal to the hyperplane $Z = bTW$ in the dual. This is easy to illustrate in two dimensions by means of an example.

15.3.1 Example

Given the linear program:

Maximize $z = -3x1 - 23x2 - 4x3$
subject to: $x1A1 + x2A2 + \ldots + x5A5 = b$
$x1, x2, \ldots, x5 \geq 0$,

---

where

$$A_1 = \begin{bmatrix} 2 \\ -1 \end{bmatrix}, \; A_2 = \begin{bmatrix} -3 \\ -4 \end{bmatrix}, \; A_3 = \begin{bmatrix} -2 \\ 3 \end{bmatrix}, \; A_4 = \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \; A_5 = \begin{bmatrix} 0 \\ 1 \end{bmatrix}, \quad \text{and } b = \begin{bmatrix} -2 \\ 1 \end{bmatrix}.$$

The dual linear program is:

Minimize $Z = -2w1 + w2$
subject to: $2w1 - 1w2 > -3$
$-3w1 - 4w2 \geq -23$
$-2w1 + 3w2 > -4$
$w1 \geq 0$
$w2 \geq 0$

(Note that the appearance of slack variables in the primal linear program have caused the variables in the

dual to be non-negative.) In Figure 15.1, we have drawn the *requirement-space configuration* of the primal and in Figure 15.2, the *convex set of feasible solutions* is shown as a shaded region.
Whenever two of the constraints hold as strict equalities, the vectors normal to these constraints are a basis for the primal (if the normals are linearly independent). In *w*1*w*2-space the point *w* where two dual constraints hold as strict equalities is the intersection of the two lines representing these two constraints. A basis solution to the primal can then be associated with the intersection of each pair of bounding lines for the half-spaces representing the dual constraints.

There are $\binom{5}{2} = 10$ basis solutions to the primal. They are represented by the points *Pij*, $1 \le i < j \le 5$, in Figure 15.2. The point *Pij* corresponds to having *Ai*, *Aj* in the primal basis. In Table 15.1 we display the simplex tableaux corresponding to the various choice of basis *{Ai, Aj}*. Two of them yield feasible solutions to the primal, but only one corresponds to a point that is also feasible in the dual. This is basis {*A*2, *A*3}, corresponding to the point *P*23=(5, 2). Furthermore this basis yields the value *z*=−8, obtained by setting *x*1=*x*4= *x*5=0, *x*2=0.24 and *x*3=0.65. This yields the value *Z*=[−2, 1]T[5, 2]= −8. Thus by Lemma 15.3, this an optimal point. Therefore using the dual simplex method we move from one extreme point of the convex polyhedron to an adjacent one until an optimal extreme point is reached. At this point, the corresponding solution to the primal becomes feasible.

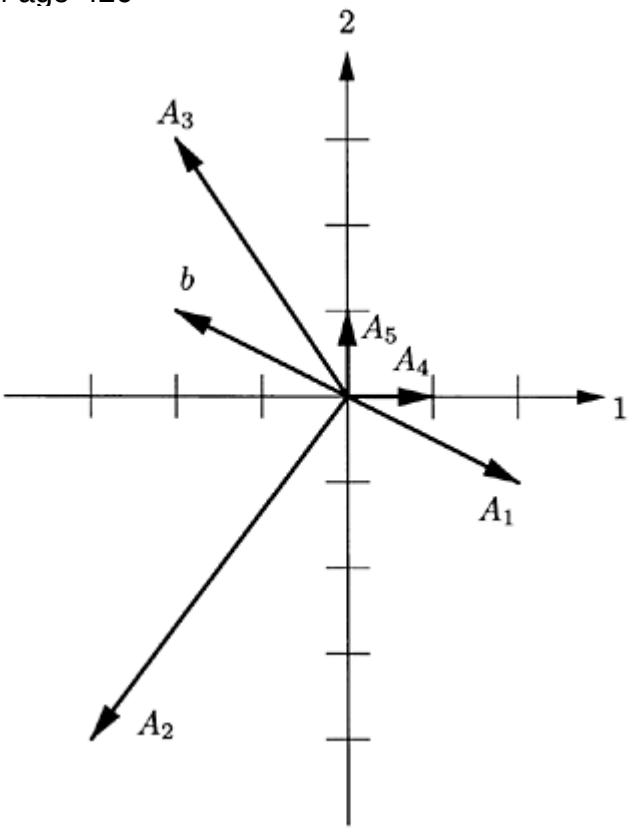


FIGURE 15.1
Requirement-space configuration for the primal

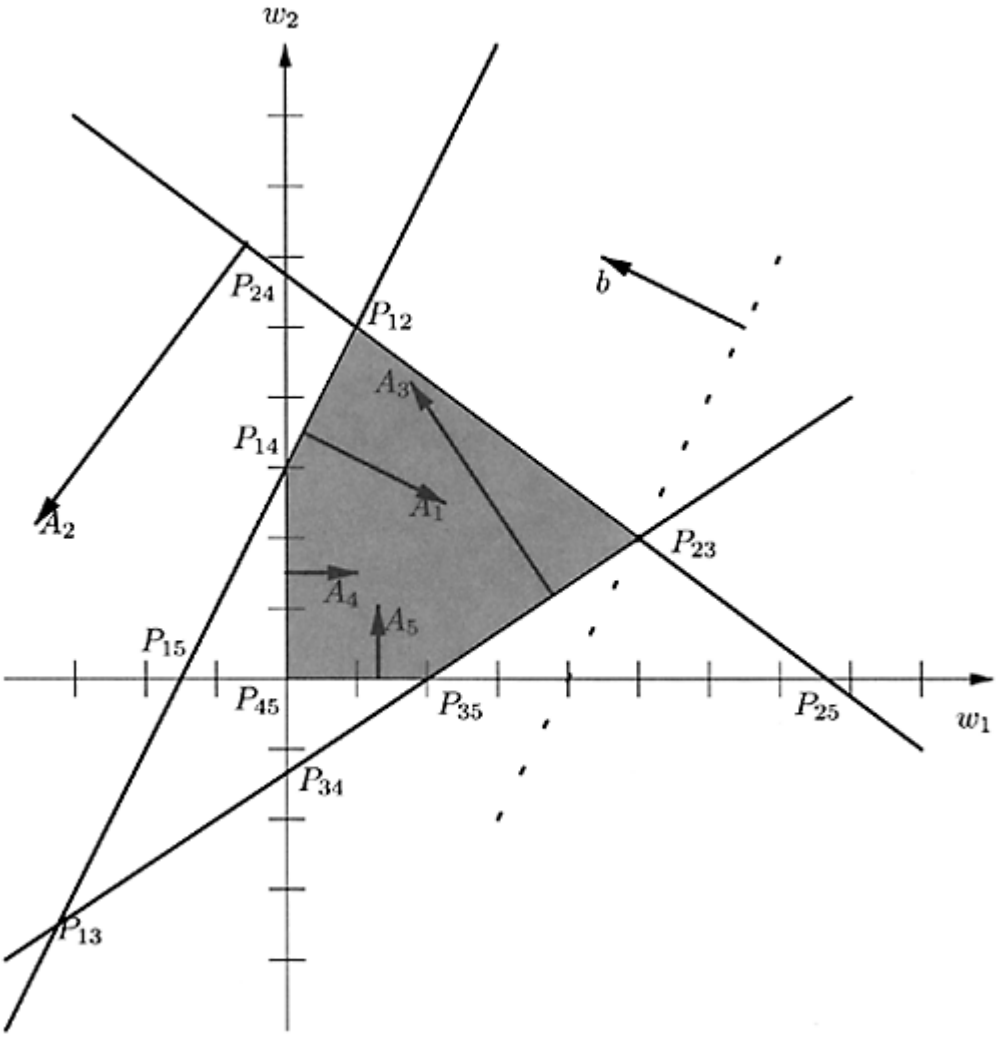


FIGURE 15.2
The convex set of feasible solutions to the dual

Page 428

TABLE 15.1
Table of simplex tableaux

$\left[\begin{array}{ccccc\|c} 1 & 0 & -1.55 & 0.36 & -0.27 & -1 \\ 0 & 1 & -0.36 & -0.09 & -0.18 & 0 \\ \hline 0 & 0 & -17 & -1 & -5 & -3 \end{array}\right]$	Basis $\{A1, A2\}$ is infeasible.
$\left[\begin{array}{ccccc\|c} 1 & -4.25 & 0 & 0.75 & 0.50 & -1 \\ 0 & -2.75 & 1 & 0.25 & 0.50 & 0 \\ \hline 0 & -46.75 & 0 & 3.25 & 3.50 & -3 \end{array}\right]$	Basis $\{A1, A3\}$ is infeasible.
$\left[\begin{array}{ccccc\|c} 1 & 4 & -3 & 0 & -1 & -1 \\ 0 & -11 & 4 & 1 & 2 & 0 \\ \hline 0 & -11 & -13 & 0 & -3 & -3 \end{array}\right]$	Basis $\{A1, A4\}$ is infeasible.
$\left[\begin{array}{ccccc\|c} 1 & -1.50 & -1 & 0.50 & 0 & -1 \\ 0 & -5.50 & 2 & 0.50 & 1 & 0 \\ \hline 0 & -27.50 & -7 & 1.50 & 0 & -3 \end{array}\right]$	Basis $\{A1, A5\}$ is infeasible.
$\left[\begin{array}{ccccc\|c} -0.24 & 1 & 0 & -0.18 & -0.12 & 0.24 \\ -0.65 & 0 & 1 & -0.24 & 0.18 & 0.65 \\ \hline -11 & 0 & 0 & -5 & -2 & 8 \end{array}\right]$	Basis $\{A2, A3\}$ yields $z = 8$.

$$\left[\begin{array}{ccccc|c} 0.25 & 1 & -0.75 & 0 & -0.25 & -0.25 \\ 2.75 & 0 & -4.25 & 1 & -0.75 & -2.75 \\ \hline 2.75 & 0 & -21.25 & 0 & -5.75 & -5.75 \end{array}\right]$$

Basis **{***A***2,***A***4}** is infeasible.

$$\left[\begin{array}{ccccc|c} -0.67 & 1 & 0.67 & -0.33 & 0 & 0.67 \\ -3.67 & 0 & 5.67 & -1.33 & 1 & 3.67 \\ \hline -18.33 & 0 & 11.33 & -7.67 & 0 & 15.33 \end{array}\right]$$

Basis **{***A***2,***A***5}** yields *z*=**15.33**

$$\left[\begin{array}{ccccc|c} -0.33 & -1.33 & 1 & 0 & 0.33 & 0.33 \\ 1.33 & -5.67 & 0 & 1 & 0.67 & -1.33 \\ \hline -4.33 & -28.33 & 0 & 0 & 1.33 & 1.33 \end{array}\right]$$

Basis **{***A***3,***A***4}** is infeasible.

$$\left[\begin{array}{ccccc|c} -1 & 1.50 & 1 & -0.50 & 0 & 1 \\ 2 & -8.50 & 0 & 1.50 & 1 & -2 \\ \hline -7 & -17 & 0 & -2 & 0 & 4 \end{array}\right]$$

Basis **{***A***3,***A***5}** is infeasible.

$$\left[\begin{array}{ccccc|c} 2 & -3 & -2 & 1 & 0 & -2 \\ -1 & -4 & 3 & 0 & 1 & 1 \\ \hline -3 & -23 & -4 & 0 & 0 & 0 \end{array}\right]$$

Basis **{***A***4,***A***5}** is infeasible.

Page 429

15.4 Complementary slackness

There is a battle of balance between the primal and dual linear programs.
As the constraints tighten in one, they loosen in the other.
In this section we denote by *[Rowi(D)]* and *[Colj(D)]* the *ith* row and *jth* column of the matrix *D,* respectively.
THEOREM 15.8 **(Complementary slackness conditions)** *A primal-dual feasible solution pair X, W is optimal if and only if*

$$xj([Colj(D)]TW-cj)=0 \text{ for all } j \qquad (15.5)$$
$$wi(di-[Rowi(D)]TX)=0 \text{ for all } i \qquad (15.6)$$

PROOF Let

$$uj=xj([Colj(D)]TW-cj), \text{ and}$$
$$ui=wi(di-[Rowi(D)]TX).$$

Then, because of the feasibility and the duality relations we have *uj*≥0 for all *j* and *ui*≥0 for all *i*. Let

$$u = \sum_j u_j$$

$$v = \sum_i v_i.$$

Then *u, u*≥0, *u*=0 if and only if Equation 15.5 holds for all *j* and *u*=0 if and only if Equation 15.6 holds for all *j*. Observe that

$$\begin{aligned} u+v &= \sum_j u_j + \sum_i v_i \\ &= \sum_j x_j([Col_j(D)]^T W - c_j) + \sum_i w_i(d_i - [Row_i(D)]^T X) \\ &= -\sum_j x_j c_j + \sum_i w_i d_i + \sum_j x_j([Col_j(D)]^T W - \sum_i w_i [Row_i(D)]^T X \\ &= -c^T X + d^T W + \sum_j x_j \sum_i D[i,j] w_i - \sum_i w_i \sum_j D[i,j] x_j \\ &= -c^T X + d^T W + \sum_i \sum_j w_i D[i,j] x_j - \sum_i \sum_j w_i D[i,j] x_j \\ &= -c^T X + d^T W \end{aligned}$$

Therefore Equations 15.5 and 15.6 hold for all *j* and *i,* respectively, if and only if *u*+*u*=0 if and only if *cTX*=*dTW* if and only if *X* and *W* are both optimal.
Note that Theorem 15.8 says at optimality if a constraint is not met with equality, i.e. has slack, in the primal linear program, then the corresponding variable in the dual is zero and vice versa.

15.5 The dual of the shortest-path problem

In this section we study the *shortest-path problem* for directed graphs.

Problem 15.1: Shortest Path (directed graph)

Instance: a directed graph *G*=*(V,E),* nodes *s,* $t \in V$ and non-negative weights *cj,* for each edge $e_j \in E.$
Find: a directed path *P* from *s* to *t* with minimum total weight

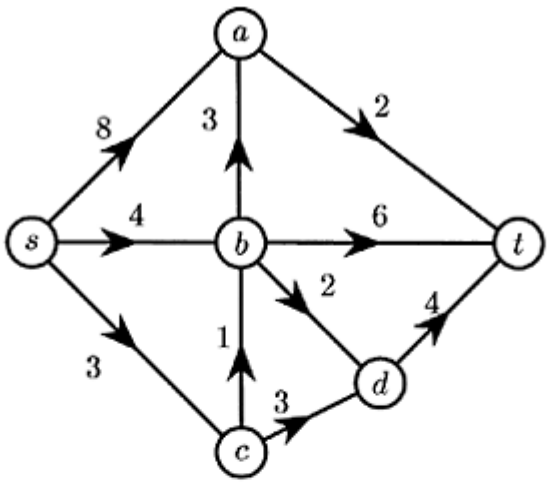
$$C(P) = \sum_{e_j \in E(P)} c_j$$

Let *V*={*u*1, *u*2,..., *un*} *and E*={*e*1, *e*2,..., *em*} define the m by *n node-edge incidence matrix A* by

$$A[i,j] = \begin{cases} +1, & \text{if } e_j = (v_i, u) \text{ for some vertex } u, \\ -1, & \text{if } e_j = (u, v_i) \text{ for some vertex } u, \\ 0, & \text{otherwise} \end{cases}$$

In Figure 15.3 an example is given.
We model the shortest-path problem as a network flow problem by assigning a capcity of 1 on each edge. Let *wj* denote the flow on edge *j*.
The conservation of flow constraints are

$$[Row_i(A)] \cdot W = \begin{cases} 0, i \notin \{s,t\} \\ +1, i = s \\ -1, i = t \end{cases}$$



$A =$

	(s,a)	(s,b)	(s,c)	(a,t)	(b,a)	(b,d)	(b,t)	(c,b)	(c,d)	(d,t)
s	+1	+1	+1	0	0	0	0	0	0	0
a	−1	0	0	+1	−1	0	0	0	0	0
b	0	−1	0	0	+1	+1	+1	−1	0	0
c	0	0	−1	0	0	0	0	+1	+1	0
d	0	0	0	0	0	−1	0	0	−1	+1
t	0	0	0	−1	0	0	−1	0	0	−1

$c^T =$ [8, 4, 3, 2, 3, 2, 6, 3, 3, 4]

FIGURE 15.3
An instance of the shortest-path problem

Page 432
Then W satisfies

$$AW = \begin{bmatrix} +1 \\ -1 \\ 0 \\ \vdots \\ 0 \end{bmatrix} \begin{matrix} \\ \leftarrow \text{row } s \\ \leftarrow \text{row } t \\ \\ \\ \end{matrix}$$

and the capacity constrains are

$$0 \leq wj \leq 1$$

A solution to the shortest-path problem is given by a flow *W* that minimizes *Z=cTW*. As far as we know it is possible that the *wj* in general take on non-integer values, but in Section 16.4 we will show that there is an optimal solution *W* to linear program 15.7 with only integer entries.

$$\begin{aligned} &\text{Minimize} && Z = c^T W \\ &\text{subject to:} && AW = \begin{bmatrix} +1 \\ -1 \\ 0 \\ \vdots \\ 0 \end{bmatrix} \begin{matrix} \\ \leftarrow \text{row } s \\ \leftarrow \text{row } t \\ \\ \\ \end{matrix} \\ & && W \geq 0, \end{aligned} \tag{15.7}$$

Indeed it not to hard to see that there is an optimal solution to linear program 15.7 in which each *wi* is either zero or one. A one represents a unit flow along a shortest path from *s* to *t*.
The dual linear program for the shortest-path problem is

$$\begin{aligned} &\text{Maximize} && z = x_s - x_t \\ &\text{subject to:} && x_i - x_j \leq c_{ij} \quad \text{for each } (i,j) \in E \\ & && X \text{ unrestricted.} \end{aligned} \tag{15.8}$$

The complementary slackness conditions (Theorem 15.8) are easy to interpret in the shortest-path problem. A path *W* and an assignment of variables *X* are jointly optimal if and only if
1. Each edge in the shortest path (i.e., a positive *wi* in the primal linear program 15.7) corresponds to equality in the corresponding constraint in the dual linear program 15.8, and
2. Each strict inequality in the dual linear program 15.8 corresponds to an edge not in the shortest path.

Page 433
For the graph in Figure 15.3 an optimal solution to the primal is

w(s,a)=0, *w(s,b)*=1, *w(s,c)*=0, w*(a,t)*=1, *w(b,a)*=1,
w(b,d)=0, *w(b,t)*=0, *w(c,b)*=0, *w(c,d)*=0, *w(d,t)*=0,

which has cost=9. In the dual we see by complementary slackness that this means

xs−xb=4
xb−xa=3
xa−xt=2

Summing these equations we obtain *z=xs−xt*=9.

Exercises

15.5.1 An enginer takes measurements of a variable *y(x);* the results are in the form of pairs *(xi,yi)*. The engineer wishes to find the straight line that fits this data best in the sense that the maximum vertical distance between any point *(xi,yi)* and the line is as small as possible. Formulate this as a linear program. Why might you decide to solve the dual?

15.5.2 Consider the node-edge incidence matrix *A* of the directed graph *G=(V,E)* as described in Section 15.5. Show that a set of $|V|-1$ columns is linearly independent if and only if the corresponding edges, when considered as undirected edges, form a tree. (Thus a basis corresponds to a tree.) If a network problem is

formulated with this graph, what does this reult say about the pivot step?
15.5.3 It was shown in Section 15.2 that the dual of

Maximize $z=cTX$
subject to: $AX=b$
$X\geq 0$.

has unrestricted variables. However, if some slack and/or surplus variables appear in *A,* show that the dual variable for a constraint having a slack variable is non-negative and the dual variable for a constraint having a surplus variable is non-positive. Hence, show that the only dual variables which are really unrestricted are those that correspond to constraints that were originally equations and not inequalities. Thus show that the dual of any linear program is esentially unique and is independent of the particular manner in which we write the primal.

15.6 The primal-dual algorithm

Consider a linear program in standard form

$$\begin{aligned} \text{Minimize} \quad & Z = c^T X \\ \text{subject to:} \quad & AX = b \\ & X \geq 0 \end{aligned} \tag{P}$$

and its dual

$$\begin{aligned} \text{Maximize} \quad & z = b^T W \\ \text{subject to:} \quad & A_1^T W \leq c_1 \\ & A_2^T W \leq c_2 \\ & \vdots \\ & A_n^T W \leq c_n \\ & W \text{ unrestricted.} \end{aligned} \tag{D}$$

We may assume that $b>0$, because the equalities in (P) can be multiplied by -1 where necessary. The complementary slackness conditions (Theorem 15.8) are: if X is a feasible solution to (P) and W is a feasible solution to (D), then X and W are both optimal if and only if

$$wi([Rowi(A)]TX-bi)=0 \text{ for all } i \tag{15.9}$$

and

$$(c_j - A_j^T W)x_j = 0 \text{ for all } j. \tag{15.10}$$

Condition 15.9 is automatically satisfied because of the equality in (P), so we will focus on condition 15.10. The main idea of the primal dual algorithm is:

Given a feasible solution W to (D), find a feasible solution X to (P) such that

$$X_j = 0, \textit{ whenever } A_j^T W < c_j.$$

In order to construct such a pair *W, X* we will iteratively improve *W,* while maintaining its feasibility in (D).

Suppose W is a feasible solution to (D). Then with respect to W some of the inequalities $A_j^T W \leq c_j$ in (D) still have slack and some do not. Let

$$J = \{j : A_j^T W = c_j\} = \{j_1, j_2, \ldots, j_{n'}\},$$

be the set of *admissible columns*. So X, a feasible solution to (P), is optimal if $xj=0$ for all $j \notin J$. Let

$$A_J = [A_{j_1}, A_{j_2}, \ldots, A_{j_{n'}}]$$

---

and

$$X_J = [x_{j_1}, x_{j_2}, \ldots, x_{j_{n'}}].$$

If we can find XJ such that

$$AJXJ=b \tag{15.11}$$
$$XJ=0,$$

then by complementary slackness, the X defined by

$$X[j] = \begin{cases} 0 & \text{if } j \notin J \\ X_{j_\ell} & \text{if } j = j_\ell \in J \end{cases}$$

is optimal in (P). To find *XJ* we construct a new linear program called the *restricted primal* (RP).

$$\begin{aligned} \textbf{Minimize} \quad & \zeta = \gamma \begin{bmatrix} X_J \\ Y \end{bmatrix} = y_1 + y_2 + \cdots + y_m \\ \textbf{subject to:} \quad & A_J X_J + Y = [A_J, I] \begin{bmatrix} X_J \\ Y \end{bmatrix} = b \\ & X_J \geq 0 \\ & Y \geq 0 \end{aligned} \qquad \text{(RP)}$$

where $Y=[y1,y2,\ldots,ym]^T$ are new variables one for each equation in (P) and

$$\gamma = [\underbrace{0,0,\ldots,0}_{|J|},\underbrace{1,1,\ldots,1}_{m}]$$

Let ζOPT be the optimal value of (RP) and suppose it occurs at $Y=Y$OPT, with basis *B*. If ζOPT=0, then *Y*OPT=0 and the corresponding *XJ* solves the constraints 15.11. Thus we have an optimal solution to (P). What happens when ζOPT>0?
The dual of (RP) is

$$\begin{aligned} \textbf{Maximize} \quad & z = b^T W \\ \textbf{subject to:} \quad & A_j^T W \leq 0 \\ & W \leq \overrightarrow{1} \\ & W \text{ unrestricted.} \end{aligned} \qquad \text{(DRP)}$$

where $\overrightarrow{1} = [1,1,1,\ldots,1]^T$. Let *W*OPT be the solution to (DRP) corresponding to *Y*OPT, that is *W*OPT=$B-T\gamma B$ (see Theorem 15.7). We call (DRP) the *dual-restricted primal*. The situation is that we tried to find a feasible *X* in (P) using only the columns in *J*, but failed. However, we do have the optimal feasible solution pair *Y*OPT, *W*OPTto (RP) and (DRP), respectively. We also know ζOPT>0,

---

the value of (RP) at *Y*OPT is positive. Let's try correcting *W* by a linear combination of the old *W* and *W*OPT. Let

$$W_\star = W + \theta W_{\text{OPT}}.$$

The value of the objective function at $W_\star$ is

$$b^T W_\star = b^T W + \theta b^T W_{\text{OPT}}.$$

Now we know bTWOPT=ζOPT>0, because *Y*OPT, *W*OPT is a primal-dual feasible solution pair. Thus to maximize the value of the objective function we can take $\theta>0$ and large. We also need to maintain feasibility so we need

$$A^T W_\star = A^T W + \theta A^T W_{\text{OPT}} \leq c$$

Consider the *j*-th equation

$$A_j^T W_\star = A_j^T W + \theta A_j^T W_{\text{OPT}} \leq c_j$$

If $A_j^T W_{\text{OPT}} \leq 0,$ then this equation is satisfied because *W* is feasible solution to (D). Thus, in particular, $W_\star$ is feasible, if $A_j^T W_{\text{OPT}} \leq 0$ for all *j*, but in this case we may take $\theta>0$ arbitrarily large and the value of the objective function at $W_\star$ will be unbounded. Therefore the primal (P) is infeasible, by Theorem 15.6. Hence:
THEOREM 15.9 If ζOPT>0 *in (RP) and W*OPT *the optimal solution to (DRP) satisfies*

$$A_j^T W_{\text{OPT}} \leq 0, \textit{ for all } j \notin J$$

then (P) is infeasible.
Therefore we need only worry when

$$A_j^T > 0 \text{ for some } j \notin J.$$

Consequently, the feasibility conditions are

$$A_j^T W_\star = A_j^T W + \theta A_j^T W_{\text{OPT}} \leq c_j, \text{ whenever } j \notin J \text{ and } A_j^T W_{\text{OPT}} > 0.$$

Thus for all $j \notin J$ and $A_j^T W_{OPT} > 0$ we need to chose θ such that

$$\theta \leq \frac{c_j - A_j^T W}{A_j^T W_{OPT}}$$

This yields the following theorem.

Page 437

THEOREM 15.10 When ζOPT>0 *in (RP) and there is a* $j \notin J$ *with* $A_j^T W_{OPT} > 0$, *the largest* θ *that maintains feasibility* $W_\star = W + \theta W_{OPT}$ *is*

$$\theta_\star = min \left\{ \frac{c_j - A_j^T W}{A_j^T W_{OPT}} : j \notin J \text{ and } A_j^T W_{OPT} > 0 \right\} \tag{15.12}$$

Given a feasible solution *W* to (D) Algorithm 15.6.1 constructs in *W* an optimal solution to (D) or determines if (P) is infeasible.

Algorithm 15.6.1: PRIMAL-DUAL(W)

FEASIBLE←**true**
OPTIMAL←**false**
while FEASIBLE **and not** OPTIMAL
do
- $J \leftarrow \{j : A_j^T W = c_j\}$
- **if** $|J| = n$
  - **then** OPTIMAL ← **true**
  - **else**
    - Solve (RP) by Simplex Algorithm
    - obtain solution with basis B and objective value ζ_{OPT}
    - **if** $\zeta_{OPT} = 0$
      - **then** OPTIMAL ← **true**
      - **else**
        - $W_{OPT} \leftarrow B^{-T} \gamma_B$
        - **if** $A_j^T W_{OPT} \leq 0$ for all $j \in J$
          - **then** FEASIBLE ← **false**
          - **else**
            - compute $\theta_\star$ using equation 15.12
            - $W \leftarrow W + \theta_\star W_{OPT}$

15.6.1 Initial feasible solution

In order to start Algorithm 15.6.1 we must first have a feasible solution *W* to (D). If *ci*≥0 for all *i,* we can take $W = \overrightarrow{0}$ as an initial feasible solution. When *ci*<0 for some *i,* we can use the following method to obtain a feasible solution *W* to (D). Introduce a new variable *x*0 to the primal problem (P), set *c*0=0 and add the constraint:

$$x0 + x1 + \ldots + xn = b0,$$

where *b*0 is taken to be larger than $\sum_i^n x_i$, for every basis solution *X*= [*x*1,...,*xn*] to (P). For example, take *b*0=*nM* where *M* is given in Lemma 15.11. This new primal is

Page 438

$$\begin{aligned} \text{Minimize} \quad & Z = c^T X = [\,0, c^T] \begin{bmatrix} x_0 \\ X \end{bmatrix} \\ \text{subject to:} \quad & x_0 + x_1 + \cdots + x_n = b_0 \\ & AX = b \\ & x_0 \geq 0, X \geq 0 \end{aligned} \tag{P'}$$

and its dual is

$$\begin{array}{lrcl} \textbf{Maximize} & z = w_0 b_0 + b^T W & & \\ \textbf{subject to:} & w_0 & \leq & 0 \\ & w_0 + A_1^T W & \leq & c_1 \\ & w_0 + A_2^T W & \leq & c_2 \\ & & \vdots & \\ & w_0 + A_n^T W & \leq & c_n \\ & W \text{ unrestricted.} & & \end{array} \qquad \text{(D')}$$

where *w*0 is the new variable corresponding to the new equation in the new primal. A feasible solution to this new dual is

$$W_i = \begin{cases} \text{MIN}\{c_1, c_2, \ldots, c_n\}, & \text{if } i = 0 \\ 0, & \text{if } i = 1, 2, \ldots, m \end{cases}.$$

Also [*x*0,*X*] is an optimal solution to the new primal (P') if and only if *X* is an optimal solution to (P). A suitable value for 60 is provided by the following lemma.

LEMMA 15.11 Let X=[*x*1,*x*2,...,*xn*] *be a basis solution to (P). Then*

$$|xj| \leq M = m! \alpha m - 1 \beta$$

where

$$a = \text{MAX}i,j\{|ai,j|\}$$

and

$$\beta = \text{MAX}\{b1, b2, \ldots, bm\}.$$

PROOF Without loss of generality we assume that the entries of *A, b,* and *c* are integers. If *X* is a basis solution, then there is a set *J* of *m* columns such that *X[j]*=0 whenever $j \notin J$, and $B = [A_j : j \in J]$ is nonsingular. Thus *BXJ*=*b,* where $X_J = [X[j] : j \in J]$ and so by Cramer's rule

$$x_j = \frac{\det(B')}{\det(B)},$$

Page 439

where *B′* is the matrix *B* with column *j* replaced with *b*. By integrality we have |det*(B)*|≥1, and so

$$\begin{aligned} |x_j| &\leq |\det(B')| \\ &= \sum_{\sigma \in S_m} \text{SIGN}(\sigma) \prod_{h=1}^{m} B'[h, \sigma(h)] \\ &\leq m! \alpha^{m-1} \beta, \end{aligned}$$

where *Sm* is the set of permutations of {1,2,...,*m*} and

$$\text{SIGN}(\sigma) = \begin{cases} +1, & \text{if } \sigma \text{ is an even permutation;} \\ -1, & \text{if } \sigma \text{ is an odd permutation.} \end{cases}$$

We now proceed to show that Algorithm 15.6.1 solves (P) in finitely many iterations.

THEOREM 15.12 Every admissible column in the optimal basis of (RP) remains admissible at the start of the next iteration.

PROOF Suppose that the optimal basis *B* of (RP) includes the column *Aj* Then by Corollary 15.7 the optimal solution to (DRP) corresponding to *X* is *W*OPT = *B*−*TγB,* where *γB* are the entries of

$$\gamma = [\underbrace{0, 0, \ldots, 0}_{|J|}, \underbrace{1, 1, \ldots, 1}_{m}]$$

corresponding to the columns of *[AJ,I]* that are in *B*. Consequently, for some *ℓ,* 1≤*ℓ*≤|*J*|,

$$\begin{aligned} A_j^T W_{\text{OPT}} &= A_j^T (B^{-T} \gamma_B) \\ &= (A_j^T B^{-T}) \gamma_B \\ &= (B^{-1} A_J)^T \gamma_B \\ &= E_\ell^T \gamma_B \\ &= \gamma_B[\ell] = 0, \end{aligned}$$

where $E\ell$=[0,0,...,0,1,0,0,...,0] T with 1 in position ℓ. This implies that
$A_j^T W_\star = A_j^T (W + \theta W_{\text{OPT}})$

Page 440

$$\begin{aligned} &= A_j^T W + \theta W_{\text{OPT}}^T A_j \\ &= \gamma_j + 0 \\ &= \gamma_j \end{aligned}$$

Thus if $j \in J$ at the start of an iteration of the while loop in Algorithm 15.6.1, then *j* remains in *J* in the next iteration.
How do we know that the primal-dual algorithm will terminate? If the minimum calculation of $\theta_\star$ by Equation 15.12 occurs at $j = j_\star$, then

$$\theta_\star = \frac{c_{j_\star} - A_{j_\star}^T W}{A_{j_\star}^T W_{\text{OPT}}}$$

and so

$$c_{j_\star} = A_{j_\star}^T W + \theta_\star A_{j_\star}^T W_{\text{OPT}} = A_{j_\star}^T W_\star.$$

Consequently, $j_\star$ becomes a new column of *J,* and we see that |*J*| monotonically increases. If *J*={1,2,...,*n*}, then *W* satisfies the complementary slackness condition in Equation 15.10 and is therefore optimal. Consequently we have the following theorem.
THEOREM 15.13 Algorithm 15.6.1 correctly solves (P) in at most n iterations.

15.6.2 The shortest-path problem

We illustrate the primal-dual method with the shortest-path problem introduced in Section 15.5. Observe that the row sum of the coefficient matrix in linear program 15.8 is zero. Hence any row may be omitted because it is redundant. If we choose to omit row t, the resulting linear program 15.13 is our primal.

$$\begin{aligned} &\text{Minimize} && Z = c^T X \\ &\text{subject to:} && AX = \begin{bmatrix} +1 \\ 0 \\ \vdots \\ 0 \end{bmatrix} \leftarrow \text{row } s \\ & && X \geq 0 \end{aligned} \tag{15.13}$$

The vector of flow is *X* and each entry of *X* is 0 or 1. The dual linear program for the shortest-path problem is

$$\begin{aligned} &\text{Maximize} && z = w_s \\ &\text{subject to:} && w_i - w_j \leq c_{ij} \text{ for each edge} (i,j) \in E \\ & && W \text{ unrestricted, except } w_t = 0. \end{aligned} \tag{15.14}$$

Page 441
We fix *wt*=0, because its row was omitted from the primal. The set of admissible columns (i.e., edges) is
J={*(i,j):xi xj=cij*}

and the restricted primal is

$$\begin{aligned}
&\text{Minimize} && \zeta = y_1 + y_2 + \cdots + y_{m-1} \\
&\text{subject to:} && AX + Y = \begin{bmatrix} +1 \\ 0 \\ \vdots \\ 0 \end{bmatrix} \leftarrow \text{row } s \\
& && x_j \geq 0 \text{ for all } j \in J \\
& && x_j = 0 \text{ for all } j \notin J \\
& && Y \geq 0.
\end{aligned} \tag{15.15}$$

Consequently, the dual-restricted primal is

$$\begin{aligned}
&\text{Maximize} && z = w_s \\
&\text{subject to:} && w_i - w_j \leq c_{ij} \text{ for each edge} (i,j) \in J \\
& && \overrightarrow{W \leq 1} \\
& && W \text{ unrestricted.}
\end{aligned} \tag{15.16}$$

It is easy to solve the dual-restricted primal. Note that $ws \leq 1$, and that we are maximizing ws, so we can try ws=1. If $wi=1$ and $(i,j) \in J$, then we can satisfy the constraint

$$wi - wj \leq 0,$$

by also setting $wj=1$. Hence if $P=i1, i2 \ldots ih$ is a path with $(i_a, i_{a+1}) \in J$ for all $a=1, 2, \ldots, h-1$ and $w_{i_1} = 1$. Then we can set $w_{i_a} = 1$ for each a without violating the constraints. Hence if there is no st-path using edges only in J, then an optimal solution to the dual-restricted primal is given by

$$W_{\text{OPT}}[i] = \begin{cases} 1, & \text{if there is an } si\text{-path using only edges in } J \\ 0, & \text{if there is an } it\text{-path using only edges in } J \\ 1, & \text{for all other vertices } i \end{cases}.$$

We then calculate

$$\theta_\star = \text{MIN}\{c_{ij} - (w_i - w_j) : (i,j) \notin J \text{ and } W_{\text{OPT}}[i] - W_{\text{OPT}}[j] > 0\}$$

and update W and J, and obtain and solve the new dual-restricted primal. If we get to a point where there is an st-path using edges in J, then $ws=0$ and we are

Page 442

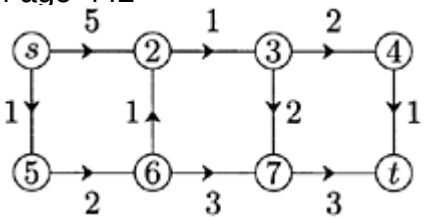


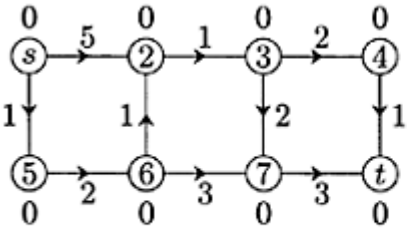
FIGURE 15.4
Shortest-path example

at an optimal solution, because minimum ζ is equal to the maximum z which is $ws=0$. Notice that any st-path that uses only edges in J is optimal.

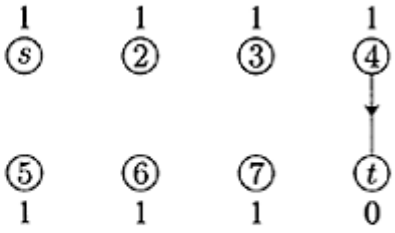
The primal-dual algorithm reduces the shortest-path problem to the easy calculation which vertices are reachable from the source s.

We illustrate the primal-dual algorithm when applied to the shortest-path problem with the directed graph given in Figure 15.4.

Iteration 1

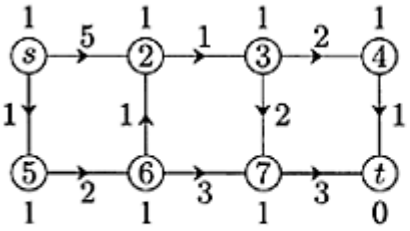


(D): $W = [0,0,0,0,0,0,0,0]$
$J = \{\}$

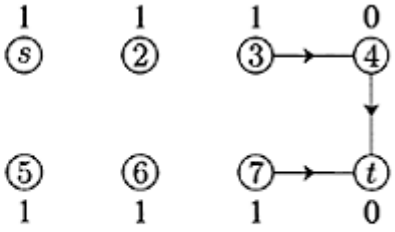


(DRP): $W_{OPT} = [1,1,1,1,1,1,1,0]$
$\theta^* = 1$ for edge $(4,7)$

Iteration 2

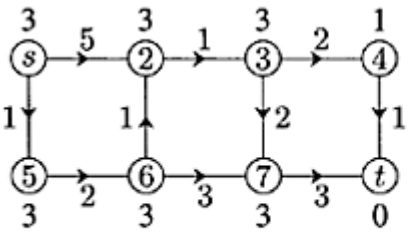


(D): $W = [1,1,1,1,1,1,1,0]$
$J = \{(4,7)\}$

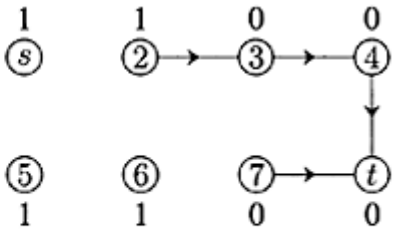


(DRP): $W_{OPT} = [1,1,1,0,1,1,1,0]$
$\theta^* = 2$ for edges $(3,4)$ and $(7,t)$

Iteration 3

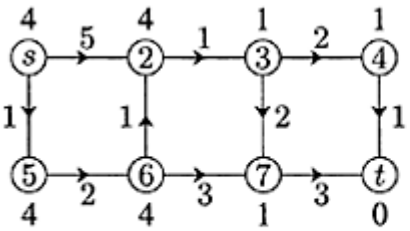


(D): $W = [3,3,3,1,3,3,3,0]$
$J = \{(4,t),(3,4),(7,t)\}$

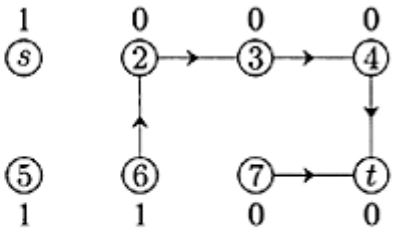


(DRP): $W_{OPT} = [1,1,0,0,1,1,0,0]$
$\theta^* = 1$ for edge $(2,3)$

Iteration 4

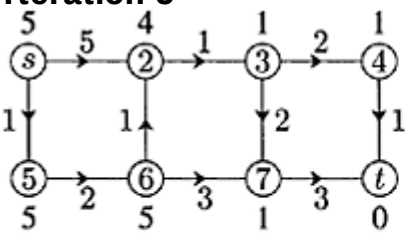


(D): $W = [4,4,1,1,4,4,1,0]$
$J = \{(4,t),(3,4),(7,t),(2,3)\}$

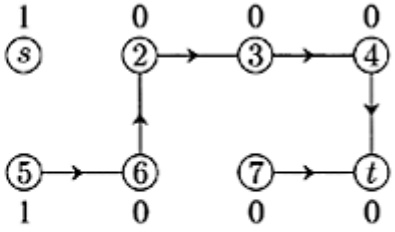


(DRP): $W_{OPT} = [1,0,0,0,1,1,0,0]$
$\theta^* = 1$ for edge $(6,2)$

Iteration 5

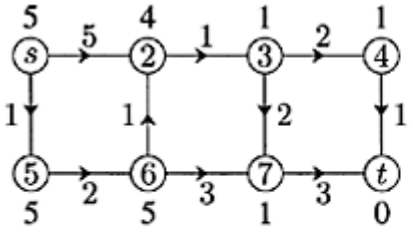


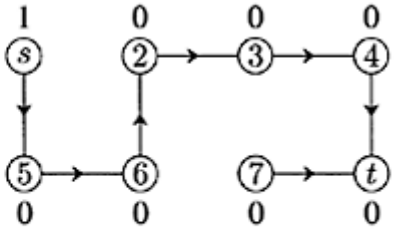
(D): $W = [5,4,1,1,5,5,1,0]$
$J = \{(4,t),(3,4),(7,t),(2,3),(6,2)\}$



(DRP): $W_{OPT} = [1,0,0,1,0,0,0,0]$
$\theta^* = 2$ for edge $(5,6)$

Iteration 6





(D): $W = [7, 4, 1, 3, 7, 7, 1, 0]$

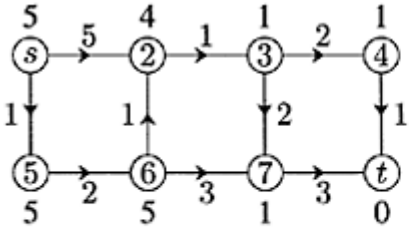
$$J = \left\{ \begin{matrix} (4,t), (3,4), (7,t), (2,3), \\ (6,2), (5,6) \end{matrix} \right\}$$

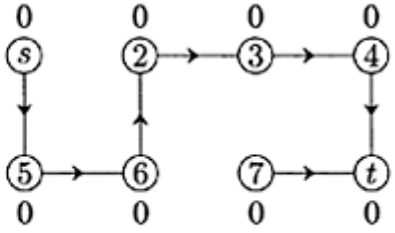
(DRP): $W_{OPT} = [1, 0, 0, 0, 0, 0, 0, 0]$

$\theta^* = 1$ for edge $(s, 5)$

Page 444

Iteration 7





(D): $W = [8, 4, 1, 3, 7, 7, 1, 0]$

$$J = \left\{ \begin{matrix} (4,t), (3,4), (7,t), (2,3), \\ (6,2), (5,6), (s,5) \end{matrix} \right\}$$

(DRP): $W_{OPT} = [0, 0, 0, 0, 0, 0, 0, 0]$

15.6.3 Maximum flow

A network *N* with source *s* and target *t* can be described as a linear program as follows:

$$\begin{aligned} &\text{Maximize} \quad v \\ &\text{subject to:} \quad Af = \begin{bmatrix} +v \\ -v \\ 0 \\ \vdots \\ 0 \end{bmatrix} \begin{matrix} \\ \leftarrow \text{row } s \\ \leftarrow \text{row } t \\ \\ \\ \\ \end{matrix} \\ &\qquad f \leq c \\ &\qquad f \geq 0, \end{aligned} \tag{15.17}$$

where u=VAL(f) is the value of the flow f and $c = [\text{CAP}(e) : e \in E(N)]^T$ are the capacities of the edges e in N. This is of course equivalent to

$$\begin{aligned} &\text{Maximize } u \\ &\text{subject to: } Af + uT \leq 0 \\ &\qquad f \leq c \\ &\qquad -f \leq 0, \end{aligned} \tag{15.18}$$

where

$$T[x] = \begin{cases} -1, \text{ if } x = s \\ +1, \text{ if } x = t \\ 0, \text{ otherwise} \end{cases}$$

(Note that $Af + uT \leq 0$ implies $Af + uT = 0$ because we maximize u.) The set of admissible columns (i.e., edges) is

$$J = \{\overrightarrow{uv} : f(\overrightarrow{uv}) = c \text{ or } -f(\overrightarrow{uv}) = 0\}.$$

Page 445

Thus J is the set of saturated edges together with the zero-flow edges and the dual-restricted primal is:

$$\begin{aligned}
&\text{Maximize} \quad v \\
&\text{subject to:} \quad Af + vT \leq 0 \\
&\qquad f(\overrightarrow{uv}) \leq 0, \text{for all saturated edges } \overrightarrow{uv} \text{ in 15.18} \\
&\qquad -f(\overrightarrow{uv}) \leq 0, \text{for all zero flow edges } \overrightarrow{uv} \text{ in 15.18} \\
&\qquad f \leq \overrightarrow{1} \\
&\qquad v \leq \overrightarrow{1}
\end{aligned}$$

It is easy to solve the dual-restricted primal in 15.19. We wish to maximize *u* and so we can try *u*=1. Thus we must choose a flow *f* such that

$$[Rows(A)]f=1$$

Concerning the edges incident to *s,* the inequalities in 15.19 tell us that we can set $f(\overrightarrow{su}) = 1$ if the edge $\overrightarrow{su}$ has zero flow or is unsaturated and we can set $f(\overrightarrow{us}) = -1$ if the edge $\overrightarrow{us}$ is saturated and/or does not have zero flow. Let *S*1 be the set of vertices *u* incident to *s* satisfying if $u \longrightarrow s$, then $\overrightarrow{su}$ has zero flow or is unsaturated; if $u \longrightarrow s$, then $\overrightarrow{us}$ is saturated or does not have zero flow.

Hence we choose $v_1 \in S_1$ and set the flow on the associated edge to 1 if it leaves *s* and to −1 if it enters *s*. Now we must consider [*Row*u1]*(A)* and consider the edges incident to *u*1. Let *S*2 be the set of vertices *u* incident to some *u* in *S*1 satisfying

$$\left.\begin{array}{l}\text{if } u \longrightarrow s, \text{ then } \overrightarrow{su} \text{ has zero flow or is unsaturated;} \\ \text{if } u \longrightarrow s, \text{ then } \overrightarrow{us} \text{ is saturated or does not have zero flow.}\end{array}\right\} \tag{15.20}$$

In general let *Sk*+1 be the set of vertices *u* incident to some *u* in *Sk* satisfying conditions in 15.20 and continue until some *Sk* contains the target *t*. When and if it does, we can choose vertices $v_i \in S_i, i$=1,2, …,*k*−1 such that *s*u1u2u3…u*k*−1*t* is a *st*-path *P*. We obtain an optimal solution *f*OPT to the dual-restricted primal in 15.19 as follows:

$$f_{\text{OPT}}(\overrightarrow{uv}) = \begin{cases} +1, & \text{if } \overrightarrow{uv} \text{ is a forward edge of } P \\ -1, & \text{if } \overrightarrow{uv} \text{ is a backward edge of } P \\ 0, & \text{if } \overrightarrow{uv} \text{ is not an edge of } P \end{cases}$$

We then calculate

$$\theta_\star = \text{MIN}_{uv \in E(P)} \begin{cases} (c(\overrightarrow{uv}) - f(\overrightarrow{uv})), & \text{if } f_{\text{OPT}}(\overrightarrow{uv}) = 1 \\ (0 - (-f(\overrightarrow{uv})), & \text{if } f_{\text{OPT}}(\overrightarrow{uv}) = -1 \end{cases}$$

$$\begin{aligned}
&= \text{MIN}_{uv \in E(P)} \begin{cases} (\text{CAP}(\overrightarrow{uv}) - f(\overrightarrow{uv})), & \text{if } f_{\text{OPT}}(\overrightarrow{uv}) = 1 \\ f(\overrightarrow{uv}), & \text{if } f_{\text{OPT}}(\overrightarrow{uv}) = -1 \end{cases} \\
&= \text{MIN}\{\text{RESCAP}(uv) : uv \in E(P)\},
\end{aligned}$$

where

$$\text{RESCAP}(uv) = \begin{cases} \text{CAP}(\overrightarrow{uv}) - f(\overrightarrow{uv}), & \text{if } uv \text{ is a forward edge,} \\ f(\overrightarrow{vu}), & \text{if } uv \text{ is a backward edge.} \end{cases}$$

We update the flow by

$$f \leftarrow f + \theta_\star f_{\text{OPT}},$$

recompute the set of admissible edges *J* to get the new dual-restricted primal and repeat until *J* is empty at which point the flow *f* will be maximum.

It is now easy to see the similarity of this method to that in Section 8.2 and realize that the primal-dual algorithm for network flow is exactly the Ford-Fulkerson algorithm.

Exercises

15.6.1 Consider Problem 15.2 the Weighted Matching problem.

Problem 15.2: Weighted Matching

Instance: undirected graph G and weight $we \geq 0$ for each edge e of G.
Find: a matching M of G with maximal possible weight

$$\text{WT}(M) = \sum_{e \in E(M)} w_e.$$

Formulate a primal-dual algorithm for solving Problem 15.2 and give an interpretation for the restricted primal.
15.6.2 Use the primal-dual algorithm as discussed in Section 15.6.2 to find a shortest path between each pair of nodes in the graph given in Figure 15.5.

15.7 Notes

The primal-dual algorithm was first described in 1956 by DANTZIG; see [33]. Our treatment is similar to that of PAPADIMITRIOU and STIEGLITZ; see [94].

page_446

Page 447

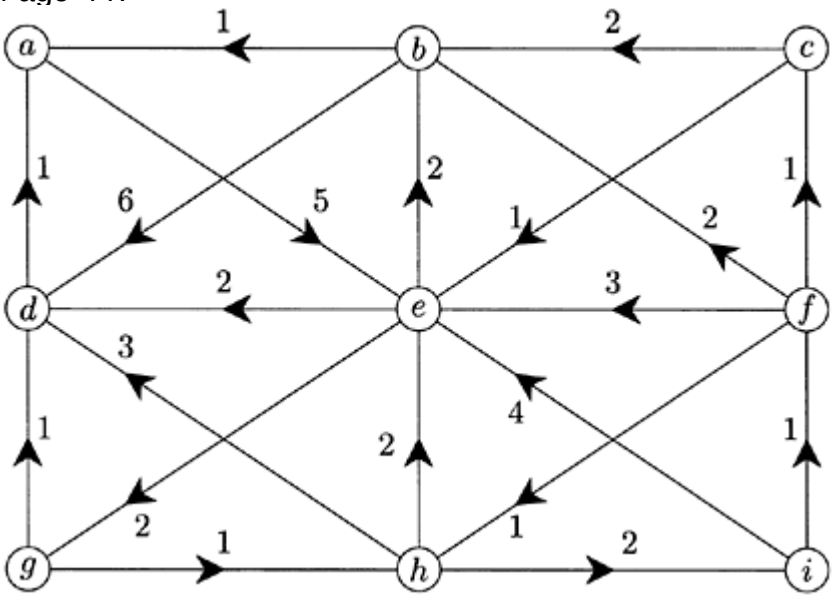


FIGURE 15.5
An instance of the shortest-path problem

page_447

Page 448
This page intentionally left blank.

page_448

Page 449
16
Discrete Linear Programming

16.1 Introduction

An *integer linear program* (ILP) is a linear program in which the variables have been constrained to be integers.

$$\begin{aligned} &\text{Minimize: } Z = cTX && (16.1)\\ &\text{subject to: } AX \leq b\\ &\qquad X > 0\\ &\qquad X \text{ integral.} \end{aligned}$$

If all of the variables are each constrained to a finite set of values, we say that the integer linear program is discrete. Notice that frequently the equality constraints force the variables to be discrete, for if $bi/ai, j>0$ for all j in the constraint

$$\sum_{i=1}^{n} a_{i,j} x_j = b_i,$$

then xj cannot exceed $m_j = \lfloor b_i / a_i, j \rfloor$. Hence $x_j \in \{0, 1, 2, \ldots, m_j\}$ for all j.
DEFINITION 16.1: A *discrete linear program* (DLP) is an integer linear program in which the variables are a

bounded

$$\begin{array}{c} \text{Minimize: } Z = cTX \\ \text{subject to: } AX = b \\ 0 \le xj \le mj,\ j = 1, 2, \ldots, n \\ X \text{ integral.} \end{array} \qquad (16.2)$$

Consider Problem 16.1, the Knapsack problem. This problem is motivated by what to carry on a proposed hiking trip. The weight limit on how much can be carried is the capacity *M*. Each of the *n* objects under consideration have a

certain weight *wi* and each has a certain value or profit *pi, i*=1, 2,..., *n*−1. Furthermore each object can be either carried or left behind. We cannot choose to carry a fraction of an object.

Problem 16.1: Knapsack

Instance: *profits* $p1, p1, p2, \ldots, pn;$
weights $w1, w1, w2, \ldots, wn;$ and
capacity M;

Find: the maximum value of

$$P = \sum_{i=1}^{n} p_i x_i$$

subject to

$$\sum_{i=1}^{n} w_i x_i \le M$$

and $[x_1, \ldots, x_n] \in \{0, 1\}^n.$

This problem can be formulated as the discrete linear program:

$$\begin{array}{c} \text{Minimize: } Z = -(p1x1 + p1x1 + \ldots + pnxn) \\ \text{subject to: } w1x1 + w1x1 + \ldots + wnxn \le M \\ 0 \le xj \le 1,\ j = 1, 2, \ldots, n \\ X \text{ integral.} \end{array}$$

16.2 Backtracking

A discrete integer linear program can be solved with a backtracking search. For example, the Knapsack problem can be solved with Algorithm 16.2.1.

Algorithm 16.2.1: KNAPSACK 1(ℓ)

if $\ell > n$

then
- **if** $\sum_{i=1}^{n} w_i x_i \le M$
  - **then**
    - $CurrentProfit \leftarrow \sum_{i=1}^{n} p_i x_i$
    - **if** $CurrentProfit > OptimalProfit$
      - **then**
        - $OptimalProfit \leftarrow CurrentProfit$
        - $OptimalX \leftarrow [x_1, \ldots, x_n]$

else
- $x_\ell \leftarrow 1$
- KNAPSACK1$(\ell + 1)$
- $x_\ell \leftarrow 0$
- KNAPSACK1$(\ell + 1)$

Initially Algorithm 16.2.1 is started with ℓ=1.

In general the backtracking method to solve a discrete integer linear program is performed by computing for a partial solution, in which values for *x*1, *x*2,..., *xℓ*−1 have been assigned, the set of possible values *Cℓ* for *xℓ*. Each possible value is examined to see whether the partial solution can be extended with it. The general backtracking algorithm is provided in Algorithm 16.2.2

Algorithm 16.2.2: BACKTRACK(ℓ)

if [$x1$, $x1$,..., $x\ell$] is a feasible solution

then process it

Compute $C\ell$

for each $x \in C_\ell$

do $\begin{cases} x_\ell \leftarrow x \\ \text{BACKTRACK}(\ell+1) \end{cases}$

Algorithm 16.2.1, is an application of Algorithm 16.2.2 with $C\ell$={1,0}. We can improve the running time of a backtracking algorithm if we can find efficient ways of reducing the size of the choice sets $C\ell$ This process is called *pruning*.

For the Knapsack problem, one simple method is to observe that we must have

$$\sum_{i=1}^{\ell} w_i x_i \leq M$$

for any partial solution [$x1$, $x2$,..., $x\ell-1$]. In other words, we can check partial solutions to see if the feasibility condition is satisfied. Consequently, if $\ell \leq n$ and

---

Page 452

we set

$$CurrentWt = \sum_{i=1}^{\ell-1} w_i x_i,$$

then we have

$$C_\ell = \begin{cases} \{1,0\}, & \text{if } CurrentWt + w_\ell \leq M; \\ \{0\}, & \text{otherwise.} \end{cases}$$

Using Algorithm 16.2.2 as a template, we obtain Algorithm 16.2.3, which is invoked with ℓ=1 and *CurrentWt*=0.

Algorithm 16.2.3: KNAPSACK2(ℓ)

if $\ell > n$

then $\begin{cases} \textbf{if } \sum_{i=1}^{n} p_i x_i > OptimalProfit \\ \quad \textbf{then} \begin{cases} OptimalProfit \leftarrow \sum_{i=1}^{n} p_i x_i \\ OptimalX \leftarrow [x_1, \ldots, x_n] \end{cases} \end{cases}$

if $\ell = n$

then $C_\ell \leftarrow \emptyset$

else $\begin{cases} \textbf{if } CurrentWt + w_\ell \leq M \\ \quad \textbf{then } C_\ell \leftarrow \{1,0\} \\ \quad \textbf{else } C_\ell \leftarrow \{0\} \end{cases}$

for each $x \in C_\ell$

do $\begin{cases} x_\ell \leftarrow x \\ CurrentWt = CurrentWt + w_\ell x_\ell \\ \text{KNAPSACK2}(\ell+1) \\ CurrentWt = CurrentWt - w_\ell x_\ell \end{cases}$

A backtracking algorithm with simple pruning for solving the discrete integer linear program 16.2 in which the coefficient matrix A=[Ai, $A2$,..., An] and b consists of only non-negative entries is given as Algorithm 16.2.4.

Page 453

Algorithm 16.2.4: (16.3) BACKTRACK2(ℓ)

```
                    if ℓ>n
                then { if Σ_{i=1}^{n} c_i x_i < OptimalZ
                     { then { OptimalZ ← Σ_{i=1}^{n} c_i x_i
                            { OptimalX ← [x_1, ..., x_n]
        comment: Compute Cℓ
                    if ℓ=n
                then C_ℓ ← ∅
                else { C_ℓ ← {}
                     { for x = 0 to m_j
                     { do { if CurrentWt[i] + A_i x ≤ b
                          { then C_ℓ ← C_ℓ ∪ {x}
                for each x ∈ C_ℓ
                do { x_ℓ ← x
                   { CurrentWt = CurrentWt + A_i x_ℓ
                   { BACKTRACK2(ℓ + 1)
                   { CurrentWt = CurrentWt − A_i x_ℓ
```

16.3 Branch and bound

Another strategy for solving an integer linear program is to first ignore the integer constraints and solve the corresponding linear program. This linear program is called the *relaxation* of the integer linear program. If a solution is found to the relaxed integer linear program, it can be rounded to the nearest integer solution. Although this may seem plausible, it generally leads to solutions that are either infeasible or far from the optimal solution. We illustrate this difficulty with the following integer linear program:

$$\begin{aligned} &\text{Minimize: } Z = -x1 - x2 \qquad (16.3)\\ &\text{subject to: } -2x1 + 7x2 \le 14\\ &\qquad 2x1 - 2x2 \le 1\\ &\qquad x1,\ x2 \ge 0\\ &\qquad x1,\ x2 \text{ integral.} \end{aligned}$$

This problem is solved graphically in Figure 16.1 and we see that there are six

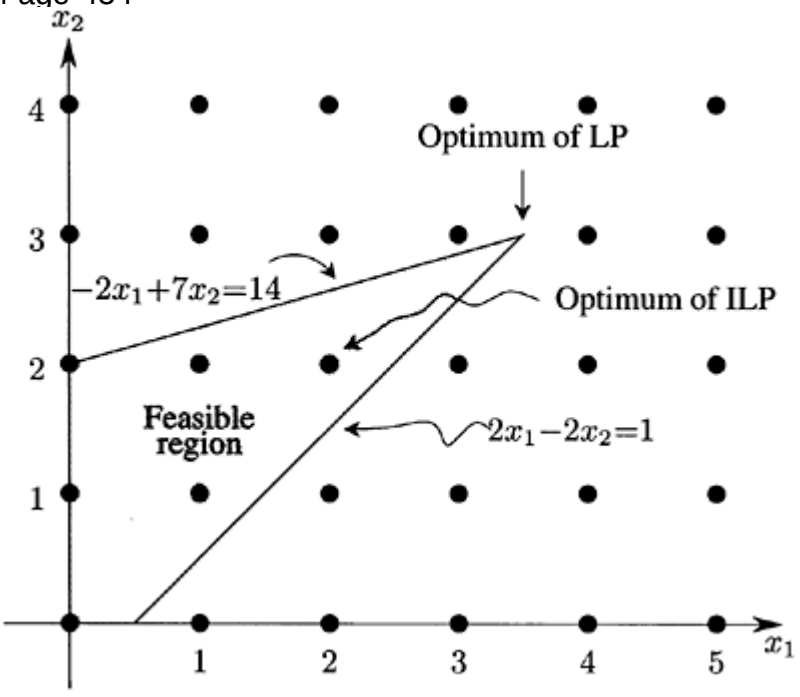


FIGURE 16.1
The integer points closest to the optimal solution to the standard linear program are infeasible in the integer linear program.

feasible solutions to this integer linear program: (0, 0), (0, 1), (1, 1), (0, 2), (1, 2) and (2, 2). The value of the object function, respectively, at these points is: 0, −1, −2, −2, −3, and −4. Hence (2, 2) is the optimal solution to this integer linear program. On the other hand, the optimal solution to the relaxed integer linear program is (3.5, 3) and the value of the objective function at this point is −6.5. The nearest integer points are (3, 3) and (4,3) having values −6 and −7. Neither of these points are feasible and the value of the objective function at them is far from the value at the true optimal solution.

Thus in practice the relaxation strategy can result in misleading information. However, not all is lost. It is not to difficult to see that a bound on the value of the objective function is obtained.

THEOREM 16.1 Let $\widehat{X}$ *be an optimal solution to relaxation of the integer linear program*

$$\begin{aligned} &\text{Minimize: } Z = cTX \\ &\text{subject to: } AX = b \\ &\qquad X \geq 0 \\ &\qquad X \text{ integral} \end{aligned} \tag{16.4}$$

Page 455

Then any solution X to the integer linear program 16.4 satisfies

$$c^T X \leq c^T \widehat{X}$$

PROOF If *X* is a feasible solution to the integer linear program, it is also a feasible solution to the linear program, in which the integer constraints have been removed.

If values have been assigned to each of *x*1, *x*2,..., *xℓ*−1, then the remaining variables *xℓ, xℓ*+1,..., *xn* must satisfy

$$\begin{aligned} \text{Minimize:} \quad & \widehat{Z} = \widehat{c}^T \widehat{X} \\ \text{subject to:} \quad & \widehat{A}\widehat{X} \leq \widehat{b} \\ & \widehat{X} \geq 0 \\ & \widehat{X} \quad \text{integral,} \end{aligned}$$

where

$$\begin{aligned} \widehat{A} &= [A_\ell, A_{\ell+1}, \ldots, A_n] \\ \widehat{X} &= [x_\ell, x_{\ell+1}, \ldots, x_n] \\ \widehat{c} &= [c_\ell, c_{\ell+1}, \ldots, c_n] \\ \widehat{b} &= b - \mathit{CurrentWt} \end{aligned}$$

and

$$\mathit{CurrentWt} = \sum_{j=1}^{\ell-1} x_j A_j.$$

(Here *Aj* is the *jth* column of the coefficient matrix *A*.) Thus according to Theorem 16.1, any feasible solution *X* that extends *x*1, *x*2,..., *xℓ*−1 has value *Z*=*cTX* no larger than

$$B = \mathit{CurrentZ} + \lfloor \widehat{Z}_{\text{OPT}} \rfloor,$$

where $\widehat{Z}_{\text{OPT}}$ is value of the objective function $\widehat{Z} = \widehat{c}^T \widehat{X}$ for the linear program

$$\begin{aligned} \text{Minimize:} \quad & \widehat{Z} = \widehat{c}^T \widehat{X} \\ \text{subject to:} \quad & \widehat{A}\widehat{X} \leq \widehat{b} \\ & \widehat{X} \geq 0. \end{aligned}$$

Consequently, if a feasible solution *X*0 has already been obtained that has value *Z*=*cTX*0≤*B,* then no extension of the given assignment to the variables *x*1, *x*2,..., *xℓ*−1 will lead to an improved *Z* value. Hence the search for such extensions can be aborted. This method is known as *branch and bound* and is recorded as Algorithm 16.3.1, which we start with *ℓ*=1, *CurrentWt*=0, *CurrentZ*=0, and *OptimalZ*=∞.

Page 456

Algorithm 16.3.1: BACKTRACK3(ℓ)

if $\ell > n$

then
- **if** $CurrentWt = b$
  - **then**
    - **if** $CurrentZ < OptimalZ$
      - **then**
        - **for** $j = 1$ **to** n **do** $OptimalX[j] \leftarrow X[j]$
        - $OptimalZ \leftarrow CurrentZ$
    - **return**

Compute $C\ell$

Use the simplex algorithm to solve the linear program:

$$\begin{array}{lrcl} \text{Minimize:} & \widehat{Z} = \widehat{c}^T \widehat{X} & & \\ \text{subject to:} & \widehat{A}\widehat{X} & = & \widehat{b}, \text{ where} \\ & \widehat{X} & \geq & 0 \end{array} \quad \begin{cases} \widehat{A} = [A_\ell, A_{\ell+1}, \ldots, A_n] \\ \widehat{X} = [x_\ell, x_{\ell+1}, \ldots, x_n] \\ \widehat{c} = [c_\ell, c_{\ell+1}, \ldots, c_n] \\ \widehat{b} = b - CurrentWt \end{cases}$$

if the linear program has optimal value $\widehat{Z}_{OPT}$ at $\widehat{X}_{OPT}$

then
- $B \leftarrow CurrentZ + \lfloor \widehat{Z}_{OPT} \rfloor$
- **if** $\widehat{X}_{OPT}$ is integer valued
  - **then**
    - **if** $B < OptimalZ$
      - **then**
        - **for** $i \leftarrow 1$ **to** $(\ell - 1)$ **do** $OptimalX[i] \leftarrow X[i]$
        - **for** $i \leftarrow \ell$ **to** n **do** $OptimalX[i] \leftarrow \widehat{X}_{OPT}[i]$
        - $OptimalZ \leftarrow B$
  - **else**
    - **for each** $x \in C_\ell$
      - **do**
        - **if** $B \geq OptimalZ$ **then return**
        - $x_\ell \leftarrow x$
        - $CurrentWt = CurrentWt + A_\ell x_\ell$
        - $CurrentZ = CurrentZ + c_\ell x_\ell$
        - BACKTRACK3$(\ell + 1)$
        - $CurrentWt = CurrentWt - A_\ell x_\ell$
        - $CurrentZ = CurrentZ - c_\ell x_\ell$

if the linear program is unbounded

then
- **for each** $x \in C_\ell$
  - **do**
    - $x_\ell \leftarrow x$
    - $CurrentWt = CurrentWt + A_\ell x_\ell$
    - $CurrentZ = CurrentZ + c_\ell x_\ell$
    - BACKTRACK3$(\ell + 1)$
    - $CurrentWt = CurrentWt - A_\ell x_\ell$
    - $CurrentZ = CurrentZ - c_\ell x_\ell$

if the linear program is infeasible **then return**

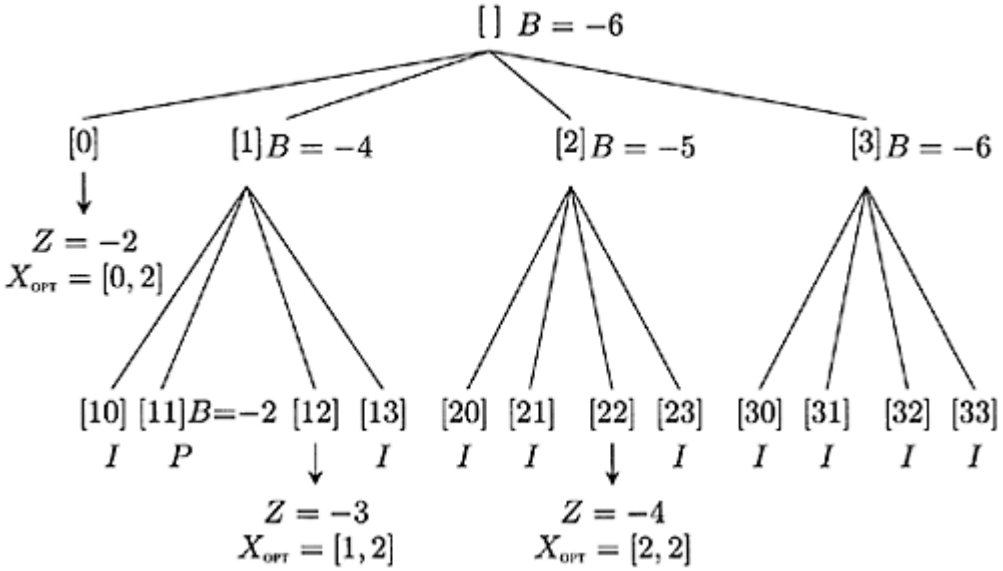


FIGURE 16.2
The backtracking state space search tree that results when Algorithm 16.3.1 is applied to the integer linear program 16.3.

By way of example we provide the backtracking state space search tree in Figure 16.2 that results when Algorithm 16.3.1 is applied to the integer linear program 16.3. Observe that adding the two constraints of linear program 16.3 we see that $x2 \le 3 = m2$ and thus $x1 \le 3 = m1$ as well. These bounds means that the linear program is a discrete linear program and indeed we can apply Algorithm 16.3.1. Each leaf-node in Figure 16.2 is decorated with *I, P,* or an arrow. The *I* indicates that the corresponding reduced integer linear program is infeasible, the *P* indicates pruning by a previously obtained feasible solution and an arrow indicates that a new optimal solution XOPT has been found. Its value $Z = cTX$OPT is also given. For this example the final optimal solution is XOPT=[2,2] with value $Z=-4$. This agrees with our previous analysis.

As a second example consider the linear program

$$\begin{aligned} &\text{Minimize: } Z = -2x1 - 3x2 - x3 - 2x4 \qquad (16.5)\\ &\text{subject to: } x1 + 2x2 + 3x3 + 2x4 \le 8\\ &\qquad -3x1 + 4x2 + x3 + 3x4 \le 8\\ &\qquad 3x1 - 4x2 - 6x3 - 10x4 \le -20\\ &\qquad x1, x2, x3, x4 \ge 0\\ &\qquad x1, x2, x3, x4 \text{ integral.} \end{aligned}$$

The constraint $x1 + 2x2 + 3x3 + 2x4 \le 8$ provides the upper bounds $m1=8$, $m2=4$, $m3=2$, and $m4=4$, on the variables $x1$, $x2$, $x3$, and $x4$, respectively. Thus we can take $C1=\{0,1,\ldots,8\}$, $C2=C4=\{0,1,\ldots,4\}$, and $C3=\{0,1,2\}$. The backtracking state space search tree is given in Figures 16.3 and 16.4.

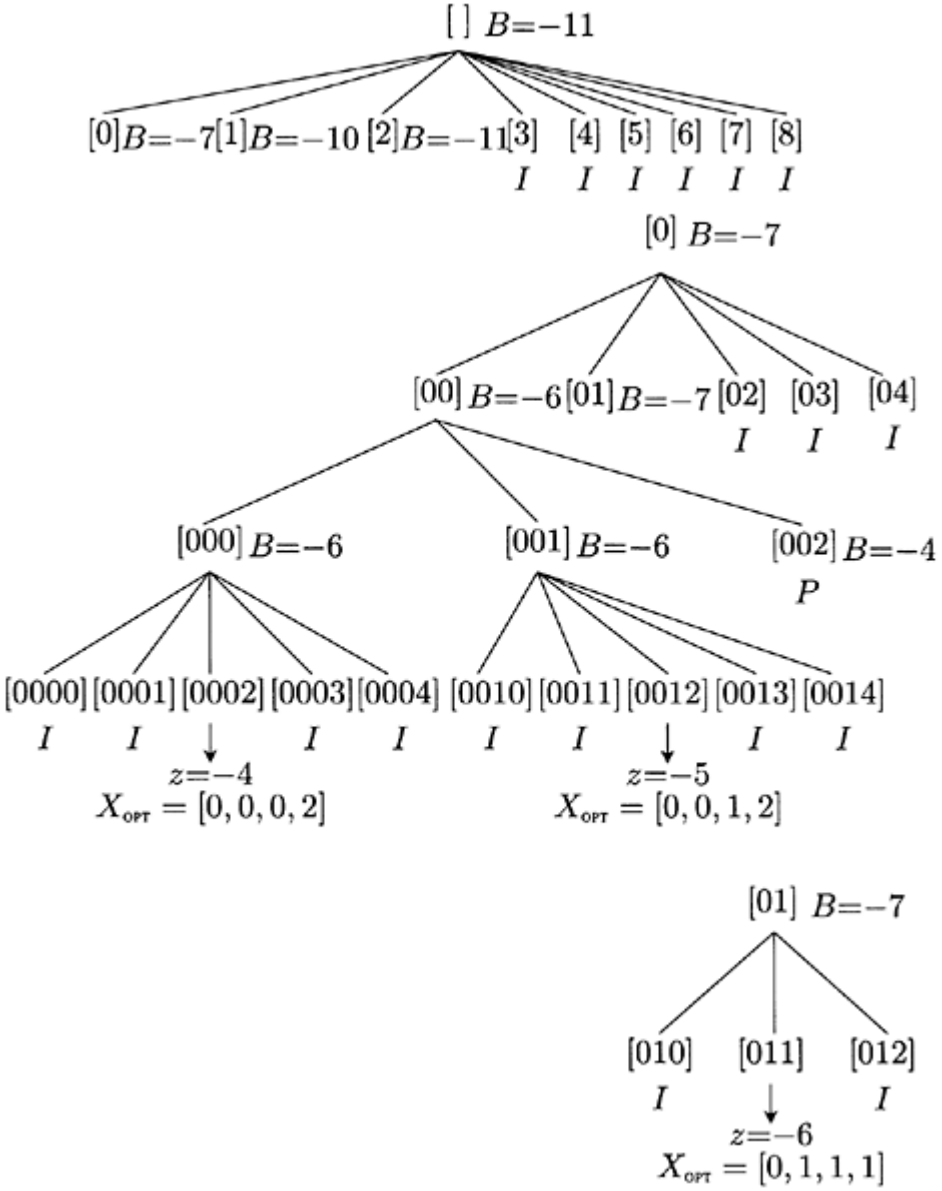


FIGURE 16.3
The first part of the backtracking state space search tree that results when Algorithm 16.3.1 is applied to the integer linear program 16.5. So far we see that X_{OPT}=[0, 1, 1, 1] gives z=−6.

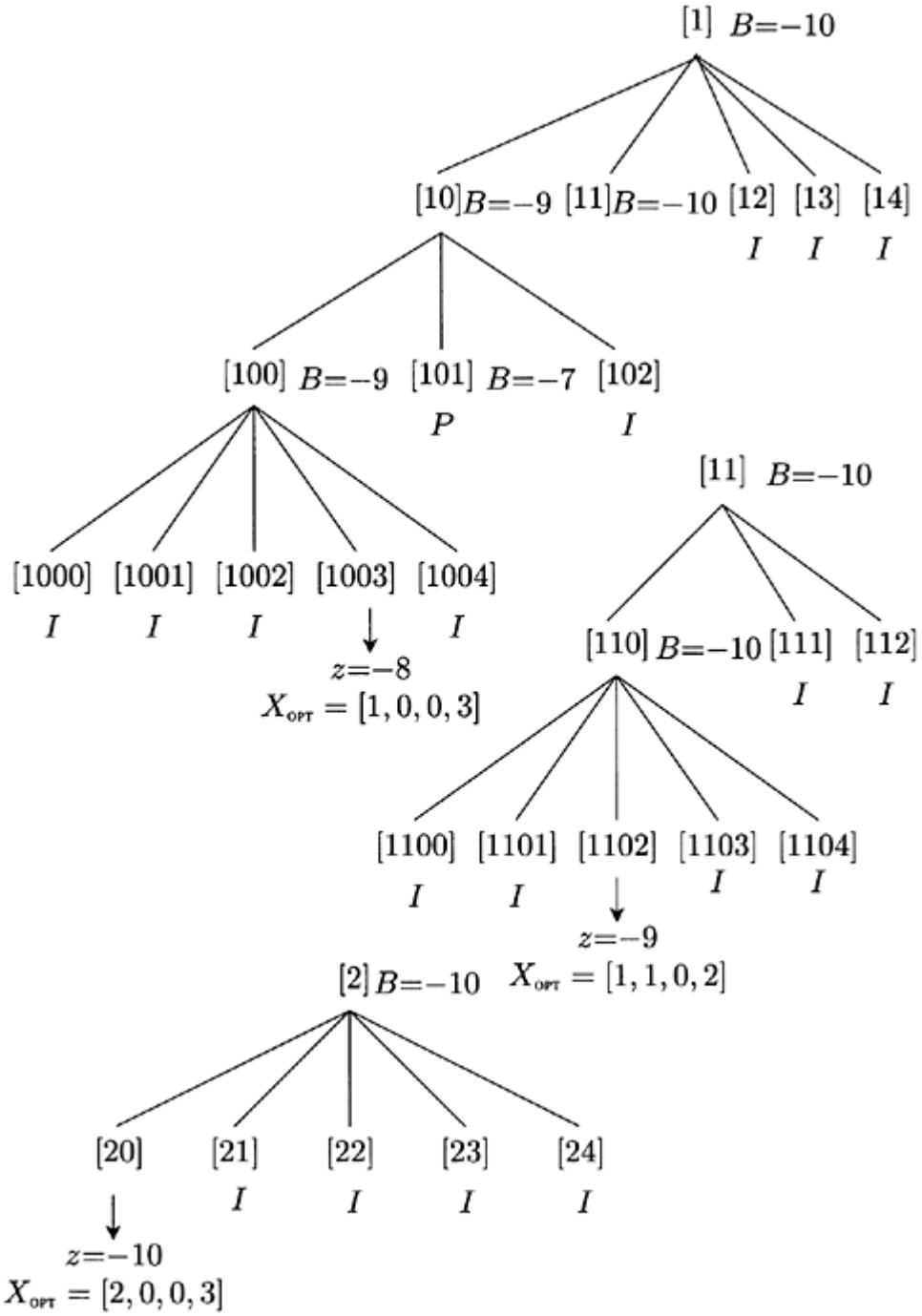


FIGURE 16.4
Continuation of Figure 16.3. Notice the subtree with root [101] is pruned because [1003] gave $z=-8$. The final solution appears at [20] and is XOPT=[2, 0, 0, 3] with objective value $z=-10$.

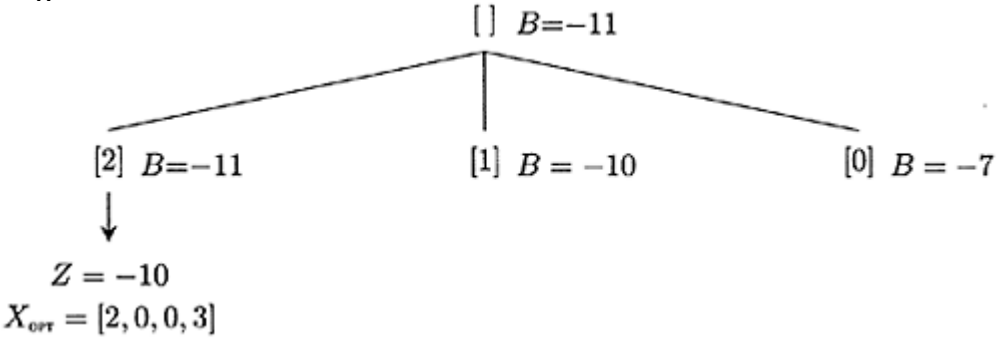


FIGURE 16.5
Dramatic pruning when the subtrees are processed in order of the bounds
Notice in this second example, that if we were to process the subtree with root [2] prior to the subtree [0] and [1], pruning would be dramatic and the search would be quite short. This is because the bound obtained when the relaxed linear program with $x1=2$ leads to a bound of -10, and there is an integer solution that achieves this bound, namely, $X=[2, 0, 0, 3]$ in the subtree with root [2]. The bound obtained when the root is [0] and [1] is -7 and -10, respectively, so these subtrees will be pruned if [2] is examined first. The search tree with this type of pruning is given in Figure 16.5.
One way to implement this type of pruning is as follows. After values for $x1$, $x2$,..., $x\ell-1$ have been assigned, precompute the bounds obtained for each possible assignment for $x\ell$ and then process the assignment in order of increasing bound. Algorithm 16.3.2 describes this method. The danger in using this method is that at each node we are required to do sorting. If there are $m\ell$ choices for $x\ell$, this will require $O(m\ell \log(m\ell))$ additional steps at each node that occurs at level $\ell-1$ in the search tree. This computation may be prohibitive. (There no additional calls to the simplex algorithm.)

Page 461

Algorithm 16.3.2: BACKTRACK4(ℓ)

if $\ell > n$

then **if** $CurrentWt = b$
then **if** $CurrentZ < OptimalZ$
then **for** $j = 1$ **to** n **do** $OptimalX[j] \leftarrow X[j]$
$OptimalZ \leftarrow CurrentZ$
return

$N\ell \leftarrow 0$

if $\ell < n$ **then**

for $x \leftarrow 0$ **to** m_ℓ
do Use the simplex algorithm to solve the linear program:

$$\text{Minimize: } \widehat{Z} = \widehat{c}^T \widehat{X} \qquad \text{subject to: } \widehat{A}\widehat{X} = \widehat{b},\ \widehat{X} \geq 0$$

where $\widehat{A} = [A_{\ell+1}, \ldots, A_n]$, $\widehat{X} = [x_{\ell+1}, \ldots, x_n]$, $\widehat{c} = [c_{\ell+1}, \ldots, c_n]$, $\widehat{b} = b - CurrentWt - x * A_\ell$

if the linear program has optimal value $\widehat{Z}_{OPT}$ at $\widehat{X}_{OPT}$
then $B \leftarrow CurrentZ + \lfloor \widehat{Z}_{OPT} \rfloor$
if $\widehat{X}_{OPT}$ is integer valued
then **if** $B < OptimalZ$ **then**
for $i \leftarrow 1$ **to** $\ell - 1$ **do** $OptimalX[i] \leftarrow X[i]$
$OptimalX[\ell] \leftarrow x$
for $i \leftarrow \ell + 1$ **to** n **do** $OptimalX[i] \leftarrow \widehat{X}_{OPT}[i]$
$OptimalZ \leftarrow B$
else if $B < OptimalZ$ **then**
$N_\ell \leftarrow N_\ell + 1$
$C_\ell[N_\ell] \leftarrow (x, B)$
if the linear program is unbounded
then $N_\ell \leftarrow N_\ell + 1$
$C_\ell[N_\ell] \leftarrow (x, -\infty)$
if the linear program is infeasible **then return**

Sort the ordered pairs in $c\ell$ in order of increasing second coordinate

for $h \leftarrow 1$ **to** $N\ell$

$$\textbf{do}\ \begin{cases} \textbf{if}\ Bound_\ell[h] \geq OptimalZ\ \ \textbf{then return} \\ x_\ell \leftarrow \text{first coordinate of the ordered pair } C_\ell[h] \\ CurrentWt \leftarrow CurrentWt + A_\ell x_\ell \\ CurrentZ \leftarrow CurrentZ + c_\ell x_\ell \\ \text{BACKTRACK3}(\ell+1) \\ CurrentWt \leftarrow CurrentWt - A_\ell x_\ell \\ CurrentZ \leftarrow CurrentZ - c_\ell x_\ell \end{cases}$$

Page 462

The relaxed linear program for the Knapsack problem is easy to solve. A straightforward method which uses a *greedy strategy,* to solve the relaxed Knapsack problem is given in Algorithm 16.3.3. (See Exercises 16.3.4, 16.3.5, and 16.3.6.) It returns the optimal profit for the Knapsack problem, in which the integer constraint has been relaxed to allow non-integer (i.e., rational) solutions.

Algorithm 16.3.3: RELAXEDKNAPSACK (*p*1,*p*2,...,*Pn*, *w*1,...,*wn,M*)

permute the indices so that *p*1/*w*1≥*p*2/*w*2≥*Pn*/*wn*

i←1

P←0

W←0

for *j*←1 **to** *n*

do *xj*←0

while *W*<*M* **and** *i*<*n*

$$\textbf{do}\ \begin{cases} \textbf{if}\ W + w_i \leq M \\ \quad \textbf{then}\ \begin{cases} x_i \leftarrow 1 \\ W \leftarrow W + w_i \\ P \leftarrow P + p_i \\ i \leftarrow i + 1 \end{cases} \\ \quad \textbf{else}\ \begin{cases} x_i \leftarrow (M - W)/w_i \\ W \leftarrow M \\ P \leftarrow P + x_i\, p_i \\ i \leftarrow i + 1 \end{cases} \end{cases}$$

return *(P)*

To solve an instance of the Knapsack problem, it will be useful to presort the objects in non-decreasing order of the profit/weight ratio, before we begin the backtracking algorithm. Then, when we apply Algorithm 16.3.3, the first step will be unnecessary, and consequently RELAXEDKNAPSACK will run faster. Thus, we will assume that

$$\frac{p_1}{w_1} \geq \frac{p_2}{w_2} \geq \cdots \geq \frac{p_n}{w_n}.$$

The improved Knapsack algorithm is given as Algorithm 16.3.4.

Page 463

Algorithm 16.3.4: KNAPSACK3 *(ℓ)*

if *ℓ*=*n*

$$\textbf{then}\ \begin{cases} \textbf{if}\ CurrentProfit > OptimalProfit \\ \quad \textbf{then}\ \begin{cases} OptimalProfit \leftarrow CurrentProfit \\ OptimalX \leftarrow [x_1, \ldots, x_n] \end{cases} \end{cases}$$

if *ℓ*=*n*

then $C_\ell \leftarrow \emptyset$

$x \in C_\ell$

B←*CurrentProfit*

+RELAXEDKNAPSACK*(pℓ,pn,wℓ,...,wn,M—CurrentWt)*

for each $x \in C_\ell$

$$\textbf{do}\ \begin{cases} \textbf{if } B \le \mathit{OptimalProfit}\ \ \textbf{then return} \\ x_\ell \leftarrow x \\ \mathit{CurrentWt} = \mathit{CurrentWt} + w_\ell x_\ell \\ \mathit{CurrentProfit} = \mathit{CurrentProfit} + p_\ell x_\ell \\ \textsc{Knapsack3}(\ell+1) \\ \mathit{CurrentWt} = \mathit{CurrentWt} - w_\ell x_\ell \\ \mathit{CurrentProfit} = \mathit{CurrentProfit} - p_\ell x_\ell \end{cases}$$

Exercises

16.3.1 Solve graphically the following integer linear programs:

$$\text{(a)}\ \begin{cases} \text{Minimize:} & Z = 5x_1 - 3x_2 \\ \text{subject to:} & x_1 + x_2 \le 6 \\ & 2x_1 + 4x_2 \le 15 \\ & x_1, x_2 \ge 0 \\ & x_1, x_2 \ \text{integral.} \end{cases}$$

$$\text{(b)}\ \begin{cases} \text{Minimize:} & Z = x_1 - x_2 \\ \text{subject to:} & x_1 + x_2 \le 6 \\ & -8x_1 - 5x_2 \le -40 \\ & -5x_1 + 6x_2 \le 30 \\ & 6x_1 + 7x_2 \le 88 \\ & 9x_1 - 5x_2 \le 18 \\ & x_1, x_2 \ge 0 \\ & x_1, x_2 \ \text{integral.} \end{cases}$$

Use Algorithm 16.3.4 to solve the following instances of the Knapsack problem.

(a)	*Profits*	122	2	144	133	52	172	169	50	11	87	127	31	10	132	59
	Weights	63	1	71	73	24	79	82	23	6	43	66	17	5	65	29
	Capacity	323														
(b)	*Profits*	143	440	120	146	266	574	386	512	106	418	376	124	48	535	55
	Weights	72	202	56	73	144	277	182	240	54	192	183	67	23	244	29
	Capacity	1019														
(c)	*Profits*	818	460	267	75	621	280	555	214	721	427	78	754	704	44	371
	Weights	380	213	138	35	321	138	280	118	361	223	37	389	387	23	191
	Capacity	1617														

16.3.2 Algorithm 16.3.4 does not take advantage of the fact that given a partial solution *X′,* if the optimal solution to the corresponding relaxed knapsack problem is integer-valued it gives that best solution *X* that extends *X′*. Hence there is no need to pursue further extensions, and the search can be pruned. This type of pruning was done for the general problem in Algorithm 16.3.1. Construct a new algorithm that takes advantage of this pruning for the Knapsack problem. Test your algorithm on the data in Exercise 1. How does it compare with Algorithm 16.3.4?

16.3.3 Program Algorithm 16.3.1 and use it to solve the following integer linear program.

$$\begin{cases} \text{Minimize:} & Z = x_1 - 3x_2 + x5 \\ \text{subject to:} & x_1 + 2x_2 + 3x_4 \le 6 \\ & 5x_1 + 4x_2 + 9x_5 \le 20 \\ & X \ge 0 \\ & X \ \text{integral.} \end{cases}$$

16.3.4 Prove that Algorithm 16.3.3 does indeed solve

Problem 16.2: Relaxed Knapsack

Instance: *profits p*1, *p*1, *p*2,..., *pn;*
*weights w*1, *w*1, *w*2,... *wn;* and
capacity M;

Find: the maximum value of

$$P = \sum_{i=1}^{n} p_i x_i$$

subject to

$$\sum_{i=1}^{n} w_i x_i \leq M$$

and *x*1, *x*2..., *xn* are rational

as was claimed.

16.3.5 Verify that the simplex algorithm when applied to Problem 16.2 gives exactly the same result as Algorithm 16.3.3.

16.3.6 Determine the running time complexity of Algorithm 16.3.3.

16.3.7 In Section 9.6 we studied the traveling salesman problem. Let *W(uiuj)* be the non-negative weight on the edge *uiuj* of the complete graph with vertices *V*=

{*u*1, *u*2,..., *un*}. The traveling salesman problem is to find a hamilton cycle *C* that has minimum weight

$$\sum_{uv \in E(C)} W(uv).$$

Let $x_{uv} \in \{0, 1\}$ variable that denotes an edge *uu* in the hamilton cycle if *xuu*=1 and an edge not in the cycle if *xuu*=0. Show that the optimal solution to the discrete linear program

$$\begin{aligned}
\textbf{Minimize:} \quad & Z = \sum_{uv} x_{uv} W(uv) \\
\textbf{subject to:} \quad & \sum_{u \in V \setminus \{v\}} x_{uv} W(uv) = 2, \qquad v \in V \\
& \sum_{uv \in [S, \bar{S}]} x_{uv} W(uv) \geq 1, \qquad \emptyset \neq S \subset V \\
& x_{uv} \in \{0, 1\}, \quad \text{for each edge } uv
\end{aligned}$$

solves the traveling salesman problem.

16.3.8 Prove that the the problem

Problem 16.3: ILP decision

Instance: an integer linear program.

Question: does the the given integer linear program have a feasible solution?

is NP-complete. (*Hint:* Transform from 3-Sat.)

16.3.9 Use Algorithm 16.3.1 to determine the maximum number of edge disjoint triangles in the complete graph *Kn,* for *n*=7, 8, 9,..., 13. *Hint:* Use the $\binom{n}{2}$ by $\binom{n}{3}$ matrix *A* whose rows are labeled by the edges, whose columns are labeled by the triangles and whose *[e, t]*-entry is 1 if e is an edge on triangle *t* and is 0 otherwise. When $n \equiv 1, 3 \pmod 6$, then the maximum number edge disjoint triangles is $n(n-1)/6$. The corresponding collection of edge disjoint triangles is called a *Steiner triple system*.

16.4 Unimodular matrices

In Sections 15.6.2 and 15.6.3 we studied primal-dual algorithms for the Shortest Path and Max-Flow problems, respectively. Surprisingly we found that their optimal solutions were always integral although we never required that the variables be constrained to be integers. The reason for this is that the node-edge incidence matrix of any digraph is *totally unimodular*.

DEFINITION 16.2: An *m* by *n* integer valued matrix is *totally unimodular* if the determinant of each square submatrix is equal to 0, 1, or −1.

THEOREM 16.2 Every basis feasible solution to the linear program

Minimize: Z=cTX

subject to: AX=b

X≥0,

where A is a totally unimodular m by n matrix, and b is integer-valued, is integer-valued.

PROOF If *X* is the basis feasible solution corresponding to the submatrix *B* composed of *m* linearly independent columns $A_{j_1}, A_{j_2}, \ldots, A_{j_m}$, then

$$X_B = B^{-1}b = \frac{\text{ADJ}(B)}{\det(B)}b,$$

where ADJ*(B)* is the adjoint of *B*. Hence *XB* has integer values, because the total unimodularity of *A* implies that the det*(B)*=±1. Finally

$$X[j] = \begin{cases} X_B[j_\ell], & \text{if } j_\ell = j \\ 0, & \text{otherwise,} \end{cases}$$

and so the entries of *X* are integers.

THEOREM 16.3 Every basis feasible solution to the linear program

Minimize: Z=cTX
subject to: AX≤b
X≥0,

where A is a totally unimodular m by n matrix, and b is integer-valued, is integer-valued.

PROOF Adding slack variables *Y* we obtain the following equivalent linear pro gram:

$$\begin{aligned} \textbf{Minimize:} \quad & Z = c^T X \\ \textbf{subject to:} \quad & [A, I_m] \begin{bmatrix} X \\ Y \end{bmatrix} = b \\ & X, Y \geq 0. \end{aligned}$$

Thus we need only show that if *A* is a totally unimodular, then *[A,Im]* is totally unimodular, where *Im* is the *m* by m identity matrix. Then the result follows

Page 467

from Theorem 16.2. Let *M* be a square nonsingular submatrix of *[A,Im];* then after a suitable permutation of the rows and columns we see that *M* has the form

$$\left[\begin{array}{c|c} B & 0 \\ \hline N & I_{m-k} \end{array}\right]$$

where *B* is a square *k* by *k* submatrix of *A* and *Iℓ* is the *ℓ* by *ℓ* identity matrix, for some *k* and *ℓ*. The determinant of *B* is ±1, because *A* is totally unimodular and permutations of the rows and columns of *M* only change the determinant of *M* by a factor of ±1. Thus

$$\det(M) = \pm\det(B)\det(Ik) = \pm 1.$$

THEOREM 16.4 The node-edge incidence matrix of a directed graph is totally unimodular.

PROOF Let *G=(V,E)* be a directed graph and let *A* be its node-edge incidence matrix. Then

$$A[v,e] = \begin{cases} +1, & \text{if } e \text{ leaves } v, \\ -1, & \text{if } e \text{ enters } v, \\ 0, & \text{otherwise.} \end{cases}$$

In particular, *A* has exactly two non-zero entries in each column, one is a −1 and the other is +1. Let *M* be any *k* by *k* submatrix of *A*. If *k*=1, then clearly det*(M)*=0, +1, or −1. So suppose *k*>1 and proceed by induction. If *M* contains a column of zeros, then det*(M)*=0. If *M* contains a column *j* with a single non-zero entry *a*=±1 say in row *i*, then det*(M)=a* det*(N)* where *N* is the *k*−1 by *k*−1 submatrix obtained by removing column *j* and row *i*. By induction we have det*(N)*=0, +1 or −1, and so det*(M)*=0, +1 or −1. Finally we have the case when each column has two non-zero entries in each column. One is a −1 and the other is a +1; hence each column sums to zero and therefore *M* is singular and hence has determinant zero.

Exercises

16.4.1 Show that the following statements are all equivalent:
(a) *A* is totally unimodular.
(b) The transpose of *A* is totally unimodular.
(c) *[A,Im]* is totally unimodular.

Page 468

(d) A matrix obtained by deleting a row or column of *A* is totally unimodular.
(e) A matrix obtained by multiplying a row or column of *A* by −1 is totally unimodular.
(f) A matrix obtained by duplicating a row or column of *A* is totally unimodular.

(g) A matrix obtained by pivoting on an entry of *A* is totally unimodular.
16.4.2 Show that the matrix

$$\begin{bmatrix} 1 & -1 & 0 & 0 & -1 \\ -1 & 1 & -1 & 0 & 0 \\ 0 & -1 & 1 & -1 & 0 \\ 0 & 0 & -1 & 1 & -1 \\ -1 & 0 & 0 & -1 & 1 \end{bmatrix}$$

is totally unimodular.
16.4.3 Let *G* be an undirected bipartite graph with bipartition *(X,Y)*. Show that the vertex-edge incidence matrix *M* of *G* is totally unimodular.

$$M[v,e] = \begin{cases} 1, & \text{if } v \text{ is incident to } e \\ 0, & \text{otherwise.} \end{cases}$$

16.5 Notes

An excellent treatment of backtracking algorithms is given in the book by KREHER and STINSON [81]. The treatment of the Knapsack problem and exercise 16.3.1 is taken from this book. Two other good books that discuss general integer linear programming are PAPIDIMITRIOU and STEIGLITZ [94] and NEMHAUSER and WOLSEY [92].

Page 469

Bibliography

[1] A.V.AHO, J.E.HOPCROFT, AND J.D.ULLMAN, *The Design and Analysis of Computer Algorithms,* Addison-Wesley Publishing Co., Reading, Massachusetts, 1974.
[2] M.AIGNER, *Graph Theory, a Development from the 4-Color Problem,* BCS Associates, Moscow, Idaho, 1987.
[3] F.ALLAIRE, Another proof of the four colour theorem, Proceedings of the Seventh Manitoba Conference on Numerical Mathematics and Computing (Univ. Manitoba, Winnipeg, Man., 1977), pp. 3–72, *Congressus Numeran tium,* XX, Utilitas Mathematica, Winnipeg, Man.,1978.
[4] K.APPEL AND W.HAKEN, Every planar map is four colorable. part I: discharging, *Illinois Journal of Mathematics* 21 (1977), pp. 429–490.
[5] K.APPEL AND W.HAKEN, Every planar map is four colorable. part II: reducibility, *Illinois Journal of Mathematics* 21 (1977), pp. 491–456.
[6] D.ARCHDEACON, A Kuratowski theorem for the projective plane, *Journal of Graph Theory* 5 (1981), pp. 243–246.
[7] E.ARJOMANDI, An efficient algorithm for colouring the edges of a graph with A+1 colours, *Discrete Mathematical Analysis and Combinatorial Computation,* University of New Brunswick, 1980, pp. 108–132.
[8] J.BANG-JENSEN AND G.GUTIN, *Digraphs,* Springer-Verlag, New York, 2002.
[9] N.BIGGS, *Algebraic Graph Theory,* second edition, Cambridge University Press, Cambridge, 1993.
[10] M.BEHZAD AND G.CHARTRAND, *Introduction to the Theory of Graphs,* Allyn & Bacon, Boston, 1971.
[11] L.W.BEINEKE AND R.J.WILSON, *Selected Topics in Graph Theory,* Academic Press, London, 1978.
[12] L.W.BEINEKE AND R.J.WILSON, *Selected Topics in Graph Theory 2,* Academic Press, London, 1983.

Page 470

[13] L.W.BEINEKE AND R.J.WILSON, *Selected Topics in Graph Theory 3,* Academic Press, London, 1988.
[14] C.BERGE, *Graphs and Hypergraphs,* North-Holland Publishing Co., Amsterdam, 1979.
[15] J.-C.BERMOND, Hamiltonian graphs, in [11], pp. 127–168.
[16] R.G.BLAND, New finite pivoting rules for the simplex method. *Mathematics of Operations Research* 2 (1977), 103–107.
[17] B.BOLLOBÁS, *Graph Theory, An Introductory Course,* Springer-Verlag, New York, 1979.
[18] B.BOLLOBÁS, *Modern Graph Theory,* Springer-Verlag, New York, 2002.
[19] J.A.BONDY AND U.S.R.MURTY, *Graph Theory with Applications,* American Elsevier Publishing Co., New York, 1976.
[20] D.BRELAZ, New methods to color the vertices of a graph, *Communications ACM* 22 (1970), pp. 251–256.
[21] M.CAPOBIANCO AND J.C.MOLLUZZO, *Examples and Counterexamples in Graph Theory,* Elsevier North-Holland, New York, 1978.
[22] D.CHERITON AND R.E.TARJAN, Finding minimum spanning trees, *SIAM Journal of Computing* 5 (1976), pp. 724–742.

[23] , G.CHARTRAND AND F.HARARY, Graphs with prescribed connectivities, in, *Theory of Graphs, Proceedings Tihany, 1966,* Ed. P.Erdös and G. Katona, Academic Press, 1968, pp. 61–63.
[24] G.CHARTRAND AND L.LESNIAK, *Graphs and Digraphs,* Wadsworth & Brooks/Cole, Monterey, California, 1986.
[25] G.CHARTRAND AND O.OELLERMANN, *Applied and Algorithmic Graph Theory,* McGraw-Hill, Inc., New York, 1993.
[26] N.CHRISTOFIDES, *Graphs Theory, An Algorithmic Approach,* Academic Press, London, 1975.
[27] V.CHVÁTAL, *Linear Programming,* W.H.Freeman and Co., 1983.
[28] S.A.COOK, The complexity of theorem proving procedures, *Proc. Third ACM Symposium on the Theory of Computing, ACM* (1971), pp. 151–158.
[29] H.S.M.COXETER, *Regular Polytopes,* Dover Publications, New York, 1973.
[30] D.CVETKOVIC, M.D.DOOB, AND H.SACHS, *Spectra of Graphs: Theory and Applications,* John Wiley & Sons, 1998.
[31] G.B.DANTZIG, Programming of independent activities, II, mathematical model. *Econometrics* 17 (1949), pp. 200–211.
[32] G.B.DANTZIG, Programming of independent activities, II, mathematical

model, in *Activative of Production and Allocations,* ed. T.C.Koopermans, John Wiley & Sons, New York, 1951, pp. 19–32.
[33] G.B.DANTZIG, *Linear Programming and Extensions,* Princeton University Press, 1963.
[34] N.DEO, *Graph Theory with Applications to Engineering and Computer Science,* Prentice-Hall, Englewood Cliffs, New Jersey, 1974.
[35] R.DIESTEL, *Graph Theory,* Graduate Texts in Mathematics 173, Springer-Verlag, New York, Berlin, Heidelberg, 1997.
[36] G.A.DIRAC, Some theorems on abstract graphs, *Proceedings London Mathematical Society* 2 (1952), pp. 69–81.
[37] J.R.EDMONDS, A combinatorial representation for polyhedral surfaces, *Notices American Mathematical Society* 7 (1960), pp. 646.
[38] J.R.EDMONDS, Paths, trees, and flowers, *Canadian Journal of Mathematics* 17 (1965), pp. 449–467.
[39] J.R.EDMONDS AND R.M.KARP, Theoretical improvements in algorithmic efficiency for network flow problems, *Journal of the Association of Computing Machinery* 19 (1972), pp. 248–264.
[40] S.EVEN, *Graph Algorithms,* Computer Science Press, Potomac, Maryland, 1979.
[41] J.R.FIEDLER, J.P.HUNEKE, R.B.RICHTER, AND N.ROBERTSON, Computing the orientable genus of projective graphs, *Journal of Graph Theory* 20 (1995), pp. 297–308.
[42] J-C.FOURNIER, Colorations des aretes d'un graphe, *Cahiers du CERO* 15 (1973), pp. 311–314.
[43] L.R.FORD, JR. AND D.R.FULKERSON, Maximal flow through a network, *Canadian Journal of Mathematics* 8 (1956), pp. 399–404.
[44] M.FRÉCHET AND KY FAN, *Initiation to Combinatorial Topology,* Prindle, Weber, & Schmidt, Inc., Boston, 1967.
[45] C.FREMUTH-PAEGER AND D.JUNGNICKEL, An introduction to balanced network flows, in *Codes and Designs,* Ohio State University, Math. Res. Inst. Publ. 10 (2000), pp. 125–144.
[46] D.R.FULKERSON, ED., *Studies in Graph Theory, Parts I and II,* Mathematical Association of America, Washington, D.C., 1975.
[47] D.R.FULKERSON, Flow networks and combinatorial operations research, in [46], pp. 139–171.
[48] A.GAGARIN, *Graph Embedding Algorithms,* Ph.D. thesis, University of Manitoba, 2003.
[49] A.GAGARIN AND W.KOCAY, Embedding graphs contianing *K*5 subdi-

visions, *Ars Combinatoria* 64 (2002), pp. 33–49.
[50] M.R.GAREY AND D.S.JOHNSON, *Computers and Intractability, A Guide to the Theory of NP-Completeness,* W.H.Freeman, San Francisco, California, 1979.
[51] A.GIBBONS, *Algorithmic Graph Theory,* Cambridge University Press, Cambridge, 1985.
[52] C.GODSIL AND G.ROYLE, *Algebraic Graph Theory,* Springer-Verlag, New York, 2001.
[53] R.GOULD, *Graph Theory,* Benjamin/Cummings Publishing, Menlo Park, California, 1988.
[54] J.L.GROSS AND T.W.TUCKER, *Topological Graph Theory,* John Wiley & Sons, New York, 1987.

[55] J.GROSS AND J.YELLEN, *Graph Theory and Its Applications,* CRC Press, Boca Raton, Florida, 1999.
[56] B.GRÜNBAUM, *Convex Polytopes,* Springer-Verlag, New York, 2003.
[57] G.HADLEY, *Linear programming,* Addison-Wesley Publishing Co., 1962.
[58] P.HALL, On representatives of subsets, *Journal london Mathematical Society* 10 (1935), pp. 26–30.
[59] F.HARARY, *Graph Theory,* Addison-Wesley Publishing Co., Reading, Massachusetts, 1972.
[60] L.HEFFTER, Über das Problem der Nachbargebiete, *Mathematisce Annalen* 38 (1891), pp. 477–508.
[61] D.HILBERT AND S.COHN-VOSSEN, *Geometry and the Imagination,* Chelsea Publishing Co., New York, 1983.
[62] A.J.HOFFMAN, Eigenvalues of graphs, in [46], pp. 225–245.
[63] A.J.HOFFMAN AND R.R.SINGLETON, On Moore graphs with diameters 2 and 3, *IBM Journal of Research and Development* 4 (1960), pp. 497–504.
[64] I.HOLYER, The NP-completeness of edge-coloring, *SIAM Journal of Computing* 10 (1981), pp. 718–720.
[65] J.E.HOPCROFT AND R.M.KARP, An $n5/2$ algorit hm for maximum matching in bipartite graphs, *SIAM Journal of Computing* 2 (1973), pp. 225–231.
[66] J.HOPCROFT AND R.E.TARJAN, Efficient planarity testing, *Journal of the Association of Computing Machinery* 21 (1974), pp. 449–568.
[67] J.HOPCROFT AND R.E.TARJAN, Algorithm 447: efficient algorithms for graph manipulation, *CACM* 16 (1973), pp. 372–378.
[68] J.HOPCROFT AND R.E.TARJAN, Dividing a graph into triconnected com-

ponents, *SIAM Journal of Computing* 2 (1973), pp. 135–158.
[69] D.S.JOHNSON, Worst case behavior of graph coloring algorithms, *Proceedings of the Fifth Southeastern Conference on Combinatorics, Graph Theory, and Computing, Congressus Numerantium* 10 (1974), pp. 513–527.
[70] R.M.KARP, Reducibility among combinatorial problems, in *Complexity of Computer Computations,* eds. R.E.Miller and J.W.Thatcher, Plenum Press, New York, 1972, pp. 85–103.
[71] L.G.KHACHIAN, A polynomial algorithm in linear programing, *Doklady Adademiia Nauk SSR* 224 (1979), 1093–1096. (English translation: *Soviet Mathematics Doklady* 20 (1979), 191–194.)
[72] V.KLEE AND G.J.MINTY, How good is the simplex algorithm?, in *Inequalities-III,* ed. O.Shisha, Acdemic Press, New York, pp. 159–175, 1972.
[73] W.KLOTZ, A constructive proof of Kuratowski's theorem, *Ars Combinatoria,* 28 (1989), pp. 51–54.
[74] D.E.KNUTH, *The Art of Computer Programming,* Addison-Wesley Publishing Co., Reading, Massachusetts, 1973.
[75] D.E.KNUTH, *Searching and Sorting,* Addison Wesley Publishing Co., Reading, Massachusetts, 1973.
[76] W.KOCAY, D.NEILSON AND R.SZYPOWSKI, Drawing graphs on the torus, *Ars Combinatoria* 59 (2001), 259–277.
[77] W.KOCAY, An extension of the multi-path algorithm for finding hamilton cycles, *Discrete Mathematics* 101, (1992), pp. 171–188.
[78] W.KOCAY AND PAK-CHING LI, An algorithm for finding a long path in a graph, *Utilitas Mathematica* 45 (1994) pp. 169–185.
[79] W.KOCAY AND C.PANTEL, An algorithm for constructing a planar layout of a graph with a regular polygon as outer face, *Utilitas Mathematica* 48 (1995), pp. 161–178.
[80] W.KOCAY AND D.STONE, Balanced network flows, *Bulletin of the Institute of Combinatorics and Its Applications* 1 (1993), pp. 17–32.
[81] D.L.KREHER AND D.R.STINSON, *Combinatorial Algorithms: Generation, Enumeration and Search,* CRC Press, Boca Raton, Florida, 2000.
[82] C.KURATOWSKI, Sur le problème des curbes gauche en topologie, *Fund. Math.* 15 (1930), pp. 271–283.
[83] E.L.LAWLER, J.K.LENSTRA, A.H.G.RINNOOY KAN, AND D.B. SHMOYS, EDS., *The Traveling Salesman Problem,* John Wiley & Sons, Essex, U.K., 1990.
[84] L.LOVÁSZ, Three short proofs in graph theory, *Journal of Combinatorial*

Theory (B) 19 (1975), pp. 111–113.
[85] L.LOVÁSZ AND M.D.PLUMMER, *Matching Theory,* Elsevier Science, 1986.
[86] D.MCCARTHY AND R.G.STANTON, EDS., *Selected Papers of W.T. Tutte,* Charles Babbage Research

Centre, St. Pierre, Manitoba, 1979.
[87] B.MOHAR, Projective planarity in linear time, *Journal of Algorithms* 15 (1993), pp. 482–502.
[88] B.MOHAR AND C.THOMASSEN, *Graphs on Surfaces,* Johns Hopkins University Press, Baltimore, 2001.
[89] J.W.MOON, *Topics on Tournaments,* Holt, Rinehart, and Winston, New York, 1968.
[90] W. MYRVOLD, personal communication, 2004.
[91] W. MYRVOLD AND J.ROTH, Simpler projective planar embedding, *Ars Combinatoria,* to appear.
[92] G.L.NEMHAUSER AND L.A.WOLSEY, *Integer and Combinatorial Optimization,* John Wiley & Sons, 1988.
[93] O.ORE, *The Four-Colour Problem,* Academic Press, New York, 1967.
[94] C.H. PAPADIMITRIOU AND K.STEIGLITZ, *Combinatorial Optimization,* Dover Publications Inc., Mineola, New York, 1998.
[95] H.PRÜFER, Neuer Beweis eines Satzes über Permutationen, *Arch. Math. Phys.* 27 (1918), pp. 742–744.
[96] P.W.PURDOM, JR. AND C.A.BROWN, *The Analysis of Algorithms,* Holt, Rinehart, Winston, New York, 1985.
[97] R.C.READ, ED., *Graph Theory and Computing,* Academic Press, New York, 1972.
[98] R.C.READ, The coding of various kinds of unlabeled trees, in [97].
[99] R.C. READ, Graph theory algorithms, in *Graph Theory and its Applications,* ed. Bernard Harris, Academic Press, New York, 1970, pp. 51–78.
[100] R.C.READ, A new method for drawing a planar graph given the cyclic order of the edges at each vertex, *Congressus Numerantium* 56 (1987), pp. 31–44.
[101] R.C.READ AND W.T.TUTTE, Chromatic polynomials, in [13], pp. 15–42.
[102] K.B.REID AND L.W.BEINEKE, Tournaments, in [11], pp. 83–102.
[103] N.ROBERTSON AND P.D.SEYMOUR, Graph minors- a survey, in *Surveys in Combinatorics 1985,* Proceedings of the Tenth British Combinatorial Conference, (I.Anderson, ed.), London Math. Society Lecture Notes 103, Cambridge, 1985, pp. 153–171.

[104] N.ROBERTSON, D.P.SANDERS, P.SEYMOUR, AND R.THOMAS, The four-colour theorem, *Journal of Combinatorial Theory (B)* 70 (1997), pp. 2–44.
[105] F.RUBIN, A search procedure for hamilton paths and circuits, *JACM* 21 (1974), pp. 576–580.
[106] T.L.SAATY AND P.C.KAINEN, *The Four-Color Problem, Assaults and Conquest,* Dover Publications, New York, 1977.
[107] A.J.SCHWENK AND R.J.WILSON, Eigenvalues of graphs, in [11], pp. 307–336.
[108] R.SEDGEWICK, *Algorithms in C++*, Addison-Wesley Publishing Co., Boston, 1998.
[109] J.STILLWELL, *Classical Topology and Combinatorial Group Theory,* Spinger-Verlag, New York, 1980.
[110] H.A.TAHA, *Operations Research: An introduction,* Prentice Hall, Englewood Cliffs, New Jersey, 2003.
[111] R.E.TARJAN, Depth-first search and linear graph algorithms, *SIAM Journal of Computing* 1 (1972) pp. 146–160.
[112] C.THOMASSEN, Kuratowski's Theorem, *Journal of Graph Theory* 5 (1981), pp. 225–241.
[113] C.THOMASSEN, The graph genus problem is NP-complete, *Journal of Algorithms* 10 (1989), pp. 568–576.
[114] C.THOMASSEN, Planarity and duality of finite and infinite graphs, *Journal of Combinatorial Theory (B)* 29 (1980), pp. 244–271.
[115] R.TRUDEAU, *Introduction to Graph Theory,* Dover Publications, Mineola, 1994.
[116] W.T.TUTTE, How to draw a graph, in [86], pp. 360–388.
[117] W.T.TUTTE, A short proof of the factor theorem for finite graphs, in [86], pp. 169–175.
[118] W.T.TUTTE, The factorization of linear graphs, in [86], pp. 89–97.
[119] W.T.TUTTE, Chromials, in [46], pp. 361–377.
[120] W.T.TUTTE, *Connectivity in Graphs,* University of Toronto Press, 1966.
[121] R.C.WALKER, *Introduction to Mathematical Programming,* Prentice Hall, Englewood Cliffs, New Jersey, 1999.
[122] M.A.WEISS, *Data Structures and Algorithm Analysis,* Benjamin Cummings Publishing Co., Redwood City, California, 1992.
[123] D.B.WEST, *Graph Theory,* Prentice Hall, Upper Saddle River, New Jersey, 1996.

[124] A.T.WHITE, The Proof of the Heawood Conjecture, in [11].
[125] H.WHITNEY, 2-Isomorphic Graphs, *Amerirican Journal of Mathematics* 55 (1933), pp. 245–254.
[126] S.G.WILLIAMSON, Embedding graphs in the plane algorithmic aspects, *Annals of Discrete Mathematics* 6

(1980), pp. 349–384.
[127] R.J.WILSON, *Four Colours Suffice,* Penguin Books, London, 2002.
[128] D.R.WOODALL AND R.J.WILSON, The Appel-Haken proof of the four-colour theorem, in [11], pp. 83–102.
[129] G.M.ZIEGLER, *Lectures on Polytopes,* Springer-Verlag, New York, 1998.

Index

Symbols

*K*5-component, 381
f-factor, 152
k-connected, 122
k-cube, 49
k-face, 374
k-factor, 148
k-factorization, 149
k-regular, 9
m-chromatic, 245
m-coloring, 245
m-critical, 254
m-edge-coloring, 258
(i, j)-subgraph, 259
2-Sat, 240–243, 378–380
2-cell, 329
2-cell embedding, 329
3-Colorability, 269–271
3-Sat, 207, 209, 240, 269, 271–274, 465

A

abstract dual, 299
activity graph, 224
acyclic, 224
adjacency list, 11
adjacent, 2
admissible columns, 434
All Paths, 30
alternating path, 140
antipodal automorphism, 369
antisymmetry, 241
augmented *K*5-component, 381
augmenting path, 140, 167
augmenting the flow, 167
auxiliary network, 175

B

backward edges, 165
balanced network, 177
barycentric coordinatization, 308
basis feasible solution, 396
basis solution, 395
Berge's theorem, 140
BFS, 31
bicentral trees, 91
binary heap, 44
binary plane, 97
binary tree, 92
bipartite, 47
blocks, 122
blossoms, 158

bond, 68
bond space, 68
Bondy-Chvátal theorem, 198
bottleneck, 172
branch and bound, 455
branches, 89
breadth-first numbering, 71
breadth-first search, 30
bridge, 373
Brooks' theorem, 247
bundle, 316

C

capacity, 161, 164
Catalan numbers, 97
Cayley's theorem, 108
center, 91
central trees, 91
certificate, 205
Christofides' algorithm, 219
Chromatic Index, 271, 272, 274
chromatic index, 258
chromatic number, 245,383
chromatic polynomial, 256
chromial, 274
Class I graphs, 261
Class II graph, 261
clause, 206
Clique, 271
clique, 249
clockwise, 283
closed surface, 331, 332
co-cycles, 68
co-tree, 67
cofactor, 111
color class, 249
color rotation, 261
color-isomorphic, 249
coloring, 245
combinatorial embedding, 286
combinatorial planar dual, 288
complement, 3
complementary slackness, 429
complementary strong component, 241
complementing, 8
complete bipartite, 47
complete graph, 3
complexity, 19
component representative, 24
condensation, 230, 242
conflict graph, 316
connected component, 23
connected graph, 23
connects, 23
continuous, 328
contract, 109, 278
contractible, 339

converse, 231
convex, 291
convex set of feasible solutions, 425
corners, 281
covering, 154
critical, 254
critical path, 224
critical path method, 224
crosscap, 333, 337
crosscap number, 339, 340
crossover, 192, 195
cube, 187
current bundle, 316
curvilinear polygons, 332
cycle, 23
cycle space, 68
cylinder, 329
cylindrical embedding, 345

D
degree, 5,283
degree matrix, 111
degree saturation method, 250
degree sequence, 9
Dehn and Heegard theorem, 334
depth-first search, 128
Desargues, 371
Descartes' formula, 385
DFS, 128, 228, 229
diagonal cycle, 350
diagonal path, 350
diagonally opposite, 350
diameter, 57
differences, 115
digon, 332
digraph, 161
directed graph, 161
disc embedding, 345

discrete linear program, 449
distance, 30
distinct embeddings, 329
double cover, 368
dual linear program, 417
dual-restricted primal, 435

E
edge set, 1
edge-chromatic number, 258
edge-connectivity, 120
edge-cut, 67
edges, 23, 124
elementary branches, 97
ellipsoid method, 416
embedding, 328
empty graph, 3
end-block, 135
endpoints, 2

equivalent embeddings, 297, 346
Erdös-Gallai conditions, 17
Erdös-Gallai theorem, 16
essential, 340
Euler characteristic, 337
Euler tour, 58
Euler trail, 58
Euler's formula, 283
Euler-Poincaré formula, 340
Eulerian, 58
excluded minor, 342

F
Fáry's theorem, 306
fabric graph, 236
faces, 282, 328
facewidth, 366
facial cycle, 283, 328
facial walk, 283
feasible solutions, 389
first chord below, 315
five-color theorem, 304
flow, 161
forest, 76
forward edges, 165
four-color theorem, 302
fronds, 131
fundamental cycle, 65
fundamental edge-cut, 68

G
generating function, 97
genus, 339, 340
girth, 52
Graph Embeddability, 341
Graph Embeddings, 341
Graph Genus, 341, 385
graph minor theorem, 342
Graphic, 10
graphic, 10
greedy strategy, 462
growth rate, 20

H
Hall's theorem, 141
HamCycle, 188, 205, 207, 210, 214, 215
hamilton closure, 197
hamilton cycle, 187
hamilton path, 187
hamiltonian, 187
hamiltonian cycle, 187
handle, 336
Havel-Hakimi theorem, 15
heap, 37
heap property, 37
Heawood graph, 348
Heawood map coloring theorem, 384
Heawood's theorem, 382
height of a branch, 97
hexagon, 344

hexagon edges, 374
homeomorphic, 278, 329
homeomorphic embeddings, 346
homeomorphism, 329
Hungarian algorithm, 144

Page 480

I

ILP decision, 465
implication digraph, 241
in-degree, 223
in-edges, 223
in-flow, 161
independent set, 249
induced subgraph, 5
initial Stableau, 398
inner vertices, 146, 281
integer linear program, 449
internally disjoint, 123
inverter, 271
inverting component, 271
isomorphic, 5, 346
isomorphic embeddings, 297
isomorphism, 4

J

Jordan curve, 276
Jordan curve theorem, 276

K

König's theorem, 154
Kempe, 302
Kempe chain, 302
Kirchhoff matrix, 111
Knapsack, 449–451, 462, 464, 468
Kuratowski graphs, 312
Kuratowski subgraphs, 342
Kuratowski's theorem, 309

L

labeled trees, 106
Laplacian matrix, 111
leading chord, 315
leaf, 90
leftist binary tree, 79
line-graph, 50
linear, 387
Linear programming, 387
logarithmic growth, 20
longest path, 28
loops, 2
low-point, 131
lower bound, 183

M

Möbius band, 329
Möbius ladder, 365
Möbius lattice, 365

matching, 139
Matrix-tree theorem, 111
Max-Flow, 163, 465
max-flow, 163
max-flow-min-cut theorem, 169
maximal matching, 140
maximum, 216
maximum clique, 250
maximum degree, 9
maximum genus, 385
maximum independent set, 250
maximum matching, 140
medial digraph, 300, 347, 369
merge-find, 24
min-cut, 164
minimal, 341
minimal edge-cut, 68
minimal forbidden minor, 342
minimum, 166
minimum amount of time, 224
minimum covering, 154
minimum degree, 9
minimum spanning tree problem, 72
minor, 278
minor-order obstruction, 342
Moore graphs, 52
multi-path method, 189
multigraph, 2, 343

N
near 1-factorization, 158
near perfect matching, 158
neighbor set, 141
network, 161
node, 3

Page 481
node-edge incidence matrix, 430
non-contractible, 340
non-deterministic polynomial, 205
non-orientable, 297, 331
non-orientable embedding, 346
non-planar, 277
normal form, 334
NP, 205
NP-complete, 188, 206
NP-completeness, 205
null-homotopic, 339
number, 110

O
obstructions, 341
one-sided, 331
only, 108
open disc, 329
open surface, 329
optimization problems, 387
order, 2
Ore's lemma, 262

orientable, 297, 331
orientable embedding, 346
orientable genus, 339
orientation, 283
oriented graph, 223
out-degree, 223
out-edges, 223
out-flow, 161
outer face, 282
outer vertices, 146

P
P, 240
parity lemma, 259
path, 23
path compression, 25
perfect matching, 140
Petersen graph, 357
Phase 1 linear program, 408
Phase 1 tableau, 408
pivoting, 396
planar, 275
planar dual, 286
Platonic maps, 350
Platonic solids, 290
point, 3
polygons, 290
polyhedron, 290, 331
polynomial, 205
polynomial transformation, 205
positive integral weight, 34
Prüfer sequence, 106
PrevPt, 145
primal linear program, 417
priority queue, 78
Programming problems, 387
projective map, 360
proper coloring, 245
proper edge coloring, 258

Q
quadragon edges, 374
queue, 30

R
reachable, 168
reducibility, 305
reducible configuration, 305
regular, 9
regular polygon, 290
regular polyhedron, 290
relaxation, 453
Relaxed Knapsack, 464
representativity, 366
requirement-space, 425
residual capacity, 166
restricted primal, 435
Ringel-Youngs theorem, 384
Robbins' theorem, 239
root, 71

root vertex, 89
rooted trees, 89
rotation system, 284, 344

S

Sat, 206–208, 243
Satisfiability, 221
satisfiability of boolean expressions, 206
satisfiable, 206
saturated, 140
saturation degree, 251
Schläfli symbols, 291
segments, 200
self-complementary, 6
self-converse, 231
self-dual, 291
separable, 122
separating cycle, 298
separating set, 120
sequential algorithm, 245
shadow set, 141
Shannon's theorem, 262
shortest, 172
Shortest Path, 30, 465
Shortest Path (directed graph), 430
shortest-path problem, 430
signature, 358
simple, 2, 223
simple graph, 1
skeleton, 290
slack, 392
slack variable, 392
smallest index rule, 415
source, 161
spanning tree bound, 216
spanning trees, 64
splitting, 279
stable set, 250
standard form, 392
standard linear program, 392
star, 100
Steiner triple system, 465
Steinitz's theorem, 291
stereographic projection, 295
strict, 223
strong, 229
strong component, 229
strongly connected, 229
subdivided, 277
subdivision, 277
subdivision graph, 51
subgraph, 5
Subgraph Problem, 152
support of the flow, 176
surface, 328
surplus, 392
surplus variable, 392

symmetric difference, 140
system of distinct representatives, 143

T
target, 161
ternary heap, 44
theta-graph, 342
throughput, 178
topological embedding, 286
topological minor, 278
topological obstruction, 341
topological ordering, 225
topologically equivalent, 278
torus map, 344
totally unimodular, 465
tournament, 238
transitive tournament, 238
tree, 63
tree algorithm, 218
tree graph, 69
triangle inequality, 216
triangle traveling salesman problem, 216
triangulation, 292
truncated tetrahedron, 187
TSP, 214
TSP Decision, 214, 215
Tutte’s theorem, 155