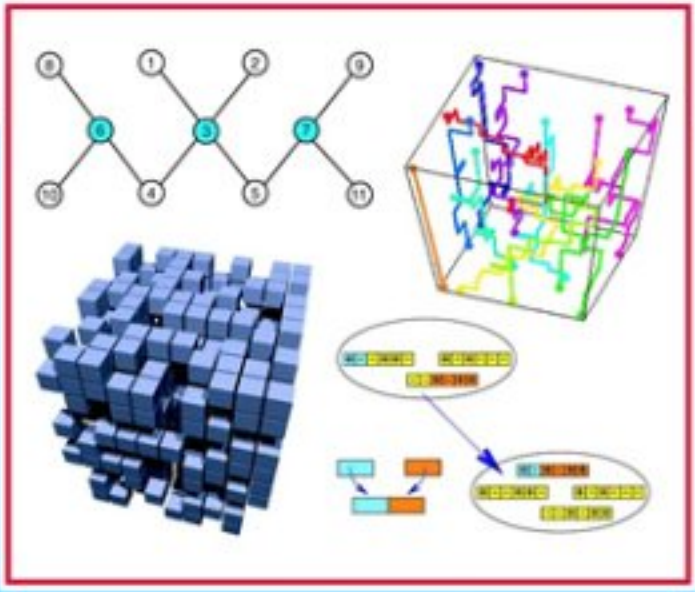
Alexander K. Hartmann, Heiko Rieger

Optimization Algorithms in Physics



WILEY-VCH

Alexander K. Hartmann, Heiko Rieger

Optimization Algorithms in Physics

Alexander K. Hartmann, Heiko Rieger

Optimization Algorithms in Physics

WILEY-VCH

Authors:
Alexander K. Hartmann, Institute of Theoretical Physics, University of Goettingen, Germany
e-mail: hartmann@theorie.physik.uni-goettingen.de
Heiko Rieger, Institute of Theoretical Physics, Saarland University, Germany
e-mail: rieger@lusi.uni-sb.de

1st edition

Library of Congress Card No: applied for

British Library Cataloguing-in-Publication Data: A catalogue record for this book is available from the British Library.

Die Deutsche Bibliothek – CIP Cataloguing-in-Publication-Data
A catalogue record for this publication is available from Die Deutsche Bibliothek

ISBN 3-527-40307-8

Printed on non-acid paper.

Printing: Strauss Offsetdruck GmbH, Mörlenbach
Bookbinding: Wilhelm Osswald & Co., Neustadt (Weinstraße)

Printed in the Federal Republic of Germany.

WILEY-VCH Verlag Berlin GmbH
Bühringstrasse 10
D-13086 Berlin

Preface

This book is an interdisciplinary book: it tries to teach physicists the basic knowledge of combinatorial and stochastic optimization and describes to the computer scientists physical problems and theoretical models in which their optimization algorithms are needed. It is a unique book since it describes theoretical models and practical situation in physics in which optimization problems occur, and it explains from a physicists point of view the sophisticated and highly efficient algorithmic techniques that otherwise can only be found specialized computer science textbooks or even just in research journals.

Traditionally, there has always been a strong scientific interaction between physicists and mathematicians in developing physics theories. However, even though numerical computations are now commonplace in physics, no comparable interaction between physicists and computer scientists has been developed. Over the last three decades the design and the analysis of algorithms for decision and optimization problems have evolved rapidly. Most of the active transfer of the results was to economics and engineering and many algorithmic developments were motivated by applications in these areas.

The few interactions between physicists and computer scientists were often successful and provided new insights in both fields. For example, in one direction, the algorithmic community has profited from the introduction of general purpose optimization tools like the simulated annealing technique that originated in the physics community. In the opposite direction, algorithms in linear, nonlinear, and discrete optimization sometimes have the potential to be useful tools in physics, in particular in the study of strongly disordered, amorphous and glassy materials. These systems have in common a highly non-trivial minimal energy configuration, whose characteristic features dominate the physics at low temperatures. For a theoretical understanding the knowledge of the so called "ground states" of model Hamiltonians, or optimal solutions of appropriate cost functions, is mandatory. To this end an efficient algorithm, applicable to reasonably sized instances, is a necessary condition.

The list of interesting physical problems in this context is long, it ranges from disordered magnets, structural glasses and superconductors through polymers, membranes, and proteins to neural networks. The predominant method used by physicists to study these questions numerically are Monte Carlo simulations and/or simulated annealing. These methods are doomed to fail in the most interesting situations. But, as pointed out above, many useful results in optimization algorithms research never reach the physics community, and interesting computational problems in physics do not come to the attention of algorithm designers. We therefore think that there is a definite need

to intensify the interaction between the computer science and physics communities. We hope that this book will help to extend the bridge between these two groups. Since one end is on the physics side, we will try to guide a number of physicists to a journey to the other side such that they can profit from the enormous wealth in algorithmic techniques they will find there and that could help them in solving their computational problems.

In preparing this book we benefited greatly from many collaborations and discussions with many of our colleagues. We would like to thank Timo Aspelmeier, Wolfgang Bartel, Ian Campbell, Martin Feix, Martin Garcia, Ilia Grigorenko, Martin Weigt, and Annette Zippelius for critical reading of the manuscript, many helpful discussions and other manifold types of support. Furthermore, we have profited very much from fruitful collaborations and/or interesting discussions with Mikko Alava, Jürgen Bendisch, Ulrich Blasum, Eytan Domany, Phil Duxbury, Dieter Heermann, Guy Hed, Heinz Horner, Jermoe Houdayer, Michael Jünger, Naoki Kawashima, Jens Kisker, Reimer Kühn, Andreas Linke, Olivier Martin, Alan Middleton, Cristian Moukarzel, Jae-Dong Noh, Uli Nowak, Matthias Otto, Raja Paul, Frank Pfeiffer, Gerhard Reinelt, Federico Ricci-Tersenghi, Giovanni Rinaldi, Roland Schorr, Eira Seppälaä, Klaus Usadel, and Peter Young. We are particularly indebted to Michael Baer, Vera Dederichs and Cornelia Reinemuth from Wiley-VCH for the excellent cooperation and Judith Egan-Shuttler for the copy editing.

Work on this book was carried out at the University of the Saarland, University of Göttingen, Forschungszentrum Jülich and the University of California at Santa Cruz and we would like to acknowledge financial support from the Deutsche Forschungsgeimeinschaft (DFG) and the European Science Foundation (ESF).

Santa Cruz and Saarbrücken May 2001 Alexander K. Hartmann and Heiko Rieger

Contents

1 Introduction to Optimization **1**
Bibliography 6

2 Complexity Theory **9**
2.1 Algorithms 9
2.2 Time Complexity 16
2.3 NP Completeness 19
2.4 Programming Techniques 26
Bibliography 34

3 Graphs **37**
3.1 Graphs 37
3.2 Trees and Lists 39
3.3 Networks 42
3.4 Graph Representations 44
3.5 Basic Graph Algorithms 48
3.6 NP-Complete Graph Problems 50
Bibliography 52

4 Simple Graph Algorithms **53**
4.1 The Connectivity-percolation Problem 53
4.1.1 Hoshen-Kopelman Algorithm 54
4.1.2 Other Algorithms for Connectivity Percolation 57
4.1.3 General Search Algorithms 57
4.2 Shortest-path Algorithms 61
4.2.1 The Directed Polymer in a Random Medium 62
4.2.2 Dijkstra's Algorithm 63
4.2.3 Label-correcting Algorithm 67
4.3 Minimum Spanning Tree 68
Bibliography 71

5 Introduction to Statistical Physics **73**
5.1 Basics of Statistical Physics 73
5.2 Phase Transitions 78
5.3 Percolation and Finite-size Scaling 81

5.4 Magnetic Transition 84
5.5 Disordered Systems 87
Bibliography 90

6 Maximum-flow Methods **91**
6.1 Random-field Systems and Diluted Antiferromagnets 92
6.2 Transformation to a Graph 96
6.3 Simple Maximum Flow Algorithms 102
6.4 Dinic's Method and the Wave Algorithm 107
6.5 Calculating all Ground States 115
6.6 Results for the RFIM and the DAFF 121
Bibliography 125

7 Minimum-cost Flows **129**
7.1 Motivation 129
7.2 The Solution of the N-Line Problem 133
7.3 Convex Mincost-flow Problems in Physics 136
7.4 General Minimum-cost-flow Algorithms 139
7.5 Miscellaneous Results for Different Models 147
Bibliography 155

8 Genetic Algorithms **159**
8.1 The Basic Scheme 159
8.2 Finding the Minimum of a Function 164
8.3 Ground States of One-dimensional Quantum Systems 172
8.4 Orbital Parameters of Interacting Galaxies 178
Bibliography 183

9 Approximation Methods for Spin Glasses **185**
9.1 Spin Glasses 185
9.1.1 Experimental Results 187
9.1.2 Theoretical Approaches 190
9.2 Genetic Cluster-exact Approximation 192
9.3 Energy and Ground-state Statistics 203
9.4 Ballistic Search 208
9.5 Results 219
Bibliography 222

10 Matchings **227**
10.1 Matching and Spin Glasses 227
10.2 Definition of the General Matching Problem 228
10.3 Augmenting Paths 230
10.4 Matching Algorithms 231
10.4.1 Maximum-cardinality Matching on Bipartite Graphs 231
10.4.2 Minimum-weight Perfect Bipartite Matching 235

10.4.3 Cardinality Matching on General Graphs 241
10.4.4 Minimum-weight Perfect Matching for General Graphs 242
10.5 Ground-state Calculations in 2d 250
Bibliography 252

11 Monte Carlo Methods **255**
11.1 Stochastic Optimization: Simple Concepts 255
11.2 Simulated Annealing 257
11.3 Parallel Tempering 260
11.4 Prune-enriched Rosenbluth Method (PERM) 262
11.5 Protein Folding 266
Bibliography 270

12 Branch-and-bound Methods **273**
12.1 Vertex Covers 274
12.2 Numerical Methods 277
12.3 Results 287
Bibliography 291

13 Practical Issues **293**
13.1 Software Engineering 293
13.2 Object-oriented Software Development 300
13.3 Programming Style 306
13.4 Programming Tools 310
13.4.1 Using Macros 310
13.4.2 *Make* Files 314
13.4.3 Scripts 317
13.5 Libraries 319
13.5.1 Numerical Recipes 319
13.5.2 LEDA 321
13.5.3 Creating your own Libraries 323
13.6 Random Numbers 324
13.6.1 Generating Random Numbers 324
13.6.2 Inversion Method 327
13.6.3 Rejection Method 328
13.6.4 The Gaussian Distribution 330
13.7 Tools for Testing 331
13.7.1 *gdb* 332
13.7.2 *ddd* 334
13.7.3 *checkergcc* 334
13.8 Evaluating Data 338
13.8.1 Data Plotting 338
13.8.2 Curve Fitting 340
13.8.3 Finite-size Scaling 343
13.9 Information Retrieval and Publishing 347

13.9.1 Searching for Literature 347
13.9.2 Preparing Publications 349
Bibliography 355

Index **359**

1 Introduction to Optimization

Optimization problems [1, 2, 3] are very common in everyday life. For example, when driving to work one usually tries to take the *shortest* route. Sometimes additional *constraints* have to be fulfilled, e.g. a bakery should be located along the path, in case you did not have time for breakfast, or you are trying to avoid busy roads when riding by bicycle.
In physics many applications of optimization methods are well know, e.g.

- Even in beginners courses on theoretical physics, in classical mechanics, optimization problems occur: e.g. the Euler-Lagrange differential equation is obtained from an optimization process.
- Many physical systems are governed by minimization principles. For example, in thermodynamics, a system coupled to a heat bath always takes the state with minimal free energy.
- When calculating the quantum mechanical behavior of atoms or small molecules, quite often a variational approach is applied: the energy of a test state vector is minimized with respect to some parameters.
- Frequently, optimization is used as a tool: when a function with various parameters is fitted onto experimental data points, then one searches for the parameters which lead to the best fit.

Apart from these classical applications, during the last decade many problems in physics have turned out to be in fact optimization problems, or can be transformed into optimization problems, for recent reviews, see Ref. [4, 5, 6]. Examples are:

- Determination of the self affine properties of polymers in random media
- Study of interfaces and elastic manifolds in disordered environments
- Investigation of the low-temperature behavior of disordered magnets
- Evaluation of the morphology of flux lines in high temperature superconductors
- Solution of the protein folding problem
- Calculation of the ground states of electronic systems
- Analysis of X-ray data

- Optimization of lasers/optical fibers
- Reconstruction of geological structures from seismic measurements

On the other hand, some classical combinatorial optimization problems occurring in theoretical computer science have attracted the attention of physicists. The reason is, that these problems exhibit *phase transitions* and that methods from statistical physics can be applied to solve these problems.
An optimization problem can be described mathematically in the following way: let $\underline{\sigma} = (\sigma_1, \ldots, \sigma_n)$ be a vector with n elements which can take values from a domain X^n: $\sigma_i \in X$. The domain X can be either discrete, for instance $X = \{0, 1\}$ or $X = Z$ the set of all integers (in which case it is an integer optimization problem) or X can be continuous, for instance $X = R$ the real numbers. Moreover, let $\mathcal{H}$ be a real valued function, the cost function or objective, or in physics usually the Hamiltonian or the energy of the system. The *minimization problem* is then:

$$\textit{Find } \underline{\sigma} \in X^n, \textit{ which minimizes } \mathcal{H}!$$

A maximization problem is defined in an analogous way. We will consider only minimization problems, since maximizing a function H is equivalent to minimizing $-H$. Here, only minimization problems are considered where the set X is *countable*. Then the problem is called *combinatorial* or *discrete*. Optimization methods for real valued variables are treated mainly in mathematical literature and in books on numerical methods, see e.g. Ref. [7].
Constraints, which must hold for the solution, may be expressed by additional equations or inequalities. An arbitrary value of $\underline{\sigma}$, which fulfills all constraints, is called *feasible*. Usually constraints can be expressed more conveniently without giving equations or inequalities. This is shown in the first example.

Example: Traveling Salesman Problem (TSP)

The TSP has attracted the interest of physicist several times. For an introduction, see Ref. [8]. The model is briefly presented here. Consider n cities distributed randomly in a plane. Without loss of generality the plane is considered to be the unit square. The minimization task is to find the shortest round-tour through all cities which visits each city only once. The tour stops at the city where it started. The problem is described by

$$X = \{1, 2, \ldots, n\} \tag{1.1}$$

$$H(\underline{\sigma}) = \sum_{i=1}^{n} d(\sigma_i, \sigma_{i+1}) \tag{1.2}$$

where $d(\sigma_\alpha, \sigma_\beta)$ is the distance between cities σ_α and σ_β and $\sigma_{n+1} \equiv \sigma_1$. The constraint that every city is visited only once can be realized by constraining the vector $\underline{\sigma}$ to be a permutation of the sequence $[1, 2, \ldots, n]$.

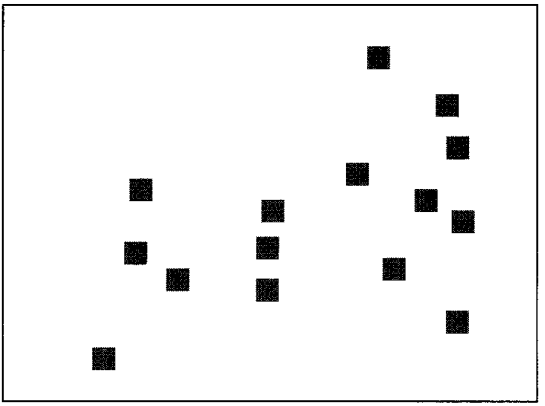


Figure 1.1: 15 cities in a plane.

As an example 15 cities in a plane are given in Fig. 1.1. You can try to find the shortest tour. The solution is presented in Chap. 2. For the general TSP the cities are not placed in a plane, but an arbitrary distance matrix d is given. □

The optimum order of the cities for a TSP depends on their exact positions, i.e. on the random values of the distance matrix d. It is a feature of all problems we will encounter here that they are characterized by various random parameters. Each random realization of the parameters is called an *instance* of the problem. In general, if we have a collection of optimization problems of the same (general) type, we will call each single problem an instance of the general problem.
Because the values of the random parameters are fixed for each instance of the TSP, one speaks of *frozen* or *quenched* disorder. To obtain information about the general structure of a problem one has to average measurable quantities, like the length of the shortest tour for the TSP, over the disorder. Later we will see that usually one has to consider many different instances to get reliable results.
While the TSP originates from everyday life, in the following example from physics a simple model describing complex magnetic materials is presented.

Example: Ising Spin Glasses

An Ising spin σ_i is a small magnetic moment which can take, due to anisotropies of its environment, only two orientations called *up* and *down*, e.g $\sigma_i = \pm 1$. For the simplest model of a magnetic material one assumes that spins are placed on the sites of a simple lattice and that a spin interacts only with its nearest neighbors. In a *ferromagnet* it is energetically favorable for a spin to be in the same orientation as its neighbors, i.e. parallel spins

give a negative contribution to the total energy. On the other hand the thermal noise causes different spins to point randomly up or down. For low temperatures T the thermal noise is small, thus the system is *ordered*, i.e. ferromagnetic. For temperatures higher than a critical temperature T_c, no long range order exists. One says that a *phase transition* occurs at T_c, see Chap. 5. For a longer introduction to phase transitions, we refer the reader e.g. to Ref. [9].

A spin configuration which occurs at $T = 0$ is called a *ground state*. It is just the absolute minimum of the energy $H(\underline{\sigma})$ of the system since no thermal excitations are possible at $T = 0$. They are of great interest because they serve as the basis for understanding the low temperature behavior of physical systems. From what was said above, it is clear that in the ground state of a ferromagnet all spins have the same orientation (if quantum mechanical effects are neglected).

A more complicated class of materials are *spin glasses* which exhibit not only ferromagnetic but also *antiferromagnetic* interactions, see Chap. 9. Pairs of neighbors of spins connected by an antiferromagnetic interaction like to be in different orientations. In a spin glass, ferromagnetic and antiferromagnetic interactions are distributed randomly within the lattice. Consequently, it is not obvious what ground state configurations look like, i.e. finding the minimum energy is a non-trivial minimization problem. Formally the problem reads as follows:

$$X = \{-1, 1\} \tag{1.3}$$

$$H(\underline{\sigma}) = -\sum_{<i,j>} J_{ij}\sigma_i\sigma_j \tag{1.4}$$

where J_{ij} denotes the interaction between the spins on site i and site j and the sum $\langle i, j\rangle$ runs over all pairs of nearest neighbors. The values of the interactions are chosen according to some probability distribution. Each random realization is given by the collection of all interactions $\{J_{ij}\}$. Even the simplest distribution, where $J_{ij} = 1$ or $J_{ij} = -1$ with the same probability, induces a highly non-trivial behavior of the system. Please note that the interaction parameters are frozen variables, while the spins σ_i are free variables which are to be adjusted in such a way that the energy becomes minimized.

Fig. 1.2 shows a small two-dimensional spin glass and one of its ground states. For this type of system usually many different ground states for each realization of the disorder are feasible. One says, the ground state is *degenerate*. Algorithms for calculating degenerate spin-glass ground states are explained in Chap. 9.

□

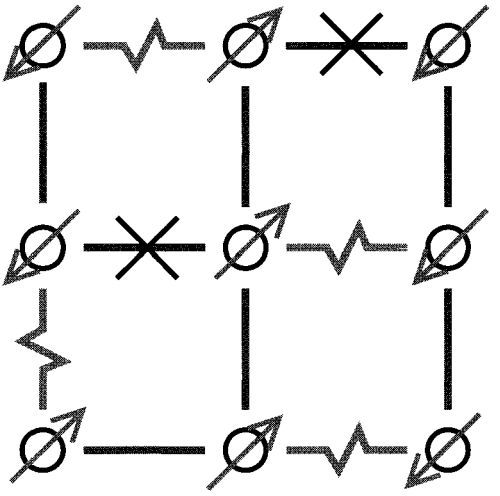


Figure 1.2: Two-dimensional spin glass. Solid lines represent ferromagnetic interactions while jagged lines represent antiferromagnetic interactions. The small arrows represent the spins, adjusted to a ground-state configuration. For all except two interactions (marked with a cross) the spins are oriented relative to each other in an energetically favorable way. It is not possible to find a state with lower energy (try it!).

These two examples, which are in general of equivalent computational complexity as we will learn when reading this book, are just intended as motivation, as to why dealing with optimization problems is an interesting and fruitful task. The aim of this book is to give an introduction to *methods* how to *solve* these problems, i.e. how to find the optimum. Interestingly, there is no single way to achieve this. For some problems it is very easy while for others it is rather hard, this refers to the time you or a computer will need at least to solve the problem, it does not say anything about the elaborateness of the algorithms which are applied. Additionally, within the class of hard or within the class of easy problems, there is no universal method. Usually, even for each kind of problem there are many different ways to obtain an optimum. On the other hand, there are several universal algorithms, but they find only approximations of the true optima. In this book algorithms for easy and algorithms for hard problems are presented. Some of the specialized methods give exact optima, while other algorithms, which are described here, are approximation techniques.

Once a problem becomes large, i.e. when the number of variables n is large, it is impossible to find a minimum by hand. Then computers are used to obtain a solution. Only the rapid development in the field of computer science during the last two decades has pushed forward the application of optimization methods to many problems from science and real life.

In this book, efficient discrete computer algorithms and recent applications to problems from physics are presented. The book is organized as follows. In the second chapter, the foundations of complexity theory are explained. They are needed as a basis for understanding the rest of the book. In the next chapter an introduction to graph theory is given. Many physical questions can be mapped onto graph theoretical optimization problems. Then, some simple algorithms from graph theory are explained, sample applications are from percolation theory are presented. In the following chapter, the basic notions from statistical physics, including phase transitions and finite-size scaling are given. You can skip this chapter if you are familiar with the subject. The main part of the book starts with the sixth chapter. Many algorithms are presented along with sample problems from physics, which can be solved using the algorithms. First, techniques to calculate the maximum flow in networks are exhibited. They can be used to calculate the ground states of certain disordered magnetic materials. Next, minimum-cost-flow methods are introduced and applied to solid-on-solid models and vortex glasses. In the eighth chapter genetic algorithms are presented. They are general purpose optimization methods and have been applied to various problems. Here it is shown how ground states of electronic systems can be calculated and how the parameters of interacting galaxies can be determined. Another type of general purpose algorithm, the Monte Carlo method, is introduced along with several variants in the following chapter. In the succeeding chapter, the emphasis is on algorithms for spin glasses, which is a model that has been at the center of interest of statistical physicists over the last two decades. In the twelfth chapter, a phase transition in a classical combinatorial optimization problem, the vertex-cover problem, is studied. The final chapter is dedicated to the practical aspects of scientific computing. An introduction to software engineering is given, along with many hints on how to organize the program development in an efficient way, several tools for programming, debugging and data analysis, and finally, it is shown how to find information using modern techniques such as data bases and the Internet, and how you can prepare your results such that they can be published in scientific journals.

Bibliography

[1] C.H. Papadimitriou and K. Steiglitz, *Combinatorial Optimization*, (Dover Publications, Mineola (NY) 1998)

[2] W.J. Cook, W.H. Cunningham, W.R. Pulleyblank, and A. Schrijver, *Combinatorial Optimization*, (J. Wiley & Sons, New York 1998)

[3] B. Korte and J. Vygen, *Combinatorial Optimization*, (Spinger, Berlin and Heidelberg 2000)

[4] J.C. Anglès d'Auriac, M. Preissmann, and A. Seb, *J. Math. and Comp. Model.* **26**, 1 (1997)

[5] H. Rieger,in : J. Kertesz and I. Kondor (ed.), *Advances in Computer Simulation*, Lecture Notes in Physics **501**, (Springer, Heidelberg 1998)

[6] M.J. Alava, P.M. Duxbury, C. Moukarzel, and H. Rieger, *Exact Combinatorial Algorithms: Ground States of Disordered Systems*, in: C. Domb and J.L. Lebowitz (ed.), *Phase Transitions and Critical Phenomena* **18**, (Academic press, New York 2001)

[7] W.H. Press, S.A. Teukolsky, W.T. Vetterling, and B.P. Flannery, *Numerical Recipes in C*, (Cambridge University Press, Cambridge 1995)

[8] S. Kirkpatrick , C. D. Gelatt, Jr., and M. P. Vecchi, *Science* **220**, 671 (1983)

[9] J.M. Yeomans, *Statistical Mechanics of Phase Transitions*, (Clarendon Press, Oxford 1992)

2 Complexity Theory

Programming languages are used to instruct a computer what to do. Here no specific language is chosen, since this is only a technical detail. We are more interested in the general way a method works, i.e. in the *algorithm*. In the following chapters we introduce a notation for algorithms, give some examples and explain the most important results about algorithms provided by theoretical computer sciences.

2.1 Algorithms

Here we do not want to try to give a precise definition of what an *algorithm* is. We assume that an algorithm is a sequence of statements which is computer readable and has an unambiguous meaning. Each algorithm may have input and output (see Fig. 2.1) which are well defined objects such as sequences of numbers or letters. Neither user-computer interaction nor high-level output such as graphics or sound are covered. Please note that the communication between the main processing units and keyboards or graphic-/sound- devices takes place via sequences of numbers as well. Thus, our notion of an algorithm is universal.

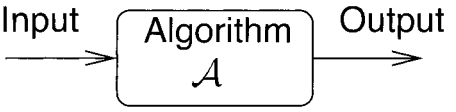


Figure 2.1: Graphical representation of an algorithm.

Algorithms for several specific purposes will be presented later. We will concentrate on the main ideas of each method and not on implementational details. Thus, the algorithms will not be presented using a specific programming language. Instead, we will use a *notation* for algorithms called *pidgin Algol*, which resembles modern high-level languages like Algol, Pascal or C. But unlike any conventional programming language, variables of an arbitrary type are allowed, e.g. they can represent numbers, strings, lists, sets or graphs. It is not necessary to declare variables and there is no strict syntax.
For the definition of pidgin Algol, we assume that the reader is familiar with at least one high-level language and that the meaning of the terms *variable*, *expression*, *condition*

and *label* is clear. A pidgin Algol program is *one* of the following *statements*, please note that by using the *compound* statement, programs can be of arbitrary length:

1. *Assignment*
   variable := *expression*

   A value is assigned to a variable. Examples: $a := 5 * b + c$, $A := \{a_1, \ldots, a_n\}$

   Also more complex and informal structures are allowed, like
   let z be the first element of the queue Q

2. *Condition*
   if *condition* **then** *statement 1*
   else *statement 2*

   The **else** clause is optional. If the condition is true, statement 1 is executed, else statement 2, if it exists.

   Example: **if** *money*>100 **then** *restaurant* := 1 **else** *restaurant* := 0

3. *Cases*
   case: *condition 1*
   statement1_A;
   statement1_B;
   ...
   case: *condition 2*
   statement2_A;
   statement2_B;
   ...
   case: *condition 3*
   statement3_A;
   statement3_B;
   ...
   ...
   end cases

   This statement is useful, if many different case can occur, thus making a sequence of **if** statements too complex. If condition 1 is true, then the first block of statements is executed (here no **begin** ... **end** is necessary). If condition 1 is true, then the second block of statements is executed, etc.

4. *While loop*
   while *condition* **do** *statement*

   The statement is performed as long as the condition is true.

   Example: **while** *counter* < 200 **do** *counter* := *counter*+1

5. *For loop*
   for *list* **do** *statement*

   The statement is executed for all parameters in the list. Examples:

for $i := 1, 2, \ldots, n$ **do** *sum* := *sum*+*i*

for all elements q of queue Q **do** *waits*[*q*] := *waits*[*q*]+1

6. *Goto statement*
   a) *label: statement*
   b) **goto** *label*

   When the execution of an algorithm reaches a goto statement the execution is continued at the statement which carries the corresponding label.

7. *Compound statement*
   begin
   statement 1;
   statement 2;
   ...
   statement n;
   end

   The compound statement is used to convert a sequence of statements into one statement. It is useful e.g. if a for-loop should be executed for a body of several statements.

   Example:
   for $i := 1, 2, \ldots, n$ **do**
   begin
   $a := a + i$;
   $b := b + i * i$;
   $c := c + i * i * i$;
   end

   For briefness, sometimes a compound statement is written as a list of statements in one line, without the **begin** and **end** keywords.

8. *Procedures*
   procedure *procedure-name* (*list of parameters*)
   begin
   statements
   return *expression*
   end

   The **return** statement is optional. A procedure is used to define a new name for one statement or, using a compound statement, for a collection of statements. A procedure can be invoked by writing: *procedure-name* (*arguments*)

   Example:

```
procedure minimum (x, y)
begin
   if x>y then return y
   else return x
end
```

9. *Comments*

   comment *text*

   Comments are used to explain parts of an algorithm, i.e. to aid in its understanding. Sometimes a comment is given a the right end of a line without the **comment** keyword.

10. *Miscellaneous statements*: practically any text which is self-explanatory is allowed. Examples:

    Calculate determinant D of matrix M
    Calculate average waiting time for queue Q

As a first example we present a simple heuristic for the TSP. This method constructs a tour which is quite short, but it does not guarantee to find the optimum. The basic idea is to start at a randomly chosen city. Then iteratively the city which has the shortest distance from the present city, i.e. its *nearest neighbor*, is chosen from the set of cities which have not been visited yet. The array v will be used to indicate which cities already belong to the tour. Please remember that $d(i,j)$ denotes the distance between cities i and j and n is the number of cities.

algorithm TSP-nearest-neighbor($n, \{d(i,j)\}$)
begin
  for $i := 1, 2, \ldots, n$ **do**
    $v[i] := 0$;
  σ_1 := one arbitrarily chosen city;
  $v[\sigma_1] := 1$;
  for $i := 2, 3, \ldots, n$ **do**
  begin
    min := ∞;
    for all unvisited cities j **do**
      if $d(\sigma_{i-1}, j) <$ min **then**
        min := $d(\sigma_{i-1}, j)$; $\sigma_i := j$;
    $v[\sigma_i] := 1$;
  end
end

Please note that the length of the tour constructed in this way depends on the city where the tour starts and that this city is randomly chosen. This algorithm and many other heuristics for the TSP can be found on the well presented TSP web-pages

of Stephan Mertens [1]. On these pages different TSP algorithms are implemented using Java-applets. It is possible to run the algorithms step by step and watch the construction of the tour on the screen. In Fig. 2.2 the results for one sample of 15 cities are shown. The top part presents a Java-applet which contains results for the heuristic while in the bottom part the shortest tour is given.

The basis tools and results for the analysis of algorithms were developed in the field of theoretical computer science. For a beginner many of the results may seem unimportant for practical programming purposes. But in fact, for the development of effective algorithms their knowledge is essential. Here we give the reader just a short glimpse into the field by presenting the most fundamental definitions and results. As an example we will prove in the second part of this section that there are functions of natural numbers which cannot be programmed on a computer. For this purpose an important technique called *diagonalization* is used. Now we will prepare the proof in several steps.

Pidgin Algol is sufficient to present and analyze algorithms. But for a theoretical treatment exact methods and tools are necessary. For this purpose a precise definition of algorithms is needed. Formal models of computation such as the *Turing machine* are used, where everything is stored on a tape via a read/write head. Also very common is the *Random access machine* which is a simple model of real computers consisting of an RAM memory and a central processing unit. It can be shown that all reasonable formal machine models are equivalent. This means that for any program on one model an equivalent program can be written for a different model. For more information the reader is referred e.g. to [2].

The observation that all reasonable machine models are equivalent has led to the *Church's thesis*: "For any algorithm a program can be written on all reasonable machine models." Since the term algorithm cannot be defined exactly it is impossible to prove Church's thesis. Nobody has come up with an algorithm that cannot be transfered to a computer. Hence, it seems reasonable that this thesis is true.

In the following we will concentrate on programs which have just one natural number as input and one natural number as output. This is not a restriction because every input/output sequence can be regarded as one long list of bits, i.e. one (possibly large) natural number.

Every program of this kind realizes a *partial* function $f : \mathcal{N} \to \mathcal{N}$ from natural numbers to natural numbers. The term partial means that they may be not defined for every input value, the corresponding program for f will run forever for some input x. If f is not defined for the argument x we write $f(x) = \text{div}$.

As a next step towards the proof that there are functions which are not computable, we present a method of how to *enumerate* all computable functions. This enumeration works by assigning a code-number to each program. For a precise definition of the assignment, one must utilize a precise machine model like the Turing machine or the random access machine. Here a simple treatment is sufficient for our purpose. Thus, we can assume that the programs are written in a high level language like C, but restricted to the case where only one input and one output number (with arbitrary high precision) is allowed. The code-number is assigned to a program in the following way: when the program is stored in memory it is just a long sequence of bits. This is

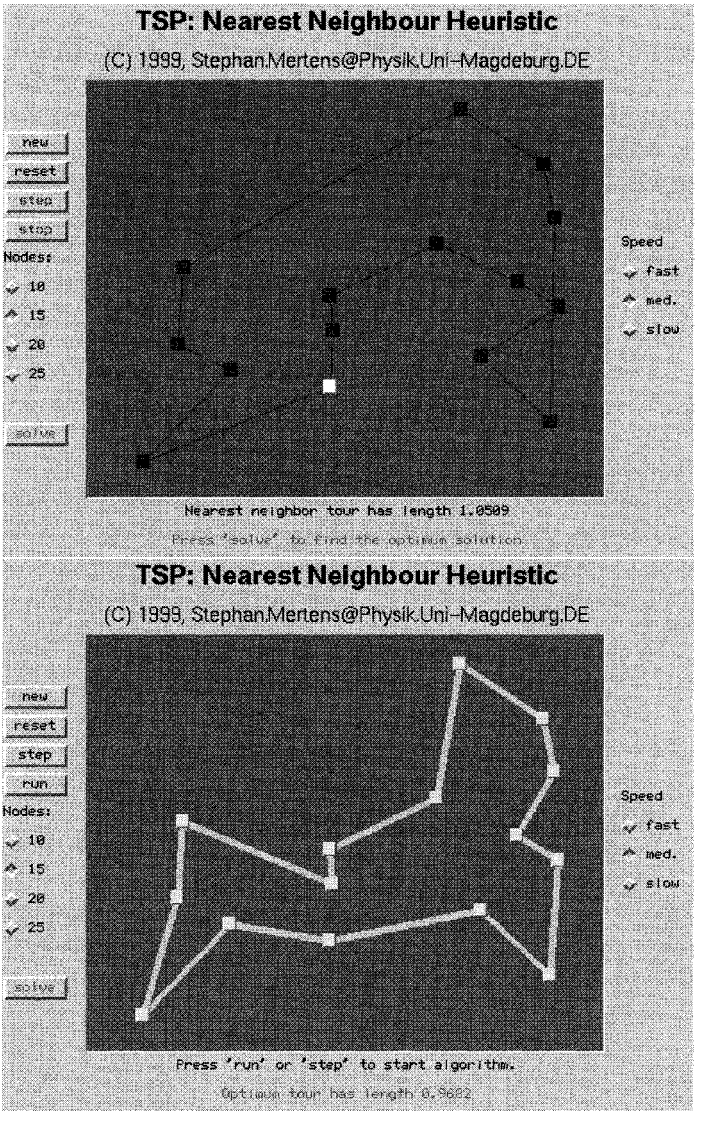


Figure 2.2: A sample TSP containing 15 cities. The results for the nearest-neighbor heuristic (top) and the exact optimum tour (bottom) are shown. The starting city for the heuristic is marked by a white square. The nearest neighbor of that city is located above it.

quite a long natural number, representing the program in a unique way. Now, let f_n be the function which is defined through the text with number n, if the text is a valid

program. If text n is not a valid program or if the program has more than one input or output number, then we define $f_n(x) = \text{div}$ for all $x \in \mathcal{N}$. In total, this procedure assigns a function to *each* number.
All functions which can be programmed on a computer are called *computable*. Please note that for every computable function f there are multiple ways to write a program, thus there are many numbers n with $f_n = f$. Now we want to show:
There are functions $f : \mathcal{N} \to \mathcal{N}$ which are not computable
Proof: We define the following function

$$f^*(x) = \begin{cases} 1 & \text{for} \quad f_x(x) = \text{div} \\ \text{div} & \text{else} \end{cases} \tag{2.1}$$

Evidently, this is a well defined partial function on the natural numbers. The point is that it is different from all computable functions f_n, i.e. f^* itself is not computable:

$$\forall n : f_n(n) \neq f^*(n), \quad \Rightarrow \quad \forall n : f_n \neq f \tag{2.2}$$

QED
The technique applied in the proof above is called *diagonalization*. The reason is that if one tabulates the infinite matrix consisting of the values $f_n(i)$ then the function f^* is different from each f_n on the diagonal $f_n(n)$. The principle used for the construction of f^* is visualized in Fig. 2.3. The technique of diagonalization is very useful for many proofs occurring not only in the area of theoretical computer science but also in many fields of mathematics. The method was probably introduced by Georg Cantor at the beginning of the century to show that there are more than a countable number of real numbers.

f_n	x=1	2	3	4	5	▪ ▪
f_1	$f_1(1)$	$f_1(2)$	$f_1(3)$	$f_1(4)$	$f_1(5)$	▪ ▪
f_2	$f_2(1)$	$f_2(2)$	$f_2(3)$	$f_2(4)$	$f_2(5)$	▪ ▪
f_3	$f_3(1)$	$f_3(2)$	$f_3(3)$	$f_3(4)$	$f_3(5)$	▪ ▪
f_4	$f_4(1)$	$f_4(2)$	$f_4(3)$	$f_4(4)$	$f_4(5)$	▪ ▪
f_5	$f_5(1)$	$f_5(2)$	$f_5(3)$	$f_5(4)$	$f_5(5)$	▪ ▪
▪	▪	▪	▪	▪	▪	▪ ▪
▪	▪	▪	▪	▪	▪	

Figure 2.3: Principle of diagonalization: define a function which differs from all computable functions on the diagonal.

It should be pointed out that the existence of f^* is not a contradiction to Church's thesis since f^* is *not* defined through an algorithm. If someone tries to implement the function f^* from above, he/she must have an algorithm or test available which tells whether a given computer program will halt at some time or whether it will run

forever ($f_n(x) = \text{div}$). The question whether a given program stops or not is called the *halting problem*. With a similar and related diagonalization argument as we have seen above, it can be shown that there is indeed no solution to this problem. It means that no universal algorithm exists which *decides* for *all* programs whether the program will halt with a given input or run forever. On the other hand, if a test for the halting problem was available it would be easy to implement the function f^* on a computer, i.e. f^* would be computable. Thus, the undecidability of the halting problem follows from the fact that f^* is also not computable.
In principle, it is always possible to prove for a *given* program whether it will halt on a given input or not by checking the code and deep thinking. The insolvability of the halting problem just means that there is no systematic way, i.e. no algorithm to construct a proof for *any* given program. Here, as for most proofs in mathematics, the person who carries it out must rely on his/her creativity. But with increasing length of the program the proof usually becomes extremely difficult. It is not surprising that for realistic programs like word processors or databases no such proofs are available. The same is true for the *correctness problem*: There is no systematic way to prove that a given program works according a given specification. On the other hand, this is fortunate, since otherwise many computer scientists and programmers would be unemployed.
The halting problem is a so called *recognition problem*: for the question "will Program P_n halt on input x" only the answers "yes" or "no" are possible. In general, we will call an instance (here a program) *yes-instance* if the answer is "yes" for it, otherwise *no-instance*. As we have seen, the halting-problem is not *decidable*, because it is not possible to prove the answer "no" systematically. But if the answer is "yes", i.e. if the program stops, this can always be proven: just take the program P_n, supply input x, run it and wait till it stops. This is the reason why the halting problem at least is *provable*.

2.2 Time Complexity

After we have taken a glimpse at the theory of computability, we will proceed with defining the *time complexity* of an algorithm which describes its speed. We will define under what circumstances we call an algorithm *effective*. The speed of a program can only be determined if it halts on every input. For all optimization problems we will encounter, there are algorithms which stop on all inputs. Consequently, we will restrict ourself to this case.
Almost always the time for executing a program depends on the input. Here, we are interested in the dependence on the *size* $|x|$ of the input x. For example, finding a tour visiting 10 cities usually takes less time than finding a tour which passes through one million cities. The most straightforward way of defining the size of the input is counting the number of bits (without leading zeros). But for most problems a "natural" size is obvious, e.g. the number of cities for the TSP or the number of spins for the spin-glass problem. Sometimes there is more than one characteristic size, e.g. a general TSP is given through several distances between pairs of cities. Then the

execution time of a program may also depend on the number of distances, i.e. the number of nonzero entries of the matrix $d(i,j)$.
Usually one is not interested in the actual running time $t(x)$ of an algorithm for a specific implementation on a given computer. Obviously, this depends on the input, the skills of the programmer, the quality of the compiler and the amount of money which is available for buying the computer. Instead, one would like to have some kind of measure that characterizes the algorithm itself.
As a first step, one takes the longest running time over all inputs of a given length. This is called the *worst case running time* or worst case *time complexity* $T(n)$:

$$T(n) = \max_{x:|x|=n} t(x) \tag{2.3}$$

Here, the time is measured in some arbitrary units. Which unit is used is not relevant: on a computer B which has exactly twice the speed of computer A a program will consume only half the time. We want to characterize the algorithm itself. Therefore, a good measure must be independent of such constant factors like the speed of a computer. To get rid of these constant factors one tries to determine the *asymptotic* behavior of a program by giving *upper* bounds:
Definition: O/Θ notation Let T, g be functions from natural numbers to real numbers.

- We write $T(n) \in O(g(n))$ if there exists a positive constant c with $T(n) \leq cg(n)$ for all n. We say: $T(n)$ is of order at most $g(n)$.
- $T(n) \in \Theta(g(n))$ if there exist two positive constants c_1, c_2 with $c_1 g(n) \leq T(n) \leq c_2 g(n) \quad \forall n$. We say: $T(n)$ is of order $g(n)$.

Example: O/Θ-notation

For $T(n) = pn^3 + qn^2 + rn$, the cubic term is the fastest growing part. Let $c \equiv p + q + r$. Then $T(n) \leq cn^3 \quad \forall n$, which means $T(n) \in O(n^3)$. Since e.g. $n^4, 2^n$ are growing faster than n^3, we have $T(n) \in O(n^4)$ and $T(n) \in O(2^n)$. Let $c' \equiv \min\{p,q,r\}$. Then $c'n^3 \leq T(n) \leq cn^3$. Hence, $T(n) \in \Theta(n^3)$. This smallest upper bond characterizes $T(n)$ most precisely. □

We are interested in obtaining the time complexity of a given algorithm without actually implementing and running it. The aim is to analyze the algorithm given in pidgin Algol. For this purpose we have to know how long basic operations like assignments, increments and multiplications take. Here we assume that a machine is available where all basic operations take *one* time-step. This restricts our arithmetic operations to a fixed number of bits, i.e. numbers of arbitrary length cannot be computed. If we encounter a problem where numbers of arbitrary precision can occur, we must include the time needed for the arithmetic operations explicitly in the analysis.
As an example, the time complexity of the TSP heuristic will now be investigated, which was presented in the last section. At the beginning of the algorithm a loop

is performed which sets all variables $v[i]$ to zero. Each assignment costs a constant amount of time. The loop is performed n times. Thus, the loop is performed in $\Theta(n)$. The random choice of a city and the assignment of $v[\sigma_1]$ take constant time. The rest of the algorithm is a loop over all cities which is performed $n-1$ times. The loop consists of assignments, comparisons and another loop which runs over all unvisited cities. Therefore, the inner loop is performed $n-1$ times for $i=2$, $n-2$ times for $i=3$, etc. Consequently, the **if**-statement inside that loop is performed $\sum_{i=2}^{n}(n+1-i) = \sum_{i=1}^{(n-1)}(n-i) = n(n-1)/2$ times. Asymptotically this pair of nested loops is the most time-consuming part of the algorithm. Thus, in total the algorithm has a time complexity of $\Theta(n^2)$.
Can the TSP heuristic be considered as being fast? Tab. 2.1 shows the growth of several functions as a function of input size n.

Table 2.1: Growth of functions as a function of input size n.

$T(n)$	$T(10)$	$T(100)$	$T(1000)$
n	10	100	1000
$n \log n$	10	200	3000
n^2	10^2	10^4	10^6
n^3	10^3	10^6	10^9
$n^{\log n}$	10	10^4	10^9
2^n	1024	1.3×10^{30}	1.1×10^{301}
$n!$	3.6×10^6	10^{158}	4×10^{2567}

For algorithms which have a time complexity that grows faster than all polynomials, even moderate increases of the system size make the problem impossible to treat. Therefore, we will call an algorithm *effective* if its time complexity is bounded by a polynomial: $T(n) \in O(n^k)$. In practice, values of the exponent up to $k=3$ are considered as suitable. For very large exponents and small system sizes algorithms with exponentially growing time complexity may be more useful. Compare for example an algorithm with $T_1(n) = n^{80}$ and another with $T_2(n) = 2^n$. The running-time of the first algorithm is astronomical even for $n=3$, while the second one is able to treat at least small input sizes.
The application of the O/Θ-notation neglects constants or lower order terms for the time complexity. Again, in practice an algorithm with running time $T_3(n) = n^3$ may be faster for small input sizes than another with $T_4(n) = 100n^2$. But these kinds of examples are very rare and rather artificial.
In general, finding an algorithm which has a lower time complexity is always more effective than waiting for a computer to arrive that is ten times faster. Consider two algorithms with time complexities $T_5(n) = n \log n$ and $T_6(n) = n^3$. Let n_5 respectively n_6 be the maximum problem sizes which can be treated within one day of computer time. If a computer is available which is ten times faster, the first algorithm can treat approximately inputs of size $n_5 \times 10$ (if n_5 is large) within one day while for the second the maximum input size grows only as $n_6 \times 10^{1/3}$.

To summarize, algorithms which run in polynomial time are considered as being fast. But there are many problems, especially optimization problems, where no polynomial-time algorithm is known. Then one must apply algorithms where the running time increases exponentially or even greater with the system size. This holds e.g. for the TSP if the exact minimum tour is to be computed. The study of such problems led to the concept of *NP-completeness*, which is introduced in the next section.

2.3 NP Completeness

For the moment, we will only consider recognition problems. Please remember that these are problems for which only the answers "yes" or "no" are possible. We have already have introduced the halting and the correctness-problem which are not decidable. The following example of a recognition problem, called *SAT* is of more practical interest. In the field of theoretical computer science it is one of the most basic recognition problems. For SAT it was first shown that many other recognition problems can mapped onto it. This will be explained in detail later on. Recently SAT has attracted much attention within the physics community [3].

Example: k-satisfiability (k-SAT)

A *boolean variable* x_i may only take the values 0 (false) and 1 (true). Here we consider three *boolean operations*:

- NOT $\overline{}$ (*negation*): the *clause* $\overline{x_i}$ ("NOT x_i") is true ($\overline{x_i} = 1$), if and only if (iff) x_i is false: $x_i = 0$
- AND $\wedge$ (*conjunction*): the *clause* $x_i \wedge x_j$ ("x_i AND x_j") is true, iff both variables are true: $x_i = 1$ AND $x_j = 1$.
- OR $\vee$ (*disjunction*): the *clause* $x_i \vee x_j$ ("x_i OR x_j") is true, iff at least one of the variables is true: $x_i = 1$ OR $x_j = 1$

A variable x_i or its negation $\overline{x_i}$ is called a *literal*. Using parentheses different clauses may be combined to produce complex *boolean formulas*, e.g. $(x_1 \vee \overline{x}_2) \wedge (x_3 \vee x_2)$.

A formula is called *satisfiable*, if there is at least one assignment for the values of its variables such that the whole formula is true. For example, the formula $(\overline{x_1} \vee x_2) \wedge \overline{x_2}$ is satisfiable, because for $x_1 = 0, x_2 = 0$ it is true. The formula $(\overline{x_1} \vee x_2) \wedge \overline{x_2} \wedge x_1$ is not satisfiable, because $\overline{x_2} \wedge x_1$ implies $x_1 = 1, x_2 = 0$, but then $(\overline{x_1} \vee x_2)$ is false.

For the k-SAT problem, formulae of the following type are considered, called k-CNF (*conjunctive normal form*) formulae: each formula F consists of m clauses C_j combined by the AND operator:

$$F = C_1 \wedge C_2 \wedge \ldots \wedge C_m. \tag{2.4}$$

Each clause C_p has k literals l_{pq} containing distinct variables combined by the OR-operator:

$$C_p = l_{p1} \vee l_{p2} \vee \ldots \vee l_{pk}. \tag{2.5}$$

For example $(x_1 \vee x_2) \wedge (\overline{x_4} \vee \overline{x_2}) \wedge (\overline{x_1} \vee x_3)$ is a 2-CNF formula, while $(x_1 \vee \overline{x_2} \vee x_3) \wedge (\overline{x_2} \vee x_3 \vee \overline{x_4})$ is a 3-CNF formula.

The class k-SAT consists of all problems of the form "is F satisfiable?" where F is a k-CNF formula. The question whether an arbitrary formula is satisfiable is an instance of such a defined SAT problem. Please note that every boolean formula can be rewritten as a conjunction of clauses each containing only disjunctions and negations. This form is called CNF. □

We have already seen that some recognition problems are undecidable. For these problems it has been proven that no algorithm can be provided to solve it. The k-SAT problem is decidable, i.e. there is a so called *decision-algorithm* which gives for each instance of a k-SAT problem the answer "yes" or "no". The simplest algorithm uses the fact that each formula contains a finite number n of variables. Therefore, there are exactly 2^n different assignments for the values of all variables. To check whether a formula is satisfiable, one can scan through all possible assignments and check whether the formula evaluates to true or to false. If for one of them the formula is true, then it is satisfiable, otherwise not. In the Tab. 2.2 all possible assignments for the variables of $(x_2 \vee x_3) \wedge (x_1 \vee \overline{x_3})$ and the results for both clauses and the whole formula is displayed. A table of this kind is called a *truth table*.

Table 2.2: Truth table.

x_1	x_2	x_3	$x_2 \vee x_3$	$x_1 \vee \overline{x_3}$	$(x_2 \vee x_3) \wedge (x_1 \vee \overline{x_3})$
0	0	0	0	1	0
0	0	1	1	0	0
0	1	0	1	1	1
0	1	1	1	0	0
1	0	0	0	1	0
1	0	1	1	1	1
1	1	0	1	1	1
1	1	1	1	1	1

Since for each formula up to 2^n assignments have to be tested, this general algorithm has an exponential time complexity (in the number of variables). Since the number of variables is bounded by the number km (m = number of clauses), the algorithm is of order $O(2^{km})$. But there are special cases where a faster algorithm exists. Consider for example the 1-SAT class. Here each formula has the form $l_1 \wedge l_2 \wedge \ldots \wedge l_m$, where l_i are literals, i.e. $l_i = x_k$ or $l_i = \overline{x_k}$ for some i. Since each literal has to be true so

that the formula is true, the following simple algorithm tests whether a given 1-SAT formula is satisfiable. Its idea is to scan the formula from left to right. Variables are set such that each literal becomes true. If a literal cannot be satisfied because the corresponding variable is already fixed, then the formula is not satisfiable. If on the other hand the end of the formula is reached, it is satisfiable.

algorithm 1-SAT
begin
  initially all x_i are unset;
  for $i := 1, 2 \ldots, m$ **do**
  begin
    let k be the number of variables occurring in literal l_i: $l_i = x_k$ or $l_i = \overline{x_k}$;
    if x_k is unset **then**
      choose x_k such that l_i=true;
    else
      if literal l_i=false **then**
        return(no); **comment** not satisfiable
  end
  return(yes); **comment** satisfiable
end

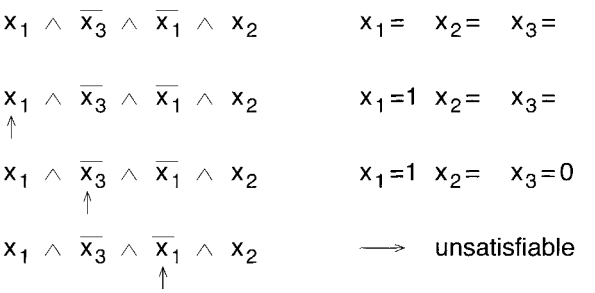


Figure 2.4: Sample run of algorithm 1-SAT for formula $x_1 \wedge \overline{x_3} \wedge \overline{x_1} \wedge x_2$.

Obviously the algorithm tests whether a 1-SAT formula is satisfiable or not. Fig. 2.4 shows, as an example, how the formula $x_1 \wedge \overline{x_3} \wedge \overline{x_1} \wedge x_2$ is processed. In the left column the formula is displayed and an arrow indicates the literal which is treated. The right column shows the assignments of the variables. The first line shows the initial situation. The first literal ($l_1 = x_1 \rightarrow k = 1$) causes $x_1 = 1$ (second line). In the second round ($l_2 = \overline{x_3} \Rightarrow k = 3$) $x_3 = 0$ is set. The variable of the third literal ($l_3 = \overline{x_1} \Rightarrow k = 1$) is set already, but the literal is false. Consequently, the formula is not satisfiable.
The algorithm contains only one loop. The operations inside the loop take a constant time. Therefore, the algorithm is $\Theta(m)$, which is clearly faster than $\Theta(2^{km})$. For 2-

SAT also a polynomial-time algorithm is known, for more details see [4]. Both 1-SAT and 2-SAT belong to the following class of problems:

Definition: P (polynomial) The class P contains all recognition-problems for which there exists a polynomial-time decision algorithm.

is of more practical interest. For 3-SAT problems no polynomial-time algorithm which checks satisfiability is known. On the other hand, up to now there is no proof that 3-SAT$\notin$P! But, since many clever people have failed to find an effective algorithm, it is very likely that 3-SAT (and k-SAT for $k > 3$) is not decidable in polynomial time.

There is another class of recognition problems A, which now will be defined. For this purpose we use *certificate-checking* (CC) algorithms. These are algorithms $\mathcal{A}$ which get as input instances $a \in$A like decision algorithms and additionally strings $s = s_1 s_2 \ldots s_n$, called *certificates* (made from a suitable alphabet). Like decision algorithms they halt on all inputs (a, s) and return only "yes" or "no". The meaning of the certificate strings will become clear from the following. A new class, called NP, can be described as follows:

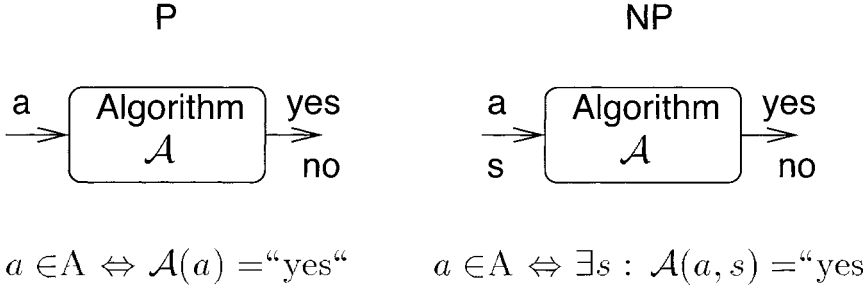


Figure 2.5: Classes P and NP.

The difference between P and NP is (see Fig. 2.5): for a yes-instance of a P problem the decision algorithm answers "yes". For a yes-instance of an NP problem there *exists* at least one certificate-string s such that the CC algorithm answers "yes", i.e. there may be many certificate strings s with $\mathcal{A}(a, s)$="no" even if a is a yes-instance. For a no-instance of a P problem the decision algorithm answers "no", while for a no-instance of an NP problem the CC algorithm answers "no" for *all* possible certificate strings s. As a consequence, P is a subset of NP, since every decision algorithm can be extended to a certificate-checking algorithm by ignoring the certificate.

The formal definition of NP is as follows:

Definition: NP (nondeterministic polynomial) A recognition-problem A is in the class NP, if there is a polynomial-time (in $|a|$, $a \in$A) certificate-checking algorithm with the following property:

An instance $a \in$A is a yes-instance if there is at least one certificate s with $\mathcal{A}(a, s)$=yes, for which the length $|s|$ is polynomial in $|a|$ ($\exists z : |s| \leq |a|^z$).

In fact, the requirement that the length of s is polynomial in $|a|$ is redundant, since the algorithm is allowed to run only a polynomial number of steps. During that time the

algorithm can read only a certain number of symbols from s which cannot be larger than the number of steps itself. Nevertheless, the length-requirement on s is included for clarity in the definition.
The concept of certificate-checking seems rather strange at first. It becomes clearer if one takes a look at k-SAT. We show k-SAT $\in$ NP: is of more practical interest.
Proof: Let $F(x_1, \ldots, x_n)$ be a boolean formula. The suitable certificate s for the k-SAT problem represents just one assignment for all variables of the formula: $s = s_1 s_2 \ldots s_n$, $s_i \in \{0, 1\}$. Clearly, the number of variables occurring in a formula is bounded by the length of the formula: $|s| \leq |F|$[1]. The certificate-checking algorithm just assigns the values to the variables ($x_i := s_i$) and evaluates the formula. This can be done in linear time by scanning the formula from left to right, similar to the algorithm for 1-SAT. The algorithm answers "yes" if the formula is true and "no" otherwise. If a formula is satisfiable, then, by definition, there is an assignment of the variables, for which the formula F is true. Consequently, then there is a certificate s for which the algorithm answers $\mathcal{A}(F, s)$=yes. QED
The name "nondeterministic polynomial" comes from the fact that one can show that a *nondeterministic algorithm* can decide NP problems in polynomial time. A normal algorithm is deterministic, i.e. from a given *state* of the algorithm, which consists of the values of all variables and the program line where the execution is at one moment, and the next state follows in a deterministic way. Nondeterministic algorithms are able to choose the next state randomly. Thus, a machine executing nondeterministic algorithms is just a theoretical construct, but in reality cannot be built yet[1]. The definition of NP relies on certificate-checking algorithms. For each CC algorithm an equivalent nondeterministic algorithm can be formulated in the following way. The steps where a CC algorithm reads the certificate can be replaced by the nondeterministic changes of state. An instance is a yes-instance if there is at least one run of the nondeterministic algorithm which answers "yes" with the instance as input. Thus, both models are equivalent.
As we have stated above, different recognition problems can be mapped onto each other. Since all algorithms which we encounter in this context are polynomial, only transformations are of interest which can be carried through in polynomial time as well (as a function of the length of an instance). The precise definition of the transformation is as follows:
Definition: Polynomial-time reducible Let A, B be two recognition problems. We say A is *polynomial-time* reducible to B ($A \leq_p B$), if there is a polynomial-time algorithm f such that

$$x \text{ is yes-instance of A} \quad \Leftrightarrow \quad f(x) \text{ is yes-instance of B}$$

Fig. 2.6 shows how a certificate-checking algorithm for B can be transformed into a certificate-checking algorithm for A using the polynomial-time transformation f.
As an example we will prove SAT$\leq_p$3-SAT, i.e. every boolean formula F can be written as a 3-CNF formula F^3 such that F^3 is satisfiable iff F is satisfiable. The transformation runs in polynomial time in $|F|$.

[1] Quantum computers can be seen as a realization of nondeterministic algorithms.

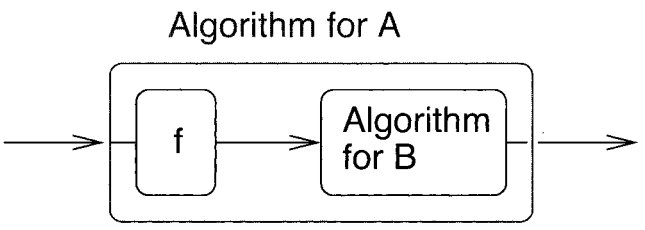


Figure 2.6: Polynomial-time reducibility: a certificate-checking algorithm for problem A consisting of the transformation f and the algorithm for B.

Example: Transformation SAT $\rightarrow$ 3-SAT

Let $F = C_1 \wedge C_2 \wedge \ldots \wedge C_m$ be a boolean formula in CNF, i.e. every clause C_p contains disjunctions of literals l_{pq}. We are now constructing a new formula F^3 by replacing each clause C_p by a sequence of clauses in the following way:

- If C_p has three literals, we do nothing.
- If C_p has more than three literals, say $C_p = l_1 \vee l_2 \vee \ldots \vee l_z$ $(z > 3)$, we introduce $z - 3$ new variables $y_1, y_2, \ldots, y_z$ and replace C_p by $z - 2$ clauses $(l_1 \vee l_2 \vee y_1) \wedge (\overline{y_1} \vee l_3 \vee y_2) \wedge \ldots (\overline{y_{z-3}} \vee l_{z-1} \vee l_z)$.
  Now assume that $C_p = 1$, then at least one $l_p = 1$. Now we choose $y_i = 1$ for all $i \leq p - 2$ and $y_i = 0$ for all $i > p - 2$. Then all new $z - 2$ clauses are true. On the other hand, if the conjunction of the $z - 2$ clauses is true, there must be at least one $l_i = 1$. Consequently, if C_p is satisfiable, then the new clauses are satisfiable as well and vice versa.
- Finally the case where C_p has less than three literals. If $C_p = l$ we replace it by $l \vee y_1 \vee y_2$ and if $C_p = l_1 \vee l_2$ we replace it by $l_1 \vee l_2 \vee y_1$. In order to keep (un)satisfiability we have to ensure that the new variables y_1, y_2 are always false. We cannot just add $\overline{y_1} \wedge \overline{y_2}$, because every clause has to contain three literals. Therefore, we have to add, with z_1, z_2 two additional new variables: $(\overline{y_1} \vee z_1 \vee z_2) \wedge (\overline{y_1} \vee \overline{z_1} \vee z_2) \wedge (\overline{y_1} \vee z_1 \vee \overline{z_2}) \wedge (\overline{y_1} \vee \overline{z_1} \vee \overline{z_2}) \wedge (\overline{y_2} \vee z_1 \vee z_2) \wedge (\overline{y_2} \vee \overline{z_1} \vee z_2) \wedge (\overline{y_2} \vee z_1 \vee \overline{z_2}) \wedge (\overline{y_2} \vee \overline{z_1} \vee \overline{z_2})$.

In the end we have a 3-CNF formula F^3 which is (un)satisfiable iff F is (un)satisfiable. The construction of F^3 obviously works in polynomial time. Consequently, SAT $\leq_p$ 3-SAT. □

There is a special subset of NP problems which reflects in some sense the general attributes of all problems in NP: it is possible to reduce all problems of NP to them. This leads to the following definition:

Definition: NP-completeness The recognition problem A $\in$ NP is called *NP-complete* if all problems B$\in$NP are polynomial reducible to A:

$$\forall \text{ B } \in \text{NP: B} \leq_p \text{A}$$

It can be shown that SAT is NP-complete. The proof is quite technical. It requires an exact machine model for certificate-checking algorithms. The basic idea is: each problem in NP has a certificate-checking algorithm. For that algorithm, a given instance and a given certificate, an equivalent boolean formula is constructed which is only satisfiable if the algorithm answers "yes" given the instance and the certificate. For more details see [4, 2, 5].
Above we have outlined a simple certificate-checking algorithm for SAT. Consequently, using the transformation from the proof that SAT is NP-complete, one can construct a certificate-checking algorithm for every problem in NP. In practice this is never done, since it is always easier to invent such an algorithm for each problem in NP directly.
Since SAT is NP-complete, A$\leq_p$SAT for every problem A$\in$NP. Above we have shown SAT$\leq_p$3-SAT. Since the $\leq_p$-relation is transitive, we obtain A$\leq_p$3-SAT. Consequently, 3-SAT is NP-complete as well. There are many other problems in NP which are NP-complete. For a proof it is sufficient to show A$\leq_p$B for any other NP-complete problem A, e.g. 3-SAT$\leq_p$B. The list of NP-complete problems is growing permanently. Several of them can be found in [6].
As we have said above, P is a subset of NP. If for one NP-complete problem a polynomial-time decision algorithm will be found one day, then, using the polynomial time reducibility, this algorithm can decide every problem in NP in polynomial time. Consequently, P=NP would hold. But for no problem in NP has a polynomial-time decision algorithm been found so far. On the other hand for no problem in NP is there a proof that no such algorithm exists. Therefore the so called P-NP-problem, whether P$\neq$NP or P=NP, is still unsolved, but P=NP seems to be very unlikely. We can draw everything that we know about the different classes in a figure: NP is a subset of the set of decidable problems. The NP-complete problems are a subset of NP. P is a subset of NP. If we assume P$\neq$NP, then problems in P are not NP-complete (see Fig. 2.7).
In this section we have concentrated on recognition problems. Optimization problems are not recognition problems since one tries to find a minimum or maximum. This is not a question that can be answered by "yes" or "no". But, every problem $\min H(\underline{\sigma})$ can be transformed into a recognition problem of the form

$$\text{"given a value } K\text{, is there a } \underline{\sigma} \text{ with } H(\underline{\sigma}) \leq K\text{?"}$$

It is very easy to see that the recognition problems for the TSP and the spin-glass ground state are in NP: given an instance of the problem and given a tour/a spin configuration (the certificates) the length of the tour/energy of the configuration can be computed in polynomial time. Thus, the question "is $H(\underline{\sigma}) \leq K$" can be answered easily.
If the corresponding recognition problem for an optimization problem is NP-complete, then the optimization problem is called *NP-hard*. In general, these are problems which

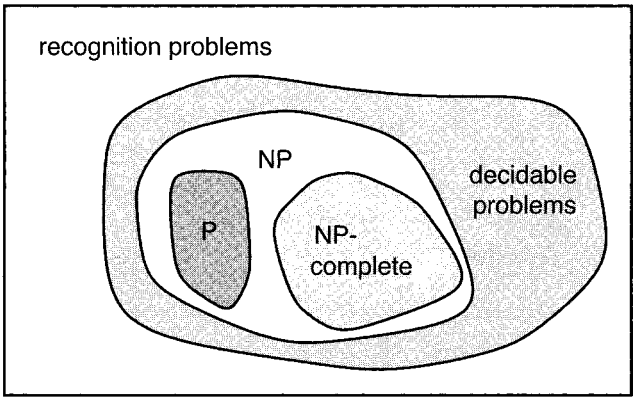


Figure 2.7: Relation between different classes of recognition problems.

are harder than problems from NP or which are not recognition problems, but every problem in NP can be reduced to them. This leads to the definition:

Definition: NP-hard Let A be a problem such that every problem in NP is polynomial reducible to A. If A$\notin$NP then A is called *NP-hard*.

From what we have learned in this section, it is clear that for an NP-hard problem no algorithm is known which finds the optimum in polynomial time. Otherwise the corresponding recognition problem could be solved in polynomial time as well, by just testing whether the thus obtained optimum is lower than m or not.

The TSP and the search for a ground state of spin glasses in three dimensions are both NP-hard. Thus, only algorithms with exponentially increasing running time are available, if one is interested in obtaining the exact minimum. Unfortunately this is true for most interesting optimization problems. Therefor, clever programming techniques are needed to implement fast algorithms. Here "fast" means slowly but still growing exponentially. In the next section, some of the most basic programming techniques are presented. They are not only very useful for the implementation of optimization methods but for all kinds of algorithms as well.

2.4 Programming Techniques

In this section useful standard programming techniques are presented: recursion, divide-and-conquer, dynamic programming and back-tracking. Since there are many specialized textbooks in this field [7, 8] we will demonstrate these fundamental techniques only by presenting simple examples. Furthermore, for efficient data structures, which also play a key role in the development of fast programs, we have to refer to these textbooks. On the Internet the LEDA library is available [9] which contains lots of useful data types and algorithms written in C++.

If a program has to perform many similar tasks this can be expressed as a loop, e.g.

with the **while**-statement from pidgin Algol. Sometimes it is more convenient to use the concept of *recursion*, especially if the quantity to be calculated has a recursive definition. One speaks of recursion if an algorithm calls itself. As a simple example we present an algorithm for the calculation of the factorial $n!$ of a natural number $n > 0$. Its recursive definition is given by:

$$n! = \begin{cases} 1 & \text{if } n = 1 \\ n \times (n-1)! & \text{else} \end{cases} \tag{2.6}$$

This definition can be translated directly into an algorithm:

```
algorithm factorial(n)
begin
   if n ≤1 then
      return 1;
   else
      return n*factorial(n − 1);
end
```

In the first line it is tested whether $n \leq 1$ instead of testing for $n = 1$. Therefore, it is guaranteed that the algorithm halts on all inputs.
During the execution of factorial(n) a sequence of nested calls of the algorithm is created up to the point where the algorithm is called with argument 1. The call to factorial(n) begins before and is finished after all other calls to factorial(i) with $i < n$. The hierarchy in Fig. 2.8 shows the calls for the calculation of factorial(4).

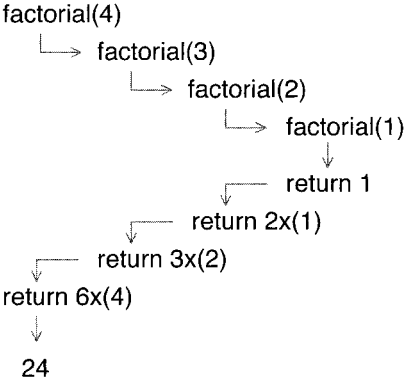


Figure 2.8: Hierarchy of recursive calls for calculation of factorial(4).

Every recursive algorithm can be rewritten as a sequential algorithm, containing no calls to itself. Instead loops are used. Usually, sequential versions are faster by some constant factor but harder to understand, at least if the algorithm is more complicated than in the present example. The sequential version for the calculation of the factorial reads as follows:

```
algorithm factorial2(n)
begin
   t := 1;   comment this is a counter
   f := 1;   comment here the result is stored
   while t ≤ n do
   begin
      f := f * t;
      t := t + 1;
   end
   return f;
end
```

The sequential factorial algorithm contains one loop which is executed n times. Thus, the algorithm runs in $\Theta(n)$ steps. For the recursive variant the time complexity is not so obvious. For the analysis of recursive algorithms, one has to write down a *recurrence* equation for the execution time. For $n = 1$, the factorial algorithm takes constant time $T(1)$. For $n > 1$ the algorithm takes the time $T(n-1)$ for the execution of factorial$(n-1)$ plus another constant time for the multiplication. Here and in the following, let C be the maximum of all occurring constants. Then, we obtain

$$T(n) = \begin{cases} C & \text{for } n = 1 \\ C + T(n-1) & \text{for } n > 1 \end{cases} \tag{2.7}$$

One can verify easily that $T(n) = Cn$ is the solution of the recurrence, i.e. both recursive and sequential algorithms have the same asymptotic time complexities. There are many examples, where a recursive algorithm is asymptotically faster than a straightforward sequential solution, e.g. see [7].

An important area for the application of efficient algorithms are sorting problems. Given n numbers (or strings or complex data structures) A_i $(i = 1, 2, \ldots, n)$ we want to find a permutation B_i of them such that they are sorted in (say) increasing order: $B_i < B_{i+1}$ for all $i < n$. There is a simple recursive algorithm for sorting elements. Please note that the sorting is performed within the array A_i they were provided in. Here this means the *values* of the numbers are not taken as arguments, i.e. there are no local variables which take the values. Instead the variables (or their memory positions) themselves are passed to the following algorithm. Therefore, the algorithm can change the original data. The basic idea of the algorithm is to look for the largest element of the array, store it in the last position, and sort the first $n-1$ elements by a recursive call. The algorithmic presentation follows on the next page.

algorithm sort(n,$\{A_1, \ldots, A_n\}$)
begin
  if $n = 1$ **then**
    return;
  $max := 1$; **comment** will contain maximum of all A_i
  $pos := 1$ **comment** will contain position of maximum
  $t := 2$;
  while $t \leq n$ **comment** look for maximum
    if $A_t > max$ **then**
    begin
      $max := A_t$;
      $pos := t$;
    end
    exchange maximum and last element;
    sort($n-1$,$\{A_1, \ldots, A_{n-1}\}$)
end

In Fig. 2.9 it is shown how the algorithm runs with input $(6, \{5, 9, 3, 6, 2, 1\})$. On the left side the recursive sequence of calls is itemized. The maximum element for each call is marked. In the right column the actual state of the array before the next call is displayed.

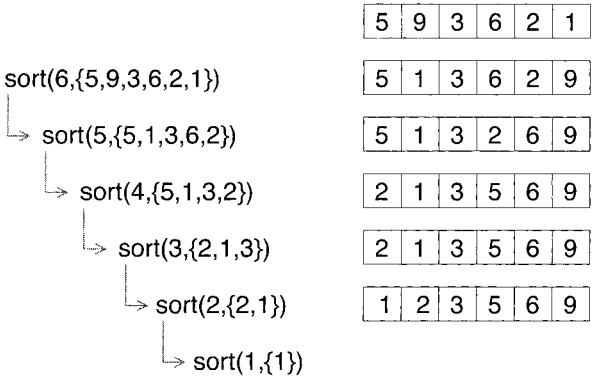


Figure 2.9: Run of the sorting algorithm with input $(6, \{5, 9, 3, 6, 2, 1\})$.

The algorithm takes linear time to find the maximum element plus the time for sorting $n-1$ numbers, i.e. for the time complexity $T(n)$ one obtains the following recurrence:

$$T(n) = \begin{cases} C & (n = 1) \\ Cn + T(n-1) & (n > 1) \end{cases} \tag{2.8}$$

Obviously, the solution of the recurrence is $\Theta(n^2)$. Compared with algorithms for NP-hard problems this is very fast. But there are sorting-algorithms which can do even

better. One of them is "mergesort" which relies on the principle of *divide-and-conquer*. The basic idea of this principle is to divide a problem into smaller subproblems, solve the subproblems and then combine the solutions of the subproblems to form the final solution. Recursive calls of the algorithm are usually applied here as well.
The basic idea of mergesort is to part the set which is to be sorted into two subsets of roughly equal size, sort them recursively and finally *merge* the two sorted sequences into one sorted sequence. The merging is performed by iteratively removing the smallest element of both sequences. Without loss of generality we assume that the number n of items to be sorted is a power of 2. The algorithm reads as follows:

algorithm mergesort(n,$\{A_1, \ldots, A_n\}$)
begin
  if $n = 1$ **then**
    return
  for $i := 1, 2, \ldots, n/2$ **do** **comment** divide set
  begin
    $B_i := A_i$;
    $C_i := A_{i+n/2}$;
  end
  mergesort($n/2, \{B_1, \ldots, B_{n/2}\}$); **comment** sort subsets
  mergesort($n/2, \{C_1, \ldots, C_{n/2}\}$);
  $x := 1$; $y := 1$; **comment** largest elements of sequences
  for $i := 1, \ldots, n$ **do** **comment** merge sorted subsets
    if $x \leq n/2$ AND $B_x < C_y$ **then**
      $A_i := B_x$; $x := x + 1$;
    else
      $A_i := C_y$; $y := y + 1$;
end

The hierarchy of recursive calls of mergesort$(4, \{5, 2, 3, 1\})$ is displayed in the upper part of Fig. 2.10. In the lower part the merging of the sorted subset is shown. For $n = 2^k$ one obtains $k + 1$ layers in the hierarchy of calls.
The division of the sets and the merge-operation takes $\Theta(n)$ time, while each recursive call takes $T(n/2)$. Hence, the recurrence for this algorithms reads:

$$T(n) = \begin{cases} C & (n = 1) \\ Cn + 2T(n/2) & (n > 1) \end{cases} \tag{2.9}$$

If n is large enough this recurrence can be solved by $T(n) = n \log n$. Consequently, the divide-and-conquer realization of sorting is asymptotically faster than the simple recursive sort-algorithm.
Another problem where the application of divide-and-conquer and recursion seems quite natural is the calculation of *Fibonachi numbers* fib(n). Their definition is as follows:

$$\text{fib}(n) = \begin{cases} 1 & (n = 1) \\ 1 & (n = 2) \\ \text{fib}(n-1) + \text{fib}(n-2) & (n > 2) \end{cases} \tag{2.10}$$

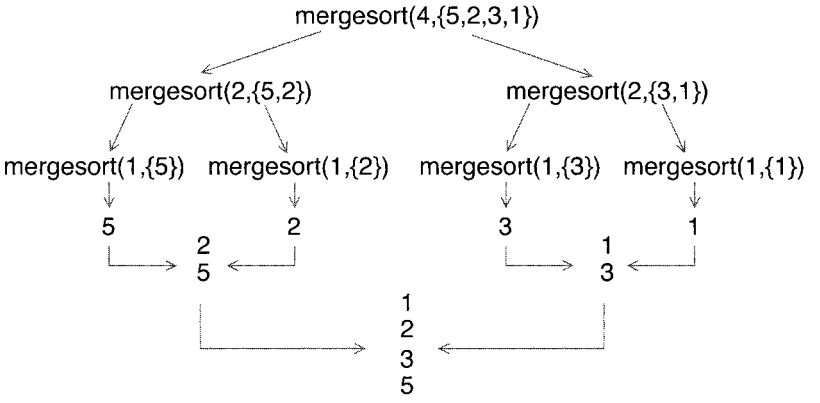


Figure 2.10: Call of mergesort$(4, \{5, 2, 3, 1\})$.

Thus, e.g. fib(4) = fib(3) + fib(2) = (fib(2) + fib(1)) + fib(2) = 3, fib(5) = fib(4) + fib(3) = 3 + 2 = 5. The functions grows very rapidly: fib(10) = 55, fib(20) = 6765, fib(30) = 83204, fib(40) $> 10^8$. Let us assume that this definition is translated directly into a recursive algorithm. Then a call to fib(n) would call fib($n-1$) and fib($n-2$). The recursive call of fib($n-1$) would call *again* fib($n-2$) and fib($n-3$) [which is called from the two calls of fib($n-2$), etc.]. The total number of calls increases rapidly, even more than fib(n) itself increases. In Fig. 2.11 the top of a hierarchy of calls is shown. Obviously, every call to fib with a specific argument is performed frequently which is definitely a waste of time.

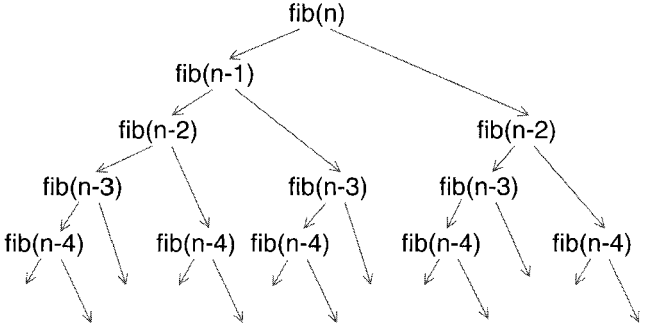


Figure 2.11: Hierarchy of calls for fib(n).

Instead, one can apply the principle of *dynamic programming*. The basic idea is to start with small problems, solve them and store them for later use. Then one proceeds with larger problems by using divide-and-conquer. The basic idea of If for the solution of a larger problem a smaller one is necessary, it is already available.

Therefore, no direct recursive calls are needed. As a consequence, the performance increases drastically. The divide-and-conquer algorithm for the Fibonachi-numbers reads as follows, the array $f[]$ is used to store the results:

algorithm fib-dynamic(n)
begin
  if $n < 3$ **then**
    return 1;
  $f[1] := 1$;
  $f[2] := 1$;
  for $i := 3, 4, \ldots, n$ **do**
    $f[i] := f[i-1] + f[i-2]$
  return$f[n]$;
end

Since the algorithm contains just one loop it runs in $\Theta(n)$ time.
The last basic programming principle which is presented here is *backtracking*. This method is applied when there is no direct way to compute a solution. This is typical for many optimization problems. Remember the simple TSP algorithm which constructs a feasible tour, but which does not guarantee to find the optimum. In order to improve the method, one has to try some (sub-)solutions and throw them away if they are not good enough. This is the basic principle of backtracking.
In the following we will present a backtracking algorithm for the solution of the N-queens problem.

Example: N queens problem

N queens are to be placed on an $N \times N$ chess board in such a way that no queen checks against any other queen.

This means that in each row, in each column and in each diagonal at most one queen is placed. □

A naive solution of the algorithms works by enumerating all possible configurations of N queens and checking for each configuration whether any queen checks against another queen. By restricting the algorithm to place at most one queen per column, there are N^N possible configurations.
The idea of backtracking is to place one queen after the other. One stops placing further queens if a non-valid configuration is already obtained at an intermediate stage. Then one goes one step back, removes the queen which was placed at the step before, places it elsewhere if possible and continues again.
In the algorithm, we use an array Q_i which stores the position of the queen in column i. If $Q_i = 0$, no queen has been placed in that column. For simplicity, it is assumed that N and the array Q_i are global variables. Initially all Q_i are zero.

The algorithm starts in the last column and places a queen. Next a queen is placed in the second last column by a recursive call and so forth. If all columns are filled a valid configuration has been found. If at any stage it is not possible to place any further queen in a given column then the backtracking-step is performed: the recursive call finishes, the queen which was set in the recursion-step before is removed. Then it is placed elsewhere and the algorithm proceeds again.

```
algorithm queens(n)
begin
   if n = 0 then
      print array Q_1, ..., Q_N; problem solved;
   for i := 1, ..., N do
   begin
      Q_n := i
      if Queen n does not check against any other then
         queens(n - 1);
   end
   Q_i := 0;
   return
end
```

In Fig. 2.12 it is shown how the algorithm solves the problem for $N = 4$. It starts with a queen in column 4 and row 1. Then queens(3) is called. The positions where no queen is allowed are marked with a cross. For the third column no queens in row 1 and row 2 are allowed. Thus, a queen is placed in row 3 and queens(2) is called. In the second column it is not possible to place a queen. Hence, the call to queens(2) finishes. The queen in column 3 is placed on a deeper row (second line in Fig. 2.12). Then, by calling queens(2), a queen is placed in row 2. Now, no queen can be placed in the first column. Since there was only one possible position in column 2, the queen is removed and the call finishes. Now, both possible positions for the queen in column 3 have been tried. Therefore, the call for queens(3) finishes as well and we are back at queens(1). Now, the queen in the last column is placed in the second row (third line in Fig. 2.12). From here it is straight forward to place queens in all columns and the algorithm succeeds.

Although this algorithm avoids many "dead ends" it still has an exponential time complexity as a function of N. This is quite common to many hard optimization problems. More sophisticated algorithms, which we will encounter later, try to reduce the number of configurations visited even more.

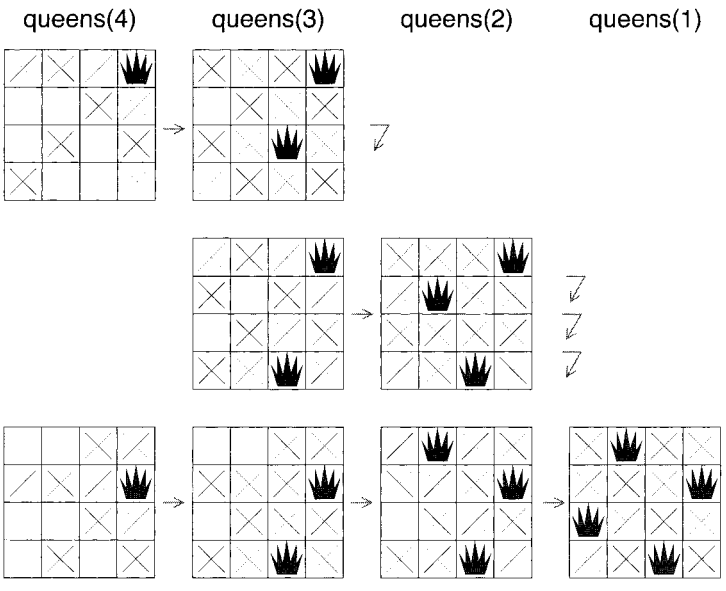


Figure 2.12: How the algorithm solves the 4-queens problem.

Bibliography

[1] Stephan Mertens, http://itp.nat.uni-magdeburg.de/~mertens/TSP/index.html (1999)

[2] M.D. Davis, R. Sigal, and E.J. Wyuker, *Computability, Complexity and Languages*, (Academic Press, San Diego 1994)

[3] B. Hayes, *Americ. Sci.* **85**, 108 (1997)

[4] S.A. Cook, *J. Assoc. Comput. Mach.* **18**, 4-18 (1971)

[5] C.H. Papadimitriou and K. Steiglitz, *Combinatorial Optimization*, (Dover Publications, Mineola (NY) 1998)

[6] M.R. Garay and D.S. Johnson, *Computer and Interactability A Guide to the Theory of NP-Completeness*, (W.H. Freeman & Company, San Francisco 1979)

[7] A.V. Aho, J.E. Hopcroft, and J.D. Ullman, *The design and analysis of computer algorithms*, (Addison-Wesley, Reading (MA) 1974)

[8] R. Sedgewick, *Algorithms in C*, (Addison-Wesley, Reading (MA) 1990)

[9] K. Mehlhorn and St. Näher, *The LEDA Platform of Combinatorial and Geometric Computing*, (Cambridge University Press, Cambridge 1999);
see also http://www.mpi-sb.mpg.de/LEDA/leda.html

3 Graphs

3.1 Graphs

Many optimization problems from physics or other areas can be mapped on optimization problems on graphs. Some of these transformations will be useful later on. For this reason a short introduction to graph theory is given here. Only the basic definitions and algorithms are presented. For more information, the reader should consult a specialized textbook on graph theory, e.g. Refs. [1, 2, 3]. We will sometimes only mention a real world application which can be treated with a given graph theoretical concept. The precise definitions of the applications will be given later on.

Consider a map of a country where several towns, villages or other places are connected by roads or railways. Mathematically this setting can be represented by a *graph*. A graph consists of *nodes* and *edges*. The nodes represent the towns, villages or other places and the edges describe the roads or railways. Formally, the definition of a graph is given by:

Definition: Graph A graph G is an ordered pair $G = (V, E)$ where V is a set and $E \subset V \times V$. An element of V is called a *vertex* or *node*. An element $(i, j) \in E$ is called an *edge* or *arc*. In a physical context, where edges represent interactions between particles, edges are often called *bonds*.

If the pairs $(i, j) \in E$ are ordered pairs, then G is called a *directed* graph. Otherwise G is called *undirected* then (i, j) and (j, i) denote the same edge. A graph $G' = (V', E')$ is called *subgraph* of G if it has the properties $V' \subset V$ and $E' \subset E$ ($E' \subset V' \times V'$ by definition). The empty graph $(\emptyset, \emptyset)$ is a subgraph of all graphs.

First, some further notations are given which apply to both directed and undirected graphs. Usually we restrict ourselves to *finite* graphs, i.e. the set of nodes and edges are finite. In this case we denote by $n = |V|$ the number of vertices and by $m = |E|$ the number of edges. Let $i \in V$ be a vertex. If $(i, j) \in E$ we call j a *neighbor* of i (and vice versa). Both nodes are *adjacent* to each other. The set $N(i)$ of neighbors of i is given by $N(i) = \{j | (i, j) \in E \vee (j, i) \in E\}$. The degree $d(i)$ of node i is the cardinality of the set of neighbors: $d(i) = |N(i)|$. A vertex with degree 0 is called *isolated*.

A *path* from v_1 to v_k is a sequence of vertices $v_1, v_2, \ldots, v_k$ which are connected by edges: $(v_i, v_{i+1}) \in E \; \forall i = 1, 2, \ldots, k-1$. The *length* of the path is $k - 1$. If $v_1 = v_k$ the path is called *closed*. If no node except the first and the last one appears twice in a closed path, it is called a cycle. A set of nodes is called a *connected component*, if it contains all nodes where from each node a path to each other node of the set exists. A graph is called *connected* if it has only one connected component.

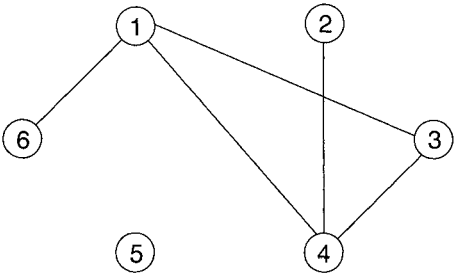


Figure 3.1: An undirected graph.

Example: Graph

In Fig. 3.1 the graph $G = (\{1,2,3,4,5,6\}, \{(1,3), (3,4)\ (4,1), (4,2), (6,1)\})$ is shown. The nodes are represented by circles and the edges by lines connecting the circles. The graph has $n = 6$ vertices and $m = 5$ edges; e.g., nodes 3 and 4 are adjacent. The set of neighbors of vertex 1 is $N(1) = \{3,4,6\}$. Thus, node 1 has degree 3 while node 2 has only degree 1. Node 5 is isolated. The graph contains the path $6,1,4,3$ from node 6 to node 3 of length 3 and the cycle $1,3,4,1$. □

Now some definitions are given which apply only to directed graphs. For an edge $e = (i,j)$, i is the *head* and j the *tail* of e. The edge e is called *outgoing* from i and *incoming* to j. Please note that for a directed path it is important that all edges point into the direction of the path, formally the definition is the same as in the case of an undirected graph. A set of nodes is called a *strongly connected component* (SCC), if from each of its nodes a directed path to each other node of the set exists. In a directed graph, the outgoing and incoming edges can be counted separately. The indegree is given by $id(i) = |\{j|(j,i) \in E\}|$ and the outdegree is $od(i) = |\{j|(i,j) \in E\}$. Obviously, for all vertices $d(i) = id(i) + od(i)$.

Example: Directed graph

When drawing a directed graph, the edges (i,j) are represented by arrows pointing from i to j. If one considers the example graph from above as directed, one obtains Fig. 3.2. Here, the sequence $6,1,4,3$ is not a path since the edges $(4,1)$ and $(3,4)$ point in the wrong direction. On the other hand, the graph contains the path $6,1,3,4$. Node 1 has the outdegree $od(1) = 1$ and indegree $id(1) = 2$. The total degree is $d(1) = id(1) + od(1) = 3$ as in the case of the undirected graph. □

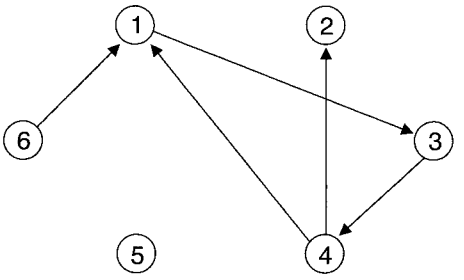


Figure 3.2: A directed graph.

3.2 Trees and Lists

A very important subclass of graphs are connected graphs without cycles. They are called *trees*. The name arises from the fact that it is possible to draw a tree in the following way:

- All nodes are arranged in several levels $l = 0, 1, \ldots, h$.
- There are edges only between vertices of adjacent levels l and $l + 1$. A node on level l is called *father* and its neighbor on level $l + 1$ is called *son*.
- On level zero there is only one node, called the *root*.

Hence, trees are very often used to represent hierarchical structures. Let v be a node. The sons of v, their sons, and so on are called *descendants* of v. The father of v, its father, and so on are called *ancestors* of v. Thus, the root is an ancestor of all other nodes. The nodes which have no descendants are called *leaves*. The index h of the largest level is called the *height* of a tree. The height is equal to the length of the longest path from the root to a leaf. Each node v can be regarded as a root of a *subtree*, which is given by v and its descendants.

Example: Tree

In Fig. 3.3 an undirected tree is shown. The root is usually displayed at the top. Here, node 7 is the root. Node 6 is a descendant of the root and an ancestor of node 5. Nodes 1, 2, 5, 9, 10, 11 are leaves. The tree has height 3. The subtree which has node 3 as a root contains the nodes 2, 3, 4, 11. □

For directed graphs, usually graphs without cycles are called trees only in the case where there are only edges from levels l to $l + 1$ and no edges in the other direction. Sometimes a directed graph is called a tree only if the undirected version of the graph contains no cycles.

An important application of trees in the field of computer science are *search trees*. They are used to store a collection of objects in an ordered way. Thus, an order relation "$\leq$" must be given for the objects. Search trees have the following properties:

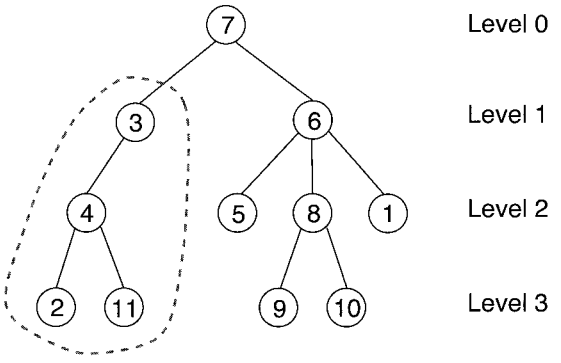


Figure 3.3: A tree containing 11 nodes.

- They are *binary* trees. This means that each node has at most two sons, called *left* and *right* son. The subtree which has the left (right) son as the root is called *left (right) subtree.*
- In each node one object O_1 is stored.
- The following property is true for all objects O_1: in the left (right) subtree only objects are stored which are smaller (equal to or larger) than O_1 according to the order relation "$\leq$".

This special organization allows the design of very fast algorithms for finding, inserting and deleting elements within a search tree while always keeping the correct order. An example will be given in Sec. 3.5. In Fig. 3.4 a search tree containing natural numbers sorted in ascending order is shown.

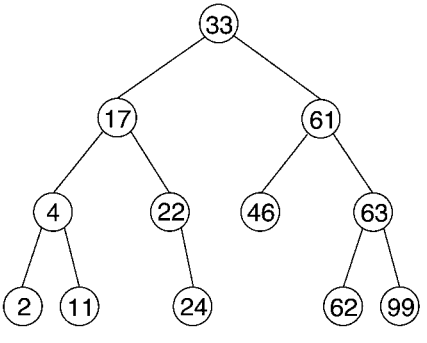


Figure 3.4: A binary tree containing the nodes 2, 4, 11, 17, 22, 24, 33, 46, 61, 62, 63, 99 sorted in ascending order.

Similar to search trees are *heaps*. They are binary trees as well, but here the smallest element of each subtree is always stored at the root of the subtree, which means that the smallest element of the whole tree is stored at the root of the tree. This allows for an efficient implementation of *priority queues*, which always give you the next element with the currently lowest priority (*remove* operation). After the root element has been removed, it is replaced by the lower of the two elements in the roots of the left and right subtree, i.e. the lower of both elements moves up one level. In the same way the element which has moved up is replaced by its smaller son, etc. Adding an element to a heap is done in the following fashion. The element is added as a son of a leaf. If it is smaller than the element of its father, both elements are swapped. This process is continued until the current father of the new element is smaller than the new element. Thus, eventually the new element may rise to the root, if it is smaller than all other elements in the tree. As a consequence, the *insert* and *remove* operations can be performed in logarithmic time.
A binary tree is called *complete* if each node is either a leaf or has two sons. It can easily be shown by induction that a complete tree with height h has $n = 2^{h+1} - 1$ nodes and 2^h leaves.

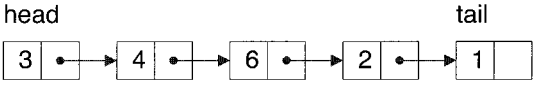


Figure 3.5: A list containing 5 elements.

A very simple type of graph is a *list*. Lists can be regarded as special types of trees which have exactly one root and exactly one leaf and every node has at most one son. The nodes of a list are called *elements*, the root is called the *head* of the list and the leaf is called the *tail*. For historic reasons lists are often drawn in a slightly different manner to other graphs. In Fig. 3.5 a list containing 5 elements is shown.

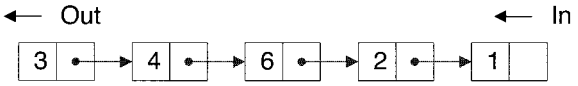


Figure 3.6: A queue.

Several special types of lists are very useful in computer science. For *queues* it is only possible to add new elements at the tail and to remove elements at the head, see Fig. 3.6. They are called FIFO ("first in – first out") lists as well. An application of queues are printer queues. The job which enters the queue at first will be printed at first. Other jobs which are added while a job is being printed have to wait.
Lists are called *stacks* if adding and removing elements takes place only at the head (see Fig. 3.7). They behave in an LIFO ("last in first out") manner. Stacks are used for example to realize recursion. Whenever a call to a procedure occurs the computer

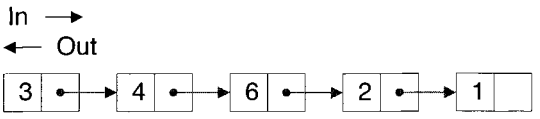


Figure 3.7: A stack.

has to remember where to proceed when the procedure is finished. These so called *return addresses* are stored on a stack, because the procedure which was called last will be finished first.

3.3 Networks

Consider again the TSP (see Chap. 1). It can be represented by an undirected graph if one identifies the cities with nodes and the connections between cities with edges. In order to describe the TSP completely, the distances between different cities have to be represented as well. This can be done by introducing a function $d : E \to \mathcal{R}$ from edges to real numbers. For an edge $e = (i, j) \in E$ the distance between the cities i and j is given by $d(e)$.
Sometimes functions $f : V \to A$ (A an arbitrary set) on vertices are useful as well. An arbitrary function on vertices or on edges is called *labeling*. A graph together with a labeling is called a *labeled* graph. Typical examples for labelings are distances, costs or capacities.
The case where the edges are labeled with capacities is so important that it has its own name: a *network*. For example a system of pumps connected via pipes which can transport some fluid like water is a network. Our kind of networks contain two special pumps: the source where the fluid enters the network and the sink where the fluid leaves the network. The capacities of the pipes tell how much fluid can be transported through each pipe per time unit. The exact definition reads as follows:
Definition: Network A network is a tuple $N = (G, c, s, t)$ where

- $G = (V, E)$ is a directed graph without edges of the form (i, i).
- $c : E \to \mathcal{R}_0^+$ is a positive labeling of the edges. Additionally, we assume $c((i, j)) = 0$ if $(i, j) \notin E$.
- $s \in V$ is a vertex called *source* with no incoming edges $di(s) = 0$.
- $t \in V$ is a vertex called *sink* with no outgoing edges $do(t) = 0$.

In Fig. 3.8 an example of a network is shown. The capacities are the numbers written next to the edges. Please note that it is possible to define a network via an undirected graph as well. For the models we encounter here, directed networks are sufficient.
An actual *flow* of fluid through the network can be described by introducing another labeling $f : V \times V \to \mathcal{R}$. A flow through an edge is always bounded by its capacity.

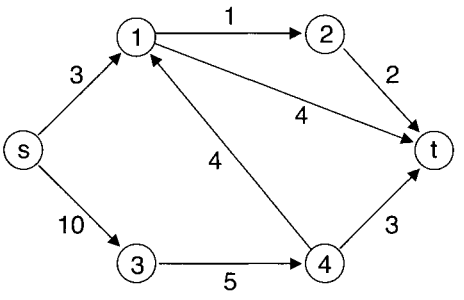


Figure 3.8: A network.

For technical reasons, negative values of the flow are allowed as well. A negative value $f(i,j) < 0$ means a positive flow from node i to node j and vice versa. Furthermore, the flow is conserved in all nodes, except the source and the sink. In total we obtain [by writing $f_{ij} \equiv f(i,j)$, $c_{ij} \equiv c((i,j))$]

- Capacity constraint: $f_{ij} < c_{ij}$ for all $i,j \in V$
- Negative flows: $f_{ij} = -f_{ji}$ for all $i,j \in V$
- Flow conservation: $\sum_j f_{ij} = 0$ for all $i \in V$ with $i \neq s,t$

Please note that by combining the first and second properties $f_{ji} > -c_{ij}$ is obtained. The total flow through the graph is given by $f_0 = \sum_j f_{s,j} = -\sum_j f_{tj}$. Determining the maximum possible amount which can flow from s to t through a network, called the *maximum flow*, is an optimization problem. For example, the maximum flow for the network in Fig. 3.8 is eight. Later we will see that the determination of the maximum flow can be solved in polynomial time. In physics, we apply these methods to calculate ground states of disordered magnetic materials such as random field systems or diluted antiferromagnets.

The *minimum cost flow* problem is related to the maximum-flow problem. Here, additional costs $c' : E \to \mathcal{R}$ and a total flow f_0 are given. $c'(e)$ specifies the cost of sending one unit of flow through edge e. More generally, the cost may depend arbitrarily but in a convex way on the flow through an edge. The minimum cost flow problem is to find a flow of minimum total cost $\sum_e c'(e)f(e)$, which is compatible with the capacity/conservation constraints and has the total value f_0. Later it will be shown how ground states of solid-on-solid systems or of vortex glasses (from the theory of superconductivity) can be obtained by minimizing cost flows. For this problem polynomial time algorithms are known as well. The case $f_0 \equiv 0$ frequently occurs. Please note that the optimum flow may not vanish inside the network in this case as well, since the costs can take negative values thus making circular flows favorable.

3.4 Graph Representations

If we want to apply graphs on a computer, somehow we have to represent them in the memory. Different ways of storing graphs are known. Which one is most efficient choice depends on the kind of application. We start with the storage of lists, since they are used for the representation of the other graph types. Then trees are discussed. At the end of the section two methods of representating arbitrary graphs are shown.
Since lists are simply linear sequences of elements, the straight forward implementation uses arrays. A list of n elements can be stored in array $L[1 \ldots n]$ e.g. by storing the head in $L[1]$, its son in $L[2]$, etc., and the tail in $L[n]$. The advantage of this realization is its simplicity. Furthermore, it is possible to access all elements of the list in constant time. A minor problem is that one has to know the maximum size of the list when allocating the memory for the array. But, if the maximum size is not known, one can reallocate the memory if the list grows too long. If the reallocation occurs rarely, the additional computer time needed can be neglected.

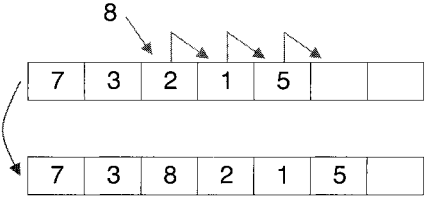


Figure 3.9: Element 8 is inserted into an array.

One major problem is the following: if elements are to be inserted or to be removed somewhere within the list, large chunks of memory have to be moved around each time the operation occurs (see Fig. 3.9). Thus, insert and delete operations take $\Theta(n)$ time. For stacks and queues this problem does not occur since elements are inserted and deleted only at the head or at the tail. Consequently, stacks and queues are usually realized by arrays. A stack is usually stored in reverse order in an array, adding and removing elements occurs at the end of the stack which is given by a pointer variable. Please remember that a pointer is just a memory address. For a queue two pointers are necessary: one which stores the head and one which holds the tail of the queue. If a pointer reaches the end of the array it is set to the beginning again.
An implementation of general lists that is more flexible uses pointers as well. For each element of a list, a small chunk of memory is allocated separately. In addition to the information associated with each element, a pointer which contains the address of the following element is stored. At the tail of the list the pointer has the special value NULL. This realization is reflected directly by the manner in which lists are drawn: the arrows represent the pointers. If the position where an operation is to be performed is given already, insert and delete operations can be performed in constant time just by (re-)assigning some pointers. One has to be a little bit careful not to loose parts of the chain, for details see e.g. [4]. The insert and delete operations become especially

simple if along each element *two* pointers are stored: one to the next element and one to the preceding element. In this case the list is called *double linked*. A drawback of the realization with pointers is that it is not possible to access elements of the list (say the 20th element) directly. One has to start at the head or the tail and go through the list until the element is reached. Furthermore, this type of realization consumes more memory, since not only the elements but the pointers have to be stored as well. The situation which we encounter here is typical for computer science. *There are now perfect data structures*. The application has to be taken into account to find the best realization.

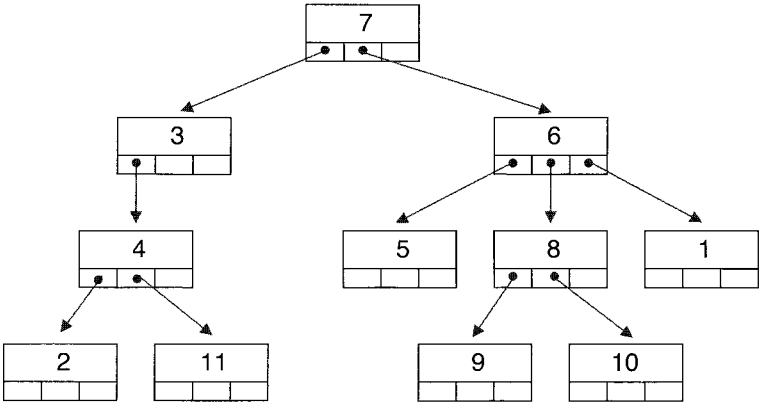


Figure 3.10: Representation of the tree from Fig. 3.3 via nodes containing arrays of pointers. Here each array contains at most three entries per node.

This is true for trees as well. If the maximum number of sons for each node is known, usually the following type of realization is used, which is similar to the representation of lists via pointers. Along with the information which is to be stored in a node, an array containing pointers to the sons is kept. If a node has less than the maximum number of sons, some pointer values contain the value NULL. In Fig. 3.10 a realization of the tree from Fig. 3.3 is shown.
For binary trees (among them heaps) there is a very simple realization using one array T. The basic idea is to start at the root and go through each level from left to right. This gives us the order in which nodes are stored in the array. One obtains:

- The root is stored in $T[1]$.
- If a node is empty, a special value (say -1) is assigned.
- For each node $T[k]$ the left son is stored in $T[2k]$ and the right son is kept in $T[2k+1]$.

In Fig. 3.11 the array representation of the search tree from Fig. 3.4 is shown. This type of tree realization is comparable to the array representation of lists: it is very

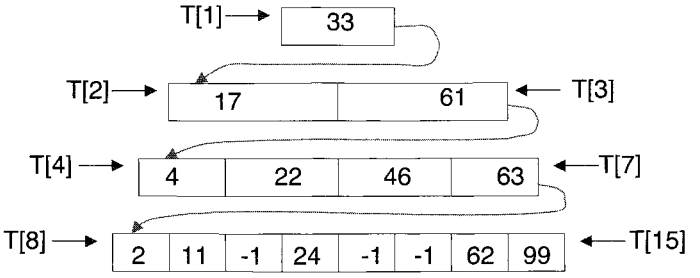


Figure 3.11: The representation of a binary tree via an array.

easy to build up the structure and to access nodes but insert and delete operations are time consuming.
We finish this section by explaining two methods to represent arbitrary graphs. Let $G = (V, E)$ be a graph. Without loss of generality we assume $V = \{1, 2, \ldots, n\}$.
An *adjacency matrix* $\underline{\underline{A}} = (a_{ij})$ for G is an $n \times n$ matrix with entries $0, 1$, where n is the number of nodes. In the case G is directed we have:

$$a_{ij} = \begin{cases} 1 & \text{for} \quad (i,j) \in E \\ 0 & \text{else} \end{cases} \tag{3.1}$$

For an undirected graph, the adjacency matrix $\underline{\underline{A}}$ contains nonzero elements $a_{ij} = a_{ji} = 1$ for each edge $(i, j) \in E$, i.e. in this case the matrix is symmetric.
The adjacency matrices for the example graphs of Sec. 3.1 are shown in Fig. 3.12.

$$\begin{pmatrix} 0 & 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} \qquad \begin{pmatrix} 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

undirected directed

Figure 3.12: Adjacency matrices for graph $G = (\{1, 2, 3, 4, 5, 6\}, \{(1, 3), (3, 4) (4, 1), (4, 2), (6, 1)\})$. The left matrix shows the case when G is regarded as an undirected graph, while the right one is for the case of a directed graph.

The advantage of the matrix representation is that one can access each edge within a constant amount of computer time. On the other hand, an adjacency matrix needs $\Theta(n^2)$ memory elements to be stored. Consequently, this choice is inefficient for storing *sparse* graphs, where the number of edges is $m \in \Theta(n)$.
For sparse graphs, it is more efficient to use *adjacency lists*: for each node i a list L_i is stored which contains all its neighbors $N(i)$. Hence, for directed graphs, L_i contains

all vertices j with $(i,j) \in E$ while for an undirected graph it contains all vertices j with $(i,j) \in E$ or $(j,i) \in E$. Please note that the elements in the list can be in arbitrary order. The list representation uses only $O(m)$ memory elements. With this realization it is not possible to access an edge (i,j) directly, since one has to scan through all elements of L_i to find it. But for most applications a direct access is not necessary, so this realization of graphs is widely used. A similar method is applied for the LEDA-library package [5].

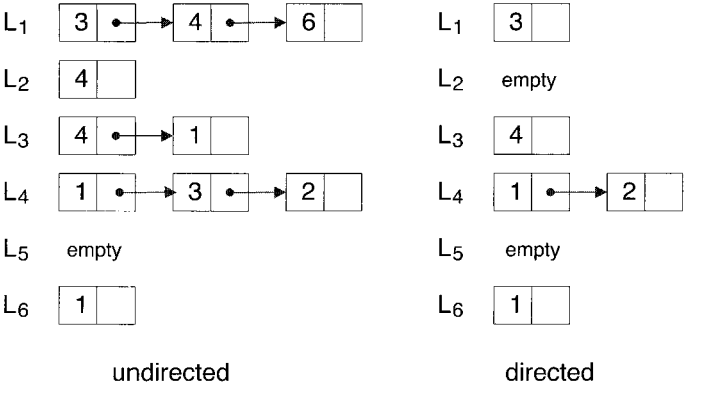


Figure 3.13: Adjacency lists for the example graphs. The left lists represent the case when G is regarded as an undirected graph, while the right are for the case of a directed graph.

The adjacency lists for the sample graphs from above are shown in Fig. 3.13.
For a directed graph, it is sometimes very convenient to have all incoming edges for a given vertex j available. Then one can store a second set of lists K_j containing all nodes i with $(i,j) \in E$. The lists K_j for the directed version of the sample graph are shown in Fig. 3.14.

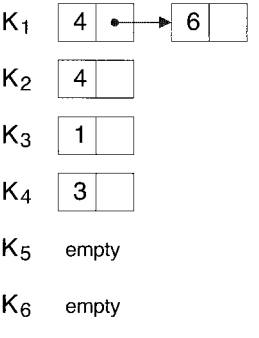


Figure 3.14: List of incoming edges for directed example graph.

The methods presented above can easily be extended to implement labeled graphs. Labels of vertices can be represented by arrays. For labels of edges one can either use matrices or store the labels along with the list elements representing the edges.

3.5 Basic Graph Algorithms

Most basic algorithms on graphs, trees and lists can be found in [6, 4]. Here we present just one algorithm for searching in a tree This should be sufficient to make the reader familiar with the subject. More simple graph algorithms can be found in Chap. 4.
Consider a search tree, as introduced in Sec. 3.2. One basic operation is the *find* operation. It tests whether an object O_1 is contained in the tree or not. The algorithm starts at the root of the tree. If the tree is empty, then the object is not contained in the tree. If O_1 is stored at the root it is found. In both cases the algorithm terminates. Otherwise, if O_1 is smaller than the object at the root, the search continues in the left subtree. If O_1 is larger, the search continues in the right subtree. The recursive formulation of the algorithm reads as follows:

```
algorithm find(tree, object)
begin
   if tree is empty then
      return(not found)
   if root contains object then
      return(tree)
   if object at root-node ≥ object then
      find(left subtree, object)
   else
      find(right subtree, object)
end
```

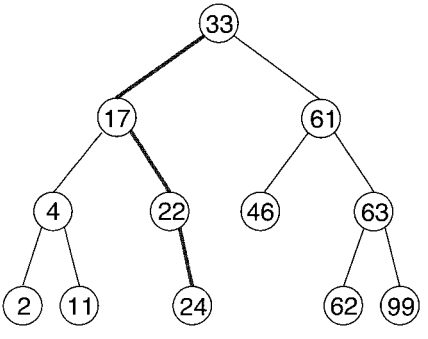


Figure 3.15: Search for element 24 in a sorted binary tree.

Example: Search in tree

We trace the search for element 24 in the tree shown in Fig. 3.15. The algorithm is called with the root of the tree and number 24. Since the tree is not empty and 24 is smaller than the object at the root (33), a recursive call with the left subtree occurs. Here, number 17 is stored at the root. Thus, the procedure is called with the right subtree of the first subtree. In the next step again the search continues in the right subtree where finally the object is found. □

The algorithm performs one descent into the tree. Hence, its time complexity is $\Theta(h)$ if h is the height of the search tree. If the search tree is complete, the height is $h \in \Theta(\log n)$, where n is the number of elements in the tree.

The find algorithm can be directly extended to insert elements into the tree: if the algorithm reaches an empty subtree, then the element is not stored in the tree. Thus, a node containing the object can be added at the position where the search has stopped. Consider as example where number 21 is to be inserted into the tree from Fig. 3.15. At node 22 the algorithm branches to the left subtree, but this tree is empty. Therefore, a node with the number 21 will be added as the left son of the node 22.

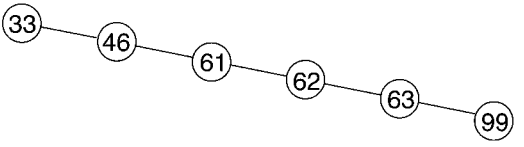


Figure 3.16: A binary search tree built by inserting numbers 33, 46, 61, 62, 63, 99 in the given order using a simple insert operation.

This simple algorithm for inserting new objects has a major drawback. Assume that numbers sorted in increasing order are inserted into a tree which is empty at the beginning. During the execution of the algorithm the right subtree is always taken. Consequently, the tree which is built in this way is in fact a list (see Fig. 3.16). In this case the height of the tree is equal to the number of elements, i.e. the search for an object takes $\Theta(n)$ time instead of logarithmic time.

In order to avoid this trouble the tree can always be kept *balanced*. This means that for every node the heights of the left and right subtree differ at most by one. There are several clever algorithms which assure that a search tree remains balanced. The main idea is to reorganize the tree in case of imbalance while keeping the sorted order. More details can be found in [6, 4]. For balanced trees, it is guaranteed that the height of the tree grows only logarithmically with the number of nodes n. Thus, all operations like insert, search and delete can be realized in $\Theta(\log n)$.

3.6 NP-Complete Graph Problems

The time complexities of the algorithms presented in the last chapter are all bounded by a polynomial. But there are many hard graph problems as well. Interestingly, there are many pairs of similar problems where one problem is in P while the other in NP. Some of them will be presented now.
The algorithms presented in the last chapter are very simple. Usually the algorithms for solving graph problems, even for the problems in P, are very complicated. Thus, in this section no algorithms will be presented, just some basic graph problems are explained. For the proofs that certain problems are NP-complete the reader is referred to the literature as well. But later on, for the optimization problems we encounter here, all necessary algorithms will be discussed in detail.

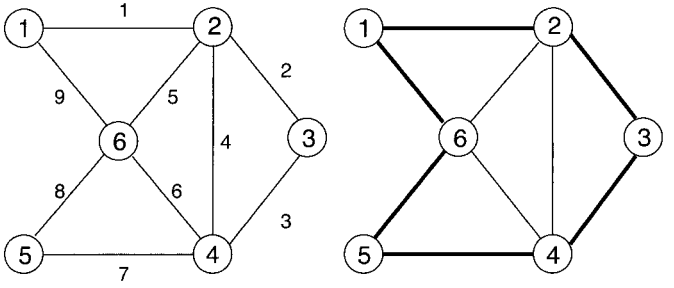


Figure 3.17: An undirected graph containing a Euler cycle (left) and a Hamilton cycle (right). The numbers next to the edges give one possible order of the edges within an Euler cycle.

The first example is about cycles. Let $G = (V, E)$ be an undirected graph. An *Euler cycle* is a cycle which contains all *edges* of the graph exactly once. It was shown by Euler in 1736 that an undirected graph contains an Euler cycle if and only if each vertex has an even degree. Thus, the recognition problem, whether an undirected graph has an Euler cycle or not (EC), can be decided in polynomial time. The first algorithm for constructing an Euler cycle was presented 1873 in [7]. A *Hamilton cycle* is a cycle which contains each *vertex* exactly once. The problem of whether an undirected graph contains a Hamilton cycle or not (HC) is NP-complete. A proof can be found in [8], the main idea is to show 3-SAT$\leq_p$HC. In Fig. 3.17 a graph containing both an Euler Cycle and a Hamilton cycle is shown.
The next pair of graph problems is defined via *covers* of undirected graphs. An *edge cover* is a subset $E' \subset E$ of edges such that each vertex is contained in at least one edge $e \in E'$. Each graph which has no isolated vertices has an edge cover, since in that case E itself is an edge cover. A *minimum edge cover* is an edge cover where $|E'|$ is minimal. In Fig. 3.18 a graph and a minimum edge cover are shown. An algorithm which constructs a minimum edge cover in polynomial time can be found in [9]. Consequently, the corresponding recognition problem "does graph G have an

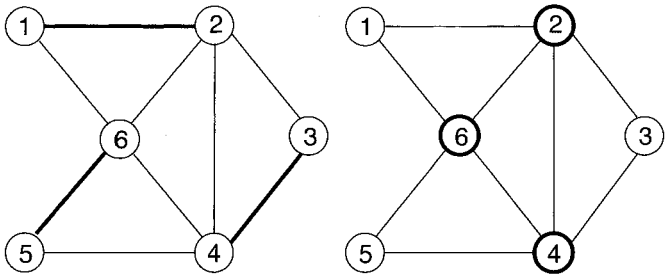


Figure 3.18: A graph and a minimum edge cover (left) and a minimum vertex cover (right). The edges/vertices belonging to the cover are shown in bold.

edge cover with $|E'| \leq K$" is in P. Related to the edge cover is the *vertex cover*. It is a subset $V' \subset V$ of vertices such that each edge contains at least one node from V'. In Fig. 3.18 a graph together with its minimum vertex cover is displayed. It has been shown in [10] that the problem "does graph G have a vertex cover with $|V'| \leq K$" (VC) belongs to the class of NP-complete problems. The proof works by transforming 3-SAT$\leq_p$VC. Thus, there is no algorithm known which finds a minimum vertex cover in polynomial time.

We have already seen that the TSP is a hard optimization problem. The corresponding decision problem is NP-complete [10]. The proof works by demonstrating HC$\leq_p$TSP. HC has already been recognized as being NP-complete (see above). The proof is short, which is quite unusual for this type of problem.

Proof: Let $G = (V, E)$ be a graph, $n = |V|$. Then we built a second graph $G' = (V, V \times V)$ with the following distance labeling

$$d(i,j) = \begin{cases} 1 & \text{if} \quad (i,j) \in E \\ 2 & \text{else} \end{cases} \tag{3.2}$$

The transformation works in polynomial time since it contains just two nested loops over all nodes. Each tour in G' contains exactly n edges and visits all n nodes. Thus, a tour must have the length n at least. If it has exactly length n all distances along the tour must have value 1, i.e. all edges from the tour are in G as well, and vice versa. Consequently, the question "does G' have a shortest round tour with length $\leq n$" has the answer "yes" if and only if G has a Hamilton cycle. Hence, HC$\leq_p$TSP. QED

On the other hand, if one is interested only in the shortest distance between two given cities, this problem can be solved in polynomial time. Several algorithms can be found in [11]. Some of them will be presented in Chap. 4. A recent application in physics is the study of so called *directed polymers*.

Another type of graph problems, which can be solved in polynomial time, are matching problems. They will be introduced in Chap. 10.

Bibliography

[1] B. Bolobas, *Modern Graph Theory*, (Springer, New York 1998)

[2] J.D. Claiborne, *Mathematical Preliminaries for Computer Networking*, (Wiley, New York 1990)

[3] M.N.S. Swamy and K. Thulasiraman, *Graphs, Networks and Algorithms*, (Wiley, New York 1991)

[4] R. Sedgewick, *Algorithms in C*, (Addison-Wesley, Reading (MA) 1990)

[5] K. Mehlhorn and St. Näher, *The LEDA Platform of Combinatorial and Geometric Computing*, (Cambridge University Press, Cambridge 1999);
see also http://www.mpi-sb.mpg.de/LEDA/leda.html

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

[7] C. Hierholzer, *Math. Ann.* **6**, 30 (1873)

[8] C.H. Papadimitriou and K. Steiglitz, *Combinatorial Optimization*, (Dover Publications, Mineola (NY) 1998)

[9] E.L. Lawler, *Combinatorial Optimization: Networks and Matroids*, (Holt, Rinehard and Winston, New York 1976)

[10] M.R. Garey and D.S. Johnson, *Computer and Intractability, A Guide to the Theory of NP-Completeness*, (W.H. Freeman & Company, San Francisco 1979)

[11] D.P. Bertsekas, *Network Optimization, Continuous and Discrete Models*, (Athena Scientific, Belmont (MA) 1998)

4 Simple Graph Algorithms

Consider an amorphous conductor which is composed of individual micro grains that are either conducting or insulating, both randomly with a probability p and $1-p$, respectively. A current can flow only between two neighboring occupied sites, which we then denote as being connected. Obviously an electrical current can cross the macroscopic sample if and only if it has a connected cluster extending from one boundary to the opposite one. This physical situation can easily be represented by a graph by identifying the conducting grains by nodes and the connections between neighboring conducting grains by edges. The question of whether the whole sample is conducting is then equivalent to the search for a connected path from one subset of the nodes (one boundary) to another subset (the opposite boundary). In this section we will discuss therefore simple graph algorithms that answer the question, whether there is a connected path between two given nodes i and j of the graph. In addition to this they allow the determination of how many (and how large) connected components or clusters a given graph has and what are their statistical properties (fractal dimension etc) if the graph is random.
Moreover, the physical situation could be complicated by the fact that not all conducting grains have the same conductance. If the concentration of conducting grains is large enough to have many connecting paths from one end of the sample to the opposite end or if simply all grains are conducting, but some of them are better and some worse in a random pattern, one also wants to know the paths of the lowest resistance. This situation can again be represented by graph with non-negative weights or costs c_{ij} assigned to each edge representing the resistance. In this section we will present simple graph algorithms that determine the path $\mathcal{L}$ with minimum total cost $E = \sum_{(ij)\in\mathcal{L}} c_{ij}$ between two nodes i and j, which is the so-called shortest path problem. In addition, for later applications, we will present a way to detect cycles (closed paths) that have negative total costs for the case where some of the weights given to the edges are negative.

4.1 The Connectivity-percolation Problem

An idealized model for such an amorphous conductor as described above would be a regular lattice (e.g. a square lattice in 2d or a simple cubic lattice in 3d) in which sites are removed with a probability $1-p$. To be precise, in the random graph representing the diluted lattice with each removed node one also removes all edges adjacent to

it. At low concentrations one does not expect any *percolating* clusters, i.e. connected clusters that extend from one boundary to the opposite one. On the other hand, at high concentration most probably there will be one, in other words: at low p the sample will be (with probability one) insulating, at large p it will be a conducting, see Fig. 4.1. As it turns out, there is a critical value p_c above which almost all (i.e. with probability 1) samples have a percolating cluster (or "infinite cluster") in the thermodynamic limit, and below which they do not. The precise value of p_c depends on the lattice structure (i.e. the class of random graphs) one considers, but otherwise many features of this transition are identical for different lattices in two dimensions, but different in 3, 4, etc. dimensions.

Figure 4.1: Example for site percolation on the square lattice ($L = 50$) with site occupation probability $p = 0.3$ (left), $p = 0.5$ (middle), $p = 0.7$ (right). The percolation threshold is known to be at $p_c \approx 0.59275$ [1]. Therefore the right picture is above the percolation threshold and indeed one recognizes easily a percolating cluster extending from the bottom to the top of the system.

For instance exactly at the critical point $p = p_c$ at least one infinite cluster exists and it is a fractal, i.e. its volume V_c (defined as the number of sites it contains) scales with the linear system size as $V_c \sim L^{d_f}$. The value of this fractal dimension is one of the aforementioned features that do not depend on the lattice but only on the dimension in which one studies percolation. The number d_f, called a critical exponent is *universal* (for instance $d_f = 91/48$ for 2d-percolation [2], and $d_f \approx 2.524$ for 3d percolation [3]), as other exponents describing the percolation transition: The infinite-cluster probability (which is the probability that a site is on the infinite cluster) close to the transition behaves like $P_\infty \sim (p - p_c)^\beta$ ($\beta = 5/36$ in 2d [2], $\beta \approx 0.417$ in 3d [3]); the correlation length $\xi \sim |p - p_c|^{-\nu}$ ($\nu = 4/3$ in 2d [2], $\nu \approx 0.875$ in 3d [3]) and the average cluster size $S \sim |p - p_c|^{-\gamma}$ ($\gamma = 43/18$ in 2d [2], $\gamma \approx 1.795$ in 3d [3]). In Sec. 5.3 we will show how these exponents are computed using the methods presented in the next section.

4.1.1 Hoshen-Kopelman Algorithm

There are a large number of efficient algorithms for connectivity percolation, namely: the Leath algorithm [4]; the Hoshen-Kopelman algorithm [5]; the forest-fire and burning algorithms [6]; and the invasion algorithm [7]. Because of its widespread use

in random lattice problems in physics we will discuss the Hoshen-Kopelman algorithm first.

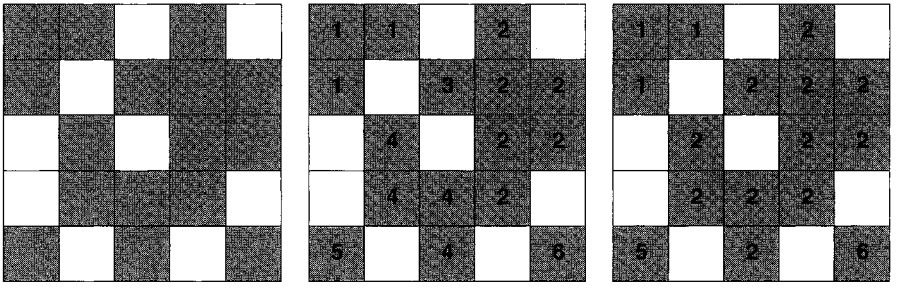


Figure 4.2: Example for the Hoshen-Kopelman algorithm. One starts in the upper left corner and proceeds row by row. In the first row the first site is occupied and gets label 1, the second gets the same label since it is a neighbor of the first, the third is empty and the fourth gets label 2. In the second line the first site has an occupied upper neighbor with label 1, thus it gets also label 1. The second is empty and the third is labeled 3. The fourth site is now a neighbor of two sites, one labeled 2 and one labeled 3. All three sites belong to the same cluster, which was first labeled 2. Accordingly we also assign the label 2 to the new site, but we have to keep track that cluster 2 and 3 are connected. This is achieved by setting the array $size(3)$=-2, saying that 3 is not a proper label, since $size(3)$ is negative, and that the proper label is 2. After distributing labels to all sites one changes them into the proper ones.

The Hoshen-Kopelman algorithm [5] grows many clusters simultaneously by assigning cluster labels to each new cluster that is nucleated during growth. This is done "row-by-row" (or "layer-by-layer" in three dimensions) and must take into account the merging of different growing clusters. This merging is accounted for by defining "equivalence classes" which are sets of cluster labels that are set to be equivalent due to merging. Simulations on lattices of up to 4×10^{11} sites have been carried out using this method [8].
We list here a meta-code for 2d (the generalization to higher dimension or other lattices is tedious but straightforward) since the Hoshen-Kopelman algorithm allows cluster identification in many disordered systems, where a cluster structure appears, not only for percolation problems. The function that checks for occupied left and upper neighbor-sites is $neigh(x, y) = (s(x-1, y), s(x, y-1))$, where $s(x, y) = 1$ if the site (x, y) is occupied and $s(x, y) = 0$ if empty. The global variable c carries the label of the current cluster.

```
algorithm Hoshen-Kopelman [2d square lattice]
begin
   c := 1;
   for y := 1,...,L_y do
      do x := 1,...,L_x do
      begin
         if neigh(x,y) = (0,0) then start-new(x,y);
         if neigh(x,y) = (1,0) then connect(x,y,x-1,y);
         if neigh(x,y) = (0,1) then connect(x,y,x,y-1);
         if neigh(x,y) = (1,1) then
            if label(x-1,y) = label(x,y-1) then
               connect(x,y,x-1,y);
            else
               merge-clusters(x,y);
      end
   for y := 1,...,L_y do
      for x := 1,...,L_x do
         while size(label(x,y))<0 do
            label(x,y) := -size(label(x,y));
end


procedure start-new (x,y)
begin
   label (x,y) := c;
   size(c) := 1;
   c := c + 1;
end


procedure connect (x_1,y_1, x_2,y_2)
begin
   label(x_1,y_1) := label(x_2,y_2);
   size(c) := size(c)+1;
end


procedure merge-clusters (x,y)
begin
   c_1 := min(label(x-1,y), label(x,y-1));
   c_2 := max(label(x-1,y), label(x,y-1));
   label(x,y) := c_1;
   size(c_1) := size(c_1) + size(c_2) + 1;
   size(c_2) := -c_1;
end
```

In Fig. 4.2 we show an example of the cluster labeling at work. For the implementation it might be advantageous to replace the cluster labels by the proper one as soon as neighbor sites are checked. In Sec. 5.3 we show how the results generated by an application of this algorithm are analyzed with finite-size scaling to estimate the critical exponents for percolation mentioned in the last section.

4.1.2 Other Algorithms for Connectivity Percolation

(i) *Search algorithms at fixed p*
The Hoshen-Kopelman algorithm discussed in the previous section is a special search algorithm particularly suitable for diluted grid graphs. In general one considers a connected graph $G(X, E)$ which is then diluted (i.e. edges are removed with probability $1-p$) and search algorithms are then used to study the cluster structure of the diluted graph. A breadth-first search (see below) from a source site s finds all sites which are connected to the source and labels them with their "chemical distance" [1]. This is equivalent to the "forest fire" or "burning" method. These methods were developed independently of, and quite a bit later than, the corresponding developments in the computer-science community. A new breadth-first-search is needed for each cluster, so iterative application of breadth-first search identifies the cluster structure.
(ii) *Growth algorithms at fixed p*
It is storage inefficient to generate the entire graph and then to classify its cluster structure. Instead *growth* algorithms keep track of information only at a growth or "invasion" front (these methods are also called "epidemic" methods). As discussed in the next section, breadth-first search, Dijkstra's algorithm and Prim's algorithm are also best implemented in this way. The *Leath* algorithm [4] for site percolation grows from a seed site. Each site adjacent the growth front is assigned a uniform random number r_j. At each step, choose a site j which has not yet been tested and which is a nearest neighbor to a site which has already been invaded. If $r_j \leq p$ then add site j to the growing cluster. If $r_j > p$, mark site j as tested and do not test it again. For $p > p_c$, the Leath algorithm grows a cluster of infinite size, while for $p < p_c$, the growth of the cluster ceases at a finite distance from the source. Very high precision results for percolation have been found using this method, for example, lattices of size 2048^3 have been studied recently [9].

4.1.3 General Search Algorithms

The Hoshen-Kopelman algorithm discussed in the last section is a special search algorithm adapted to grid graphs or lattices occuring frequently in physics. In this section we will discuss more general search algorithms that help to identify connected components of a graph with any topology.
Consider a connected graph $G(X, E)$ containing a vertex set X and edge set E. A *connected graph* has sufficient edges such that a connected path exists between any two nodes i and j. We input the graph connectivity, i.e. a list of edges which are present. There are two basic search strategies on $G(X, E)$: breadth-first search and depth-first search, which are closely related.

The main work of the first algorithm is done within the procedure depth-first(), which starts at a given vertex i and visits all vertices which are connected to i. The main idea is to perform recursive calls of depth-first() for all neighbors of i which have not been visited at that moment. The array *comp*[] is used to keep track of the process. If $comp[i] = 0$ vertex i has not been visited yet. Otherwise it contains the number t of the component. Please note that the array *comp*[] is passed as reference, i.e. it behaves like a global variable.

```
procedure depth-first(G, i, comp, t)
begin
   comp[i]:=t;
   for all neighbors j of i do
      if comp[j] = 0 then
         depth-first(G, j, comp, t);
end


algorithm components(G)
begin
   initialize comp[i] := 0 for all i ∈ V;
   t := 1;
   while there is a node i with comp[i]=0 do
      depth-first(G, i, comp, t); t := t + 1;
end
```

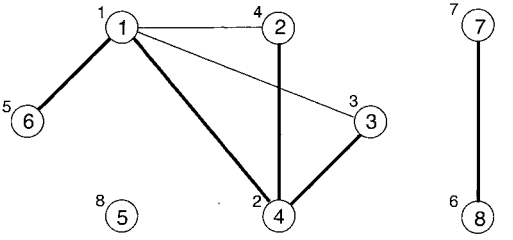


Figure 4.3: Depth-first spanning forest for a graph with three connected components.

In Fig. 4.3 an example graph is shown. The numbers next to the nodes indicate a possible order in which the nodes are visited. The edges which are used by the algorithm are indicated by thick lines. Since each node is visited not more than once, there are no cycles in the subgraph given by these thick edges. Thus, they constitute a tree for each component. Since all nodes are visited, the tree *spans* the whole component. For this reason the tree is called a *spanning tree*. The collection of all spanning trees of a graph is called its spanning *forest*.

Please note that the algorithm tries to go as far as possible within a graph, before visiting other neighbors of the node where the search started. It is for this reason that the procedure has its name. The spanning trees constructed by this procedure are also called *depth-first spanning trees*.
For readers more interested in the subject, we should note that a depth-first search uses *singly connected edges* in its first probe to maximum depth. These are edges, which are not contained in loops, which include the source and transverse each edge only once.
Since depth-first search excludes crossing an edge more than once, the node at the end of a singly connected edge (or sequence of singly connected edges) which is closest to the source of the search, defines an *articulation point* in a *depth-first tree*. The articulation points divide the depth-first-search tree into its *biconnected components*.

This procedure was well known in computer science in the early 1970s [10], and has recently provided efficient algorithms for the *backbone* in connectivity percolation [11]. The backbone is a subset of the infinite cluster, and is found from the infinite cluster by trimming off "dangling ends" which are unable to transport an electric current. The original burning algorithm for backbone identification uses forward and reverse breadth-first searches to find the backbone [6]. This is inefficient compared with the depth-first search procedure, though high accuracy results have been found by applying this method at p_c [12].
A similar algorithm, which instead first visits all neighbors of a node before proceeding with nodes further away, is called *breadth-first search* (BFS). This means at first all neighbors of the source are visited, they have distance one from the source. In the previous example in Fig. 4.3, the edge $(1, 2)$ would be included in the spanning tree, if it was constructed using a breadth-first search. In the next step of the algorithm, all neighbors of the vertices treated in the first step are visited, and so on. Thus, a queue (see Sec. 3.2) can be used to store the vertices which are to be processed. The neighbors of the current vertex are always stored at the end of the queue. Initially the queue contains only the source. The algorithmic representation reads as follows, $level(i)$ denotes the distance of vertex i from the source s and $pred(i)$ is the predecessor of i in a shortest path, which is obtained as a byproduct (see next page):

```
algorithm breadth-first search()
begin
   Initialize queue Q := {s};
   Initialize level(s) := 0; level(i) := −1 (undefined) for all other vertices;
   Initialize pred(0) := −1;
   while Q not empty
   begin
      Remove first vertex i of Q;
      for all neighbors j of i do
         if level(j) = −1 then
         begin
            set level(j) := level(i) + 1;
            set pred(j) := i;
            add j at the end of Q;
         end
   end
end
```

During a run of the algorithm, for each vertex all neighbors are visited. Thus each edge is touched twice, which results in a time complexity of $\Theta(|E|)$.

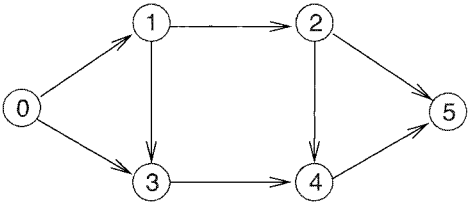


Figure 4.4: Example graph for breadth-first search. The search starts at vertex 0. In the first iteration, vertices 1 and 3 are visited. In the next iteration, vertices 2 and 4 are treated, finally vertex 5.

Example: Breadth-first search

We consider the graph shown in Fig. 4.4. Initially the queue contains the source and all values $level(i)$ are undefined, except $level(0) = 0$.

$Q = \{0\}$, $level(0) = 0$

While treating the source, its neighbors, vertices 1 and 3, are added to the queue, thus $pred(1) = pred(3) = 1$. They have distance 1 from the source

($level(1) = level(3) = 1$). The order the vertices are added has no influences on the final results, only on the details of the computation.

$Q = \{1, 3\}$, $level(0) = 0$, $level(1) = 1$, $level(3) = 1$

Next vertex 1 is processed. It has two neighbors, vertices 2 and 3, but vertex 3 has been visited already ($level(3) \neq -1$), thus only vertex 2 is added to Q with $pred(2) = 1$, $level(2) = level(1) + 1 = 2$. After this iteration the situation is as follows:

$Q = \{3, 2\}$, $level(0) = 1$, $level(1) = 1$, $level(2) = 2$, $level(3) = 1$

The treatment of vertex 3 adds vertex 4 to the queue ($level(4) = level(1) + 1 = 2$, $pred(4) = 3$). At the beginning of the next iteration vertex 2 is taken. Among its two neighbors, only the sink has not been visited. Thus $level(5) = level(2) + 1 = 3$ and $pred(5) = 2$. Now all values of the *pred* and *level* arrays are set. Finally, vertices 4 and 5 are processed, without any change in the variables.

A shortest path from vertex 5 to vertex 0 is given by the iterated sequence of predecessors of the sink: $pred(5) = 2, pred(2) = 1, pred(1) = 0$. □

4.2 Shortest-path Algorithms

In the last section we were concerned with algorithms answering the question whether a path in a given graph or network from node i to node j exists. In many applications in daily use we know that there are various paths of this sort but they have different *length* and we want to know the *shortest* of them. The breadth-first algorithm presented in the last section is able to calculate shortest paths for the case where each edge is assumed to represent a distance = 1. In this section, we will discuss methods which allow the assignment of different arbitrary (sometimes even negative) distances to the edges.

In physics distances between nodes have, typically, the meaning of resistance or potential energy or other physical parameters: think for instance of a random resistor network as described in the last paragraph made up not of conducting and insulating edges (usually called *bonds* in this context), but of bonds of varying resistance. Interpreting the resistance between two nodes as distances, the shortest path from one side to the other side of the sample is the one that carries the most current. Similarly, if you think of a random static network where bonds between nodes have different strength, again interpreted as distances, the shortest path from one side of the material to the other is the one where fracture will occur under sufficiently strong strain. Finally, the polymer in a random medium is essentially an elastic thread (a one-dimensional object -- a line) lying in a rough landscape and searching for its lowest potential and elastic energy minimum. Again, this problem can be mapped onto a network with the random potential energies as distances and the shortest path is the ground state

configuration of the polymer. In what follows we will present the algorithms with which these problems can be solved efficiently.

4.2.1 The Directed Polymer in a Random Medium

The directed polymer in a random medium (DPRM) [13] is a directed optimal path on the links of a lattice (see Fig. 4.5). For a path along the {10} orientation, the path is allowed to step forward, to the left or to the right, with an increased energy cost for motion to the left or right, which models the elasticity of the DPRM [14, 15]. An even simpler model is a path in the {11} orientation (see Fig. 4.5). In this case, there is no explicit elasticity to the DPRM, but the motion is restricted to the transverse direction, and it is believed that this constraint is sufficient to maintain the DPRM universality class. The random potentials are modeled by a random energy on the bonds of the lattice, see Fig. 4.5.

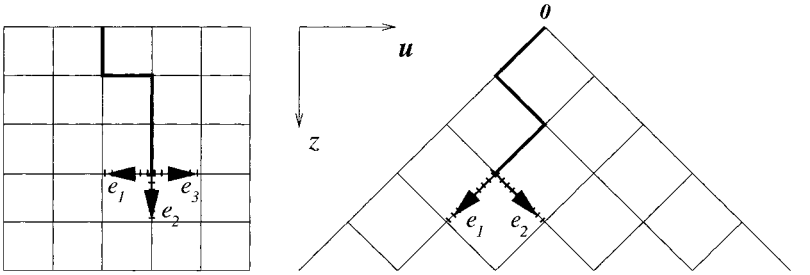


Figure 4.5: Models for a DPRM. Left: In the {10} orientation. Right: In the {11} orientation.

Thus the lattice Hamiltonian is simply

$$H = \sum_{(ij)} e_{ij} \cdot x_{ij} \tag{4.1}$$

where the sum is over all bonds (ij) of the lattice and x_{ij} represents the DPRM configuration starting at one particular point s of the lattice and ending at another point t (see Fig. 4.5). It is $x_{ij} = 1$ if the DPRM passes through the bond (ij) and $x_{ij} = 0$ otherwise. Typically s is on one side of a lattice of linear size L and t on the opposite and the ground state of (4.1) is the minimum energy path from s to t.
Interpreting the energies as distances (after making them all positive by adding a sufficiently large constant to all energies), and the lattice as a directed graph, this becomes a shortest path problem that can be solved by using, for instance, Dijkstra's algorithm, which will be described below. In addition, owing to the directed structure of the lattice one can compute the minimum energies of DP configurations ending at (or shortest paths leading to) target nodes t recursively (this is the way in which Dijkstra's algorithm would proceed for this particular case) [16]. This is the same as the transfer-matrix algorithm, encountered in statistical mechanics [17]. It reduces in

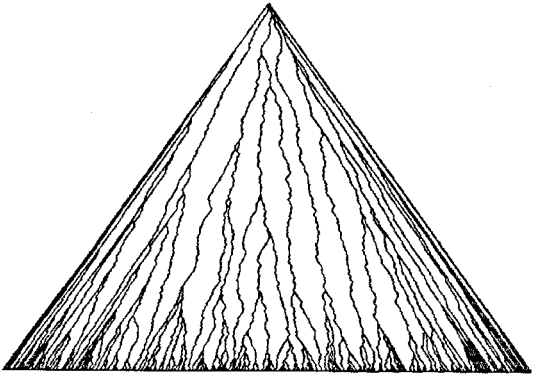


Figure 4.6: A collection of polymers of lowest energy directed along the diagonals of a square lattice with random bonds. Each polymer (crossing 500 bonds) has one end fixed to the apex of the triangle, the other to various points on its base, and finds the optimal path in between.

the zero temperature limit to a simple recursion relation for the energies or distances from s to nodes t in the $(n+1)$th layer, once we know the shortest paths to the n-th layer:

$$E_s^{(n+1)} = Min\{E_{s'}^{(n)} + e_{ss'} | s' \in \ n'\text{th layer } \textit{and } s' \text{ nearest neighbor of } s\} \quad (4.2)$$

In Fig. 4.6 we show a collection of such optimal paths in the $1+1$ dimensional case. In the following we will discuss the two basic algorithms to find the shortest paths in a general graph.

4.2.2 Dijkstra's Algorithm

Given a set of *costs* c_{ij} on each edge of a graph, we calculate the *distance label* $d(i)$, which is the *cost* of a *minimum-cost path* [18, 19, 20, 21] from a starting node s to the node i. *Dijkstra's algorithm*, which works for non-negative costs, is a so called *label-setting* algorithm to solve the shortest-path problem. It is a label-setting algorithm because it finds the exact distance label correctly at the first attempt. In contrast, *label-correcting* algorithms, which be will presented as well, iteratively approach the exact distance label, and work even if some of the costs are negative, provided there are no negative-cost cycles in the graph. In order to reconstruct the set of bonds which make up the minimal-cost path from site s to site i, the algorithms also store a *predecessor label*, *pred(i)*, which stores the label of the previous site from which the minimal path reached i. The set of minimal paths from a starting site s to all of the other sites in the graph forms a *spanning tree*, T_P. An example is presented in Fig. 4.7.

Both label-setting and label-correcting algorithms use the key properties which shortest paths obey:

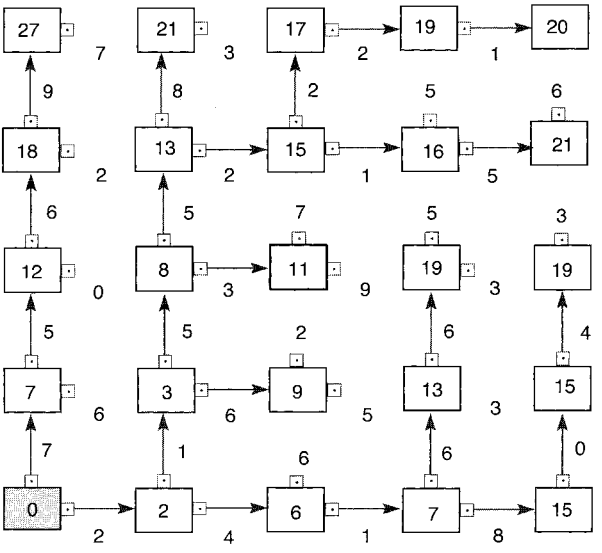


Figure 4.7: The tree of minimal paths from the source node (shaded) to all other nodes in a directed square lattice (the edges of the graph only allow paths which are in the positive {01} and positive {10} directions). All bonds between nearest neighbors are labeled with their costs, but only the tree of minimal paths is shown. Each node is labeled with the cost of a minimal path from the source to that node. (Generated using the demonstration programs from the LEDA library available at http://www.mpi-sb.mpg.de/LEDA/)

(i) For each edge belonging to the shortest-path tree from node s:

$$d(j) = d(i) + c_{ij} \qquad (i,j) \text{ in } T_P.$$

(ii) For each edge which *does not belong* to a shortest path:

$$d(j) \leq d(i) + c_{ij} \qquad (i,j) \text{ not in } T_P.$$

These properties are often discussed in terms of *reduced costs*, defined as,

$$c_{ij}^d = c_{ij} + d(i) - d(j). \tag{4.3}$$

Properties (i) and (ii) above are then,

$$c_{ij}^d = 0, \quad \text{if} \quad (i,j) \in T_P, \tag{4.4}$$

and

$$c_{ij}^d \geq 0, \quad \text{otherwise.} \tag{4.5}$$

The proof of these properties relies on the spanning-tree structure of the set of minimal paths, namely that each site of the tree has *only one predecessor*. Thus if there is a

bond with $c_{ij}^d < 0$ which is not on the minimal-path tree, adding that bond to the tree and removing the current predecessor bond for site j [which from condition (4.4) has zero reduced cost], leads to a reduction in the cost of the minimal-path tree. Thus any tree for which there exists a bond with $c_{ij}^d < 0$, is *not* a minimal-path tree.
For later reference we also note that a directed cycle, W, has the property,

$$\sum_{(i,j)\in W} c_{ij}^d = \sum_{(i,j)\in W} c_{ij} \tag{4.6}$$

which follows from Eqs. (4.3-4.5).
We will now discuss Dijkstra's method for the minimum path, which works by *growing outward* from the starting node s in a manner very similar to breadth-first search. At each step Dijkstra's algorithm chooses to advance its growth front to the next unlabeled site which has the *smallest* distance from the starting node.

algorithm Dijkstra($G, \{c_{ij}\}$)
begin
  $S := \{s\}$; $\overline{S} := X \backslash \{s\}$;
  $d(s) := 0$, $pred(s) := 0$;
  while $|S| < |X|$ **do**
  begin
    choose $(i,j) : d(j) := \min_{k,m}\{d(k) + c_{km} | k \in S, m \in \overline{S}, (k,m) \in E\}$;
    $\overline{S} := \overline{S} \backslash \{j\}$; $S := S \cup \{j\}$;
    $pred(j) := i$;
  end
end

The algorithm maintains the minimal distance *growth front* by adding the node $j \in \overline{S}$ with *minimal* distance label $d(j)$. The proof that $d(j)$ generated in this way is actually a minimal-cost label proceeds as follows:
(i) Assume that we have a growth front consisting of sites which are labeled with their minimal path lengths to the source s.
(ii) The next candidate for growth is chosen to be a site which is not already labeled, and which is connected to the growth front by an edge $(i,j) \in E$.
(iii) We choose the site j for which $d(j)$ is minimal $d(j) = \min_{k,m}\{d(k) + c_{km}, k \in S, m \in \overline{S}, (k,m) \in E\}$.
(iv) Because of (iii) *and* because all of the costs are non-negative there can be no path from the current growth front to site j which has a smaller distance than $d(j)$. This is because any such path must originate at the current growth front and hence must use a non-optimal path to generate any alternative path to j (*negative costs* can compensate for locally non-optimal paths from the growth front and hence Dijkstra's method is restricted to non-negative costs). In Fig. 4.8 an example demonstrates how the Dijkstra's algorithm works.
The "generic" Dijkstra's algorithm scales as $\mathcal{O}(|X|^2)$, if the *choose* statement in the above algorithm requires a search over all the sites in the lattice. It is easy to do much

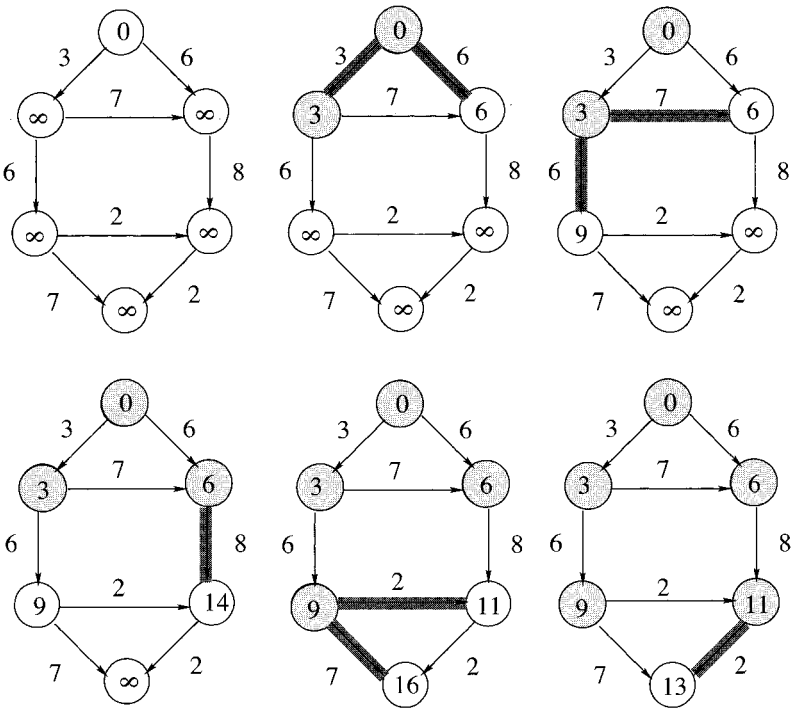


Figure 4.8: Demonstration of Dijkstra's algorithm. The number in the circles denote the distance labels $d(i)$. White circle stand for temporary nodes, filled circles for permanently labeled nodes. The numbers on the edges are the distances c_{ij}. The edges that have been considered in the last step are marked.

better than this by maintaining a list of active sites at the growth front (as in breadth-first search). However now we must choose the *lowest-cost site* from among this list. Thus the potential growth sites must be ordered according to their distance label. This ordering must be reshuffled every time a new growth site (with a new distance label) is added to the list. In the computer science community this is typically done with *heaps* or *priority-queues* which consist of a tree-like data structure, see Sec. 3.2. Heap reshuffling is $\mathcal{O}(ln|E|)$ which reduces the algorithmic bound to $\mathcal{O}(|E|ln|E|)$. Each element of the heap has a key, which is here the temporary distance. The heap operations create-heap(), find-min() and delete-min() are self-explanatory. The decrease-key() operation assigns a new lower temporary distance to an element of the heap and moves it eventually towards the root of the heap by exchanging elements. By $A(i)$, we denote the edges adjacent to vertex i. The heap implementation of the Dijkstra algorithm reads as follows:

```
algorithm heap-dijkstra(G, {c_ij})
begin
    create-heap(H);
    d(i) := ∞ for each node i ∈ N;
    d(s) := 0 and pred(s) := 0;
    insert(s,H);
    while H ≠ ∅ do
    begin
        find-min(i, H);
        delete-min(i, H);
        for each (i, j) ∈ A(i) do
        begin
            value := d(i) + c_ij;
            if d(j) > value then
            begin
                if d(j) = ∞ then
                    insert(j, H);
                else
                    decrease-key(value, i, H);
                d(j) := value, pred(j) := i
            end
        end
    end
end
```

If the bond costs are integers, the site distance labels themselves can be used as pointers. Thus we can set up a queue with the distance label as pointers and the site labels with that distance in the queue (or, in computer science terminology, we use *buckets*). The number of buckets that is required, n_b, is $n_b > 2C$ where $C = \max_{(i,j)}\{c_{ij}\}$ is the maximum cost. For example if the costs are chosen from the set $1, 2, \ldots, 10$, then $C = 10$. As long as C is finite, and the graph is sparse, buckets are very efficient both in speed and storage, for details see Ref. [22]. Using buckets, one can implemented a Dijkstra's algorithm for integer costs which has a running time scaling as $\mathcal{O}(|E|)$.

4.2.3 Label-correcting Algorithm

As we said, Dijkstra's algorithm is a label-setting algorithm. The above mentioned criterion

$$d(\cdot)\text{shortest-path distances} \Leftrightarrow d(j) \leq d(i) + c_{ij} \; \forall \, (i,j) \in A \tag{4.7}$$

also gives rise to a so called *label-correcting* algorithm.
Let us define reduced edge length (or *reduced costs*) costs!reduced via

$$c_{ij}^{d} := c_{ij} + d(i) - d(j). \tag{4.8}$$

As long as one reduced edge length is negative, the distance labels $d(i)$ are not shortest-path distances:

$$d(\cdot) \text{ shortest path distances} \quad \Leftrightarrow \quad c_{ij}^d \geq 0 \qquad \forall (i,j) \in A \tag{4.9}$$

The criterion (4.9) suggests the following algorithm for the shortest-path problem:

algorithm label-correcting
begin
  $d(s) := 0$; $pred(s) := 0$;
  $d(j) := \infty$ for each node $j \in N \backslash \{s\}$;
  while some edge (i,j) satisfies $d(j) > d(i) + c_{ij}$ $(c_{ij}^d < 0)$ **do**
  begin
    $d(j) := d(i) + c_{ij}$ $(\Rightarrow c_{ij}^d = 0)$;
    $pred(j) := i$;
  end
end

Initially the distance labels at each site are set to a very large number [except the reference site s which has distance label $d(s) = 0$]. This method requires that the starting distance labels $d(j)$ are greater than the exact shortest distances and the choice of $d(j) = \infty$ ensures that. In Fig. 4.9 an example of the operation of the algorithm is shown.
In practice it is efficient to grow outward from the starting site s. The algorithm may sweep the lattice many times until the correct distance labels are identified. The worst case bound on running time $\mathcal{O}(\min\{|X|^2|E|C, |E|2^{|X|}\})$ with $C = \max\{|c_{ij}|\}$, which is pseudo-polynomial. An alternative procedure is to sweep the lattice once to establish approximate distance labels and then to iterate locally until local convergence is found. This so called *first in, first out* (FIFO) implementation has complexity $\mathcal{O}(|X||E|)$.
Note that if there are negative cycles, the instruction $d(j) = d(i) + c_{ij}$ would decrease some distance labels *ad (negative) infinitum*. However if there *are* negative cycles, one can detect them with an appropriate modification of the label-correcting code: One can terminate if $d(k) < -nC$ for some node i (again $C = \max|c_{ij}|$) and obtain these negative cycles by tracing them through the predecessor indices starting at node i. This will be useful in the *negative-cycle-canceling method* for minimum-cost flow (Chap. 7).

4.3 Minimum Spanning Tree

In practical daily use minimum spanning tree problems generally arise in one of two ways, directly or indirectly. In some *direct* applications one wishes to connect a set of points using the least cost or least length collection of edges. Frequently the points represent physical entities such as the components of a computer chip, or users of a system who need to be connected to each other or to a central service such as a central processor in a computer system. In *indirect* applications one wishes to connect some

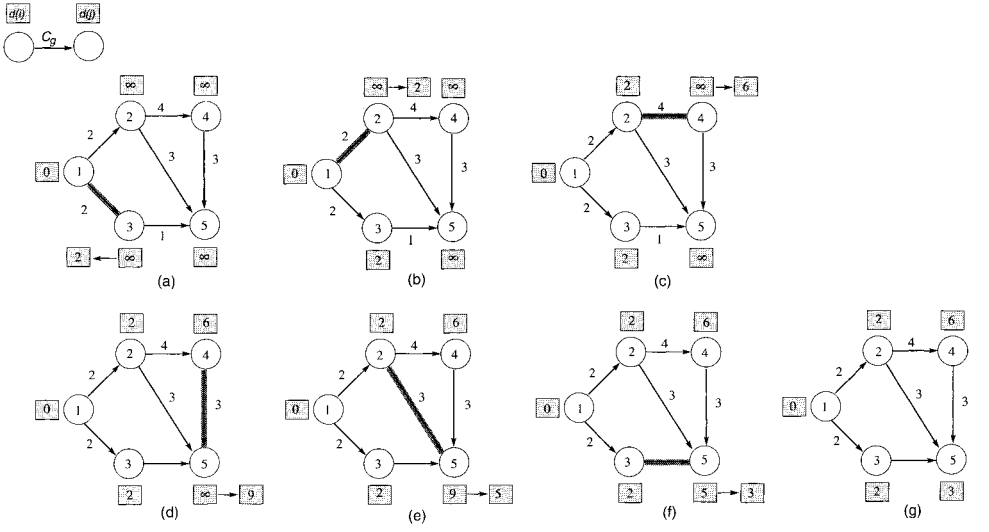


Figure 4.9: Illustration of the label-correcting algorithm. If the algorithm selects the edges (1,3), (1,2), (2,4), (4,5), (2,5) and (3,5) in this sequence, we obtain the distance labels shown in parts (b) to (g). At this point, no edge violates its optimality condition and the algorithm terminates.

set of points using a measure of performance that on the surface has little resemblance to the minimum spanning tree objective (sum of edge costs), or the problem itself bears little resemblance to an optimal tree problem.

The minimum spanning tree [18, 19] of a connected graph with edge costs c_{ij} is a tree which: (i) visits each node of the graph and; (ii) for which $\sum_{tree} c_{ij}$ a minimum. Prim's algorithm and Kruskal's algorithm are two methods for finding the minimal spanning tree [18, 23]. The two algorithms are based on the following (equivalent) optimality conditions:

Cut-optimality condition: A spanning tree T^* is a minimum spanning tree if and only if: for all tree edges $(ij) \in T^*$: c_{ij} is smaller than every capacity c_{kl} contained in the cut[1], which is induced by deleting edge (ij) from T^*, i.e. after deleting the edge from the tree, the tree falls apart in two parts. The cut, which is induced by this, contains all edges in the graph G, which run between the two parts.

Path-optimality condition: A spanning tree T^* is a minimum spanning tree if and only if: For every non-tree edge (kl) of G: $c_{ij} \leq c_{kl}$ for every edge (ij) contained in the path in T^* connecting nodes k and l.

[1] A *cut* of a graph G is a separation of the node set V of G in two disjoint subsets S and S' with $S \cup S' = V$, see also Chap. 6. Usually one also identifies all edges (i,j) with $i \in S$ and $j \in S'$ with the cut.

Kruskal's algorithm is based on the path optimality condition whereas Prim's algorithm is based on the cut optimality condition. The latter is very similar in structure to Dijkstra's algorithm. Recently it has been observed [24] that Prim's algorithm is essentially equivalent to the invasion algorithm for percolation.

In Prim's algorithm we *start by choosing the lowest cost bond in the graph.* The algorithm then uses the two sites at the ends of this minimal cost bond as the starting sites for growth. Growth is to the lowest cost bond which is adjacent to the growth front. The algorithm terminates when every site has been visited. The cost of the minimal spanning tree is stored in C_T, and the bonds making up the minimal spanning tree are stored in T.

algorithm Prim()
begin
  choose (s,r) : $c_{sr} := \min\{c_{km}|(k,m) \in E\}$;
  $S := \{s,r\}$; $\overline{S} := X \backslash \{s,r\}$;
  $T := \{(s,r)\}$; $C_T = c_{sr}$;
  while $|S| < |X|$ **do**
  begin
    choose (i,j) : $c_{ij} := \min\{c_{km}|k \in S, m \in \overline{S}, (k,m) \in E\}$;
    $\overline{S} = \overline{S} \backslash \{j\}$; $S = S \cup \{j\}$;
    $C_T = C_T + c_{ij}$; $T = T \cup \{(i,j)\}$;
  end
end

In Fig. 4.10 it is demonstrated how this algorithm works in a particular example. It is evident that Prim's algorithm is almost identical to Dijkstra's algorithm. The only difference is in the *choose* instruction, which gives the cost criterion for the extremal move at the growth front. In the minimal-path problem, one chooses the site at the growth front which has the *minimum-cost path to the source*, while in the minimal-spanning-tree problem one simply chooses the minimum-cost bond. Both of these problems lead to spanning trees, but Prim's has a lower total cost (it is less constrained). In Dijkstra's algorithm one is checking the cost of the path from the tested site all the way back to the "source" or "root" of the tree. It is thus more non-local. In case of integer costs, using the same bucket strategy as described above for Dijkstra's method, Prim's algorithm is $\mathcal{O}(|E|)$. Heap implementations of Prim's method are $\mathcal{O}(|E|ln|E|)$ as for Dijkstra's method.

The alternative method for finding the minimal spanning tree is Kruskal's algorithm, which nucleates many trees and then allows trees to merge successively into one tree by the end of the procedure. This is achieved by allowing growth sites to nucleate *off the growth front*, that is, growth occurs at the lowest cost unused bond which does not lead to a cycle, regardless of whether it is on or off the growth front. Efficient Kruskal's algorithms have similar efficiency to heap implementations of Dijkstra's method (i.e. $\mathcal{O}(|E|ln|E|)$).

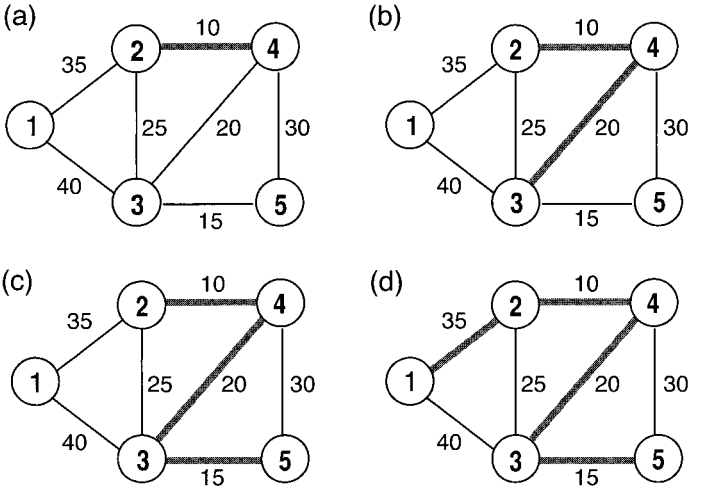


Figure 4.10: Illustration of Prim's algorithm. From (a), starting with the lowest cost edge, to (d) successively new edges are added until a minimum spanning tree is reached.

Bibliography

[1] D. Stauffer and A. Aharony, *Perkolationstheorie: eine Einfuehrung*, (Wiley-VCH, Weinheim 1995)

[2] M.P.M. den Nijs, *J. Phys. A* **12**, 1857 (1979), B. Nienhuis, *J. Phys. A* **15**, 199 (1982)

[3] P.N. Strenski, R.M. Bradley, and J.M. Debierre, *Phys. Rev. Lett.* **66**, 133 (1991)

[4] P.L. Leath, *Phys. Rev. B* **14**, 5046 (1976)

[5] J. Hoshen and R. Kopelman, *Phys. Rev. B* **14**, 3438 (1976)

[6] H.J. Herrmann, D.C. Hong, and H.E. Stanley, *J. Phys. A* **17**, L261 (1984)

[7] D. Wilkinson and J.F. Willemson, *J. Phys. A* **16**, 3365 (1983); D. Wilkinson and M. Barsony, *J. Phys. A* **17**, L129 (1984)

[8] D.C. Rapaport, *J. Stat. Phys.* **66**, 679 (1992)

[9] C. Lorenz and R. Ziff, *Phys. Rev. E* **57**, 230 (1998)

[10] R. Tarjan, *SIAM J. Comput.* **1**, 146 (1972)

[11] P. Grassberger, *J. Phys. A* **25**, 5475 (1992); *Physica A* **262**, 251 (1999)

[12] M. Rintoul and H. Nakanishi, *J. Phys. A* **25**, L945 (1992); *J. Phys. A* **27**, 5445 (1994)

[13] T. Halpin-Healy and Y.-C. Zhang, *Phys. Rep.* **254**, 215 (1995) and references therein

[14] M. Kardar, G. Parisi, and Y.-C. Zhang, *Phys. Rev. Lett.* **56**, 889 (1986)

[15] M. Kardar and Y.-C. Zhang, *Phys. Rev. Lett.* **58**, 2087 (1987); T. Nattermann and R. Lipowski,*Phys. Rev. Lett.* **61**, 2508 (1988); J. Derrida and H. Spohn, *J. Stat. Phys.* **51**, 817 (1988); G. Parisi, *J. Physique* **51**, 1695 (1990); M. Mézard, *J. Physique* **51**, 1831 (1990); D. Fisher and D. Huse,*Phys. Rev. B* **43**, 10728 (1991)

[16] M. Kardar, *Phys. Rev. Lett.* **55**, 2235 (1985); D. Huse and C. L. Henley, *Phys. Rev. Lett.* **54**, 2708 (1985); M. Kardar, *Phys. Rev. Lett.* **55**, 2923 (1985)

[17] L.E. Reichl, *A Modern Course in Statistical Physics*, (John Wiley & sons, New York 1998)

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

[19] C.H. Papadimitriou and K. Steiglitz, *Combinatorial Optimization: Algorithms and Complexity*, (Prentice Hall, New Jersy 1992)

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

[21] R. Sedgewick, *Algorithms in C*, (Addison-Wesley, Reading (MA) 1990)

[22] P.M. Duxbury and R. Dobrin, *Physica A* **270**, 263 (1999)

[23] D.B. West, *Introduction to Graph Theory*, (Prentice-Hall, New Jersy 1996)

[24] A.L. Barabasi, *Phys. Rev. Lett.* **76**, 3750 (1996)

5 Introduction to Statistical Physics

This section gives a short introduction to statistical physics, as far as it is relevant to this book. For a thorough presentation, we refer the reader to various excellent books, as for instance [1, 2, 3].

Statistical physics deals with systems that consist of many particles: typically gases, liquids, solids, radiation, polymers, membranes, proteins or complicated networks of simple (idealized) agents, like populations, metabolic networks, or the Internet. The general observation is that in a system of many degrees of freedom a collective behavior might emerge, that is *not* a feature of any of its components: the whole theory of phase transition (liquid/gas, crystallization, paramagnetic/ferromagnetic) is based on the thermodynamic limit in which the number of particles of the system is mathematically infinite. Of course there are already clear *indications* of a phase transition if the number of particle is huge but finite. In a typical experiment on a macroscopic sample for instance 10^{23} particles are studied, while $10^5 - 10^{10}$ particles can be considered in a typical state of the art computer simulation. 10^{23} is already such a large number that the experimented resolution and finiteness of the observation time can no longer account for the difference between the observed behavior and an ideal phase transition, but in computer simulation one needs a tool, namely finite-size scaling, to infer from the observed finite-size system behavior to the ideal underlying phase transition (of the infinite system).

In what follows we present a brief overview over these phenomena and the necessary tools to analyze them. Only classical systems are considered, i.e. quantum mechanical effects are neglected. First, the basic ideas are introduced. Next, a short introduction to phase transitions is given. In the third section, the ideas of finite-size scaling are explained by studying percolation problems. In the following section, the ideas are applied to magnetic phase transitions. In the last section, systems with quenched disorder are considered.

5.1 Basics of Statistical Physics

Let us consider a physical system of N particles or entities that can take on a discrete set of states $S_i, 1 = 1, \ldots, N$. An archetypical example is the *Ising model* for magnetism in which $S_i \in \{+1, -1\}$ represents an atomic magnetic moment called *spin*, that can point upwards ($S_i = +1$) or downwards ($S_i = -1$). The configuration of the system is described by the state of all N particles $\underline{S} = (S_1, \ldots, S_N)$. The *Hamilton*

function (or *Hamiltonian*) or energy function $\mathcal{H}(\underline{S})$ of the system assigns to each state an energy – from a microscopical view point this is all one needs to know to describe the physics of the system. In case of the aforementioned Ising model it is, for instance,

$$\mathcal{H}(\underline{S}) = -\sum_{i,j=1}^{N} J_{ij} S_i S_j - h \sum_{i=1}^{N} S_i \tag{5.1}$$

where J_{ij} is a magnetic interaction strength ($J_{ij} > 0$ favoring parallel orientation of the spin i and j, $J_{ij} < 0$ favoring anti-parallel orientation) and h an external field strength.

A typical task in the statistical physics of systems that are described by a Hamiltonian $\mathcal{H}(\underline{S})$ consists in the computation of expectation values of observables such as energy, pressure, or magnetization. The result depends on the *ensemble* which is chosen. For instance, the *microcanonical ensemble* describes a system which is isolated from its environment. This means that some quantities like the number of particles, the volume or the energy are conserved. For other ensembles e.g. the chemical potential or the pressure can be fixed. Here we consider the *canonical ensemble*, which describes systems connected to a *heat bath* (an infinite source/sink of energy), which always holds the system at *temperature* T. Then, the so called *thermal expectation values* of observables $A(\underline{S})$ are given by

$$\langle A \rangle_T \equiv \frac{1}{Z} \sum_{\underline{S}} A(\underline{S}) \exp(-\mathcal{H}(\underline{S})/k_\mathrm{b}T). \tag{5.2}$$

Here the sum is over all states of the system, k_b the Boltzmann constant $Z = \sum_{\underline{S}} \exp(-\mathcal{H}(\underline{S})/k_\mathrm{b}T)$ is the *partition function*, which simply ensures the normalization of the Boltzmann distribution

$$P(\underline{S}) = \frac{1}{Z} \exp(-\mathcal{H}(\underline{S})/k_\mathrm{b}T) \tag{5.3}$$

that is the probability for a system coupled to a heat bath with temperature T to be in state $\underline{S}$. A thermal expectation value of an observable $\mathcal{H}(S)$ is therefore just a weighted sum of values $A(\underline{S})$ that observable A takes on as in state $\underline{S}$. In the Ising system defined above, one such observable is for instance the magnetization

$$M(\underline{S}) = \sum_{i=1}^{N} S_i \tag{5.4}$$

that can be used to discriminate between different magnetic phases of a ferromagnet. The magnetization per site is $m = M/N$. The internal energy E of the system at temperature T is simply the thermal expectation value of $\mathcal{H}$

$$E = \langle \mathcal{H} \rangle_T. \tag{5.5}$$

However, it is the *free energy* $\mathcal{F}$ and not the internal energy that is crucial for the thermodynamic properties of a physical system at non-vanishing temperatures ($T > 0$). It is defined to be proportional to the logarithm of the partition function:

$$\mathcal{F} = -k_\mathrm{b} T \ln Z. \tag{5.6}$$

All thermodynamic quantities can be calculated from the free energy/partition function by taking derivatives, e.g. the expectation values of the energy E, the magnetization M, the specific heat c and the susceptibility χ (with $\beta = 1/k_\mathrm{b}T$)

$$E = -\frac{\partial \ln Z}{\partial \beta} \tag{5.7}$$

$$M = -\frac{\partial \mathcal{F}}{\partial h} \tag{5.8}$$

$$c = \frac{\partial E}{\partial T} = -T\frac{\partial^2 \mathcal{F}}{\partial T^2} \tag{5.9}$$

$$\chi = \frac{\partial M}{\partial h} = -\frac{\partial^2 \mathcal{F}}{\partial h^2} \tag{5.10}$$

By a short calculation, you will find that the second derivatives specific heat and susceptibility are related to the fluctuations of the related first derivatives:

$$c = (\langle \mathcal{H}^2 \rangle_T - \langle \mathcal{H} \rangle_T^2)/k_\mathrm{b}T^2 \tag{5.11}$$

$$\chi = (\langle M^2 \rangle_T - \langle M \rangle_T^2)/k_\mathrm{b}T \tag{5.12}$$

The importance of the free energy is due to the fact that for the thermodynamics not only the energetics of the states of the system but also their number, the density of states, play a role. The measure for the latter is the (Boltzmann) *entropy* defined by

$$S = -\frac{\partial \mathcal{F}}{\partial T} = -k_\mathrm{b}\langle \ln P(\underline{S}) \rangle_T. \tag{5.13}$$

We do not have the space here to uncover all the deep implications of this quantity, *and* it is not necessary for the purpose of this book, which is mainly concerned with *zero* temperature problems. If we insert $P(\underline{S})$ from (5.3) into (5.13), use (5.6) and compare with (5.5), we see immediately that

$$\mathcal{F} = E - TS, \tag{5.14}$$

Thus the entropy does not play a role at $T = 0$ where the internal energy becomes equal to the free energy. In that case, a closer look at Eq. (5.3) for the Boltzmann probability reveals the crucial role that the energy $\mathcal{H}(\underline{S})$ has in determining the thermodynamics of a system: for fixed temperature T the weight $P(\underline{S})$ of state $\underline{S}$ gets smaller the larger its energy is. In the zero-temperature limit $T \to 0$ only the state $\underline{S}_0$ with the lowest energy $E_0 = \min_{\underline{S}}\{\mathcal{H}(\underline{S})\}$, the ground state, contributes to the thermodynamic average (5.2) of any observable A. All the other states with energies $\mathcal{H}(\underline{S}) > E_0$ are exponentially suppressed and are irrelevant at $T = 0$. This clarifies the importance of the ground state (S) of a physical system described by a Hamiltonian $\mathcal{H}(\underline{S})$.
For non-zero temperature it is a tremendous task to perform the sum given in (5.2) explicitly and it can be achieved analytically only in a few exactly solvable cases. Note that the number of states grows exponentially with the size N of the system; for instance an Ising system with N spins can take on 2^N different states $\underline{S}$. In many cases the individual particle have even to be described by a continuous variable with one or more components, e.g. $S_i = (\cos\varphi, \sin\varphi)$, a two-component vector parameterized

by the angle $\varphi \in [0, 2\pi)$, also called a XY-spin, which makes the task even more difficult. On the other hand, not all states $\underline{S}$ are equally important in an approximate calculation of the thermodynamic expectation value – at low temperature, for instance, those with a high energy are exponentially suppressed.
The key trick in evaluating the sum in (5.2) approximately on a computer, instead of an exact enumeration, is to generate a random sequence of states that obeys by construction the desired Boltzmann probability. This a physicist calls *importance sampling*. In general, an algorithm to evaluate some quantities by means of random processes is called *Monte Carlo* simulation. Thus we want to generate a sequence of states $\underline{S}_1 \to \underline{S}_2 \to \underline{S}_3 \to \ldots$ recursively one from the other with a carefully designed transition probability $w(\underline{S} \to \underline{S}_{n+1})$ such that states $\underline{S}$ occur within the sequence with probability $P_{\mathrm{eq}}(\underline{S}) = Z^{-1} \exp(-\beta \mathcal{H}(\underline{S}))$. How do we find the appropriate transition probabilities $w(\underline{S} \to \underline{S}')$ that gives the probability to generate the state $\underline{S}'$ in the next step if state $\underline{S}$ is given? The sequence under consideration can be interpreted as one realization of a Markov process, in which states $\underline{S}$ occur with probability $P_t(\underline{S})$ in the t-th step and whose evaluation is given by the Master equation

$$P_{t+1}(\underline{S}) = P_t(\underline{S}) + \sum_{\underline{S}'} \big\{ w(\underline{S}' \to \underline{S}) P_t(\underline{S}') - w(\underline{S} \to \underline{S}') P_t(\underline{S}) \big\}. \tag{5.15}$$

In the sum on the right hand side over all states $\underline{S}'$, the first term describes processes which move a system into state $\underline{S}$, while the second term accounts for processes leaving state $\underline{S}$. After a large number of steps ($t \to \infty$) the probabilities $P_t(\underline{S})$ will approach a stationary distribution $P(\underline{S}) = \lim_{t\to\infty} P_t(\underline{S})$, which we want to be the Boltzmann distribution $P(\underline{S}) = P_{\mathrm{eq}}(\underline{S}) = Z^{-1} \exp(-\beta \mathcal{H}(\underline{S}))$. We can design the transition probabilities $w(\underline{S} \to \underline{S}')$ in such a way, that for $P_t(\underline{S}) = P_{\mathrm{eq}}(\underline{S})$ each term in the sum on the right hand side of (5.15) vanishes:

$$\forall \underline{S}, \underline{S}' : \quad w(\underline{S}' \to \underline{S}) P_{\mathrm{eq}}(\underline{S}') = w(\underline{S} \to \underline{S}') P_{\mathrm{eq}}(\underline{S}). \tag{5.16}$$

This relation is called *detailed balance*. Now it is ensured that the desired distribution *is* the Markov process defined by (5.15). Please note that, in principle, you can use other measures to ensure that the equilibrium distribution is obtained. To ensure detailed balance, Metropolis et al. [4] chose

$$w(\underline{S} \to \underline{S}') = \begin{cases} \Gamma & \text{for} \quad \Delta\mathcal{H} \le 0 \\ \Gamma \exp(-\Delta\mathcal{H}/k_{\mathrm{b}}T) & \text{for} \quad \Delta\mathcal{H} > 0 \end{cases} \tag{5.17}$$

where $\Delta\mathcal{H} = \mathcal{H}(\underline{S}') - \mathcal{H}(\underline{S})$ is the energy difference between the old state $\underline{S}'$ and the new state $\underline{S}$ and Γ an arbitrary transition rate such that w has the meaning of a transition probability per unit time. By inserting these transition probabilities into (5.16), one sees that they fulfill the detailed balance condition.
There is a considerable amount of freedom in the choice of the move $\underline{S} \to \underline{S}'$, but one should note that because of $w(\underline{S} \to \underline{S}')/w(\underline{S}' \to \underline{S}) = \exp(-\Delta\mathcal{H}/k_{\mathrm{b}}T)$ the energy change $\Delta\mathcal{H}$ should not be too large when going from $\underline{S} \to \underline{S}'$ or vice versa. Hence typically it is necessary to consider small changes of $\underline{S}$ only, since otherwise the acceptance rate of either $\underline{S} \to \underline{S}'$ or $\underline{S}' \to \underline{S}$ would be very small (and a lot of computer

time would be wasted since the procedure would be poorly convergent). There are exceptions from this rule, like the cluster algorithms for spin models [5, 6], where clever schemes make large changes in configuration space possible, while the energy changes $\Delta\mathcal{H}$ are kept small. But these cluster algorithms do not work in general.
So, for instance, for the Ising model one conventionally uses a single spin flip dynamics, where at each move $\underline{S} \to \underline{S}'$ only a single spin, say S_i, is flipped: $S_i \to -S_i$ and the other spins do not change $S_j = S'_j \quad \forall j \neq i$. Then

$$w(S_i \to -S_i) = \begin{cases} \Gamma & \text{for} \quad S_i = -\text{sign}(h_i) \\ \Gamma \exp(-2S_i h_i / k_\text{b} T) & \text{for} \quad S_i = \text{sign}(h_i) \end{cases} \tag{5.18}$$

where $h_i = \sum_{j(\neq i)} J_{ij} S_j + b_i$ is the local field acting upon spin S_i in state $\underline{S}$.
In practice one realizes the sequence of states $\underline{S}$ with chosen transition probabilities w by executing the following Monte Carlo program to calculate an estimate of the thermal expectation value $\langle A \rangle_T$ of an observable $A(\underline{S})$:

algorithm Monte Carlo (T, N_steps)
begin
  choose start configuration $\underline{S}$; set $A_\text{av} := 0$;
  for step $:= 1, \ldots, N_\text{steps}$ **do**
  begin
    generate trial state $\underline{S}'$;
    compute $w(\underline{S} \to \underline{S}') =: w$;
    generate uniform random number $x \in [0, 1]$;
    if $(w < x)$ **then**
      reject $\underline{S}'$;
    else
      accept $\underline{S}'$ (i.e. $\underline{S} := \underline{S}'$);
    sum up average $A_\text{av} := A_\text{av} + A(\underline{S})$;
  end
  $A_\text{av} := A_\text{av} / N_\text{steps}$;
end

After a number steps N_steps, the number of trial states is usually called the number of *Monte Carlo steps*, one stops and we get $A_\text{av}(N_\text{steps})$ as an estimate of $\langle A \rangle_T$. This is true because according to what we have said about the stationary distribution of this process, we have

$$\langle A \rangle_T = \lim_{N_\text{steps} \to \infty} A_\text{av}(N_\text{steps}). \tag{5.19}$$

Of course it is impossible to perform the limit $N_\text{steps} \to \infty$ on a computer. Thus, one has to devise a reliable criterion to decide, which value of N_steps for the number of Monte Carlo steps is sufficient to get an estimate for $\langle A \rangle_T$ with the desired accuracy. This issue would fill another book, we therefore refer the reader to [7, 8]. We only mention that the interval $[A_\text{av} - \Delta A, A_\text{av} + \Delta A]$ contains the true value of $\langle A \rangle_T$ with

a probability of 0.65 (assuming that A_{av} is Gaussian distributed). ΔA is the *standard error bar* of the result, given by

$$\Delta A = \sqrt{\frac{\langle A^2\rangle_T - \langle A\rangle_T^2}{(N_{\mathrm{steps}} - 1)(1 + 2\tau/\delta t)}} \tag{5.20}$$

where τ is the *correlation time* of the quantity, which describes how long you have to simulate to obtain two independent measurements. The correlation time can be obtained from an *autocorrelation function* $C_A(t)$ in equilibrium

$$C_A(t) = \frac{\langle A(0)A(t)\rangle_T - \langle A\rangle_T^2}{\langle A^2\rangle_T - \langle A\rangle_T^2}, \tag{5.21}$$

through $C_A(\tau) = e^{-1}C_A(0)$. The value of δt measures the distance between two consecutive measurements, using the same time unit that is chosen to measure τ. For the algorithm above, $\delta t = 1$ (one Monte Carlo step). In case you take your samples very rarely ($\delta t \gg \tau$), Eq. (5.20) becomes

$$\Delta A = \sqrt{\frac{\langle A^2\rangle_T - \langle A\rangle_T^2}{(N_{\mathrm{steps}} - 1)}} \tag{5.22}$$

What is important for our topic "optimization" is that the Monte Carlo procedure can also be used to find the ground state configuration $\underline{S}_0$ of the system under consideration: for low temperature only the states with the lowest energies allow a significant weight $P_{\mathrm{eq}}(\underline{S})$, and therefore by reducing the temperature step by step down to $T = 0$ the process just described should converge to the state with the lowest energy $\underline{S}_0$. This procedure is called *simulated annealing* [9] and is one of many methods among what is known as the field of "stochastic optimization", see Chap. 11. Of course for lower and lower temperatures the equilibration time of the process gets larger and larger, necessitating huge numbers of steps N_{steps} to guarantee equilibration. Since this condition cannot be fulfilled for arbitrary low temperatures there is no guarantee of finding the exact ground state with simulated annealing. However, sometimes this is the only method at hand.

5.2 Phase Transitions

We all know that when heating water (at a particular pressure p) it starts to boil at $T_{\mathrm{c}} = 100°C$ and transforms into vapor, see Fig. 5.1. This is the archetypical example of a *phase transition*, it separates a temperature range (at fixed pressure), where water is a liquid to a range where it is a gas . The transition is characterized by an *order parameter* of the system, in this case the density which can be obtained also by a suitable derivative of the free energy. Here the density jumps discontinuously at the transition, the liquid density being much larger than the gas density. Only at one particular point, the *critical point*, of the (p,T)-diagram, the so called *phase diagram*, the transition is continuous. The latter transition is referred to as being of *second order*, which means that discontinuities appear only in the second derivative of

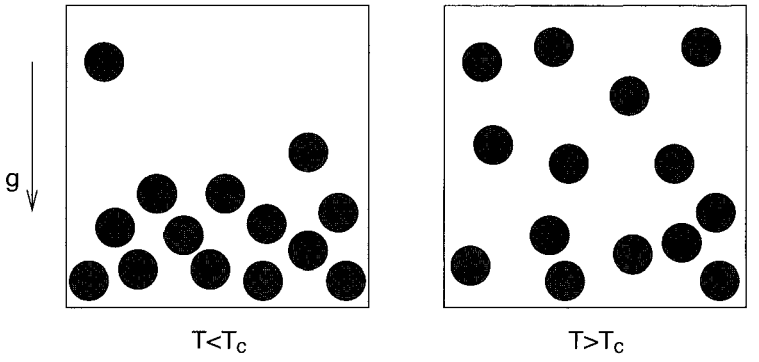


Figure 5.1: Schematic picture of the liquid-gas transition of water. Below T_c the system is a fluid (left), while for higher temperatures it is a gas (right).

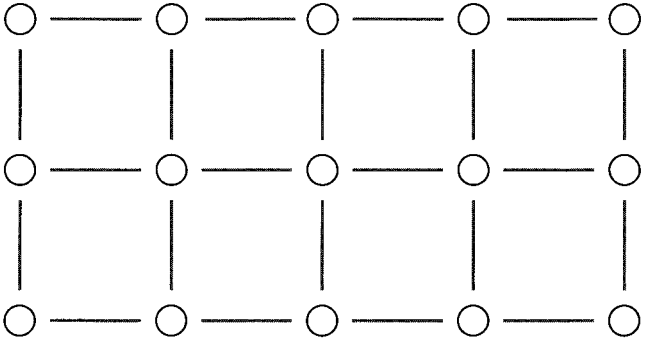


Figure 5.2: A two dimensional Ising ferromagnet. Near neighbors of the lattice interact ferromagnetically, indicated by straight lines joining them.

the free energy, whereas in the former case it is of *first order*, meaning that a first derivative of the free energy (the density) is already discontinuous.

In general, a phase transition in a many particle system is induced by the variation of a parameter (such as temperature, pressure, etc.) and is (usually) mathematically characterized by a discontinuity or a singularity in an appropriately chosen order parameter. It separates regions in the phase diagram with different physical properties, most elegantly expressed by their symmetries and reflected by the behavior of the order parameter. The concept is most easily visualized by an example, the Ising model in an external field that can simultaneously serve as a lattice gas model for the liquid/gas transition (spin up $\hat{=}$ particle present, spin down $\hat{=}$ particle absent, magnetic interaction $\hat{=}$ nearest neighbor attraction, field $\hat{=}$ chemical potential). The

spins $S_i = \pm 1$ are localized on the sites of a 2-dimensional lattice (Fig. 5.2), the Hamiltonian is

$$\mathcal{H} = -J \sum_{\langle ij \rangle} S_i S_j - h \sum_i S_i \tag{5.23}$$

$\langle ij \rangle$ means the sum over all nearest neighbor pairs of the lattice. For all nonzero magnetic fields h, the model is paramagnetic, i.e. the average orientation of the spins is determined by the sign of h. Below a critical temperature T_c, the order parameter, the magnetization per spin m as a function of the field h, has a discontinuous jump at $h = 0$, which is the sign of a first order phase transition (Fig. 5.3).

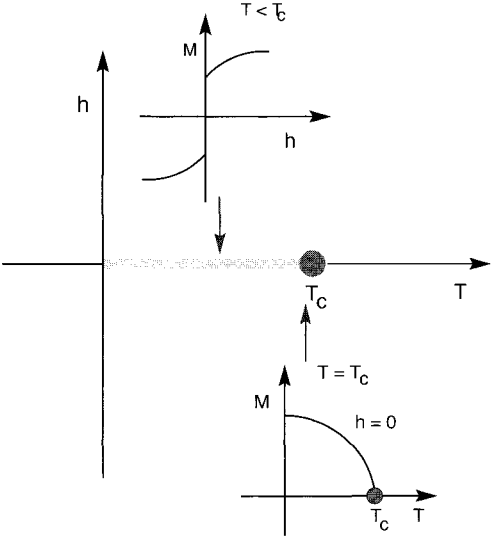


Figure 5.3: The (T, h) phase diagram of the 2d Ising model. The insets show the magnetization as a function of the field for two different temperatures indicating the critical behavior for $T = T_c$ and the first order transition at $T < T_c$.

For $h = 0$ the model possesses a 2nd order phase transition at the critical temperature T_c, where the magnetization has a singularity $m \sim (T_c - T)^\beta$ for $T < T_c$, but $m = 0$ for $T > T_c$. The exponent β, not to be confused with $\beta = 1/k_b T$, characterizes the behavior near the phase transition, called the *critical behavior*. Thus, β is called a *critical exponent*[1]. Also for other interesting quantities critical exponents are defined, see Secs. 4.1 and 5.4. It can be shown that for many models the actual values of the critical exponents do not depend on details of the model. In that case one speaks of *universality*.

For the purpose of this book (the computation of various quantities in a finite system) we should emphasize that the above scenario describes the behavior of the *infinite*

[1] It turns out that $\beta = 1/8$.

system. Mathematically, there is no phase transition in a finite system: the partition function is a sum of a *finite* (although exponentially large) number of analytical terms, thus the free energy is analytic and no singularities occur. In what follows we describe how this infinite system phase transition can be located and studied in finite systems. Using a technique called *finite-size scaling* (FSS), it is possible e.g. to calculate the values of the critical exponents of a given model. The mathematical background for finite-size scaling is provided by the renormalization-group theory [10, 11], which is beyond the scope of this book. Here we show how the ideas evolve when considering e.g. the percolation problem.

5.3 Percolation and Finite-size Scaling

One of the simplest and most studied phase transitions of second order is the percolation transition. Percolation is defined in an operative way: take an arbitrary d-dimensional lattice (square, triangular, sc, fcc, ...) and occupy each site randomly with a probability p. So a fraction $1-p$ of the sites are not occupied. We identify the connected *clusters* of this randomly occupied lattice: two sites are connected if one finds a path from one site to the next only using bonds between occupied nearest neighbors. There exists a critical value p_c for the occupation probability below which only finite clusters (of size $s = 1, 2, 3, \ldots$) even in the infinite system exist, whereas above it ($p > p_c$) at least one percolating i.e. infinite cluster exists that spins from one end of the system to the other (Fig. 4.1).
We define the probability that an occupied site is part of the percolating cluster: $P_\infty(p)$, the concentration of clusters consisting of s occupied sites: $n_s(p)$ and the percolation susceptibility $\chi(p) = \frac{1}{p}\sum_{s=1}^{\infty}{}' s^2 n_s(p)$, where $\sum'$ means the sum over all clusters except the infinite one ($p > p_c$). At the percolation transition we encounter critical singularities [12] in the variable $\delta = (p - p_c)/p_c \to 0$, described by the critical exponents β, γ, τ and σ via

$$P_\infty(p) = \begin{cases} B\delta^\beta & \delta > 0 \quad (p > p_c) \\ 0 & \delta < 0 \quad (p < p_c) \end{cases} \tag{5.24}$$

$$\chi(p) = \begin{cases} \Gamma^- \delta^{-\gamma} & \delta > 0 \quad (p > p_c) \\ \Gamma^+ (-\delta)^{-\gamma} & \delta < 0 \quad (p < p_c) \end{cases} \tag{5.25}$$

and

$$n_s(p) = s^{-\tau} \tilde{n}(s^\sigma \delta) \,, \tag{5.26}$$

where $\tilde{n}(x)$ is a (usually unknown) scaling function. For all different lattice types (square, triangle, cubic, ...) the exponents depend on each other in the same way: $\tau = 2 + \beta/(\gamma + \beta)$ and $\sigma = 1/(\gamma + \beta)$,
but the actual values are not the same for different lattice types. The percolation cluster at $p = p_c$ is a fractal and its mass scales as $M \propto r^{d_f}$ with its radius (or linear size) r, where d_f is the *fractal dimension*. In Fig. 5.4 we show an example for the determination of the fractal dimension for $d = 2$ using the methods describe in Sec. 4.1 for percolation problems.

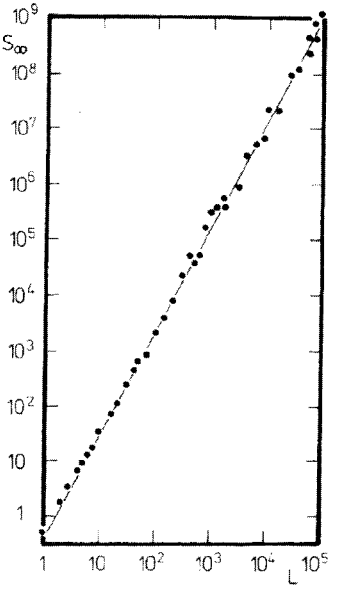


Figure 5.4: Monte Carlo data for the size of the largest cluster at the percolation threshold $p = p_c = 1/2$ of the triangular lattice, as a function of the linear system size L of the lattice. The slope of the straight line in this log-log plot is the exactly know fractal dimension $d_f = 91/48$. From [12].

In a *finite* system quantities such as $\chi(p)$ cannot diverge but reach a maximum, also $P_\infty(p)$ does not vanish [for all p the probability (αp^L) for a percolating cluster is exponentially small]. As a consequence, singularities are smeared out for finite systems. In a finite subsystem of linear size L a percolating cluster has radius $r_s = L$. Thus, using (5.26), the probability for the occurrence of a percolating cluster $\tilde{P}_L^{\text{perc}}(p_c)$ is

$$\tilde{P}_L^{\text{perc}}(p_c) \approx L^d \int_{r_s=L} ds\, n_s(p_c) \approx L^d \tilde{n}(0) \int_{L^{d_f}} ds\, s^{-\tau} \propto L^{d+d_f(1-\tau)} \tag{5.27}$$

Hence one finds a percolating cluster in the finite subsystem with the probability of order 1 (i.e. $\tilde{P}_L^{\text{perc}}(p_c) \approx 1$) if

$$d_f = \frac{d}{\tau - 1} = d - \beta/\nu \tag{5.28}$$

where we used $\tau = 2 + \beta/(\gamma + \beta)$ and the hyper scaling relation [10, 11] $d\nu = \gamma + 2\beta$, which involves the dimension d of the system. Furthermore, ν is the critical exponent of the *correlation length* ξ, which is defined as the typical length scale over which the connectedness probability, i.e. the probability that two sites a distance r apart belong to the same connected cluster, decays in the infinite system. It diverges at the critical

point as

$$\xi \propto |p - p_c|^{-\nu} \tag{5.29}$$

The mass (number of sites) of a percolating cluster in a finite system is simply the mass of a cluster of radius L

$$M \propto L^{d_f} \propto L^{d-\beta/\gamma} \tag{5.30}$$

which means that the probability for a site to belong to a percolation cluster is

$$P_L(p_c) \propto L^{-\beta/\gamma} \tag{5.31}$$

since $P_L(p_c) = M/L^d$.
The distribution $n_s(p)$ in a finite system is cut off at $s \propto L^{d_f}$. Therefore also the divergence of the susceptibility is

$$\begin{aligned} & \chi_L(p) && \approx && \tfrac{1}{p} \sum_{s=1}^{L^{d_f}} s^2 n_s(p) \\ \Rightarrow\ & \chi_L(p_c) && \propto && \int_0^{L^{d_f}} ds\ s^{2-\tau} \propto L^{d_f(3-\tau)} = L^{\gamma/\nu} \\ \text{and}\ & \chi_L(p) && \propto && \int_0^{L^{d_f}} ds\ s^{2-\tau} \tilde{n}(s^\sigma \delta) \\ & && \propto && \delta^{-(3-\tau)/\sigma} \int_0^{(L\delta^{1/\sigma d_f})^{d_f \sigma}} dx\, x^{(2-\tau)\sigma} \tilde{n}(x) \end{aligned} \tag{5.32}$$

We will look more closely at the combination of exponents that appear in the expression for $\chi_L(p)$: since $(3-\tau)/\sigma = \gamma$ and $1/\sigma d_f = \nu$ we get

$$\chi_L(p) = \delta^{-\gamma} \tilde{\chi}(L\delta^\nu) \tag{5.33}$$

in other words: $\chi_L(p)$ is composed of a diverging factor $\delta^{-\gamma}$, which describes the behavior in the infinite system, and a scaling function that depends on the scaling variable L/ξ, where $\xi = \delta^{-\nu}$. Since there cannot be a divergence in a finite system, one expects that $\tilde{\chi}(x)$ vanishes for $x \to 0$ in such a way that the divergence of the prefactor is canceled. In this case, $\tilde{\chi}(x) \propto x^{\gamma/\nu}$ for $x \to 0$. This yields at $p = p_c$ ($\delta = 0$) the correct finite size behavior $\chi_L(p = p_c) \propto L^{\gamma/\nu}$.
This is the essence of finite-size scaling (FSS), scaling functions always depend on the ratio of the two lengths L and ξ and the prefactors describe the singularities, either in terms of the distance from the critical point $\delta = \xi^{-\gamma}$, or in terms of the system size L, which then replaces ξ, i.e. for (5.33)

$$\chi_L(p) = L^{\gamma/\nu} \tilde{\chi}(L\delta^\nu) \tag{5.34}$$

Analogously

$$P_L(p) = L^{-\beta/\gamma} \tilde{p}(L\delta^\nu) \tag{5.35}$$

$$\tilde{P}_L^{\text{perc}}(p) = \tilde{P}^{\text{perc}}(L\delta^\nu) \tag{5.36}$$

Note that these relations only hold asymptotically, i.e. close to p_c and for large values of L. Further away from p_c and for smaller system sizes corrections to scaling occur. The general scheme is to perform simulations at different systems sizes and use these data and equations like (5.34), (5.35) and (5.36) to extract all information. Equation (5.36) is particularly useful for the numerical determination of p_c: for $p = p_c$ the value of $\tilde{P}_L^{\text{perc}}$ is independent of the system size, $\tilde{P}_L^{\text{perc}}(p_c) = \tilde{p}(0)$ and therefore the curves for different fixed system sizes should intersect at the critical point $p = p_c$!
Practical issues of performing a FSS analysis are covered in Sec. 13.8.3. In the following section, we will explain the FSS behavior of magnets.

5.4 Magnetic Transition

In a temperature or a disorder driven phase transition the scenario is similar. At the transition singularities in various quantities occur, and these are smeared out for a finite system in a systematic way that enables one to extract the critical exponents, which determine the singularities of the infinite system.
Let us, for instance, consider the aforementioned Ising model (5.1) at temperature T. In $d \geq 2$ space dimensions a phase transition at a critical temperature T_c occurs, from a paramagnetic phase at high temperature ($T > T_c$) to a ferromagnetic phase at low temperatures ($T < T_c$). The order parameter, indicating the appearance of magnetic order in this system, is the magnetization per site

$$m(T) := \langle |M|/N \rangle_T = \begin{cases} B\delta^\beta & \delta < 0 \\ 0 & \delta > 0 \end{cases} \tag{5.37}$$

where $M = M(\underline{S}) = \sum_{i=1}^{N} S_i$ and

$$\chi(T) := T^{-1}(\langle M^2 \rangle_T - \langle M \rangle_T^2)/N \quad = \quad \begin{cases} \Gamma^+ \delta^{-\gamma} & \delta > 0 \\ \Gamma^- (-\delta)^{-\gamma} & \delta < 0 \end{cases} \tag{5.38}$$

is the susceptibility. Here $\delta := (T - T_c)/T_c$ denotes the distance from the critical point. The singular dependencies of $m(T)$ and $\chi(T)$ on δ are valid for the infinite system. If $N = L^d$ is finite one gets

$$m_L(T) = L^{-\beta/\nu} \tilde{m}(L^{1/\nu}\delta) \tag{5.39}$$

$$\chi_L(T) = L^{\gamma/\nu} \tilde{\chi}(L^{1/\nu}\delta) \tag{5.40}$$

The correlation length $\xi = |\delta|^{-\nu}$ is defined via the decay of spatial spin correlations:

$$C(r) = N^{-1} \sum_{i=1}^{N} \{ \langle S_i S_{i+\mathbf{r}} \rangle_T - \langle S_i \rangle_T \langle S_{i+\mathbf{r}} \rangle_T \} = r^{-(d-2+\eta)} \exp(-r/\xi) \tag{5.41}$$

close to the critical point. Since the correlation length diverges at the critical point, $C(r) \sim r^{-(d-2+\eta)}$ holds for $T = T_c$. η is another critical exponent that is related to γ via [since $\int d\mathbf{r} C(\mathbf{r}) = \chi$]

$$2 - \eta = \gamma/\nu. \tag{5.42}$$

A useful way in which one can estimate the critical exponents from a set of Monte Carlo data for finite sizes L and different temperatures T (or distances from the critical point δ) is to plot for instance $\chi_L(T)/L^{\gamma/\nu}$ versus $L\delta^\nu$ or $t = L^{1/\nu}\delta$. Then data for different system sizes should fall on a single curve, the scaling function $\tilde{\chi}$, according to (5.40). The exponents γ and ν as well as the critical temperature T_c are then fit parameters, by which one has to try to achieve the best possible data collapse for different system sizes. In Fig. 5.5 we show such a scaling plot for the two-dimensional Ising model.
Note that the presence of three free fit parameters makes the data determination of the critical quantities using such a data collapse open to large systematic errors. It would be much better if one could estimate one or two of these quantities separately such that the number of free parameters is reduced. Fortunately there is such a method for determining the critical temperature, similar to the use of the percolating cluster probability in the aforementioned percolation problem, namely the dimensionless ratio of moments ("Binder cumulant")

$$G_L(T) = \frac{1}{2}\left(3 - \frac{\langle m^4\rangle_T}{\langle m^2\rangle_T^2}\right) = \tilde{g}(L^{1/\nu}\delta). \tag{5.43}$$

This quantity is the second cumulant of the probability distribution of the fluctuating magnetization per site m and it is 0 for a Gaussian and 1 for a double-delta function. Since this is exactly what one expects for the paramagnetic and the ferromagnetic phase, respectively, of a ferromagnet in the infinite system-size limit, this cumulant is a step function in the thermodynamic limit. Since for any moment of the magnetization one expects a scaling behavior similar to (5.39):

$$m_L^{(n)}(T) = \langle m^n\rangle_T = L^{-n\beta/\nu}\tilde{m}^{(n)}(L^{1/\nu}\delta) \tag{5.44}$$

it follows that $G_L(T = T_c)$ is independent of system size and a family of curves for $G_L(T)$ with different but fixed system sizes intersects at $T = T_c$ (with corrections to scaling, i.e. the deviation from the asymptotic behavior for smaller system sizes).
The practical procedure is exemplified in Fig. 5.5 for the two-dimensional Ising model. First the location of the critical point, T_c, is determined by the intersection point of the curves for different system sizes of the dimensionless ratio of moments $G_L(T)$, c.f. Fig. 5.5a. Then the same data for $G_L(T)$ are plotted against the scaling variable $L^{1/\nu}(T - T_c)/T_c$, where one chooses the exponent ν such that the best data collapse is achieved, c.f. Fig. 5.5b. Next the magnetization exponent β and the susceptibility exponent γ are estimated by plotting the rescaled magnetization $L^{-\beta/\nu}m_L(T)$ and the rescaled susceptibility $L^{\gamma/\nu}\chi_L(T)$ against the scaling variable $L^{1/\nu}(T - T_c)/T_c$, see Fig. 5.5c and Fig. 5.5d, where one chooses the exponents β and γ such that the best data collapse is achieved and T_c and ν are, for instance, taken from a and b. Other quantities, like e.g. the specific heat can also be studied in this way, and the critical exponents ($\alpha = 0$, i.e. a logarithmic divergence in case of the specific heat of the 2d Ising model) extracted.
One recognizes the similarity between the critical singularities for percolation and the one for a thermal phase transition – and also of the finite-size scaling behavior. Actually for the case that there is *one* single length scale ξ determining the transition (the

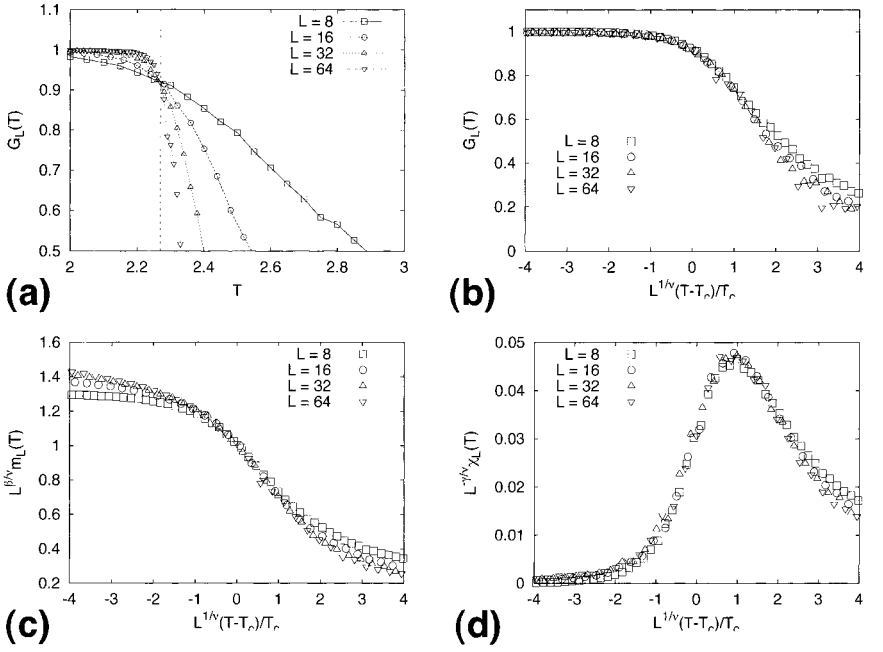


Figure 5.5: Finite-size scaling (FSS) behavior of the two-dimensional Ising model on the square lattice (with $J = 1$). The data are obtained with a conventional single spin-flip Monte Carlo simulation using Metropolis transition rates (5.17). **(a)** The dimensionless ratio of moments $g_L(T)$, Eq. (5.43). The data for different system sizes intersect at the temperature $T_c = 2.27$, which is indicated by the vertical line. This is the estimate for the critical temperature (which is exactly $T_c = 2.269\ldots$). **(b)** Scaling plot of the dimensionless ratio of moments according to (5.43). The data of (a) are plotted against the scaling variable $L^{1/\nu}(T - T_c)/T_c$ with $T_c = 2.27$ (from a) and $\nu = 1$ (which is the exact value). Note that the data collapse is good close to the critical point (around 0 on the x-axis) and gets worse far away from it. **(c)** The magnetization $m_L(T)$, rescaled by its FSS behavior at T_c, $L^{-\beta/\nu}$, versus the scaling variable $L^{1/\nu}(T - T_c)/T_c$, with $T_c = 2.27$ (from a), $\nu = 1$ and $\beta = 1/8$ (which are the exact values). Close to the critical point the data collapse according to (5.39) is good. **(d)** FSS-plot of the susceptibility $\chi_L(T)$, Eq. (5.40), which is rescaled by its FSS at T_c, $L^{\gamma/\nu}$, versus the scaling variable $L^{1/\nu}(T - T_c)/T_c$, with $T_c = 2.27$ (from a), $\nu = 1$ and $\gamma = 7/4$ (which are the exact values).

correlation length) this is the generic scenario (there are always exceptions, but as a working hypothesis it is good for many systems); an order parameter, its susceptibility and the correlation function (including a characteristic lengths scale, the correlation length) yield a set of critical exponents: α, for the specific heat, β, for the order parameter, γ, for the order parameter susceptibility, ν for the correlation lengths, etc.,

and usually only 2 of them are independent; in the case when the hyper-scaling relation $2d = 2 - \alpha$ is violated, 3 of them. These determine the *universality class* of the system.
Now we will shortly introduce the notion of *universality*. According to the theory of critical phenomena including the renormalization group, the exponents determining the critical singularities of a many particle system at a second order phase transition depend only on features like the space dimension d, the number of components of the order parameter (e.g. 1 for the Ising model, 2 for XY spins etc.), the range of the interactions (short-range versus long-range, like e.g. Coulomb), possibly on quenched disorder (see below, depending on whether it is relevant or not), sometimes on the type of frustration (e.g. for spin glasses, see Chap. 9), etc. But they do *not* depend on microscopic features like the detailed lattice structure of next and next-to-next nearest neighbor interactions (as long as no frustration arises through these additional interactions), i.e. they are *universal* The two-dimensional Ising model has the *same* critical exponents (α, β, γ, etc.) for nearest neighbor interactions on the square lattice, the triangular lattice, the hexagonal lattice, the Kagomé lattice etc. Even if ferromagnetic next-nearest (or n-nearest) neighbor interactions are considered, the exponents will not change. The same for the three-dimensional case, which is particularly useful, since the critical point in the aforementioned (p, V) phase diagram of for instance water is in the universality class of the 3d Ising model (the density corresponds to the magnetization, the chemical potential to the external field. and on a lattice the presence of a molecule corresponds to spin up, the absence to spin down). This is one of the most important reasons why physicists are interested in the models we discuss in this book. These models will never describe an experimental system in all details – but the concept of universality tells us that this is not necessary as far as critical phenomena are concerned.

5.5 Disordered Systems

In all solid materials there is some *quenched* or *frozen in* disorder, even extremely pure materials contain impurities and defects. Surprisingly small amounts of impurities can significantly influence the phase transitions by which ordered structures form and in some classes of materials the quenched randomness is substantial and completely dominates their physical properties. Glasses and alloys, for instance, can be designed to have properties not achievable with pure elements or periodic structures.
A typical example for a system in which the least amount of disorder (in finite concentration, though) changes the universality class of a phase transition (i.e. alters the critical exponents characterizing it) is the random bond Ising ferromagnet in three space dimensions:

$$H = -\sum_{\langle ij \rangle} J_{ij} S_i S_j \tag{5.45}$$

where $S_i = \pm 1$, the sum runs over all nearest neighbor pairs of spins of a simple cubic lattice and the $J_{ij} (> 0)$ are quenched random variables (independently identically

distributed, in other words: uncorrelated, random variables) obeying a probability distribution, for instance a *binary* (or called *bimodal*) distribution $P(J_{ij}) = 1/2\delta(J_{ij} - J_1) + 1/2\delta(J_{ij} - J_2)$ with $J_2 > J_1 > 0$.
The fact that these interaction strengths are quenched or frozen in, modeling the quenched disorder in a real 3d magnet with short range interactions and an Ising symmetry, implies that they have to be chosen according to their distribution and fixed before calculating any physical quantity with this Hamiltonian. Therefore one cannot expect a priori that the thermodynamic expectation values, such as magnetization or specific heat, are independent of the specific choice of these random variables. Indeed it turns out that for instance correlation functions at the critical point (where the correlation length becomes infinite) show strong sample to sample fluctuations (i.e. strong variations among their thermal expectation values for different realizations of the disorder). Wiseman and Domany have shown [13] that ratios such as

$$R_{\mathcal{O}} = \frac{[(\langle \mathcal{O} \rangle_T - [\langle \mathcal{O} \rangle_T]_{\mathrm{av}})^2]_{\mathrm{av}}}{[\langle \mathcal{O} \rangle_T]^2_{\mathrm{av}}} \tag{5.46}$$

describing the variance of the the distribution of the observable $\mathcal{O}$ generated by the disorder realization, do in general *not* decay to zero when the ratio of system size and correlation length, $L/\xi(T)$, approach zero, i.e. when ξ diverges, e.g. at a critical point. One speaks of *lack of self-averaging* in this case. Away from a critical point, however, one expects self-averaging of all quantities $\mathcal{O}$, since then $R_{\mathcal{O}} \sim (\xi/L)^d$ for $L \gg \xi$.
Other quantities, like the free energy, have the property of being self-averaging even at the critical point, which means that their fluctuations between different disorder realizations become smaller with increasing system size. As a rule of thumb one can say that the most interesting variables, namely those providing information about the order emerging at a phase transition do show strong sample to sample fluctuations and things usually get worse when considering larger system sizes. Therefore in most cases it is mandatory to average the results over many thousands of disorder realizations and it is recommended not only to study average values but the whole distribution of the observables.
Hence we have to compute two averages: one over the different disorder realizations and one thermal average for each disorder realization. For instance the average magnetization is then defined as

$$m = \left[\left\langle \left| \frac{1}{N} \sum_{i=1}^{N} S_i \right| \right\rangle_T \right]_{\mathrm{av}} \tag{5.47}$$

or the average susceptibility

$$\chi = \frac{1}{N} \left[\left\langle \left(\sum_{i=1}^{N} S_i \right)^2 \right\rangle_T - \left\langle \sum_{i=1}^{N} S_i \right\rangle_T^2 \right]_{\mathrm{av}} \tag{5.48}$$

where $[\ldots]_{\mathrm{av}}$ denotes the average over the quenched disorder, and $\langle \ldots \rangle_T$ the thermal average with fixed realization of the disorder. More explicitly this means for the random Ising ferromagnet (5.45):

$$m = \int \prod_{\langle ij \rangle} \left(dJ_{ij} P(J_{ij}) \right) \left\{ \sum_{\underline{S}} \left| \frac{1}{N} \sum_{i=1}^{N} S_i \right| \cdot \exp\left(\frac{1}{k_{\mathrm{b}} T} \sum_{\langle ij \rangle} J_{ij} S_i S_j \right) \Big/ Z \right\}. \tag{5.49}$$

In practice (i.e. in a computer simulation) the exact average over the random variable has to be replaced by an unbiased sum over a large enough number of disorder realizations.

When computing ground-state properties of a model with quenched disorder the thermal average is simply replaced by the computation of the (exact) ground state(s) $\underline{S}$ and an evaluation of the observables under consideration in this state(s) $\underline{S}$.

The random-field Ising model, for instance (see Chap. 6),

$$H = -\sum_{\langle ij \rangle} J_{ij} S_i S_j \; - \sum_i h_i S_i \tag{5.50}$$

where h_i is a random variable, modeling a random external field, obeying some distribution with zero mean and variance h, has in three space dimensions a phase transition also at zero temperature ($T = 0$) from a paramagnetic phase ($m = 0$) to a ferromagnetic phase ($m > 0$) at a critical strength h_c of the random fields. If $\underline{S}^0(\underline{h})$ denotes the ground state of the Hamiltonian H with random fields $\underline{h}$ the average magnetization is simply

$$m_{T=0} = \left[\left| \frac{1}{N} \sum_{i=1}^{N} S_i^0(\underline{h}) \right| \right]_{\text{av}} = \int \prod_i \Big(dh_i P(h_i) \Big) \left| \frac{1}{N} \sum_{i=1}^{N} S_i^0(\underline{h}) \right|. \tag{5.51}$$

Sometimes one is interested in correlated disorder, for which only the generation of the random variables has to be adopted such that the joint distribution is obeyed, for instance $P(h_1, \ldots, h_N)$ instead of $\prod_i P(h_i)$.

The physical applications presented in this book are predominantly disordered systems so that we can skip the presentation of examples here: they will come in abundance in the later chapters.

The concept of universality, which we mentioned in the previous section concerning critical phenomena, carries over to critical points in disordered systems as well and implies here, besides the irrelevance of microscopic details of the model, that the critical exponents do not depend on the detailed probability distribution of the disorder. However, this idea is far from being as well established for disordered systems as it is for homogeneous systems, simply because exact results for disordered systems are rare and renormalization group calculations for field theories of disordered systems are difficult. Here one should note that only the exactly solvable disordered models do display this universality (e.g. the Mc-Coy-Wu model which is a 2d random ferromagnet with disorder only in one space direction).

On the other hand a number of numerical calculations (Monte Carlo at finite temperature as well as ground-state calculations at zero temperature) for spin-glass models [14] as well as random-field models [15] appear to be incompatible with the concept of universality: so a binary distribution seems to yield a different set of critical exponents than a continuous distribution. The usual argument brought against these numerical results by the community of believers in universality is that these computations are still in a pre-asymptotic regime, where finite-size effects are still strong and the true and unique disorder fixed point is still too far away when renormalizing the system sizes that could be studied. We are inclined to use Occam's razor in any unclear situation and simply recommend that it is always good to think twice before one abandons

an appealingly elegant concept like universality, even if there is no rigorous proof that it holds in the particular case one is considering. Nevertheless, when the numerical evidence against it grows in strength over the years, it might be a good time to start to think about something new.

Bibliography

[1] D. Chandler, *Introduction to Modern Statistical Mechanics*, (Oxford University Press, Oxford 1987)

[2] J.M. Yeomans, *Statistical Mechanics of Phase Transitions*, (Larendon Press, Oxford 1992)

[3] L.E. Reichl, *A Modern Course in Statistical Physics*, (John Wiley & sons, New York 1998)

[4] N. Metropolis, A.W. Rosenbluth, M.N. Rosenbluth, A.H. Teller, and E. Teller, *J. Chem. Phys.* **21**, 1087 (1953)

[5] R.H. Swendsen and J.S. Wang, *Phys. Rev. Lett.* **58**, 86 (1987)

[6] U. Wolff, *Phys. Rev. Lett.* **60**, 1461 (1988)

[7] M.E.J. Newman and G. T. Barkema, *Monte Carlo Methods in Statistical Physics*, (Clarendon Press, Oxford 1999)

[8] D.P. Landau and K. Binder, *A Guide to Monte Carlo Simulations in Statistical Physics*, (Cambridge University Press, Cambridge 2000)

[9] S. Kirkpatrick, C.D. Gelatt, and M.P. Vecchi, *Science* **220**, 671 (1983)

[10] D.J. Amit, *Field Theory, The Renormalization Group and Critical Phenomena*, (World Scientific, Singapore 1984)

[11] J. Cardy, *Scaling and Renormalization in Statistical Physics*, (Cambridge University Press 1996)

[12] D. Stauffer and A. Aharony, *Perkolationstheorie: eine Einfuehrung*, (Wiley-VCH, Weinheim 1995)

[13] S. Wiseman and E. Domany, *Phys. Rev. E* **52**, 3469 (1995)

[14] L. Bernardi and I. A. Campbell, *Phys. Rev. B/* **52**, 12501 (1995)

[15] J.-C. Anglès d'Auriac and N. Sourlas, *Europhys. Lett./* **39**, 473 (1997)

6 Maximum-flow Methods

This chapter introduces a model for a certain class of random magnetic systems. They consist of lattices where the atoms have a small magnetic moment, i.e. a *spin*. Within this model, for neighboring spins it is energetically favorable to point in the same directions, which means they interact *ferromagnetically*. A magnetic field acts on each spin, its sign and strength may change randomly and independently from spin to spin. These types of systems are described by *random-field* models. They have been studied widely by means of computer simulations and analytical calculations.
There are no direct experimental realizations of random-field systems, but it can be shown that another class of systems, the *diluted antiferromagnet in a field (DAFF)*, can be mapped onto random-field systems. The DAFF consists of spins on lattices as well, but neighboring spins interact *antiferromagnetically*. The system is called *diluted*, because not all sites of the lattice are occupied by magnetic atoms. Similar to the random-field model, a magnetic field acts on the spins, but it has the same direction and the same strength for all spins, i.e. it is homogeneous.
Here we are interested in the low temperature-properties of random-field systems and diluted antiferromagnets, especially in the ground states. From the viewpoint of a computational physicist, the calculation of ground states of this type of system belongs to the class of P problems, i.e. the $T = 0$ behavior can be studied for large system sizes.
Initially both models and some basic results are presented. Then it is shown how these systems can be mapped onto networks. It is demonstrated that the maximum flow through such a network is equivalent to the minimum energy of the corresponding system. Next, for pedagogical reasons, a simple algorithm for calculating the maximum flow is presented. Unfortunately, this algorithm turns out to be very slow. Therefore, in the main part of this chapter a highly efficient method is presented. Sometimes random-field systems and diluted antiferromagnets exhibit a huge number of ground states, i.e. the ground state is *degenerate*. It is explained how all different ground states of such a system can be calculated by a second calculation after the maximum flow has been obtained. Finally some of the most interesting results obtained with this algorithms are shown.

6.1 Random-field Systems and Diluted Antiferromagnets

In this chapter *Ising models* are studied. They are d-dimensional lattices of Ising spins $\sigma_i = \pm 1$. For the *random-field Ising magnets* (RFIM), they interact ferromagnetically with their nearest neighbors. A local random magnetic field B_i acts on each spin. The system has the following Hamilton function

$$H = -\sum_{\langle i,j \rangle} J\sigma_i\sigma_j - \sum_i B_i\sigma_i \tag{6.1}$$

The sum $\langle i,j \rangle$ runs over pairs of nearest neighbors. $J > 0$ denotes the magnitude of the interaction. Here only simple cubic lattices are considered. Since physical properties depend on the values of the B_i one has to implement *quenched disorder*. This means that the local magnetic fields are drawn independently from a probability distribution $p(B)$ and remain fixed throughout a calculation or simulation. A system with a certain choice of the values $\{B_i\}$ is called a *realization* of the disorder. Since the result of each calculation depends on the choice of the $\{B_i\}$, one has to repeat the process several times and average the results over many realizations. It turns out that for random systems the results vary strongly from realization to realization. Therefore one needs many of them to obtain reliable results, which is very common when studying random systems.

For the distribution of the random fields, usually either a bimodal or a Gaussian distribution are used to draw the random realizations. In the case of a bimodal distribution the local fields take one of two values $B_i = \pm\Delta$ with equal probability. For the Gaussian case arbitrary fields are allowed. The probability density functions are

$$p_{\pm}(B) = 0.5\delta(B-\Delta) + 0.5\delta(B+\Delta) \tag{6.2}$$

$$p_G(B) = \frac{1}{\sqrt{2\pi\Delta^2}} \exp\left(-\frac{B^2}{2\Delta^2}\right) \tag{6.3}$$

The parameter Δ, the width of the distribution, is a measure for the strength of the disorder. For $\Delta = 0$ a pure ferromagnet is obtained. Several attempts to treat the system analytically have been performed [1, 2]. For a review of results obtained by means of computer simulation, see [3].

As already mentioned, there is no direct experimental realization of a random-field system. Instead diluted antiferromagnets in a field are studied. Later we will see that indeed the behavior of the RFIM agrees remarkably with that of the DAFF. Before we present the algorithms to calculate ground states for these models, we will focus a little bit on the DAFF.

The system FeF_2 is an antiferromagnet. By replacing some of the magnetic Fe by non-magnetic Zn one obtains $Fe_xZn_{1-x}F_2$, and a diluted antiferromagnet is created. The crystal structure is that of rutile (TiO_2) (see Fig. 6.1). The only relevant type of interaction is the antiferromagnetic *superexchange* next-nearest-neighbor interaction between the Fe atoms on the body corner and the body center sites. This kind of interaction is generated by the intermediate F atoms. Otherwise the Fe atoms would be ferromagnetic.

At low temperatures this system exhibits many peculiar properties, e.g. it develops frozen domains, as found by neutron-scattering experiment [4]. An overview of experimental results is given in [5].

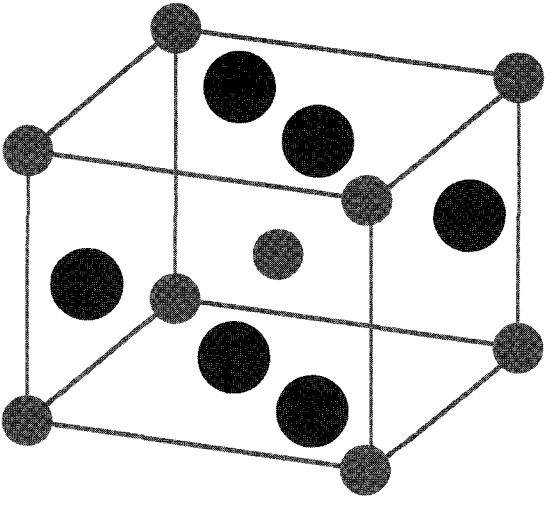


Figure 6.1: Crystal structure of $Fe_xZn_{1-x}F_2$. Small circles denote the Fe/Zn sites while the large circles are the F sites. The eight corner sites (Fe/Zn) are shared by 8 lattice cells, while the body-center site is not shared. The upper and lower pairs of F sites are shared by two cells, while the left and right sites are not shared. All together there are 2 Fe/Zn sites and 4 F sites per lattice cell.

For diluted antiferromagnets with a high anisotropy, like $Fe_xZn_{1-x}F_2$, at low temperatures, the spins, i.e. the magnetic Fe moments, are usually oriented along one axis (the c axis of the crystal). Thus, they can take only two directions, called "*up*" and "*down*". It is assumed this the reason for that anisotropy is a slight deviation of the crystal structure from the pure cubic symmetry.
The Ising model, which has been presented above, is very well suited to describe such type of system. For the case of a DAFF, the spins interact antiferromagnetically with their neighbors. Please note that only the strong interactions are considered in the following model. So the F atoms are not considered here, since they do not contribute directly to the magnetic behavior. As a consequence, a simple cubic crystal structure is sufficient. To describe the dilution, i.e. the fact that not all Fe/Zn sites are occupied by magnetic Fe atoms, a second variable $\epsilon_i = 0, 1$ is introduced. A non-magnetic Zn site is represented by $\epsilon = 0$ while $\epsilon = 1$ holds for a site occupied by a spin (Fe). Additionally, an external magnetic field B can act on the system. Therefore, the energy of a diluted antiferromagnet in a field is given by

$$H = \sum_{\langle i,j \rangle} J\epsilon_i\epsilon_j\sigma_i\sigma_j - B\sum_i \epsilon_i\sigma_i. \tag{6.4}$$

The strength of the interaction is denoted with $J > 0$. The sum $\langle i, j \rangle$ runs over pairs of nearest neighbors. Here only simple cubic lattices are considered. Like the random-field model, diluted antiferromagnets exhibit quenched disorder. Each realization is characterized by a set $\{\epsilon_i\}$ of independent random numbers, here we will consider $\epsilon_i = 0, 1$ with equal probability.

What can we expect for the behavior of this model? In zero field and with zero temperature the ground state, i.e. the state with the lowest energy, is taken. Therefore, in (6.4) configurations $\{\sigma_i\}$ are favorable where neighboring spins take different orientations, because their contribution to the energy $J\sigma_i\sigma_j = -J$ turns out to be negative. It means that, apart from the fact that not all lattice sites are occupied, the system can be divided into two sublattices, which penetrate each other in checkerboard fashion. On one of these sublattices all spins point in one direction (say up), while the spins of the other sublattice are oriented in the opposite way. In Fig. 6.2 an example of a $d = 2$ dimensional diluted antiferromagnet at $B = T = 0$ is shown. In higher dimensions the model behaves similarly, but it is more difficult to draw.

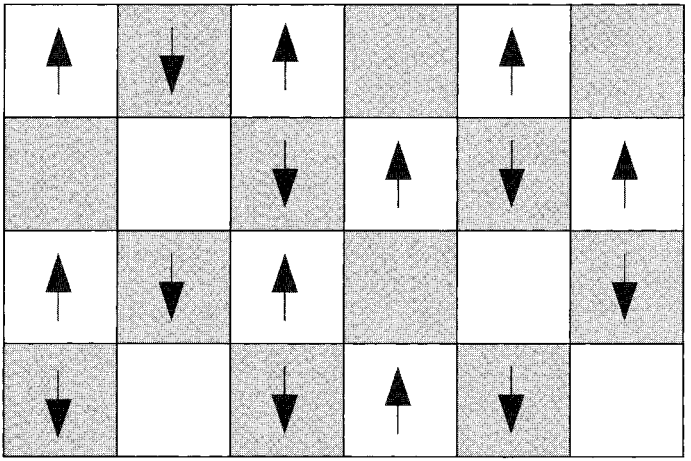


Figure 6.2: Two-dimensional diluted antiferromagnet without external field. The spins are indicated by arrows. Shown is a ground state $T = 0$. Please note that the configuration which is obtained by reversing all spins is a ground state as well.

By increasing the temperature, the DAFF is driven away from the ground state. This means the spins start to fluctuate thermally. When some critical temperature is reached the system becomes paramagnetic (PM). Also by increasing the magnitude of the external magnetic field, the antiferromagnetic order is disturbed. For large values of B, especially if $|B| > 2dJ$, all spins point in the direction of the field (at $T = 0$).

These expectations can be recovered in the schematic phase diagram of the three-dimensional DAFF model (50% dilution) which is shown in Fig. 6.3. It was measured by means of Monte Carlo simulations at finite temperature T, see Refs. [6, 7]. The

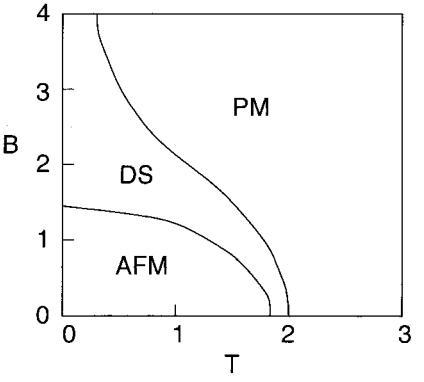


Figure 6.3: Schematic phase diagram of the DAFF in the B-T plane, obtained by MC simulation [6]. Three regions can be identified: antiferromagnetic (AFM), domain state (DS) and paramagnetic (PM).

ordered phase for low temperatures and low fields was established by evaluating the staggered magnetization

$$a = \frac{1}{N} \sum_i (-1)^{x+y+z} \sigma_i \,. \tag{6.5}$$

Here, x, y, z are the spatial coordinates of spin i. This order parameter accounts for the fact that in the presence of order the two penetrating sublattices have opposite magnetizations. Therefore $a = 1$ holds for antiferromagnetic order on a cubic lattice. The transition of the so called *domain state* (DS) to the disordered phase is established in the following way: for a given field B the system is initialized in an ordered state at $T = 0$ and the MC simulation starts. Then, slowly the temperature is increased, here up to $T = 2.5$. This process is called field heating (FH). The temperature is then slowly decreased again (FC=field cooling). Now the system is not able find the way back to the starting configuration. This can be seen by recording the staggered magnetization. At some temperature $T_{irr}(B)$ the FH and FC curves begin to deviate from each other. This is called an onset of *irreversibility*. The line which separates the PM and the DS regions in Fig. 6.3 is just T_{irr} as a function of B. Please note that the value of $T_{irr}(B)$ defined in this fashion depends on the dynamics, i.e. on the heating/cooling rate. A more detailed study has shown that the DS region can be characterized by large fractal domains penetrating the system.
The behavior of true diluted antiferromagnets agrees very well with these computer simulations. But it is relatively hard to treat this model by means of an analytical approach. As a consequence random-field systems are usually studied. It can be shown fairly easily that quite often a DAFF can be mapped onto an RFIM. The mapping works for example in the case of a simple cubic (or square) lattice. One performs a *gauge transformation* by introducing new spin variables $\sigma_i' = (-1)^{x+y+z}\sigma_i$, where x, y, z are the spatial coordinates of spin i. This transformation multiplies in a

checkerboard fashion every second spin with minus one, thus all bonds become ferromagnetic. The resulting Hamiltonian describes a diluted ferromagnet with staggered field $B_i = (-1)^{x+y+z}B$. Please note that this kind of transformation does not work for all lattice types, it fails for example for triangular lattices. An example of such transformation can be found in Sec. 9.2.
As already mentioned, random-field systems are easier to treat analytically. For this reason, most of the theoretical research has focused on this model. We finish this section by stating the expectations about the basic behavior of the model. With low temperatures and low fields the system exhibits a ferromagnetic long range order. For large temperatures the system becomes paramagnetic. By increasing the strength Δ of the random fields, the spins tend to be oriented along the direction of the field. If $|B_i| > 2dJ$ spin i is fixed (at $T = 0$). But at zero temperature for intermediate values of the field a peculiar behavior of the RFIM appears, similar to the DS phase of the diluted antiferromagnet. This region of the phase diagram can be studied by means of ground-state calculations. The basic idea of the method for obtaining the ground states is explained in the next section.

6.2 Transformation to a Graph

Now it is shown that for a given RFIM an equivalent network can be constructed such that the maximum flow through the network is equal to the ground-state energy of the RFIM. This transformation was introduced by Picard and Ratliff in 1975 [8]. Additionally, the orientations of the spins in a ground state can be easily constructed from the flow values the network takes. In the case where the ground state is degenerate, all ground states can be obtained by a subsequent calculation. This case is treated in a Sec. 6.5.
Please note that there are other systems, where the ground-state calculation can be mapped onto maximum-flow problems in networks. Examples are interfaces in random elastic media [9, 10] and fracture surfaces in random fuse networks [11].
It is more instructive to start with a network and to show how it can be transformed into an RFIM. Let $N = (G, c, s, t)$ a network, where $G = (V, E)$ is a directed graph which has $n + 2$ vertices, $c : V \times V \to \mathcal{R}_0^+$ are the capacities of the edges $(i, j) \in E$, with $c(i, j) = 0$ if $(i, j) \notin E$. The vertices $s, t \in V$ are the source and the sink of the network, respectively. For convenience, we use $V = \{0, 1, \ldots, n, n + 1\}$ where $s \equiv 0$ and $t \equiv n + 1$, and the notation $c_{ij} \equiv c(i, j)$. The vertices in $V \setminus \{s, t\}$ are called *inner* vertices. Edges connecting only inner vertices are also called *inner edges*.
As we have seen, a network can be interpreted as a system of pipes connecting the source with the sink. Now, assume that the network should be divided into two pieces in a way that the source and the sink are separated. You can think of a pipeline system and a group of terrorists who like to prevent the oil from being transported to the refinery. Mathematically, this separation is a (s, t)-*cut* $(S, \overline{S})$ which has the following properties:

- $S \cup \overline{S} = V$
- $S \cap \overline{S} = \emptyset$
- $s \in S$
- $t \in \overline{S}$

Usually, we denote the elements of S *left* and the elements of $\overline{S}$ *right* of the cut. In Fig. 6.4 an example network with 5 nodes and 6 edges is shown.

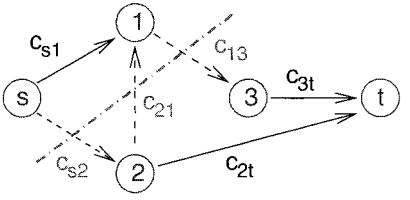


Figure 6.4: A graph with 3 inner vertices $1, 2, 3$. A cut $(S, \overline{S}) = (\{s, 1\}, \{2, 3, t\})$ is represented by a dashed-dotted line. The capacity of the cut is $C(\{s, 1\}, \{2, 3, t\}) = c_{s2} + c_{13}$). The edge $(1, 2)$ does not contribute to the cut, because it is oriented in the opposite direction to the cut.

Since the amount of work (or TNT) to remove a pipeline grows with its capacity, the terrorist are interested in the capacity of the pipes they have to remove. These are exactly the pipes which go from S to $\overline{S}$, that means from left to right. We say these pipes/edges *cross* the cut. Edges in the opposite direction cannot contribute to a flow from the source to the sink, thus they are disregarded. The sum of the capacities of these edges is called the *capacity* $C(S, \overline{S})$ of the cut

$$C(S, \overline{S}) = \sum_{i \in S, j \in \overline{S}} c_{ij} \,. \tag{6.6}$$

In the example presented in Fig. 6.4 the capacity of the cut is $C(S, \overline{S}) = c_{s2} + c_{13}$. The edge $(1, 2)$ does not contribute, since it leads from right to left.
The basic idea to represent a Hamiltonian by a network is to represent a cut by a binary vector $\underline{X} = (x_0, x_1, \ldots, x_n, x_{n+1})$ with

$$x_i = \begin{cases} 1 & i \in S \\ 0 & i \in \overline{S} \end{cases} \tag{6.7}$$

For an (s, t)-cut the values of $x_0 = 1$ and $x_{n+1} = 0$ are fixed. An edge (i, j) goes from left to right of the cut only if $x_i = 1$ and $x_j = 0$. Therefore, the formula for the capacity of a cut can be rewritten in the following way (all sums run from 0 to $n + 1$):

$$\begin{aligned} C(\underline{X}) \equiv C(S, \overline{S}) &= \sum_{i,j} x_i (1 - x_j) c_{ij} \\ &= -\sum_{i,j} c_{ij} x_i x_j + \sum_i \left(\sum_j c_{ij} \right) x_i \end{aligned} \tag{6.8}$$

Here, it can already be seen that the structure of the formula resembles the energy functions for the random-field system: it consists of a linear and a quadratic term. To make the correspondence complete, the values of the source and the sink $x_0 = 1, x_{n+1} = 0$ have to be inserted, additionally $c_{ii} = 0 \quad \forall i$ is assumed. For didactic reasons, the capacities involving the source 0 or the sink $n+1$ are written as extra terms. One obtains:

$$\begin{aligned} C(x_1, \ldots, x_n) &= -\sum_{i,j=1}^{n} c_{ij} x_i x_j + \sum_{i=1}^{n} \left(-c_{0i} + c_{i,n+1} + \sum_{j=1}^{n} c_{ij} \right) x_i \\ &\quad + \sum_{i=1}^{n} c_{0i} + c_{0,n+1} \end{aligned} \tag{6.9}$$

The last slight difference to a Hamiltonian is that the capacity is given in terms of zero-one variables $x_i = 0, 1$ while a spin may take $\sigma_i = \pm 1$. Thus, one can identify $x_i = 0.5(\sigma_i + 1)$. In the Hamiltonian the sum runs over all bonds, while for $C(x_1, \ldots, x_n)$ each pair of vertices appears twice in the quadratic term. Thus, the identity $\sum_{ij} c_{ij} = \sum_{i<j} (c_{ij} + c_{ji})$ is used. In the final formula all sums run from 1 to n:

$$\begin{aligned} C(\sigma_1, \ldots, \sigma_n) &= -\sum_{i<j} \frac{1}{4} (c_{ij} + c_{ji}) \sigma_i \sigma_j \\ &\quad + \sum_{i} \left(-\frac{1}{2} c_{0i} + \frac{1}{2} c_{i,n+1} + \frac{1}{4} \sum_{j} (c_{ij} - c_{ji}) \right) \sigma_i \\ &\quad + \frac{1}{4} \sum_{i<j} (c_{ij} + c_{ji}) + \frac{1}{2} \sum_{i} (c_{0i} + c_{i,n+1}) + c_{0,n+1} \end{aligned} \tag{6.10}$$

This formula represents a system which has $n+2$ vertices: one source, one sink and n (inner) vertices, one for each spin σ_i. The model is slightly more general than the system represented by the Hamiltonians in (6.1) and (6.4). Thus, formula (6.10) can be compared with an energy function which generalizes both Hamiltonians ($\epsilon_i = 0, 1$):

$$H = -\sum_{i<j} J_{ij} \epsilon_i \epsilon_j \sigma_i \sigma_j - \sum_{i} B_i \epsilon_i \sigma_i \,. \tag{6.11}$$

This is a diluted random-field system with bonds of variable strengths. Another kind of model which falls into this class is the *random-bond ferromagnet* [12, 13], were the system is not diluted nor does it have a random field, but the strengths of the ferromagnetic bonds are drawn randomly.

Now we want to choose the capacities c_{ij} in such a way that the system (6.11) is represented by a network. By comparison with (6.10) we find $c_{ij} + c_{ji} = 4 J_{ij} \epsilon_i \epsilon_j$ for $i, j \in 1, \ldots, n$. Since for a network only non-negative capacities are allowed, the bond values have to be non-negative as well. Both c_{ij} and c_{ji} appear in the equality, so there is some freedom of choice. We chose that non-zero capacities shall be present only for edges (i, j) with $i < j$. The reason for this choice is that we have only edges

going from vertices with smaller number to vertices with higher number. This allows some algorithms to be implemented in a way that they run faster. For the capacities we obtain $i, j \in 1, \ldots, n$:

$$c_{ij} \equiv \begin{cases} 0 & \text{if } i \geq j \\ 4J_{ij}\epsilon_i\epsilon_j & \text{else} \end{cases} \tag{6.12}$$

Now we are left with the capacities $c_{0i}, c_{i,n+1}$ $(i = 1, \ldots, n)$ and $c_{0,n+1}$. Next, the linear terms in (6.10) and (6.11) have to be compared. Defining

$$w_i \equiv -2B_i\epsilon_i - \frac{1}{2}\sum_j (c_{ij} - c_{ji}) \tag{6.13}$$

we obtain $-c_{0i} + c_{i,n+1} = w_i$. The value of w_i may be positive or negative. Again, all capacities have to be non-negative. Therefore, we get

$$\begin{aligned} c_{0i} &\equiv 0 \quad c_{i,n+1} \equiv w_i \qquad \text{if} w_i > 0 \\ c_{0i} &\equiv -w_i \quad c_{i,n+1} \equiv 0 \qquad \text{else} \end{aligned} \tag{6.14}$$

Finally, since the sum of the constant terms in (6.10) must vanish, we obtain

$$c_{0,n+1} \equiv -\frac{1}{4}\sum_{i<j}(c_{ij} + c_{ji}) - \frac{1}{2}\sum_i (c_{0i} + c_{i,n+1}) \tag{6.15}$$

The capacity $c_{0,n+1}$ of the edge may be positive or negative. But in this case it does not matter, since this edge crosses every (s,t)-cut of the network. Consequently, it can be removed from the network at the beginning. Later, after the capacity of a cut has been obtained, the value of $c_{0,n+1}$ has to be added to obtain the energy of the corresponding system.

To summarize, for a system defined by the energy function (6.11) an equivalent network can be constructed by creating a graph which has one vertex for each spin and additionally a source and a sink. The capacities of the edges are chosen by performing the steps given through Eqs. (6.12, 6.14, 6.15). Then, every configuration of the system corresponds to a cut in the network ($\sigma_i = 2x_i - 1$) and the energy $H(\sigma_1, \ldots, \sigma_n)$ is equal to the capacity $C(x_1, \ldots, x_n)$ of the corresponding cut.

Since we are interested in ground states, a minimum of the energy is to be obtained. Consequently, we are looking for a *minimum cut*, that is a cut among all cuts which has minimum capacity. Such a cut cannot be obtained directly, but it is related to the flow going through the network. Each flow must pass the edges crossing an arbitrary cut, especially the minimum cut. Therefore, the minimum cut capacity is an upper bound for the flow. On the other hand, it can be shown that the maximum flow which is possible is indeed given by the capacity of a minimum cut. The proof was found by Ford and Fulkerson in 1956 [14]. Versions of the proof which are more instructive can be found in [15, 16, 17]. Along with the proof a simple method for finding the maximum flow was introduced. After constructing an equivalent network such a technique can be applied to find a ground state of a random-field Ising system. The Ford-Fulkerson algorithm and an extension of it are presented in the next section.

But before we proceed with the algorithms, as an example, we look at a small random-field system, which is shown in Fig. 6.5. There are four spins arranged in a square,

neighbors interact via ferromagnetic interactions of strength $J = 1$. The local fields have the values $B_1 = 0$, $B_2 = -2\Delta$, $B_3 = 2\Delta$, and $B_4 = 0$. Even for this tiny system many different effects, depending on the magnitude of Δ, can be studied.

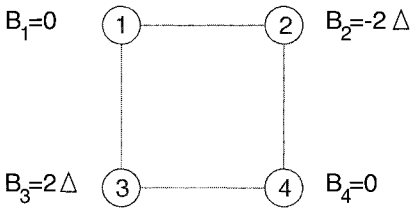


Figure 6.5: A small random-field Ising magnet. It contains 4 spins coupled via ferromagnetic interactions ($J_{12} = J_{13} = J_{24} = J_{34} = 1$). The local fields have the magnitudes $B_1 = 0$, $B_2 = -2\Delta$, $B_3 = 2\Delta$, and $B_4 = 0$.

Now, an equivalent network is constructed. It contains $n = 4$ inner nodes, one for each spin, and additionally one source 0 and one sink $n + 1 = 5$. We start by setting the capacities of the inner nodes. According to (6.12) we get

$$\begin{aligned} c_{12} &= 4J_{12} = 4 \\ c_{13} &= 4 \\ c_{24} &= 4 \\ c_{34} &= 4 \\ c_{ij} &= 0 \text{ for all other cases } 1 \le i, j \le 4 \end{aligned} \tag{6.16}$$

These values do not depend on the strength of the local magnetic fields. But they enter the expressions for the auxiliary values w_i, see (6.13)

$$\begin{aligned} w_1 &= -2B_1 - \frac{1}{2}\sum_s (c_{ij} - c_{ji}) \\ &= -2 \times 0 - \frac{1}{2}(c_{12} + c_{13}) \\ &= -4 \\ w_2 &= -2B_2 - \frac{1}{2}(-c_{12} + c_{24}) \\ &= 4\Delta \\ w_3 &= -4\Delta \\ w_4 &= 4 \end{aligned} \tag{6.17}$$

At first the case $\Delta = 0$ is investigated. Thus, no magnetic field acts on the spins. A ferromagnetically ordered ground state is to be expected, thus all spins point up or all spins point down at $T = 0$. Since each bond contributes an amount of -1 to the energy, the total ground-state energy sums up to $E_0 = -4$. This behavior can be extracted from the corresponding network as well. By setting $\Delta = 0$ in (6.17) we

obtain $w_1 = -4$, $w_2 = 0$, $w_3 = 0$, and $w_4 = 4$. According to (6.14) the capacities of the edges connecting the inner vertices to the source and the sink are

$$\begin{aligned} c_{01} &= 4 & c_{15} &= 0 \\ c_{02} &= 0 & c_{25} &= 0 \\ c_{03} &= 0 & c_{35} &= 0 \\ c_{04} &= 0 & c_{45} &= 4 \end{aligned} \tag{6.18}$$

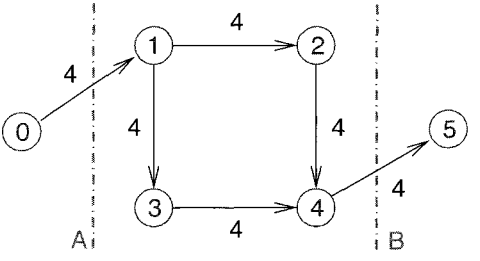


Figure 6.6: The network obtained for the system from Fig. 6.5 for the case $\Delta = 0$. Vertex 0 is the source and vertex 5 the sink. The numbers next to the edges denote their capacities.

The capacity of the phantom edge connecting source and sink evaluates according to (6.15) to $c_{05} = -0.25(4 + 4 + 4 + 4) - 0.5(4 + 0 + 0 + 4) = -8$. The final network is presented in Fig. 6.6. The network allows for two minimum $(0,5)$-cuts, indicated by dot-dashed lines in the drawing. Both cuts have capacity $C(S, \overline{S}) = 4 + c_{0,5} = -4$ which is indeed equal to the ground-state energy obtained above. The cut denoted with an "A" is given by $(S, \overline{S}) = (\{0\}, \{1,2,3,4,5\})$. This means $x_0 = 1$ and $x_1 = x_2 = x_3 = x_4 = x_5 = 1$. Using $\sigma_i = 2x_i - 1$ $\sigma_i = 1$ for $i = 1,2,3,4$ is obtained. Thus, really all spins are oriented in the same direction. The cut "B" $(S, \overline{S}) = (\{0,1,2,3,4\}, \{5\})$ corresponds to the configuration where all spins are pointing down.
In the case $\Delta = 1$ the following values for the capacities are obtained:

$$\begin{aligned} c_{01} &= 4 & c_{15} &= 0 \\ c_{02} &= 0 & c_{25} &= 4 \\ c_{03} &= 4 & c_{35} &= 0 \\ c_{04} &= 0 & c_{45} &= 4 \\ & c_{05} = -12 \end{aligned} \tag{6.19}$$

The resulting network is shown in Fig. 6.7. Now six different minimum cuts are possible. At a first glance the figure might look somewhat complicated. But it is easy to verify that indeed all minimum cuts A to F have capacity $C = 8 + c_{05} = -4$. Please note that only the edges contribute which go from the left side of the cut to the

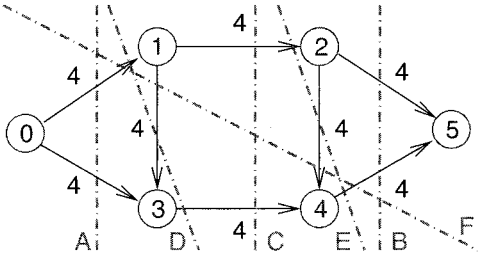


Figure 6.7: The network obtained for the system from Fig. 6.5 for the case $\Delta = 1$. Vertex 0 is the source and vertex 5 the sink. The numbers next to the edges denote their capacities.

right side. Thus, to the cut F, the edges $(0,1)$ and $(4,5)$ contribute to the capacity, but not the edges $(1,3)$ or $(2,4)$. The corresponding spin configurations are drawn in Fig. 6.8. There the small plus and minus signs state the contributions of the single spins/spin pairs to the total ground-state energy. In all 6 cases the ground-state energy is $E_0 = -4$, which is again equal to the capacity of all minimum cuts. The six ground states can be described as follows: for two configurations the system is ordered, all spins either point up or all spins point down. For the two ordered configurations one of the spins 2/3 is oriented against its local field. The remaining four ground states are characterized by the fact that spins 2,3 point in the direction of their local magnetic fields and the remaining spins can be either up or down independently.
The example graphs we have presented here will also be used in the next section to elucidate how the algorithms for calculating maximum flows work.

6.3 Simple Maximum Flow Algorithms

In this section two algorithms for calculating the maximum flow in a network from the source to the sink are given. First a description of the historically first method is given which was invented by Ford and Fulkerson in 1956 [14]. Then it is shown by a simple example that the running time of this algorithm has no polynomial bound in the number of nodes and edges, thus it is not suitable in practice. Finally, a variant of the Ford-Fulkerson algorithm, developed in 1972 by Edmonds and Karp [18] is explained, which indeed has a polynomial time complexity. As usual, the way the algorithms work are illuminated by giving sample runs. A compact description of the methods explained here along with proofs for correctness and time complexity can be found in [19].
Let $N = (G, c, s, t)$ a network, where $G = (V, E)$ is a directed graph which has $n + 2$ vertices, $c : E \to \mathcal{R}_0^+$ are the capacities of the edges $(i, j) \in E$ and $s, t \in V$ are the source and the sink, respectively. Let f be a flow in N, i.e. f_{ij} denotes the flow from vertex i to vertex j. Please remember that $f_{ij} = -f_{ji}$ and the that flow is conserved

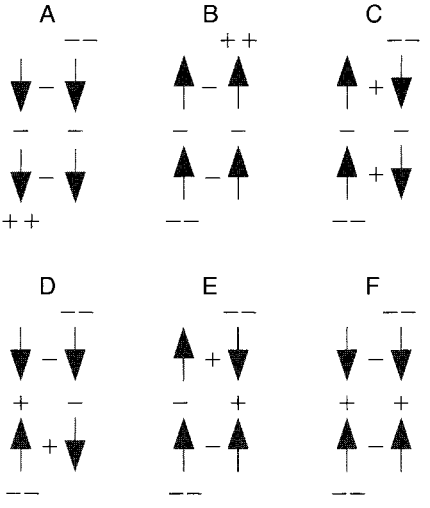


Figure 6.8: Ground states of the RFIM shown in Fig. 6.5 for the case $\Delta = 1$. The system has 6 ground states with the same energy $E_0 = -4$. The arrows indicate the orientations of the spins. The plus and minus symbols represent contributions to the ground-state energy.

in all vertices except the source and the sink. It is assumed that all capacities are non-negative integer (or rational) values. The aim is to find the maximum flow from the source to the sink.

The basic idea of the Ford-Fulkerson algorithm is to start with an empty network and to try to push additional flow from the source to the sink. This is done by searching for paths along which the flow can be increased, they are called *augmenting paths*. If no such path is found, the algorithm stops and the maximum flow has been found.

But for each single edge, the flow through the edge is not always increased or remains the same. Sometimes it is necessary to decrease the flow in some edges, to increase the total flow through the network. This case appears when a certain amount of flow is redirected within the graph. To treat this redirection process efficiently the notion of a *residual network* is useful. The residual network R (often called *residual graph*) for network N and flow f has the same vertices as the graph G. It contains edges with nonzero capacities r_{ij} whenever in G with a given flow f it is possible to increase the flow along edge (i,j). Please note that it is also possible to increase a negative flow in (i,j) by decreasing the flow in the reverse edge (j,i). Formally R is defined as follows: $R = (G^{'}, r, s, t)$, $G^{'} = (V, E^{'})$ with

$$r_{ij} \equiv c_{ij} - f_{ij} \,. \tag{6.20}$$

The graph $G^{'}$ contains directed edges (i,j) for all nonzero capacities $r_{ij} > 0$. Please note that in the residual network two vertices (i,j) may be connected by two edges (i,j) and (j,i).

The Ford-Fulkerson algorithm reads as follows

algorithm ford-fulkerson(N)
begin
  Initially set $f_{ij} := 0, f_{ji} := 0$ for all $(i, j) \in E$;
  do
    construct residual network R with capacities r_{ij};
    if there is an augmenting path from s to t in G' **then**
    begin
      Let $r_{\min}$ the minimum capacity of r along this path;
      Increase the flow in N along the path by a value of $r_{\min}$;
    end
  until no such path from s to t in G' is found;
end

For each augmenting path the edge with the smallest capacity is a bottleneck. This is the reason why the flow along the path can be increased only by the amount of the minimum capacity $r_{\min}$. The augmenting path can be constructed using a breadth-first search. I.e. beginning at the source iteratively neighboring vertices are visited which have not been visited before. During this search at each vertex i the predecessor in the actual path and the minimum residual capacity $r_{\min}(i)$ along the path up to i are stored. In this way each vertex which is connected to the source is visited exactly once. If an augmenting path is found, i.e. if the sink has been visited during the breadth-first search, the flow can be augmented directly by starting at the sink. Iteratively the predecessor is visited and the flow increased by $r_{\min}(t)$ until the source is reached again.
Please note that the algorithm may not converge to the true maximum flow if the capacities are irrational, an example is given in [20].

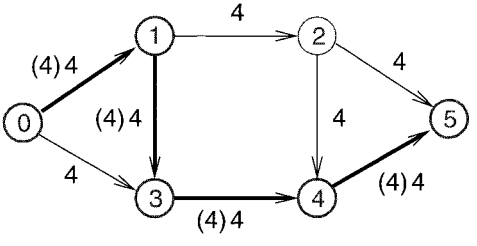


Figure 6.9: Initial residual network of small random-field system. An augmenting path is highlighted. The values next to the edges denote the residual capacities. The values in parentheses state the amount of flow which can be pushed along the augmenting path.

As an example we will investigate how the algorithm calculates the maximum flow through the network given in Fig. 6.7. At the beginning the flow is empty, so the

residual network is equal to the original network. A possible augmenting path from the source 0 to the sink 5 is given by the vertices 0, 1, 3, 4, 5. Each edge along the path has (residual) capacity 4, thus it is possible to increase the flow along the path by this value. The residual graph is is shown in Fig. 6.9, the augmenting path is highlighted. The values in parentheses next to the edges state the amount of additional flow which can be pushed through each edge along the augmenting path. In this case the resulting original network with the flow after the augmentation looks the same.

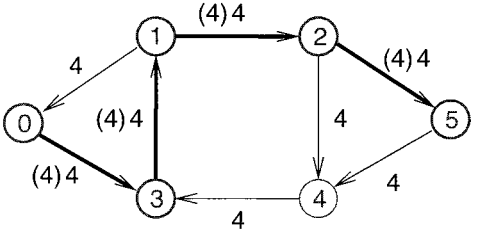


Figure 6.10: Residual network of small random-field system at the second iteration. Now, only one augmenting path exists (highlighted). Again, the values next to the edges denote the residual capacity. The values in parentheses state the amount of flow which can be pushed along the augmenting path.

For the second iteration, again the residual network is has to be calculated. Now, for example in edge $(0,1)$ the flow $f_{01} = 4 = c_{01}$ is present. This means that the residual capacity in direction $(0,1)$ is $r_{01} = c_{01} - f_{01} = 0$, but for the reversed direction a value of $r_{10} = c_{10} - f_{10} = 0 - (-4) = 4$ is obtained. The complete residual network is shown in Fig. 6.10. In this case only one augmenting path is feasible: $0, 3, 1, 2, 5$. The capacity of this path is 4. Please note that by augmenting the flow through edge $(3,1)$ in the residual network, this results in a decrease of the flow through edge $(1,3)$ in the original graph. The resulting total flow for the graph is displayed in Fig. 6.11. Now both edges $(0,1)$ and $(0,3)$ are satisfied. As a consequence, in the corresponding residual network $r_{01} = 0$ and $r_{03} = 0$, i.e. no edge leaves the source. This means that no augmenting path exists. Consequently, the maximum flow has been found.
How fast is the Ford-Fulkerson algorithm? Since in each step the flow is increased at least by the amount of one (if the capacities are integers, as assumed), a time bound of $O(f_{\max})$ is obtained, where $f_{\max}$ is the maximum flow. Unfortunately, it is not possible to find a polynomial in the number of vertices and the number of edges which bounds the time complexity of the method. The reason is that the way an augmenting path is chosen is not determined in any way. The effect of this can be demonstrated by a simple example. Consider the graph in Fig. 6.12. It is similar to the preceding example graph, but the capacities are different. Assume that the augmenting path is $0, 1, 3, 4, 5$ (highlighted). Then the residual capacity of this path is 1, by this amount the flow through the graph is increased. The resulting residual network is shown in 6.13. Now, an augmenting path is given by $0, 3, 1, 2, 5$. Again, the flow is increased by

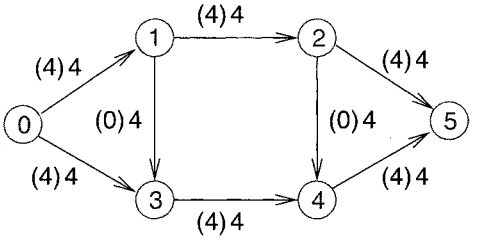


Figure 6.11: Final flow in network of small random-field system. The values in parentheses state the amount of flow which flows through each edge.

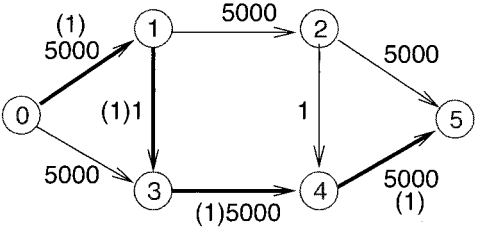


Figure 6.12: A tiny network where the Ford-Fulkerson algorithm may spend much time to calculated the maximum flow. Displayed is the residual graph of the first iteration. The resulting flow in the network after the first iteration is represented by the bold edges and the numbers in parentheses.

one unit and the flow in Fig. 6.14 is obtained.

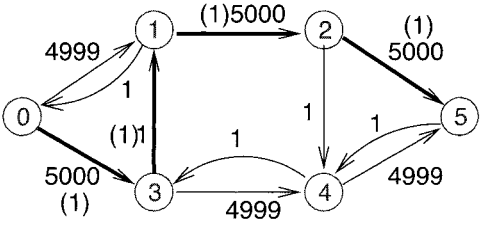


Figure 6.13: The residual network of the tiny network after one iteration of the Ford-Fulkerson algorithm.

Obviously, by always using these bad choices of the augmenting paths, the Ford-Fulkerson algorithm can be iterated for 10000 times, each time the flow is increased by one unit. On the other hand, it is possible to calculate the maximum flow within two iterations if the augmenting paths $0, 1, 2, 5$ and $0, 2, 4, 5$ are chosen.

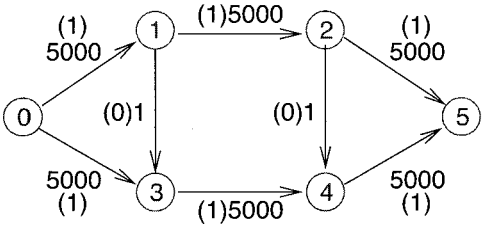


Figure 6.14: The tiny network after the flow has been increased two times by the Ford-Fulkerson algorithm. The embraced number give the flow through the edges, the other numbers state the capacities of the edges.

This unfavorable behavior is avoided by an extension of the algorithm, which was presented by Edmonds and Karp in 1972 [18]. The basic idea is to choose an augmenting path which has the shortest distance from the source to the sink, where each edge counts with distance one[1]. Algorithms for finding the shortest paths can be found in Chap. 4. It can be shown that indeed this algorithm has a polynomial worst-case time complexity of $\Theta(|V||E|^2)$, which is in fact independent of the capacities of the edges. Since the networks we are considering here are lattices, each vertex has a maximum number of neighbors, i.e. the number of edges is proportional to the number of vertices. This results in a running time of $\Theta(|V|^3)$. Modern algorithms are even faster. One is presented in the next section. Another approach, the *push-relabel* method, is described in Refs. [21, 22, 23].

6.4 Dinic's Method and the Wave Algorithm

The efficiency of the method of Edmonds and Karp can be further increased by considering many augmenting paths in parallel as proposed by Dinic [24]. A comprehensive description of a simpler version can be found in [19]. This can be additionally speed up by applying a method described in [25]. For three-dimensional RFIM systems with bimodal couplings the corresponding networks can be treated, as found empirically [26], in $\Theta(N^{4/3})$ time, where N is the number of spins (remember $|V| = N + 2$).
The basic idea is to augment the flow through a network along *several* shortest paths in parallel. The actual algorithm comprises of several steps. First the *level network* is constructed, which is explained below. Using the level network a *blocking flow* is calculated, that is a flow which cannot be increased by adding flow along some paths. The blocking flow is calculated with Tarjan's *wave-algorithm* [19] in the version given by Träff [25]. The description of this algorithm will be the central part of this section. The whole algorithm may contain several iterations beginning at the construction of the level network, which is now explained.

[1] Another version of the algorithm always chooses the augmenting path with the maximum (residual) capacity. This results in a time complexity of $\Theta(|E| \log c_{\max})$, where c is the maximum capacity of all edges.

The starting point for the level network[2] is the residual network R, as defined previously. The basic idea is the same as for the algorithm by Edmonds and Karp: the level network contains all shortest paths from the source to the sink. In contrast to the previous method, here the flow can be increased along several paths in parallel. This is the key idea to obtaining an efficient algorithm. Let $level(i)$ be the length of the shortest path in the residual network R from the source to vertex i. The level can be constructed using a breadth-first search, i.e. the algorithm given in the preceding section. This introduces the so called *topological order* of the vertices, which is just the order the vertices are visited during the search. Consequently, for each vertex all its predecessors have lower topological numbers while all successors have higher numbers. After the breadth-first search, the level network $LN(R)$ is obtained by removing all edges (i, j) from R which do not fulfill $level(j) = level(i) + 1$, thus only edges between neighboring levels are left. Please note that the level network is unique although the topological order may not be unique. We denote the capacities of the level network with $\hat{c}(i, j)$.
Since the methods presented here are more involved than the techniques presented previously, we need a slightly more complicated network to illustrate the algorithms. It is given in Fig. 6.15. The graphical representation clearly shows the levels of the vertices. For example $level(2) = 1$, $level(7) = 3$ and $level(12) = 5$. The vertices are already numbered according an arbitrary topological order.

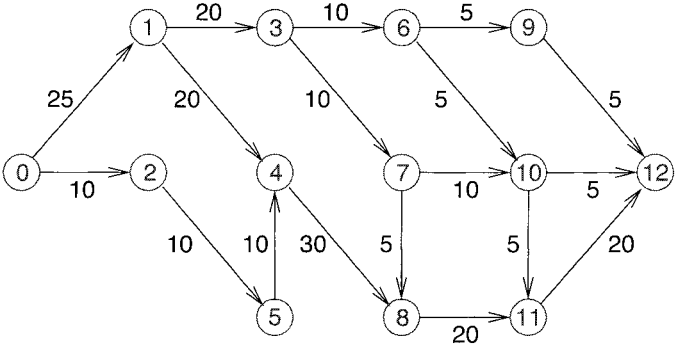


Figure 6.15: An example network for demonstrating the wave algorithm.

The resulting level network does not contain the edge $(2, 6)$ since $level(2) = level(6)$. For the same reason the edges $(7, 10)$ and $(8, 11)$ are not included. Now, there may be "dead ends" in the graph, i.e. vertices which are not connected to the sink. Thus, they cannot contribute to the maximum flow. These vertices are removed in the next step of the algorithm. For the example graph vertices 1 and 3 are removed. The outcome is displayed in Fig. 6.16.
The reader may already have noticed the values written next to the vertices in Fig. 6.16. They are due to the idea of Träff who introduced capacities $p(i)$ also for the

[2]The level network is often called level graph.

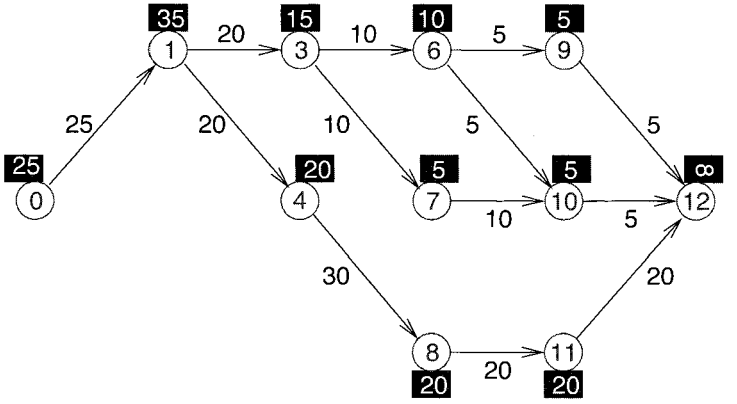


Figure 6.16: The resulting level network contains only edges (i, j) with $level(j) = level(i) + 1$. Dead ends are removed. The values next to the vertices are the vertex capacities introduced by Träff.

vertices i. These capacities give upper bounds for the maximum amount of flow which can leave a vertex i due to the capacities of all succeeding edges on paths to the sink. Since the sink itself has no successors, its capacity can be set to infinity. All its predecessors allow for a flow which is bounded by the capacities of the edges to the sink. For all other vertices we have to realize that each edge cannot carry more flow than that given by its capacity and given by the capacity of the vertex at the end of the edge. Therefore, the amount of flow leaving a vertex cannot exceed the sum of the bounds calculated in this way for all outgoing edges. Since the capacity of a vertex is determined by its succeeding vertices, the values of $p(i)$ can be calculated in a recursive way, by starting from the sink and visiting all vertices in *reversed* topological order:

$$p(N+1) = \infty \tag{6.21}$$

$$p(i) = \sum_j \min\{\hat{c}(i,j), p(j)\} \quad \forall i \tag{6.22}$$

The resulting values for the sample graph are also presented in Fig. 6.16. Please note that indeed $p(i)$ is only an upper bound. Vertex 3 has $p(3) = 15$, while it is not actually possible to have a flow through this vertex which is larger than $10 = c(9,12) + c(10,12)$. Now the main part of the method will be explained, the *wave algorithm*. The target is to obtain a blocking flow. The algorithm starts by calculating a *preflow* $\{\hat{f}(i,j)\}$. It is a kind of flow but the flow may not be conserved at the inner vertices. The *excess* $e(i) = \sum_j \hat{f}(i,j)$ states the amount of flow which is lost $[e(i) < 0]$ or created $e(i) > 0$ at vertex i. A vertex with $e(i)$ is called *overflowing* while it is called *balanced* if $e(i) = 0$. Thus, a flow is a preflow where all inner vertices are balanced. The wave

algorithm starts $\hat{f}(0,j) = c(0,j)$, i.e. with a preflow where all edges leaving the source are saturated [3], while all other edges are empty. The maximum flow can certainly not be larger than $\sum_j c(0,j)$. During the execution of the algorithm, for all vertices it is recorded whether they are *blocked* or not. A vertex $i \neq (N+1)$ is called blocked if it is not possible to push additional flow towards the sink through i. Initially all vertices are not blocked.

After this initialization the wave algorithm starts. It consists of *forward waves* and *backward waves*. Within a forward wave, the preflow is pushed into the direction of the sink as far as possible. If some part of the preflow stops somewhere, the corresponding vertex is blocked for further iterations. Within a backward wave, flow which has been blocked is pushed in the opposite direction. Later, during the next forward wave, pushing it along other paths it attempted. Both steps, forward wave and backward wave, are iterated until all excess flow has disappeared. This means that at the end the preflow has been turned into a flow[4]. The two main steps of the wave algorithm read in detail as follows:

- **forward wave**:

  All vertices i are visited in topological order. All outgoing edges (i,j) are treated which are not satisfied and where the end vertex j is not blocked. The preflow through (i,j) is increased by the value of $\min\{\hat{c}(i,j) - \hat{f}(i,j), p(j) - e(j), e(i)\}$. This means the preflow cannot be increased by more than the current residual capacity, by not more than vertex j can take and by not more than available as excess. The excess $e(i)$ is decreased and $e(j)$ is increased by that amount.

  If it is not possible to balance vertex i, i.e. if not all of the excess flow can be pushed forward, then vertex i is blocked. So in subsequent iterations pushing further flow through this edge is not tried.

- **backward wave**:

  All vertices j except the sink are visited in reversed topological order. If j is blocked all incoming edges (i,j) are treated: the preflow $\hat{f}(i,j)$ is decreased by $\min\{\hat{f}(i,j), e(j)\}$ as long as the excess $e(j) > 0$. The excess $e(i)$ of the predecessor is increased by the same amount.

  This means, for each blocked node, that all positive excess is pushed backwards. In this way the flow is moved towards the source until a non-blocked vertex is reached. There the flow may be directed along another path during the next forward wave.

[3] Since $\hat{f}(i,j) = -\hat{f}(j,i)$, $\hat{f}(j,0) = -c(0,j)$. But the negative values are not used within the wave algorithm, they are considered only for the calculation of the residual network.

[4] Please note that by the wave algorithm a flow which has already found a path to the source will never be redirected during the current waves. So the flow obtained in this way is indeed a blocking flow and may not be a maximum flow. Nevertheless, the multiple execution of the whole procedure (see later) guarantees that at the end a maximum flow is obtained.

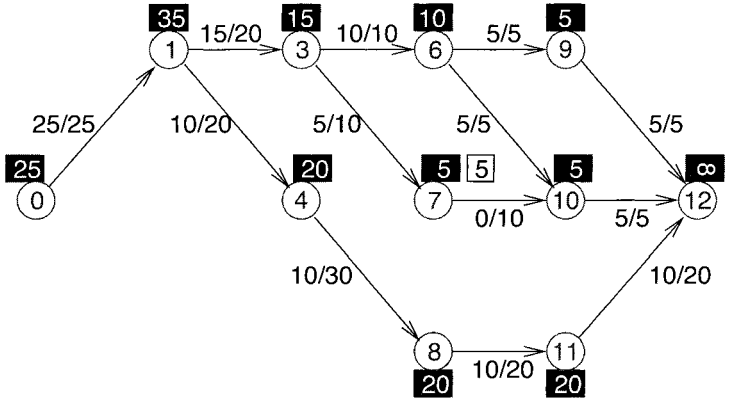


Figure 6.17: The level network after the first forward wave. Vertex 7 is blocked. The value next to the vertex capacity $p(7)$ is the excess $e(7) = 5$. All other vertices are not blocked, so they have excess zero. The values at the edges have the format $\hat{f}(i,j)/\hat{c}(i,j)$.

Example: Wave algorithm

For the sample graph the preflow is initialized to $\hat{f}(0,2) = 25$. All excess values are zero except $e(2) = 25$. Now the first forward wave will be explained in detail. The reader should write the values of the preflow and the current excess values with a pencil (!) on the graph while simulating the algorithm.

- First vertex $i = 1$ is treated. For edge $(i,j) = (1,3)$ the flow is increased by $\min\{\hat{c}(i,j) - \hat{f}(i,j), p(j) - e(j), e(i)\} = \min\{20 - 0, 15 - 0, 25\} = 15$. Consequently $e(3) = 15$ and the excess for vertex 1 is decreased to $e(1) = 10$. Then edge $(1,4)$ is considered. The remaining flow can be directed along that edge, consequently $\hat{f}(1,4) = 5$, $e(4) = 10$ and $e(1) = 0$.
- Next vertex $i = 3$ is treated. Edge $(3,6)$ can carry flow of amount $\hat{f}(3,6) = 10$, so $e(6) = 10$ and now $e(3) = 5$. The remaining excess can travel along edge $(3,7)$, resulting in $\hat{f}(3,7) = 5$, $e(7) = 5$ and $e(3) = 0$ after completing the treatment of vertex 3.
- The excess of vertex $i = 4$ can be passed on completely to vertex 8, resulting in $\hat{f}(4,8) = 10$, $e(4) = 0$ and $e(8) = 10$.
- Next vertex $i = 6$ is considered. The excess flow $e(6) = 10$ can be completely partitioned among the successors $j = 9, 10$, resulting in $e(6) = 0$, $e(9) = 5$, $e(10) = 5$, $\hat{f}(6,9) = 5$ and $\hat{f}(6,10) = 5$.
- Vertex $i = 7$ is the first one, which will be blocked: For edged $(i,j) = (7,10)$ the flow can be increased by $\min\{\hat{c}(i,j) - \hat{f}(i,j), p(j) - e(j), e(i)\}$

$= \min(10 - 0, 5 - 5, 5) = 0$. Therefore, the excess of vertex 7 remains unchanged, i.e. $e(7) = 5$.

- For vertex $i = 8$ the flow is just passed on to vertex 11, thus we get $\hat{f}(8, 11) = 10$, $e(8) = 0$ and $e(11) = 10$.
- In a similar way the flow is moved forward in vertices $9, 10$ and 11. So finally we obtain $\hat{f}(9, 12) = 5$, $\hat{f}(10, 12) = 5$, $\hat{f}(11, 12) = 10$, $e(9) = 0$, $e(10) = 0$, $e(11) = 0$ and $e(12) = 20$.

The resulting preflow system is shown in Fig. 6.17. Only vertex 7 is blocked, it is marked by a box containing the value of the excess.

During the backward wave only vertex 7 is treated. The flow is moved back to vertex 3 which then has excess $e(3) = 5$, while now $e(7) = 0$ □

After completing one forward and one backward wave the vertex capacities are dynamically adjusted by visiting all vertices again in reversed topological order. Therefore, they represent only non-used capacities:

$$p(i) = \sum_j \min\{\hat{c}(i, j) - \hat{f}(i, j), p(j)\} \tag{6.23}$$

Example: Readjusting of the vertex capacities

The readjustment of the vertex capacities results in the network shown in Fig. 6.18. Clearly, now most of the paths are not able to carry additional flow.

During the next forward wave, only vertex 3 is treated. It is not possible to move the flow to either of its both successors $6, 7$, consequently vertex 3 is marked as blocked.

Within the next backward wave, the excess of vertex 3 is pushed back to vertex 1, i.e. $e(1) = 5$ and $e(3) = 0$. The following recalculation of the vertex capacities does not change anything.

The next forward wave moves the excess of vertex 1 via the vertices $4, 8, 11$ just to the sink. Now all excess has disappeared, the preflow has been turned into a flow. The final situation is shown in Fig. 6.19

□

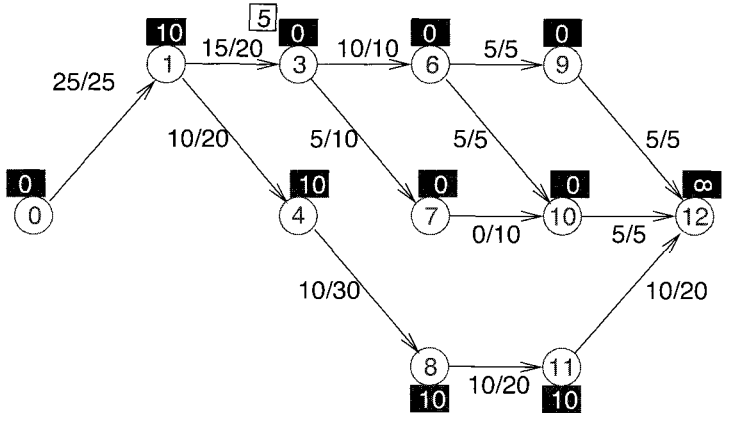


Figure 6.18: The level network after the recalculation of the vertex capacities. Vertex 3 has excess $e(3) = 5$.

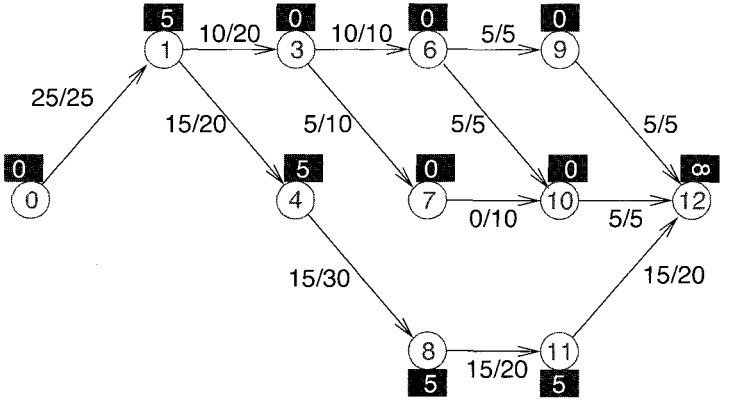


Figure 6.19: The level network after the final iteration of the wave algorithm.

Dinic's algorithm iterates the construction of the level network and the calculation of a blocking until the sink is not part of the level network any more. Then the maximum flow has been obtained, since there is no path from the sink to the source in the residual network. Finally a summary of the algorithm is given:

algorithm dinics-algorithm(N, $\{c(i,j)\}$)
begin
  initialize flow $f(i,j) := 0 \quad \forall i,j$
  build residual network R
  build level network $LN(R)$
  while ($N+1 \in LN(R)$)
  begin
    initialize vertex capacities $p(i)$
    initialize preflow $\{\hat{f}(i,j)\}$
    while ($\exists$ unbalanced vertices $e(i) > 0$)
    begin
      scan $LN(R)$ in topological order:
        push forward preflow
      scan $LN(R)$ in reversed topological order:
        push backward preflow
      recalculate vertex capacities $p(i)$
    end
    increase flow: $f(i,j) := f(i,j) + \hat{f}(i,j) \quad \forall i,j$
    build residual network R
    build level network $LN(R)$
  end
  return $\{f(i,j)\}$
end

Example: The second execution of the wave algorithm

After finishing the inner loop containing the forward and backward waves, the flow values obtained are transfered to the original graph [$f(i,j) := f(i,j) + \hat{f}(i,j) \quad \forall i,j$]. The residual network after the first iteration of the inner loop is displayed in Fig. 6.20 [please remember $f(i,j) = -f(j,i)$].

The corresponding level network, after removing dead ends and calculating the vertex capacities, is shown in Fig. 6.21. Please note that now the topological order of the nodes is 0, 2, 5, 4, 8, 11, 12. Obviously, the wave algorithm treats this network by performing just one forward wave. This is due to the vertex capacities introduced by Träff. By adding the flow obtained in such a way to the total flow, the maximum flow has been calculated, since no further increment is possible: the level network obtained after this step is not connected to the sink.

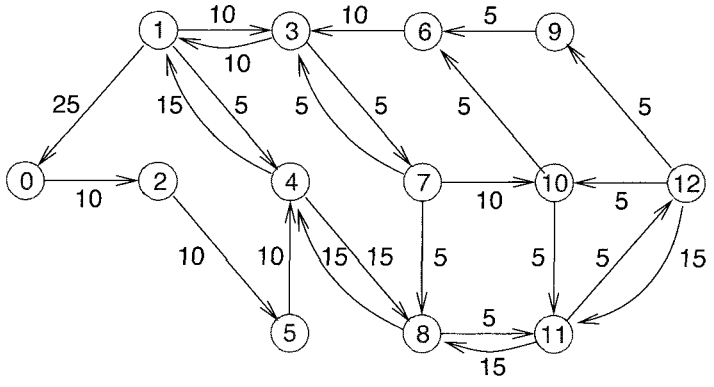


Figure 6.20: The residual network after the first execution of the inner loop.

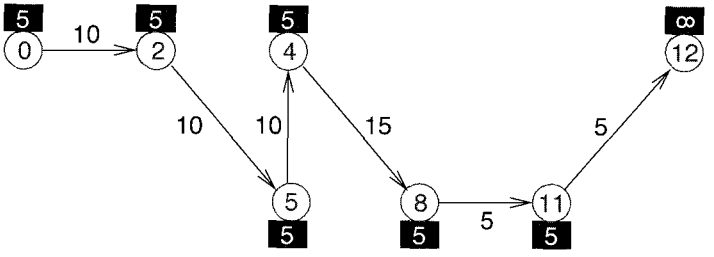


Figure 6.21: The level network graph before executing the wave loop a second time.

□

Please note, the graph in Fig. 6.13, which may keep the original Ford-Fulkerson algorithm busy for a while, is treated by the wave algorithm within one forward wave.

6.5 Calculating all Ground States

We have seen that in order to calculate a ground state of an RFIM one first has to construct the corresponding network and then calculate the maximum flow through the network. *One* single minimum cut, which represents one spin configuration of minimum energy, can be found in the following way, which is a special variant of a breadth-first search: the basic idea is to start at the source and then to follow only edges where the flow is less than the capacity. Consequently, one always stays on the left side of the minimum cut, since only edges which are satisfied cross it. After all vertices have been found which are accessible from the source in this way, the iteration

stops. Then the spins which have been visited are left to the cut, so they are set to orientation $+1$, see the definition of $\underline{X}$ in (6.7) and remember $\sigma_i = 2x_i - 1$. The remaining spins are set to orientation -1.
Please note that in this way only one ground state can be obtained, even if the system is degenerate, i.e. if it has many ground states. A simple method to find *different* ground states in different runs of an algorithm has been presented in [27]. To actually find *all* different ground states one must find *all* existing minimum cuts. This can be achieved with a method explained in [28]. The result is a graph which describes all possible minimum cuts. Now the method will be explained in detail. In this section, we will show why the method is indeed correct. For a formal proof the reader is referred to [28].
The basic idea is to describe all ground states as a set of clusters of spins and as dependencies between these clusters. Each cluster consists of spins which have in *all* ground states the same relative orientation to each other. This means that no minimum cut will divide any two of the corresponding vertices of one cluster in the network. We have already mentioned that only satisfied edges may cross a cut. Thus, the vertices of one cluster are connected somehow by paths of unsatisfied edges.

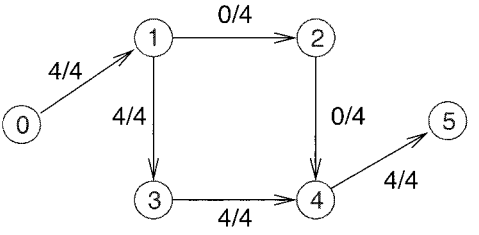


Figure 6.22: Maximum flow for the network already presented in Fig. 6.6. The flow through edges $(1,3)$ and $(3,4)$ may be (partially) redirected through edges $(1,2)$, $(2,4)$. Therefore, only edges $(0,1)$ and $(4,5)$ are satisfied in all possible distributions of the maximum flow among the edges.

On the other hand, if an edge is satisfied this does not guarantee that at least one of the minimum cuts crosses this edge. The reason is that it may be possible to redirect the minimum flow such that there is another topology of the flow values resulting in the same amount of total flow. In this case it is just accidental that the edge was satisfied. This is illustrated by an example given in Fig. 6.22.
The starting point for constructing the cluster graph representing all minimum cuts is the residual network. There the edges just represent unused capacities $[f(i,j) < c(i,j)]$ or they represent directions which can be used to redirect the flow $[f(j,i) < 0]$ in the original network. Here we are only interested if there is an edge in the residual network or not, so we can neglect the residual capacities. The main statement of [28] is: the strongly connected components (SCC) in the residual graph are the above mentioned clusters of vertices, which are for all minimum cuts on the same side of the cut, i.e. for each cut all vertices of a cluster are left or all are right of a cut. Please remember

the definition of the SCC: these are subsets of vertices of maximum cardinality, where each vertex is connected to each other one. An algorithm for the calculation of the SCC was given in Chap. 3.
After all clusters have been obtained, a new *reduced* graph R'' is built, which contains one vertex for each SCC. Each edge in the residual network which runs between different SCCs is also present in R'' (multiple edges are kept only once). All existing minimum cuts can be built using the reduced graph. This works in the following way: the following implication gives the meaning of each edge (k, l) in R''

if SCC k is on the left side of the cut, then SCC l has to be on the left side as well **(IMP)**

Please remember that the left side of a cut $(S, \overline{S})$ is S and that the source is always left of the cut. The central statement of this section is: all cuts are represented by the reduced graph. All cuts are allowed that are compatible with *all* constraints imposed by the reduced graph.
We will now explain why this interpretation of the edges in R'' is meaningful and how the notion of SCC as clusters follows from this. Please note that the implication **IMP** also applies to the edges in the residual network, since the vertices can be seen as tiny clusters. Let us assume that (i, j) is such an edge. So **IMP** reads: if vertex i is left of the cut, then vertex j must be left as well. The existence of edge (i, j) allows for three cases regarding the original network after the maximum flow has been calculated:

- $f(i,j) = 0 < c(i,j)$. In this case the edge (i, j) must not be removed by any cut, because only satisfied edges may cross a cut. So the three possibilities
  - both i, j on the left side (in short: "i, j left")
  - both i, j right
  - j left and i right

  are allowed. The third case is allowed, since an edge contributes only to the cut, if it goes from the left to the right side but not if it runs from right to left. That these three cases are allowed is exactly what **IMP** tells us.

- $f(i,j) = -c(j,i)$, this is the other extreme. Here there is a non-vanishing flow in the opposite direction (j, i) and the edge (j, i) is satisfied. Now again it may be allowed to have both i, j on the same side of the cut, also edge (j, i) may be removed (j left, i right, if there is no contradiction from other constraints). But it is not allowed to have i on the left and j on the right side of the cut, since in that case there is flow from the right to the left side of the cut. Since all flow originates at the source, which is always left, this means it must cross the cut from left to right twice. This contradicts the fact that the cut is minimal. Summarizing, again **IMP** holds.

- $0 < f(i,j) < c(i,j)$. In this case, since $f(i,j) = -f(j,i)$ in addition to edge (i, j), also edge (j, i) is part of the residual network. Please note that the case $-c(j,i) < f(i,j) < 0$ is covered here as well. Consequently we have two **IMP**

clauses which combine to "both i, j must be on the same side of the cut". This is reasonable, since there is a non-vanishing flow as well as an unsatisfied capacity, so both reasonings given above for the other two cases hold.

If there is a chain of implications (i_1 left) implies (i_2 left), (i_2 left) implies (i_3 left), ..., (i_n left) implies (i_1 left), then from the transitivity of the implication rule follows that all vertices $i_1, \ldots, i_n$ must always be on the same side of a minimum cut. On the other, the chain of implications corresponds to a closed loop of edges in the residual graph. Thus, all vertices belong to the same strongly connected component. This explains why the method to construct all minimum cuts given above is indeed correct.

Example: All minimum cuts

We proceed by constructing in Fig. 6.23 the residual network (without capacities) for the small sample graph. Since vertices $1, 2, 3, 4$ are connected in a circular way, the SCCs are given by $A = \{0\}$, $B = \{1, 2, 3, 4\}$ and $C = \{5\}$. Please note that in case another distribution of the maximum flow among the edges is taken, the residual network may look different. But the resulting SCC and the final reduced graph are always the same. The resulting

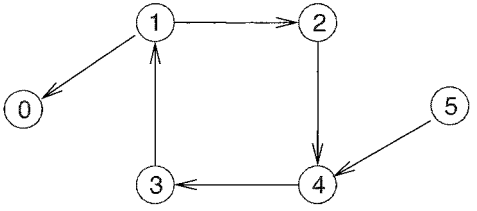


Figure 6.23: Residual network of the network with maximum flow of Fig. 6.22.

reduced graph is presented in Fig. 6.5. Since SCC A contains the source, it must always be left of the cut. C contains the sink, so it must always be right of the cut. So the reduced graph tells us that two minimum cuts are allowed: $(\{A\}, \{B; C\})$ and $(\{A, B\}, \{C\})$. They are identical with the minimum cuts which were obtained by direct inspection in Sec. 6.2. The reduced graph

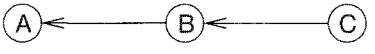


Figure 6.24: Reduced graph of the network with maximum flow of Fig. 6.22. All minimum cuts are represented by this graph.

for the network of Fig. 6.7 calculated from its maximum flow (see Fig. 6.11) is shown in Fig. 6.5.

□

After one has obtained the reduced graph describing all minimum cuts, for their evaluation, it is possible to enumerate all of them. This can be done by applying the

following algorithm [29]. It considers only the N'' components which do not contain the source or the sink. We assume that the SCCs are numbered in a decreasing topological order, i.e. whenever (k, l) is an edge in the reduced graph, then $k < l$. The basic idea is to start with all SCCs right of the cut. Similar to Sec. 6.2, we use a variable $X_k = 0, 1$ to describe whether an SCC k is left ($X_k = 1$) or right of the cut. Thus, the string $\underline{X} = \{X_k\}$ describes a cut. This string can be read as a binary number X. This means all states can be enumerated by just increasing X starting from 0 up to $2^{N''} - 1$ and considering only the configurations which are compatible with the constraints imposed by the reduced graph. This is done by the following algorithm:

algorithm enumerate-cuts(R'')
begin
  Let $X_k := 0$ for all k
  while not all $X_k = 1$ **do**
  begin
    Find the smallest component l_0 with $X_{l_0} = 0$
    Let $X_{l_0} = 1$
    for all $l < l_0$ **do**
      if component l is not a successor of l_0 in R'' **then**
        Let $X_l = 0$
  end
end

This algorithm can be used for all kinds of graphs or relations R'' which denote a priority relation, i.e. they must not contain cycles. Then all configurations which obey all priority rules can be enumerated with this method.
The system shown in Fig. 6.5 has only one strongly connected component which contains neither the source nor the sink, so only the cuts $X_1 = 0, 1$ are possible. The application to the network which was shown in Fig. 6.11 is more instructive.

Example: Enumerating all cuts

The resulting reduced graph is shown in Fig. 6.5. All SCCs have size one, i.e. contain one vertex.

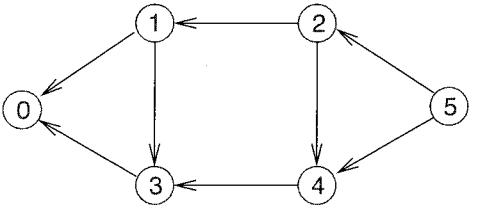


Figure 6.25: Residual graph of a small network which was presented along with a maximum flow in Fig. 6.11.

Only four components $k = 1, 2, 3, 4$ have to be considered. After renumbering them according a topological order, the reduced graph looks like that shown in Fig. 6.5.

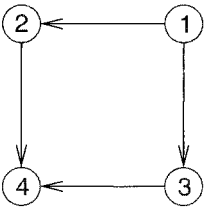


Figure 6.26: Residual graph of a small network after renumbering the vertices and omitting the source and the sink.

The algorithm starts with $\underline{X} = X_4X_3X_2X_1 = 0000$ (we denote at the right the least significant bit/component with the smallest number), i.e. all inner components are to the right of the cut: $(S, \bar{S}) = (\{0\}, \{1, 2, 3, 4, 5\})$.

Within the first iteration $j_0 = 1$, $\underline{X}$ is increased to $\underline{X} = 0001$. So we obtain $(S, \bar{S}) = (\{0, 1\}, \{2, 3, 4, 5\})$.

By the second iteration $j_0 = 2$ and $\underline{X} = 0011$ is obtained. Since component 1 is a successor of component 2, the string $\underline{X}$ is not altered by the **for**-loop. Therefore, we obtain $(S, \bar{S}) = (\{0, 1, 2\}, \{3, 4, 5\})$.

Next $j_0 = 3$ and initially $\underline{X} = 0111$. Since component 2 is not a successor of 3 (but SCC 1 is), we get $\underline{X} = 0101$. This means $(S, \bar{S}) = (\{0, 1, 3\}, \{2, 4, 5\})$.

The next iteration results in $j_0 = 2$, thus $\underline{X} = 0111$. Similar to the second iteration, now $\underline{X}$ remains unchanged and we get $(S, \bar{S}) = (\{0, 1, 2, 3\}, \{4, 5\})$.

For the final iteration $j_0 = 4$ and $\underline{X} = 1111$. All other components are successors of j_0, so the **for**-loop does not alter $\underline{X}$ and we get $(S, \bar{S}) = (\{0, 1, 2, 3, 4\}, \{5\})$.

Summarizing, 6 minimum cuts are obtained. They are the same as those already found in Sec. 6.2 by direct inspection, see Fig. 6.7 (please note that the order of the vertices is different from the order of the corresponding components due to the topological order). □

Many types of graphs exhibit an exponential number of minimum cuts, so the enumeration is quite time consuming. The number of different cuts itself has to be obtained by the enumeration as well, so it is fairly time consuming for large systems. But in the case of the systems considered here, DAFF and RFIM which have an exponential degeneracy as well, it turns out that the reduced graph is usually very sparse. Therefore, the reduced graph can be divided into many small groups of connected SCCs without edges to other clusters. Then all groups can be treated independently, i.e. the number of minimum cuts is the product of the number of cuts allowed by each group.

This drastically reduces the running time of the algorithm. Furthermore, it is possible to extract information from the graph directly. For example one can use the extremal cuts, i.e. the cuts having all/no (except one) SCCs left of the cut, to calculate the minimum/maximum magnetization of the corresponding system.
Now all tools for examining the zero temperature properties of random field systems and diluted antiferromagnets are available. To summarize: first an equivalent network is constructed (see Sec. 6.2), then the maximum flow is obtained with the wave algorithm (Sec. 6.4) and finally the graph describing all degenerate ground states is calculated. Some basic results for RFIM and DAFF, which were found with these techniques, are shown in the next section.

6.6 Results for the RFIM and the DAFF

In this section results of the ground-state calculations for random field systems and diluted antiferromagnets are presented. Mainly the RFIM with bimodal distribution of the random fields is considered. First the behavior of the order parameter for one realization is presented, then the average results for different system sizes. Using the technique of finite-size scaling, the thermodynamic limit is executed. At the end of this section properties of the degenerate ground-state landscape in the domain region of the phase diagram (see Fig. 6.3) are discussed.

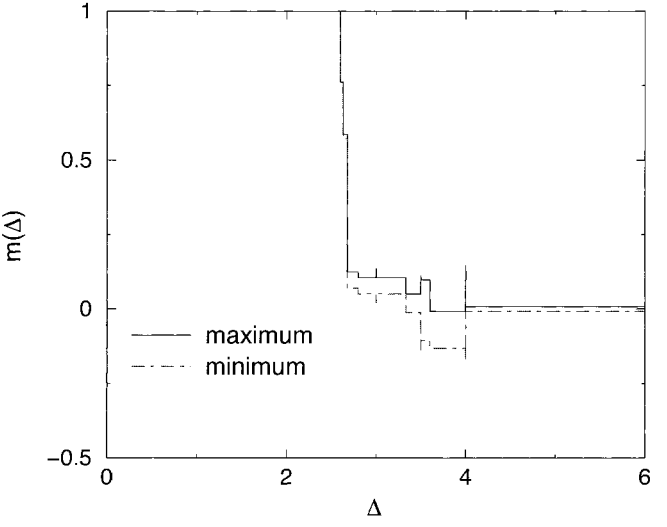


Figure 6.27: Magnetization m as a function of the strength of the random fields for one realization of an RFIM ($L = 8$) with bimodal distribution.

In Fig. 6.27 the behavior of the magnetization $m = 1/N \sum_i \sigma_i$ of one small $N = 8^3$ realization of a random-field system with bimodal distribution of the fields is shown.

Since this type of system exhibits a ground-state degeneracy, the magnetization may vary from one ground state to the other. In the figure the maximum and minimum values which the magnetization can take are shown. How many intermediate values are possible, depends on the degree of degeneracy. More results can be found in [30]. In general, the behavior turns out to be as expected in the first section: for small fields the system is ferromagnetically ordered. With increasing strength of the random fields the spins tend to be oriented in the direction of the local fields and the magnetization vanishes. The order parameter changes only at a finite number of values for Δ, in between it is constant. A detailed analysis [26] shows that the steps in $m(\Delta)$ occur whenever the sum of local fields on some cluster is strong enough to flip the whole cluster.

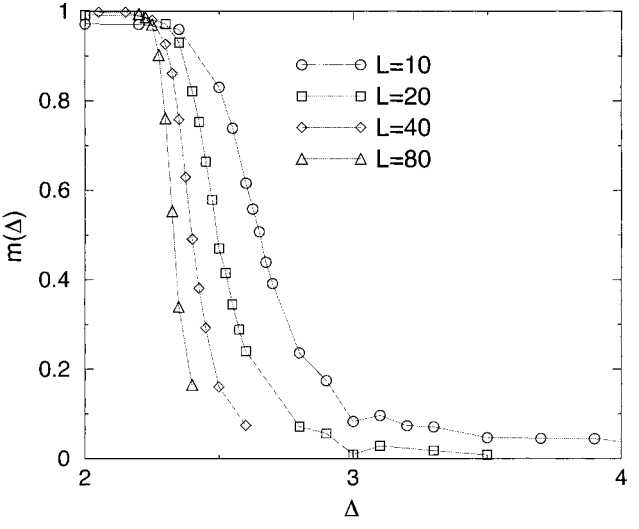


Figure 6.28: Average maximum of the absolute value of the magnetization as a function of the strength of the random fields. An average for 1000 realizations of the RFIM with bimodal distribution was taken. Lines are guides to the eyes only.

The behavior is similar for other realizations, but the values where the cluster turn around vary from system to system. Only the jumps at some integer values of Δ appear in all realizations. These discontinuities are due to the reversing of several single spins. By averaging over different realizations one gets a smooth curve. The result is shown in Fig. 6.28 for the maximum of the absolute value of the magnetization averaged over typically 1000 different realizations. For other quantities like the minimum or the average of the magnetization the results look similar. In the figure the data for 4 different system sizes $N = 10^3$, 20^3, 40^3 and 80^3 are presented. With increasing system size a monotonic shift of the curves can be observed. We are interested in performing the thermodynamic limit, i.e. in obtaining the behavior of very large systems. This

can be done by applying the technique of finite-size scaling. The basic assumption is that the average magnetization shows the following behavior [31]:

$$m = L^{-\beta/\nu}\tilde{m}((\Delta - \Delta_c)L^{1/\nu}) . \tag{6.24}$$

Δ_c is the critical value of the infinite system where the magnetization vanishes. The values of the critical exponents β and ν describe the asymptotic behavior of the order parameter and the correlation length near the transition respectively. The exact form of the function $\tilde{m}$ is in general unknown. The values for Δ_c, ν and β have to be determined. They can be obtained by rescaling the numerical data in a way that all data points collapse onto one curve. The resulting *scaling plot* is shown in Fig. 6.29. The values obtained in this way are $\Delta_c = 2.20$, $\nu = 1.67$ and $\beta = -0.01$.

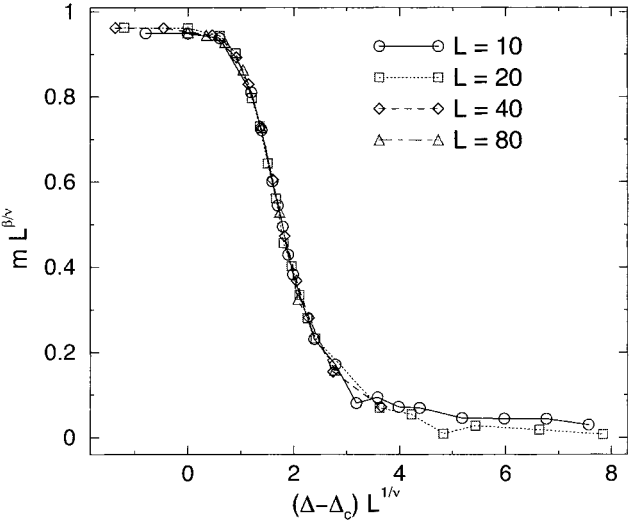


Figure 6.29: Finite-size scaling plot for RFIM with bimodal distribution. The parameters are $\Delta_c = 2.20$, $\nu = 1.67$ and $\beta = -0.01$.

As mentioned at the beginning, it is assumed that the RFIM is similar to the DAFF. In this case the values for the exponents should be equal. Additionally, they should not depend on details of the model such as the choice of the distribution of the random fields. In this case one says that the exponents are *universal*. On the other hand, the exact value of the critical strength of the field is not expected to be universal. By calculating similar finite-size scaling plots for the RFIM with Gaussian distribution and the DAFF, the following values are found:

- Gaussian RFIM: $\Delta_c = 2.29$, $\nu = 1.19$ and $\beta = 0.02$
- DAFF: $\Delta_c = 0.62$, $\nu = 1.14$ and $\beta = 0.02$

Consequently, the exponents seem not to be universal, since ν is different for the different distributions of the RFIM. On the other hand, the Gaussian random-field system seems to be a good model for diluted antiferromagnets, since in this case the values of the exponents agree.

For an intermediate strength of the field, the order parameter vanishes. Nevertheless, in this region the systems exhibit fractal domains, which are described by non-trivial exponents. An example of such a domain, is displayed in Fig. 6.30. Quantitative results can be found e.g. in Ref. [32].

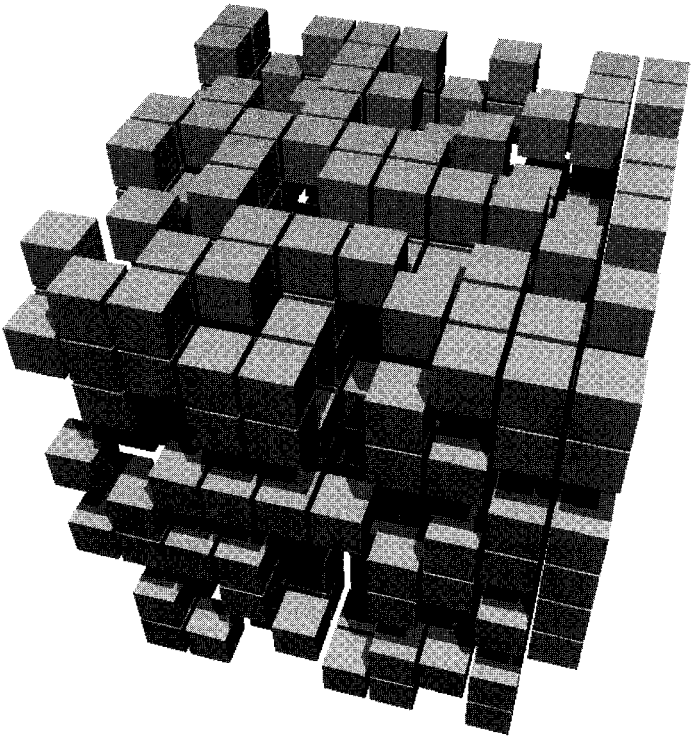


Figure 6.30: The largest ferromagnetically ordered domain in a small RFIM ($N = 10^3$) at $\Delta = 3.0$.

Finally, we will give some examples regarding the ground-state degeneracy of the DAFF and the RFIM with bimodal interactions [26]. It turns out that the number of ground states grows exponentially with system size, so there is a finite entropy, see also Ref. [33]. On the other hand, the ground-state landscape is rather simple. Most of the spins have the same orientation in all ground states, i.e. they are frozen. This fraction is more than 95% for the DAFF and even 98% for the RFIM. The clusters of spins, which may have two orientations in different ground states, interact only rarely

with each other. This means the reduced graph contains only few edges. Therefore, the clusters can choose one of their orientations more or less independently of each other.
To describe the ground-state landscape a quantity called *overlap* q can be used. Let $\{\sigma_i^A\}$ and $\{\sigma_i^B\}$ be two independent ground-state configurations for the same system, i.e. the same realization of the disorder. One can compare these two configurations by calculating

$$q = \frac{1}{N'} \sum_i \sigma_i^A \sigma_i^B, \tag{6.25}$$

where N' is the number of spins, i.e. $N' = \sum_i \epsilon_i$ for the DAFF and $N' = N$ for the RFIM. Thus, if A, B are equal $q = 1$, while $q = -1$ if $\{\sigma_i^A\}$ and $\{\sigma_i^B\}$ are inverted relative to each other. In general $-1 \leq q \leq 1$.
For a set of M ground states one compares each state with each other one, resulting in $M(M-1)/2$ values. So one gets a whole distribution of overlaps. To describe the behavior of a random ensemble, again one takes different realizations of the disorder and calculates an average distribution $P(q)$. The results for the DAFF are shown in Fig. 6.31, see also [27]. The fact that the ground-state landscape is rather simple, is reflected by the fact that $P(q)$ is zero for most of the overlap values and by the shrinking of the width of $P(q)$. In the thermodynamic limit the distribution becomes a delta function. The result for the RFIM looks similar.
Other recent results for the ground states of diluted antiferromagnets and random-field systems can be found e.g. in [32, 34, 35, 36, 37, 38, 39, 40].
For another class of random systems, the spin glass model, the ground state landscape for individual realizations looks much more interesting and $P(q)$ is very broad for finite sizes. This subject is covered in Chap. 9.

Bibliography

[1] S. Fishman and A. Aharony, *J. Phys.* **C12**, L729 (1979)

[2] J.L. Cardy, *Phys. Rev. B* **29**, 505 (1984)

[3] H. Rieger, in: D. Stauffer (ed.), *Annual Reviews of Computational Physics II*, (World Scientific, Singapore 1995)

[4] R.A. Cowley, H. Yoshizawa, G. Shirane, and B.J. Birgeneau, *Z. Phys. B* **58**, 15 (1984)

[5] D.P. Belanger, in: A.P. Young (ed.), *Spin Glasses and Random Fields*, (World Scientific, Singapore 1998)

[6] U. Nowak and K.D. Usadel, *Phys. Rev. B* **44**, 7426 (1991)

[7] K.D. Usadel and U. Nowak, *JMMM* **104-107**, 179 (1992)

[8] J.-C. Picard and H.D. Ratliff, *Networks* **5**, 357 (1975)

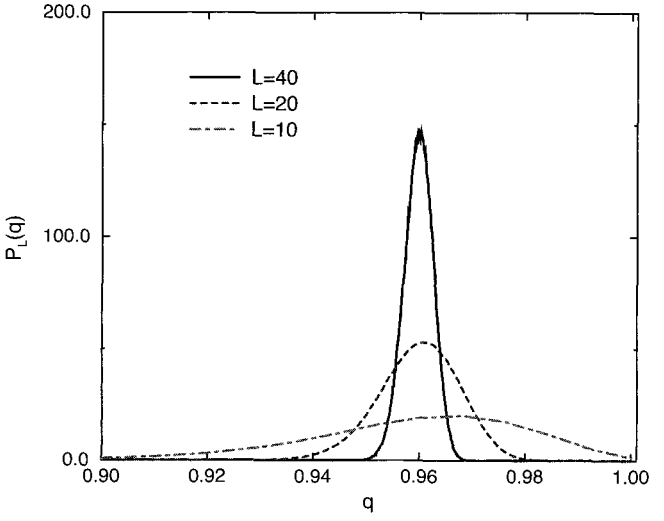


Figure 6.31: Average distribution of overlaps for the three-dimensional DAFF in the domain phase at $B = 3.99$ for finite sizes $L = 10, 20, 40$. For each realization 50 ground states were selected randomly.

[9] E.T. Seppälä, M.J. Alava, and P. M. Duxbury, *Phys. Rev. E* **63**, 036126 (2001)

[10] E.T. Seppälä and M.J. Alava, *Phys. Rev. Lett.* 84, 3982 (2000)

[11] E.T. Seppälä, V.I. Raisanen, and M.J. Alava, *Phys. Rev. E* **61**, 6312 (2000)

[12] R. Fisch, *J. Stat. Phys.* **18**, 111 (1978)

[13] W. Kinzel and E. Domany, *Phys. Rev. B* **23**, 3421 (1981)

[14] L.R. Ford and D.R. Fulkerson, *Canadian J. Math.* **8**, 399 (1956)

[15] J.D. Claiborne, *Mathematical Preliminaries for Computer Networking*, (John Wiley & Sons, New York 1990)

[16] W. Knödel, *Graphentheoretische Methoden und ihre Anwendung*, (Springer, Berlin 1969)

[17] M.N.S. Swamy and K. Thulasiraman, *Graphs, Networks and Algorithms*, (John Wiley & Sons, New York 1991)

[18] J. Edmonds and R.M. Karp, *J. ACM* **19**, 248 (1972)

[19] R.E. Tarjan, *Data Structures and Network Algorithms*, (Society for Industrial and Applied Mathematics, Philadelphia 1983)

[20] L.R. Ford and D.R. Fulkerson, *Flows in Networks*, (Princeton University Press, Princeton 1962)

[21] A.V. Goldberg and R.E. Tarjan, *J. ACM* **35**, 921 (1988)

[22] B. Cherkassky and A. Goldberg, *Algorithmica* **19**, 390 (1997)

[23] A.V. Goldberg, R. Satish, *J. ACM* **45**, 783 (1998)

[24] E.A. Dinic, *Soviet Math. Dokl.* **11**, 1277 (1970)

[25] J.L. Träff, *Eur. J. Oper. Res.* **89**, 564 (1996)

[26] A.K. Hartmann, *Physica A* **248**, 1 (1998)

[27] A.K. Hartmann and K.D. Usadel, *Physica A* **214**, 141 (1995)

[28] J.-C. Picard and M. Queyranne, *Math. Prog. Study* **13**, 8 (1980)

[29] L. Schrage and K.R. Baker, *Oper. Res.* **26**, 444 (1978)

[30] A.K. Hartmann and U. Nowak, *Eur. Phys. J. B* **7**, 105 (1999)

[31] K. Binder and D.W.Heermann, *Monte Carlo Simulation in Statistical Physics*, (Springer, Heidelberg 1988)

[32] J. Esser and U. Nowak, *Phys. Rev. B* **55**, 5866 (1997)

[33] S. Bastea and P.M. Duxbury, *Phys. Rev. E* **58**, 4261 (1998)

[34] A.T. Ogielski, *Phys. Rev. Lett.* **57**, 1251 (1986)

[35] J.C. Angls d'Auriac, M. Preissmann, and A. Seb, *J. Math. Computer Modelling* **26**, 1 (1997)

[36] J.-C. Anglès d'Auriac and N. Sourlas, *Europhys. Lett.* **39**, 473 (1997)

[37] C. Frontera and E. Vives, *Phys. Rev. E* **59**, R1295 (1999)

[38] E.T. Seppälä, V. Petäjä, and M.J. Alava, *Phys. Rev. E* **58**, R5217 (1998)

[39] U. Nowak, K.D. Usadel, and J. Esser, *Physica A* **250**, 1 (1998)

[40] N. Sourlas, *Comp. Phys. Comm.* **121-122**, 183 (1999)

7 Minimum-cost Flows

7.1 Motivation

The ground-state configuration of a directed polymer in a random environment, in which all (bind)-energies are non-negative, can be obtained with Dijkstra's algorithm to find the shortest path in a directed network with non-negative costs on the edges. We will also refer to this problem as the 1-line-problem since the directed polymer configuration or shortest path is a line-like object threading, in typical physical situations, a two- or three-dimensional system.

At this point it appears natural to ask for the minimal energy configuration (or ground state) of more than one, say N, lines in the same disordered environment under the constraint (the "capacity constraint") that only one line can pass a single bond in the lattice or edge in the graph. Having a particular physical situation in mind, namely magnetic flux lines threading through a disordered superconductor in the mixed phase (Shubnikov phase), we usually want the lines to enter the system through a top surface and to leave the system through a bottom surface as, for instance, the $z = H$ and the $z = 0$ plane, respectively, in a two-dimensional square lattice or a three-dimensional simple cubic geometry, see Fig. 7.1.

The physical situation one is interested in by considering such cases of interacting elastic lines in a random potential are commonly described by the Hamiltonian

$$\mathcal{H} = \sum_{i=1}^{N} \int_0^H dz \Big\{ \tfrac{\gamma}{2} \left[\tfrac{d\mathbf{r}_i}{dz} \right]^2 + \textstyle\sum_{j(\neq i)} V_{\rm int}[\mathbf{r}_i(z) - \mathbf{r}_j(z)] + V_r[\mathbf{r}_i(z), z] + V_{\rm p}[\mathbf{r}_i(z)] \Big\} . \tag{7.1}$$

$\mathbf{r}_i(z) \in \mathcal{R}^{d-1}$ (usually $d = 2$ and $d = 3$) is a displacement vector, while z is the longitudinal coordinate in a system of height H; $V_r[\mathbf{r}, z]$ describes the point disorder, which we can take to be delta-correlated with variance ϵ; $V_{\rm int}[\mathbf{r} - \mathbf{r}']$ is a short-range repulsive interaction between the lines (e.g. hard-core) and $V_{\rm p}[\mathbf{r}]$ can describe columnar defects or a periodic potential with period a in all transverse space directions. For the case of a single line one recovers the directed polymer Hamiltonian that we have previously considered in Sec. 4.2.1 in its lattice version in connection with Dijkstra's algorithm.

A naive way to approach the N-line problem would be to search for the shortest path from the top to the bottom boundary, giving a 1-line configuration, then removing the edges occupied by this line from the graph, searching for the shortest path from the

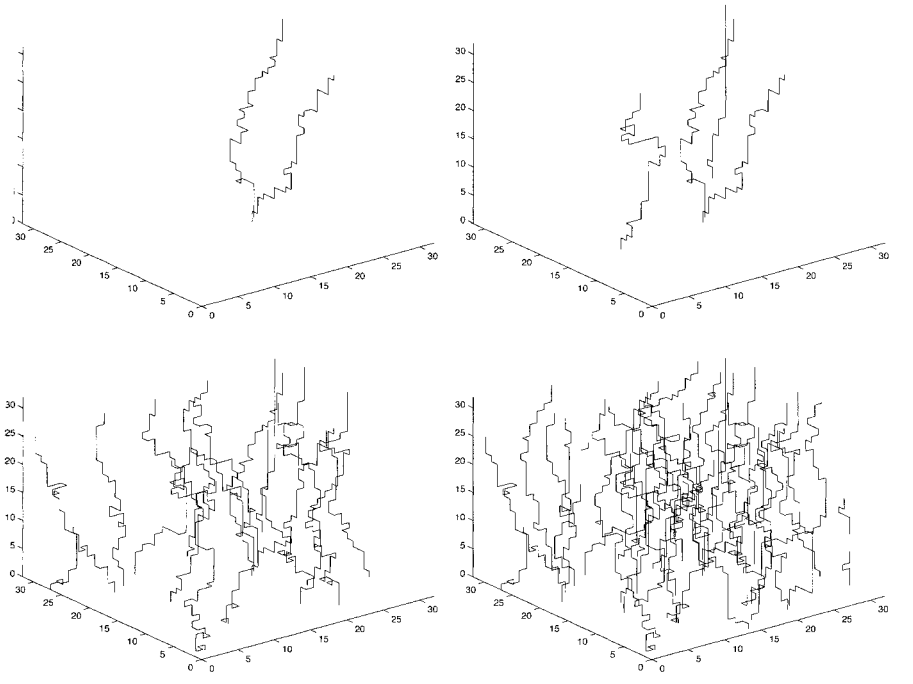


Figure 7.1: Ground-state configuration of N lines in a random environment for one particular 3d sample of size $L \times L \times H = 32 \times 32 \times 32$, with $N = 2, 4$ (top), $16, 32$ (bottom).

top to the bottom in the remaining network, giving (together with the removed bonds) a 2-line configuration. One Proceeds successively until N lines thread the sample from top to bottom. Apart from the obvious problems one could run into by successively removing edges from the graph (it could happen that we cannot add further lines since all paths from top to bottom are blocked) a little bit of thinking will convince us that in this way one will, in general, *not* find the minimum energy configuration: it could be that the ground state of 2 lines is not separable into the 1-line ground state plus the "first excited state" that one constructs in the way described above (see Fig. 7.2 and Fig. 7.3). In many cases one has to deform pieces (or even all) of the 1st line before we add the 2nd line in order to minimize the *total* energy. As long as there are only a few lines in a large system the difference will vanish in the thermodynamic limit. However, if the density of lines is fixed a difference will exist with probability one even in the infinite system size limit. For *dense* systems, as we will consider in what follows, the difference will be essential.

How does one take into account all possible deformations of $N-1$ lines when one wants to add the Nth line to the system? At first sight this appears to be a tremendous

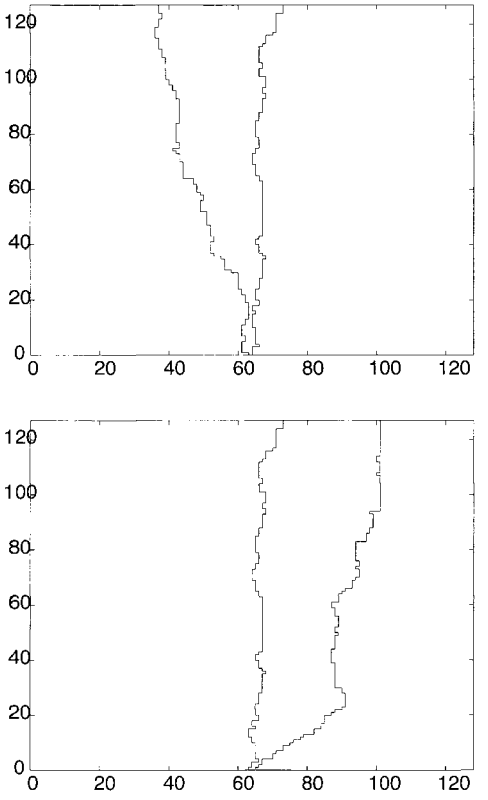


Figure 7.2: (*top*) Two-polymer ground-state in 2d, (*bottom*) the same system but with the first (1-line GS) frozen first. The energy of the configuration (*top*) is lower. Note that in (*top*) case the 1-line GS is to the left line compared to the (*bottom*) figure. The fact that in (*top*) the 1-line GS is minimally deformed (in the lower part in order to give the 2nd line a bit of space which is energetically favorable) produces a 2-line GS that is totally different (concerning the 2nd line) from the configuration (*bottom*). In both cases the disorder landscape is the same.

task, but there is an elegant trick by which one can devise an efficient algorithm for the N-line problem. First one does not work with the original network but with the so called residual network that depends on the number of lines one has to put into the system and their actual configuration. It is defined as follows: since we confine ourselves in the beginning to capacity-one edges (i.e. only a single line can pass each edge) we *remove* each edge (ij) that is occupied by a line segment ($x_{ij} = 1$) and *insert* the reversed edge (ji) with cost $c_{ji} = -c_{ij} \leq 0$! These reversed edges can now be occupied by (virtual) line segments ($x_{ji} = 1$) through which one *gains* energy ($c_{ji} \leq 0$), which is due to the fact that in reality one removes a line segment from edge

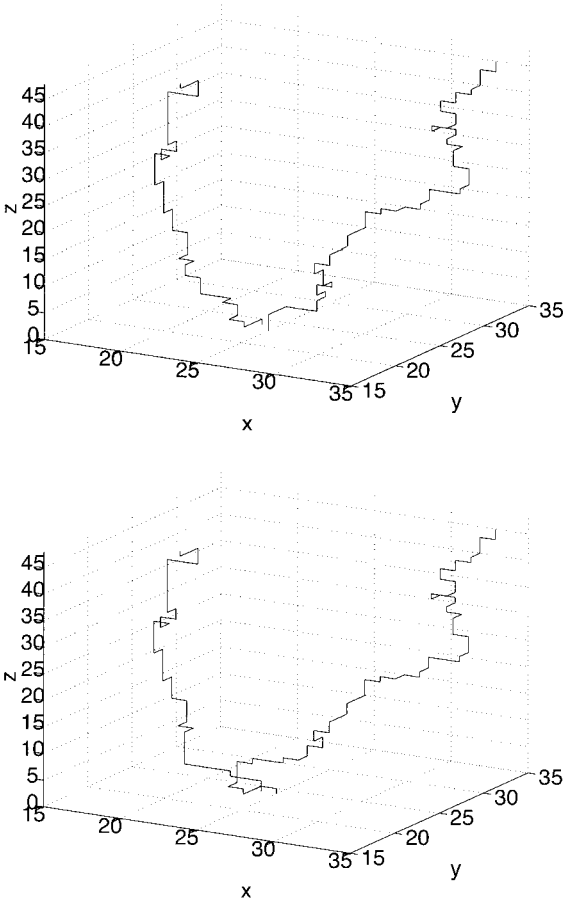


Figure 7.3: Similar to Fig. 7.2 but in 3d. In both cases the disorder landscape is the same.

(ij) when occupying edge (ji) and thus reduces the total energy by an amount c_{ij}. In this way one incorporates elegantly the possibility of deforming the already existing lines when adding a new line into the residual network.

One difficulty occurs. Now a number of edges have *negative* costs, which renders the efficient shortest path finder, Dijkstra's algorithm, inapplicable. However, in this particular situation, in which the $N-1$–line configuration is indeed the one with the lowest total cost, there is *no negative cycle* in the residual network (i.e one cannot find a closed path in the residual whose addition to the existing configuration would lower the energy). Therefore it is possible to find so called *node potentials* $\pi(i)$ for all nodes i that can be used to modify the costs in the residual network in such a way that they

are all non-negative. These *reduced costs* are then defined as $c_{ij}^{\pi} = c_{ij} - \pi(i) + \pi(j)$, π is chosen such that $c_{ij}^{\pi} \geq 0$ for all edges in the residual network and the shortest paths with respect to these reduced cost c_{ij}^{π} are still the shortest paths with respect to the original costs c_{ij} of the residual network. In this residual network with these reduced costs one proceeds now in the same way as we proposed naively at the beginning of this section: one adds lines successively to the system by finding the shortest paths in the residual network, updating the line (or flow) configuration and then updating the residual network including the reduced costs again. We will now put this idea on a formal basis.

7.2 The Solution of the N-Line Problem

First we will introduce the notation for the model of N repulsive lines in a disordered environment that we described above. In a lattice version of the model (7.1) without periodic potential one has to incorporate three terms: the elastic energy $[d\mathbf{r}_i/dz]^2$, the interaction energy V_{int} and the random potential V_r. A single line will be represented by a path along the bonds of a lattice (or the edges of a grid graph), the random potential energies will be then random variables on theses bonds or edges. If they are all positive one does not need to take explicitly into account the elastic energy, since all transverse excursions will cost energy, which yields an *effective* elastic energy for the lines. The interactions will be hard core repulsion, which allows only single occupancy of the bonds. Thus we can consider the Hamiltonian or energy or cost function

$$H(\mathbf{x}) = \sum_{(ij)} c_{ij} \cdot x_{ij}, \tag{7.2}$$

where $\sum_{(ij)}$ is a sum over all *bonds* (ij) joining site i and j of a d-dimensional lattice, e.g. a rectangular $(L^{d-1} \times H)$ lattice, with arbitrary boundary conditions (b.c.) in $d-1$ space direction (i.e. they can be specified later) and free b.c. in one direction. The bond energies $c_{ij} \geq 0$ are quenched random variables that indicate how much energy it costs to put a segment of fluxline on a specific bond (ij). The fluxline configuration $\mathbf{x}$ $(x_{ij} \geq 0)$, also called a *flow*, is given by specifying $x_{ij} = 1$ for each bond i, which is occupied by the fluxline and otherwise $x_{ij} = 0$. For the configuration to form *lines* on each site of the lattice all incoming flow should balance the outgoing flow, i.e. the flow is divergence free

$$\nabla \cdot \mathbf{x} = 0\,, \tag{7.3}$$

where $\nabla\cdot$ denotes the lattice divergence. Obviously the fluxline has to enter, and to leave, the system somewhere. We attach all sites of one free boundary to an extra site (via energetically neutral edges, $e = 0$), which we call the source s, and the other side to another extra site, the target, t as indicated in Fig. 7.4. Now one can push one line through the system by inferring that s has a source strength of $+1$ and that t has a sink strength of -1, i.e.

$$(\nabla \cdot \mathbf{x})_s = +N \quad \text{and} \quad (\nabla \cdot \mathbf{x})_t = -N, \tag{7.4}$$

with $N = 1$. Thus, the 1-line problem consists of minimizing the energy (7.2) by finding a flow $\mathbf{x}$ in the network (the lattice plus the two extra sites s and t) fulfilling the constraints (7.3) and (7.4).

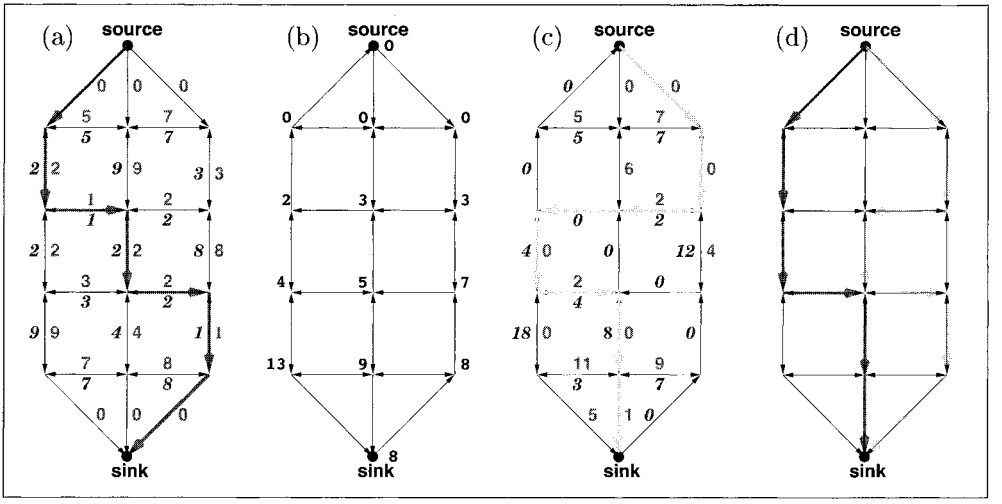


Figure 7.4: Sketch of the successive shortest path algorithm for the solution of the minimum cost flow problem described in the text. **(a)** Network for $N = 0$, the numbers are the bond energies (or costs) c_{ij}. The bold thick line is a shortest path from s to t. **(b)** The residual network G_c^0 for a flow as in (a) with the updated node potentials. **(c)** G_c^0 from (b) with the updated reduced costs plus the the shortest path from s to t in G_c^0 indicated by the thick light line. **(d)** Optimal flow configuration for $N = 2$ in the original network. Note that the 2-line state is *not separable*, i.e. it does *not* consist of the line of (a) plus a 2nd line.

To solve this problem along the lines described in 7.1 we first define the *residual network* $G_c(\mathbf{x})$ corresponding to the actual fluxline configuration. As described in 7.1 it also contains the information about possibilities of sending flow backwards (now with energy $-c_{ij}$ since one wins energy by reducing x_{ij}), i.e. to modify the actual flow. Suppose that we put one fluxline along a shortest path $P(s,t)$ from s to t, which means that we set $x_{ij} = 1$ for all edges on the path $P(s,t)$. Then the residual network is obtained by reversing all edges and inverting all energies along this path, indicating that here we cannot put any further flow in the forward direction (since we assume hard-core interaction, i.e. $x_{ij} \leq 1$), but can send flow backwards by reducing x_{ij} on the forward edges by one unit. This procedure is sketched in Fig. 7.4.
Next we introduce a *node potential* π that fulfills the relation

$$\pi(j) \leq \pi(i) + c_{ij} \tag{7.5}$$

for all edges (ij) in the residual network, indicating how much energy $\pi(j)$ it would *at least* take to send one unit of flow from s to site j, IF it would cost an energy $\pi(i)$

to send it to site i. With the help of these potentials one defines the *reduced costs*

$$c_{ij}^{\pi} = c_{ij} + \pi(i) - \pi(j) \geq 0. \tag{7.6}$$

The last inequality, which follows from the properties of the potential π (7.5) actually ensures that there is no loop $\mathcal{L}$ in the current residual network (corresponding to a flow $\mathbf{x}$) with negative total energy, since $\sum_{(ij)\in\mathcal{L}} c_{ij} = \sum_{(ij)\in\mathcal{L}} c_{ij}^{\pi}$, implying that the flow $\mathbf{x}$ is optimal (for a proof see Sec. 7.3).
It is important to note that the inequality (7.5) is reminiscent of a condition for shortest path distances $d(i)$ from s to all sites i with respect to the energies c_{ij}: they have to fulfill $d(j) \leq d(i) + c_{ij}$. Thus, one uses these distances d to construct the potential π when putting one fluxline after the other into the network.
The iterative procedure we described in 7.1 now works as follows. We start with the empty network (zero fluxlines) $\mathbf{x}^0 = 0$, which is certainly an optimal flow for $N = 0$, and set $\pi = 0$, $c_{ij}^{\pi} = c_{ij}$. Next, let us suppose that we have an optimal $N-1$-line configuration corresponding to the flow $\mathbf{x}^{N-1}$. The current potential is π^{N-1}, the reduced costs are $c_{ij}^{N-1} = c_{ij} + \pi^{N-1}(i) - \pi^{N-1}(j)$ and we consider the residual network G_c^{N-1} corresponding to the flow $\mathbf{x}^{N-1}$ with the reduced costs $c_{ij}^{N-1} \geq 0$. The iteration leading to an optimal N-line configuration x_{ij}^{N} is

algorithm successive shortest path for N-line problem
begin
  $\mathbf{x} := 0$; $\pi(i) := 0$; $G_c(\mathbf{x}) := G$;
  for Line-counter $:= 1, \ldots, N$ **do**
  begin
    compute the reduced costs $c^{\pi}(\mathbf{x})$ Eq.(7.6);
    find the shortest paths $d(i)$ from s to all other nodes i;
      in the residual network $G_c(\mathbf{x})$ w.r. to the reduced costs c_{ij}^{π};
    augment flow on the shortest path from s to t by *one* unit;
    compute $\pi(i) := \pi(i) - d(i)$;
  end
end

An example of how the algorithm operates is given in Fig. 7.4. The complexity of this iteration is the same as that of Dijkstra's algorithm for finding the shortest paths in a network, which is $\mathcal{O}(M^2)$ in the worst case (M is the number of nodes in the network). We find, however, for the cases we consider (d-dimensional lattices) it roughly scales linearly in $M = L^d$. Thus, for N fluxlines the complexity of this algorithm is $\mathcal{O}(NL^d)$.

For the actual implementation of the above iteration it is important to note that it is not necessary actually to find the shortest paths to *all* other nodes: in the **find** routine one uses Dijkstra's algorithm *only* to find a shortest path from s to t and in the **compute** statement it is sufficient to update only those potentials of the nodes that have been *permanently labeled* during Dijkstra's algorithm. It is easy to show that these node potentials still fulfill the requirement $c_{ij} \geq 0$ (7.6).

7.3 Convex Mincost-flow Problems in Physics

Actually the N-line problem is not the only one in physics that can be mapped onto a minimum-cost-flow problem. The physical models we describe in this section have Hamiltonians or cost functions similar to the flux-line energy (7.2) and the same mass-balance constraint (7.3). However, they are more general since now the capacity constraint $x_{ij} \in \{0, 1\}$ is given up and the flow variables x_{ij} can take on any integer, including negative values. Instead the local costs $c_{ij} \cdot x_{ij}$ are replaced by *convex* functions $h_{ij}(x_{ij})$, an example being $h_{ij}(x_{ij}) = (x_{ij} - b_{ij})^2$, where the variables b_{ij} are random and usually different for different edges (ij). Moreover, in most cases there are no external nodes s and t, as in the flux-line problem — the optimal flow will be non-trivial (non-zero) without them (note that in the flux-line problem a non-zero flow only occurred due to the external source and target nodes with mass-balance $+N$ and $-N$, respectively.

Random SOS model

Consider a solid-on-solid (SOS) model with random offsets, modeling a crystalline surface on a disordered substrate as indicated in Fig. 7.5. It is defined by the following Hamiltonian (or energy function):

$$H = \sum_{(ij)} (h_i - h_j)^2 \tag{7.7}$$

where (ij) are nearest neighbor pairs on a d-dimensional lattice ($d = 1, 2$). Each height variable $h_i = d_i + n_i$ is the sum of an integer particle number which can also be negative, and a substrate offset $d_i \in [0, 1)$. For a flat substrate, $d_i = 0$ for all sites i, we have the well known SOS-model [1]. The disordered substrate is modeled by random offsets $d_i \in [0, 1)$ [2], which are distributed independently.

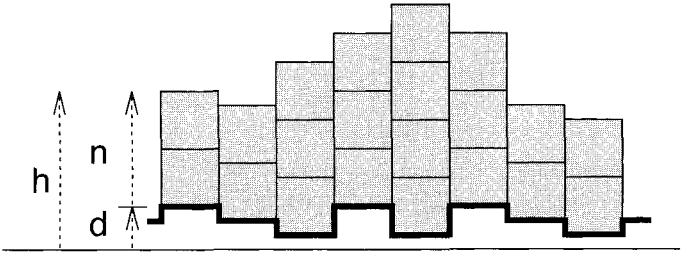


Figure 7.5: The SOS model on a disordered substrate. The substrate heights are denoted by $d_i \in [0, 1]$, the number of particle on site i by $n_i \bullet Z$, which means that they could also be negative, and the total height on site i by $h_i = d_i + n_i$

The model (7.7) has a phase transition at a temperature T_c from a (thermally) rough phase for $T > T_c$ to a *super-rough* low temperature phase for $T < T_c$. In two dimensions "rough" means that the height-height correlation function diverges loga-

rithmically with the distance $C(r) = [\langle (h_i - h_{i+r})^2 \rangle]_{\rm av} = a \cdot T \cdot \log(r)$ [with $a = 1/\pi$ for $f(x) = x^2$], "super-rough" means that either the prefactor at the front of the logarithm is significantly larger than $a \cdot T$ (e.g. T^α with $\alpha > 1$), or that $C(r)$ diverges faster than $\log(r)$, e.g. $C(r) \propto \log^2(r)$.

A part of the motivation to study this model thus comes from its relation to flux lines in disordered superconductors, in particular high-T_c superconductors. The phase transition occurring for (7.7) is in the same universality class as a flux line array with point disorder defined via the two-dimensional Sine-Gordon model with random phase shifts [3, 4]

$$H = -\sum_{(ij)} (\phi_i - \phi_j)^2 - \lambda \sum_i \cos(\phi_i - \theta_i), \tag{7.8}$$

where $\phi_i \in [0, 2\pi)$ are phase variables, $\theta_i \in [0, 2\pi)$ are quenched random phase shifts and λ is a coupling constant. One might anticipate that both models (7.7) and (7.8) are closely related by realizing that both have the same symmetries [the energy is invariant under the replacement $n_i \to n_i + m$ ($\phi_i \to \phi_i + 2\pi m$) with being m an integer]. Close to the transition one can show that all higher order harmonics apart from the one present in the Sine-Gordon model (7.8) are irrelevant in a field theory for (7.7), which establishes the identity of the universality classes. Note, however, that far away from T_c, as for instance at zero temperature, there might be differences in the two models.

To calculate the ground states of the SOS model on a disordered substrate with general interaction function $f(x)$ we map it onto a minimum cost flow model. Let us comment, however, that the special case $f(x) = |x|$ can be mapped onto the interface problem in the random bond Ising ferromagnet in 3d with *columnar* disorder [5] (i.e. all bonds in a particular direction are identical), by which it can be treated with the maximum flow algorithm we have discussed already (see Chap. 6).

We define a network G by the set of nodes N being the sites of the dual lattice of our original problem (which is simply made of the centers of each elementary plaquette of the square lattice, thus being again a square lattice) and the set of *directed* edges A connecting nearest neighbor sites (in the dual lattice) (i, j) and (j, i). If we have a set of height variables n_i we define a flow $\mathbf{x}$ in the following way. Suppose two neighboring sites i and j have a positive (!) height difference $n_i - n_j > 0$. Then we assign the flow value $x_{ij} = n_i - n_j$ to the directed edge (i, j) in the dual lattice, for which the site i with the larger height value is on the right hand side, and assign zero to the opposite edge (j, i), i.e. $x_{ji} = 0$. Also $x_{ij} = 0$ whenever sites i and j are of the same height. See Fig. 7.6 for a visualization of this scheme. The flow pattern is made up of closed cycles that separate regions of different height and therefore we have:

$$\forall i \in N : \quad \sum_{\{j \,|\, (i,j) \in A\}} x_{ij} = \sum_{\{j \,|\, (j,i) \in A\}} x_{ji}. \tag{7.9}$$

On the other hand, for an arbitrary set of values for x_{ij} the constraint (7.9) has to be fulfilled in order to be a flow, i.e. in order to allow a reconstruction of height variables out of the height differences. This observation becomes immediately clear by looking at Fig. 7.6.

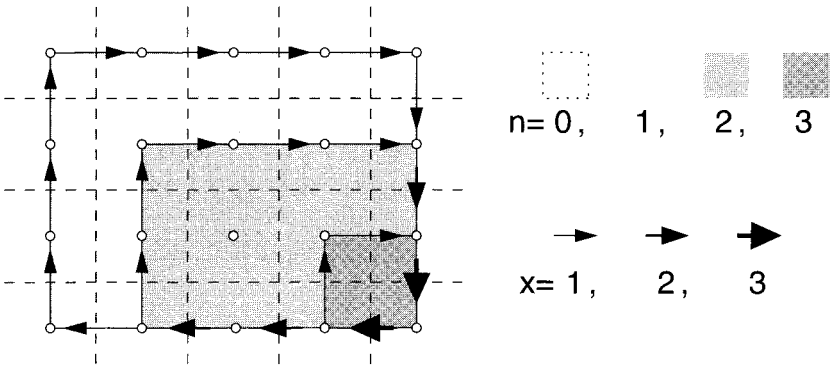


Figure 7.6: The flow representation of a surface (here a "mountain" of height $n_i = 3$). The broken lines represent the original lattice, the open dots are the nodes of the dual lattice. The arrows indicate a flow on the dual lattice, which results from the height differences of the variables n_i on the original lattice. Thin arrows indicate a height difference of $x_{ij} = 1$, medium arrows $x_{ij} = 2$ and thick arrows $x_{ij} = 3$. According to our convention the larger height values are always on the *right* of an arrow. Observe that on each node the mass balance constraint (7.9) is fulfilled.

We can rewrite the energy function (7.7) as

$$H(\mathbf{x}) = \sum_{(i,j)} (x_{ij} - d_{ij})^2 \,, \tag{7.10}$$

with $d_{ij} = d_i - d_j$. Thus our task is to minimize $H(\mathbf{x})$ under the constraint (7.9). In Sec. 7.4 we will show how this can be achieved with a refined version of the successive shortest path algorithm we know already from the N-line problem.

Vortex glass in 3d

The starting point of a field theory for superconductors is the Ginzburg-Landau theory containing the phase variables (or XY-spins) for the local superconducting order parameter. As is well known from the classical XY-model [6] the spin-wave degrees of freedom of such a model can be integrated out and one is left with an effective Hamiltonian for the topological defects, the vortices, which are the singularities of the phase field θ interacting with one another like currents in the Biot-Savat law from classical electrodynamics, i.e. like $1/r$, where $1/r$ is the distance (see also [7]). An additional integration over the fluctuation vector potential sets a cutoff for this long-range interaction beyond which the interaction decays exponentially, and can thus be neglected. Thus a standard model for interacting magnetic flux lines in high temperature superconductors [8] is, in the vortex representation, the Hamiltonian [9]

$$H = -\frac{1}{2} \sum_{(ij),(mn)} (x_{ij} - b_{ij}) K(\mathbf{r}_{(ij)} - \mathbf{r}_{(nm)}) (x_{mn} - b_{mn}) \tag{7.11}$$

where the sum runs over all pairs of bonds of a simple cubic lattice (thus it is a three-dimensional model!) and the integer variables x_{ij}, assigned to these bonds, can take on any integer but have to satisfy the divergence free condition

$$(\nabla \cdot \mathbf{x})_i = 0 \tag{7.12}$$

on every site i, the same as in the two-dimensional SOS model (7.9). The b_{ij} are magnetic fields which are constructed from quenched random vector potentials $\mathbf{A}$ by a lattice curl, i.e. one obtains b_{ij} as $1/(2\pi)$ times the directed sum of the vector potentials on the plaquette surrounding the link (ij) on which b_{ij} lives. By definition, the magnetic fields satisfy the divergence free condition $(\nabla \cdot \mathbf{b})_i = 0$ on every site i, since they stem from a lattice curl. The vortex interaction is given by the lattice Green's function

$$K(\mathbf{r}_{(ij)} - \mathbf{r}_{(nm)}) = J\frac{(2\pi)^2}{L^3} \sum_{\mathbf{k}\neq 0} \frac{1 - \exp(\mathrm{i}\mathbf{k} \cdot (\mathbf{r}_i - \mathbf{r}_j))}{2\sum_{n=1}^{3}[1 - \cos(k_n)] + \lambda_0^{-2}}, \tag{7.13}$$

where r_{ij} is the position of link (ij) in real space, J an interaction strength, $\mathbf{k} = (k_1, k_2, k_3)$ the reciprocal lattice vectors, λ_0 the screening length and L the linear lattice size of a simple cubic lattice.
In the strong screening limit $\lambda_0 \to 0$, $K(\mathbf{r})$ reduces to $K(0) = 0$ for $\mathbf{r} = 0$ and $K(\mathbf{r}) = J(2\pi\lambda_0)^2$ for $\mathbf{r} \neq 0$ with exponentially small corrections [10]. Thus if we subtract $J(2\pi\lambda_0)^2$ from the interaction and measure the energy in units of $J(2\pi\lambda_0)^2$, one obtains the simpler Hamiltonian

$$H_V = \frac{1}{2}\sum_{(ij)}(x_{ij} - b_{ij})^2. \tag{7.14}$$

Thus we have to minimize (7.14) under the mass balance constraint (7.12), which is a convex minimum-cost-flow problem.

7.4 General Minimum-cost-flow Algorithms

In this section we will discuss the algorithm to calculate the ground state of the convex minimum cost problem as it occurs in the physical context described in 7.3. It is a straightforward generalization of the successive shortest path algorithm for the N-line problem described in 7.2.
Let $G(X, E)$ be a network with a *cost* c_{ij} and a *capacity* u_{ij} associated with every edge $(i, j) \in E$. Moreover, we associate with each node $i \in X$ a number $b(i)$ which indicates its *supply* or *demand* depending on whether $b(i) > 0$ (a source) or $b(i) < 0$ (a target). The *minimum-cost-flow problem* is [11]:

$$\text{Minimize} \quad z(\mathbf{x}) = \sum_{(i,j)\in A} h_{ij}(x_{ij}), \tag{7.15}$$

subject to the mass-balance constraints,

$$\sum_{\{j|(i,j)\in A\}} x_{ij} - \sum_{\{j|(j,i)\in A\}} x_{ji} = b(i) \qquad \forall i \in X, \tag{7.16}$$

and the capacity constraints

$$0 \leq x_{ij} \leq u_{ij} \qquad \forall (i,j) \in E. \tag{7.17}$$

The parameters $b(i)$, c_{ij} are integers, while x_{ij}, u_{ij} are non-negative integers. Note that we have allowed for a general set of sources and sinks $b(i)$, though we will restrict attention to one source, s, and one sink, t, in the applications. In fact, it is easy to convert a problem with a general set $b(i)$ to one with just one source s and one target t in the following way:
(i) Connect all of the nodes with $b(i) > 0$ to s by edges with capacity $u_{si} = b(i)$
(ii) Connect all of the nodes with $b(i) < 0$ to t by edges with capacity $u_{it} = -b(i)$.
Flow conservation requires $\sum_i b(i) = 0$ so that the flow into the network is equal to the flow out of the network.
The quantity $h_{ij}(x_{ij})$ is the cost function and may be a different function on each bond. The cost functions $h_{ij}(x_{ij})$ can be any convex function, that is,

$$\forall x, y \text{ , and } \theta \in [0,1] \quad h_{ij}(\theta x + (1-\theta)y) \leq \theta h_{ij}(x) + (1-\theta)h_{ij}(y) \tag{7.18}$$

Linear cost is the special case, $h(x_{ij}) = c_{ij}x_{ij}$. There are faster algorithms for linear cost than for convex cost, but for the applications considered here, the difference is not large. The general convex-cost case is as simple to discuss as the linear-cost case, so we will discuss the general algorithm.
The residual network $G(\mathbf{x})$, corresponding to a flow, $\mathbf{x}$, is defined as in the N-line problem. However the *residual costs* need to be constructed differently due to the non-linearity of the local cost functions $h_{ij}(x_{ij})$. We need to know the cost of augmenting the flow on arc (i,j), when there is already a flow x_{ij} in that edge. In the general convex-cost problem, we *always augment the flow by one flow unit*. Because we have defined $x_{ij} \geq 0$ and $x_{ji} \geq 0$, we must treat three cases:
(i) If $x_{ij} \geq 1, \Rightarrow x_{ji} = 0$

$$\begin{aligned} c^r_{ij}(x_{ij}) &= h_{ij}(x_{ij}+1) - h_{ij}(x_{ij}), \\ c^r_{ji}(x_{ij}) &= h_{ij}(x_{ij}-1) - h_{ij}(x_{ij}), \end{aligned} \tag{7.19}$$

(ii) If $x_{ji} \geq 1, \Rightarrow x_{ij} = 0$

$$\begin{aligned} c^r_{ji}(x_{ji}) &= h_{ji}(x_{ji}+1) - h_{ji}(x_{ji}), \\ c^r_{ij}(x_{ji}) &= h_{ji}(x_{ji}-1) - h_{ji}(x_{ji}), \end{aligned} \tag{7.20}$$

(iii) If $x_{ij} = 0$, and $x_{ji} = 0$

$$\begin{aligned} c^r_{ij}(0) &= h_{ij}(1) - h_{ij}(0), \\ c^r_{ji}(0) &= h_{ji}(1) - h_{ji}(0). \end{aligned} \tag{7.21}$$

As seen in the second part of Eqs. (7.19) and (7.20), negative residual costs may occur when reducing the flow in an edge. The residual network $G(\mathbf{x})$ is then a graph with residual capacities $r_{ij} = u_{ij} - x_{ij} + x_{ji}$, and residual costs found from Eqs. (7.19)-(7.21). An intuitively appealing way of thinking about the convex-cost problem is to replicate each edge, (i,j) many times, with each replicated edge having capacity one. The kth replicated edge has cost $h_{ij}(k) - h_{ij}(k-1)$. As flow is pushed along the edge (i,j), the first unit of flow goes into the 1st replicated edge, the 2nd unit of flow

in the 2nd replicate etc. When the flow is reversed, the flow is canceled first in the highest replicated edge *provided the cost function is convex*. That is, we need $h_{ij}(k) - h_{ij}(k-1) > h_{ij}(k-1) - h_{ij}(k-2)$ so that this replication procedure makes sense. Unfortunately no analogous procedure is possible when the convexity condition is violated. In the case of linear costs there is no need to replicate the edges as the cost for incrementing the flow does not depend on the existing flow.
We will now discuss two methods for solving minimum-cost-flow problems, namely the negative-cycle-canceling method and the successive-shortest-path method, both of which rely on residual-graph ideas. The first method starts with an arbitrary feasible solution that is not yet optimal and improves it iteratively until optimality is reached. The latter method starts with an optimal solution that violates certain constraints and fulfills the constraints one after the other keeping optimality all the time so that when all constraints are fulfilled simultaneously optimality is guaranteed. This method is more efficient, but we will discuss the negative cycle algorithm since the negative-cycle theorem presented below is needed to prove the correctness of the successive-shortest-path method.

Negative-cycle-canceling Algorithm

The idea of this algorithm is to find a *feasible flow*, that is, one which satisfies the mass-conservation rules, and then to improve its cost by canceling negative-cost cycles. A negative-cost cycle in the original network is also a negative-cost cycle in the *residual graph*, so we can work with the residual graph. Moreover, flow cycles do not change the total flow into or out of the network, and they do not alter the mass-balance conditions at each node. Thus, augmenting the flow on a negative-cost cycle maintains feasibility and reduces the cost, which forms the basis of the negative-cycle-canceling algorithm. This is formalized as follows.

Theorem: (Negative cycle)
A feasible solution $\mathbf{x}^*$ is an *optimal* solution of the minimum-cost-flow problem, *if and only if* the residual network $G(\mathbf{x}^*)$ contains *no negative-cost cycle*.

Proof: Suppose the flow $\mathbf{x}$ is feasible and $G(\mathbf{x})$ contains a negative cycle. Then a flow augmentation along this cycle improves the function value $z(\mathbf{x})$, thus $\mathbf{x}$ is not optimal. Now suppose that $\mathbf{x}^*$ is feasible and $G(\mathbf{x}^*)$ contains no negative cycles and let $\mathbf{x}^0 \neq \mathbf{x}^*$ be an optimal solution. Now decompose $\mathbf{x}^0 - \mathbf{x}^*$ into augmenting cycles, the sum of the costs along these cycles is $\mathbf{c} \cdot \mathbf{x}^0 - \mathbf{c} \cdot \mathbf{x}^*$. Since $G(\mathbf{x}^*)$ contains no negative cycles $\mathbf{c} \cdot \mathbf{x}^0 - \mathbf{c} \cdot \mathbf{x}^* \geq 0$, and therefore $\mathbf{c} \cdot \mathbf{x}^0 = \mathbf{c} \cdot \mathbf{x}^*$ because optimality of $\mathbf{x}^*$ implies $\mathbf{c} \cdot \mathbf{x}^0 \leq \mathbf{c} \cdot \mathbf{x}^*$. Thus $\mathbf{x}^0$ is also optimal. QED
A minimum-cost algorithm based on the negative-cost-canceling theorem, valid for graphs with convex costs and no negative-cost cycles in $G(0)$, is given below, an example is presented in Fig. 7.7.

algorithm cycle canceling (convex costs)
begin
  establish a feasible flow $\mathbf{x}$;
  calculate the residual costs c^r_{ij} as in Eqs. (3.11-3.13);
  while $G(\mathbf{x})$ contains a negative cost cycle **do**
  begin
    use some algorithm to identify a negative cycle W;
    augment *one* unit of flow in the cycle;
    update $c^r_{ij}(x_{ij})$ and r_{ij};
  end
end

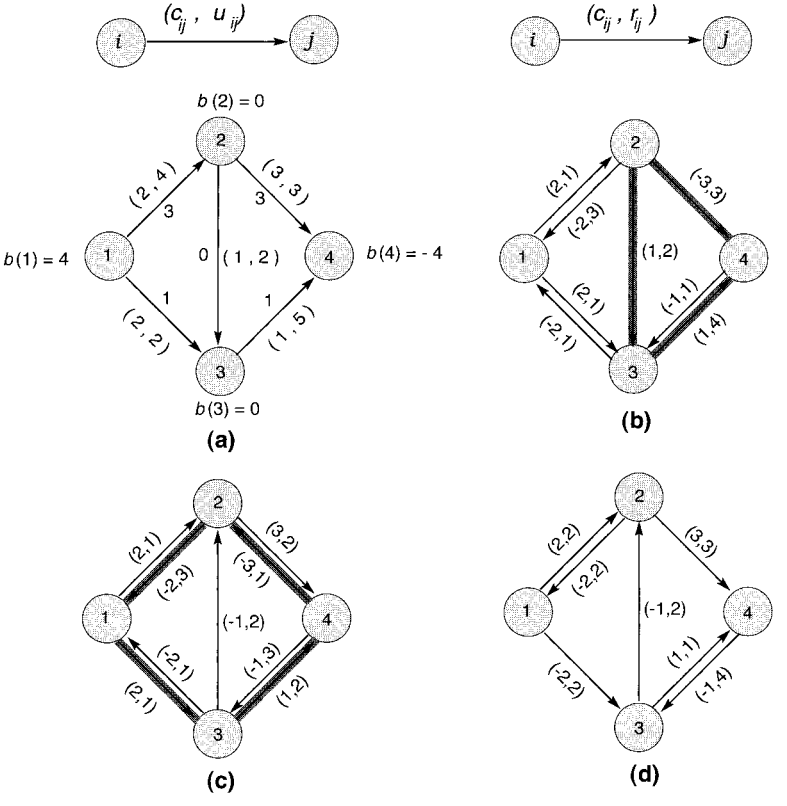


Figure 7.7: Illustrating the cycle canceling algorithm for linear costs: (a) network example with a feasible flow $\mathbf{x}$; (b) residual network $G(\mathbf{x})$; (c) residual network after augmenting 2 units along the cycle 4–2–3–4; (d) residual network after augmenting 1 unit along the cycle 4–2–1–3–4.

To begin the algorithm, it is necessary to find a feasible flow, which is a flow which satisfies the injected flow at each of the sources, the extracted flow at each of the sinks, and which satisfies the mass-balance constraints at each node. A robust procedure to find a feasible flow is to find a flow which satisfies the capacity constraints using the maximum-flow algorithm (see Chap. 6). To detect negative cycles in the residual network, $G(\mathbf{x})$, one can use the label-correcting algorithm for the shortest-path problem presented Sec. 4.2.3.
In the linear cost case, the maximum possible improvement of the cost function is $\mathcal{O}(|E|CU)$, where $C = \max|c_{ij}|$ and $U = \max u_{ij}$. Since each augmenting cycle contains at least one edge and at least one unit of flow, the upper bound on the number of augmenting cycle iterations for convergence is also $\mathcal{O}(|E|CU)$. Negative-cycle detection is $\mathcal{O}(|X^2|)$ generically, but for sparse graphs with integer costs it is $\mathcal{O}(|E)|)$. Thus for sparse graphs with integer costs, negative cycle canceling is $\mathcal{O}(|E|^2CU)$.

Successive-shortest-path algorithm

The successive-shortest-path algorithm iteratively sends flow along minimal-cost paths from source nodes to sink nodes to finally fulfill the mass-balance constraints. A *pseudo flow* satisfies the capacity and non-negativity constraints, but not necessarily the mass-balance constraints. Such flows are called *infeasible* as they do not satisfy all the constraints. The successive-shortest-path algorithm is an *infeasible* method which maintains an optimal-cost solution at each step. In contrast, the negative-cycle-canceling algorithm always satisfies the constraints (so it is a *feasible* method) and it iteratively produces a more optimal solution.
The *imbalance* of node i is defined as

$$e(i) = b(i) + \sum_{\{j|(ji)\in A\}} x_{ji} - \sum_{\{j|(ij)\in A\}} x_{ij}. \tag{7.22}$$

If $e(i) > 0$ then we call $e(i)$ the *excess* of node i, if $e(i) < 0$ then we call it the *deficit*. The successive-shortest-path algorithm sends flow along minimal-cost paths from excess sites to deficit sites until no excess or deficit nodes remain in the graph.
Dijkstra's algorithm (Sec. 4.2.2) is efficient in finding minimum-cost paths, but it only works for *positive costs*. The successive-shortest-path algorithm uses Dijkstra's method to find augmenting paths, but to make this work we have to develop a different sort of residual network with positive *reduced costs* [remember that the residual costs can be negative — see Eqs. (7.19-7.21)]. Surprisingly, this is possible. To construct positive reduced costs from which the optimal flow can be calculated, we use the concept of *node potential* $\pi(i)$ already encountered in Sec. 7.2.
The reduced costs used in the successive-shortest-path problem are inspired by the reduced costs, c_{ij}^d, introduced in the shortest-path problem [Eq. (4.3)]. c_{ij}^d has the attractive feature that, with respect to the optimal distances, every arc has a non-negative cost. To generalize the definition (4.3) so that it can be used in the minimum-cost-flow problem, one defines the *reduced cost* of edge (i, j) in terms of a set of *node potentials* $\pi(i)$,

$$c_{ij}^{\pi} = c_{ij}^r - \pi(i) + \pi(j). \tag{7.23}$$

We impose the condition that *a potential is only valid* if $c_{ij}^{\pi} \geq 0$ as for reduced costs in the minimal-path problem. Note that the *residual costs* defined in Eqs. (7.19–7.21) appear here. All of the quantities in Eq. (7.23) depend on the flow x_{ij}, though we do not explicitly state this dependence.
From the definition (7.23), we obtain for c_{ij}^{π}
(i) For any directed path P from k to l:

$$\sum_{(i,j)\in P} c_{ij}^{\pi} = \sum_{(i,j)\in P} c_{ij}^{r} - \pi(k) + \pi(l).$$

(ii) For any directed cycle W:

$$\sum_{(i,j)\in W} c_{ij}^{\pi} = \sum_{(i,j)\in W} c_{ij}^{r}. \tag{7.24}$$

In particular, property (ii) means that negative cycles with respect to c_{ij}^{r} are also negative cycles with respect to c_{ij}^{π}. We define the residual network $G^{\pi}(\mathbf{x})$ to be the residual graph with residual capacities defined as before, but with reduced costs as given by Eq. (7.23).
The next step is to find a way to construct the potentials $\pi(i)$. This is carried out recursively, starting with $\pi(i) = 0$ when there is no flow in the network. The procedure for generating potentials iteratively relies on the *potential* lemma given below.

Lemma: (Potential)
(i) Given: a valid node potential $\pi(i)$: a set of reduced costs c_{ij}^{π} and a set of distance labels, $d(i)$ (found using $c_{ij}^{\pi} \geq 0$) *then* the potential $\pi'(i) = \pi(i) - d(i)$ also has positive reduced costs, $c_{ij}^{\pi'} \geq 0$.
(ii) $c_{ij}^{\pi'} = 0$ for all edges (i,j) on shortest paths.

Proof: Properties (i) and (ii) follow from the analogous properties for the minimal path [see Eqs. (4.3–4.5)]. To prove (i), using (4.3–4.5), we have, $d(j) \leq d(i) + c_{ij}^{\pi}$, then we have, $c_{ij}^{\pi'} = c_{ij}^{r} - [\pi(i) - d(i)] + [\pi(j) - d(j)] = c_{ij}^{\pi} - d(j) + d(i) \geq 0$. For (ii) simply repeat the discussion after replacing the inequality by an equality. QED
Now that we have a method for constructing potentials, it is necessary to demonstrate that this construction produces an optimal flow.

Theorem: (Reduced cost optimality)
A feasible solution, $\mathbf{x}^*$, is an optimal solution of the min-cost flow problem *if and only if* there exists a set of node potentials, $\pi(i)$, such that $c_{ij}^{\pi} \geq 0 \quad \forall (i,j)$ in $G^{\pi}(\mathbf{x}^*)$.

Proof: For the implication "$\Leftarrow$" suppose that $c_{ij}^{\pi} \geq 0 \, \forall (i,j)$. Because of (7.24) it contains no negative cycles. Thus by property (ii) above, an arbitrary potential difference may be added to the costs on each edge on each cycle W. For the other direction "$\Rightarrow$" suppose that $G(\mathbf{x}^*)$ contains no negative cycles. Denote with $d(\cdot)$ the shortest path distances from node 1 to all other nodes. Hence $d(j) \leq d(i) + c_{ij} \; \forall (i,j) \in G(\mathbf{x}^*)$. Now

define $\pi = -d$ then $c_{ij}^{\pi} = c_{ij} + d(i) - d(j) \geq 0$. Hence we have constructed a set of node potentials associated with the optimal flow. QED

The reduced cost optimality theorem proves that each optimal flow has an associated set of potentials, while the potential lemma shows how to construct these potentials. The final step is to demonstrate how to augment the flow using the potentials. To demonstrate this, suppose that we have an optimal flow $\mathbf{x}$ and its associated potential $\pi(i)$ which produces reduced costs c_{ij}^{π} that satisfy the reduced cost optimality condition. Suppose that we want to add one unit of flow to the system, injecting at a source at site, k, and extracting at a site l. Find a minimal path, P_{kl}, (using the reduced costs c_{ij}^{π}) from an excess site k to a deficit site l. Now augment the flow by one unit for all edges $(i,j) \in P_{kl}$. We call this flow augmentation δ. The following *augmentation* lemma ensures that this procedure maintains optimality.

Lemma: (Flow augmentation)
The flow $\mathbf{x}' = \mathbf{x} + \delta$ is optimal *and* it satisfies the reduced cost optimality conditions.

Proof: Take π and π' as in the potential Lemma and let P be the shortest path from node s to node k. Part (ii) of the potential lemma implies that $\forall\,(i,j) \in P:\ c_{ij}^{\pi'} = 0$. Therefore $c_{ji}^{\pi'} = -c_{ij}^{\pi'} = 0$. Thus a flow augmentation on $(i,j) \in P$ might add (j,i) to the residual network, but $c_{ji}^{\pi'} = 0$, which means that still the reduced cost optimality condition $c_{ji}^{\pi'} \geq 0$ is fulfilled. QED

The strategy for the successive-shortest-path algorithm is now clear. Given a set of excess nodes $E = \{i | e(i) > 0\}$ and a set of deficit nodes $D = \{i | e(i) < 0\}$, we iteratively find minimal paths from a node $i \in E$ to a node $j \in D$ until no excess or deficit remains:

algorithm successive shortest paths [convex costs]
begin
  Initialize $\mathbf{x}$ and π such that the reduced costs $c^{\pi}(\mathbf{x}) \geq 0$;
  while there is a node s with $e(s) > 0$ **do**
  begin
    compute the reduced costs $c^{\pi}(\mathbf{x})$(Eq. 3.15);
    find the shortest paths $d(i)$ from s to
      all other nodes in $G(\mathbf{x})$ w.r. to the reduced costs c_{ij}^{π};
    choose a node t with $e(t) < 0$;
    augment flow on the shortest path from s to t by *one* unit;
    compute $\pi(i) := \pi(i) - d(i)$;
  end
end

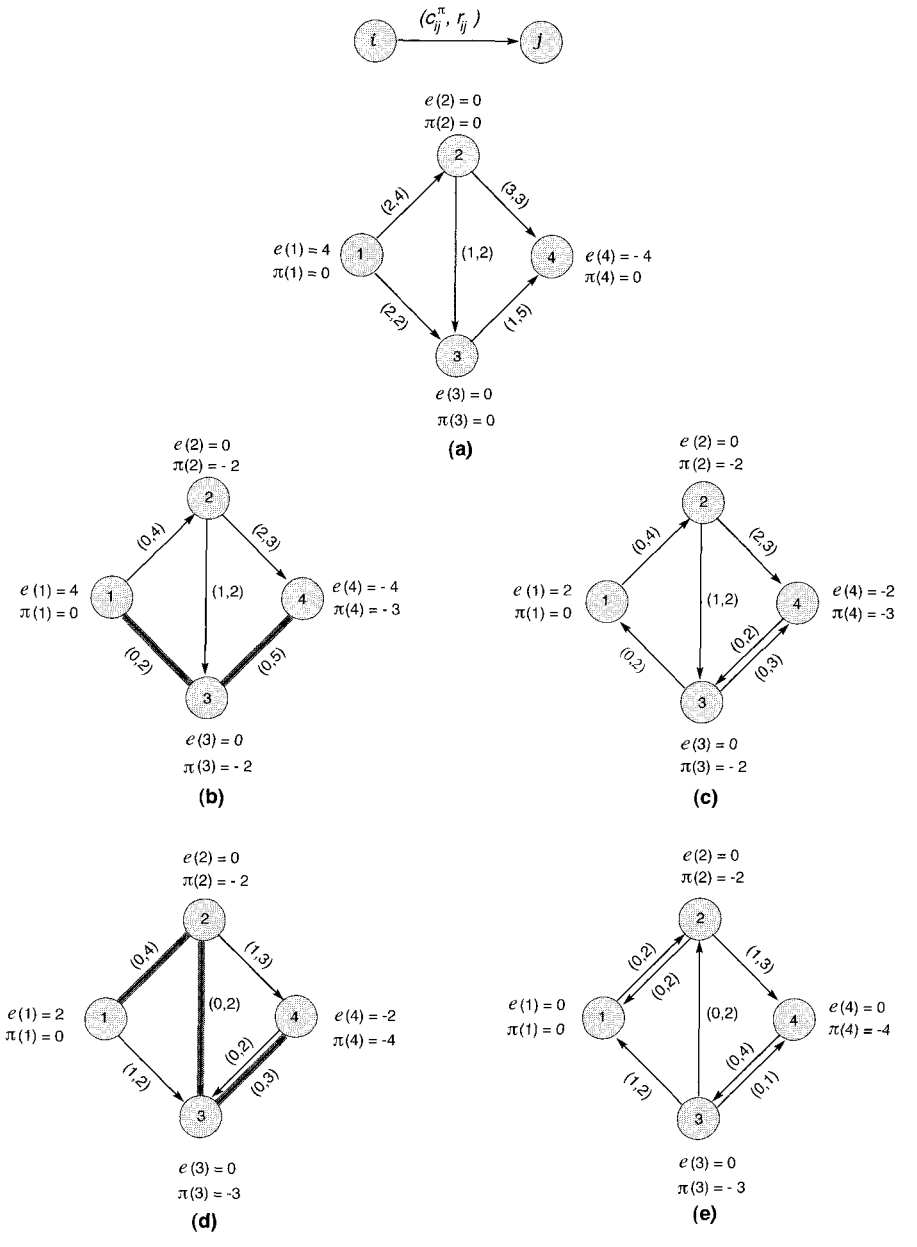


Figure 7.8: Illustrating the successive-shortest-path algorithm: (a) initial network for $\mathbf{x} = 0$ and $\pi = 0$; (b) network after updating the potentials π; (c) network after augmenting 2 units along the path 1–2–3; (d) network after updating the potentials π; (e) network after augmenting 2 units along the path 1–2–3–4.

The minimal path for flow augmentation is found using Dijkstra's method on the residual network with the reduced costs given by Eq. (7.23). After each flow augmentation, the node potentials are recalculated using the potential lemma. We demonstrate the algorithm in Fig. 7.8, where for simplicity we use a *linear cost function*.
Since we worked hard to construct a system with positive reduced costs, the "find" operation above can be carried out using Dijkstra's algorithm. If we denote the sum of all the sources to be $v = \sum_{i|b(i)>0} b(i)$, then the number of flow augmentations needed to find the optimal flow is simply v. Each flow augmentation requires a search for a minimum-cost path from a node $k \in E$ to a node $l \in D$ which for sparse graphs and integer flows can be efficiently accomplished with Dijkstra's method, which is $\mathcal{O}(|E|)$. Thus for integer flows on sparse graphs with positive costs (as is typical of the physics applications) the successive-shortest-path algorithm is $\mathcal{O}(v|E|)$.
A final note on the initialization statement: for the physical problems we presented in the preceding sections (7.2 and 7.3) it is not hard to find a flow $\mathbf{x}$ and a note potential π that fulfills the requirement that the reduced costs $c^{\pi}(\mathbf{x}) = c_{ij}(x_{ij}) + \pi_i - \pi_j \geq 0$ are all non-negative. In the N-line problem it is simply $\mathbf{x} = \mathbf{0}$ and $\pi = \mathbf{0}$, i.e. the system without any flux line (FL), since all bond energies are non-negative. In the convex flow problem with a general local costs $h_{ij}(x_{ij})$ one just chooses the integer x_{ij} that is closest to the minimum of $h_{ij}(x)$, for the specific examples of 7.3, where $h_{ij}(x_{ij}) = (x_{ij} - d_{ij})^2$ it is simply the integer that is closest to the real number d_{ij}. With this configuration $c_{ij}(x_{ij}) \geq 0$ and with $\pi = \mathbf{0}$ also the reduced costs are non-negative.

7.5 Miscellaneous Results for Different Models

In this section we will present a number of results that highlight the flexibility of the described methods that allows not only study of the ground state of the model itself but also of the energetics of excitations and topological defects.

Flux-line array in a periodic potential

As one example for typical flux-line (FL) problems we demonstrate here how one can study the competition between point disorder and a periodic potential. Depending on the strength of the disorder with respect to the depth of the periodic potential one may or may not have a roughening transition. As a starting point we use the continuum model (7.1) with a periodic potential V_{p} and write down an appropriate lattice model for it. Here we study the situation in which each potential valley is occupied by one line and its minima are well localized, i.e. they have a width that is small against the interaction range of the lines.
We use the N-line model defined in Sec. 7.2 and simply modify the bond energies e_{ij} in (7.2) appropriately: to the uncorrelated bond energy variables, taken from some probability distribution $P(\epsilon_{ij})$ with variance ϵ, we add a periodic part setting $e_{ij} = \epsilon_{ij} + \Delta_{ij}$. The structure of the periodic part resembles periodically arranged *columnar defects* and is depicted in Fig. 7.9, where one has valleys of effective depth

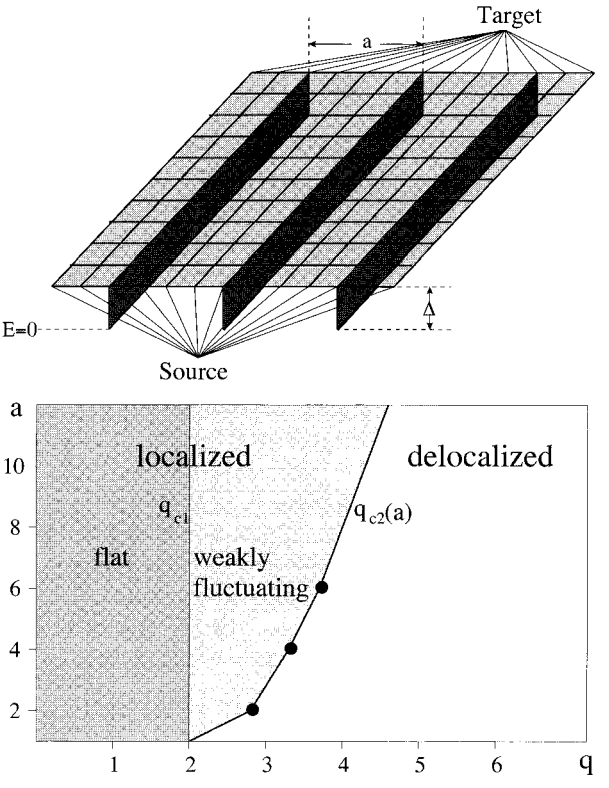


Figure 7.9: Top: Periodic potential in 2d. The depth of the valleys is denoted by Δ and the nearest neighbor distance by a. Additional point disorder ϵ accomplishes the energy landscape. The FL can only enter and leave the system via the energetically neutral edges connecting the source and the sink, respectively, with the potential valleys. **Bottom:** Schematic phase diagram in 2d and 3d for bounded disorder. In the case of unbounded disorder the flat phase vanishes.

$-\Delta$ such that the Δ_{ij} values are zero inside the potential wells and constant elsewhere. This also reproduces the elastic energy, since all bonds cost some positive energy, defining the ratio disorder strength and the depth of the potential valleys as $q = \epsilon/\Delta$.

Figure 7.10 demonstrates with a series of snapshots the geometry involved in the calculations, and the typical behavior with increasing g in 2d and 3d. In both cases the lines are pinned to the energetically favorable valleys for small q, and finally for large q a cross-over to a rough state takes place. In 3d one can observe that the lines wander almost freely under such conditions. The examples of Fig. 7.10 represent different regions in the a-q phase diagram, which is sketched in Fig. 7.9.

One discriminates between the different regions in the phase diagram by looking at

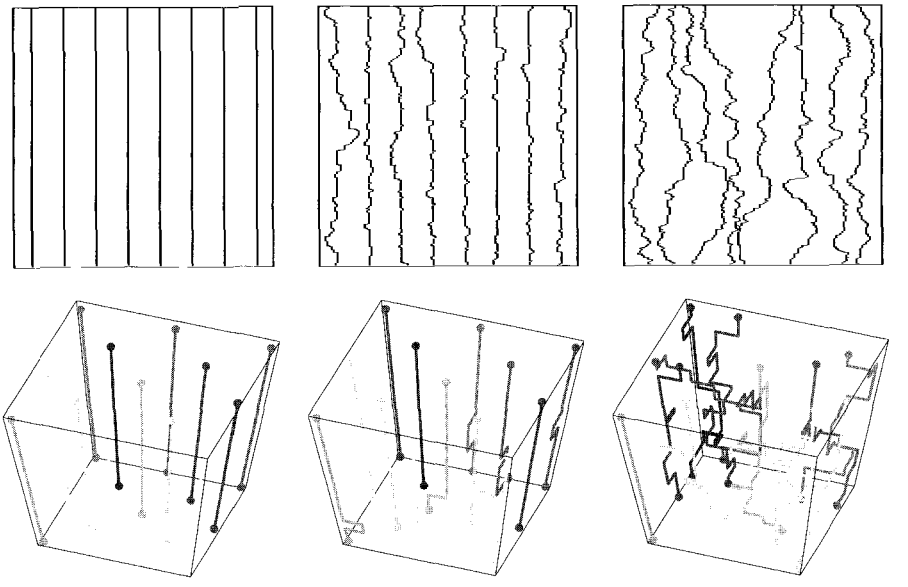


Figure 7.10: Optimal ground state configurations in 2d (top) and 3d (bottom) for different point disorder strengths q, increasing from left to right. In the flat phase (left) the FL are trapped completely inside the potential valleys.

the behavior of the average transverse fluctuation or *roughness* w of the lines:

$$w(L,H) = \left[\frac{1}{N} \sum_{i=1}^{N} \frac{1}{H} \int_0^H dz \left(\mathbf{r}_i(z) - \bar{\mathbf{r}}_i \right)^2 \right]_{\mathrm{av}} , \tag{7.25}$$

where $\bar{\mathbf{r}}_i = H^{-1} \int_0^H dz\, \mathbf{r}_i(z)$ and $[\ldots]_{\mathrm{av}}$ denotes the disorder average. By studying very large longitudinal system sizes $H \geq 10^4$ we are able to extract the saturation roughness $w(L) = \lim_{H\to\infty} w(L,H)$ for a finite system of transverse size L. Note that we have chosen open boundary conditions: the transverse fluctuations cannot get larger than the system size. Other quantities of interest are the size $l_\parallel$ of the longitudinal excursions (the average distance between the locations at which a line leaves a valley and returns to it); and the total number of potential valleys PV that a line visits between its entry and terminal point in the limit $H \to \infty$.
In Fig. 7.11 data for the roughness w and $l_\parallel$ as a function of $1/q$ in 2d are shown. The picture that emerges is the following. In the *flat region* we have $w(L) = 0$, $l_\parallel = 0$ and $PV = 1$, i.e. the lines lie completely in the potential valleys. This region $q < q_{\mathrm{c1}}$ exist only for *bounded* disorder. For the uniform distribution no energetically favorable transverse fluctuation can exist as long as $q < \Delta$. That $q_{\mathrm{c1}} > 1$ follows from the fact that we are at full occupancy, $N = N_{\mathrm{V}}$ where N_{V} is the number of valleys, for $q \leq q_{\mathrm{c1}} \sim 2$ the ground state consists always of N straight lines regardless of dimension.

For *unbounded* disorder this flat region does not exist, since the probability for a sequence of high-energy bonds in the valleys that pushes the lines out of it is always positive. In the *weakly fluctuating region* for $q_{c1} \leq q \leq q_{c2}$ the lines roughen locally. Here one has $w > 0$ and $l_{\|} > 0$, independent of the systems size L, and $PV = 1$. The transverse fluctuations of flux lines are bounded by the average line distance or valley separation a. The central feature is that lines fluctuate individually, so that a columnar defect competes with point disorder. Both in 2d and in 3d a strong columnar pin strictly localizes the line [12] reducing the line-to-line interaction to zero. More details of this analysis can be found in [13].

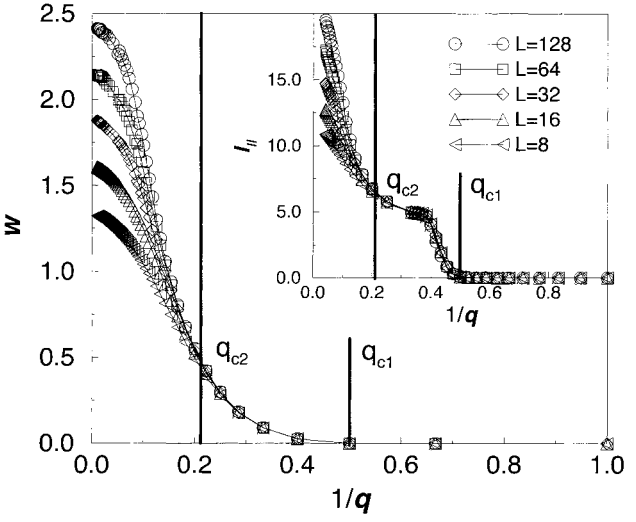


Figure 7.11: Roughness w in 2d as a function of disorder strength q for bounded disorder. q_{c1} and q_{c2} are shown. In the flat phase $w = 0$, whereas $w > 0$ for $q > q_{c1}$. No transversal system size dependence is observed in the weakly fluctuating phase. The inset shows $l_{\|}$. Each data point is averaged over $n = 20$ ($L = 128$) up to $n = 600$ ($L = 8$) disorder configurations, $a = 4$.

Dislocations in the SOS model

In Sec. 7.3 we mapped the SOS Hamiltonian (7.7) onto the convex cost function (7.10) $H(\mathbf{x}) = \sum_{(i,j)}(x_{ij} - d_{ij})^2$ with the constraint (7.9) $(\nabla \cdot \mathbf{x})_i = 0$. Without this constraint the configuration $\mathbf{x}$ with the lowest value for $H(\mathbf{x})$ is easy to find (and actually is the one with which the successive shortest path algorithm starts): for each bond (ij) choose the integer x_{ij} as the one that is closest to the real number d_{ij}. Obviously, for a typical disordered substrate the minimal configuration $\{\mathbf{x}\}_{min}$ violates the mass

balance constraint (7.9). Figure 7.12 shows an example of a disordered substrate with substrate height $d_i \in \{0.0, 0.2, 0.4, 0.6\}$. Consider the differences d_{ij}: across the dashed line we have $d_{ij} = 0.6$ and $|d_{ij}| < 0.5$ elsewhere. Consequently, the absolute minimum-energy configuration without any balance constraint is given by $x_{ij} = 1$ and $x_{ij} = 0$, respectively.

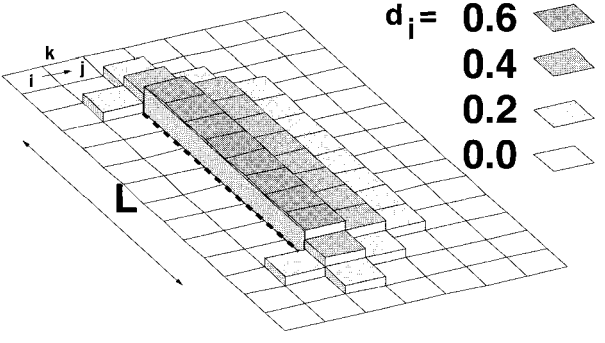


Figure 7.12: Example of disordered substrate heights d_i in a random-surface model with a single dislocation pair connected along a straight line of size L (dashed line). The optimal surface without dislocations would be flat, i.e. $n_i = 0$ for all sites i, however, allowing dislocations would decrease the ground-state energy (see text).

With respect to the balance constraint (7.9) the only feasible optimal solution (ground state) is a flat surface, i.e. $x_{ij} = 0$ for all links (ij). On the other hand, dislocations can be introduced if one treats the height field h_i as a multi-valued function which may jump by 1 along lines that connect two point defects (i.e. a dislocation pair) [14]. Therefore, for the given example (Fig. 7.12) it should be clear that the minimal configuration $\{\mathbf{x}\}_{min}$ (see above) is exactly the optimal (i.e. ground state) configuration with one dislocation pair. One of the two defects has a Burgers charge $b = +1$ and the other one $b = -1$. The pair is connected by a dislocation line (dashed line in Fig. 7.12) along which one has $x_{ij} = 1$. This already demonstrates that due to the disorder the presence of dislocations decreases the ground state energy and a proliferation of defects appears. Alternatively one can introduce a dislocation pair by fixing the boundary to zero and one [15].
The defect pairs in the disordered SOS model are source and sink nodes of strength $+b$ and $-b$, respectively, for the network flow field x_{ij}, which otherwise fulfills $(\nabla \cdot \mathbf{x})_i = 0$, i.e. we have to modify the mass balance constraint (7.9) as follows

$$(\nabla \cdot \mathbf{x})_i = \begin{cases} 0 & \text{no dislocation at } i \\ \pm b & \text{dislocation at } i \end{cases} \tag{7.26}$$

Thus the ground-state problem is to minimize the Hamiltonian (7.10) subjected to the mass balance constraint (7.26). In the following we concentrate on defect pairs with $b = \pm 1$.

The defect energy ΔE is the difference of the minimal energy configuration *with* and *without* dislocations for each disorder realization, i.e. $\Delta E = E_1 - E_0$. More precisely, for the configuration *with* N defect pairs of Burgers charge $b = \pm 1$ we introduce two extra nodes s and t with source strength $n_s = +N$ and $n_t = -N$, respectively, and connect them via external edges or bonds with particular sites of the lattice depending on the degree of optimization: (a) with two sites separated by $L/2$ [Fig. 7.13(a)], (b) the source node with one site i and the sink node with the sites on a circle of radius $L/2$ around i [Fig. 7.13(b)] and (c) both nodes with the whole lattice. Case (a) corresponds to a *fixed* defect pair, (b) to a *partially optimized* pair along a circle, both separated by a distance $L/2$, and (c) to a *completely optimized* pair with an arbitrary separation. In all cases the energy costs for flow along these external edges are set to a positive value in order to ensure the algorithm will find the optimal defect pair on the chosen sites. These "costs" have no contribution to the ground-state energy. In case of *multi pairs* we always use graph (c). Here the optimal number N of defects in the system is gradually determined starting with one pair ($N = 1$) with a vortex core energy $2E_c$ and checking whether there is an energy gain or not. If yes, add a further pair (with $2E_c$) and repeat the procedure until there is no energy gain from the difference of the ground-state energy between two iterations.

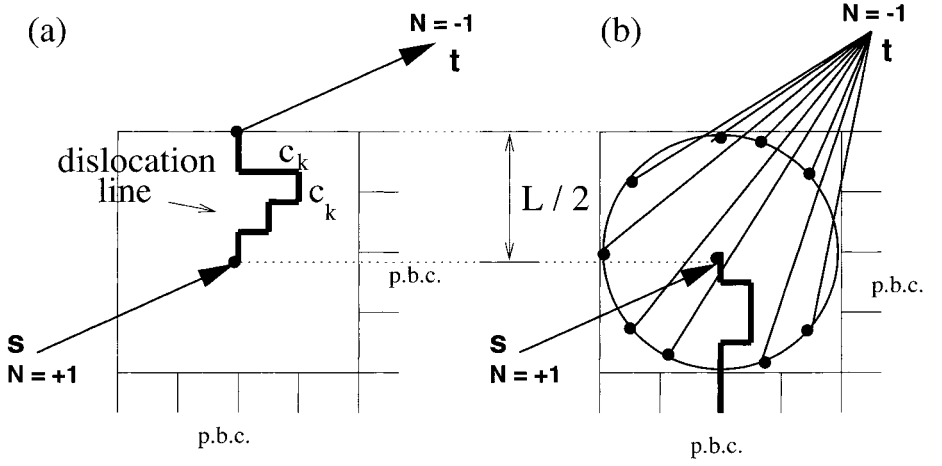


Figure 7.13: Graph of an $L \times L$ lattice with periodic boundary conditions (p.b.c.) for the implementation **(a)** of one *fixed* defect pair and **(b)** of a *partially optimized* pair. Both are separated by $L/2$. Dislocations are induced by two extra nodes s and t, which are connected with the possible positions of the defects (big dots).

One can study the defect energy ΔE and its probability distribution $P(\Delta E)$ on an $L \times L$ lattice with $L = 6$, 12, 24, 48, 96, 192 and $2 \cdot 10^3 - 10^5$ samples for each size and consider the three cases (a)-(c) (see earlier). With an increasing degree of optimization a negative defect energy ΔE becomes more probable and its probability distribution

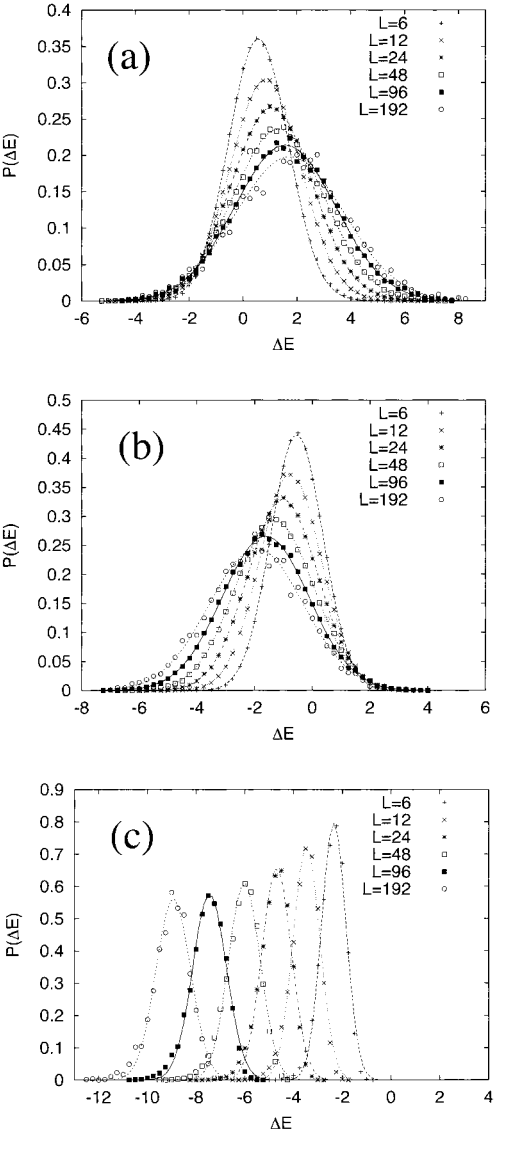


Figure 7.14: Probability distribution $P(\Delta E)$ of a large-scale topological excitation with a Gaussian fit for different optimizations: (a) for a *fixed* defect pair, (b) for a *partially optimized* pair and (c) for a *completely optimized* pair with different system sizes L.

$P(\Delta E)$ differs more and more from the Gaussian fit, Fig. 7.14. The resulting disorder averaged defect energy $[\Delta E]_{\rm dis}$ scales like

$$[\Delta E]_{\rm dis} \sim \begin{cases} \ln(L) & \text{fixed defect pair} \\ -0.27(7) \cdot \ln^{3/2}(L) & \text{partially optimized} \\ -0.73(8) \cdot \ln^{3/2}(L) & \text{completely optimized} \end{cases} \tag{7.27}$$

From the fact that the partially and completely optimized dislocation pairs have on average a negative energy that increases in modulus for increasing distance L it follows that the ground state is unstable with respect to the formation of unbound dislocation pairs. More details can be found in [16].

Low-energy excitations in the vortex glass

As a last example we consider the vortex glass model in the strong screening limit (7.14) and study the scaling behavior of low-energy excitations $\Delta E(L)$ of length scale L (to be defined below) in the presence of an external field. The magnetic filed $\mathbf{b_i}$ can be constructed from the quenched vector potentials A_{ij} by a lattice curl, in our case with the homogeneous external field we have:

$$\mathbf{b}_i = \frac{1}{2\pi}[\nabla \times \mathbf{A}^{\rm rand}]_i + \mathbf{B}_i^{\rm ext}. \tag{7.28}$$

We choose $\mathbf{B}_i^{\rm ext} = B\,\mathbf{e}_z$ i.e. the external field points in the z-direction. The dependence of $\Delta E(L)$ on L provides the essential evidence about the stability of the ground state with respect to thermal fluctuations. If $\Delta E(L)$ decreases with increasing length L it implies that it costs less energy to turn over larger domains thus indicating the absence of a true ordered (glass) state at any $T \neq 0$. Usually one studies such excitation of length scale L by manipulating the boundary condition for the phase variables of the original gauge glass Hamiltonian [17, 18]. One induces a so called *domain wall* of length scale L into the system by changing the boundary condition of a particular sample from periodic to anti-periodic (or vice versa) in one space direction and measure the energy of such an excitation by comparing the energy of these two ground-state configurations. This is the common procedure for a domain-wall renormalization group (DWRG) analysis, as it was first introduced in connection with spin glasses (see Chap. 9), which, however, contains some technical complications [17] and some conceptual ambiguities [18, 19] in it.

Here we follow the basic idea of DWRG, we will, however, avoid the complications and the ambiguities that appear by manipulating the boundary conditions (b.c.) and try to induce the low energy excitation in a different way, as first been done by one of us in [20] for the zero-field case. First we will clarify what a low energy excitation of length scale L is: in the model under consideration here it is a global vortex loop encircling the 3d torus (i.e. the L^3 lattice with periodic b.c.) once (or several times) with minimum energy cost. How can we induce the above mentioned global vortex loop, if not by manipulating the b.c.? Schematically the solution is the following numerical procedure:

1. Calculate the exact ground-state configuration $\{\mathbf{J}^0\}$ of the vortex Hamiltonian Eq. (7.14).

2. Determine the resulting global flux along, say, the x-axis $f_x = \frac{1}{L}\sum_i J_i^{0x}$.

3. Study a minimum-cost-flow problem in which the actual costs for increasing the flow on any bond in the x-direction $\Delta c_i^x = c_i(J_i^{0x}+1) - c_i(J_i^{0x})$ is smoothly modified letting the cost of a topologically simple connected loop remain unchanged and only affecting global loops.

4. Reduce the Δc_i^x until the optimal flow configuration $\{\mathbf{J}^1\}$ for this min-cost-flow problem has the global flux $(f_x + 1)$, corresponding to the so called *elementary* low energy excitation on the length scale L.

5. Finally, the defect energy is $\Delta E = H(\{\mathbf{J}^1\}) - H(\{\mathbf{J}^0\})$.

Two remarks: 1) In the pure case this procedure would not work, since at some point spontaneously *all* links in the z-direction would increase their flow value by one. It is only for the disordered case with a continuous distribution for the random variables $\mathbf{b}_i$ that a unique loop can be expected. 2) In the presence of a homogeneous external field one has to discriminate between different excitation loops: those parallel and those perpendicular to the external field need not have the same energy.
As for the zero-field case [20] one expects for the disordered averaged excitation energy (or defect energy)

$$[\Delta E(B, L)]_{\text{av}} \sim L^{\theta}, \tag{7.29}$$

where B is fixed, $[\cdots]_{\text{av}}$ denotes the disorder average and θ is the stiffness exponent and its sign determines whether there is a finite temperature phase transition or not, as explained above. If $\theta < 0$, i.e. the transition to a true superconducting vortex state appears only at $T = 0$ [10, 20], as shown in Fig. 7.15.
For any fixed value of B the finite size scaling relation (7.29) is confirmed and gives $\theta = -0.95 \pm 0.04$, c.f. Ref. [20], independent of the field strength B. Nevertheless Fig. 7.16 shows that in one individual sample the excitation loops themselves change their form dramatically with B. Only small parts of the loop seem to persist over a significant range of the field strength, see for instance in the vicinity of the plane $z = 20$ in Fig. 7.16.

Bibliography

[1] See e.g. S.T. Chui and J.D. Weeks, *Phys. Rev. B* **14**, 4978 (1976)

[2] Y.-C. Tsai and Y. Shapir, *Phys. Rev. Lett.* **69**, 1773 (1992); *Phys. Rev. E* **40**, 3546, 4445 (1994)

[3] J. Toner and D.P. Di Vincenzo, *Phys. Rev. B* **41**, 632 (1990)

[4] J.L. Cardy and S. Ostlund, *Phys. Rev. B* **25**, 6899 (1992)

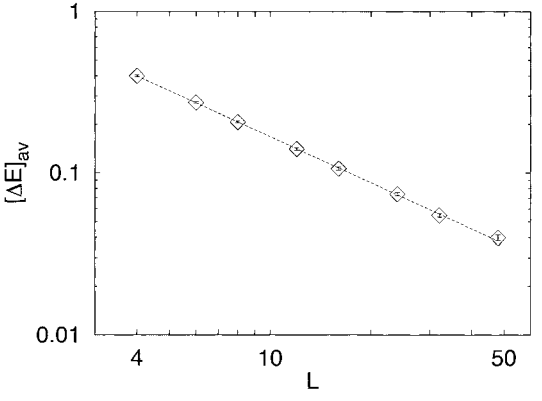


Figure 7.15: The domain-wall energy $[\Delta E]_{\rm av}$ in a log-log plot. The straight line is a fit to $[\Delta E]_{\rm av} \sim L^{\theta}$ with $\theta = -0.95 \pm 0.03$. This implies a thermal exponent of $\nu = 1.05 \pm 0.03$. The disorder average is over 500 samples for $L = 48$, 1500 samples for $L = 32$, and for the smaller sizes several thousand samples have been used.

[5] C. Zeng, A.A. Middleton, and Y. Shapir, *Phys. Rev. Lett.* **77**, 3204 (1996)

[6] H. Kleinert, *Gauge Fields in Condensed Matter*, (World Scientific, Singapore 1989)

[7] M.S. Li, T. Nattermann, H. Rieger, and M. Schwartz, *Phys. Rev. B* **54**, 16024 (1996)

[8] G. Blatter, M.V. Feigel'man, V.B. Geshkenbein, A.I. Larkin, and V.M. Vinokur, *Rev. Mod. Phys.* **66**, 1125 (1994)

[9] M.P.A. Fisher, T.A. Tokuyasu, and A.P. Young, *Phys. Rev. Lett.* **66**, 2931 (1991)

[10] C. Wengel and A.P. Young, *Phys. Rev. B* **54**, R6869 (1996)

[11] R. Ahuja, T. Magnanti, and J. Orlin, *Network Flows*, (Prentice Hall, New Jersey 1993)

[12] L. Balents and M. Kardar *Phys. Rev.* **B 49**, 13030 (1994); T. Hwa and T. Nattermann, *Phys. Rev.* **B 51**, 455 (1995)

[13] T. Knetter, G. Schröder, M. J. Alava, and H. Rieger, preprint cond-mat/0009108

[14] P.M. Chaikin and T.C. Lubensky, *Principles of Condensed Matter Physics*, (Cambridge University Press, Cambridge 1997)

[15] H. Rieger and U. Blasum, *Phys. Rev. B* **55**, 7394 (1997)

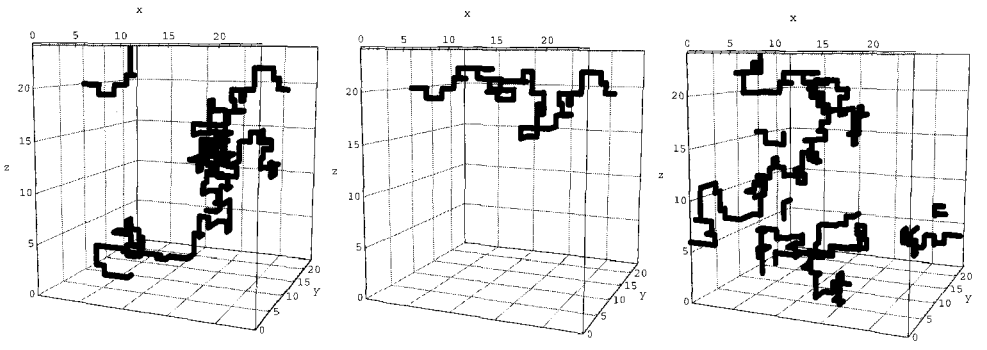


Figure 7.16: The minimum energy global excitation loop *perpendicular* to the external field in the z-direction is shown for one particular sample ($L = 24$) and three different field strengths B (note the periodic b.c. in all space directions). **a)** (left) $B \in [0.0065, 0.0069]$ is in a range, where the defect energy ΔE varies linearly with respect to the field. Note that the loop also has a winding number $n_z = 1$ in the direction *parallel* to the external field. Hence $\partial \Delta E / \partial B = 2L$. **b)** (middle) The same sample as in (a) with $B \in [0.0070, 0.0075]$. In this interval the defect energy is constant, no loop along the direction of the applied field occurs. **c)** (right) The same sample as in (a, b) with $B \in [0.0076, 0.0081]$. The system is very sensitive to the variation of applied field ΔB. Even for a small change by $\Delta B = 0.0001$ the form of the excitation loop changes drastically.

[16] F. Pfeiffer and H. Rieger, *J. Phys. A* **33**, 2489 (2000)

[17] H.S. Bokil and A.P. Young, *Phys. Rev. Lett.* **74**, 3021 (1995)

[18] J.M. Kosterlitz and N. Akino, *Phys. Rev. Lett.* **81**, 4672 (1998)

[19] J.M. Kosterlitz and M. V. Simkin, *Phys. Rev. Lett.* **79**, 1098 (1997)

[20] J. Kisker and H. Rieger, *Phys. Rev. B* **58**, R8873 (1998)

8 Genetic Algorithms

In this chapter, genetic algorithms (GA) are explained. For a detailed introduction, see e.g. [1, 2, 3]. The basic idea is to mimic the evolution of a group of creatures of the same species. Individuals which adapt better to the requirements imposed by their environment have a higher probability of survival. Thus, they pass their genes more frequently to subsequent generations than others. This means, the average fitness of the population increases with time. This basic idea can be transfered to optimization problems: instead of looking for skilled creatures, the aim is to find a minimum of a maximum of an objective function. Rather than different individuals, different vectors of arguments of the objective function are treated. The evolutionary process can easily be adapted for algorithms creating better and better vectors. The scheme has already been applied to various problems.
In this section we first give the basic framework of a GA. Then we present an example in detail, which enables us to understand the underlying mechanisms better. Finally two applications from physics of the smallest and the largest particles are presented: finding ground states in one-dimensional electronical quantum systems and determining the parameters of interacting galaxies. In Chap. 9 another application is shown, where genetic algorithms are applied among other techniques for the calculation of spin-glass ground states.

8.1 The Basic Scheme

The basic observation, which led to the development of genetic algorithms [4], is that for millions of years a huge optimization program has been running: nature tries to adjust the way creatures are formed such that they have a high probability of survival in a hostile environment (unfortunately the hostile environment evolves as well). At the beginning, all animals and plants were very simple. Only by *reproduction* and *mutation* did the structure of the creatures become more and more complex, so that they adapted better and better to the requirements. This works, because of the presence of *selection*: individuals which are better equipped have a higher probability of staying alive ("survival of the fittest"). Hence, they can pass on the information about how they are built, i.e. their *genes*, to subsequent generations more often. By iterating this scheme millions of millions of times on a large *population*

of individuals, on average better and better creatures appear. Please note that this scheme is over simplified, because it neglects the influence of learning, society (which is itself determined somehow by the genes as well) etc.

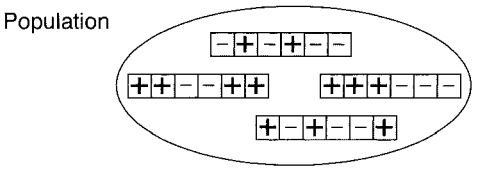


Figure 8.1: A population of individuals. Each individual is a collection of characteristic values, e.g. genes, parameters or positions. Here it is a vector of values $+/-$.

This simple principle can be transfered to other optimization problems. One is not necessarily interested in obtaining an optimum creature, but maybe one would like to have a configuration of minimum energy (ground state) of a physical system, a motor with low energy consumption or a scheme to organize a company in an efficient way. Physical systems, motors or companies are not represented by a sequence of genes but are given through a configuration of particles or a vector of parameters. These will be denoted as *individuals—ii* as well. The *population* (see Fig. 8.1) is again just a set of different individuals.

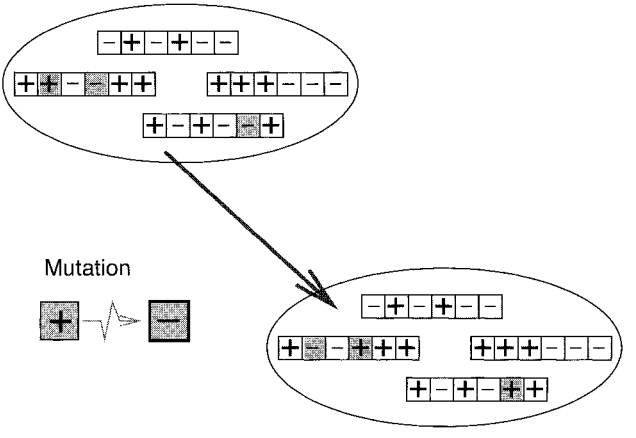


Figure 8.2: The effect of the mutation operation on a population. Individuals are randomly changed. Here, the values of the vectors are turned from $+$ to $-$ or vice versa with a given small probability. In the upper part the initial population is shown, in the lower part the result after the mutation has been carried through. The values which have been turned are highlighted.

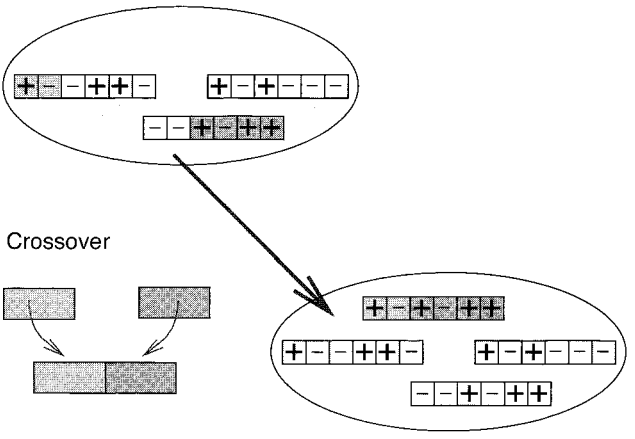


Figure 8.3: The crossover operation. Offspring are created by assembling parts from different individuals (parents). Here just one part from a $+/-$ vector and another part from a second vector are taken.

Now the different operations affecting the population will be considered. Mutations of genes correspond to random changes of the individual, e.g. displacing a particle or changing a parameter, see Fig. 8.2. The reproduction scheme, often called *crossover*, can be transfered in many different ways to genetic algorithms. The general principle is that one or several individuals (the *parents*) are taken, divided into small parts and reassembled in different ways to create new individuals, called *offspring* or *children*, see Fig. 8.3. For example a new particle configuration can be created by taking the positions of some particles from one configuration and the positions of the other particles from a second configuration.

The selection step can be performed in many ways as well. First of all one has to evaluate the *fitness* of the individuals, i.e. to calculate the energy of the configurations or to calculate the efficiency of the motor with given parameters. In general, better individuals are kept, while worse individuals are thrown away, see Fig. 8.4. The details of the implementation depend on the problem. Sometimes the whole population is evaluated, and only the better half is kept. Or one could keep each individual with a probability which depends on the fitness, i.e. bad ones have a nonzero probability of being removed. Other selection schemes just compare the offspring with their parents and replace them if the offspring are better.

Another detail, which has to be considered first when thinking about implementing a GA, is the way the individuals are represented in a computer. In general, arbitrary data structures are possible, but they should facilitate the genetic operations such as mutation and crossover. In many cases a *binary representation* is chosen, i.e. each individual is stored via a string of bits. This is the standard case where one speaks of a "genetic algorithm". In other cases, where the data structures are more complicated, sometimes the denotation "evolutionary program" is used. For simplicity, we just keep

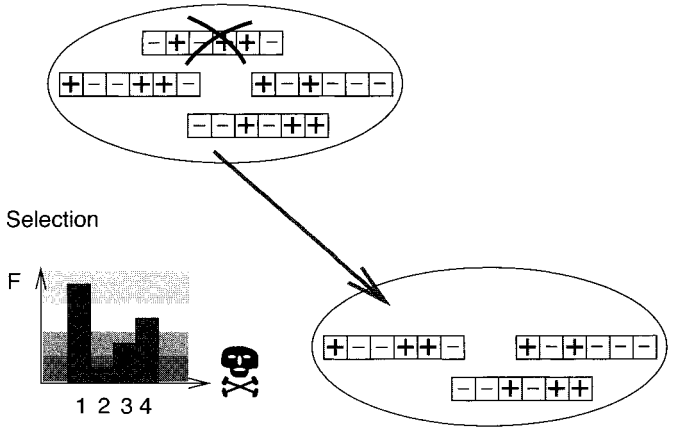


Figure 8.4: The effect of the selection operation on the population. The fitness F is evaluated, here for all four individuals. Individuals with a low fitness have a small probability of survival. In this case, individual two is removed from the population.

the expression "genetic algorithm", regardless of which data structure is chosen.

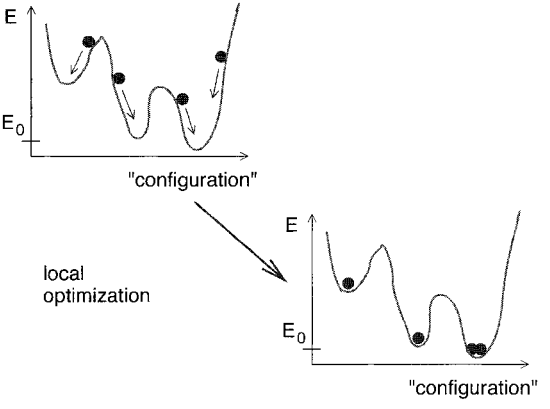


Figure 8.5: Local optimization. Here the population is shown in an energy landscape. Large energy means small fitness. This local optimization moves individuals to the next local optimum.

Quite frequently, the performance of a genetic algorithm can be increased by applying *local optimizations*. This means that individuals are taken and altered in a more or less deterministic way, such that its fitness is increased. Usually a local optimum is found which is close to the current individual, see Fig. 8.5. For example, when searching for

a ground state, one could move particles in a way that the energy is decreased, using the forces calculated from a given configuration. Whether and how local optimizations can be applied depends strongly on the current problem.
We finish this section by summarizing a possible general structure of a GA. Please note that many different ways of implementing a genetic algorithm exist, in particular the order of the operations crossover, mutation, local optimization and selection may vary. At the beginning the population is usually initialized randomly, M denotes its size and n_{R} the number of iterations.

algorithm genetic
begin
  Initialize population $x_1, \ldots, x_M$;
  for $t := 1$ **to** n_{R} **do**
  begin
    select parents $p_1, \ldots, p_k$;
    create offspring $c_1, \ldots, c_l$ via crossover;
    perform mutations;
    eventually perform local optimization;
    calculate fitness values;
    select individuals staying alive;
  end
end

Genetic algorithms are very general optimization schemes. It is possible to apply them to various problems appearing in science and everyday life. Furthermore, the programs are relatively easy to implement, no special mathematical knowledge is required. As a result an enormous number of applications have appeared during the last two decades. There are several specialized journals and conferences dedicated to this subject. When you search in the database INSPEC for articles in scientific journals which contain the term "genetic algorithm" in the abstract, you will obtain more than 15000 references. On the other hand, applications in physics are less common, about several hundred publications can be found in INSPEC. Maybe this work will encourage more physicists to employ these methods. Recent applications include lattice gauge theory [5], analysis of X-ray data [6], study of the structures of clusters [7, 8, 9], optimization of lasers [10]/ laser pulsed [11] and optical fibers [12], examining nuclear reactions [13], data assimilation in meteorology [14] and reconstructing geological structures from seismic measurements [15]. An application to a problem of statistical physics in conjunction with other methods is presented in Chap. 9. Below, sample applications from quantum physics and astronomy are covered.
However, one should mention that GAs have two major drawbacks. Firstly the method is not exact. Hence, if your are interested in finding the exact optimum along with a proof that the global optimum has really been obtained, then you should use other techniques. But if you want to get only "very good" solutions, genetic algorithms could be suitable. It is also possible to find the global optimum with GAs, but usually you have to spend a huge numerical effort when the problem is NP-hard. Secondly,

although the method has a general applicability, the actual implementation depends on the problem. Additionally, you have to tune parameters like the size of the population or the mutation rate, to obtain a good performance. Very often the effort is worthwhile. The implementation is usually not very difficult. Some people might find it helpful to use the package *Genetic and Evolutionary Algorithm Toolbox* (GEATbx) [16], which is written to use in conjunction with the program Matlab [17]. In the next section a simple example is presented, which can be implemented very easily without any additional libraries.

8.2 Finding the Minimum of a Function

As an example we will consider the following problem. Given a one-dimensional function $f(x)$, we want to find its minimum in an interval $[a, b]$. For this kind of problem many mathematical methods already exist, so in practice one would not apply a genetic algorithm. This example has the advantage that due to its simplicity one can concentrate on the realization of the genetic operators. Furthermore, it allows us to understand better how GAs work. For this purpose, the development of the population in time is studied in detail. Also the benefit gained from a local optimization will be shown later on. Here, we are considering the function

$$f(x) = 10|x - 0.5| - \cos(100(x - 0.5)) + 1 \tag{8.1}$$

in the interval $[0, 1]$. The function is plotted in Fig. 8.6. The goal is to find, via a genetic algorithm, the minimum of $f(x)$, which is obviously located at $x_0 = 0.5$ with $f(x_0) = 0$.
Here the individuals are simply different real values $x_i \in [0, 1]$. In order to apply genetic operations like mutation and crossover, we have to find a suitable representation. We chose the method used to store numbers in a computer, the binary representation: it is a string $x_i^1 \dots x_i^P$ of zeros and ones, where P denotes the length of the strings, i.e. the precision. Since all values belong to the interval $[0, 1]$ we use:

$$x_i \equiv \sum_{j=1}^{P} 2^{-j} x_i^j \tag{8.2}$$

For example the string "011011" represents $x = 2^{-2} + 2^{-3} + 2^{-5} + 2^{-6} = 0.421875$. From a given string, the corresponding number can be calculated just be performing the sum in (8.2). The inverse transformation is given by the following small procedure.

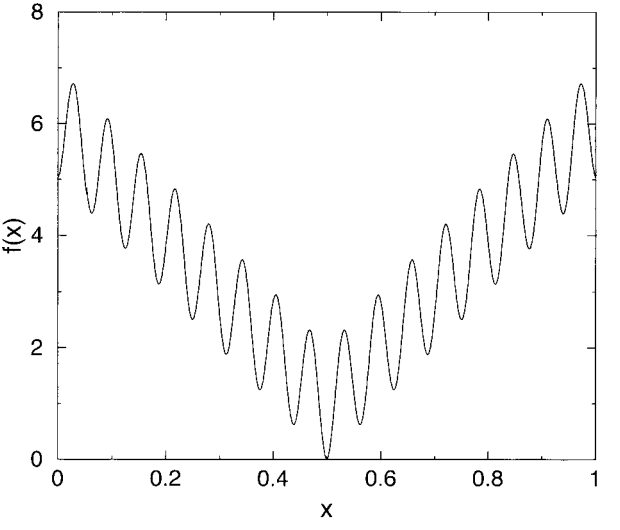


Figure 8.6: One-dimensional sample fitness function $f(x)$.

procedure bit-sequence(x, P)
begin
  $f := 0.5$
  for $q := 1$ **to** P
  begin
    if $x \geq f$ **then**
      $x^q := 1$; $x := x - f$;
    else
      $x^q := 0$;
    $f := f/2$;
  end
  return$(x^1, \ldots, x^P)$;
end

Next, we present the realization of the genetic operations. For the mutation with rate p_m, each bit is reversed with probability p_m (the random numbers drawn in this algorithm are assumed to be equally distributed in $[0, 1]$):

```
procedure mutation({x^q})
begin
   for q := 1 to P do
   begin
      r := random number in [0, 1];
      if r < p_m then
         x^q := 1 − x^q;
   end
   return (x^1, ..., x^P);
end
```

Example: Mutation

We will consider for the example the following bit string ($P = 20$)

$$x = 00110111011101000011$$

For a mutation rate of $p_\mathrm{m} = 0.2$, by chance 3 bits (on average two bits) could be reversed, e.g. bit 5, bit 10 and bit 17, resulting in

$$x = 00111\underline{1}1110\underline{0}1101001\underline{0}11$$

□

The crossover is slightly more complicated. It creates two children c_1, c_2 from two parents x_i, x_j in the following way. A random **crossover point** $s \in \{1, 2, \ldots, P\}$ is chosen. The first child is assigned the bits x_i^1 to x_i^s from the first parent and the bits x_j^{s+1} to x_j^P from the second parent, while the second child takes the remaining bits $x_j^1, \ldots, x_j^s, x_i^{s+1}, \ldots, x_i^P$:

```
procedure crossover(x_i, x_j)
begin
   s := random integer number in {1, 2, ..., P};
   for q := 1 to s do
      c_1^q := x_i^q; c_2^q := x_j^q;
   for q := s + 1 to P do
      c_1^q := x_j^q; c_2^q := x_i^q;
   return (c_1, c_2);
end
```

Example: Crossover

We assume that the crossover point is $s = 7$ (denoted by a vertical bar |). For the two parents ($P = 20$)

$$\begin{aligned} x_i &= 0011011|1011101000011 \\ x_j &= 1011101|0010010100100 \end{aligned}$$

the following children are obtained:

$$\begin{aligned} c_1 &= 0011011|0010010100100 \\ c_2 &= 1011101|1011101000011 \end{aligned}$$

□

In this problem, the selection is done in the following way: each child replaces a parent, if it has a better fitness, i.e. the evaluation of the function f results in a lower value. For simplicity we just compare the first child with parent x_i and the second with x_j. The complete genetic algorithm is organized as follows. Initially M strings of length P are created randomly, zeroes and ones appear with the same probability. The main loop is performed n_{R} times. In the main loop, two parents x_i, x_j are chosen randomly each individual has the same probability, and two children c_1, c_2 are created via the crossover. Then the mutation is applied to the children. Finally, the selection is performed. After the main loop is completed, the individual x having the best fitness $f(x)$ is chosen as a result of the algorithm. The following representation summarizes the algorithm:

algorithm minimize-function($M, P, n_{\mathrm{R}}, p_{\mathrm{m}}, f$)
begin
  initialize M bit strings of length P randomly;
  for $t := 1$ **to** n_{R} **do**
  begin
    choose parents x_i, x_j with $i \neq j$ randomly in $[1, M]$;
    $(c_1, c_2) :=$ crossover(x_i, x_j);
    mutation(c_1, p_{m});
    mutation(c_2, p_{m});
    if $f(c_1) < f(x_i)$ **then**
      $x_i := c_1$;
    if $f(c_2) < f(x_j)$ **then**
      $x_j := c_2$;
  end
  return best individual from $x_1, \ldots, x_M$;
end

Please note that before the evaluation of the fitness $f(x_i)$, the value of the bit string $x_i^1, \ldots, x_i^P$ has to be converted into the number x_j.
Now we study the algorithm with the parameters $M = 50$ and $p_m = 0.1$. We recommend the reader to write the program itself. It is very short and the implementation allows to learn much about genetic algorithms. The choice of $p_m = 0.1$ for the mutation rate is very typical for many optimization problems. Much smaller mutation rates do not change the individuals very much, so new areas in configuration space are explored only very slowly. On the other hand, if p_m is too large, too much genetic information is destroyed by the mutation.
The optimum size M of the population usually has to be determined by tests. It depends on whether one is interested in really obtaining the global optimum. As a rule of a thumb, the larger the size of the population is, the better the results are. On the other hand, one does not want to spend much computer time on this, so one can decrease the population size, if the optimum is rather easy to find.

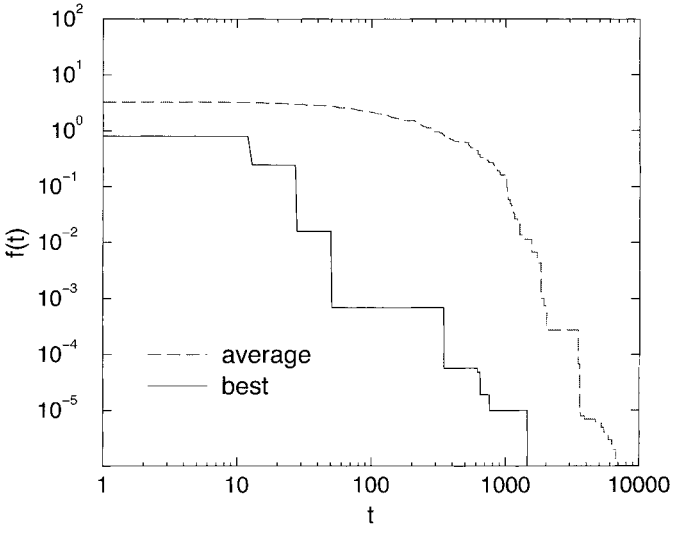


Figure 8.7: Evolution of the current minimum and average fitness with time t, here $M = 50, p_m = 0.1, n_R = 10000$.

In Fig. 8.7 the evolution of the fitness of the best individual and the average fitness are shown as a function of the step size. Here $n_R = 10000$ iterations have been performed. The global minimum has been found after 1450 steps. Since in each iteration two members of the population are considered, this means on average each member has been treated $2 \times 1450/M = 58$ times. Please note, if you are only interested in a very good value, not in the global optimum, you could stop the program after say only 100 iterations. At timestep 1450 only one individual has found the global optimum at $x_0 = 0.5$, the average value still decreases.

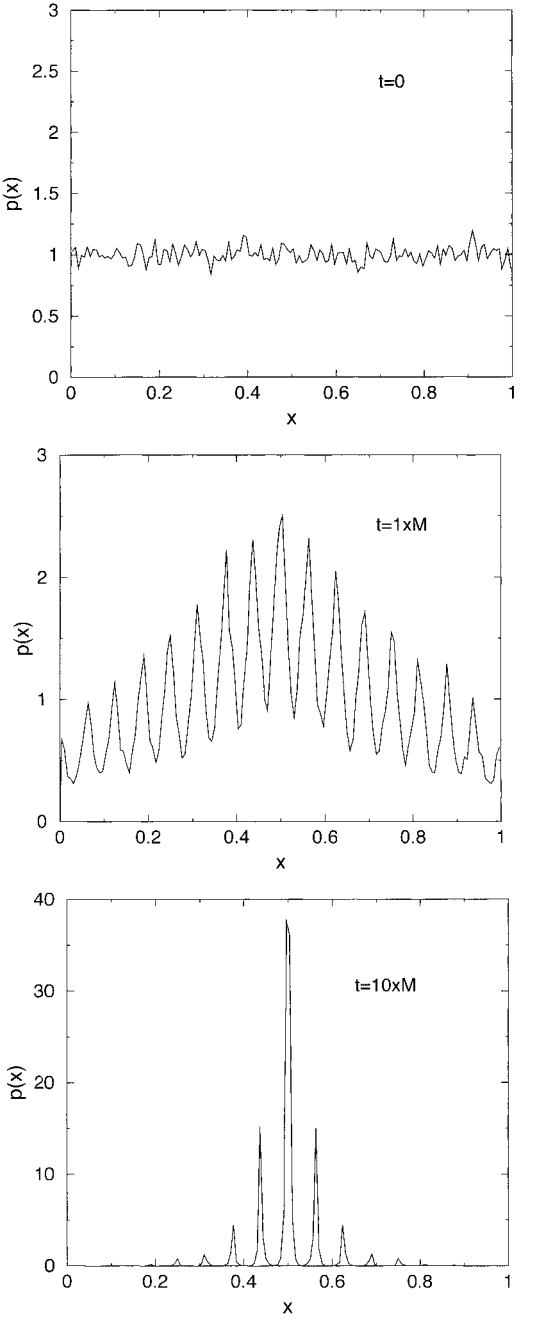


Figure 8.8: Evolution of population with time t, here $M = 50000, p_m = 0.1$ and $t = 0, 1 \times M, 10 \times M$.

To gain more insight into the driving mechanism of a GA, we will study the development of the population with time. Since we are interested in average values, we will study a much larger population $M = 50000$ to reduce the statistical fluctuations. At the beginning, the members of the population are independently distributed in the interval $[0, 1]$, see the top of Fig. 8.8.
The situation after $t = 1 \times M$ steps is shown in the middle part of Fig 8.8. Now it is more likely to find individuals near the local minima. The highest density of the population is near the global minimum. After $t = 10 \times M$ steps, most members of the population have gathered near the global optimum. Please note the change of the scale in the lower part of Fig. 8.8. Finally, after $t = 100 \times M$, the population is completely centered at the minimum (not shown).
As mentioned before, the efficiency of a genetic algorithm may be increased by applying local optimizations. Which method is suitable, depends highly on the nature of the problem. Here, for the minimization of a function of one variable, we use a very simple technique: an individual x is just moved to the next local minimum. This can be achieved by the following method. If the function f has a positive slope at x, then x is decreased by a value δ, otherwise x is increased. The sign of δ indicates the direction of the change. This process is iterated. Each time the slope of f changes compared with the last step, the size of $|\delta|$ is reduced and the sign reversed. The iteration stops if δ becomes of the order of 2^{-P}. The procedure reads as follows:

```
procedure local-optimization(x, P)
begin
   δ := 0.01;
   while |δ| > 2^-P do
   begin
      if δ × f'(x) ≥ 0 then
         δ := −0.5 × δ;
      x := x + δ;
   end
end
```

The algorithm iteratively approaches the closest local minimum. This simple local optimization is suitable, because f has a smooth behavior. For arbitrary functions more sophisticated methods have to be applied to search for local minima, see e.g. [18]. Please note that due to the limited numerical accuracy, it is possible that the exact numerical local optimum cannot be found.
The above algorithm is a special case of the *steepest descend method*, which is applied for functions of several variables. The basic idea is to calculate the gradient of the function at the current value of the arguments and alter the arguments in the direction of the gradient. Similar to the simple method presented above, this technique is not very fast. More efficient for minimizing functions of real variables are *conjugate gradient* methods [18].
The GA is altered in such a way that the local optimization is applied to each offspring after the mutation. This drastically decreases the number of steps which are necessary

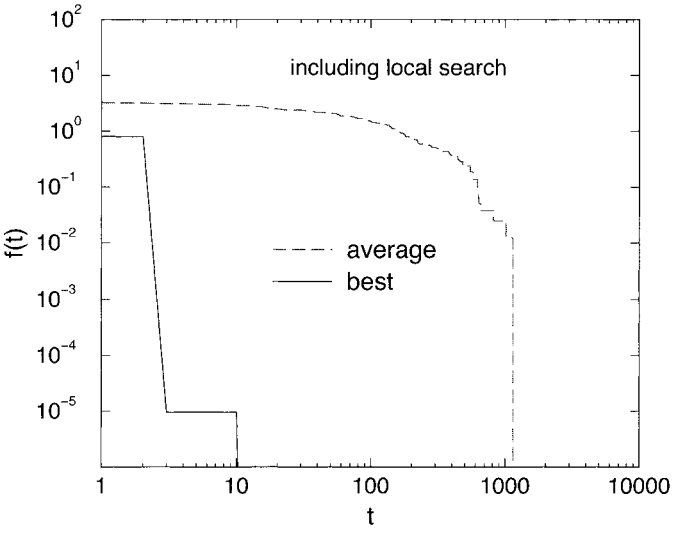


Figure 8.9: Evolution of the current minimum and average fitness with time t when additionally a local optimization is applied, here $M = 50, p_{\mathrm{m}} = 0.1, n_{\mathrm{R}} = 1000$.

to find the global optimum. The minimum and average fitness as a function of step number of the extended algorithm (run with the same parameters as before) are shown in Fig. 8.9. Please note that the decrease of the number of steps needed to obtain the minimum does not mean that the minimum is found quicker in terms of CPU time. In many cases the numerical demand of the local optimization is very large. Also one has to take into account the additional effort for the implementation. Whether it is worth including a local optimization or not depends heavily on the problem and must be decided each time again.

The impact on the population can be observed in Fig. 8.10. The distribution of the individuals after $t = M$ steps is shown. Most individuals are located near the local minima. Only the members of the population, which have not been considered so far, remain at their original positions.

The minimization problem presented in this section serves just as an example to illustrate the basic concepts of genetic algorithms. Two interesting applications from physics are presented in the following sections, while another special-purpose genetic algorithm is presented in Chap. 9.

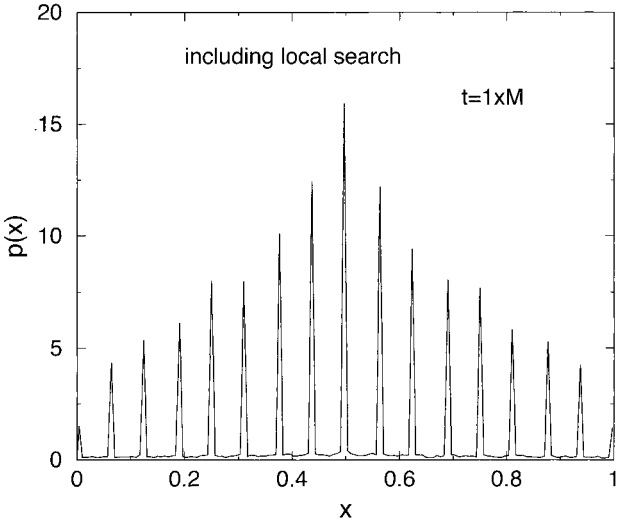


Figure 8.10: Distribution of the population at time t, here $M = 50000, p_m = 0.1$ and when applying the local optimization.

8.3 Ground States of One-dimensional Quantum Systems

During the last decade much progress has been made in the technology for manufacturing micro structures. This has happened particularly for semiconductor systems, where it is possible to create two dimensional confined potential landscapes for electrons of arbitrary geometry, so called *quantum dots*.
When studying the behavior of electrons in such systems one has to solve the quantum mechanical time-independent Schrödinger equation (with $\hbar^2/m = 1$),

$$\hat{H}\psi = E\psi, \quad \hat{H} = -\frac{1}{2}\nabla^2 + V, \tag{8.3}$$

where ∇ is the Nabla operator and V the potential, here just one particle in the quantum dot is considered. For some standard cases it is possible to solve the equation analytically, e.g. one particle in a trap or in a harmonic potential. These cases are usually presented in basic courses at university level. For arbitrary shapes of the potential V, exact analytical solutions cannot be obtained. In this chapter, we describe a recent approach developed by I. Grigorenkp and M.E. Garcia [19], where the ground-state wave function of low-dimensional systems with one particle can be found via a GA. Here we restrict ourself to one-dimensional systems, i.e. $\psi = \psi(x)$, $V = V(x)$, $\nabla = \frac{\mathrm{d}}{\mathrm{d}x}$, but the method can be easily extended to higher dimensions.
The ground state is the normalized wave function $\psi(x)$ minimizing the expectation

value

$$E[\psi] = \langle\psi|\hat{H}\psi\rangle = \int \psi^*(x)\hat{H}\psi(x)\,dx \tag{8.4}$$

of the energy, where ψ^* denotes the conjugate complex of ψ. To solve the ground-state problem via a genetic algorithm, one considers a population $\{\psi_j\}$ of M different wave functions. For the problem presented here, we assume that the electrons are confined by potential walls of infinite height to the interval $[a, b]$. So the population contains wave functions obeying the boundary conditions $\psi(a) = \psi(b) = 0$. Since we are interested in time-independent problems, we can neglect the phase of the wave function, i.e. concentrate on real-value functions.

Numerically, the functions are represented as a list of $K+1$ numbers $\psi_j(a+k(b-a)/K)$ with $k = 0, 1, \ldots, K$. The fitness of individual ψ_j is given by $E[\psi_j]$. For the evaluation of the fitness, a cubic spline [18] can be used to interpolate the wave function. Then the integral (8.4) can be calculated numerically.

To specify the details of the GA, we have to state the choice of the initial population, the crossover mechanism and the way mutations are performed. No local optimization is applied here.

The initial population $\{\psi_j(x)\}$ consists of Gaussian like functions of the form

$$\psi_j(x) = A_j \exp\left(\frac{x - x_j}{\sigma_j^2}\right)\Bigg(x - a)(b - x). \tag{8.5}$$

For each individual the values for the center $x_j \in [a, b]$ and the width $\sigma_j \in (0, b - a]$ are chosen randomly. The constant A_j is calculated from the normalization condition $\int |\psi_j(x)|^2\,dx = 1$. Please note that the boundary conditions are fulfilled by definition. For the mutation operation a random mutation function $\psi^{\mathrm{m}}(x)$ is added and the result normalized again:

$$\chi(x) = A\left[\psi_j(x) + \psi^{\mathrm{m}}(x)\right]. \tag{8.6}$$

The simplest approach is just to draw the mutation function ψ^{m} from the same ensemble as the initial population. The only difference is, that ψ^{m} is not normalized. This allows for slight changes of the wave function. In Fig. 8.11 a sample crossover operation is presented.

The crossover operation combines two randomly chosen parent functions $\psi_1(x)$, $\psi_2(x)$ and creates two offspring $\phi_1(x)$, $\phi_2(x)$. From the four wave functions $\psi_1(x)$, $\psi_2(x)$, $\phi_1(x)$, $\phi_2(x)$ the best two are taken over to the next generation. Similar to the crossover presented in the last section, the resulting wave functions consist of two parts, where the left part comes from one parent and the right part from another. Since wave functions have to be smooth functions, the offspring interpolate smoothly between both parents. For that purpose a smooth step function $St(x)$ is used:

$$\begin{aligned} \phi_1(x) &= B_1\left\{\psi_1(x)St(x) + \psi_2(x)[1 - St(x)]\right\} \\ \phi_2(x) &= B_2\left\{\psi_2(x)St(x) + \psi_1(x)[1 - St(x)]\right\} \end{aligned} \tag{8.7}$$

The constants B_1, B_2 are chosen in such a way that ϕ_1, ϕ_2 are again normalized. Here $St(x) = \frac{1}{2}[1 + \tanh((x - x_0)/k^2)]$ is taken, where $x_0 \in (a, b)$ is chosen randomly and

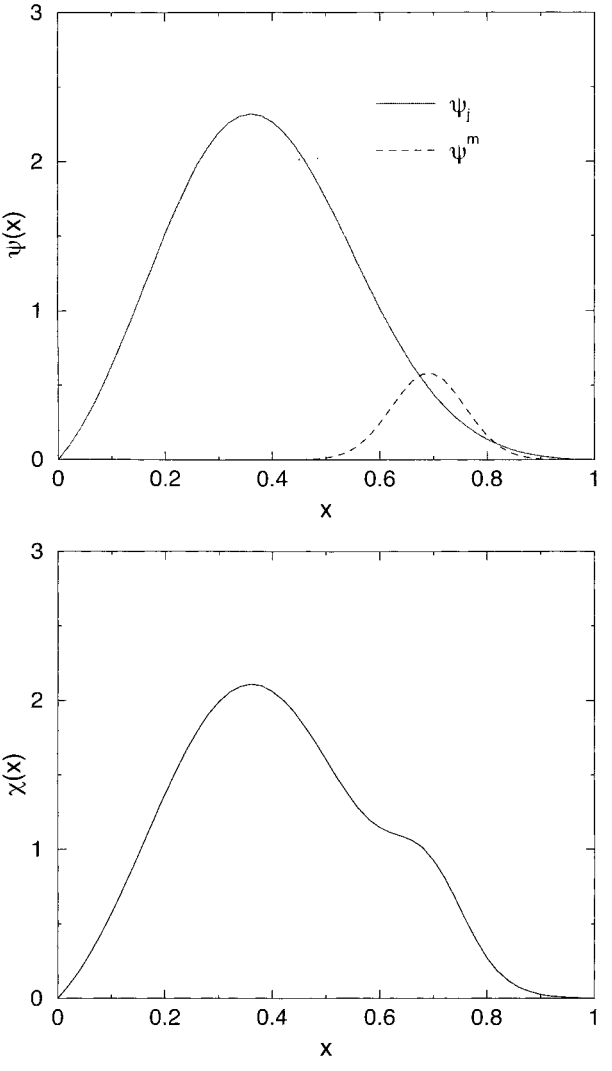


Figure 8.11: The mutation operation for the quantum ground-state calculation. Top: The original function ψ_j and the random mutation ψ^{m} function. Bottom: The resulting wave function.

k is a parameter which determines the sharpness of the step function. In Fig. 8.12 an example of the crossover operation is shown. The two parent functions belong to the initial population. The step function has $x_0 = 0.5, k^2 = 0.1$.

For this specific genetic algorithm, at each iteration step M times either a mutation

or a crossover operation is performed on randomly chosen wave functions. For the applications presented here, a probability of $p_m = 0.03$ for a mutation and a probability of $1 - p_m = 0.97$ for a crossover was used. $K = 300$ grid points were taken for storing the wave functions.

As a first test we consider the harmonic potential $V_h(x) = \frac{1}{2}\omega^2(x - 0.5)^2$ with $\omega = \sqrt{20} \times 10^2$, $a = 0$, $b = 1$. In Fig. 8.13 the probability density $|\psi(x)|^2$ for the ground state is shown which was found after 100 iterations. The inset shows the development of the lowest energy for the first 30 iterations. The probability density of the ground state is so close to the analytical solution that it is not possible to distinguish them in the plot. The ground-state energy is $E_0 = 316.29$ (atomic units) while the exact analytical result is $E_0 = 316.22$, i.e. a difference of just 0.02%.

Next, an example is presented where it is not possible to obtain an exact analytical solution. The potential is produced by five positive charges with charge $Q = 5$ placed at positions $x_i' = 13, 19, 25, 31, 37$. Here, the interval $[a, b] = 50$ is considered. The potential is softened by a cutoff distance $c = 0.5$ and given by

$$V_Q(x) = \sum_{i=1}^{5} \frac{Q}{\sqrt{(x - x_i)^2 + c^2}}. \tag{8.8}$$

Such potentials occur for example when clusters are studied, which are created by intense laser pulses. To speed up the convergence, the initial wave functions ψ_j should represent the symmetry of the potential. Thus, superpositions of five random functions of the form (8.5) are taken. The resulting ground-state probability distribution after 20 iterations is shown in Fig. 8.14. Please note that the symmetry of the potential is reflected by the wave function.

With an extension of the genetic algorithm it is possible to calculate excited states as well. More details can be found in [19]. Recently the method has been applied to two-dimensional quantum systems as well [20]. Now we leave the world of quantum physics and turn to the physics of large objects which provides another example where GAs have been applied successfully.

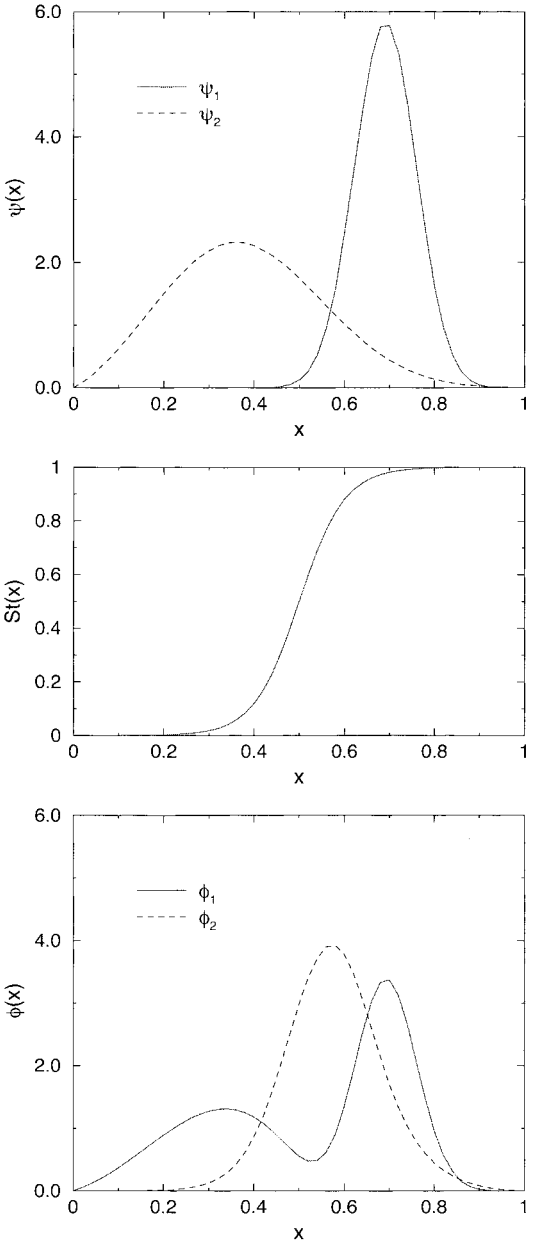


Figure 8.12: The crossover operation for the quantum ground-state calculation. Top: two parent functions. Middle: the step function ($k^2 = 0.1, x_0 = 0.5$) regulating the crossover. Bottom: resulting children functions.

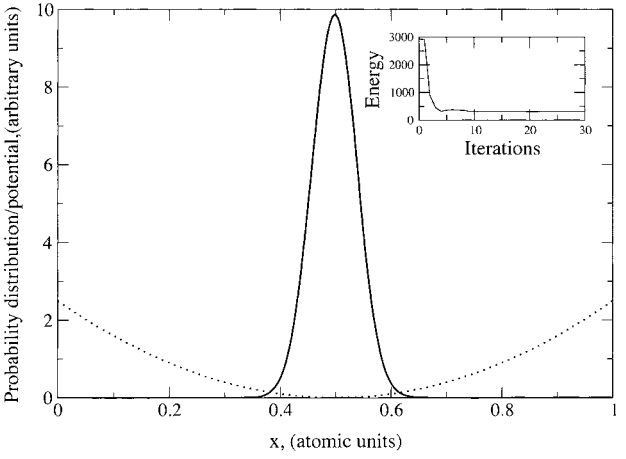


Figure 8.13: Calculated spatial distribution of electron density $|\psi(x)|^2$ (solid line) for the 1d harmonic potential (dotted line). The inset figure shows the evolution of the fitness during the GA-iterations. Reprinted from [19] with permission from Elsevier Science.

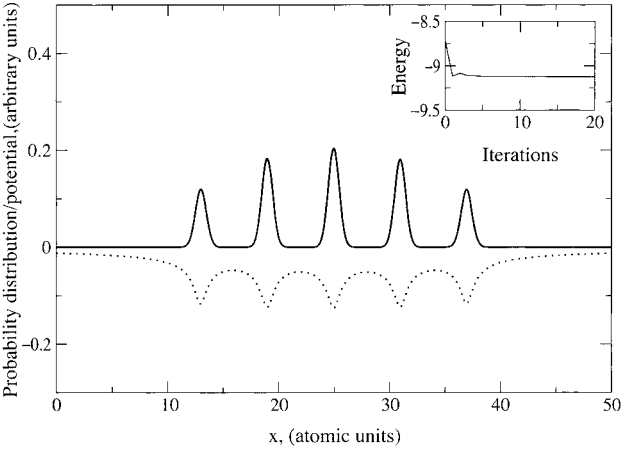


Figure 8.14: Calculated $|\psi(x)|^2$ (solid line) for an electron on a potential produced by a chain of positive ions (dotted line). The inset figure shows the convergence behavior. Reprinted from [19] with permission from Elsevier Science.

8.4 Orbital Parameters of Interacting Galaxies

When studying large scale objects in universe like galaxies, one is unfortunately restricted to watching the physical processes from far away. As a consequence, it is only possible to obtain two-dimensional images via telescopes instead of measuring all three coordinates in space. A rough impression of the three-dimensional structure can be obtained from the Doppler shift of the light. Furthermore, seen from the earth, the movement of galaxies is slow in comparison with the human lifetime. Hence, it is only possible to obtain a snapshot of the development, in contrast to e.g. the observation of asteroids or comets. In this section, the case is considered were two galaxies are colliding and it is shown how some hidden parameters can be obtained using a genetic algorithm [21], revealing the full three-dimensional description of the process.

Figure 8.15: Two colliding galaxies. The figure is generated from a simulation.

In Fig. 8.15 a snapshot of two colliding galaxies is shown. A coordinate system is used such that the x-y plane is the plane of the sky (with the x-axis horizontal) and

the z-axis points towards the observer. The movement of the centers of the masses of the galaxies can be described by the distances $\Delta x, \Delta y, \Delta z$, the relative velocities $\Delta v_x, \Delta v_y, \Delta v_z$, the masses m_1, m_2 and the spins s_1, s_2 (clockwise or counterclockwise). Using astronomical observations it is possible to measure the separation Δx, Δy in the plane of the sky and the relative velocity Δv_z along the line of sight [22]. The distance Δz, the velocities $\Delta v_x, \Delta v_y$, the masses m_1, m_2 and the spins s_1, s_2 are not known. In the following it is shown how these parameters can be estimated by applying a genetic algorithm. For that purpose, the individuals of the underlying population are chosen to be vectors of values $(\Delta z, \Delta v_x, \Delta v_y, m_1, m_2, s_1, s_2)$.
First of all, since the distance Δz cannot be measured, how do we know that the galaxies are interacting at all? Since one can observe only the projection of the scenery to the x-y plane, the galaxies might be far away. The basic idea to answer this question is that the evolution of a single galaxy is well studied. Its distribution of emitted light obeys an exponential falloff. Usually it is assumed that the amount of emitted light is proportional to the mass density, thus one can use an exponential density distribution. Only the length and mass scales may vary from galaxy to galaxy. For the case when two galaxies are far away from each other, both distribution of masses obey this standard picture. On the other hand, if two galaxies interact with each other, the standard distribution is perturbed. For example in Fig. 8.15 a bridge of stars connecting both galaxies can be observed. The main idea of the genetic algorithm to determine the unknown parameters is to take advantage of this type of perturbation for the analysis. We start the description of the GA with the encoding scheme for the individuals. Secondly, it is shown how the fitness of the individuals is calculated, i.e. how the collusion which results from the parameters is compared with the measurement data. Next the mutation and crossover operations are given. Finally a result from a sample simulation is presented.

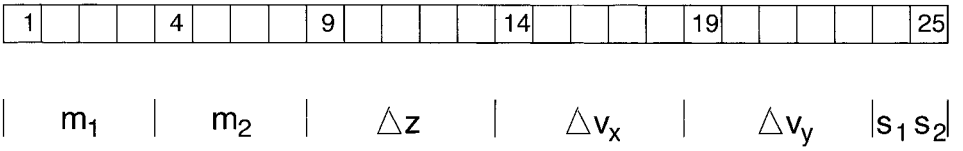


Figure 8.16: The encoding of the parameters $(m_1, m_2, \Delta z, \Delta v_x, \Delta v_y, s_1, s_2)$ via 25 digits $g_i \in \{0, 1, \ldots, 9\}$. Four digits are used for each mass m_1, m_2, while the values $\Delta z, \Delta v_x, \Delta v_y$ take 5 digits (one for the sign), the spins $s_1, s_2 = \pm$ are stored in the last two digits. An odd number represents the sign $+$, otherwise it is a negative sign.

Similar to the quantum-system example, here no binary representation is chosen. Instead, each individual is a vector $\mathbf{g} = (g_1, g_2, \ldots, g_{25})$ of 25 numbers from $\{0, 1, \ldots, 9\}$, see Fig. 8.16. Each vector represents the unknown values (m_1, m_2, Δz, Δv_x, Δv_y, s_1, s_2). To avoid large numbers, suitable units are chosen: the gravitational constant is $G = 1$, the unit of length is 1 kpc ($= 3.0856 \times 10^{19}$ m $= 3.2615$ light years), the unit

of time is 1.05 Myr $= 3.3143 \times 10^{13}$ s, the unit of the mass is 2×10^{11} times the mass of the sun (3.9782×10^{41} kg). The unit of velocity is then 931 km/s. Owing to the rescaling, all masses are taken in the interval 0 and 100, Δz between -1000 and $+1000$ and $\Delta v_x, \Delta v_y$ between -10 and 10. For encoding the masses 4 digits are used, while the distance Δz and the velocities take 5 digits. Thus, the first mass is obtained from the vector $\mathbf{g}$ through $m_1 = g_1 \times 10^1 + g_2 \times 10^0 + g_3 \times 10^{-1} + g_4 \times 10^{-2}$. The distances and velocities are encoded in a similar way using $4 + 1$ digits. One digit is reserved to store the sign, it is negative if the corresponding value is odd and positive otherwise. The spins $s_1, s_2 = \pm 1$ of the galaxies are stored in the last two digits g_{24}, g_{25} in the same way.

The part of the GA which consumes the most computer time is the evaluation of the fitness function, because for each set of parameters $\mathbf{g}$ a complete many-particle simulation has to be carried through. This works as follows. First the centers of mass of the two galaxies are calculated. With the set of known (i.e. fixed) parameters and with given values for the parameters in $\mathbf{g}$, the development of the system consisting of both of the centers of masses is completely determined. Hence, it is possible to integrate the equations of motion backwards in time until a state is obtained where the two galaxies are far away from each other. Next, the two centers of masses are replaced by two sets of stars with a rescaled standard mass distribution. The number of particles in the galaxies are denoted by $N_{p,1}$ and $N_{p,2}$, respectively. For the examples presented here, $N_{p,1} = N_{p,2} = 1000$ were chosen. For simplicity, it is assumed that the distribution of the stars is rotationally symmetric. Then the system is propagated forwards in time. To speed up calculations, the simulations can be performed non-self-gravitatingly, i.e. inter-particle forces are neglected. In this case only the gravitational forces between the particles and the center of masses are included. For an improved simulation the self gravitation and even gas dynamics and dark matter can be added, but this is beyond the scope of a first test of the method. Another enhancement is to drop the assumption of rotational symmetry and to allow galaxies to be oriented differently in space relative to each other, see [21]. Nevertheless, when the two galaxies approach each other, they begin to interact, i.e. the motion of the stars of the two systems are affected by the other galaxy and the morphology of the galaxies change. The system is propagated until the initial time 0 has been reached.

Now the distribution of masses obtained by the simulation can be compared with the given distribution from the two galaxies under investigation. This comparison works as follows. A grid of size $n_x \times n_x$ is superimposed on the data, see Fig. 8.17. The grid corresponds to a grid of pixels which is the result of an observation with a telescope, digitized and stored on a computer[1]. For each grid cell (i, j) the total mass $m_{i,j}$ in the cell is evaluated, each particle from the first galaxy contributes with $m_1/N_{p,1}$ and each particle from the second one with $m_2/N_{p,2}$. When comparing with observations, the amount of light corresponding to each mass $m_{i,j}$ has to be calculated. Here the method is evaluated for testing purposes. Thus, only artificial observations are taken, i.e. they are generated via a simulation of particles in the same way as explained. This

[1] Please note that usually galaxies consist of several billion of stars, so it is not possible to identify individual stars via a telescope.

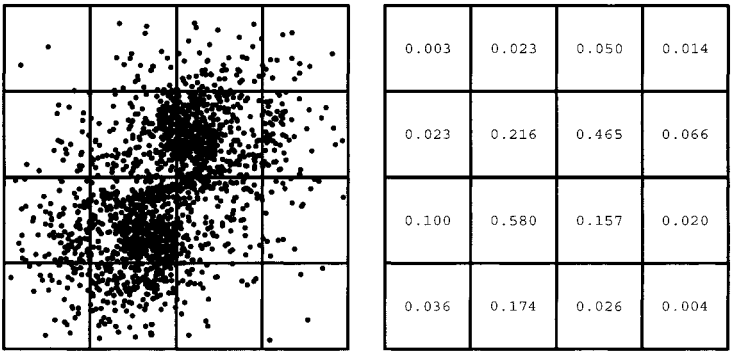


Figure 8.17: Computation of the data for the comparison. A grid is superimposed on the image of interacting galaxies. For each grid cell the corresponding mass is calculated. For clarity, only a coarse grid is shown here. Reprinted from [21] with permission from Springer Science.

allows the final result of the best set $\mathbf{g}$ obtained from the GA to be compared with the values which have been used to set up the system. For the "observed" system, the masses $m_{i,j}^{\text{obs}}$ can be calculated in the same way as the values of $m_{i,j}$. The deviation δ the result of the simulation and the observational data is defined here as

$$\delta \equiv \sum_{i,j} \frac{|m_{i,j} - m_{i,j}^{\text{obs}}|}{m_\epsilon + m_{i,j}^{\text{obs}}}. \tag{8.9}$$

The sum runs over all grid cells. The contribution m_ϵ in the denominator prevents a divergence in case the mass observed in a grid cell is zero. In Ref. [21] $m_\epsilon = 1/(N_{p,1} + N_{p,2})$ has been chosen, so particles in a region where no particle is supposed to be have the largest impact on the value of δ. Finally, the fitness F of the individual g is taken as

$$F \equiv \frac{1}{1+\delta} \tag{8.10}$$

Now the structure of the genetic algorithm will be described. Similar to the first example, the initial population of M individuals is chosen completely at random. Each generation is treated in the following way. First, the fitness values for all individuals are calculated, as explained above. Then the individuals are ranked according to their fitness values, the highest fitness comes last, the lowest fitness first. The best individual

is always taken over twice to the next generation. The other $M-2$ members of the next generation are obtained by crossover and mutation.
For each crossover two parents are selected randomly from the population. Here the *linear fitness ranking* is applied. This means each parent is chosen with a probability which is proportional to its position in the ranking. Thus, individuals with a low fitness have a lower probability of being selected. This can be achieved by drawing a natural random number r between 0 and $M(M+1)/2$. The individual which has position i in the ranking is chosen if $(i-1)i/2 \leq r < i(i+1)/2$. The crossover is carried out as explained in Sec. 8.2: a random crossover point $s \in 1, 2, \ldots, 25$ is chosen. The first child consists of the left part up to digit g_s from the first parent and the right part from the second parent. The second child takes the remaining digits.
Finally, mutations are applied for all new individuals. With probability p_{m} a digit is set to a new randomly chosen value from $\{0, 1, \ldots, 9\}$. The result is a new generation of M individuals. The whole process is repeated for n_G generations and after the last iteration the best individual is picked.

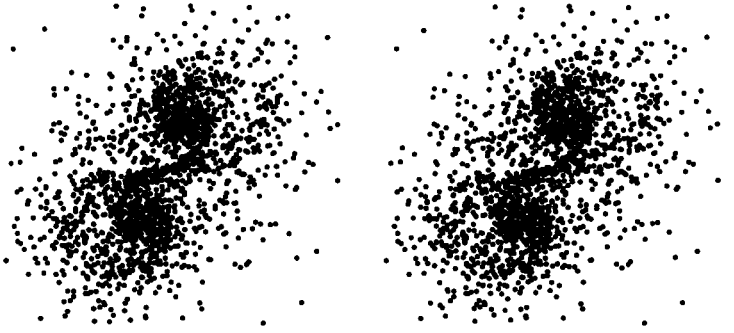


Figure 8.18: Comparison of the simulation with the best parameters with the original system. The two pictures are not identical, but very similar.

To test the genetic algorithm, an artificial pair of colliding galaxies was created with a given set of parameters ($m_1 = 1.0$, $m_2 = 1.0$, $\Delta z = 3.0$, $\Delta v_x = -0.672$, $\Delta v_y = 0.839$ and $s_1 = s_2 = 1$), the measurable parameters are $\Delta x = 3.5$, $\Delta y = 8.0$, and $\Delta v_z = 0.44$, see Fig. 8.15. Then the GA was run with the aim of finding the parameters again. A size $M = 500$ of the population was chosen, the program ran for $n_g = 100$ generations with $p_{\mathrm{m}} = 0.003$. To decrease the numerical effort, the values of m_1 and m_2 were constrained to be between 0.3 and 3.0, Δz in $[-50, 50]$ and $\Delta v_x, \Delta v_y$ between -1 and 1. When all individuals obey these restrictions, the children created by the crossover

remain in the same sample space as well. Mutations were accepted only if the resulting parameters remained in the given intervals. These constraints reduce the number of possible combinations of the parameters to 1.2×10^{15}, still much too large to be searched systematically. The GA was able to find these parameters in 100 generations within an accuracy of about 10%. The best individual exhibited $m_1 = 0.99$, $m_2 = 1.00$, $\Delta z = 2.899$, $\Delta v_x = -0.676$, $\Delta v_y = 0.897$, $s_1 = 1$, $s_2 = 1$. The resulting distribution of the stars is compared in Fig. 8.18 with the given observation. Only slight deviations are visible.

The results of further tests are presented in Ref. [21]. It can be shown that the GA is much more effective than a random search. Furthermore the algorithm is not sensitive to noise. Even when the observational data $m_{i,j}^{\text{obs}}$ were disturbed with a 30% level of noise, the parameters could be recovered, but with slightly lower accuracy. Finally it should be mentioned, that for real observational data, usually features such as bars, rings etc. in the central regions of the galaxies occur. This may cause problems to the GA. Hence, one should leave out the inner region for the calculation of the fitness F.

Bibliography

[1] M. Mitchell, *An Introduction to Genetic Algorithms*, (MIT Press, Cambridge (USA) 1996)

[2] Z. Michalewicz, *Genetic Algorithms + Data Structures = Evolution Programs*, (Springer, Heidelberg 1994)

[3] P. Sutton and S. Boyden, *Am. J. Phys.* **62**, 549 (1994)

[4] J.H. Holland, *Adaption in Natural and Artificial Systems*, (University of Michigan Press, Ann Arbor 1975)

[5] A. Yamaguchi and A. Sugamoto, *Nucl. Phys. B Proc. Suppl.* **83-84**, 837 (2000)

[6] A. Ulyanenkov, K. Omote, and J. Harada, *Physica B.* **283**, 237 (2000)

[7] T.X. Li, S.Y. Yin, Y.L. Ji, B.L. Wang, C.H. Wang, and J.J. Zhao, *Phys. Lett. A* **267**, 403 (2000)

[8] M. Iwamatsu, *J. Chem. Phys.* **112**, 10976 (2000)

[9] D. Romero, C. Barron, and S. Gomez, *Comp. Phys. Comm.* **123**, 87 (1999)

[10] Cheng Cheng, *J. Phys. D* **33**, 1169 (2000)

[11] R.S. Judson and H. Rabitz, *Phys. Rev. Lett.* **68**, 1500 (1992)

[12] F.G. Omenetto, B.P. Luce, and A.J. Taylor, *J. Opt. Soc. Am. B* **16**, 2005 (1999)

[13] D.G. Ireland, *J. Phys. G* **26**, 157 (2000)

[14] B. Ahrens, *Meteor. Atmos. Phys.* **70**, 227 (1999)

[15] H. Sadeghi, S. Suzuki, and H. Takenaka, *Phys. Earth Plan. Inter.* **113**, 355 (1999)

[16] Genetic and Evolutionary Algorithm Toolbox, can be tested three weeks for free, see http://www.geatbx.com/index.html

[17] Matlab 5.3.1 is a commercial program for data analysis, numerical computations, plotting and simulation, see http://www.mathworks.com/

[18] W.H. Press, S.A. Teukolsky, W.T. Vetterling, and B.P. Flannery, *Numerical Recipes in C*, (Cambridge University Press, Cambridge 1995)

[19] I. Grigorenko and M.E. Garcia, *Physica A* **284**, 131 (2000)

[20] I. Grigorenko and M.E. Garcia, *Physica A* **291**, 439 (2001)

[21] M. Wahde, *Astron. Astroph. Supplement Series* **132**, 417 (1998)

[22] K. Holliday, *Introductory Astronomy* (Wiley, New York 1999)

9 Approximation Methods for Spin Glasses

In this chapter we concentrate on a certain class of magnetic systems called *spin glasses* and on methods to find ground states of this systems. Similar to the RFIM, which was covered in Chap. 6, spin glasses consist of spins interacting with each other. The behavior of these materials is more complicated. Owing to the existence of competing interactions they exhibit ordered phases, but without spatial order of the orientations of the spins. From the computational point of view, spin glasses are very interesting as well, because the ground-state calculation is NP-hard. For these and other reasons, spin glasses have been among the central topics of research in material science and statistical physics during the last two decades. This is can be seen from the fact, that almost all optimization methods presented in this book, and many others, have been tested on spin glasses in the past.

This chapter is organized as follows. We begin by presenting suitable a Ising model for spin glasses. An experimental realization is shown and some important properties are mentioned. It is explained why there is still an ongoing debate about its low temperature behavior. In the second section an efficient approximation method for calculating ground states is presented. Next we show that the algorithm, although able to calculate true ground states, does not give the correct thermodynamic statistics of the states. This bias can be corrected by a post-processing method which is explained in the fourth section. Finally, some results obtained with these algorithms are presented, partly solving one question, that had been open for a long time.

9.1 Spin Glasses

An introductions to spin glasses can be found in [1, 2, 3, 4]. Recent developments are covered in [5].

A suitable theoretical model describing spin glasses is similar to the RFIM and DAFF models (see Chap. 6), but it comprises bond randomness as a key element. Again, it is sufficient to concentrate on Ising spins $\sigma_i = \pm 1$ to find the main spin-glass properties: N spins are placed on the regular sites of a lattice with linear extension L, e.g. quadratic ($N = L^2$) or cubic ($N = L^3$). The spins interact ferromagnetically or antiferromagnetically with their neighbors. A small example is shown in Fig. 9.1.

The Hamiltonian is given by

$$H \equiv -\sum_{\langle i,j \rangle} J_{ij}\sigma_i\sigma_j. \tag{9.1}$$

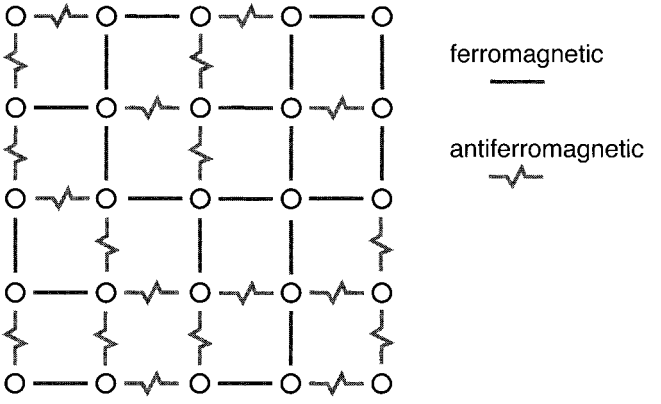


Figure 9.1: A two-dimensional spin glass with bond disorder. Spins are placed on the sites of a regular grid. They interact with their neighbors, the interaction is random, either ferromagnetic or antiferromagnetic.

The sum $\langle i,j \rangle$ runs over all pairs of nearest neighbors and J_{ij} denotes the strength of the bond connecting spins i and j. It is also possible to add a term describing the interaction with an external field B, here we will concentrate on the case $B = 0$. This kind of model was introduced by Edwards and Anderson [6] in 1975, usually it is called the EA model. It has a broad range of applications. Models involving similar energy formulae have been developed e.g. for representing neural networks, social systems or stock markets.

For each realization of the disorder, the values J_{ij} of the bonds are drawn according to a given probability distribution. Very common are the Gaussian distribution and the bimodal $\pm J$ distribution, which have the following probability densities:

$$p_G(J) = \frac{1}{\sqrt{2\pi}} \exp\left(-\frac{J^2}{2}\right) \tag{9.2}$$

$$p_{\pm J}(J) = 0.5\delta(J-1) + 0.5\delta(J+1) \tag{9.3}$$

Once the values of the bonds are fixed for a realization, they keep their values throughout the whole calculation or simulation, one speaks of *quenched disorder*. Since the system is random itself, to calculate physical quantities like magnetization or energy, one must perform not only a thermal average but also an average over different realizations of the disorder.

The main ingredients constituting a spin glass are: *mixed signs of interactions* and *disorder*. As a consequence, there are inevitably spins which cannot fulfill all constraints imposed by their neighbors and the connecting bonds, i.e. there will be some ferromagnetic bonds connecting antiparallel spins and vice versa. On says, it is not possible to *satisfy* all bonds. This situation is called *frustration*, the concept was introduced in [7]. In Fig. 9.2 an example of a small frustrated system is shown. In Sec. 9.2 it will be explained what impact the presence of frustration has on the choice of the algorithms. But first we will show how bond-randomness and frustration are created in real materials.

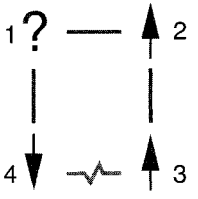


Figure 9.2: A frustrated plaquette at lowest energy. No matter which orientation spin 1 chooses, one of the bonds connecting it to its neighbors is not satisfied. Bonds 2-3 and 3-4 are satisfied.

9.1.1 Experimental Results

A spin glass can be created using well known materials: just take a lattice of a non-magnetic conducting material like gold and randomly replace a small fraction x of the gold by magnetic iron (Fe_xAu_{1-x}). To see the spin glass behavior in an experiment, the system is subjected to a weak magnetic field and the resulting magnetization is measured, i.e. one obtains the magnetic susceptibility χ. When studying χ as a function of temperature T one observes a peak at a very low temperature T_G. An example is shown in Fig. 9.3, the figure is taken from Ref. [8]. Usually T_G is of the order of 10 Kelvin, the exact value depends on the concentration x of the iron and on the way the sample is prepared. This peak is an indication of a phase transition. But when measuring the specific heat $C(T)$, a smooth behavior around T_G is found, only a broad maximum usually at higher temperatures can be observed. This is in strong contrast to usual phase transitions. Furthermore, when performing a neutron-scattering experiment, one finds that below the transition temperature T_G spin glasses exhibit no spatial (e.g. ferro- or antiferromagnetic) order of the orientation of the spins. Even more puzzling are *aging* experiments, where spin glasses are examined with respect to the time evolution and the history of the system. A sample experiment is shown in Fig. 9.4. A spin glass (here $CdCr_{1.7}In_{0.3}S_4$) is first cooled to $T = 12\,\mathrm{K}$ and kept there for a while [9]. The imaginary part χ'' of the dynamic susceptibility, describing the response of the magnetization to a weak applied alternating field (here with $\omega/2\pi = 0.01\mathrm{Hz}$), is measured for a while. After time t_1 the system is quenched

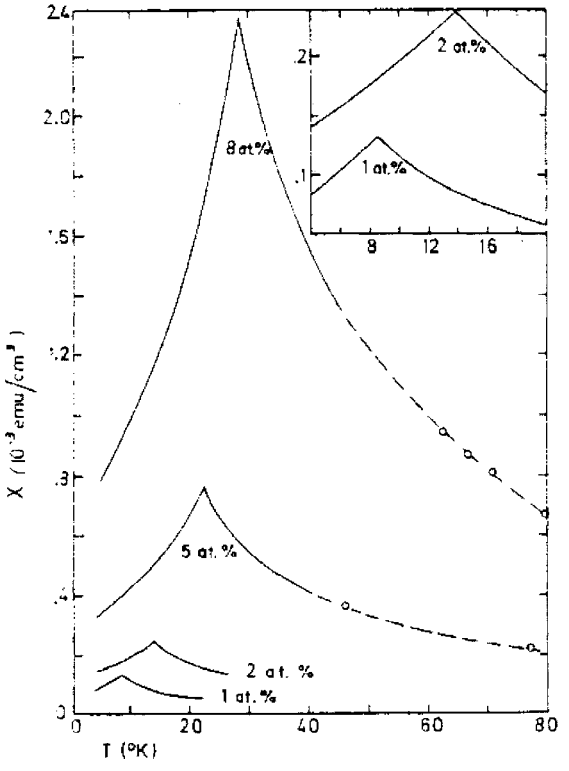


Figure 9.3: Magnetic susceptibility χ for Au_xFe_{1-x} at low temperatures. A cusp depending on the concentration x is observed. The inset magnifies the region $0 - 20$ K. The figure is taken from [8] with permission from J.A. Mydosh.

to 10 K, a jump of the susceptibility occurs. Again the system evolves for a while. After time t_2 the system is heated back to 12 K. Now the the susceptibility switches back to the value it had a time t_1. Hence, the system has remembered its state at temperature 12 K. Such types of experiments are not understood so far in detail, only heuristic explanations exist. This is one reason why spin glasses have attracted so much attention in the past and will probably also attract it in the future.
The basic reason for this strange behavior is the type of interaction which is present in this class of materials. The behavior of the Fe_xAu_{1-x} alloy is governed by the indirect-exchange interaction, usually called *RKKY* (Ruderman, Kittel, Kasuya, Yosida) interaction. Placing a magnetic spin $\mathbf{S}_i$ (iron) in a sea of conducting electrons, results in a damped oscillation in space of the susceptibility. Another spin $\mathbf{S}_j$ placed at distance r will create the same kind of oscillations resulting in an energy $H = J(r)\mathbf{S}_i \cdot \mathbf{S}_j$ where

$$J(r) \sim \frac{\sin(2k_f r)}{(2k_f r)^4} - \frac{\cos(2k_f r)}{(2k_f r)^3} \tag{9.4}$$

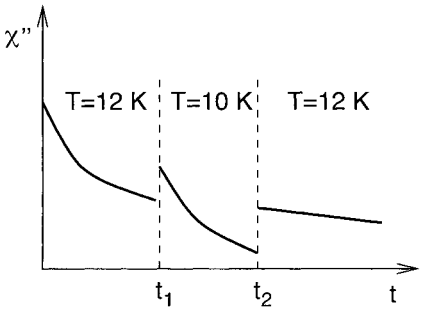


Figure 9.4: Aging experiment. A system is cooled down to $T = 12\,\mathrm{K}$, then the susceptibility is measured for a while. After time t_1 the system is suddenly quenched down to 10 K, again the susceptibility is measured. After a while the system is heated back to 12 K. At time $t_1 + t_2$ the system remembers the state it had at time t_1.

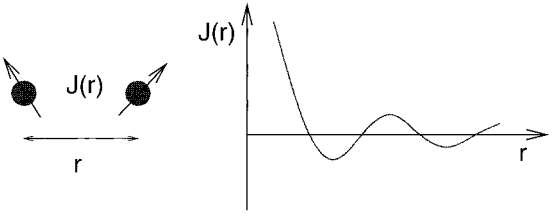


Figure 9.5: The RKKY (Ruderman, Kittel, Kasuya, Yosida) interaction sketched: strength of the interaction of two spins with distance r in a sea of conducting electrons.

(k_f: Fermi momentum of the conductor) which at larger distances r reduces to $J(r) \sim \frac{\cos(2k_f r)}{(2k_f r)^3}$, see Fig. 9.5. The main point is that the sign of the interaction changes with distance. In the iron-gold alloy, since the iron is place randomly in the gold host, each spin interacts with some spins ferromagnetically and with others antiferromagnetically. As a consequence, some pairs of spins prefer to be parallel aligned while other pairs favor an antiparallel orientation. At low temperatures this mixed interactions create a frozen non-regular pattern of the orientations of the spins, explaining why no spatial order of the spins can be detected using neutron scattering.

Apart from the RKKY interaction, there are other ways of creating interactions with different signs: some types of systems exhibit superexchange or other dipolar interactions. This leads to a huge number of materials showing spin-glass behavior at low temperatures, more details can be found in [4]. There are also different mechanisms creating disorder: in the case of Fe_xAu_{1-x} the randomness is obtained by placing the iron atoms at random chosen sites of a lattice. But also amorphous metallic alloys like $GdAl_2$ and YFe_2 also show spin glass behavior. For another class of systems,

the lattice sites are occupied in a regular way, but the sign and the strength of the bonds are random. Examples of such *random-bond* systems are $Rb_2Cu_{1-x}Co_xF_4$ and $Fe_{1-x}Mn_xTiO_3$.

9.1.2 Theoretical Approaches

Computer simulations of the EA model [1, 10] reproduce the main results found in experiments: a peak in susceptibility, a smooth behavior of the specific heat and frozen configurations of the spins. Recently, also the results of aging experiments have been found by simulations as well. Therefore, one can conclude that even the simple EA model incorporates the main properties constituting a spin glass.

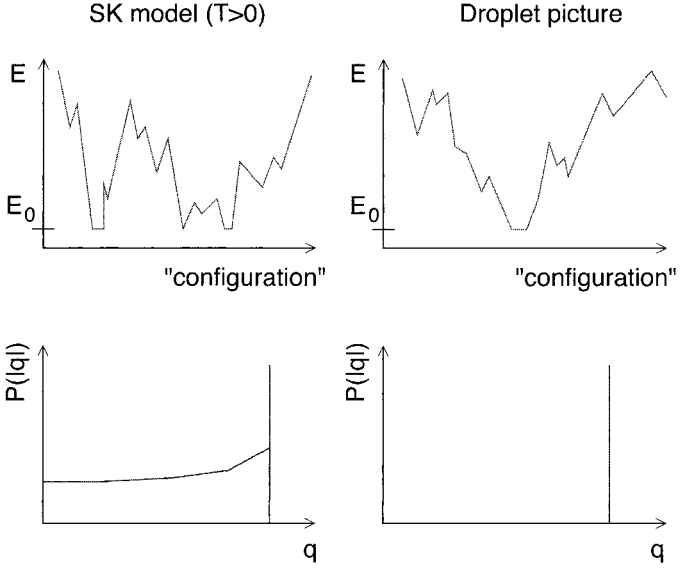


Figure 9.6: Mean-field solution vs. Droplet picture. The solution of the SK model exhibits a complicated energy landscape, resulting in a broad distribution $P(|q|)$ of the overlaps. In the Droplet picture instead the system is dominated by one pair of states at low temperatures, giving rise to a delta-distributed $P(|q|)$.

On the other hand, analytically it is very hard to treat the EA model [2, 3]. Since it is actually impossible to solve the simple cubic ferromagnet analytically, the reader may imagine that due to the additional average over the disorder and the varying sign of the bonds, only very raw approximations could be performed successfully for spin glasses. But there is a special spin-glass model, which was introduced by Sherrington and Kirkpatrick also in 1975 [11], the SK model. Its Hamiltonian is similar to the EA model, see Eq. (9.1), but it includes interactions between all pairs of spins. This means that the spins do not have any positions in space, the system does not have a

dimension. Usually one says the system is infinite-dimensional, because in the thermodynamic limit each spin has infinitely many "neighbors". The model is denoted as the *mean-field* (MF) model as well, since the MF approximation is exact here. For a Gaussian distribution of the interactions, the SK model has been solved analytically through the use of several enhanced techniques by Parisi in the 1980s [2]. The main property of the solution is that a complicated energy landscape is obtained (see upper left part of Fig. 9.6) and that the states are organized in a special hierarchical tree-line structure which is called *ultrametric*, for details see [12]. Especially at low temperatures for typical realizations, there are always many configurations which are arbitrarily different. What does this mean? To measure the difference between two configurations $\{\sigma_i^\alpha\}, \{\sigma_i^\beta\}$, the *overlap* q is introduced:

$$q^{\alpha\beta} \equiv \frac{1}{N} \sum_i \sigma_i^\alpha \sigma_i^\beta . \tag{9.5}$$

Thus if $\{\sigma_i^\alpha\}$ and $\{\sigma_i^\beta\}$ are the same, we obtain $q = 1$, while $q = -1$ if the configurations are inverted relative to each other. If only about half of the spins have the same orientation, we get $q \approx 0$. Since in a spin glass may exhibit many configurations with large thermodynamical weight even at low temperatures, one has to compare all of them pairwise. Each comparison results in an overlap value, so we end up with a *distribution of overlaps* $P_J(q)$. After averaging over the disorder an average distribution is obtained, denoted by $P(q)$. Since the Hamiltonian Eq. (9.1) does not contain an external magnetic field, it is symmetrical with respect to the inversion of all spins. Thus, $P(q)$ is symmetrical with respect to $q = 0$ and it is sufficient to study $P(|q|)$. The result for the SK model at low temperatures is shown in the lower left part of Fig. 9.6. It contains a large peak called self overlap, which results from the overlaps of states belonging to the same valley in the energy landscape. Additionally, there is a long tail down to $q = 0$ resulting from pairs of states from different valleys. Although the solution of the SK model is very elegant, it is restricted to this special spin-glass model. The question concerning the behavior of realistic, i.e. finite-dimensional spin glasses is currently unsolved. One part of the physics community believes that also for e.g. three-dimensional systems a similar hierarchical organization of the states can be found, as in the SK/mean-field model. Another group favors a description which predicts a much simpler behavior, the *Droplet* picture [13, 14, 15, 16, 17]. In that framework it is assumed that the low temperature behavior is governed by basically *one* class of similar states (and the inverses), i.e. the energy landscape is dominated by one large valley, see right half of Fig. 9.6. The main signature of this behavior is that the distribution of overlaps is a delta function. Please note that for finite system sizes, the distribution of overlaps always has a finite width. Thus, the delta function is found only in the thermodynamic limit $N \to \infty$. Recently, many results have been made available addressing this question, especially with numerical techniques. Since near the transition temperature T_G the systems are very difficult to equilibrate, one is restricted to small system sizes and even lower temperatures are not accessible using the usual Monte Carlo methods. As a consequence, a definite answer has not yet been obtained.

Here, we want to investigate whether the ground-state landscape of realistic spin glasses is better described by the mean-field like or the Droplet picture. The validity of the result will be restricted to exactly $T = 0$, at finite temperatures the behavior might change. Firstly it should be stressed that even for the original SK model, the true ground state is indeed unique, because of the Gaussian distribution of the interactions. No two bonds have exactly the same strengths, so every flip of a spin would increase the energy, sometimes by a small fraction. Therefore, exactly at $T = 0$ the overlap distribution is a delta function even for the SK model. The complicated hierarchy of states is found only for all $T > 0$.
For realistic spin glasses with Gaussian distribution of the interactions, the same arguments hold. The ground state is unique, so the distribution of overlaps will be a delta peak in the thermodynamic limit.
On the other hand, a $\pm J$ spin glass may exhibit *free spins*, i.e. spins which can be reversed without changing the energy. A system with N' isolated free spins has $2^{N'}$ different ground states. Furthermore, there are also neighboring free spins, so by flipping a free spin neighbors may become free as well or vice versa. Therefore, the degree of degeneracy is not easy to calculate. For the mean-field spin glass with $\pm J$ interaction, a free spin must have exactly half of the other spins pointing in the "right" and half of the spins pointing in the "wrong" direction. Thus, a mean-field system with an even number of spins has only one ground state as well. In general, it is so not clear so far how large the degree of degeneracy is and what $P(q)$ at $T = 0$ looks like.
Nevertheless, for finite-dimensional $\pm J$ spin glasses, the case we are interested in, each spin has only 6 neighbors, which is a small even number. The number of free spins grows linearly with the system size. Consequently, the number of ground states grows exponentially with the number of spins, so the ground state entropy is finite. Furthermore, the question about the structure of the ground-state landscape is much more difficult to answer than in the case of the Gaussian distribution. For realistic spin glasses, it may be possible that $P(|q|)$ is non-trivial even for true ground states. Since for numerical simulations we are always restricted to finite sizes, the main question concerning the mean-field like and Droplet pictures is: does $P(|q|)$ for the three-dimensional $\pm J$ model remain broad or does it converge to a delta function in the thermodynamic limit? In the next sections algorithms are presented which allow many different ground states of spin glasses to be calculated efficiently, enabling the calculation of the distribution of overlaps.

9.2 Genetic Cluster-exact Approximation

One could think that the ground-state calculation for spin glasses can be performed by the same method which was applied for the random-field model and the diluted antiferromagnet in a field (DAFF) (see Chap. 6). The main idea was to build an equivalent network and to calculate the maximum flow through the network. A precondition is that all pair interactions are positive, which is definitely not true for the spin-glass model. But, you may remember, that for the diluted antiferromagnet

even all bonds are negative. Here, the transformation to a network is possible, since via a gauge transformation all bonds can be made positive. We will now give an example of such a gauge transformation and we will show why it does not work for spin glasses.

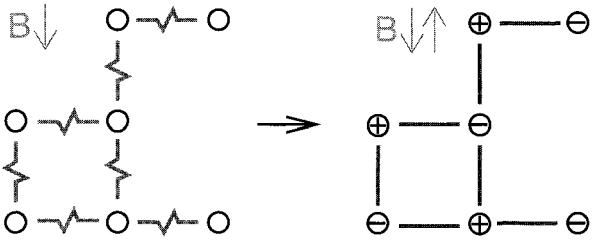


Figure 9.7: A small diluted antiferromagnet in a field B. Via a gauge transformation all bonds can be made ferromagnetic, while the magnetic field becomes staggered.

In the left part of Fig. 9.7 a small DAFF is shown. The basic idea of the gauge transformation is to multiply every second spin in a checkerboard manner with -1 and leave the other half unchanged

$$\sigma_i' \equiv t_i \sigma_i \equiv (-1)^{x+y+z} \sigma_i \tag{9.6}$$

where x, y, z are the spatial coordinates of spin i and t_i is the sign of the local gauge transformation. The resulting system is shown in the right part of Fig. 9.7. All bonds have turned ferromagnetic. The transformation has the following effect on the Hamiltonian, in particular the sign of the quadratic term has changed, so in the σ_i' a ferromagnet is obtained:

$$H = \sum_{\langle i,j \rangle} J\sigma_i\sigma_j - \sum_i B_i\sigma_i \;\rightarrow\; H' = -\sum_{\langle i,j \rangle} J\sigma_i'\sigma_j' - \sum_i (-1)^{x+y+z} B_i\sigma_i' \tag{9.7}$$

Please note that $t_i^2 = 1$ and $t_i t_j = -1$ if i, j are neighbors. In H', the magnetic field is no longer uniform; it is staggered. This does not affect the applicability of the algorithm, because it can treat arbitrary local fields.

Figure 9.8: A small $\pm J$ spin glass. Via a gauge transformation the aim is to make all bonds ferromagnetic. It fails because the system is *frustrated*.

Why is this transformation not suitable for spin glasses? In Fig. 9.8 a tiny spin glass with four spins is shown. Let us try to apply the gauge transformation. Because bonds

with different signs are present, the signs of the transformation have to be chosen for each spin individually. We start with the spin in the upper left corner. Without loss of generality we leave it unchanged: $\sigma_1' = \sigma_1$, i.e. $t_1 = 1$. Now we turn to the spin below. The bond between this and the first spin is antiferromagnetic, so we choose $\sigma_2' = -\sigma_2$ ($t_2 = -1$) which makes the bond ferromagnetic. In a similar way we have to choose $\sigma_3' = -\sigma_3$ ($t_3 = -1$). Now we are left with the spin in the upper right corner. It is connected by one ferromagnetic bond and one antiferromagnetic bond to its neighbors. Consequently, whatever sign for the transformation of σ_4 we select, always one bond remains negative. The reason that not all bonds can be made positive is equivalent to the fact that in a ground state it is not possible to satisfy all bonds, i.e. the presence of frustration. In general a system is frustrated if closed loops exist which have an odd number of antiferromagnetic bonds, e.g. an antiferromagnet on a triangular lattice.
So we can see that due to the existence of frustration a spin glass cannot be transformed into a ferromagnet. This is the reason why the fast algorithms for the ground-state calculations cannot be applied in this case. In fact it can be shown that the ground-state problem for spin glasses is NP-hard [18], i.e. only algorithms with exponentially increasing running time are known. As an exception, for the special case of two-dimensional spin glasses without external field and with periodic boundary conditions in at most one direction, efficient polynomial algorithms [19] for the calculation of exact ground states are available. The most recent applications are based on matching algorithms [20, 21, 22, 23, 24, 25, 26, 27, 28], see also Chap. 10, other exact approaches can be found in Refs. [29, 30]. Recently results for systems of size 1800×1800 were obtained [31].
For the general problem several algorithms are available, for an overview see [32, 33]. The simplest method works by enumerating all 2^N possible states and obviously has an exponential running time. Even a system size of 4^3 is too large. The basic idea of the so called *Branch-and-Bound* algorithm [34] is to exclude the parts of the state space where no low-lying states can be found, so that the complete low-energy landscape of systems of size 4^3 can be calculated [35, 36, 37, 38]. Also transfer-matrix techniques have been applied [39] for 4^3 spin glasses. To evaluate all ground states, similar algorithms have been applied to two-dimensional systems as well [40, 41, 42, 43, 44, 45].
A more sophisticated method called *Branch-and-Cut* [46, 47] works by rewriting the quadratic energy function as a linear function with an additional set of inequalities which must hold for the feasible solutions. Since not all inequalities are known a priori, the method iteratively solves the linear problem, looks for inequalities which are violated, and adds them to the set until the solution is found. Since the number of inequalities grows exponentially with the system size, the same holds for the computation time of the algorithm. With Branch-and-Cut anyway small systems up to 8^3 are feasible. Further applications of these method can be found in Refs. [48, 49].
Since finding ground states in three-dimensional systems is computaionally very demanding, several heuristic methods have been applied. At the beginning simulated annealing (see Chap. 11) was very popular, recent results can be found in Refs. [50, 51]. But usually it is very difficult to obtain true ground states using this technique. A more sophisticated method is the multicanonical method [52], which is based on Monte Carlo simulations as well, but incorporates a reweighting scheme to speed

up the simulation. Very low lying states of some systems up to size 12^3 have been obtained [53, 54, 55, 56, 57]. Heuristics which are able to find low energy states but not true ground states unless the systems are very small can be found in Refs. [58, 59, 60]. Another approach included the application of neural networks [61].
Usually with minor success several variants of genetic algorithms (see Chap. 8) have been applied [62, 63, 64, 65, 66, 67, 68]. In a similar approach the population of different configurations was replaced by a distribution describing the population [69]. At first sight these approaches looked very promising, but it is not possible to prove whether a true ground state has been found. One always has to check carefully, whether it is possible to obtain states with slightly lower energy by applying more computational effort. A genetic approach [70, 71] which was designed especially for spin glasses has been more successful. Realizations up to size 10^3 with Gaussian distributions of the bonds have been studied recently [72, 73, 74], supporting the Droplet picture for spin glasses. A combination with a recursive renormalization method can be found in Ref. [75]. The basic idea is to divide the problem into subproblems of smaller size and treat the subproblems in the same way. The technique has been applied to finite-dimensional systems with Gaussian distribution of the bonds [76, 77], again up to size 10^3, but also to other combinatorial optimization problems like the TSP.
The method presented in this chapter is able to calculate true ground states [78] up to size 14^3. The method is based on the special genetic algorithm of Ref. [70] as well but also incorporates the cluster-exact approximation (CEA) [79] algorithm. CEA is an optimization method designed specifically for spin glasses, but it should be applicable to all problems where each element (here: spin) interacts with a small number of other elements. Its basic idea is to transform the spin glass in a way that anyway the max-flow methods can be applied, which work only for systems exhibiting no frustrations. Next a description of the genetic CEA is given. We start with the CEA part and later we turn to the genetic algorithm.
The basic idea of CEA is to treat the system as if it were possible to turn it into a ferromagnet: a cluster of spins is built in such a way that all interactions between the cluster spins can be made ferromagnetic by choosing the appropriate gauge-transformation signs $t_i = \pm 1$. All other spins are left out ($t_i = 0$). In this way the frustration is broken. The interaction with the non-cluster spins is included in the total energy, but the non-cluster spins are not allowed to flip, they remain fixed. Usually one starts with some random orientations of all spins, so the non-cluster spins just keep this orientation. Let us consider a pair interaction $J_{ij}\sigma_i\sigma_j$ between a cluster spin σ_i and a non-cluster spin σ_j. Since σ_j is kept fixed, say $\sigma_j = +1$, we can write $J_{ij}\sigma_i\sigma_j = J_{ij}\sigma_i$. Now the interaction has turned into an interaction of spin j with a local field of size $B_i = \sigma_j = +1$. After the construction of the cluster has been completed, for each cluster spin all interactions with non-cluster spins and the original local field B_i are summed up to calculate the new local field:

$$B_i' = t_i(B_i + \sum_j (1 - |t_j|)\sigma_j) \tag{9.8}$$

The sum runs over all neighbors j of spin i. Because of the factor $(1 - |t_j|)$ only the interactions with non-cluster spins are included into the new local field. The gauge-

transformation factor t_i compensates the transformation of spin i (please remember $t_i^2 = 1$). The Hamiltonian of the resulting system reads

$$H_c = -\sum_{\langle ij \rangle} J_{ij} t_i t_j \sigma_i' \sigma_j' - \sum_i B_i' \sigma_i' + C \tag{9.9}$$

The constant C contains the interactions among non-cluster spins. Since the these spins will not change their orientations during the calculation, C does not change. Thus, it can be neglected for the ground-state calculation. Since the signs t_i have been chosen such that all pair interactions $J_{ij}' = J_{ij} t_i t_j$ are either zero or ferromagnetic and because several spins may not be included into the cluster ($t_i = 0$), the system we have obtained is a diluted ferromagnet with random local fields. As we have learned before, for this system the ground state can be calculated in polynomial time using the fast methods of Chap. 6.

How does the construction of the non-frustrated cluster work? The method is similar to the construction demonstrated in the example shown in Fig. 9.8: spins are chosen iteratively. If it is possible to make all adjacent bonds positive by a gauge transformation, the sign of the transformation is chosen accordingly. It is important to note that only those bonds have to become ferromagnetic where $t_j \neq 0$ on the other end of the bond, i.e. for bonds connecting cluster spins. The other bonds may have been considered already or they can be treated later on. The algorithm for the cluster construction is presented below. The variable δ_i is used to remember which spins have been treated already.

algorithm build-cluster($\{J_{ij}\}$)
begin
  Initialize $\delta_i := 0$ for all i;
  Initialize $t_i := 0$ for all i;
  while there are untreated spins ($\delta_i = 0$)
  begin
    Choose a spin i with $\delta_i = 0$;
    $\delta_i := 1$;
    Set $A := \{j | j$ is neighbor of i and $t_j \neq 0\}$;
    if $A = \emptyset$ **then**
      $t_i := 1$;
    else if $\forall j \in A : J_{ij} t_i$ has same sign α **then**
      $t_i := \alpha$;
    else
      $t_i := 0$
  end
  return($\{t_i\}$);
end

The following example should illustrate how the cluster is built.

Example: Construction of the non-frustrated cluster

A small two-dimensional spin glass consisting of nine spins is treated. The initial situation is shown in Fig. 9.2. We assume that first the center spin 5 is chosen. Since there is no cluster at the beginning (all $t_i = 0$), the first spin has no neighbors in the cluster, so $t_5 = 1$. The effect of the gauge-transformation is shown in the figure as well, i.e. for each bond the result of the gauge transformation is shown. Setting $t_5 = 1$ leaves all bonds unchanged.

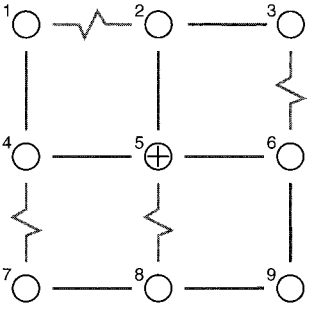


Figure 9.9: CEA algorithm, example construction of a non-frustrated cluster. Initially, the gauge-transformation sign for the center spin can be set to 1. Always the sign of the bond after applying the gauge transformation is shown.

Next, we assume that spin 6 is chosen. It has one neighbor in the cluster, spin 5. We obtain $\alpha = J_{65}t_5 = 1$. Consequently, we set $t_6 := 1$. Again, this transformation leaves the bonds unchanged. Then spin 8 is considered. It has one neighbor in the cluster: spin 5, with $\alpha = J_{85}t_5 = -1$. Therefore, we set $t_8 = -1$ to turn the bond between spins 5 and 8 ferromagnetic. The resulting situation is presented in Fig. 9.10. Spin 9 cannot be added to the cluster. It has two neighbors with $t_i \neq 0$, spins 8 and 6 with $J_{98}t_8 = -1 \neq 1 = J_{96}t_6$.

During the following iteration spin 2 is chosen. Similar to the preceding steps $t_2 = 1$ is obtained. If now spin 3 is taken into account, we see that it has two neighbors, which already belong to the cluster: spin 2 and spin 6. We get $J_{32}t_2 = 1 \neq -1 = J_{36}t_6$. Therefore, spin 3 cannot become a member of the cluster, so $t_3 = 0$.

If we assume that next spins 4,7 are treated, then both of them can be added to the cluster without creating frustration. The gauge-transformation signs obtained are $t_4 = 1$, $t_7 = -1$. Last, spin 1 cannot be added to the cluster. The final situation is shown in Fig. 9.11.

□

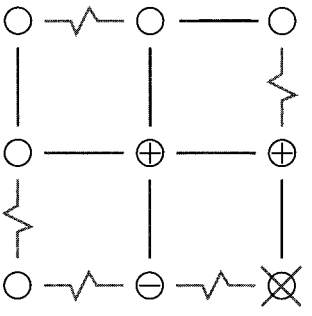


Figure 9.10: CEA algorithm, example construction of a non-frustrated cluster. The four spins in the lower right are treated. Spins 5, 6 and 8 can be included in the cluster, while spin 9 cannot be included.

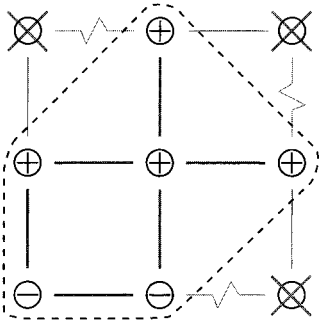


Figure 9.11: CEA algorithm, example construction of a non-frustrated cluster. Final situation.

In the algorithm above, the order in which the spins i are chosen is not specified. Several heuristics are possible. From our tests we have found that the most efficient method so far is as follows. The spins are selected randomly among the spins which have not been treated so far. For each spin the probability of selection within the current step is proportional to the number of unsatisfied bonds adjacent to it. These numbers are calculated using the current spin configuration. Thus, a spin with a high (bad) contribution to the total energy is selected more often than a spin with a low (negative = good) contribution. This results in a very quick decrease in the energy, see below.

After the cluster of non-frustrated spins has been constructed and the cluster ground state has been obtained, the cluster spins are set accordingly, while the other spins keep their previous orientation. For the whole system this means that the total energy has either been decreased or it has remained the same, because the cluster spins have been oriented in an optimum way. Please note that for the total system no ground

state has usually been found! But, since the cluster is built in a random way, another run can be performed and the cluster will probably be constructed differently. So the whole step can be repeated and maybe again the energy of the whole system is decreased. This decrease is very efficient in the beginning, because usually the clusters are quite big. For three-dimensional spin glasses on average about 55% of all spins are members of a non-frustrated cluster (70% for two dimensions). In Fig. 9.12 a sample run for a two dimensional spin glass is shown. Initially the spin configuration was chosen randomly, which results on average in energy 0 at step 0. In the beginning the energy decreases rapidly (please note the logarithmic scale of the x-axis). Later on the energy levels off and cannot be decreased further, the system has run into one (probably local) minimum.

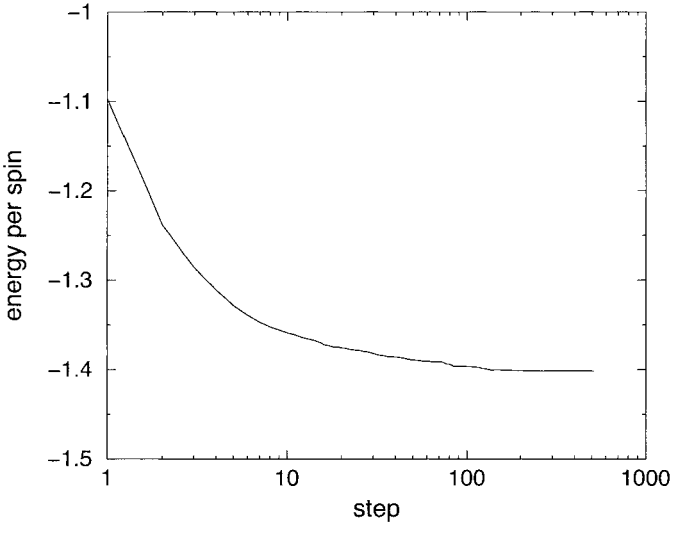


Figure 9.12: Energy per spin for a two-dimensional sample spin glass treated with the CEA algorithm as a function of the number of the step.

From comparisons with exact results for small systems, we know that the states found in this way really have a very low energy, but, except for very small systems, they are usually slightly above the true ground states. To make the method even more efficient, and to find true ground states, even for larger system sizes, the CEA method can be combined with a genetic algorithm (for an introduction, see Chap. 8).
The genetic algorithm starts [70] with an initial population of M_i randomly initialized spin configurations (= *individuals*), which are linearly arranged using an array. The last one is also a neighbor of the first one. Then $n_\mathrm{o} \times M_\mathrm{i}$ times two neighbors from the population are taken (called *parents*) and two new configurations called *offspring* are created. For that purpose the *triadic crossover* is used which turns out to be very efficient for spin glasses: a mask is used which is a third randomly chosen (usually

distant) member of the population with a randomly chosen fraction of 0.1 of its spins reversed. In a first step the offspring are created as copies of the parents. Then those spins are selected where the orientations of the first parent and the mask agree [80]. The values of these spins are swapped between the two offspring.

Example: Triadic crossover

In Fig. 9.2 a triadic crossover for one-dimensional spin glasses is presented. In the top part the parents and the mask are shown, below the resulting offspring. The idea behind this special type of crossover is as follows. It is assumed that for some regions in the systems, special domains of spins exist with a very low energy. Furthermore, it is assumed that during the optimization process these domains emerge automatically. During a crossover one would like to keep these domains stable, so one compares different configurations and identifies the domains as subsets of spins which agree in both spin configurations.

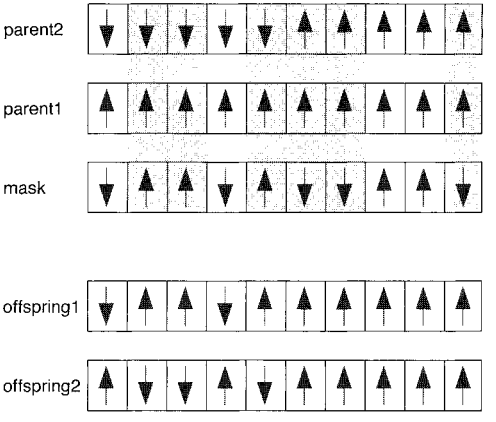


Figure 9.13: The triadic crossover. Initially the offspring are copies of the parents. Then the spins between the offspring are exchanged at positions where parent1 and a mask agree. The mask belongs to the population of configurations as well.

The effect can be seen especially in this example, because the first halves of both parents are assumed to be inverted relative to each other. Consequently, in that part the offspring are equal to the mask and to its mirror configuration, respectively. This is the reason why the mask must be similar to an existing low energy configuration. Since the parents agree in some spins, the offspring are only partly copies of the mask/its inverse. □

Then a *mutation* with a rate of p_m is applied to each offspring, i.e. a randomly chosen fraction p_m of the spins is reversed.
Next for both offspring the energy is reduced by applying CEA. This CEA minimization step is performed n_{min} times for each offspring. Afterwards each offspring is compared with one of its parents. The offspring/parent pairs are chosen in the way that the sum of the phenotypic differences between them is minimal. The phenotypic difference is defined here as the number of spins where the two configurations differ. Each parent is replaced if its energy is not lower (i.e. not better) than the corresponding offspring. After this whole step is conducted $n_o \times M_i$ times, the population is halved: from each pair of neighbors the configuration which has the higher energy is eliminated. If more than 4 individuals remain the process is continued otherwise it is stopped and the best remaining individual is taken as result of the calculation.
The following representation summarizes the algorithm.

```
algorithm genetic CEA({J_ij}, M_i, n_o, p_m, n_min)
begin
   create M_i configurations randomly;
   while (M_i > 4) do
   begin
      for i = 1 to n_o × M_i do
      begin
         select two neighbors;
         create two offspring using triadic crossover;
         do mutations with rate p_m;
         for both offspring do
         begin
            for j = 1 to n_min do
            begin
               construct unfrustrated cluster of spins;
               construct equivalent network;
               calculate maximum flow;
               construct minimum cut;
               set new orientations of cluster spins;
            end
            if offspring is not worse than related parent
            then
               replace parent with offspring;
         end
      end
      half population; M_i = M_i/2;
   end
   return one configuration with lowest energy
end
```

The whole algorithm is performed n_R times and all configurations which exhibit the

Table 9.1: Simulation parameters ($d = 2$): L = system size, M_i = initial size of population, n_o = average number of offspring per configuration, n_min = number of CEA minimization steps per offspring, τ = typical computer time per ground state on an 80MHz PPC601.

L	M_i	n_o	n_min	p_m	τ (s)
5	8	1	1	0.05	0.02
10	16	1	2	0.05	0.4
14	16	4	2	0.05	3
20	32	8	2	0.05	30
32	128	8	2	0.05	780
40	512	8	2	0.05	5400

lowest energy are stored, resulting in $n_G < n_R$ statistically independent ground-state configurations (*replicas*).

To obtain true ground states, one has to choose the simulation parameters in a proper way, i.e. initial size of population M_i, number of iterations n_o, mutation rate p_m and number of CEA steps n_min. We have tested many combinations of parameters for up to ten different realizations of the disorder for two-, three- and four-dimensional $\pm J$ spin glasses with periodic boundary conditions, see [78, 81, 82]. We assume that the lowest lying states are actually found (using a CPU time of t_min), if it is not possible to obtain configurations with lower energy by applying a parameter set which results in a CPU time $4 \times t_\mathrm{min}$. The parameter sets have to be determined for each system size separately. Usually one starts with small systems where it is relatively easy to find true ground states. The parameter sets established for small systems are taken as initial sets for testing larger system sizes. With genetic CEA system sizes up to 40^2, 14^3 and 7^4 can be treated. The resulting parameter sets are presented in Tables 9.1, 9.2 and 9.3. Please note that these parameters are not optimal in the sense that it may be possible to find parameters which yield lower running times while still ensuring true ground states.

For small system sizes like 6^3, the configurations found in this way can be compared with results from exact Branch-and-Bound calculations. In all cases the combination of the genetic algorithm and CEA actually found the exact ground states. Thus, one can be very confident that the configurations are also true ground states, which were obtained for larger systems sizes with the parameter sets established as explained. But, please note that this is not a proof. The method is just a very good approximation method, as it is usual for genetic algorithms.

Also shown in the tables are the running times of the genetic CEA algorithm implemented on 80MHz Power-PC computers. One can observe that indeed the calculation of true ground states for spin glasses takes a time which increases strongly with system size, almost exponentially.

Table 9.2: Simulation parameters ($d = 3$): L = system size, M_i = initial size of population, n_o = average number of offspring per configuration, n_min = number of CEA minimization steps per offspring, τ = typical computer time per ground state on an 80MHz PPC601.

L	M_i	n_o	n_min	p_m	τ (s)
3	16	3	1	0.2	0.2
4	16	3	1	0.2	0.5
5	16	4	2	0.2	3
6	16	4	2	0.2	5
8	32	4	5	0.2	70
10	64	6	10	0.2	960
12	64	12	10	0.2	4200
14	256	14	10	0.2	32400

Table 9.3: Simulation parameters ($d = 4$): L = system size, M_i = initial size of population, n_o = average number of offspring per configuration, n_min = number of CEA minimization steps per offspring, τ = typical computer time per ground state on an 80MHz PPC601.

L	M_i	n_o	n_min	p_m	τ (s)
2	16	1	1	0.1	0.04
3	16	4	4	0.1	3
4	16	4	4	0.1	14
5	256	6	10	0.1	4800
6	256	6	10	0.1	7300
7	512	10	20	0.1	80000

9.3 Energy and Ground-state Statistics

In this section some results obtained with the genetic CEA are presented. The ground-state energy for a very large system is estimated using a finite-size scaling fit. Secondly, it is shown that when calculating many different and independent ground states, the method presented so far has to be extended to obtain physically correct results.

It is not possible to treat very large systems numerically, e.g. one cannot simulate models having 10^{26} particles. To overcome this restriction, one can apply the technique of finite-size scaling to extrapolate results, that were obtained for different small systems sizes, to very large systems. As an example we will consider the average ground-state energy of three-dimensional $\pm J$ Ising spin glasses. In Fig. 9.14 the average ground-state energy per spin is shown as a function of the linear system size L for $3 \leq L \leq 14$. The average was taken over about 1000 realizations for each system size, except for $L = 14$, where 100 realizations were considered, because of the huge numerical effort.

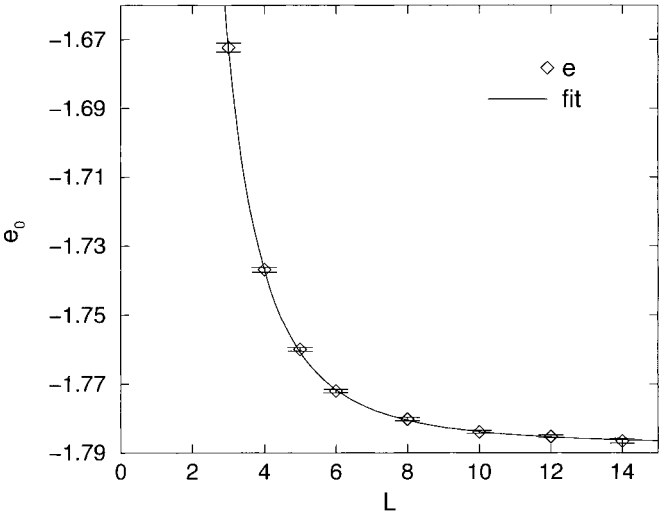


Figure 9.14: Average ground-state energy per spins e_0 as a function of system size L for three-dimensional Ising $\pm J$ spin glasses.

With increasing system size the average ground-state energy decreases monotonically. Thus, one can try to fit a function of the form $e_0(L) = e_0(\infty) + aL^{-b}$ to the data points, resulting in an estimate of the ground-state energy $e_0(\infty)$ of a very large, i.e. nearly infinite, system. Please note that so far no justification for this special form of the fitting function has been found. It just fits very well. It has been reported in the literature that other functions such as exponentials have also been tried, but the results for $e_0(\infty)$ are similar. When using the fit-procedure of the gnuplot program, which is available for free, the value $e_0(\infty) = -1.7876(3)$ is obtained (the values of a, b are not important here). This value means that in a ground state about 0.6 unsatisfied bonds per spin exist (if all bonds were satisfied this would result in a ground-state energy of -3, each unsatisfied bond increases the energy by a value of 2).

For two and four dimensions, values of $e_0(\infty) = -1.4015(3)$ and $e_0(\infty) = -2.095(1)$ have been obtained, respectively, using genetic CEA. These are currently the lowest values found for the ground-state energies of spin glasses. This is another indication that the genetic CEA method is indeed a very powerful optimization tool.

Apart from obtaining some values, the calculation of ground-state energies can tell us a lot more about the underlying physics. In particular, one is able to determine whether a system exhibits a transition from an ordered low-temperature state to a disordered high-temperature state at a non-zero temperature $T_c > 0$.

To show how this can be done, first the simple Ising ferromagnet is considered. It is known from basic courses in statistical physics [83], that the one-dimensional Ising chain exhibits a phase transition only at $T_c = 0$, i.e. for all finite temperatures the

Ising chain is paramagnetic. The two-dimensional ferromagnet on the other hand has a ferromagnetically ordered phase ($T_c = 2.269J$, where J is the interaction constant).

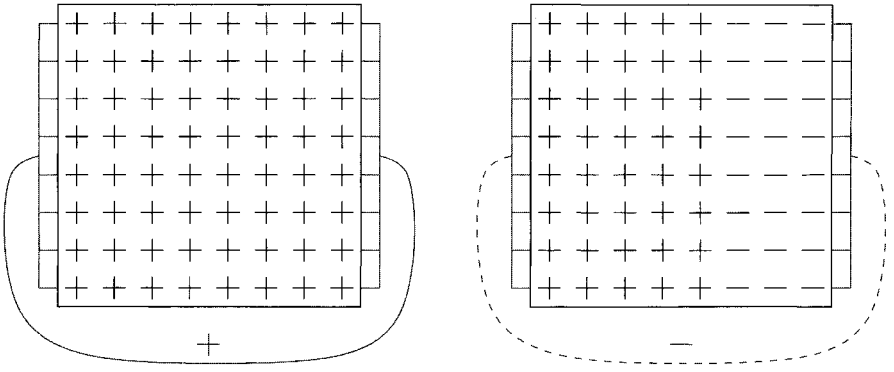


Figure 9.15: A two-dimensional Ising ferromagnet. In the case where all boundary conditions are periodic (left), the system is ferromagnetic at $T = 0$. When antiperiodic boundary conditions are imposed in one direction (right), a domain wall is introduced, raising the ground-state energy by an amount which is proportional to the length of the domain wall.

Why the one-dimensional Ising ferromagnet does not exhibit an ordered phase at $T > 0$, while the two-dimensional has one, can be seen from ground-state calculations as well. Consider a two-dimensional ferromagnet of size $N = L \times L$ with periodic boundary conditions (pbc) in all directions. Since all bonds are ferromagnetic, in the ground state all spins have the same orientation, see the left half of Fig. 9.15. The ground-state energy is $E_{\text{pbc}} = -2NJ$. Now *antiperiodic* boundary conditions (abc) in one direction are considered. This can be achieved by turning all bonds negative, which connect the spins of the first and the last column. Again the ground state is calculated. Now, it is favorable for the spins of the first and the last row to have different orientations. This means we get two domains, in one domain all spins are *up*, in the other all spins are *down*. This introduces two domain walls in the system. The first domain wall is between the first and the last row, connected through the boundary. This is compatible with the negative sign of the bonds there. The other domain wall is somewhere else in the system. This means the bonds on the second domain wall are ferromagnetic, while the spins left and right of the domain wall have different orientations. Thus, L bonds are broken[1], raising the ground-state energy E_{abc} by an amount of $2LJ$. As a consequence, we get $E_{\text{abc}} = -2NJ + 2LJ$. The difference $\Delta = E_{\text{abc}} - E_{\text{pbc}} = 2LJ$ is called the *stiffness energy*. Here it depends on the linear system size L. When performing the thermodynamic limit $L \to \infty$, the

[1] Please note that the fully ordered state is still possible. In this case both domain walls fall onto each other. But again L bonds are broken.

stiffness energy diverges, thus the ratio of the thermodynamical weights

$$\frac{p(abc)}{p(pbc)} = \frac{N_{\mathrm{abc}} \exp(-E_{\mathrm{abc}}/T)}{N_{\mathrm{pbc}} \exp(-E_{\mathrm{pbc}}/T)} = L \exp(-\Delta/T) \tag{9.10}$$

goes to zero. This indicates that the two-dimensional ferromagnet exhibits some kind of stiffness against flips of finite domains. Thus, they will not occur at low temperatures, which means that the ferromagnetically ordered state is stable at low temperatures. For higher temperatures more complicated domains with longer domain walls also have to be considered, thus for entropic reasons at some temperature $T_c > 0$ the ordered state is destroyed. Please note that only the fact that $T_c > 0$ holds can be shown from ground-state calculations, the value of T_c itself cannot be calculated this way.

In case one performs the same considerations for the one-dimensional spin chain, one sees immediately that when flipping the boundary conditions from periodic to antiperiodic, again a domain wall is introduced. But now it has only the length *one* for all chain lengths L. This means the stiffness energy $\Delta = 2J$ does not depend on the system size. In the thermodynamic limit an arbitrary small temperature is sufficient to break the long range order.

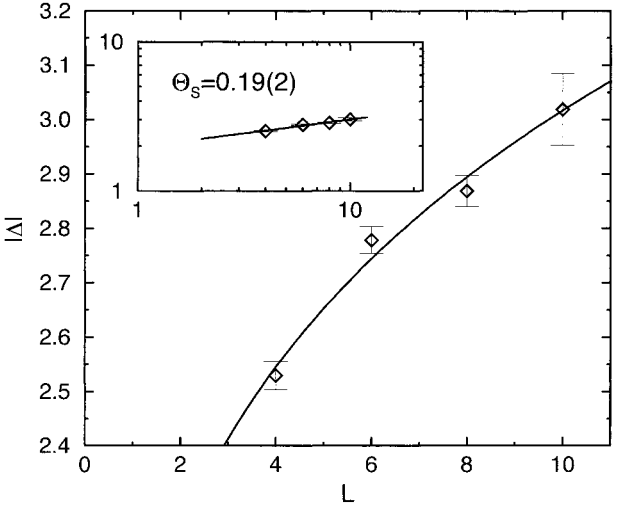


Figure 9.16: Stiffness energy $|\Delta|$ as a function of system size L. The line represents the function $|\Delta(L)| = aL^{\Theta_S}$ with $\Theta_S = 0.19(2)$. The inset shows the same figure on a log-log scale. The increase of $|\Delta|$ with system size indicates that in 3d Ising spin glasses an ordered phase exists below a non-zero temperature T_c. The figure is taken from [78].

The same kind of calculations can be applied to spin glasses as well. One obtains the ground states with periodic and antiperiodic boundary conditions, respectively, and calculates the stiffness energy $|\Delta| = |E_{\mathrm{abc}} - E_{\mathrm{pbc}}|$. Since spin glasses are disordered systems, here again an average over the disorder has to be taken. It is necessary to take the absolute value, because for some systems the state with pbc has a lower energy, while for other the abc state is favorable. From theoretical considerations [84, 85], one obtains a behavior of the form $\Delta(L) = L^{\Theta_S}$, where Θ_S is the *stiffness exponent*.
If $\Theta_S > 0$ one expects that the system has an ordered low-temperature phase, while for $\Theta_S < 0$ only ordering exactly at $T = 0$ should exist. In Fig. 9.16 the stiffness energy of three-dimensional spin glasses as a function of system size L is shown. The value of $\Theta_S = 0.19$ indicates that the three-dimensional spin glass has $T_c > 0$, which is compatible with Monte Carlo simulations [86, 87]. On the other hand, for two-dimensional spin glasses, a value of $\Theta_S < 0$ has been found [25], i.e. in two dimensions, spin glasses seem to have an ordered phase only at $T = 0$.
Now we turn from the analysis of ground-state energies to the ground-state configurations itself. As it was pointed out before, we have to perform two kinds of averages to characterize the behavior of the ground-state landscape. The first kind of average is that we have to consider many different realizations of the random bonds. Secondly, for each realization, many independent configurations have to be calculated. The statistical weight $p_{\{\sigma_i\}}$ of a spin configuration $\{\sigma_i\}$ with energy $E = H(\{\sigma_i\})$ is $p_{\{\sigma_i\}} = \exp(-H(\{\sigma_i\})/T)/Z$ where Z is the partition sum. In the limit of zero temperature only the ground states contribute to the behavior of the system. As an important consequence, we can see from this formula that each ground state contributes with the same weight, because all of them have exactly the same energy.
The genetic CEA method returns in each run at most one ground state. Thus, any such an algorithm which is used to sample a ground-state landscape of spin glasses (or any other system) must return each ground state with the same probability. In that case, by taking an average over different ground states, it is ensured that the physical quantities calculated indeed represent the true thermodynamic behavior.
For the algorithm which has been presented in the preceding section, it is not known a priori, whether the ground states obey the correct statistics, i.e. whether each ground state appears with the same probability. To test this issue [88], we take one small spin glass of size $N = 5^3$ and let the algorithm run for $n_r = 10^5$ times. Each ground state which appears is stored and it is counted how often each ground state is found. This gives us a histogram of the frequencies of how often each ground state is calculated by the algorithm. The result is shown in Fig. 9.17. Obviously the large deviations from state to state cannot be explained by the presence of statistical fluctuations. Thus, genetic CEA samples different ground states from the same realization with different weights.
Consequently, when just the configurations are taken as they are given by the algorithm, the physical quantities calculated by taking an average are not reliable [89, 90]. This is true especially for the overlap parameter q which is used to compare different configurations and evaluate the ground-state landscape.
It should be pointed out that this drawback does not appear only for the genetic CEA method. No algorithm known so far guarantees the correct statistics of the ground

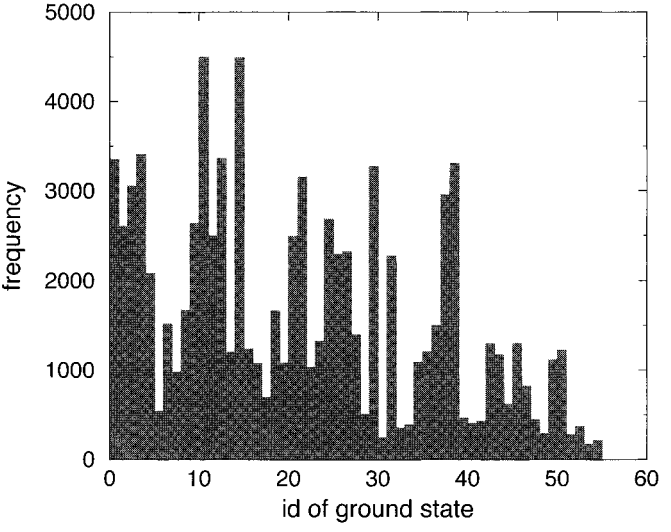


Figure 9.17: Number of times each ground state is found in 10^5 runs of the genetic CEA algorithm for one sample realization of size $N = 5^3$. This realization has 56 different ground states. Obviously, different states have significantly different probabilities of being calculated in one run. Example: ground state 10 occurred about 4500 times among the 10^5 outcomes while state 56 was found only about 200 times.

states. This is true also for methods which are based on thermodynamics itself, like simulated annealing (see Chap. 11), which is very often used to study ground states of disordered systems. For that algorithm, if the rate of the temperature decrease is chosen in a way that one in two runs results in a true ground state, then for the $N = 5^3$ system treated above a similar histogram is obtained. By decreasing the cooling process the weight of the different ground states become more equal, but one has to cool 100 times slower, i.e. spend 100 time the computational effort, to find each ground state with almost the same probability.
So far no method is known which allows the algorithms to be changed in such a way that each ground state appears with the proper weight. On the other hand, it is possible by applying a post-processing step to remove the bias which is imposed by the algorithms. A suitable method is presented in the following section.

9.4 Ballistic Search

For small systems, where it is possible to calculate all degenerate ground states, it is very easy to obtain the correct thermodynamic distribution: one just has to use each ground state once when calculating physical quantities by averaging. For larger system sizes, it is impossible to obtain all ground states. Even systems of size $N = 8^3$

exhibit about 10^{16} ground states. In this case one can only sample a small subset of states. But then each of all existing ground states must have the same probability appearing in the sample to guarantee the correct result.
The basic concept of the method, which ensures correct thermodynamic statistics of the ground states, is to apply *post processing*. The method which calculates the ground states remains unchanged. The input to the post processing is a set of ground states. The output is another set of ground states, which now have the correct thermodynamic weights, see Fig. 9.18. This post processing can be used in conjunction with all kinds of methods which calculate ground states, it is not restricted to the genetic CEA algorithm.

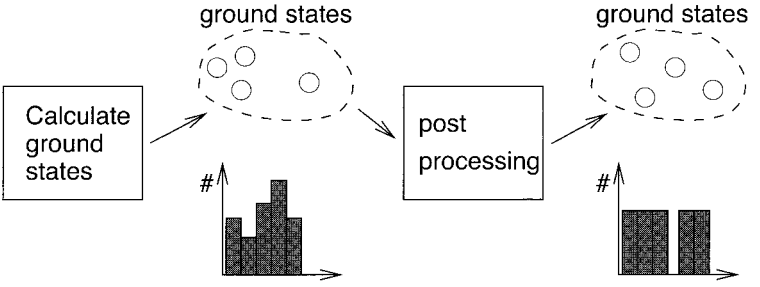


Figure 9.18: The correct thermodynamical weight of the ground states is ensured by a post-processing step which is applied after a set of ground states has been calculated.

Now the idea behind the post processing is explained. Assume we investigate a three-dimensional spin glass of size $N = 8^3$ which has about 10^{16} different ground states. Using genetic CEA we calculate 100 ground states. From the preceding section we know that some ground states are more likely to be found than others. The first step of the post processing consist of dividing the 100 states into groups according to some criterion. A convenient criterion is explained later on. For the moment we take a toy criterion, we assume that ground states have colors, e.g. blue, red and green and that they are divided according their colors.
Next we have to get access to the other ground states, which have not been found before. They have colors as well. We assume that it is possible to perform $T = 0$ Monte Carlo simulations which preserve the color and visits only ground states. This is similar to a Monte Carlo simulation which preserves the magnetization by flipping only pairs of up/down spins. We assume that the MC simulation is *ergodic*, that means starting with a blue ground state we can access all blue ground states by just running the simulation long enough. Furthermore, we assume that the simulation satisfies detailed balance, that means each ground state obtains its correct thermodynamic weight within its group. This means after performing n_{RUN} runs of the simulation we have a set of n_{RUN} ground states (the initial states are not used any more), where all existing blue ground states have the same probability of being in this set, all red have the same probability, etc..

Now, we still do not know whether a red and a blue ground state have the same probability of being visited during the MC simulation. To obtain the final sample where all ground states have the same probability, we have to estimate the size of each group. This must be done using the small number of states we have obtained for each group by the MC simulation. We cannot just count all ($\sim 10^{15}$) blue states, simply because we do not have them all available. Later on we will explain how the group sizes are estimated for the real criterion we use. For the moment we just assume that it is possible somehow to estimate the total number of blue states etc. for each realization.

The final sample of states is obtained by drawing from each group a number of states which is proportional to the size of the group, so each group is represented in the sample with the correct weight. Since we have made all ground states within each group equiprobable by the MC simulation, we end up with a sample (e.g. of size 100) where each of the 10^{16} ground states is included with the same probability. Thus, the sample is thermodynamically correctly distributed. A summary of the method is given in Fig. 9.19.

A final problem may be that ground states having a specific color, e.g. yellow, have not been detected by the initial run of the genetic CEA method, although the system may have some (e.g. 10^4) yellow states. In this case they will never occur during the MC simulation, because there the color is preserved. For the actual criterion we use, it can be shown that the probability that a member of a specific group is found increases with the size of the group [88]. Furthermore, for the system sizes which are accessible the number of groups is small compared with the number of ground states usually obtained. Thus, only some small groups are missed, representing just a tiny fraction of all ground states. Consequently, when calculating physical properties, the error made is very small.

So far we have explained how the post processing works in general. Now it is time to be precise, the actual criterion we use is given. We start with a definition. Two ground states are said to be *neighbors* if they differ by the orientation of just one spin. Thus, the spin can be flipped without changing the energy, the spin is called *free*. All ground states connected to each other by this neighbor relation in a transitive way are said to be in one *valley*. Different valleys are separated by states having a higher energy. Therefore, one can travel in phase space within a valley by iteratively flipping free spins. The valleys are used as the groups mentioned above, i.e. for obtaining each ground state with the same probability. In the first step the configurations are sorted according to the valleys and then the valley sizes are estimated.

The MC simulation, which preserves the group identity, is very simple: iteratively a spin is chosen randomly. If the spin is free, it is flipped, otherwise the spin remains unchanged. This is an ordinary $T = 0$ MC simulation. Consequently, ergodicity and detailed balance are fulfilled as usual. One just has to ensure that the runs are of sufficient length. This length can be estimated by test simulations. One starts with a given ground state and performs say 20 different runs of length n_{MC} MC steps per spin resulting in 20 new ground states of the same valley. Then one compares the ground states by calculating the distribution $P(q)$ of overlaps, this is the quantity we are finally interested in. The behavior of $P(q)$ is observed as a function of the number

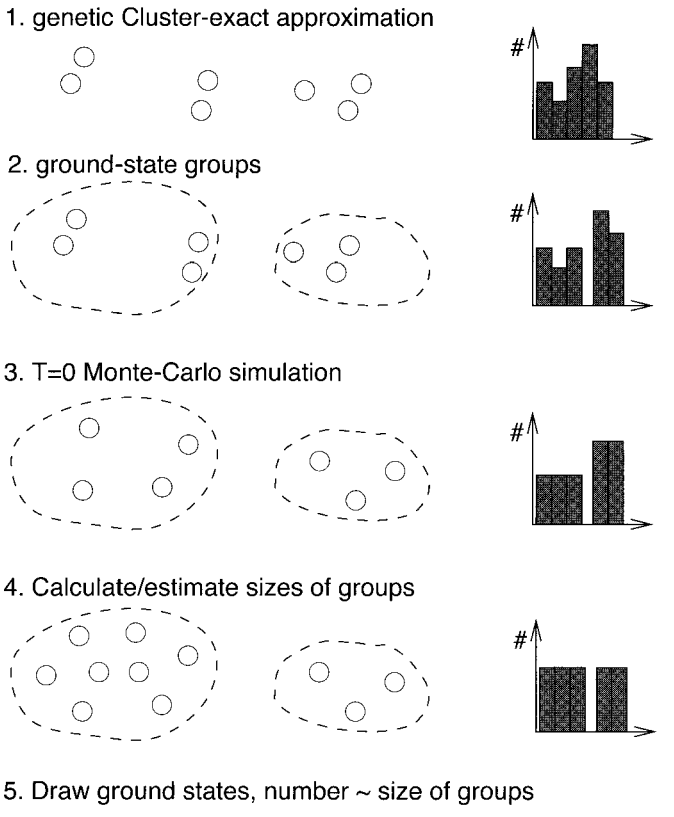


Figure 9.19: The basic idea of the post-processing step: all ground states are divided into groups. Then a $T = 0$ MC simulation is performed making all existing ground states within each group equiprobable, not only the ground states which have been found. Then the actual sizes of the group are estimated. Finally the states are drawn, from each group a number of states which is proportional to the size of the group.

n_{MC} of steps. If the shapes change no longer for even the largest valleys, the runs have a sufficient length. For three-dimensional systems $n_{\mathrm{MC}} = 100$ turned out to be sufficient for all system sizes.

More difficult to implement is the first step of the post processing, the division of the ground states into different valleys. If all ground states were available, the method for dividing the ground states would work as follows. The construction starts with one arbitrarily chosen ground state. All other states, which differ from this state by one free spin, are its neighbors. They are added to the valley. These neighbors are treated recursively in the same way: all their neighbors which are yet not included in the valley are added. After the construction of one valley is complete, the construction of the next one starts with a ground state, which has not been visited so far.

Unfortunately the number of ground states grows very quickly with system size, so even for small systems like $N = 8^3$ it is impossible to calculate all of them. Thus, the valley structure of the say 100 states which were obtained has to be established in a different way.

The basic tool is a *test*, which tells whether two ground states are in the same valley or not. The test is called *ballistic search* (BS), the reason for the choice of the name will become clear later on. Assume that it is known that some ground states belong to the same valley. Now we know that another state z belongs to the valley as well, if the test tells us that z is in the same valley as *any* of the states already treated. The main feature of the test is that it can be performed for states which are not direct neighbors in phase space. This is the reason why only a small subset of all ground states is needed.

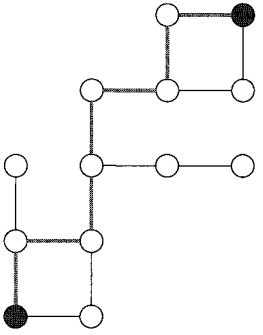


Figure 9.20: Ballistic search: A path in configuration-space which consists of flipping free spins is constructed between two ground states (dark nodes) belonging to the same valley. Depending on the order the spins are flipped the path may be found or not. Nodes represent ground states, edges represent flips of free spins. Please note that this figure is only a cartoon, the configuration space has N dimensions.

The test works as follows. Given two independent replicas $\{\sigma_i^\alpha\}$ and $\{\sigma_i^\beta\}$ let Δ be the set of spins, which are different in both states: $\Delta \equiv \{i|\sigma_i^\alpha \neq \sigma_i^\beta\}$. Now BS tries to build a path in configuration-space of successive flips of free spins, which leads from $\{\sigma_i^\alpha\}$ to $\{\sigma_i^\beta\}$. The path consists of states which differ only by flips of free spins from Δ (see Fig. 9.20). For the simplest version iteratively a free spin is selected randomly from Δ, flipped and removed from Δ, i.e. each spin is flipped at most once. Therefore, a straight path is built in phase space. This is the reason why the search for a path is called *ballistic*. This test does not guarantee to find a path between two ground states which belong to the same valley. It may depend on the order of selection of the spins whether a path is found or not, because not all free spins are independent of each other. Additionally, for very large valleys, when Δ percolates, it may be necessary to flip some spins not contained in Δ twice to allow other spins to flip in between. Thus, a path is found with a certain probability p_f, which decreases with the size of Δ. If the

size of Δ is small, p_f is very close to one. For the identification of the valley structure, it does not matter that $p_f < 1$. This is explained next.

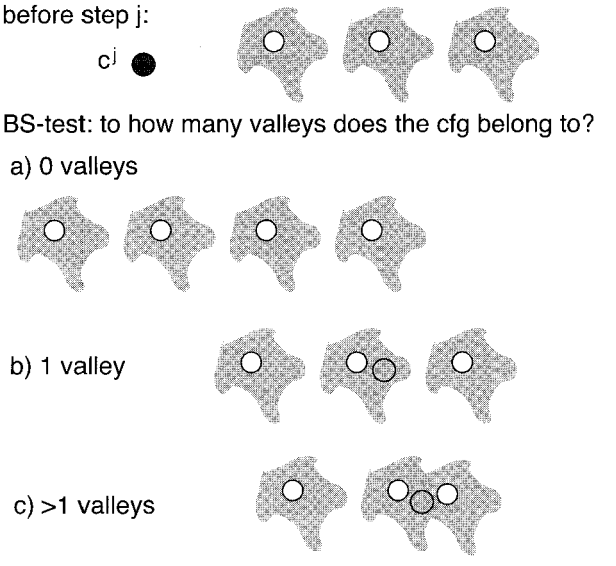


Figure 9.21: Algorithm for the identification of all valleys: several ground-states (circles) "cover" parts of valleys (filled areas). During the processing of all states a set of valleys is kept. When state c^j is treated, it is established using BS to how many of the already existing valleys the state belongs to. Three cases can occur: a) the ground state is found to belong to no valley, b) it is found in exactly one valley and c) it is found in several valleys. In the first case a new valley is found, in the second one nothing changes and in the third case several smaller valleys are identified as subsets of the same larger valley.

The algorithm for the identification of valleys using BS works as follows: the basic idea is to let a ground state represent that part of a valley which can be reached using BS with a high probability by starting at this ground state. If a valley is large it has to be represented by a collection of states, such that the whole valley is "covered". For example a typical valley of an $L = 8$ spin glass consisting of 10^{15} ground states is usually represented by only few ground states (e.g. two or three). A detailed analysis of how many representing ground states are needed as a function of valley and system size can be found in [91]. At each time the algorithm stores a set of m valleys $A = \{A(r)|r = 1, \ldots, m\}$ (r being the ID of the valley $A(r)$) each consisting of a set $A(r) = \{c^{rl}\}$ of representing configurations $c^{rl} = \{\sigma_i^{rl}\}$ ($l = 1, \ldots, |A(r)|$). At the beginning the valley set is empty. Iteratively all available ground states $c^j = \{\sigma_i^j\}$ ($j = 1, \ldots, D$) are treated: the BS algorithm tries to find paths from c^j or its inverse to all representing configurations in A. Let F be the set of valleys IDs, where a path

is found. Now three cases are possible (see Fig. 9.21):

- No path is found: $F = \emptyset$
  This means, configuration c^j does not belong to any of the valleys which have been found so far, as far as we can tell. Thus, a new valley is created, which is represented by the actual configuration treated: $A(m+1) \equiv \{c^j\}$. The valley is added to A: $A \equiv A \cup \{A(m+1)\}$.

- One or more paths are found to the representing configuration(s) of exactly one valley: $F = \{f_1\}$. Thus, the ground state c^j belongs to one valley. Valley f_1 seems to be already well represented. Consequently, nothing special happens, the set A remains unchanged.

- c^j is found to be in more than one valley: $F = \{f_1, \ldots, f_k\}$. Since a path is found from c^j to several states, they all belong in fact to the same valley. Thus, all these valleys are merged into one single valley, which is now represented by the union $\tilde{A}$ of the states, which are the representatives of the valleys in F:
  $\tilde{A} \equiv \bigcup_{j=1}^{k} A(f_j)$, $A \equiv \{\tilde{A}\} \cup A \setminus \bigcup_{j=1}^{k} \{A(f_j)\}$

This procedure is conducted for all available states. Please note that the merging mechanism ensures automatically that larger valleys are represented by more states than smaller valleys. It has been shown [91] that the number of states necessary to "cover" a valley grows only slowly with the valley size. Thus, systems exhibiting a large degeneracy can be treated.
The whole loop is performed twice. The reason is that a state which links two parts of a large valley (case 3) may appear in the sequence of ground states before states appear belonging to the second part of the valley. Consequently, this linking state is treated as being part of just one single smaller valley and both subvalleys are not recognized as one larger valley (see Fig. 9.22). During the second iteration the "linking" state is compared with all other representing states found in the first iteration, i.e. the large valley is identified correctly. With one iteration, the problem appears only if few ground-states per valley are available. Nevertheless, two iterations are always performed, so it is guaranteed that the difficulty does not occur.

Example: Valley identification using BS

Here an example is presented, how the algorithm operates. We want to establish the valley structure of 6 given ground states $c^1, \ldots, c^6$. Only the first execution of the loop is presented. Initially we start with an empty set of valleys. The way the set of valleys develops while the algorithm is running is shown in Fig. 9.23. The state c^1 belongs, like all states, surely to a valley, thus in a first step a valley with c^1 as the representative is created. Now assume that the BS test fails for c^2 and c^1. Consequently, a second valley is created. For c^3 a path is found in phase space to c^2, but not to c^1. Hence, we know that c^3 belongs to the second valley. The valley structure does not change in this step. Ground state c^3 is represented by an open circle in the figure. This means that it is not stored in the valley data structure. In the next step, a

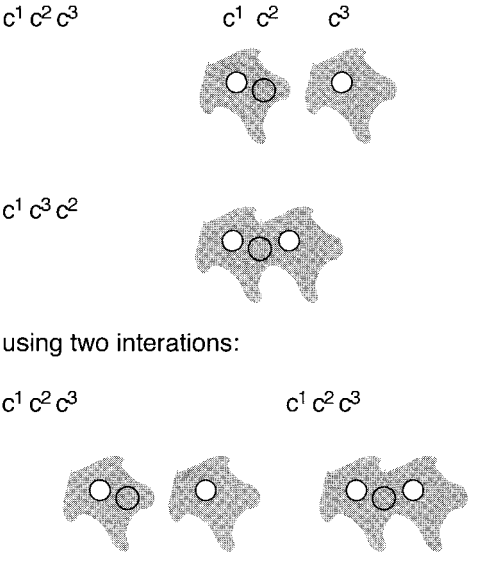


Figure 9.22: Example, that the order the states are treated may affect the result. Consider three states c^1, c^2, c^3, all belonging to the same valley. Assume that BS finds a path between (c^1, c^2) and (c^2, c^3), but not between (c^1, c^3). In the first case two valleys are found (false), in the second case one valley (correct). In order that the correct result is always obtained, two iterations are needed.

path is found from c^4 to c^1 and to c^2. Thus, all states encountered so far belong to the same valley. Both valleys are merged and now are represented by c^1 and c^2. For c^5 a path to c^2 but not to c^1 is found. Nevertheless, this means that c^5 belongs to the valley as well. Finally, for c^6 no path is found to either c^1 or c^2. Therefore, c^6 belongs to another valley, which is created in the last step. □

Since the probability p_f that a path is found between two ground states belonging to the same valley is smaller than one, how can we be sure that the valleys construction has been performed correctly? Hence, how can we avoid that a number of states belonging to one large valley are divided into two or several valleys, as a result of to some paths which have not been detected? If we only have a small number of states available, it is indeed very likely that the large valley is not identified correctly, see Fig. 9.24. If more states of the valley are available, it is more likely that all ground states are identified as being members of the same valley. The reason is that BS finds a path to neighbors which are close in phase space with a high probability. Now the probability that some path between two ground states is found increases due to the growing number of possible paths. It is always possible to generate additional ground

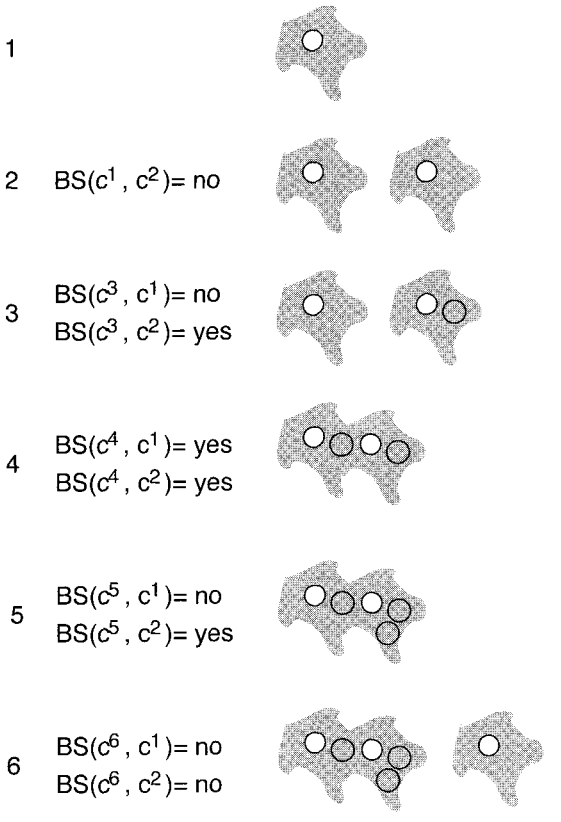


Figure 9.23: Example run of the algorithm for the identification of ground-state valleys; 6 ground states are processed. For details, see text.

states within each valley, so the probability that everything is done correctly can be increased arbitrarily close to one. It is very easy to obtain a 99.9% level of certainty, for details see [91].

So far we have explained the basic idea of the post-processing method and presented the BS algorithm which allows a number of ground states to be divided into valleys. Also the MC algorithm has been given which ensures that within each valley all ground states have the same probability of being selected. The final part which is missing is a technique that allows the size of a valley to be estimated. This allows each valley to be considered with its proper weight, which is proportional to its size.

A method similar to BS is used to estimate the sizes of the valleys. Starting from an arbitrary state $\{\sigma_i\}$ in a valley C, free spins are flipped iteratively, but each spin not more than once. During the iteration additional free spins may be generated and other spins may become fixed. When there are no more free spins left which have not been flipped already, the process stops. Thus, one has constructed a straight path in

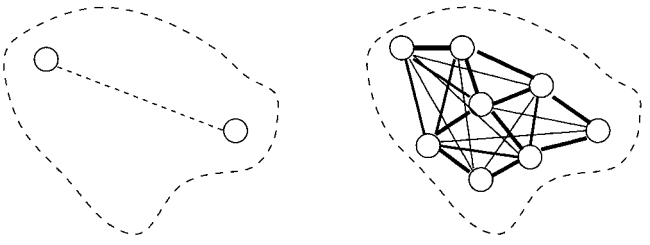


Figure 9.24: If two states, belonging to the same valley, are far apart in state space it is very unlikely that the BS algorithm detects that they belong to the same valley. If more states of the valley are available, the probability for the correct answer increases. The thickness of the lines indicates the probability that a path between two ground states is found by BS.

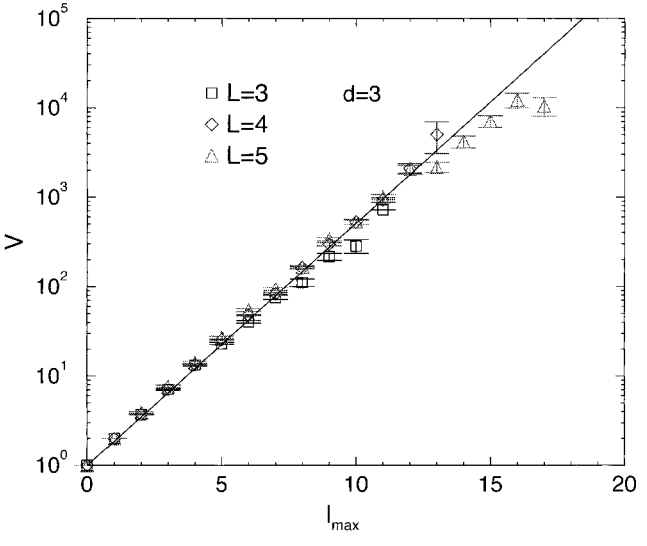


Figure 9.25: Average size V of a valley as a function of average dynamic number $l_{\max}$ of free spins (see text), here for three-dimensional $\pm J$ spin glasses. For the system sizes $L = 3, 4, 5$ all ground states have been obtained. A $V = 2^{0.9 l_{\max}}$ relationship is found, indicated by a straight line.

state space from the ground state to the border of the valley C. The number of spins that has been flipped is counted. By averaging over several trials and several ground states of a valley, one obtains an average value $l_{\max}$, which is a measure of the size of the valley.

For small ($d = 3$) system sizes $L = 3, 4, 5$ it is easy to obtain all ground states by

performing many runs of the genetic CEA algorithm. Thus, the valley sizes can be calculated exactly just by counting. Fig. 9.25 displays [91] the average size V of a valley as a function of $l_{\max}$. An exponential dependence is found, yielding

$$V = 2^{\alpha l_{\max}} \tag{9.11}$$

with $\alpha = 0.90(5)$. The deviation from the pure exponential behavior for the largest valleys of each system size should be a finite size effect. Similar measurements can be performed also for two- and four-dimensional systems.

Another method could be just to count the static number n_{f} of free spins. This is slightly simpler to implement, but it has turned out that the quantity $l_{\max}$ describes the size of a valley better than n_{f}. The reason is that by flipping spins additional free spins are created and deleted. Consider for example a one-dimensional chain of N ferromagnetically coupled spins with antiperiodic boundary conditions. Each ground state consists of two linear domains of spins. In each domain all spins have the same orientation. For each ground-state there are just two free spins, but all $2N$ ground states belong to the same valley. The possibility of similar but more complicated ground-state topologies is taken into account when using the quantity $l_{\max}$.

With relation (9.11) one can obtain for each valley an estimate of its size, even in the case when only a small number of ground states are available. Using these sizes one can draw ground states in such a way that each valley is represented with its proper weight. The selection is done in a manner such that many small valleys may contribute as a collection as well; e.g. assume that 100 states should be drawn from a valley consisting of 10^{10} ground states, then for a set of 500 valleys of size 10^7 each, a total number of 50 states is selected. This is achieved automatically by first sorting the valleys in ascending order. Then the generation of states starts with the smallest valley. For each valley the number of states generated is proportional to its size multiplied by a factor f. If the number of states grows too large, only a certain fraction f_2 of the states which have already been selected is kept, the factor is recalculated ($f \leftarrow f * f_2$) and the process continues with the next valley.

When again measuring the frequency of occurrence of each ground state for small systems, now after the post-processing step, indeed a flat distribution is obtained. Hence, now we have all the tools available to investigate the ground-state landscape of Ising spin glasses thermodynamically correctly. In the next section some results obtained with this methods are presented. We close this section by a summary of the post-processing method. Input is the realization $\{J_{ij}\}$ of the bonds and a set $\{c^k\}$ of ground states.

algorithm post-processing($\{J_{ij}\}, \{c^k\}$)
begin
  divide configuration $\{c^k\}$ into different valleys with BS;
  perform n_{RUN} runs of a $T = 0$ MC simulation
  calculate valley sizes;
  select sample of ground states according to valley sizes;
  return ground states;
end

9.5 Results

We finish this chapter by presenting some results [92] which were obtained with the combination of the genetic CEA and the method for ensuring the correct thermodynamic distribution. Since we are interested in the ground-state landscape, the distribution $P(q)$ of overlaps is a suitable quantity to study. You may remember from the first section the question of whether the mean-field scenario or the Droplet picture describes the behavior of realistic (i.e. three-dimensional) spin glasses correctly. By studying the finite-size behavior of $P(q)$, i.e. the width as a function of the system size, we will be able to show that the mean-field scenario does not hold for the ground states of three-dimensional spin glasses. But the behavior turns out to be more complex than predicted by a naive version of the Droplet theory.

For this purpose, ground states were generated using genetic CEA for sizes $L \in [3, \ldots, 14]$. The number of random realizations per lattice size ranged from 100 realizations for $L = 14$ up to 1000 realizations for $L = 3$.

Each run resulted in one configuration which was stored, if it exhibited the ground-state energy. For the smallest sizes $L = 3, 4$ all ground states were calculated for each realization by performing up to 10^4 runs. For larger sizes it is not possible to obtain all ground states, because of the exponentially rising degeneracy. For $L = 5, 6, 8$ in fact almost all valleys are obtained using at most 10^4 runs [91], only for about 25% of the $L = 8$ realizations may some small valleys have been missed.

For $L > 8$ not only the number of states but also the number of valleys grows too large, consequently only 40 independent runs were made for each realization. For $L = 14$ this resulted in an average of 13.8 states per realization having the lowest energy while for $L = 10$ on average 35.3 states were stored. This seems a rather small number. However, the probability that genetic CEA calculates a specific ground state increases (sublinearly) with the size of the valley the state belongs to [88]. Thus, ground states from small valleys do appear with a small probability. Because the behavior is dominated by the largest valleys, the results shown later on are the same (within error bars) as if all ground states were available. This was tested explicitly for 100 realizations of $L = 10$ by doubling the number of runs, i.e. increasing the number of valleys found.

Using this initial set of states for each realization ($L > 4$) a second set was produced using the techniques explained before, which ensures that each ground state enters the results with the same weight. The number of states was chosen in a way, that $n_{\max} = 100$ states were available for the largest valleys of each realization, i.e. a single valley smaller than one hundredth of the largest valley does not contribute to physical quantities, but, as explained before, a collection of many small valleys contributes to the results as well. Finally, it was verified that the results did not change by increasing $n_{\max}$.

The order parameter selected here for the description of the complex ground-state behavior of spin glasses is the total distribution $P(|q|)$ of overlaps. The result is shown in Fig. 9.26 for $L = 6, 10$. The distributions are dominated by a large peak for $q > 0.8$. Additionally there is a long tail down to $q = 0$, which means that arbitrarily different ground states are possible. Qualitatively the result is similar to the $P(|q|)$ obtained

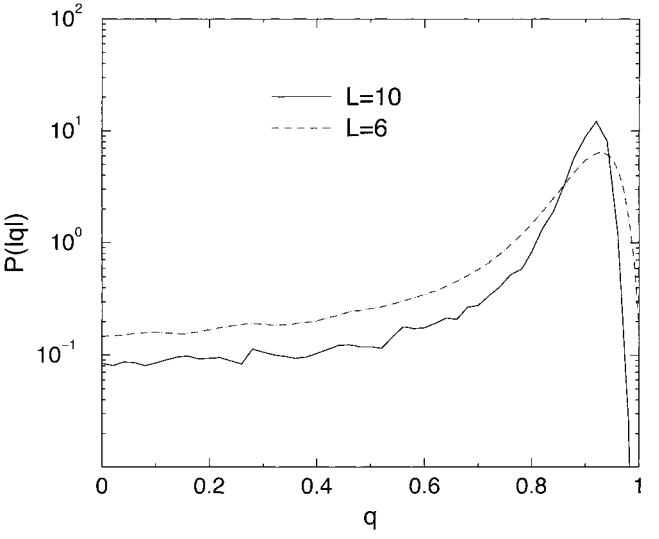


Figure 9.26: Distribution $P(|q|)$ of overlaps for $L = 6, 10$. Each ground state enters the result with the same probability. The fraction of small overlaps decreases by about a factor 0.6 by going from $L = 6$ to $L = 10$ (please note the logarithmic scale).

for the SK model for a small but nonzero temperature. But with increasing system size, the weight of the long tail decreases. To obtain a definite answer we have to extrapolate to very large system sizes.

To study the finite-size dependence of $P(|q|)$, the variance $\sigma^2(|q|)$ was evaluated as a function of the system size L. The values are displayed in Fig. 9.27. Additionally the datapoints are given which are obtained when the post-processing step is omitted [93, 94]. Obviously, by guaranteeing that every ground state has the same weight, the result changes dramatically. To extrapolate to $L \to \infty$, a fit of the data to $\sigma_L^2 = \sigma_\infty^2 + a_0 L^{-a_1}$ was performed. A slightly negative value of $\sigma_\infty^2 = -0.01(1)$ was obtained, indicating that the width of $P(|q|)$ is zero for the infinite system. For the plot a double-logarithmic scale was used. The fact that the datapoints are found to be more or less on a straight line is another indication that the width of $P(|q|)$ converges towards zero in the thermodynamic limit $L \to \infty$. Consequently, the MF picture with a continuous breaking of replica symmetry, which predicts a distribution of overlaps with finite width, cannot be true for the ground-state landscape of three-dimensional $\pm J$ spin glasses. Please note that only a small range of system sizes could be treated. Unfortunately, due to the NP-hardness of the ground-state calculation for spin glasses, larger sizes are not feasible at the moment.

By collecting all results, see also [92], one obtains the following description of the shape of $P(|q)$. It consists of a large delta-peak and a tail down to $q = 0$, but the weight of that tail goes to zero with lattice size going to infinity. This expression is

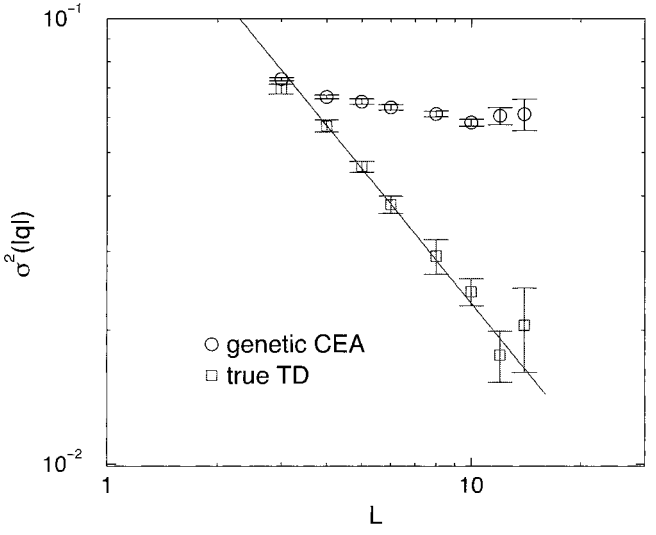


Figure 9.27: Variance $\sigma^2(|q|)$ of the distribution of overlaps as a function of linear system size L. The upper points show the case were each ground state enters with a weight determined by the genetic CEA algorithm. For the lower points each ground state has the same probability of being included in the calculation. The extrapolation to the infinite system results in a slightly negative value. Consequently, the width of distribution of overlaps appears to be zero, i.e. $P(|q|)$ is a delta-function. The line represents a fit to $\sigma_L^2 = a_0 L^{-a_1}$ resulting in $a_1 = 1.00(4)$.

used to point out that by going to larger sizes small overlaps still occur: the number of arbitrarily different ground states diverges [91]. But the size of the largest valleys, which determine the self overlap leading to the large peak, diverges even faster. The delta peak is centered around a finite value q_{EA}. From further evaluation of the results $q_{\mathrm{EA}} = 0.90(1)$ was obtained.

It should be stressed that this just tells us what the structure of the ground-state landscape looks like. To finally decide whether the mean-field like description or the Droplet picture describes the behavior of realistic spin glasses better, one has to study small but finite temperatures as well. This can be done by extending the methods described here to finite temperatures $T > 0$. Using the genetic CEA method excited states can be generated even faster than true ground states, smaller sizes of the populations and fewer minimizations steps are sufficient. The configurations obtained in this way can be divided into different valleys in the same way as has been done for the ground states. The only difference is that one has to take several different energy levels into account and one has to weight each excited state with its proper thermodynamic weight. Work in this direction is in progress.

Another approach to generate excited states has been presented recently [95] for the

three-dimensional model with Gaussian distribution. First a ground state is calculated. Please remember that it is unique, except for the state which is related to it by a global flip of all spins. Then two randomly chosen spins are forced to have different relative orientations to each other than in the ground state. These two spins remain unchanged for the following calculation. For the rest of the system again a ground state is obtained, resulting in an excited state for the total system, i.e. a Droplet of reversed spin is created. This demonstrates that ground-state methods are also useful for investigating the behavior at finite temperatures, similar approaches has been presented in Refs. [96, 97]. For the results obtained with these techniques, the behavior at finite temperatures turned out to be much richer than described by the simple Droplet picture, e.g. $P(q)$ seems not to be a delta function, see also Ref. [98]. But not all properties of the mean-field like description are found, so the debate goes on and much work still has to be done.

Bibliography

[1] K. Binder and A.P. Young, *Rev. Mod. Phys.* **58**, 801 (1986)

[2] M. Mezard, G. Parisi, and M.A. Virasoro, *Spin Glass Theory and Beyond*, (World Scientific, Singapore 1987)

[3] K.H. Fisher and J.A. Hertz, *Spin Glasses*, (Cambridge University Press, Cambridge 1991)

[4] J.A. Mydosh, *Spin Glasses: an Experimental Introduction*, (Taylor and Francis, London 1993)

[5] A.P. Young (ed.), *Spin Glasses and Random Fields*, (World Scientific, Singapore 1998)

[6] S.F. Edwards and P.W. Anderson, *J. de Phys.* **5**, 965 (1975)

[7] G. Toulouse, *Commun. Phys.* **2**, 115 (1977)

[8] Y. Cannella and J.A. Mydosh, Phys. Rev. B **6**, 4220 (1972)

[9] P. Norblad and P. Svendlidh, in: A.P. Young (ed.), *Spin Glasses and Random Fields*, (World Scientific, Singapore 1998)

[10] H. Rieger, in D. Stauffer (ed.): Annual Reviews of Computational Physics **II**, 295-341, World Scientific, 1995

[11] D. Sherrington and S. Kirkpatrick, *Phys. Rev. Lett.* **51**, 1791 (1975)

[12] R. Rammal, G. Toulouse, and M.A. Virasoro, *Rev. Mod. Phys.* **58**, 765 (1986)

[13] W.L. McMillan, *J. Phys. C* **17**, 3179 (1984)

[14] A.J. Bray and M.A. Moore, *J. Phys. C* **18**, L699 (1985)

[15] D.S. Fisher and D.A. Huse, *Phys. Rev. Lett* **56**, 1601 (1986)

[16] D.S. Fisher and D.A. Huse, *Phys. Rev. B* **38** , 386 (1988)

[17] A. Bovier and J. Fröhlich, *J. Stat. Phys.* **44**, 347 (1986)

[18] F. Barahona, *J. Phys. A* **15**, 3241 (1982)

[19] F. Barahona, R. Maynard, R. Rammal, and J.P. Uhry, *J. Phys. A* **15**, 673 (1982)

[20] J. Bendisch, *J. Stat. Phys.* **62**, 435 (1991)

[21] J. Bendisch, *J. Stat. Phys.* **67**, 1209 (1992)

[22] J. Bendisch, *Physica A* **202**, 48 (1994)

[23] J. Bendisch, *Physica A* **216**, 316 (1995)

[24] J. Kisker, L. Santen, M. Schreckenberg, and H. Rieger, *Phys. Rev. B* **53**, 6418 (1996)

[25] N. Kawashima and H. Rieger, *Europhys. Lett.* **39**, 85 (1997)

[26] J. Bendisch, *Physica A* **245**, 560 (1997)

[27] J. Bendisch and H. v. Trotha, *Physica A* **253**, 428 (1998)

[28] M. Achilles, J. Bendisch, and H. v. Trotha, *Physica A* **275**, 178 (2000)

[29] Y. Ozeki, *J. Phys. Soc. J.* **59**, 3531 (1990)

[30] T. Kadowaki, Y. Nonomura, H. Nishimori, in: M. Suzuki and N. Kawashima (ed.), *Hayashibara Forum '95. International Symposium on Coherent Approaches to Fluctuations*, (World Scientific, Singapore 1996)

[31] R.G. Palmer and J. Adler, *Int. J. Mod. Phys. C* **10**, 667 (1999)

[32] J.C. Angles d'Auriac, M. Preissmann, and A.S. Leibniz-Imag, *Math. Comp. Mod.* **26**, 1 (1997)

[33] H. Rieger, in J. Kertesz and I. Kondor (ed.): Advances in Computer Simulation, *Lecture Notes in Physics* **501**, Springer, Heidelberg, 1998

[34] A. Hartwig, F. Daske, and S. Kobe, *Comp. Phys. Commun.* **32** 133 (1984)

[35] T. Klotz and S. Kobe, *J. Phys. A* **27**, L95 (1994)

[36] T. Klotz and S. Kobe, *Act. Phys. Slov.* **44**, 347 (1994)

[37] T. Klotz-T and S. Kobe, in: M. Suzuki and N. Kawashima (ed.), *Hayashibara Forum '95. International Symposium on Coherent Approaches to Fluctuations*, (World Scientific, Singapore 1996)

[38] T. Klotz and S. Kobe, *J. Magn. Magn. Mat.* **17**, 1359 (1998)

[39] P. Stolorz, *Phys. Rev. B* **48**, 3085 (1993)

[40] E.E. Vogel, J. Cartes, S. Contreras, W. Lebrecht, and J. Villegas, *Phys. Rev. B* **49**, 6018 (1994)

[41] A.J. Ramirez-Pastor, F. Nieto, and E.E. Vogel, *Phys. Rev. B* **55**, 14323

[42] E.E. Vogel and J. Cartes, *J. Magn. Magn. Mat.* **17**, 777

[43] E.E. Vogel, S. Contreras, M.A. Osorio, J. Cartes, F. Nieto, and A.J. Ramirez-Pastor, *Phys. Rev. B* **58**, 8475 (1998)

[44] E.E. Vogel, S. Contreras, F. Nieto, and A.J. Ramirez-Pastor, *Physica A* **257**, 256 (1998)

[45] J.F. Valdes, J. Cartes, E.E. Vogel, S. Kobe, and T. Klotz, *Physica A* **257**, 557 (1998)

[46] C. De Simone, M. Diehl, M. Jünger, P. Mutzel, G. Reinelt, and G. Rinaldi, *J. Stat. Phys.* **80**, 487 (1995)

[47] C. De Simone, M. Diehl, M. Jünger, P. Mutzel, G. Reinelt, and G. Rinaldi, *J. Stat. Phys.* **84**, 1363 (1996)

[48] H. Rieger, L. Santen, U. Blasum-U, M. Diehl, M. Jünger, and G. Rinaldi, *J. Phys. A* **29**, 3939 (1996)

[49] M. Palassini and A.P. Young, *Phys. Rev. B* **60**, R9919 (1999)

[50] D. Stauffer and P.M. Castro-de-Oliveira, *Physica A* **215**, 407 (1995)

[51] P. Ocampo-Alfaro and Hong-Guo, *Phys. Rev. E* **53**, 1982 (1996)

[52] B. A. Berg and T. Celik, *Phys. Rev. Lett.* **69**, 2292 (1992)

[53] B.A. Berg and T. Celik, *Int. J. Mod. Phys. C* **3**, 1251 (1992)

[54] B. A. Berg, T. Celik, and U.H.E. Hansmann, *Europhys. Lett.* **22**, 63 (1993)

[55] T. Celik, *Nucl. Phys. B Proc. Suppl.* **30**, 908

[56] B.A. Berg, U.E. Hansmann, and T.Celik, *Phys. Rev. B* **50**, 16444 (1994)

[57] B.A. Berg, U.E. Hansmann, and T.Celik, *Nucl. Phys. B Suppl.* **42**, 905 (1995)

[58] H. Freund and P. Grassberger, *J. Phys. A* **22**, 4045 (1989)

[59] N. Kawashima and M. Suzuki, *J. Phys. A* **25**, 1055 (1992)

[60] D. Petters, *Int. J. Mod. Phys. C* **8**, 595 (1997)

[61] M. Yamashita, *J. Phys. Soc. Jap.* **64**, 4083 (1995)

[62] P. Sutton, D.L. Hunter, and N. Jan, *J. de Phys.* **4**, 1281 (1994)

[63] P. Sutton and S. Boyden, *Am. J. Phys.* **62**, 549 (1994)

[64] U. Gropengiesser, *Int. J. Mod. Phys. C* **6**, 307 (1995)

[65] U. Gropengiesser, *J. Stat. Phys.* **79**, 1005 (1995)

[66] D.A. Coley, *Cont. Phys.* **37**, 145 (1996)

[67] T. Wanschura, D.A. Coley, and S. Migowsky, *Solid State Commun.* **99**, 247 (1996)

[68] A. Pruegel-Bennett and S.L. Shapiro, *Physica D* **104**, 75 (1997)

[69] K. Chen, *Europhys. Lett.* **43**, 635 (1998)

[70] K.F. Pál, *Physica A* **223**, 283 (1996)

[71] K.F. Pál, *Physica A* **233**, 60 (1996)

[72] M. Palassini and A.P. Young, *Phys. Rev. Lett.* **83**, 5126 (1999)

[73] M. Palassini and A.P. Young, Proceedings of the Conference "Frontiers in Magnetism", Kyoto, October 4-7, 1999; *J. Phys. Soc. Jpn.* **69**, 165 (2000)

[74] M. Palassini and A.P. Young, *Phys. Rev. Lett.* **85**, 3333 (2000)

[75] J. Houdayer and O.C. Martin *Phys. Rev. Lett.* **83**, 1030 (1999)

[76] J. Houdayer and O.C. Martin, *Phys. Rev. Lett.* **82**, 4934 (1999)

[77] J. Houdayer and O.C. Martin, *Phys. Rev. Lett.* **84**, 1057 (2000)

[78] A.K. Hartmann, *Phys. Rev. E* **59**, 84 (1999)

[79] A.K. Hartmann, *Physica A* **224**, 480 (1996)

[80] K.F. Pál, *Biol. Cybern.* **73**, 335 (1995)

[81] A.K. Hartmann, *Eur. Phys. J. B* **8**, 619 (1999)

[82] A.K. Hartmann, *Phys. Rev. E* **60**, 5135 (1999)

[83] L.E. Reichl, *A Modern Course in Statistical Physics*, (John Wiley & Sons, New York 1998)

[84] A.J. Bray and M.A. Moore, *J. Phys. C* **17**, L463 (1984)

[85] W.L. McMillan, *Phys. Rev. B* **30**, 476 (1984)

[86] N. Kawashima and A.P. Young, *Phys. Rev. B* **53**, R484 (1996)

[87] E. Marinari, G. Parisi, and J.J. Ruiz-Lorenzo, in: A.P. Young (ed.), *Spin Glasses and Random Fields*, (World Scientific, Singapore 1998)

[88] A.K. Hartmann, *Physica A* **275**, 1 (1999)

[89] A. Sandvic, *Europhys. Lett.* **45**, 745 (1999)

[90] A.K. Hartmann, *Europhys. Lett.* **45**, 747 (1999)

[91] A.K. Hartmann, *J. Phys. A* **33**, 657 (2000)

[92] A.K. Hartmann, *Eur. Phys. J. B* **13**,591 (2000)

[93] A.K. Hartmann, *Europhys. Lett.* **40**, 429 (1997)

[94] A.K. Hartmann, *Europhys. Lett.* **44**, 249 (1998)

[95] F. Krzakala and O.C. Martin, *Phys. Rev. Lett.* **85**, 3013 (2000)

[96] M. Palassini and A.P. Young, *Phys Rev. Lett.* **85**, 3017 (2000)

[97] E. Marinari and G. Parisi, *Phys. Rev. Lett.* **86**, 3887 (2001)

[98] J. Houdayer and O.C. Martin, *Euro. Phys. Lett.* **49**, 794 (2000)

10 Matchings

After introducing spin glasses and discussing general approximation algorithms for ground states in Chap. 9, we now turn to two-dimensional systems. We will first show how ground states of certain two-dimensional spin glasses can be calculated by mapping the problem onto a matching problem. Next, a general introduction to matching problems is given. In the third section, the foundation of all matching algorithms, the augmenting path theorem, in presented. In the central section, algorithms for different types of matching problems are explained. Finally, an overview of some results for spin glasses is given.

10.1 Matching and Spin Glasses

The problem of determining the ground state of a two-dimensional spin-glass model on a square lattice with nearest neighbor interactions with free boundaries[1] can be mapped on to a matching problem on a general graph [1, 2, 3] as follows: Associate with each unsatisfied bond an "energy string" joining the centers of neighboring plaquettes sharing this bond, and assign to each energy string a "length" equal to $|J_{ij}|$. Clearly the energy E of the system equals the total length Λ of these energy lines, up to a constant:

$$E = -\sum_{ij} |J_{ij}| + \Lambda. \tag{10.1}$$

Let us for example put all spins pointing upwards, so that each negative bond will be unsatisfied and thus cut by an energy string [Fig. 10.1 (a)]. Unfrustrated plaquettes have by definition an even number of strings crossing their boundary, therefore energy lines always enter and leave unfrustrated plaquettes. Frustrated plaquettes on the other hand have an odd number of strings, and therefore one string must begin or end in each frustrated plaquette. These observations hold for any spin configuration.
If boundary conditions are open or fixed, some of the energy strings can end at the boundary. To unify the case of (partly) periodic/non-periodic boundary conditions, it is possible to introduce "external" plaquettes for each side with open boundary conditions. In that case plaquettes always occur in pairs. One says the plaquettes belonging to a pair are *matched*.

[1] Or with periodic boundary conditions in one direction and free boundary conditions in the other. The general rule is that the graph must be planar, i.e. it is possible to draw the graph in a plane, without crossing edges.

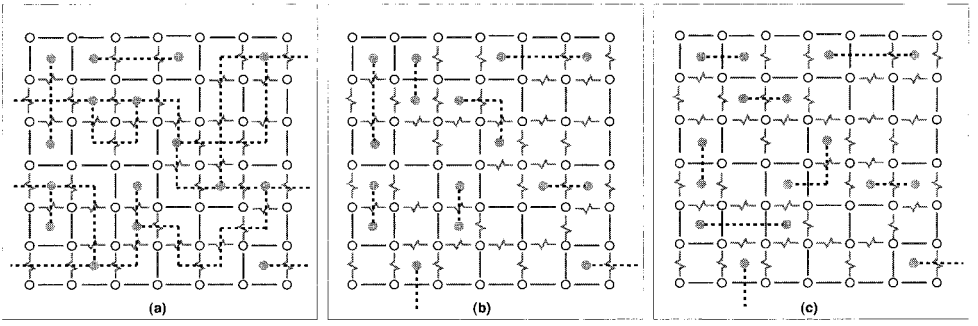


Figure 10.1: Thick lines represent negative couplings. Energy strings (dotted) are drawn perpendicular to each unsatisfied coupling. Frustrated plaquettes (odd number of negative couplings) are marked by a dot. **a)** All spins up configuration. **b)** A ground state. **c)** Another ground state.

From Eq. (10.1), finding a ground state, i.e. a state of minimum energy, is equivalent to finding a minimum length *perfect matching*, as we define it below, in the graph of frustrated plaquettes [1, 2, 4]. For the definition of frustration, see Chap. 9. If $|J_{ij}| = J$, i.e. all interactions have the same strength, Fig. 10.1 (b) shows one possible ground state. An equivalent ground state is obtained by flipping all spins inside the gray area in Fig. 10.1 (c), since the numbers of satisfied and unsatisfied bonds along its contour are equal. Degenerate ground states are related to each other by the flipping of irregularly shaped clusters, which have an equal number of satisfied and unsatisfied bonds on their boundary.

If the amplitudes of the interactions are also random, this large degeneracy of the ground state will in general be lost, but it is easy to see that there will be a large number of low-lying excited states, i.e. spin configurations which differ from the ground state in the flipping of a cluster such that the "length" of unsatisfied bonds on its boundary almost cancels the length of satisfied ones.

Please note, in the case of a two-dimensional spin glass with periodic boundary conditions in all directions, or in the presence of an external field, the ground-state problem becomes NP-hard.

10.2 Definition of the General Matching Problem

Given a graph $G(V, E)$ with vertex (or node) set V and edge set E, a matching $M \subset E$ is a subset of edges, such that no two are incident to the same vertex [5, 6]. A matching M of maximum cardinality is called *maximum-cardinality matching*. An edge contained in a given matching is called *matched*, other edges are *free*. A vertex incident to an edge $e \in M$ is *covered* (or matched) others are *M-exposed* (or exposed or free). A matching is *perfect* if it leaves no exposed vertices. If $e = (u, v)$

is matched, then u and v are called *mates*. An *alternating path* is a path along which the edges are alternately matched and unmatched. A *bipartite graph* is a graph which can be subdivided into two sets of vertices, say X and Y, such that the edges in the graph (i,j) only connect vertices in X to vertices in Y, *with no edges* internal to X or Y. Nearest-neighbor hypercubic lattices are bipartite, while the triangular and face-centered-cubic lattices are *not bipartite*.

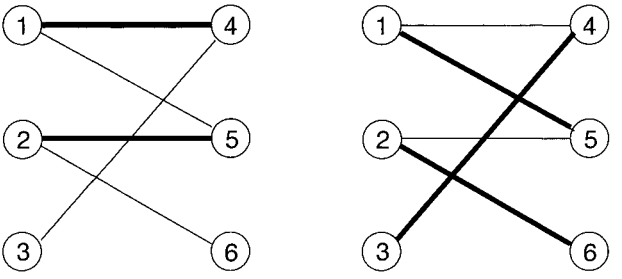


Figure 10.2: Example graph for matching, see text.

Example: Matching

In Fig. 10.2 a sample graph is shown. Please note that the graph is bipartite with vertex sets $X = \{1,2,3\}$ and $Y = \{4,5,6\}$. Matched vertices are indicated by thick lines. The matching shown in the left half is $M = \{(1,4),(2,5)\}$. This means, e.g. edge $(1,4)$ is matched, while edge $(3,4)$ if free. Vertices 1,2,4 and 5 are covered, while vertices 3 and 6 are exposed.

In the right half of the figure, a perfect matching $M = \{(1,5),(2,6),(3,4)\}$ is shown, i.e., there are no exposed vertices.

An example of an alternating path is shown in the left part of Fig. 10.3. □

The more general *weighted-matching* problems assign a non-negative weight (=cost), c_{ij}, to each edge $e = (i,j)$. M is a *maximum-weight matching* if the total weight of the edges in M is maximal with respect to all possible matchings. For perfect matchings, there is a simple mapping between maximum-weight matchings and minimum-weight matchings, namely: let $\tilde{c}_{ij} = C_{\max} - c_{ij}$, where c_{ij} is the weight of edge (i,j) and $C_{\max} > \max_{(i,j)}(c_{ij})$. A maximum perfect matching on $\tilde{c}_{ij}$ is then a minimum perfect matching on c_{ij}.

A nice historical introduction to matching problems, the origins of which may be traced to the beginnings of combinatorics, may be found in Lovász and Plummer [6]. Matching is also related to thermal statistical mechanics because the partition function for the two-dimensional Ising model on the square lattice can be found by counting *dimer coverings* (=perfect matchings) [6]. This is a graph *enumeration*

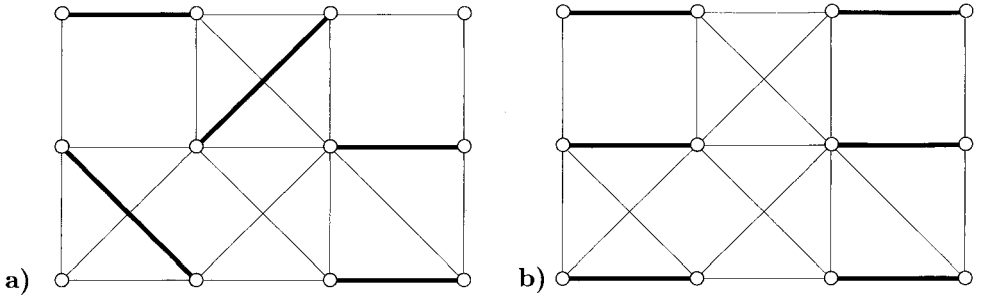


Figure 10.3: a) A matching (thick) is a subset of edges with no common end. An augmenting path (shaded) starts and ends at exposed nodes, and alternates between unmatched (thin) and matched (thick) edges. **b)** Interchanging matched and unmatched edges along an augmenting path increases the cardinality of the matching by one. This is called *augmentation* and is the basic tool for maximum-matching problems.

problem rather than the *optimization* problems we consider here. As a general rule, graph enumeration problems are *harder* than graph-optimization problems.
Owing to the fact that all cycles on bipartite graphs have an even number of edges, matching on bipartite graphs is considerably easier than matching on general graphs. In addition, maximum-cardinality matching and maximum-weight matching on bipartite graphs can be easily related to the maximum-flow and minimum-cost-flow (respectively) problems discussed in Chaps. 6 and 7, respectively. Matching on general graphs and maximum/minimum perfect matching are more complicated. Thus, after presenting a fundamental theorem in the next section, in Sec. 10.4 matching algorithms for different types of problems are explained.

10.3 Augmenting Paths

The algorithms for maximum matchings are based on the idea of successive augmentation, which is analogous to the augmenting-path methods for flow problems (see Sec. 6.3). An augmenting path A_p with respect to M is an alternating path between two exposed nodes. An augmenting path is necessarily of odd length, and, if G is bipartite, connects a node in one sublattice, Y, with a node in the other sublattice, X. Clearly, if matched and free edges are interchanged along A_p, the number of matched edges increases by one. Therefore if M admits an augmenting path it cannot be of maximum cardinality. In the case of weighted maximum matching, for each alternating path, one calculates the weight of the path by adding the weights of all unmatched edges and subtracting the weights of all matched edges. Then, a matching M cannot be of maximum weight if it has an alternating path of positive weight, since interchanging matched and free edges would produce a "heavier" matching.
The non-existence of augmenting paths is a necessary condition for maximality of a

matching. It is also a sufficient condition. A central result in matching theory states that repeated augmentation must result in a maximum matching [7, 8].

Theorem: (Augmenting path)
(i) A matching M has maximum cardinality *if and only if* it admits no augmenting path.
(ii) A matching M has maximum weight *if and only if* it has no alternating path or cycle of positive weight.

Proof: *(i)* $\Rightarrow$ is trivial. To prove $\Leftarrow$, assume M is not maximum. Then some matching M' must exist with $|M'| > |M|$. Consider now the graph G' whose edge set is $E' = M \Delta M'$, (the symmetric difference of M and M', $M \Delta M' \equiv (M \setminus M') \cup (M' \setminus M)$). Clearly each node of G' is incident to at most one edge of M and at most one edge of M'. Therefore nodes in G' have at most two incident edges and the connected components must be either simple paths or cycles of even length, and all paths are alternating paths. In all cycles we have the same number of edges from M as from M' so we can forget them. But since $|M'| > |M|$ there must be at least one path in G' with more edges from M' than from M. This path must necessarily be an augmenting path.
(ii) $\Rightarrow$ is again trivial. Assume M is not maximum. Some matching M' must therefore exist with $c(M') > c(M)$. Consider again $G' = (X, M \Delta M')$. By the same reasoning as before, we conclude that G' must contain at least one alternating path or cycle of positive weight. QED

10.4 Matching Algorithms

10.4.1 Maximum-cardinality Matching on Bipartite Graphs

Consider a bipartite graph, $B(X, Y, E)$, where X is the set of nodes on one sublattice and Y is the set of nodes on the other sublattice. It is conventional to draw bipartite graphs as shown in Fig. 10.4, with the two sublattices joined by edges, which can only go from one sublattice to the other. Now assume that an initial matching M is given (which can be the empty set), as in Fig. 10.4a. It is natural to look for alternating paths starting from exposed nodes. (If there are no exposed nodes, M is maximum. Stop.) An efficient way to do this is to consider all alternating paths from a given exposed node simultaneously in the following way.
Build a breadth-first-search (BFS) tree (see Sec. 4.1.2) starting from an exposed node, for example node v_5, as described in Fig. 10.5a. In the BFS tree, which we call in the following an *alternating tree* $\mathcal{T}$, node v_5 corresponds to level 0. All its adjacent edges are free. They lead to nodes u_3 and u_5 at level 1, which are covered. Now since we must build alternating paths, it does not make sense to continue the search along free edges. Therefore we proceed along matched edges, respectively to nodes v_1 and v_2. From these we follow free edges to u_1 and u_2, and then matched edges to nodes v_3 and v_4. In the last step, node u_4 is found exposed. Therefore $(u_4, v_4, u_2, v_1, u_3, v_5)$ is an

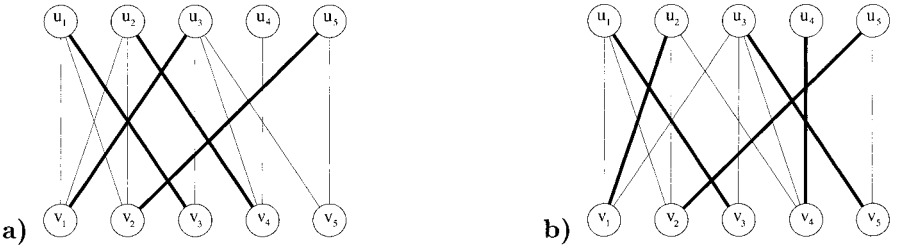


Figure 10.4: a) An initial matching is given for a bipartite graph. **b)** The enlarged matching obtained after inverting the augmenting path discovered from node v_5 (see Fig. 10.5).

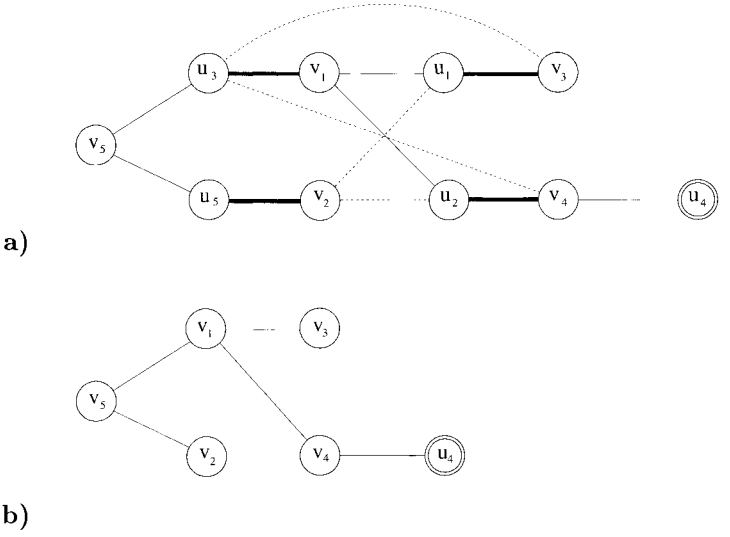


Figure 10.5: a) The BFS or alternating tree built from exposed node v_5 in Fig. 10.4a. Dashed lines represent non-tree edges to already visited nodes. The search finishes when an exposed node u_4 (double circle) is found. **b)** The auxiliary tree obtained after removing odd-level nodes and identifying them with their mates. After inverting the augmenting path $\{v_5, u_3, v_1, u_2, v_4, u_4\}$, the enlarged matching in Fig. 10.4b is obtained.

augmenting path. After inverting it, the augmented matching shown in Fig. 10.4b is obtained. If no exposed node were found when the BFS ends, then node v_5 will never be matched and can be forgotten because of the following result [9], which is valid for general graphs:

Theorem:
If there is no augmenting path from node u_0 at some stage, then there never will be an augmenting path from u_o.

Since the BFS or alternating tree is the basic structure maintained by matching algorithms, we introduce a convenient notation. In what follows we denote the odd-level set of nodes of the alternating tree $\mathcal{T}$ by $A(\mathcal{T})$ and the even-level set $B(\mathcal{T})$. These sets, beginning with $A = \emptyset$, $B = \{r\}$, can be built up, by iteratively calling the following procedure, we denote $V(\mathcal{T}) \equiv A(\mathcal{T}) \cup B(\mathcal{T})$:

procedure extend-alternating-tree($\mathcal{T}$, (i,j), M)
begin
  if $i \in B(\mathcal{T})$, $j \notin V(\mathcal{T})$, $(j,k) \in M$, **then**
    add j to $A(\mathcal{T})$, k to $B(\mathcal{T})$;
end

The set $V(\mathcal{T})$ and the edges used in its construction have the structure indicated in Fig. 10.5a, without the broken lines and without the node u_4, which is found to be exposed. Note that an alternating tree always ends in B-nodes and once an edge in G has been found having one end in $B(\mathcal{T})$ and the other end *not* in $A(\mathcal{T})$, we find an augmenting path.
Using alternating trees, the augmentation of a matching by exchanging matched and unmatched vertices along an augmenting path, can be written in the following way, this procedure will be used later on as well [we denote the edges of a path with $E(P)$]:

procedure Augment($\mathcal{T}$, (i,j), M)
begin
  Let r the root of $\mathcal{T}$;
  Let P be the path in $\mathcal{T}$ from r to i plus (i,j);
  replace M by $M \Delta E(P)$;
end

For the algorithms, which we present later on, the sets $A(\mathcal{T})$ and $B(\mathcal{T})$ are very useful. For the maximum cardinality bipartite matching, there is a difference between the search technique actually applied and the usual BFS since the searches from odd-numbered levels are trivial. They always lead to the mate of that node, if the node itself is not exposed. Therefore the search can in practice be simplified by ignoring odd-numbered nodes and going directly to their mates, as shown in Fig. 10.5b. The search for alternating paths can in fact be seen as a usual BFS on an auxiliary graph from which odd-level nodes have been removed by identifying them with their mates. The basic augmenting path algorithm is then as follows [10, 11].

algorithm Maximum-cardinality bipartite matching
begin
establish an initial feasible matching M on $B(X, Y, E)$;
while B contains an exposed node $u \in Y$ **do**
begin
Initialize alternating Tree $\mathcal{T} = (\{u\}, \emptyset)$;
while there are edges (i, j), $i \in B(\mathcal{T})$, $j \notin V(\mathcal{T})$ **do**
extend alternating tree $(\mathcal{T}, (j, k), M)$;
if $\exists j, k : k \notin V(\mathcal{T})$, k exposed, $i \in B(\mathcal{T})$ **then**
Augment$(\mathcal{T}, (i, j), M)$;
else
no bipartite matching exists;
end
end

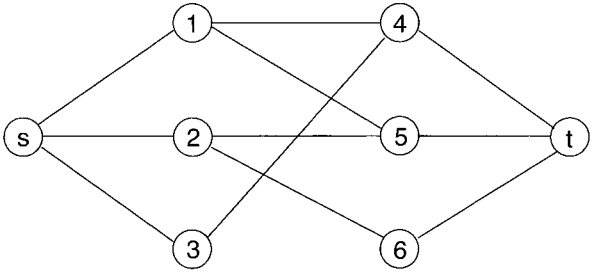


Figure 10.6: The resulting network, after adding vertices s and t to the graph from Fig. 10.2 and connecting all vertices from X with s and all vertices from Y with t.

Please note that the initial feasible matching can be empty $M = \emptyset$. The best known implementation for this algorithm is due to Hopcroft and Karp [12]. It runs in time $O(|E| \cdot \sqrt{|X|})$ and is based on doing more than one augmentation in one step. There is also a simple way in which to map the maximum-cardinality-matching problem to the maximum-flow problem, see Chap. 6. Let $B = (X, Y, E)$, and define B' by adding a source node s and a target node t, and connecting all nodes in X to s, and all nodes in Y to t, by edges of capacity 1, see e.g. Fig. 10.6. Now let all edges $e \in E$ have capacity 1. Because of the integer flow theorem, maximum flows in B' are integral. Every flow of size f thus identifies f matched edges in B, and vice versa. Since maximum flows in $0-1$ networks are computable in $\mathcal{O}(|E| \cdot \sqrt{|X|})$ time, so are maximum-cardinality matchings on B.
The mapping of matching problems to flow problems also applies to the maximum-weight-matching problem on bipartite graphs:
Given an edge-weighted bipartite graph, find a matching for which the sum of the weights of the edges is maximum.

This is also known as the *assignment problem*, because it can be identified with optimal assignment, e.g. of workers to machines, if worker $i \in Y$ produces a value c_{ij} working at machine $j \in X$:
Given an $n \times n$ matrix, find a subset of elements in the matrix, exactly one element in each column and one in each row, such that the sum of the chosen elements is maximal.
In a similar manner to that described in the previous paragraph, this problem can easily be formulated as a flow problem. Again add a source node s, a sink node t; connecting s to all nodes in U by unit-capacity, zero-cost edges; all nodes in V to t by unit-capacity, zero-cost edges. Also interpret edge e_{ij} as a directed edge with unit capacity and cost $\tilde{c}(e_{ij}) = C_{\max} - c_{ij}$, where $C_{\max} > \max_{(i,j)}(c_{ij})$. The solution to the minimum-cost-flow problem (see Chap. 7) from s to t is then equivalent to the maximum-weight matching which we seek. Please note that in this case a mapping on the maximum-flow problem is not possible, because otherwise it is not possible to take the edge weights into account *and* guarantee that each worker works exactly at one machine. The *minimum* weight perfect bipartite matching problem can be solved in the same way, for this case the weights c_{ij} can be used directly without inversion: $\tilde{c}(e_{ij}) = c_{ij}$.

10.4.2 Minimum-weight Perfect Bipartite Matching

Obviously a maximum weight matching does not need to be of maximum cardinality. For our application to two-dimensional spin glasses we need an algorithm to solve a *minimum-weight perfect matching* problem, actually on a general graph. Although, we have already solved this problem for bipartite graphs, by mapping it to a minimum-cost flow problem, it is useful to demonstrate the basic idea for bipartite graphs first. In the following, we introduce a method that is called the *Hungarian algorithm* in recognition of the mathematician Egervary that uses the linear program formulation of the minimum weighted matching problem.
Let us characterize a matching M by a vector $x \in \{0,1\}^E$ (i.e. $c_{ij} \in \{0,1\}\, \forall (i,j) \in E$), where $x_{ij} = 1$ if the edge (i,j) is in the matching, and $x_{ij} = 0$ otherwise. In order to form a perfect matching on a bipartite graph $G = (X, Y, E)$ the following conditions have to be fulfilled:

$$\begin{array}{ll} \forall i \in V = X \cup Y : & \sum_{j \in V} x_{ij} = 1 \\ \forall (i,j) \in E : & x_{ij} \geq 0 \end{array} \tag{10.2}$$

Any vector x which fullfills this conditions is called *feasible*. The optimization task to find an minimum perfect matching, can be written as a *linear program* (LP)

$$\begin{array}{ll} \text{minimize} & \sum_{(i,j) \in E} c_{ij} x_{ij} \\ \text{subject to} & (10.2) \end{array} \tag{10.3}$$

Linear programming is the problem of optimizing (minimizing or maximizing) a linear *cost* function while satisfying a set of linear equality and/or inequality constraints. The full subject of linear optimization is beyond the scope of this work, but there are

many books devoted to it (c.f. Ref. [10]) and its applications. Most of the problems described in this book can be cast as linear or, more generally, convex-programming problems. The *simplex method*, published by Dantzig in 1949, solves a standard LP

$$\begin{aligned} &\text{minimize} && c^t z \\ &\text{subject to} && \mathbf{B}z = b \quad \text{and} \quad z_i \geq 0\,, \end{aligned} \tag{10.4}$$

where c, z are real vectors with n components, b is an m-component vector and $\mathbf{B}$ is a $n \times m$ matrix. The simplex method runs *typically* in polynomial time[2]. Please note that also inequality contraints, e.g. of the form $\mathbf{B}x \geq b$, can be treated by introducing additional variables.

For the bipartite matching problem we need to have the variables x_{ij} as integers, which is guaranteed by the equivalence to a minimum-cost-flow-problem, as shown in the last section, since this always has an integer optimum. This can be seen in another way as well. It can be shown that the optimal solution of (10.4) always lies on the corners of a convex polytope. In $\mathbb{R}^E$ the polytope defined by the inequalities (10.2) has only integer corners, which is no longer true in general (non-bipartite) graphs, see Sec. 10.4.4.

For the discussion of weighted matchings it is not necessary to enter into the details of a linear programming. The Hungarian method solves the problem directly. But to understand the technique, we need the concept of *duality*: The LP (10.2) is called the *primal* linear problem, to which a *dual* linear problem belongs [10]. For each primal constraint, a dual variable y_i $(i = 1, \ldots, n)$ is introduced, the minimization translates to a maximimzation, and the vectors c, b switch its roles:

$$\begin{aligned} &\text{maximize} && b^t y \\ &\text{subject to} && \mathbf{B}^t y \leq c. \end{aligned} \tag{10.5}$$

Please note that the sign of the variables y_i is not restricted and that the dual of the dual is the primal. For the matching problem (10.3) we get

$$\begin{aligned} &\text{maximize} && \textstyle\sum_{i \in Y} y_i + \sum_{j \in X} y_j \\ &\text{subject to} && y_i + y_j \leq c_{ij}\,. \end{aligned} \tag{10.6}$$

We will now give a little background information, for a comprehensive presentation we refer to the literature. In LP theory, it can be shown that the optimum values $c^t z$ and $b^t y$ agree. The idea behind duality is, that when solving a primal LP, the algorithms always keeps feasible solutions, while stepwise decreasing the value of $c^t z$, thus keeping an *upper* bound. On the other hand, when solving the dual problem, which is a maximization problem, always a lower bound $b^t y$ is kept. Thus, one can treat a primal and a dual problem in parallel by iteratively increasing the lower bound and decreasing the upper bound. The solution is found when upper and lower bounds agree.

[2]The simplex technique does not guarantee to run in polynomial time. There is a polynomial algorithm to solve (LP), the *ellipsoid method*. But in practice, the simplex methods is usaully faster, so it is used.

Another important result, which we need for the Hungarian algorithm, is that the following *orthogonality conditions* are necessary and sufficient for optimality of the primal and dual solutions, here directly written in the form suitable for our application ($\forall i \in Y$ and $\forall j \in X$):

$$(\forall i \in Y, j \in X): \quad x_{ij} > 0 \Rightarrow y_i + y_j = c_{ij}. \tag{10.7}$$

In Fig. 10.7, a bipartite graph, a minimum-weight perfect matching and the corresponding dual solution are shown. Please note that conditions $y_i + y_j = c_{ij}$ are only necessary for matched edges, not sufficient.

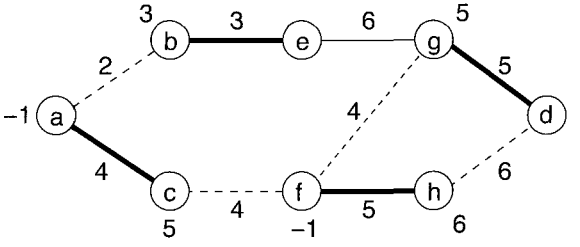


Figure 10.7: A bipartite graph. The numbers next to the edges are the weights c_{ij}. The minimum-weight perfect matching is indicated by bold lines. The numbers next to the vertices are the values y_i of the dual solution. Zero values are not shown. Edges not belonging to the matching, but having $y_i + y_j = c_{ij}$, are indicated by broken lines. The weight of the minimum perfect matching (=17) equals the value $\sum y_i$ of the dual solution.

Next, we introduce a convenient notation. Given a vector $y \in \mathbb{R}^E$ and an edge $(i,j) \in E$, we denote by $\overline{c}_{ij} = \overline{c}_{ij}(y)$ the difference $c_{ij} - (y_i + y_j)$. Thus, y is feasible in (10.6) if and only if $\overline{c}_{ij} \geq 0$ for all $(i,j) \in E$. In this case we denote by $E_=$ [or $E_=(y)$] the set

$$E_= \{(i,j) \in E | \overline{c}_{ij} = 0\}; \tag{10.8}$$

its elements are the equality edges with respect to y.
If x is the characteristic vector of a perfect matching M of G, the optimality conditions are equivalent to

$$M \subseteq E_= \tag{10.9}$$

We will now explain the basic idea of the Hungarian algorithm. Given a feasible solution y to (10.6), we can now use the maximum cardinality bipartite matching algorithm described above to search for a perfect matching having this property. If we succeed, we have a perfect matching whose characteristic vector is optimal to (10.3), as required. Otherwise the algorithm will deliver a matching M of $G_= = (V, E_=)$ and an M-alternating (BFS) tree $\mathcal{T}$ such that its B-nodes are joined by equality edges only to A-nodes (in Fig. 10.5 the A-nodes are those indicated by u_i and the B-nodes by v_i). See the example in Fig. 10.8, where a perfect matching exists, but with respect to the current y (indicated at the nodes), it is not completly contained in $E_=$.

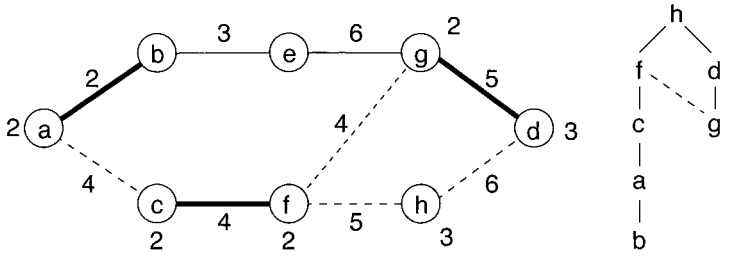


Figure 10.8: The graph of Fig. 10.7 with a preliminary matching. Vertices e and h are exposed, h is the current root of the alternating tree, which is shown on the right. By increasing the values at the B-nodes b, c, h and g by $\epsilon = 3$ and decreasing it by ϵ at the A-nodes a, f and i, the edge (b, e) joins $E_=$ and $\sum y_i$ attains the maximum value. The final result of Fig. 10.7 is obtained, by inverting the alternating path h, f, c, a, b, e.

In that case there is a natural way to change y, keeping in mind that we would like edges of M and $\mathcal{T}$ to remain in $E_=$, and that we would like $\bar{c}_{ij}$ to decrease for edges (i, j) joining nodes in $B(\mathcal{T})$ to nodes not in $A(\mathcal{T})$. Namely, we increase y_i by $\varepsilon > 0$ for all $v \in B(\mathcal{T})$, and we decrease y_i by ε for $i \in A(\mathcal{T})$. This has the desired effect; we will choose ε as large as possible so that feasibility in (10.6) is not lost and as a result (provided that G has a perfect matching at all), some edge joining a node $u \in B(\mathcal{T})$ to a node $w \notin A(\mathcal{T})$ will enter $E_=$. Since G is bipartite, we will have $w \notin V(\mathcal{T})$, leading to an augmentation or a tree-extension step. In the example Fig. 10.8 we will choose $\varepsilon = 3$ because of the edge (b, e), and then that edge enters $E_=$, allowing for an augmenting path.

To summarize, we get the following so-called *Hungarian algorithm*, due to Kuhn [13] and Munkres [14]:

algorithm Minimum-Weight Perfect Matching for Bipartite Graphs
begin
Let y be a feasible solution to (10.6), M a matching of $G_{=}(y)$;
Set $\mathcal{T} := (\{r\}, \emptyset)$, where r is an M-exposed node of G;
while 1=1
begin
while there exists $(i,j) \in E_{=}$ with $i \in B(\mathcal{T})$, $j \notin V(\mathcal{T})$
if j is M-exposed **then**
begin
Augment($\mathcal{T}$, (i,j), M);
if there is no M-exposed node in G **then**
return the perfect matching M;
else
replace $\mathcal{T}$ by $(\{r\}, \emptyset)$, where r is M-exposed;
end
else
extend-alternating-tree($\mathcal{T}$, (i,j), M);
if every $(i,j) \in E$ with $i \in B(\mathcal{T})$ has $j \in A(\mathcal{T})$ **then**
return G has no perfect matching!;
else
begin
let $\varepsilon := \min\{\bar{c}_{ij} | (i,j) \in E,\ i \in B(\mathcal{T}),\ j \notin V(\mathcal{T})\}$;
replace y_i by $y_i + \varepsilon$ for $i \in B(\mathcal{T})$, $y_i - \varepsilon$ for $i \in A(\mathcal{T})$;
end
end
end

The initial values of y, M can be $y_i := 0\ \forall i$, $M := \emptyset$, or an approximation of the true solution obtained by a heuristic. When characterizing the Hungarian method, we see that on the one hand, always a preoptimal matching is kept (primal), on the other hand, at the same time a feasible dual solution y is computed. In that sense the method is a *primal-dual* algorithm. The running time of the methods is [15] $\mathcal{O}(n^2 m)$.
This can be improved, for example, to $\mathcal{O}(n^3)$ using the method introduced for the implementation of Prim's algorithm, Chap. 4.
We close this section by an example, which demonstrates, how the algorithms works.

Example: Minimum-weight perfect bipartite matching

We consider the graph already shown in Fig. 10.7. Assume that initially all $y_i := 0$ and the matching M is empty. As a consequence, initially $E_{=} = \emptyset$ and $\bar{c}_{ij} = c_{ij}$ for all $(i,j) \in E$.

We assume that for the first alternating tree $\mathcal{T}$, vertex a is selected as the root, thus $A(\mathcal{T}) = \emptyset$ and $B(\mathcal{T}) = \{a\}$. Since $E_{=} = \emptyset$, the inner **while**-loop is not entered. Thus, next $\epsilon = \min\{\bar{c}_{ab}, \bar{c}_{ac}\} = \min\{2,4\}$ is obtained. Thus, $y_2 := 2$ and $E_{=} = \{(a,b)\}$. Now the inner **while**-loop is entered with

$j = b$. Since b is exposed, the augmenting path a, b is inverted, resulting in $M = \{(a, b)\}$. As a result, we get the situation shown in Fig. 10.9.

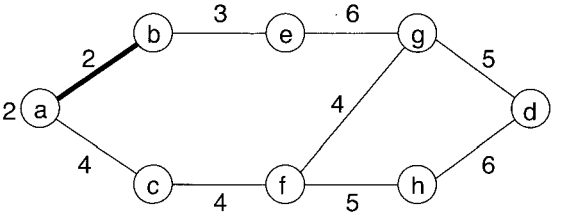


Figure 10.9: After the first edge has been matched using the minimum-weight perfect bipartite matching algorithm. The root of $\mathcal{T}$ has been a and $\epsilon = 2$.

We assume that now vertex d is selected as the root of $\mathcal{T}$, resulting in $y_d = 0+\epsilon = 5$, $\bar{c}_{dg} = 0$ and edge (d, g) is added to M. In the next step, let vertex f be the root of $\mathcal{T}$. Then $y_f = 0+\epsilon = 4$, $E_= = \{(a, b),\ (c, f),\ (d, g)\}$, $\sum y_i = 11$ and edge (c, f) joins the matching. The resulting situation is depicted if Fig. 10.10.

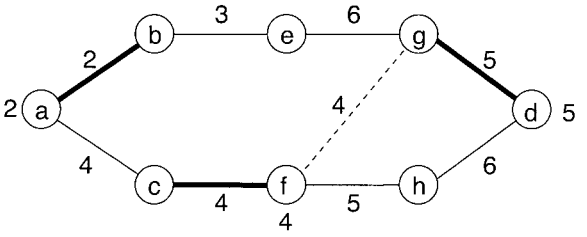


Figure 10.10: The graph after a, d and f have been the root of the alternating tree: $M = E_=\{(a, b),\ (c, f),\ (d, g)\}$ and $\sum y_i = 11$.

Now, only two exposed vertices are left. We assume that h is the root of the next alternating tree. First, no edges of $E_=$ can be used to extend the tree. Then, using $\bar{c}_{fh} = \bar{c}_{hd} = 1$, we obtain $\epsilon = 1$, leading to $y_h = 1$, $\sum y_i = 12$ and the edges (f, h) and (h, d) join $E_=$. Now, $\mathcal{T}$ can be extended twice, leading to the situation displayed in Fig. 10.11.

Now we have edges (c, a) and (g, e) which fulfill the condition $i \in B(\mathcal{T})$ and $j \notin V(\mathcal{T})$. Hence, $\epsilon = \min\{\bar{c}_{ca},\ \bar{c}_{ge}\} = \min\{6, 2\} = 2$ is obtained, leading to the situation we already encountered in Fig. 10.8. As we have seen already, $\epsilon = 3$ and edge (b, e) joins $E_=$. Then, using (b, e) the matching is augmented and the final result is obtained, which has already been presented in Fig. 10.7.

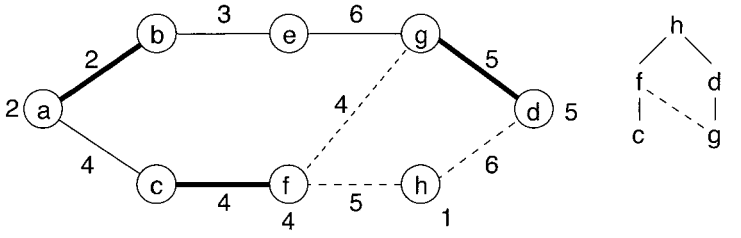


Figure 10.11: Bipartite graph and alternating tree with root h. Now $\sum y_i = 12$ and $E_= = \{(a,b),\ (c,f),\ (f,g),\ (d,g)\}$.

□

10.4.3 Cardinality Matching on General Graphs

Maximum matching on general graphs is considerably more difficult because of the presence of odd-length cycles, which are absent on bipartite graphs. Consider starting a BFS-tree for alternating paths, the alternating tree $\mathcal{T}$, from an exposed node a at level 0 (Fig. 10.12). We will now study, what happens when searching an even-level node (or a B-node) x, i.e. $x \in B(\mathcal{T})$ (necessarily covered). Let (x,y) be an unexplored edge incident to x. If y is exposed, we have found an augmenting path, and the augmentation proceeds as usual. If y is covered there are two possibilities:

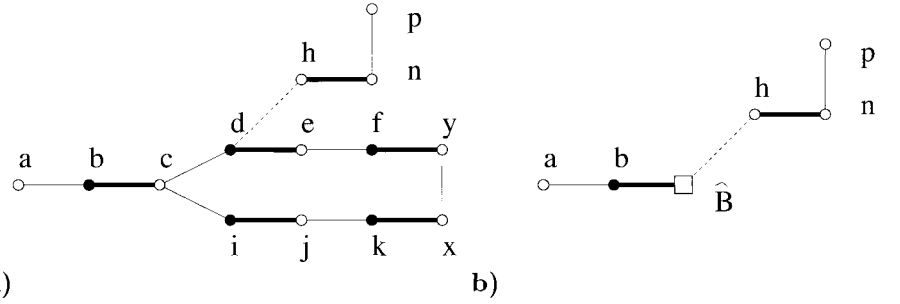


Figure 10.12: a) A blossom is an odd-cycle which is as heavy as possible in matched edges, i.e. contains $2k+1$ edges among which k are matched. **b)** The reduced graph $G \times C$ obtained by shrinking a blossom $C = \{c,d,e,f,y,x,k,j,i\}$ to a single node (the pseudo node $\hat{B}$) contains the same augmenting paths as the original graph.

- If y is an odd-level (or an A-node), nothing special happens.

- If y is marked as even (level 6 in Fig. 10.12), there is a special situation: there are two even-length alternating paths, one from a to x and one from a to y, and therefore (x, y) closes a cycle of odd-length. Let c be the last node common to both alternating paths (necessarily at an even level). The odd cycle including c is called a *blossom* B, and c its *base*. A blossom is essentially an odd-length alternating path from c to itself, as depicted in Fig. 10.12a. Its presence may conceal an existing augmenting path, like for example $\{a, b, c, i, j, k, x, y, f, e, d, h, n, p\}$ in Fig. 10.12, which would *not* be discovered by the BFS since edge (d, h) would never be explored. A blossom might also make us "find" an augmenting path where none actually exists, like for example $\{a, b, c, d, e, f, y, x, k, j, i, c, b, a\}$ in Fig. 10.12a.

The first polynomial-time algorithm to handle blossoms is due to Edmonds [16]. Edmonds' idea was to *shrink* blossoms, that is, replace them by a single *pseudo node* $\hat{B}$ obtaining a modified or *derived* graph G' as shown in Fig. 10.12b. The possibility of shrinking is justified by the following theorem due to Edmonds.

Theorem: (Edmonds)
There is an augmenting path in G *if and only if* there is an augmenting path in G'.

The existence of a blossom is discovered when edge (x, y) between two even-level nodes is first found, and its nodes and edges are identified by *backtracking* from x and y until the first common node is found (c in our example), which is the blossom's base.
Once identified, the blossom is shrunk, replacing all its nodes (among which there might be previously shrunk blossoms) by a single node, and reconnecting all edges incident to nodes in the blossom (necessarily uncovered edges) to this one. The search proceeds as usual, until an augmenting path is found, or none, in which case a is abandoned and no search will ever be started from it again. If an augmenting path is found, which does not involve any shrunk blossom, it is inverted as usual. If it contains blossom-nodes, they must be expanded first and one has to identify which way around the blossom the augmenting path goes. This may need to be repeated several times if blossoms are nested. After inverting the resulting augmenting path, a new search is started from a different node. A simple implementation of these ideas runs in $\mathcal{O}(|X|^4)$ time [10]. The fastest known algorithm for non-bipartite matching is also $\mathcal{O}(|E| \cdot \sqrt{|X|})$ time [17].

10.4.4 Minimum-weight Perfect Matching for General Graphs

The fact that, for bipartite graphs G with edge weights c, the linear-programming problem (10.3) has the same optimal value as the integer-linear programming problem that arises when we require $x_{ij} \in \{0, 1\}$ is equivalent to a classical theorem of Birkhoff, which says that G has a perfect matching if and only if (10.3) has a feasible solution, and, moreover, if G has a perfect matching then the minimum weight of a perfect matching is equal to the optimal value of (10.3). This result fails in general (non-bipartite) graphs $G = (V, E)$, where optimal solutions do appear for which some x_{ij}

are fractional (Fig. 10.13 shows an example of this) and thus do not correspond to a matching. Here, the reason for the complication is the existence of odd cycles [16]. Thus we need to explicitly impose the condition $x_{ij} = 0, 1$.

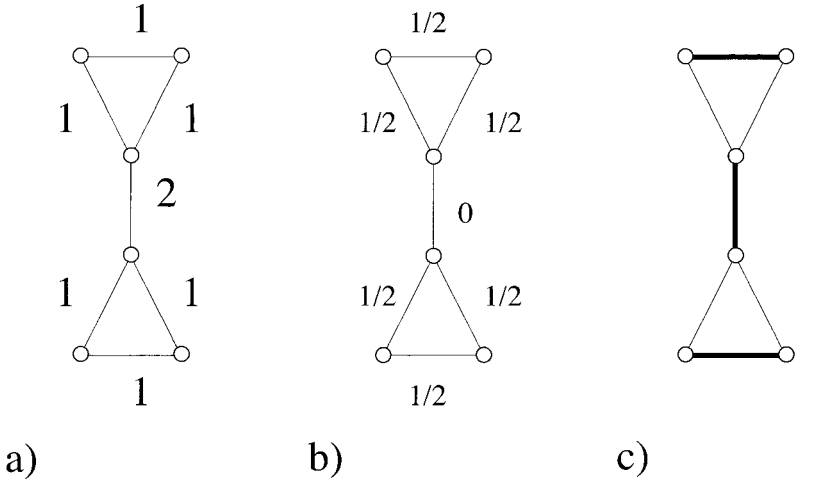


Figure 10.13: Example of fractional optimal solution on graphs with odd cycles. **a)** The graph and its weights. **b)** An optimal solution of P1 is fractional. **c)** The desired solution.

A solution to this problem for the case of general graphs has been found by Edmonds, and consists in adding new constraints, which impose $x_{ij} = 0, 1$ indirectly. In the subsequent discussion that will lead us finally to the Blossom algorithm for minimum-weight perfect matchings we follow Ref. [18], see also Ref. [15].
For each odd subset $S \subset G$ (i.e. S contains an odd number of nodes), we impose an additional set of constraints: let D be an *odd cut* generated by S, i.e.

$$D = \delta(S) = \{(i,j) \in E | i \in S, j \notin S\}. \tag{10.10}$$

If D is an odd cut and M a perfect matching, then M must contain at least one edge from D. It follows that, if x is the characteristic vector of a perfect matching, then for every odd cut D of G,

$$x(D) = \sum_{(i,j)\in D} x_{ij} \geq 1. \tag{10.11}$$

This is called a *blossom inequality*. By adding these inequlaities to the problem (10.3), we get a stronger linear-programming bound, i.e. the space of feasible vectors shrinks. [For example add the inequality (10.11) to (10.2), for D consisting of the vertical bond in Fig. 10.13, it is no longer possible to get a solution of value 3; in fact the new optimal value for the resulting problem is 4, which is also the minimum weight of a perfect matching.]

Let $\mathcal{C}$ denote the set of all odd cuts of G that are not of the form $\delta(i) = \{(i,j) \in E | j \in V\}$ for some node i. Then we are led to consider the linear-programming problem

$$\begin{array}{ll} \text{minimize} & \sum_{(i,j)\in E} c_{ij} x_{ij} \\ \\ \text{subject to} & \begin{cases} \forall i \in V: & x(\delta(i)) = 1\,, \\ \forall D \in \mathcal{C}: & x(D) \geq 1\,, \\ \forall (i,j) \in E: & x_{ij} \geq 0\,, \end{cases} \end{array} \tag{10.12}$$

where $x(\delta(i)) = \sum_{j \text{ with } (i,j)\in E} x_{ij}$.
As we have indicated, (10.12) provides a better approximation to the minimum weight of a perfect matching, but a much stronger statement can be made. Its optimal value *is* the minimum weight of a perfect matching. This is the fundamental theorem of Edmonds [19] (also "Matching Polytope Theorem"):

Theorem: Let G be a graph, and let $c \in \mathbb{R}^E$. Then G has a perfect matching if and only if (10.12) has a feasible solution. Moreover, if G has a perfect matching, the minimum weight of a perfect matching is equal to the optimal value of (10.12).

The algorithm that we will describe will construct a perfect matching M whose characteristic vector x^* is an optimal solution to (10.12), and so M is a minimum-weight perfect matching. This will provide a proof of the above theorem. The way in which we will know the x^* is optimal, is analogous to the bipartite case discussed above. We will also have a feasible solution to the dual linear-programming problem of (10.12) that satisfies orthogonality conditions for optimality of the primal and dual solutions. The dual problem to (10.12) is:

$$\begin{array}{ll} \text{maximize} & \sum_{i\in V} y_i + \sum_{D\in\mathcal{C}} Y_D \\ \\ \text{subject to} & \begin{cases} \forall (i,j) \in E: & c_{ij} \geq y_i + y_j + \sum_{D\in\mathcal{C}:\,(i,j)\in D} Y_D \\ \forall D \in \mathcal{C}: & Y_D \geq 0\,. \end{cases} \end{array} \tag{10.13}$$

Given a vector (y, Y) as in (10.13) and an edge (i,j), we denote, as in the bipartite case, by $\bar{c}_{ij} = \bar{c}_{ij}(y,Y)$ the difference

$$\bar{c}_{ij} = c_{ij} - y_i + y_j + \sum_{D\in\mathcal{C} \text{ with } (i,j)\in D} Y_D \tag{10.14}$$

which we also call the *reduced cost of the edge* (i,j). Thus (y,Y) is feasible in Eq. (10.13) if and only if $Y_D \geq 0$ for all $D \in \mathcal{C}$ and $\bar{c}_{ij} \geq 0$ for all $(i,j) \in E$.
Again, the important result from LP theory that we need for the algorithm is that the following orthogonality conditions are necessary and sufficient for the optimality of the primal and dual solutions

$$\begin{array}{lll} \forall (i,j) \in E: x_{ij} > 0 & \Rightarrow & \bar{c}_{ij} = 0 \\ \forall D \in \mathcal{C}: Y_D > 0 & \Rightarrow & x(D) = 1. \end{array} \tag{10.15}$$

If x is the characteristic vector of a perfect matching M of G, these conditions are equivalent to

$$\begin{array}{llll} \forall (i,j) \in E: & (i,j) \in M & \Rightarrow & \bar{c}_{ij} = 0 \\ \forall D \in \mathcal{C}: & Y_D > 0 & \Rightarrow & |M \cap D| = 1\,. \end{array} \tag{10.16}$$

It is not obvious how an algorithm will work with the dual variable Y, but the answer is suggested by the maximum-cardinality matching algorithm we discussed in Sec. 10.4.3. We will be working with derived graphs G' of G, and such graphs have the property

every odd cut of G' is an odd cut of G.

It follows from this, in particular, that every cut of the form $\delta_{G'}(v)$ for a pseudo node v of G' is an odd cut of G. These are the only odd cuts D of G' for which we will assign a positive value to Y_D. Note, however, that such a cut of G' need not have this property in G — it is of the form $\delta(S)$ where S is an odd subset of G which become a pseudo node of G' after (repeated) odd-circuit shrinkings. It follows that we can handle Y_D by replacing it by y_v, with the additional provision that $y_v \geq 0$.

We take the same approach as in the bipartite case, trying to find a perfect matching in $G_=$ using tree-extension and augmentation steps. When we get stuck, we change y in the same way, except that the existence of edges joining two nodes in $B(\mathcal{T})$ will limit the size of ε. In particular, there may be an equality edge joining two such nodes. Then we shrink the circuit C, but there is now a small problem: how do we take into account the variables y_i for $i \in V(C)$, when those nodes are no longer in the graph? The answer is that we update c as well. Namely, we replace c_{ij} by $c_{ij} - y_i$ for each edge (i,j) with $i \in V(C)$ and $j \notin V(C)$. Notice that by this transformation, and the setting of $y_C = 0$ for the new pseudo node, the reduced costs $\bar{c}_{ij}$ are the same for edges of the reduced graph $G \times C$ (in which the circuit C is replaced by a pseudo node) as they were for those edges in G. We will use c' to denote these updated weights, so when we speak of a *derived graph* G' of G, the weight vector c' is understood to come with G', or we may refer to the *derived pair* (G', c'). The observation about the invariance of the reduced costs, however, means that we can avoid $\bar{c}'$ in favor of $\bar{c}$. The subroutine for blossom shrinking is just the same as for the maximum-cardinality matching, c.f. Fig. 10.12. We assume that M' is a matching of a graph G', $\mathcal{T}$ an M'-alternating tree, and (i,j) is an edge of G' such that $i,j \in B(\mathcal{T})$. Here, and in the following, $E(G)$ denotes the edges of a graph (e.g. circuit or alternating tree) G:

procedure Shrink-update((i,j), M', $\mathcal{T}$, c')
begin
  Let C be the circuit formed by (i,j) together with the path in $\mathcal{T}$ from i to j;
  Replace G' by $G' \times C$, M' by $M' \backslash E(C)$;
  Replace $\mathcal{T}$ by the tree in G' having edge-set $E(\mathcal{T}) \backslash E(C)$;
  update c' and set $y_C := 0$;
end

An example of shrinking and updating weights is shown in Fig. 10.14. On the left is the graph with a feasible dual solution y (shown at the nodes) for which the edges of the left triangle are all equality edges. On the right we see the result of shrinking this circuit and updating the weights. Notice that an optimal perfect matching of this smaller graph extends to an optimal perfect matching of the original.

The essential justification of the stopping condition of the algorithm is that, when we have solved the problem on a derived graph, then we have solved the problem on the

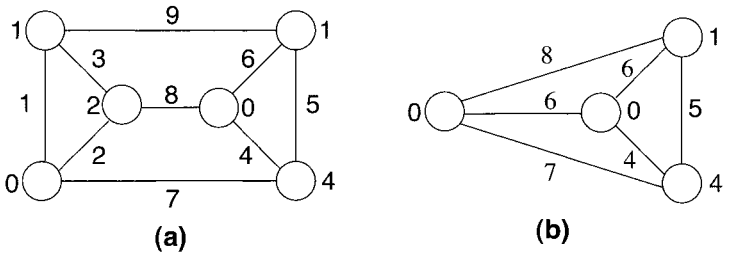


Figure 10.14: Shrinking and updating weights. The left triangle of the graph G in **(a)** is a circuit or blossom C that is shrunk to a pseudo node, yielding the reduced graph $G \times C$ in **(b)**. Its variable y_C is set to zero. The weights of the edges connected to C are updated according to $c_{ij} \to c_{ij} - y_i$ for each edge (i,j) with $i \in V(C)$ and $j \notin V(C)$. Notice that by this transformation, the reduced costs $\bar{c}_{ij} = c_{ij} - (y_i + y_j)$ stay the same for edges of G and of the reduced graph $G \times C$.

original graph. For this to be correct, we have to be very specific about what we mean by "solved the problem":

Proposition: Let G', c' be obtained from G, c by shrinking the odd circuit C of equality edges with respect to the dual-feasible solution y. Let M' be a perfect matching of G' and, with respect to G', c', let (y', Y') be a feasible solution to (10.12) such that M', (y', Y') satisfy conditions (10.16) and such that $y'_C \geq 0$. Let M be the perfect matching of G obtained by extending M' with edges from $E(C)$. Let (y, Y) be defined as follows. For $i \in V \backslash V(C)$ define $y_i = y'_i$ [for $i \in V(C)$, y_i is already defined]. For $D \in \mathcal{C}$, we put $Y_D = Y'_D$ if $Y'_D > 0$; we put $Y_D = y'_C$ if $D = \delta'(C)$; otherwise, we put $Y_D = 0$. Then with respect to G, c, (y, Y) is a feasible solution to (10.12) and M, (y, Y) satisfy the condition (10.16).

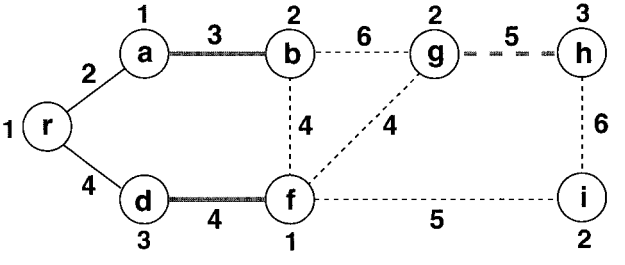


Figure 10.15: An example for a dual change where G is non-bipartite. The current matching M is indicated by the full lines, a perfect matching still needs edge (gh). Node r is exposed, r, b, and f are B-nodes, a and d are A-nodes. The weights c_{ij} are indicated at each edge, the current value of y_i is indicated on each node.

Now let us describe the dual variable change. It is the same one used in the bipartite

case, but with different rules for the choice of ε. First we need to consider edges (i,j) with $i,j \in B(\mathcal{T})$ when choosing ε, so we will need $\bar{c}_{ij}/2$ for such edges. Second we need to ensure that y_i remains nonnegative if i is a pseudo node, so we need $\varepsilon \leq y_i$ for such nodes. Since it is the nodes in $A(\mathcal{T})$ whose y-values are decreased by the change, these are the ones whose y-values affect the choice of ε. To illustrate, see Fig. 10.15, where $\varepsilon = 1/2$ is taken and (g,h) then becomes an equality edge. In the following the procedure for changing y is shown. It takes as input a derived pair (G',c'), a feasible solution y of (10.12) for this pair, a matching M' of G' consisting of equality edges, and an M'-alternating tree $\mathcal{T}$ consisting of equality edges in G'.

procedure Change-y((G',c'), y, M',$\mathcal{T}$)
begin
  Let $\varepsilon_1 := \min\{\bar{c}_{ij} | (i,j) \in E(G'), i \in B(\mathcal{T}), j \notin V(\mathcal{T})\}$;
  $\varepsilon_2 := \min\{\bar{c}_{ij}/2 | (i,j) \in E(G'), i \in B(\mathcal{T}), j \in B(\mathcal{T})\}$;
  $\varepsilon_3 := \min\{y_i | i \in A(\mathcal{T}),\ i$ pseudonode of $G'\}$;
  $\varepsilon := \min\{\varepsilon_1, \varepsilon_2, \varepsilon_3\}$;
  Replace y_i by
    $y_i + \varepsilon$, if $i \in B(\mathcal{T})$;
    $y_i - \varepsilon$, if $i \in A(\mathcal{T})$;
    y_i otherwise;
end

The last ingredient of the algorithm that we need is a way to handle the expansion of pseudo nodes. Note that we no longer have the luxury of expanding them all after finding an augmentation. The reason is that expanding the pseudo node i when y_i is positive would mean giving a value to a variable Y_D, where D is a cut of the current derived graph that is not of the form $\delta_{G'}(j)$ for some pseudo node j. Therefore, we can expand a pseudo node i *only if* $y_i = 0$. Moreover, in some sense we *need* to do such expansions, because a dual variable change may not result in any equality edge that could be used to augment, extend, or shrink (because the choice of ε was determined by some odd pseudo node). However, in this case, unlike the unweighted case, we are still in the process of constructing a tree, and we do not want to lose the progress that has been made. So the expanding step should, as well as updating M' and c', also update $\mathcal{T}$. The example in Fig. 10.16 suggests what to do.

Suppose that the odd node i of the M-alternating tree $\mathcal{T}$ is a pseudo node and expansion of the corresponding circuit C with the updating of M leaves us with the graph on the right. There is a natural way to update $\mathcal{T}$ after the expansion by keeping a subset of the edges of C. In Fig. 10.16c we show the new alternating tree after the pseudo node expansion illustrated in Fig. 10.16b.

The expansion of pseudo nodes and the related updating is performed using the following procedure. It takes as input a matching M' consisting of equality edges of a derived graph G', an M'-alternating tree $\mathcal{T}$ consisting of equality edges, costs c' and an odd pseudo node i of G' with $y_i = 0$:

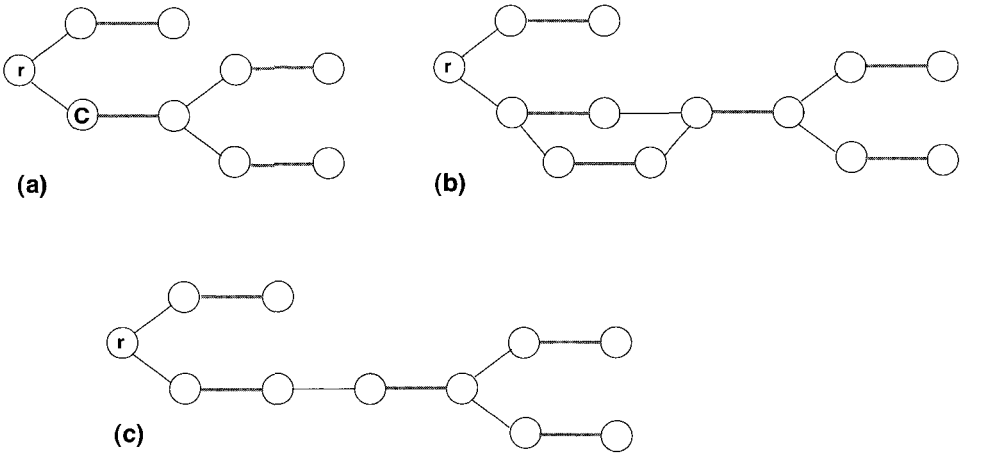


Figure 10.16: a) and b): Expanding an odd pseudo node; **c):** Tree update after pseudo node expansion (see text).

procedure expand-update (M', G', $\mathcal{T}$, c', i)
begin
Let f, g be the edges of $\mathcal{T}$ incident with i;
Let C be the circuit that was shrunk to form i;
Let u, w be the ends of f, g in $V(C)$;
Let P be the even-length path in C joining u to w;
Replace G' by the graph obtained by expanding C;
Replace M' by the matching obtained by extending M' to a matching of G';
Replace $\mathcal{T}$ by the tree having edge-sets $E(\mathcal{T}) \cup E(P)$;
for each edge (st) with $s \in V(C)$ and $t \notin V(C)$ **do**
Replace c'_{st} by $c'_{st} + y_s$;
end

Then the following proposition holds:

Proposition: After the application of the expand routine, M' is a matching contained in $E_=$, and $\mathcal{T}$ is an M'-alternating tree whose edges are all contained in $E_=$.

We can now state an algorithm for finding a minimum-weight perfect matching:

algorithm Blossom Algorithm for Minimum-weight Perfect Matching
begin
Let y be a feasible solution to (10.13), M' a matching of $G_=(y)$;
$G' = G$; Set $\mathcal{T} = (\{r\}, \emptyset)$, where r is an M'-exposed node of G';
while 1=1
begin
case: There exists $(i,j) \in E_=$ with $i,j \in G'$ and
$i \in B(\mathcal{T})$ and an M'-exposed node $j \notin V(\mathcal{T})$
Augment($\mathcal{T}$, (i,j), M');
if there is no M'-exposed node in G' **then**
begin
Extend M' to perfect batching M of G;
return;
end
else
Replace $\mathcal{T}$ by $(\{r\}, \emptyset)$, where r is M' exposed;
case: There exists $(i,j) \in E_=$ with $i,j \in G'$ and
$i \in B(\mathcal{T})$ and an M'-covered node $j \notin V(\mathcal{T})$
extent-alternating-tree($\mathcal{T}$, (i,j), M');
case: There exists $(i,j) \in E_=$ with $i,j \in G'$ and $i,j \in \mathcal{T}$
Shrink-update((i,j), M', $\mathcal{T}$, c');
case: There is a pseudo node $i \in A(\mathcal{T})$ with $y_i = 0$
Expand-update(M', G', $\mathcal{T}$, c', i);
case: None of the above
if every $(i,j) \in E$ incident in G' with $i \in B(\mathcal{T})$ has
$j \in A(\mathcal{T})$ and $A(\mathcal{T})$ contains no pseudo nodes
return G has no perfect matching;
else
Change-y((G', c'), y, M',$\mathcal{T}$);
end cases
end
end

This blossom algorithm terminates after performing $\mathcal{O}(n)$ augmentation steps, and $\mathcal{O}(n^2)$ tree-extension, shrinking, expanding, and dual change steps. Moreover, it calculates a minimum-weight perfect matching, or determines correctly that G has no perfect matching. As we have seen, the minimum weight perfect matching probably is one of the hardest combinatorial problems which can be solved in polynomial time. Therefore, before the reader starts to implement all the procedures, we recommend to have a look at the LEDA library [20], where efficient matching algorithms are available. Regarding the problem of finding the ground state of the 2d spin glass, which we found in Sec. 10.1 to be a minimum-weight perfect matching problem, we should emphasize that the underlying graph is a complete graph (all frustrated plaquettes can be matched with all others) and the number of nodes is always even (there is

always an even number of frustrated plaquettes). Therefore, the blossom algorithm always finds a perfect matching of minimum weight. To speed up the running time, it is possible to restrict the graph to edges of less than a certain weight, ($6J$ is a good rule of a thumb). Then the resulting graphs have much less edges, i.e. they are not complete any more.

10.5 Ground-state Calculations in 2d

In their pioneering work, Bieche et al. [1] studied the ground-state behavior of the ($\pm J$) spin glass as a function of the fraction x of anti-ferromagnetic bonds. For low concentrations x, we can expect a ferromagnetically ordered state, while for higher values of x, spin glass behavior can be anticipated. From simulations of systems of 22×22 spins they deduced that ferromagnetism was destroyed at $x^* = 0.145$. This zero-temperature transition is detected by the appearance of *fracture lines* which span the system, i.e. paths along which the number of satisfied and dissatisfied bonds is equal, and which can thus be inverted without any cost in energy. The authors also look at the fraction of spins in connected components, defined as sets of plaquettes which are matched together in any ground-state.

Later investigations by Barahona et al. [4] located the loss of ferromagnetism at a somewhat *lower* density of antiferromagnetic bonds, $x^* \sim 0.10$, suggesting that in the regime $0.10 \leq x \leq 0.15$ a *random antiphase* state exists which has zero magnetization but long-range order. This state is, according to the authors, characterized by the existence of *magnetic walls* (which are different from fracture lines) across which the magnetization changes sign. Thus the system is composed in this regime of "chunks" of opposite magnetization, so $\langle M \rangle = 0$ although the spin-spin correlation does not go to zero with distance. At $x = 0.15$ a second transition occurs, this time due to the proliferation of fracture lines, and rigidity (long-range order) is lost since the system is now broken into finite pieces which can be flipped without energy cost. Their conclusions were supported by later work using zero-temperature transfer-matrix methods [21]. Freund and Grassberger [22], using an approximation algorithm to find low-energy states on systems up to size 210×210, located the ferromagnetic transition at $x^* = 0.105$ but found no evidence of the random antiphase state. The largest 2d systems studied to date using exact matching algorithms appear to be $L = 300$ by Bendisch et al. [3, 23]. The authors examined the ground-state magnetization as a function of the density p of negative bonds on square lattices, and concluded that $0.096 < x^* < 0.108$, but their finite-size scaling analysis is not the best one could think of. They did not analyze the morphology of the states found, so no conclusion could be reached regarding the existence of the random antiphase state.

More recently Kawashima and Rieger [24], again using exact matching methods, compared previous analyses of the ground state of the 2d ($\pm J$) spin glass, in addition to performing new simulations. They summarized the results in this area in the phase diagram given in Fig. 10.17.

However Kawashima and Rieger found that the "spin-glass phase" is absent and that there is only one value of p_c. They thus argued for a direct transition from the

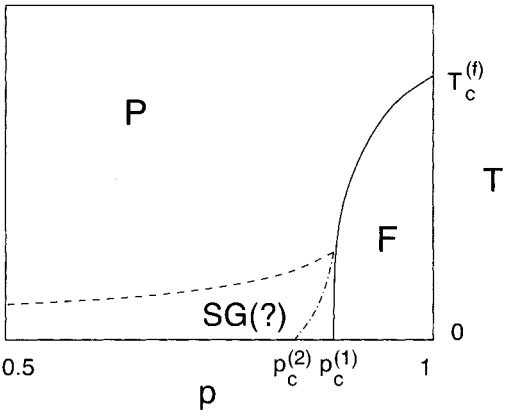


Figure 10.17: Phase diagram of a two-dimensional ($\pm J$) Ising model, with fraction $p = 1 - x$ of ferromagnetic bonds. From [24].

ferromagnetic state to a paramagnetic state, for both site- and bond-random spin-glass models. Their analysis is based on the stiffness energy, i.e. the difference $\Delta E = E_p - E_a$, where E_p is the ground-state energy with periodic boundary conditions and E_a is the ground-state energy with antiperiodic boundary conditions, see also Sec. 9.3. The scaling behavior [25],

$$[\Delta E]_{av} \sim L^{\rho}, \qquad [\Delta E^2]_{av} \sim L^{2\Theta_S} \tag{10.17}$$

was assumed. In a ferromagnetic state $\rho = 1$ and $\Theta_S = 2$, while in a paramagnetic state $\rho < 0$ and $\Theta_S < 0$. However, in an ordered spin-glass state we have $\rho < 0$, and $\Theta_S > 0$. Although the conclusion of this analysis was the absence of a spin-glass phase, the exponent they found $\Theta_S = -0.056(6)$ for $p < p_c$ is small. Although, in our view, the numerical evidence that there is no finite-temperature spin-glass transition in the two-dimensional Edwards-Anderson model with a binary bond distribution is compelling one should note that a different view has been advocated [26]. In order to support this view a defect-energy calculation similar to the one described above has been presented [27]. It was shown that the probability distribution of $|\Delta E|$ does not shrink to a delta-function centered at ΔE for $L \to \infty$, instead it maintains a finite width. However, even if $\lim_{L\to\infty} |\Delta E| = \Delta E_\infty > 0$ a *finite* value of this limit indicates that the spin-glass state will be unstable with respect to thermal fluctuations since arbitrarily large clusters will be flipped via activated processes with probability $\exp(-\Delta E_\infty / T)$ at temperature T. The correct conclusion is then that there is no finite-T spin-glass transition in the 2d EA model with binary couplings.

The 2d Ising spin glass with a binary ($\pm J$) bond distribution is in a different universality class than the model with a continuous bond distribution. The degeneracies, which are typical for a discrete bond distribution, are absent for the continuous case for which the ground state is unique (up to a global spin flip). Even in the continuous

case, the ground state is found using a minimal weighted matching algorithm (with the modification that now not only the length of a path between two matched plaquettes counts for the weight, but also the strength of the bonds laying on this path).
The latest estimate for the stiffness exponent of the 2d Ising-spin-glass model with a uniform bond distribution between 0 and 1 obtained via exact ground-state calculations [28] is

$$[\Delta E^2]_{\rm av} \propto L_{\rm S}^{\Theta} \quad \text{with} \quad \Theta_{\rm S} = -0.281 \pm 0.002, \tag{10.18}$$

which implies that in the infinite system arbitrarily large clusters can be flipped with vanishingly small excitation energy. Therefore the spin-glass order is unstable with respect to thermal fluctuations and one does not have a spin-glass transition at finite temperature.
Nevertheless, the spin-glass correlation length ξ (defining the length scale over which spatial correlations like $[\langle S_i S_{i+r}\rangle_T^2]_{\rm av}$ decay) will diverge at zero temperature as $\xi \sim T^{-1/\nu}$, where ν is the thermal exponent. A scaling theory (for a zero-temperature fixed-point scenario as is given here) predicts that $\nu = 1/|\Theta_{\rm S}|$ which, using the most accurate Monte Carlo [29] and transfer-matrix [30] calculations gives $\nu = 2.0 \pm 0.2$, is *inconsistent* with Eq. (10.18). This is certainly an important unsolved puzzle, which might be rooted in some conceptional problems concerning the use of periodic/antiperiodic boundary conditions to calculate large-scale low-energy excitations in a spin glass (these problems were first discussed in the context of the XY spin glass [31] and the gauge glass [32]). Further work in this direction will be rewarding.
Next we would like to focus our attention on the concept of *chaos* in spin glasses [33]. This notion implies an extreme sensitivity of the SG-state with respect to small parameter changes like temperature or field variations. There is a length scale λ — the so called overlap length — beyond which the spin configurations within the same sample become completely decorrelated if compared for instance at two different temperatures

$$C_{\Delta T} = [\langle \sigma_i \sigma_{i+r}\rangle_T \langle \sigma_i \sigma_{i+r}\rangle_{T+\Delta T}]_{\rm av} \sim \exp\left(-r/\lambda(\Delta T)\right). \tag{10.19}$$

This should also hold for the ground states if one slightly varies the interaction strengths J_{ij} in a random manner with amplitude δ. Let $\{\sigma\}$ be the ground state of a sample with couplings J_{ij} and let $\{\sigma'\}$ be the ground state of a sample with couplings $J_{ij} + \delta K_{ij}$, where the K_{ij} are random (with zero mean and variance one) and δ is a small amplitude. Now define the overlap correlation function as

$$C_\delta(r) = [\sigma_i \sigma_{i+r}\, \sigma'_i \sigma'_{i+r}]_{\rm av} \;\sim\; \tilde{c}(r\delta^{1/\zeta}), \tag{10.20}$$

where the last relation indicates the scaling behavior we would expect (the overlap length being $\lambda \sim \delta^{-1/\zeta}$) and ζ is the *chaos* exponent. In [28] this scaling prediction was confirmed with $1/\zeta = 1.2 \pm 0.1$ by exact ground-state calculations of the 2d Ising spin glass with a uniform coupling distribution, and the corresponding scaling plot for $C_\delta(r)$ is shown in Fig. 10.18.

Bibliography

[1] I. Bieche, R. Maynard, R. Rammal, and J.P. Uhry, *J. Phys. A* **13**, 2553 (1980)

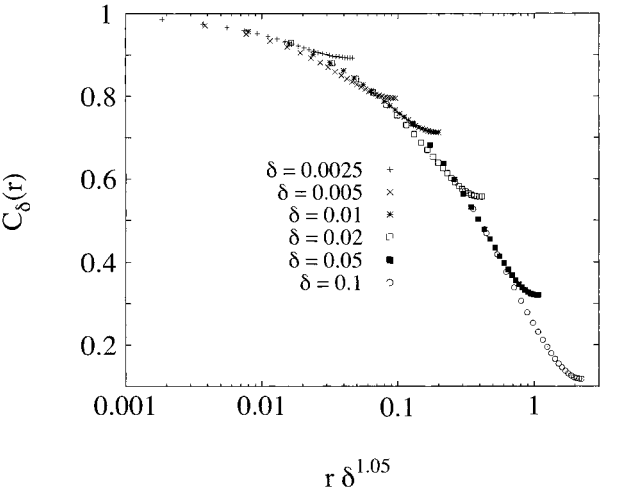


Figure 10.18: Scaling plot of the overlap correlation function $C_\delta(r)$ versus r/L^* with $L^* = \delta^{-1/\zeta}$. The best data collapse (for data confined to $r < L/4$) is obtained for $1/\zeta = 1.05$. The system size is $L = 50$ and the data are averaged over 400 samples. These were obtained by creating about 80 reference instances and creating 5 random perturbations of strength δ for each. From [28].

[2] F. Barahona, *J. Phys. A* **15**, 3241 (1982)

[3] J. Bendisch, U. Derigs, and A. Metz, *Disc. Appl. Math.* **52**, 139 (1994)

[4] F. Barahona, R. Maynard, R. Rammal, and J.P. Uhry, *J. Phys. A* **15**, 673 (1982)

[5] R.L. Graham, M. Grötschel, and L. Lovász, *Handbook of Combinatorics*, (Elsevier Science Publishers, Amsterdam 1995)

[6] L. Lovász and M.D. Plummer, *Matching Theory*, (Elsevier Science Publishers, Amsterdam 1986)

[7] C. Berge, *Proc. Am. Math. Soc.* **43**, 842 (1957)

[8] R.Z. Norman and M. Rabin, *Proc. Am. Math. Soc.* **10**, 315 (1959)

[9] J. van Leeuwen (ed.), *Handbook of Theoretical Computer Science A: Algorithms and Complexity*, (Elsevier Science Publishers, Amsterdam 1990)

[10] C.H. Papadimitriou and K. Steiglitz, *Combinatorial Optimization: Algorithms and Complexity*, (Prentice Hall, New Jersy 1982)

[11] E. Minieka, *Optimization Algorithms for Networks and Graphs*, (Marcel Dekker Inc., New York and Basel 1978)

[12] J. Hopcroft and R. Karp, *SIAM J. Comp.* **4**, 225 (1973)

[13] H. W. Kuhn, *Naval Research Logistics Quarterly* **2**, 83 (1955)

[14] J. Munkres, *SIAM J. Appl. Math.* **5**, 32 (1957)

[15] B. Korte and J. Vygen, *Combinatorial Optimization - Theory and Algorithms*, (Springer, Heidelberg 2000)

[16] J. Edmonds, *Can. J. Math.* **17**, 449 (1965)

[17] S. Micali and V.V. Vazirani, in: *Proc. Twenty-first Annual Symposium on the Foundations of Computer Science*, Long Beach, California, *IEEE* **17** (1980)

[18] W.J. Cook, W.H. Cunningham, W.R. Pulleyblank, and A. Schrijver, *Combinatorial Optimization*, (John Wiley & Sons, New York 1998)

[19] J. Edmonds, *J. Res. Natl. Bur. Stand. B* **69**, 125 (1965)

[20] K. Mehlhorn and St. Näher, The LEDA Platform of Combinatorial and Geometric Computing, (Cambridge University Press, Cambridge 1999);
see also http://www.mpi-sb.mpg.de/LEDA/leda.html

[21] Y. Ozeki, *J. Phys. Soc. Jpn.* **59**, 3531 (1990)

[22] H. Freund and P. Grassberger, *J. Phys. A* **22**, 4045 (1989)

[23] J. Bendisch, *Physica A* **245**, 560 (1997)

[24] N. Kawashima and H. Rieger, *Europhys. Lett.* **39**, 85 (1997)

[25] A.J. Bray and M.A. Moore, in: L. van Hemmen and I. Morgenstern (ed.), *Heidelberg Colloqium on Glassy Dynamics and Optimization*, (Springer, Heidelberg 1985)

[26] T. Shirakura and F. Matsubara, *J. Phys. Soc. Jpn.* **64**, 2338 91995)

[27] F. Matsubara, T. Shirakura, and M. Shiomi, *Phys. Rev. B* **58**, 11821 (1998)

[28] H. Rieger, L. Santen, U. Blasum, M. Diehl, M. Jünger, and G. Rinaldi, *J. Phys. A* **29**, 3939 (1996)

[29] S. Liang, *Phys. Rev. Lett.* **69**, 2145 (1992)

[30] N. Kawashima, N. Hatano, and M. Suzuki, *J. Phys. A* **25**, 4985 (1992)

[31] J.M. Kosterlitz and M.V. Simkin, (1997). *Phys. Rev. Lett.* **79**, 1098 (1997)

[32] J.M. Kosterlitz and N. Akino, *Phys. Rev. Lett.* **81**, 4672 (1998)

[33] A.J. Bray and M.A. Moore, *Phys. Rev. Lett.* **58**, 57 (1987)

11 Monte Carlo Methods

For many, actually most, optimization problems in physics as well in economics or industrial situations no polynomial-time algorithms appear to be at hand. Stochastic optimization is a tool that is applicable in *all* cases and with which one can at least hope to generate a *good approximation* to the optimal solution of a given energy or cost function. On the other hand, stochastic optimization methods can be applied to almost all types of problems. Thus, the models we can treat and the results we can expect are similar to genetic algorithms, which have been presented in Chap. 8.
In this chapter we give an overview of various methods that are at hand to attack physical problems for which none of the elegant polynomial algorithms we elsewhere describe in this book work. We start with a general outline of stochastic optimization methods. Next, *simulated annealing* is introduced, which can be seen as an algorithmic equivalent of cooling experiments. In the third section, *parallel tempering* is explained, which is an extension of simulated annealing. In the following section, the *prune-enriched Rosenbluth method* (PERM) is introduced. It allows the study of lattice polymers at low temperatures. Finally, PERM is applied to search for low energy configurations of folded proteins.

11.1 Stochastic Optimization: Simple Concepts

In essence one samples the configuration space $\{S\}$ of a given problem

$$\min_S\{\mathcal{H}(S)\} \tag{11.1}$$

stochastically — or randomly — and uses more or less sophisticated rules, based on some knowledge or intuition about the underlying energy landscape [we call $\mathcal{H}(S)$ the energy of state (or configuration] S or the Hamiltonian, for computer scientists these are the costs or the objective function], to improve on the trivial completely random search. These rules are expressed in terms of acceptance rates $p(S \to S')$ for random moves in the configuration space from state S to a new state S'.
The *random walk* procedure, the most trivial and most inefficient rule, has $\forall S, S' : p(S \to S') = 1$, which means *any* new suggested configuration S' is accepted (Fig. 11.1 left). Its energy $\mathcal{H}(S)$ is recorded and after a sufficiently long run the lowest energy configuration is the approximation we found to the optimal solution. Obviously this technique is doomed to fail in nearly all cases of practical interest and can only serve as a preliminary exploration of the energy landscape of the problem.

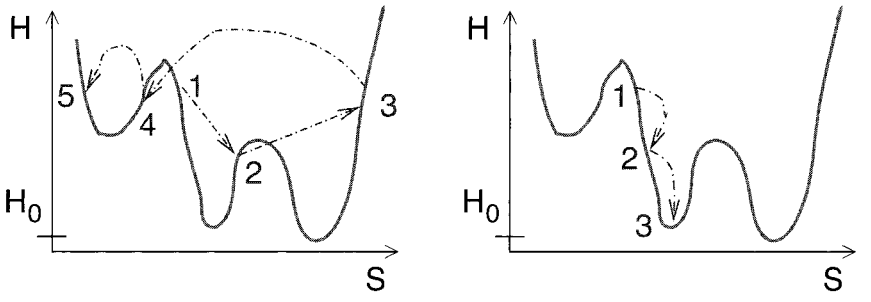


Figure 11.1: Comparison of the *random walk* method (left) with the *greedy algorithm* (right). Displayed are, respectively, the energy as a function of the state S and sample trajectories in configuration space. H_0 denotes the energy of the minimum. Here the random walk results in the second configuration as minimum, while the greedy algorithm always results in the final state.

For a well behaved energy landscape with just *one* minimum the *greedy algorithm* is often successful (Fig. 11.1 right). It accepts only moves that generate states S' with a lower energy and is defined by $p(S \to S') = \theta(-\Delta\mathcal{H})$, where $\Delta\mathcal{H} = \mathcal{H}(S') - \mathcal{H}(S)$ is the energy difference between the old and the new state and $\theta(x) = 1$ for $x \geq 0$ and 0 otherwise, the Heavyside step function.
As soon as one deals with problems that have many local energy minima one needs to accept moves to states with higher energy too, in order to escape local minima and overcome energy barriers separating different minima. One of the simplest of these techniques appears to be *threshold accepting* [1], which is defined by the acceptance rule

$$p(s \to S') = \theta(\epsilon - \Delta\mathcal{H}) \tag{11.2}$$

which means that all moves leading to an energy decrease are accepted, as well as moves with a positive energy difference $\Delta\mathcal{H}$ as long as it is smaller than the threshold ϵ. This means that in principle arbitrarily large energy barriers can be overcome, if enough intermediate configurations with small energy differences are present (Fig. 11.2 left).
Such a procedure is already an improvement on the greedy method but it gets stuck in an energy landscape that has local minima surrounded by energy barriers that are larger than the threshold ϵ — so called golf holes in the energy landscape (Fig. 11.2 right). One should note that in a high-dimensional configuration space the number of escape routes or directions out of a local energy minimum is also enlarged and with this also the chance to get out of a local minimum even at low threshold values. This parameter ϵ, which is reduced during the run down to 0, plays the role of a temperature, although the acceptance rate does not at all fulfill detailed balance and the stationary distribution of this process does not have anything to do with the Boltzmann distribution, see Chap. 5. Obviously it is advantageous to repeat the

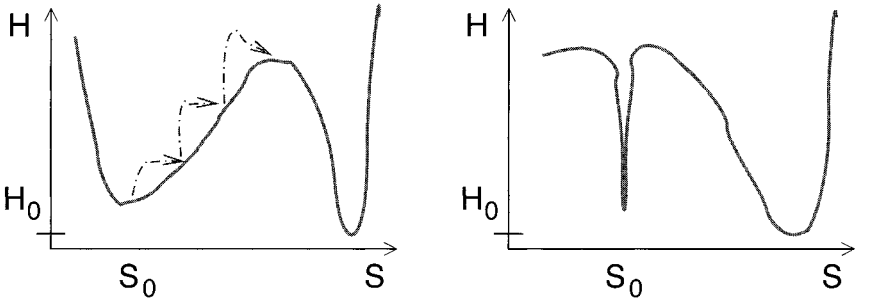


Figure 11.2: Threshold accepting: large energy barriers can be overcome starting from S_0 by several small steps (left), unless they are "golf holes" (right).

whole procedure many times starting with different initial configurations.
Another method of stochastic optimization is the *great deluge algorithm* [2]. Here one performs a random walk in a part of the configuration space that lies below a specified level ε for the energy $\mathcal{H}(S)$. The acceptance rule is therefore given by

$$p(S \to S') = \theta(\varepsilon - \mathcal{H}(S')), \tag{11.3}$$

i.e. a new configuration S' is accepted when its energy is below ε and it is rejected otherwise. Note that the acceptance rate is independent of the old state S (a random walk) and each state S' is accepted with equal probability as long as its energy is below ε, which plays the role of a temperature that is slowly decreased until the system settles in a (local) minimum. This algorithm reminds one of a great deluge, since if one inverts the energy landscape, one looks for the absolute maximum, its highest top. One lets the water level rise continuously and one can only visit spots that are not yet flooded. The great deluge algorithm obeys detailed balance (c.f. Chap. 5) — since $p(S \to S') = p(S' \to S)$ for all states S and S' that can be reached in the stationary state [in which $\mathcal{H}(S) < \varepsilon$ and $\mathcal{H}(S') < \varepsilon$] of this stochastic process. However, it is not ergodic, since for low enough ε the part of the configuration space that is sampled splits into several islands between which no transition is possible any more.

11.2 Simulated Annealing

Simulated annealing is a stochastic optimization procedure that is based on a stochastic process that leads to a stationary state which is described by the Boltzmann distribution for the underlying problem [i.e. the system described by its Hamilton function $\mathcal{H}(S)$]. In essence this procedure resembles cooling a crystal down slowly such that defects and other lattice impurities (forming metastable states) can heal out and a pure crystalline structure (the global minimum) is achieved at low temperature. In this process the temperature (which allows jumps over energy barriers between various

configurations) has to decrease very slowly since otherwise the molecular configuration gets stuck and the crystal will be imperfect. This is called annealing in materials science and various stochastic optimization methods are guided by this physical spirit such that they have been called *simulated annealing* [3].
We recall what we said in Sec. 5.1. We consider a physical system that can be in states S and is described by a Hamiltonian $\mathcal{H}(S)$. Its equilibrium properties at a temperature T are determined by the Boltzmann distribution

$$P_{\text{eq}}(S) = \frac{1}{Z} \exp(-\mathcal{H}(S)/k_{\text{B}}T) \tag{11.4}$$

where Z is its partition function $Z = \sum_S \exp(-\mathcal{H}(S)/k_{\text{B}}T)$, which is simply the normalizing factor for a canonical distribution with weights proportional to $\exp(-\mathcal{H}(S)/k_{\text{B}}T)$.
We learned in Eqs. (5.15) and (5.16) that a stochastic process with Metropolis transition rates [4]

$$p(S \to S') = \begin{cases} 1 & \text{for} \quad \Delta\mathcal{H} \leq 0 \\ \exp(-\Delta\mathcal{H}/k_{\text{B}}T) & \text{for} \quad \Delta\mathcal{H} > 0 \end{cases} \tag{11.5}$$

where $\Delta\mathcal{H} = \mathcal{H}(S') - \mathcal{H}(S)$ is the energy difference between the old state S' and the new state S, leads to a stationary state that is governed by the stationary distribution $P_{\text{eq}}(S)$. Another choice leading to the same results are the heat-bath transition rates

$$p(S \to S') = [1 + \exp(\Delta\mathcal{H}/k_{\text{B}}T)] \tag{11.6}$$

since via the Ansatz $p(S \to S') = P_{\text{eq}}(S')/[P_{\text{eq}}(S') + P_{\text{eq}}(S)]$ it also fulfills detailed balance Eq. (5.16).
Moreover, we have pointed out that for $T \to 0$ the only state(s) that are sampled by the equilibrium distribution $P_{\text{eq}}(S)$ are those with the lowest energy. Thus, if we could equilibrate the system in our computer run for arbitrarily low temperature *and* lower the temperature step by step, we may expect to find the state with the lowest energy, the ground state of the system, at the end. This is the spirit of simulated annealing, first introduced into optimization problems by Kirkpatrick et al. [3].

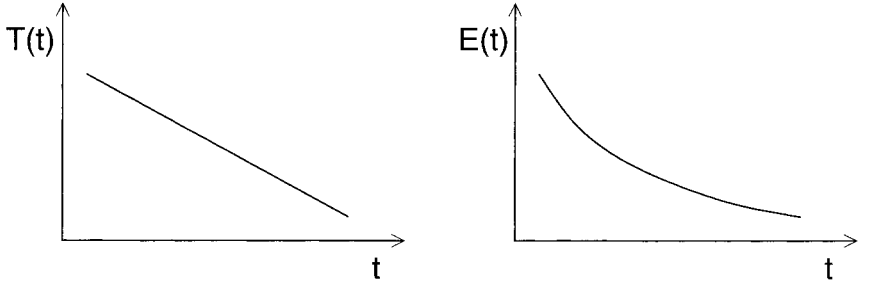


Figure 11.3: Simulated annealing. A linear cooling schedule $T(t)$ (left) and a sketch of the resulting average energy $E(t)$ as a function of time typically.

Obviously it will not be possible to equilibrate arbitrarily large systems at arbitrarily low temperatures with only a finite amount of computer time available. The crucial ingredient for a good performance of the simulated annealing procedure will therefore be the cooling protocol $T(t)$, where T is the temperature and t is a measure of the computer time (e.g. number of Monte Carlo sweeps through the system). With the help of the theory of Markov processes it has been shown that in principal cooling protocols do exist with which simulated annealing finds the optimal solution of a given (finite) problem in infinite time. These cooling protocols have the form

$$T(t) = \frac{a}{b + \log(t)} \tag{11.7}$$

where a and b are positive constants that depend on the special problem. Practical applications, of course, have to be finished in finite time and therefore prefer the following empirical cooling protocols. The *linear* scheme (Fig. 11.3 left)

$$T(t) = a - bt \tag{11.8}$$

where a is the starting temperature and b the step size for decreasing the temperature (usually $0.01 \leq b \leq 0.2$); and the *exponential* scheme:

$$T(t) = ab^t \tag{11.9}$$

where again a is the starting temperature and b the cooling rate (usually $0.8 \leq b \leq 0.999$). For completeness we present a schematic listing of a simulated annealing procedure:

algorithm simulated annealing
begin
  choose start configuration S;
  for $t := 1, \ldots, t_{\max}$ **do**
  begin
    set temperature $T := T(t)$;
    Monte Carlo$(M(t), T)$ at temperature T with $M(t)$ steps;
  end
end

The instruction **Monte Carlo** means that a Monte Carlo simulation at temperature T with transition rates derived from the detailed balance in Eq. (5.15) with respect to the equilibrium distribution (11.4) — as for instance the Metropolis (11.5) or heat-bath (11.6) rules — has to be performed according to the general outline presented in Sec. 5.1. In the right part of Fig. 11.3 a typical outcome of a cooling experiment is shown. Since only a finite number $M(t)$ of Monte Carlo steps is allowed it is crucial to try to achieve a fast equilibration within this time window. In the next section we present a Monte Carlo update procedure that is particularly helpful in this respect.

11.3 Parallel Tempering

Within the context of the random-field Ising model, which we discussed in Chap. 6, Marinari and Parisi [5] introduced a new stochastic optimization method, which they called *simulated tempering*. This Monte Carlo scheme is based on the extension of the canonical ensemble introduced in Chap. 5 by, for instance, a multiplicity of systems at different temperature and is related to the so-called *multi-canonical ensemble* [6]. A very efficient and easy-to-use realization of a similar concept has been introduced by Hukushima and Nemoto [7] and successfully applied to the 3d spin glass model, which we discussed in Chap. 9. They called it *parallel tempering* and it is particularly useful for disordered systems that show glassy (i.e. very sluggish) low temperature dynamics. The basic idea behind the parallel tempering is to perform several different simulations simultaneously on the same system but at different temperatures. From time to time one swaps the states of the system in two of the simulations with a certain probability. The latter is chosen such that the states of each system still obey the Boltzmann distribution at the appropriate temperature. By swapping the states we mean that one exchanges the configurations S (previously held at temperature T) and S' (previously at T'). The benefit from this is that the higher-temperature simulation helps the (previous) lower-temperature configuration to cross energy barriers in the system. At higher-temperatures one crosses the barriers with relative ease, and when it gets to the other side, we swap the states of the two systems, thereby carrying the lower-temperature model across a barrier which it otherwise would not have been able to cross.

For simplicity we consider only two simulations of the system at the same time, but the concept can – and should – be generalized to many simultaneous simulations of the system at different temperatures, which are not too far from each other in order to get a sufficiently large overlap in the energy distribution of the state (we shall soon see why). One simulation of the system is done at temperature T_1, the other at $T_2 > T_1$. The respective inverse temperatures are called $\beta_1 = 1/k_B T_1$ and $\beta_2 = 1/k_B T_2$. The Monte Carlo algorithm works as follows. We perform a conventional Monte Carlo simulation with systems in parallel (which is done best on a parallel computer with one CPU for one system, but it is of course also manageable on a serial machine) using for instance a simple Metropolis update rule (11.5). But from time to time instead of doing this we calculate the energy difference of the current state S_1 and S_2 of the two simulations, $\Delta E = E_2 - E_1$, where $E_1 = \mathcal{H}(S_1)$ and $E_2 = \mathcal{H}(S_2)$, and we swap the two spin configurations ($[S_1, S_2] \to [S_2, S_1]$) with an acceptance probability

$$w([S_1, S_2] \to [S_2, S_1]) = \begin{cases} \exp(-(\beta_1 - \beta_2)\Delta E) & \text{if} \quad \Delta > 0 \\ 1 & \text{otherwise} \end{cases} \tag{11.10}$$

Here we use the notation $[S, S']$ to describe the state of the two combined systems: first at temperature T_1 in state S and second at temperature T_2 in state S'. These transition rates for swapping fulfill detailed balance (5.15) with respect to the joint equilibrium distribution function for the two systems

$$P_{\text{eq}}([S, S']) = \exp(-\beta_1 \mathcal{H}(S)) \exp(-\beta_2 \mathcal{H}(S'))/Z_1 Z_2, \tag{11.11}$$

which can easily be seen by observing that

$$\frac{w([S_1,S_2]\to[S_2,S_1])}{w([S_2,S_1]\to[S_1,S_2])}=\frac{\exp(-\beta_1(E_2-E_1))}{\exp(-\beta_2(E_2-E_1))}=\frac{P_{\rm eq}([S_2,S_1])}{P_{\rm eq}([S_1,S_2])} \tag{11.12}$$

The transition rates in the conventional Monte Carlo steps where we do not swap states fulfill detailed balance anyway and they are also ergodic. Hence the combined procedure fulfills detailed balance and is ergodic, and therefore both simulations will after a sufficiently long time sample the Boltzmann distribution for this model at temperature T_1 in the first system and temperature T_2 in the second. The crucial advantage is, again, that the T_2 system helps the T_1 system to equilibrate faster:

algorithm Parallel tempering
begin
  choose start configurations S_1 and S_2;
  for $t := 1,\ldots,t_{\max}$ **do**
  begin
    Monte Carlo(M,T_1) for system 1;
    Monte Carlo(M,T_2) for system 2;
    $\Delta E := \mathcal{H}(S_2)-\mathcal{H}(S_1)$;
    if $(\Delta E<0)$ **then**
      accept $[S_1,S_2]\to[S_2,S_1]$;
    else
    begin
      $w := \exp(-(\beta_1-\beta_2)\Delta E)$;
      generate uniform random number $x\in[0,1]$;
      if $(x<w)$ **then**
        accept $[S_1,S_2]\to[S_2,S_1]$;
    end
  end
end

One question is how often one should perform the swap moves (i.e. how large should the parameter M in the above listing be). It is clear that it does not make sense to try this too frequently, the high temperature system should have time to be carried away from a local minimum, otherwise it is useless. To estimate the time needed for this one could, for instance, calculate the normalized energy autocorrelation function

$$C_E(t)=\frac{\langle H(0)H(t)\rangle-\langle H\rangle^2}{\langle H^2\rangle-\langle H\rangle^2}, \tag{11.13}$$

determine its correlation time τ from $C_E(\tau)=e^{-1}C_E(0)$ and choose $M>\tau$. A practical point concerning the swapping procedure is that obviously we do not need to shuffle the configurations S_1 and S_2 from one array to the other, the same effect is achieved by simply exchanging the temperatures T_1 and T_2 for the two systems. Finally: We mentioned before that the method becomes particularly efficient if it is done on a parallel computer (which is the origin of its name *parallel* tempering), where

each node (or a block of nodes) simulates one copy of the system at one temperature out of not only two but many (typically 32 or so), see Fig. 11.4. Since the swapping probabilities are exponentially small in the energy difference of the systems at two different temperatures and because the average energy increases with the temperature the different temperatures should not be too far from each other and one should swap only between systems at neighboring temperatures. But, of course, the temperatures also should not be chosen too close to each other if we want to find good approximations to the lowest energy configurations, i.e. to reach as low temperatures as possible. A good rule of thumb is, that one should achieve an acceptance ratio of 0.5 for the swapping move for each pair of neighboring temperatures.

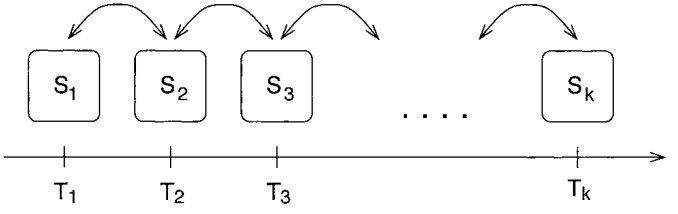


Figure 11.4: Parallel tempering with k different temperatures $T_1 < T_2 < \ldots < T_k$. At each temperature a system is simulated using conventional Monte Carlo. From time to time, configurations are exchanged between neighboring temperatures, such that detailed balance is fulfilled.

11.4 Prune-enriched Rosenbluth Method (PERM)

In this section we will discuss an alternative Monte Carlo method that has been very successful in finding the ground state of various lattice polymer models and other things. Here we follow Grassberger [8] and Grassberger and Frauenkron [9]. To exemplify the model we consider a lattice model for θ-polymers, i.e. self-avoiding walks (SAWs) with an additional nearest neighbor attractive interaction (see Fig. 11.5). In the simplest model, each pair of nearest neighbors, which are not connected by a bond, contributes the amount $-\epsilon$ to the energy. On large scales it overrides the repulsion, so that the typical configuration changes from an open "coil" to a dense "globule" at finite temperatures. The ground state of course is a compact configuration that is easy to find, but when we consider models of self-avoiding polymers with different monomers that have attractive as well as repulsive interactions, as in the context of a model-protein that we discuss in the next section, the problem of finding the ground state becomes highly non-trivial.

The algorithm that has been proposed in [8] is based on two ideas going back to the very first days of Monte Carlo simulations, the Rosenbluth-Rosenbluth method [10] and enrichment [11]. Both are modifications of simple sampling in which a chain is built by adding one new monomer after the other, placing it at a random neighbor

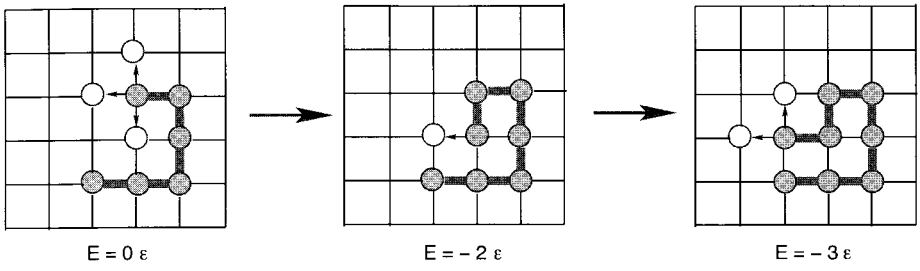


Figure 11.5: Lattice model of a θ-polymers with an additional nearest neighbor attractive interaction ϵ. The chain is grown by adding one monomer at a random neighbor site of the last placed monomer indicated by open circles.

site of the last placed monomer. In order to obtain the correct statistics, an already occupied neighbor should *not* be avoided, but any attempt to place the monomer at such a place is punished by discarding the entire chain. This leads to an exponential "attrition", which makes the method useless for long chains.

Rosenbluth and Rosenbluth [10] observed that this exponential attrition can be strongly reduced by simulating a biased sample, and correcting the bias by means of a weight associated with each configuration. The biased sample is simply obtained by replacing any "illegal" step, which would violate the self avoidance constraint, by a random "legal" one, provided such a legal step exists. More generally, assume we want to simulate a distribution in which each configuration S is weighted by a (Boltzmann-) weight $Q(S)$, so that for any observable A one has $\langle A\rangle = \sum_S A(S)Q(S)/\sum_S Q(S)$. If we sample unevenly with probability $p(S)$, then we must compensate this by giving a weight $W(S) \propto Q(S)/p(S)$,

$$\langle A\rangle = \lim_{M\to\infty} \frac{\sum_{i=1}^{M} A(S_i)Q(S_i)/p(S_i)}{\sum_{i=1}^{M} Q(S_i)/p(S_i)} \equiv \lim_{M\to\infty} \frac{1}{M}\sum_{i=1}^{M} A(S_i)W(S_i). \tag{11.14}$$

This is called the generalized Rosenbluth method. If $p(S)$ were chosen close to $Q(S)$, this would lead to importance sampling and obviously would be very efficient. But in general this is not possible, and Eq. (11.14) suffers from the problem that the sum is dominated by very few events with high weight.

Consider now a lattice chain of length $N+1$ with self avoidance and with nearest neighbor interaction $-\epsilon$ between unbonded neighbors. In the original Rosenbluth method, $p(S)$ is then a product,

$$p(S) \propto \prod_{n=1}^{N} \frac{1}{m_n}, \tag{11.15}$$

where m_n is the number of free neighbors in the n-th step, i.e. the number of possible lattice sites where to place the n-th monomer (monomers are labeled $n = 0, 1, \ldots N$).

Similarly, $Q(S)$ is a product,

$$Q(S) = e^{-\beta E} = e^{-\beta \sum_n E_n} = \prod_{n=1}^{N} e^{-\beta E_n}, \tag{11.16}$$

where $\beta = 1/kT$ and $E_n = -\epsilon \sum_{k=0}^{n-1} \Delta_{kn}$ is the energy of the n-th monomer in the field of all previous ones ($\Delta_{kn} = 1$ if and only if monomers k and n are neighbors and non-bonded, otherwise $\Delta_{kn} = 0$).

Obviously, $p(S)$ favors compact configurations where monomers have only few free neighbors. This renders the Rosenbluth method unsuitable for long chains, except near the collapse ("theta") point where simulations with $N \leq 1000$ are feasible on the simple cubic lattice [12]. In general we should find ways to modify the sampling so that "good" configurations are sampled more frequently, and "bad" ones less. The key to this is the product structure of the weights (with $w_n \equiv m_n e^{-\beta E_n}$)

$$W(S_i) = \frac{Q(S_i)/p(S_i)}{M^{-1} \sum_{k=1}^{M} Q(S_k)/p(S_k)} = \frac{1}{\hat{Z}_N} \prod_{n=1}^{N} w_n(S_i) \tag{11.17}$$

implied by Eqs. (11.15) and (11.16). Here, $\hat{Z}_N = M^{-1} \sum_{k=1}^{M} Q(S_k)/p(S_k)$ is an estimate of the partition sum. A similar product structure holds in practically all interesting cases.

We can thus watch how the weight builds up while the chain is constructed step by step. If the partial weight (from now on we drop the dependence on S)

$$W_n = \hat{Z}_n^{-1} \prod_{j=1}^{n} w_j, \qquad 1 < n < N, \tag{11.18}$$

gets too large (i.e. is above some threshold W^+), we replace the configuration by k copies, each with weight W_n/k. Growth of one of these copies is continued, all others are placed on a stack for later use. As a consequence, configurations with high weight are simulated more often. To account for this and to keep the final result correct, the weight is reduced accordingly. The whole idea is applied recursively. Following Ref. [11] we call this *enrichment*. The opposite action, when W_n falls below another threshold W^- (*pruning*), is done stochastically: with probability 1/2 the configuration is killed and replaced by the top of the stack, while its weight is doubled in the other half of cases.

In this way PERM (the pruned-enriched Rosenbluth method [8]) gives a sample with exactly the right statistical weights, independently of the thresholds $W^{\pm}$, the selection probability $p(S)$, and the clone multiplicity k. But its efficiency depends strongly on good choices for these parameters. Notice that one has complete freedom in choosing them, and can even change them during a run. Fortunately, reasonably good choices are easy to find (more sophisticated choices needed at very low temperatures are discussed in Refs. [13, 14]). The guiding principle for $p(S)$ is that it should lead as closely as possible to the correct final distribution, so that pruning and enrichment are kept to a minimum. This is also part of the guiding principles for $W^{\pm}$. In addition, W^+ and W^- have to be chosen such that roughly the same effort is spent on simulating any

part of the configuration. For polymers this means that the sample size should neither grow nor decrease with chain length n. This is easily done by adjusting W^+ and W^- "on the fly". We now show a listing for the central part of the PERM algorithm (after [8]), $\mathbf{x}$ denotes the current position of the head of the chain and n the current chain length:

```
procedure PERM-Step (x, n)
begin
  choose x' near x w density ρ(x'-x);        simplest case: ρ(x) = 1/m_n δ_{|x|,1}
  w_n := c_n ρ(x' - x)^{-1} exp(-E(x')/k_B T); if c_n=const.→grand canonical
  W_n := W_{n-1} w_n;
  begin                                      do statistics
    Z_n := Z_n + W_n;                        partition function
    R2_n := R2_n + x'^2 W_n;                 end-to-end distance
    t := t + 1;                              total number of subroutine calls
    etc.
  end
  if n < N_max and W_n > 0 then
  begin
    W^+ := c^+ Z_n/Z_1;                      adaption of W^+ (optional)
    W^- := c^- Z_n/Z_1;                      adaption of W^- (optional)
    if W_n > W^+ then
    begin
      W_n := W_n/2;
      PERM-Step(x', n + 1);
      PERM-Step(x', n + 1);                  enrichment!
    end
    else if W_n < W^- then
    begin
      W_n := W_n * 2;                        prune w. prob. 1/2
      draw ξ uniform ∈ [0, 1];
      if (ξ < 1/2) then
        PERM-Step(x', n + 1);
    end
    else
      PERM-Step(x', n + 1);                  normal Rosenbluth step
  end
  return
end
```

The subroutine PERM-Step is called from the main routine with arguments $\mathbf{x} = 0$, $n = 1$. W_n, c_n, Z_n and $R2_n$ are global arrays indexed by the chain length. W_n is the current weight, c_n a reweighting factor, which allows the simulation of different statistical ensembles, Z_n the estimate of the partition function and $R2_n$ the sum of the mean squared end-to-end distance, i.e. $R2_n/Z_n$ is the average. $N_{\max}$, t, c^+ and

c^-, are global scalar variables, where $N_{\max}$ is the maximum chain length, t counts the total number of subroutine calls and c^+/c^- control the adaptation of the weights. Without adaptation, the lines involving c^+ and c^- can be dropped, and then W^+ and W^- are global scalars. In more sophisticated implementations, ρ, c^+ and c^- will depend on n and/or on the configuration of the monomers with indices $n' < n$. Good choices for these functions may be crucial for the efficiency of the algorithm, but are not important for its correctness. To compute the energy $E(\mathbf{x}')$ of the newly placed monomer in the field of the earlier ones, one can use either bit maps (in lattice models with small lattices), hashing or neighbor lists. If none of these are suitable, $E(\mathbf{x}')$ has to be computed as an explicit sum over all earlier monomers.

In selecting the good and killing the bad, PERM is similar to evolutionary and genetic algorithms [15], see Chap. 8, to population based growth algorithms for chain polymers [16, 17, 18, 19], to diffusion type quantum Monte Carlo algorithms [20], and to the "go with the winners" strategy of Ref. [21]. The main difference with the first three groups of methods is that one does not keep the entire population of instances simultaneously in computer memory. Indeed, even on the stack one does not keep copies of good configurations but only the steps involved in constructing the configurations and flags telling us when to make a copy [8]. In genetic algorithms, keeping the entire population in memory is needed for cross-overs, and it allows a one-to-one competition between instances. But in our case this is not needed since every instance can be compared with the average behavior of all others. The same would be true for diffusion type quantum Monte Carlo simulations. The main advantage of our strategy is that it reduces computer memory enormously. This, together with the surprisingly easy determination of the thresholds $W^{\pm}$, could make PERM also a very useful strategy for quantum Monte Carlo simulations.

11.5 Protein Folding

Protein folding [22] is one of the outstanding problems in mathematical biology. It is concerned with the problem of how a given sequence of amino acids assumes precisely that geometrical shape which is biologically useful. Currently it is much easier to find coding DNA (and, thus, amino acid) sequences than to find the corresponding structures. Thus, solving the folding problem would be a major break-through in understanding the biochemistry of the cell, and in designing artificial proteins. In this section we present only the most straightforward direct approach: given a sequence, a molecular potential and no other information, find the ground state and the equilibrium state at physiological temperatures. Note that we are not concerned with the kinetics of folding, but only in the final outcome. Also, we will not address the problems of how to find good molecular potentials, and what is the proper level of detail in describing proteins. Instead, we discuss simple coarse-grained models which have been proposed in the literature and have become standards in testing the efficiency of folding algorithms.

The models we discuss are heteropolymers which live on 3d or 2d regular lattices. They are self-avoiding chains with attractive or repulsive interactions between neighboring

non-bonded monomers. These interactions can have continuous distributions [23], but the majority of authors have considered only two kinds of monomers. In the HP model [24, 25] they are hydrophobic (H) and polar (P), with $(\epsilon_{HH}, \epsilon_{HP}, \epsilon_{PP}) = -(1,0,0)$. Since this leads to highly degenerate ground states, alternative models were proposed, e.g. $\vec{\epsilon} = -(3,1,3)$ [26] and $\vec{\epsilon} = -(1,0,1)$ [27].
The algorithms that were applied in [28, 29] were variants of the pruned-enriched Rosenbluth method (PERM) described in the last section and we present here a few of the impressive results obtained in this way, improving substantially on previous work.

2d HP model

Two chains with $N = 100$ were studied in [30]. The authors claimed that their native configurations were compact, fitting exactly into a 10×10 square, and had energies -44 and -46:
Sequence 1:
$\overline{P_6HPH_2P_5}H_3PH_5PH_2P_2(P_2H_2)_2PH_5PH_{10}PH_2PH_7P_{11}H_7P_2HPH_3P_6HPH_2$
Sequence 2:
$\overline{P_3H_2P_2H_4P_2H_3}(PH_2)_3H_2P_8H_6P_2H_6P_9HPH_2PH_{11}P_2H_3PH_2PHP_2HPH_3P_6H_3$

In Fig. 11.6 we show the the respective *proposed* ground-state structures. These conformations were found by a specially designed MC algorithm which was claimed to be particularly efficient for compact configurations. We do not discuss the method here. For these two HP chains by applying the PERM algorithm at low temperatures, Grassberger et al. [29] found (within ca. 40 hours of CPU time) several compact states that had energies *lower* than those of the compact putative ground states proposed in [30], namely with $E = -46$ for sequence 1 and $E = -47$ for sequence 2. Moreover, they found (again within 1-2 days of CPU time) several non-compact configurations with energies even lower: $E = -47$ and $E = -48$ for sequences 1 and 2, respectively. Forbidding non-bonded HP pairs, Grassberger et al. [28, 29] obtained even $E = -49$ for sequence 2. Figure 11.6 shows representative non-compact structures with these energies. These results reflect the well-known property that HP sequences (and those of other models) usually have ground states that are not maximally compact, see, e.g. [31], although there is a persistent prejudice to the contrary [27, 30, 32].

3d modified HP model

A most interesting case is a 2-species 80-mer with interactions $(-1,0,-1)$ studied first in [27]. These particular interactions were chosen instead of the HP choice $(-1,0,0)$ because it was hoped that this would lead to compact configurations. Indeed the sequence [27, 33]

$$PH_2P_3(H_3P_2H_3P_3H_2P_3)_3H_4P_4(H_3P_2H_3P_3H_2P_3)H_2$$

was specially designed to form a "four helix bundle" which fits perfectly into a $4 \times 4 \times 5$ box, see Fig. 11.7. Its energy in this putative native state is -94. Although the authors

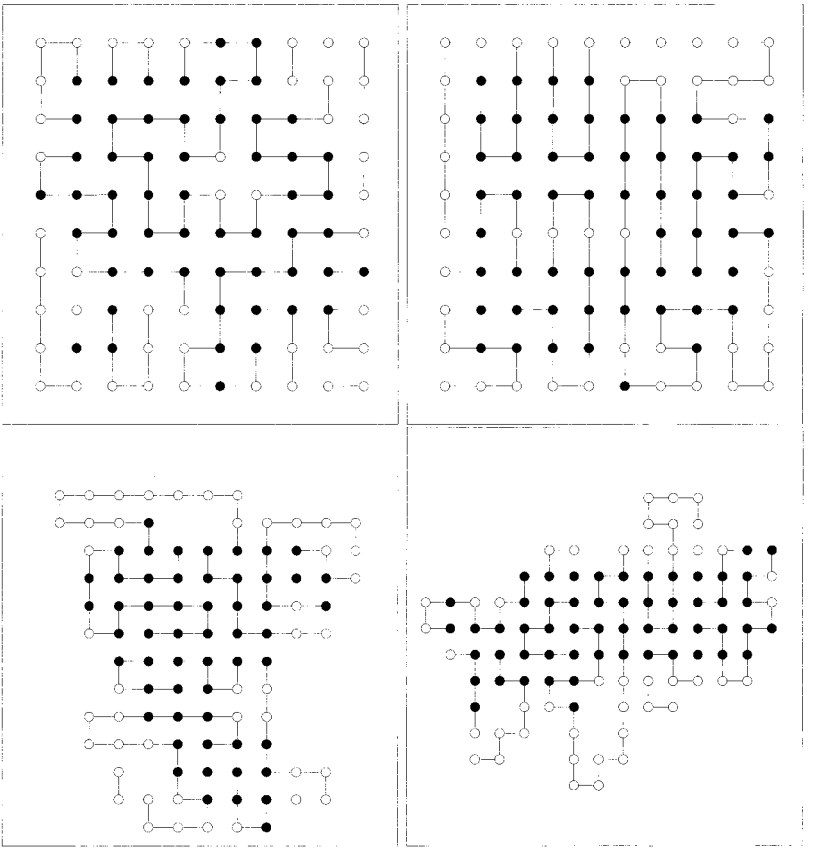


Figure 11.6: Top: Putative compact native structure of sequence 1 (left) with $E = -44$ and sequence 2 (right) with $E = -46$ according to [30]; (filled circle) H monomers, (open circle) P monomers. Bottom: One of the (non-compact) lowest energy sequences for sequence 1, left, with $E = -47$ and sequence 2, right, with $E = -49$.

of [27] used highly optimized codes, they were not able to recover this state by MC. Instead, they reached only $E = -91$. Supposedly a different state with $E = -94$ was found in [30], but figure 10 of this paper, which it is claimed shows this configuration, has a much higher value of E. Configurations with $E = -94$ but slightly different from that in [27] and with $E = -95$ were found in [33] by means of an algorithm similar to that in [30]. For each of these low energy states the author needed about one week of CPU time on a Pentium.

Grassberger et al. [28, 29] applied the PERM algorithm to the aforementioned system. Even without much tuning the algorithm gave $E = -94$ after a few hours, but it did not stop there. After a number of rather disordered configurations with successively lower energies, the final candidate for the native state has $E = -98$. It again has a

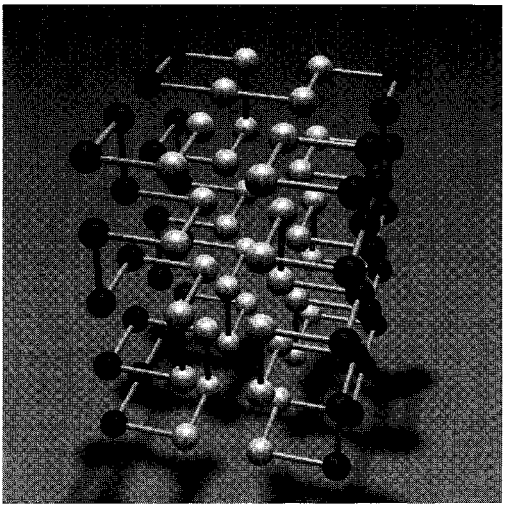


Figure 11.7: Putative native state of the "four helix bundle" sequence, as proposed by O'Toole and A. Panagiotopoulos [27]. It has $E = -94$, fits into a rectangular box, and consists of three homogeneous layers. Structurally, it can be interpreted as four helix bundles.

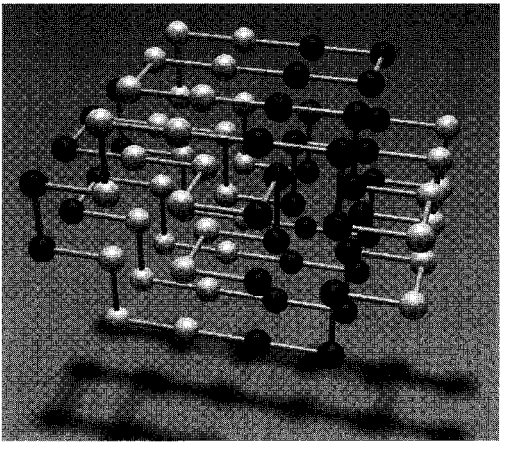


Figure 11.8: Conformation of the "four helix bundle" sequence with $E = -98$. Grassberger et al. [28, 29] proposed that this is the actual ground state. Its shape is highly symmetric although it does not fit into a rectangular box. It is not degenerate except for a flipping of the central front $2 \times 2 \times 2$ box.

highly symmetric shape, although it does not fit into a $4 \times 4 \times 5$ box, see Fig. 11.8. It has two-fold degeneracy (the central $2 \times 2 \times 2$ box in the front of Fig. 11.8 can be flipped), and both configurations were actually found in the simulations. The optimal

temperature for the ground-state search in this model is $\beta = 1/kT \approx 2.0$.
A surprising result is that the monomers are arranged in four homogeneous layers in Fig. 11.8, while they had formed only three layers in the putative ground state of Fig. 11.7. Since the interaction should favor the segregation of different type monomers, one might have guessed that a configuration with a smaller number of layers should be favored. We see that this is outweighed by the fact that both monomer types can form large double layers in the new configuration. Again, our new ground state is not "compact" in the sense of minimizing the surface, and hence it also disagrees with the widespread prejudice that native states are compact.
To classify a ground state, one can also just consider which pairs of monomers are adjacent to each other, i.e. abstract from the full spatial structure. The resulting relation is called a *secondary structure* and can be written in form of a *contact matrix*. In terms of secondary structure, the new ground state is fundamentally different from the putative ground state of Ref. [30]. While the new structure (Fig. 11.8) is dominated by so called β *sheets*, which can most clearly be seen in the contact matrix, the structure in Fig. 11.7 is dominated by *helices*.
This example demonstrates that the pruned-enriched Rosenbluth method can be very effectively applied to protein structure prediction in simple lattice models. It is suited for calculating statistical properties and is very successful in finding native states. In all cases it did better than any previous MC method, and in many cases it found lower states than those which had previously been conjectured to be native.

Bibliography

[1] G. Dueck and T. Scheuer, *J. Comp. Phys.* **90**, 161 (1990)

[2] G. Dueck, *J. Comp. Phys.* **104**, 86 (1993)

[3] S. Kirkpatrick, C.D. Gelatt, and M.P. Vecchi, *Science* **220**, 671 (1983)

[4] N. Metropolis, A.W. Rosenbluth, M.N. Rosenbluth, A.H. Teller, and E. Teller, *J. Chem. Phys.* **21**, 1087 (1953)

[5] E. Marinari and G. Parisi, *Europhys. Lett.* **19**, 451 (1992)

[6] B. Berg and T. Neuhaus, Phys. Lett B **267**, 249 (1991); *Phys. Rev. Lett.* **68**, 9 (1992)

[7] K. Hukushima and K. Nemoto, *J. Phys. Soc. Jpn.* **65**, 1604 (1996)

[8] P. Grassberger, *Phys. Rev.* **E 56**, 3682 (1997)

[9] P. Grassberger and H. Frauenkron, in: P. Grassberger, G.T. Barkema, and W. Nadler. (ed.), *Workshop on Monte Carlo Approach to Biopolymers and Protein Folding*, (World Scientific, Singapure 1998)

[10] M.N. Rosenbluth and A.W. Rosenbluth, *J. Chem. Phys.* **23**, 356 (1955)

[11] F.T. Wall and J.J. Erpenbeck, *J. Chem. Phys.* **30**, 634, 637 (1959)

[12] W. Bruns, *Macromolecules* **17**, 2826 (1984) .

[13] H. Frauenkron, U. Bastolla, E. Gerstner, P. Grassberger, and W. Nadler, in: P. Grassberger, G.T. Barkema, and W. Nadler. (ed.), *Workshop on: Monte Carlo Approach to Biopolymers and Protein Folding*, (World Scientific, Singapure 1998)

[14] U. Bastolla and P. Grassberger, in: P. Grassberger, G.T. Barkema, and W. Nadler. (ed.), *Workshop on: Monte Carlo Approach to Biopolymers and Protein Folding*, (World Scientific, Singapure, 1998)

[15] J. Holland, *Adaptation in Natural and Artificial Systems*, (The University of Michigan, Ann Arbor, MI 1992)

[16] T. Garel and H. Orland, *J. Phys.* **A 23**, L621 (1990)

[17] P.G. Higgs and H. Orland, *J. Chem.* **95**, 4506 (1991)

[18] B. Velikson, T. Garel, J.-C. Niel, H. Orland, and J.C. Smith, *J. Comput. Chem.* **13**, 1216 (1992)

[19] H. Orland, in: P. Grassberger, G.T. Barkema, and W. Nadler. (ed.), *Workshop on: Monte Carlo Approach to Biopolymers and Protein Folding*, (World Scientific, Singapure, 1998)

[20] C.J. Umrigar, M.P. Nightingale, and K.J. Runge, J. *Chem. Phys.* **99**, 2865 (1993)

[21] D. Aldous and U. Vazirani, in: *Proc. 35th IEEE Sympos. on Foundations of Computer Science* (1994)

[22] T.E. Creighton (ed.), *Protein Folding*, (Freeman, New York, 1992)

[23] D.K. Klimov and D. Thirumalai, *Prpteins: Struct., Fnct. Gen.* **26**, 411 (1996); sequences are available from http://www.glue.umd.edu/ klimov.

[24] K.A. Dill, *Biochemistry* **24**, 1501 (1985)

[25] K.F. Lau and K.A. Dill, *Macromolecules* **22**, 3986 (1989); *J. Chem. Phys.* **95**, 3775 (1991); D. Shortle, H.S. Chan, and K.A. Dill, *Protein Sci.* **1**, 201 (1992)

[26] N.D. Socci and J.N.Onuchic, *J. Chem. Phys.* **101**, 1519 (1994)

[27] E. O'Toole and A. Panagiotopoulos, *J. Chem. Phys.* **97**, 8644 (1992)

[28] H. Frauenkron, U. Bastolla, E. Gerstner, P. Grassberger, and W. Nadler, *Phys. Rev. Lett.* **80** 3149 (1998)

[29] U. Bastolla, H. Frauenkron, E. Gerstner, P. Grassberger, and W. Nadler, *Proteins: Struct., Fnct., Gen.* **32**, 52 (1998)

[30] R. Ramakrishnan, B. Ramachandran, and J.F. Pekney, *J. Chem. Phys.* **106**, 2418 (1997)

[31] K. Yue K.M. Fiebig, P.D. Thomas, H.S. Chan, E.I. Shakhnovich, and K.A. Dill, *Proc. Natl. Acad. Sci. USA* **92**, 325 (1995)

[32] E.I. Shakhnovich and A.M. Gutin, *J. Chem. Phys.* **93** 5967 (1990); A.M. Gutin and E.I. Shakhnovich, *J. Chem. Phys.* **98** 8174 (1993)

[33] J.M. Deutsch, *J. Chem. Phys.* **106**, 8849 (1997)

12 Branch-and-bound Methods

As we have already seen in Chaps. 2 and 3, it seems to be impossible to invent an algorithm which solves an NP-hard problem in polynomial time. Methods only exist where the worst-case running time increases exponentially with the size of the system, where the size is measured by the minimal length of a string you need to encode the problem for a computer.

Recently, NP-hard problems have reattracted a lot of attention in both communities of physicists and theoretical computer scientists. The reason is that instead of studying the worst-case complexity, the problems are now considered for ensembles of random instances usually characterized by only a few parameters. Examples for such parameters are the relative number α of clauses in a boolean CNF formula with respect to the number of different variables (see Chap. 2) or the number $c = M/N$ of edges M divided by the number $N$of vertices for a graph. It turns out that, although the *worst-case* time complexity is always exponential, the *typical-case* complexity in the random ensembles may depend on the parameters. This means there are regions in parameter space where the instances can be solved typically in polynomial time, while in other regions typical instances indeed require exponential time. This viewpoint is more suitable for studying real-world optimization problems, rather than the academic approach of investigating worst-case behavior. For this reason, the interest in the computer-science community in this subject is currently increasing. An introduction to the field can be found in [1].

Physicists are interested in such types of problems for two further reasons. First, the change in the typical-case complexity coincides with phase transitions on the random ensembles. Thus, we have a phenomenon which is quite familiar to physicists. Second, it turns out that with analytical methods originating from statistical physics, it is very often possible to obtain much more information or even more accurate results than by applying traditional mathematical approaches.

In this chapter, the (for most physicists) new subject will be introduced via a presentation of the *vertex-cover* (VC) problem. Other problems, where statistical physics methods have been applied recently, are the K-SAT problem [2] and the number-partitioning problem [3], for a review see Ref. [4, 5].

The chapter is organized as follows. First some basic definitions are given. Next, three algorithms for solving VC are presented. Some basic results are shown in the third section.

12.1 Vertex Covers

The vertex-cover problem is an NP-hard problem from graph theory, its basic definition has already been introduced briefly in Chap. 3. Here, we recall the definition and introduce further notions.

Vertex covers are defined as follows. Take any undirected graph $G = (V, E)$ with N vertices $i \in V = \{1, 2, ..., N\}$ and M edges $(i, j) \in E \subset V \times V$. Please note that in this case (i, j) and (j, i) denote both the same edge. We consider subsets $V_{vc} \subset V$. The set V_{vc} is a *vertex cover iff* for all edges $(i, j) \in V$ at least one of its endpoints is a member of the set: $i \in V_{vc}$ or $j \in V_{vc}$. Also please note that $V_{vc} = V$ is always a vertex cover. Furthermore, vertex covers are not unique. For example, for each vertex $i \in V$, the set $V_{vc} = V \setminus \{i\}$ is a vertex cover as well.

The term "*covered*" will be used in several circumstances later on. The vertices i with $i \in V_{vc}$ are called *covered*, and *uncovered* for $i \notin V_{vc}$. One can imagine that some *covering marks* are placed at the vertices belonging to V_{vc}. Analogously, an edge $(i, j) \in E$ is called *covered* iff at least one of its end-vertices is *covered*, $i \in V_{vc}$ or $j \in V_{vc}$. Thus, comparing with the definition given above, the set V_{vc} is a vertex cover iff all edges of the graph are covered. In this case, the graph is called *covered* as well.

Example: Vertex cover

Please consider the graph shown in the left half of Fig. 12.1. Vertices 1 and 2 are *covered* ($V_{vc} = \{1, 2\}$), while the other vertices 3, 4, 5 and 6 are *uncovered*. Thus, edges $(1, 3)$, $(1, 4)$ and $(2, 3)$ are *covered* while edges $(3, 4)$, $(3, 5)$, $(4, 6)$ and $(5, 6)$ are *uncovered*. Hence, the graph is not *covered*.

In the right half of Fig. 12.1 also vertices 4 and 5 are *covered*. Thus, edges $(3, 4)$, $(3, 5)$, $(4, 6)$ and $(5, 6)$ are now *covered* as well. This means all edges are *covered*, i.e. the graph is covered by $V_{vc} = \{1, 2, 4, 5\}$, thus V_{vc} is a vertex cover. □

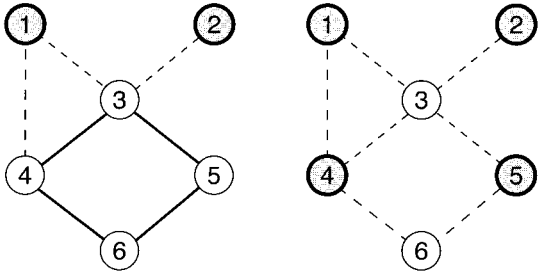


Figure 12.1: Graphs and covers. *Covered* vertices are shown in bold and dark, *covered* edges are indicated by dashed lines. Left: a partially covered graph. Vertex 1 and 2 are *covered*. Thus, edges $(1, 3)$, $(1, 4)$, and $(2, 3)$ are *covered*. Right: by also covering also vertices 4 and 5, the graph is covered.

The *vertex-cover decision problem* asks whether there are VCs of fixed given cardinality $X = |V_{vc}|$, we define $x = X/N$. In other words we are interested if it is possible to cover all edges of G by covering xN suitably chosen vertices, *i.e.* by distributing xN *covering marks* among the vertices. To measure the extent a graph is not covered, we introduce an *energy* $e(G,x) = E(G,x)/N$ with

$$E(G,x) = \min\{\text{number of } \textit{uncovered} \text{ edges when covering } xN \text{vertices}\} \quad (12.1)$$

Thus, a graph G is coverable by xN vertices if $e(G,x) = 0$. This means that you can answer the VC decision problem by first solving a minimization problem, and then testing whether the minimum $e(G,x)$ is zero or not.
For the preceding case, the energy was minimized with fixed X. The decision problem can also be solved by solving another *optimization problem*: for a given graph G it asks for a *minimum vertex cover* V_{vc}, i.e. a vertex cover of minimum size $X_c(G) = |V_{vc}|$. Thus, here the number of vertices in the vertex cover is minimized, while the energy is kept at zero. The answer to the vertex cover decision problem is "yes", if $X \geq X_c$. Also minimum vertex covers may not be unique. In case several vertex covers $V_{vc}^{(1)}$, ..., $V_{vc}^{(K)}$ exist, each with the same cardinality X (not necessarily minimum vertex covers), a vertex i is called a *backbone* vertex, if it is either a member of all covers ($\forall k = 1,\ldots,K : i \in V_{vc}^{(k)}$) or else a member of no cover ($\forall k = 1,\ldots,K : i \notin V_{vc}^{(k)}$). These vertices can be regarded as *frozen* in all vertex-cover configurations. All other vertices are called *non-backbone* vertices.

Example: Minimum vertex cover

For the graph from Fig. 12.1, in Fig. 12.2 all three minimum vertex covers ($X_c = 3$) are shown. Vertices 2 (always *uncovered*) and 3 (always *covered*) belong to the backbone, while all other vertices are non-backbone.

It is straightforward to show that vertex 3 must be a member of all minimum vertex covers. Assume that vertex 4 is not *covered*. Since all edges have to be *covered*, for all edges incident to vertex 3, the second end vertices have to be *covered*. Thus, vertices 1, 2, 4 and 5 have to be *covered*, i.e. more vertices than in the minimum vertex cover, which has size $X_c = 3$.

□

In order to be able to speak of typical or average cases, we have to introduce an ensemble of graphs. We investigate random graphs $G_{N,cN}$ with N vertices and cN edges (i,j) which are drawn randomly and independently. For a complete introduction to the field see [6]. One major result is that for small concentrations $c < 0.5$, each graph breaks up into small components, where the size $N_{\max}$ of the largest components is of the order $O(\ln N)$. This means that the fraction $N_{\max}/N$ of the vertices in the largest component converges with increasing graph size as $\ln N/N$ to zero. For concentrations c larger than the critical concentration $c_{\text{crit}} = 0.5$, a finite fraction of all vertices are collected in the largest component, called the giant component. One says the graph *percolates*, c_{crit} is called the percolation threshold.

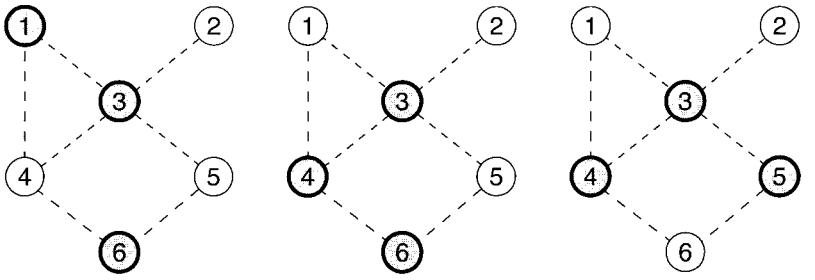


Figure 12.2: All three minimum vertex covers of the graph from the preceding example.

Turning back to VC, when the number xN of covering marks is increased (c is kept constant), the model is expected to undergo an uncoverable-coverable transition. Being able only to cover a small number xN of vertices, it is very unlikely that one will be able to cover all edges of a random graph. With increasing size of the cover set this will be become more and more likely, while for $X = N$ it is certain that all edges are covered. For a given graph G, we will denote the minimum fraction of covered vertices necessary to cover the whole graph by $x_c(G) = X_c(G)/N$. The value of $x_c(G)N$ is related to the energy $e(G, x)$, it is just the smallest number x where the energy $e(G)$ vanishes. The average of $e(G)$ over all random graphs as a function of x, will be denoted $e(x)$. Later we will see that by taking the thermodynamic limit $N \to \infty$ the size $x_c = X_c/N$ of the minimum cover set will only depend on the concentration c of the edges, i.e. $x_c = x_c(c)$.
Using probabilistic tools, rigorous lower and upper bounds for this threshold [7] and the asymptotic behavior for large connectivities [8] have been deduced. Recently, the problem has investigated using a statistical physics approach [9] and the results have been improved drastically. Up to an average concentration $c = e/2 \approx 1.359$ the transition line is given exactly by

$$x_c(c) = 1 - \frac{2W(2c) + W(2c)^2}{2c}, \tag{12.2}$$

where $W(c)$ is the Lambert-W-function defined by $W(c)\exp(W(c)) = c$. The transition along with the bounds is shown in the phase diagram in Fig. 12.3. For $x > x_c(c)$, the problem is coverable with probability one, for $x < x_c(c)$ the available covering marks are not sufficient. For higher connectivities no exact result for $x_c(c)$ could be obtained, but the result given by Eq. 12.2 is a good approximation, unless c grows too large. Please note that the region, where the exact result has been obtained, extends fairly well into the percolating regime, since the percolation threshold is $c_{crit} = 0.5 < 1.359$.
In Sec. 12.3, the analytic result is compared with data from numerical simulations, we will see that the agreement is amazing. But before we do that, in the next section, some algorithms to solve the VC are presented.

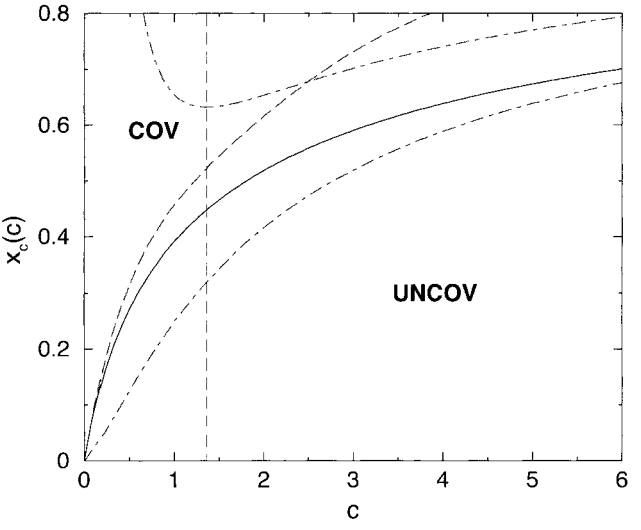


Figure 12.3: Phase diagram. Fraction $x_c(c)$ of vertices in a minimal vertex cover as a function of the average connectivity c. For $x > x_c(c)$, almost all graphs have covers with xN vertices, while they have almost definitely no cover for $x < x_c(c)$. The solid line shows the result from statistical physics. The upper bound of Harant is given by the dashed line, the bounds of Gazmuri by the dash-dotted lines. The vertical line is at $c = e/2 \approx 1.359$.

12.2 Numerical Methods

In this section, two exact numerical methods to solve the vertex-cover problem are presented. Please note that there are two ways to treat the problem. First, you can look only for minimum vertex covers, i.e. you ask for the minimum number $x_c(G)N$ of vertices you have to cover for a given graph G to ensure that all edges of G are covered. The second way is to fix an arbitrary number $X = xN$ of vertices to be covered and ask for the maximum number of edges (or minimum number of uncovered edges) which can be covered under that restriction. We always present the algorithms in a way that they are suitable for the first kind of problem. Afterwards, how the methods can be changed to treat problems of the second kind is outlined.

Before explaining the exact algorithms, we first introduce a fast heuristic, which is utilized within both exact methods. The heuristic can be applied stand-alone as well. In this case only an approximation of the true minimum vertex cover is calculated, which is found to differ only by a few percent from the exact value. All methods can easily be implemented via the help of the LEDA library [10] which offers many useful data types and algorithms for linear algebra and graph problems.

The basic idea of the heuristic is to cover as many edges as possible by using as few

vertices as are necessary. Thus, it seems favorable to cover vertices with a high degree. This step can be iterated, while the degree of the vertices is adjusted dynamically by removing edges which are *covered*. This leads to the following greedy algorithm, which returns an approximation of the minimum vertex cover V_{vc}, the size $|V_{vc}|$ is an upper bound of the true minimum vertex-cover size:

algorithm greedy-cover(G)
begin
  initialize $V_{vc} = \emptyset$;
  while there are *uncovered* edges **do**
  begin
    take one vertex i with the largest current degree d_i;
    mark i as *covered*: $V_{vc} = V_{vc} \cup \{i\}$;
    remove from E all edges (i, j) incident to i;
  end;
  return(V_{vc});
end

Example: Heuristic

To demonstrate, how the heuristic operates, we consider the graph shown in Fig. 12.4. In the first iteration, vertex 3 is chosen, because it has degree 4, which is the largest in the graph. The vertex is covered, and the incident edges $(1,3)$, $(2,3)$, $(3,4)$ and $(3,5)$ are removed. Now, vertices 6 and 7 have the highest degree 3. We assume that in the second iteration vertex 6 is covered and vertex 7 in the third iteration. Then the algorithms stops, because all edges are *covered*.

□

In the preceding example we have seen that the heuristic is sometimes able to find a true minimum vertex cover. But this is not always the case. In Fig. 12.5 a simple counter example is presented, where the heuristic fails to find the true minimal vertex cover. First the algorithm covers the root vertex, because it has degree 3. Thus, three additional vertices have to be subsequently covered, i.e. the heuristic covers 4 vertices. But, the minimum vertex cover has only size 3, as indicated in Fig. 12.5.
The heuristic can be easily altered for the case where the number X of *covered* vertices is fixed and it is asked for a minimum number of uncovered edges. Then the iteration has to stop as well, when the size of the cover set has reached X. In case a vertex cover is found, before X vertices are covered, arbitrary vertices can be added to the vertex-cover set V_{vc} until $|V_{vc}| = X$.
So far we have presented a simple heuristic to find approximations of minimum vertex covers. Next, two exact algorithms are explained: *divide-and-conquer* and *branch-and-bound*, which both incorporate the heuristic to gain a high efficiency. Without the heuristic, the algorithms would still be exact, but slower running.

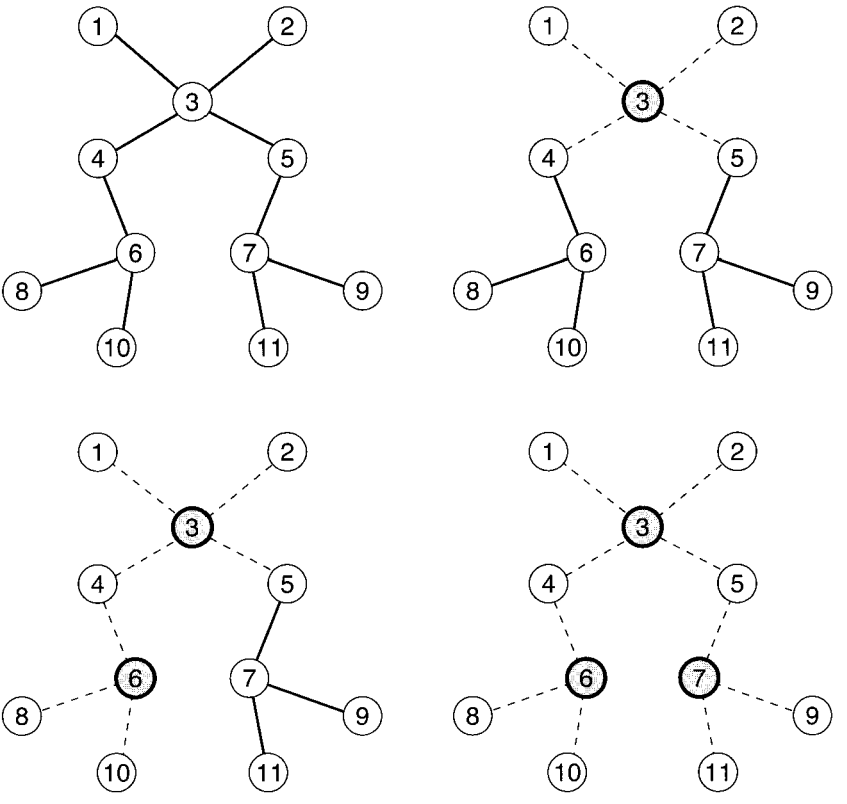


Figure 12.4: Example of the cover heuristic. Upper left: initial graph. Upper right: graph after the first iteration, vertex has been covered (shown in bold) and the incident edges removed (shown with dashed line style). Bottom: graph after second and third iteration.

The basic idea of both methods is as follows, again we are interested first in a VC of minimum size: as each vertex is either *covered* or *uncovered*, there are 2^N possible configurations which can be arranged as leafs of a binary (backtracking) tree, see Fig. 12.6. At each node, the two subtrees represent the subproblems where the corresponding vertex is either *covered* ("left subtree") or *uncovered* ("right subtree"). Vertices, which have not been touched at a certain level of the tree are said to be *free*. Both algorithms do not descend further into the tree when a cover has been found, i.e. when all edges are *covered*. Then the search continues in higher levels of the tree (backtracking) for a cover which possibly has a smaller size. Since the number of nodes in a tree grows exponentially with system size, algorithms which are based on backtracking trees have a running time which may grow exponentially with the system size. This is not surprising, since the minimal-VC problem is NP-hard, so all exact

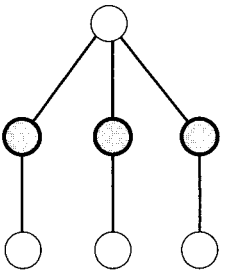


Figure 12.5: A small sample graph with minimum vertex cover of size 3. The vertices belonging to the minimum V_{vc} are indicated by dark/bold circles. For this graph the heuristic fails to find the true minimum cover, because is starts by covering the root vertex, which has the highest degree 3.

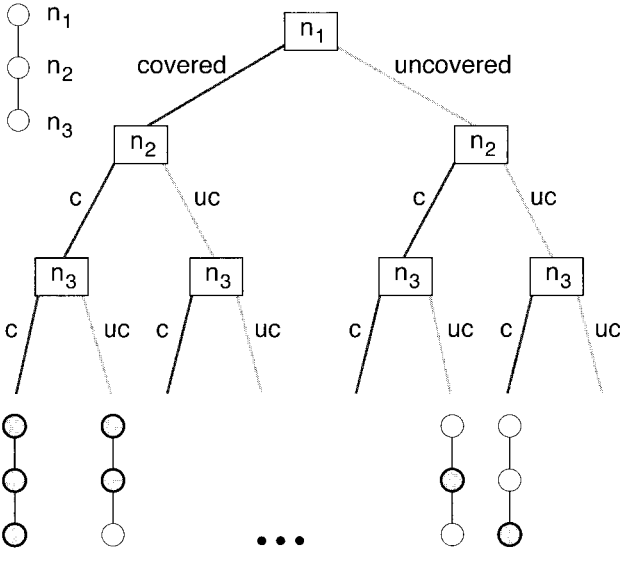


Figure 12.6: Binary backtracking tree for the VC. Each node of the backtracking tree corresponds to a vertex which is either *covered* ("left subtree") or *uncovered* ("right subtree").

methods exhibit an exponential growing worst-case time complexity.

To decrease the running time, both algorithms make use of the fact that only full vertex covers are to be obtained. Therefore, when a vertex i is marked *uncovered*, all neighboring vertices can be *covered* immediately. Concerning these vertices, only the left subtrees are present in the backtracking tree.

The divide-and-conquer [11] approach is based on the fact that a minimum VC of a graph, which consists of several independent connected components, can be obtained by combining the minimum covers of the components. Thus, the full task can be split into several independent tasks. This strategy can be repeated at all levels of the backtracking tree. At each level, the edges which have been covered can be removed from the graph, so the graph may split into further components. As a consequence, below the percolation threshold, where the size of the largest components is of the order $O(\ln N)$, the algorithm exhibits a polynomial running time. The divide-and-conquer approach reads as below, the given subroutine is called for each component of the graph separately, it returns the size of the minimum vertex cover, initially all vertices have state *free*.

algorithm divide-and-conquer(G)
begin
take one *free* vertex i with the largest current degree d_i;

mark i as *covered*; **comment** left subtree
$size_1 := 1$;
remove from E all edges (i, j) incident to i;
calculate all connected components $\{C_i\}$ of graph built by *free* vertices;
for all components C_i **do**
$size_1 := size_1 +$ **divide-and-conquer**(C_i);
insert all edges (i, j) which have been removed;

mark i as *uncovered*; **comment** right subtree;
$size_2 := 0$;
for all neighbors j of i **do**
begin
mark j as *covered*
remove from E all edges (j, k) incident to j;
end
calculate all connected components $\{C_i\}$;
for all components C_i **do**
$size_2 := size_2 +$ **divide-and-conquer**(C_i);
for all neighbors j of i **do**
mark j as *free*
insert all edges (j, k) which have been removed;

mark i as *free*;
if $size_1 < size_2$ **then**
return$(size_1)$;
else
return$(size_2)$;
end

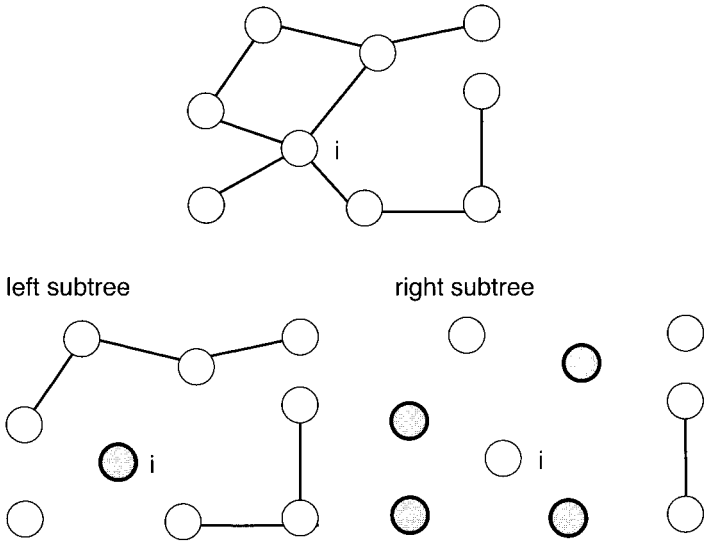


Figure 12.7: Example of how the divide-and-conquer algorithm operates. Above the graph is shown. The vertex i with the highest degree is considered. In the case where it is *covered* (left subtree), all incident edges can be removed. In the case where it is *uncovered*, (right subtree) all neighbors have to be *covered* and all edges incident to the neighbors can be removed. In both cases, the graph may split into several components, which can be treated independently by recursive calls of the algorithm.

The algorithm can be easily extended to record the cover sets as well or to calculate the degeneracy. In Fig. 12.7 an example is given of how the algorithm operates. The algorithm is able to treat large graphs deep in the percolating regime. We have calculated for example minimum vertex covers for graphs of size $N = 560$ with average edge density $c = 1.3$

For average edge densities larger than 4, the divide-and-conquer algorithm is too slow, because the graph only rarely splits into several components. Then a *branch-and-bound* approach [12, 13, 14] is favorable. It differs from the previous method by the fact that no independent components of the graph are calculated. Instead, some subtrees of the backtracking tree are omitted by introducing a *bound*, this saves a lot of running time. This is achieved by storing three quantities, assuming that the algorithm is currently at some node in the backtracking tree:

- The *best* size of the smallest vertex cover found in subtrees visited so far (initially $best = N$).
- X denotes the number of vertices which have been *covered* in higher levels of the tree.
- Always a table of *free* vertices ordered in descending current degree d_i is kept.

Thus, to achieve a better solution, at most $F = best - X$ vertices can be *covered* additionally in a subtree of the current node. This means it is not possible to cover more edges, than given by the sum $D = \sum_{l=1}^{F} d_l$ of the F highest degrees in the table of vertices, *i.e.* if some edges remain *uncovered*, the corresponding subtree can be omitted for sure. Please note that in the case that some edges run between the F vertices of the highest current degree, then a subtree may be entered, even if it contained no smaller cover.
The algorithm can be summarized as follows below. The size of the smallest cover is stored in *best*, which is passed by reference (i.e. the variable, not its value is passed). The current number of *covered* vertices is stored in variable X, please remember $G = (V, E)$:

```
algorithm branch-and-bound(G, best, X)
begin
  if all edges are covered then
  begin
    if X < best then best := X
    return;
  end;
  calculate F = best − X; D = Σ_{l=1}^{F} d_l;
  if D < number of uncovered edges then
    return;          comment bound;
  take one free vertex i with the largest current degree d_i;
  mark i as covered; comment left subtree
  X := X + 1;
  remove from E all edges (i, j) incident to i;
  branch-and-bound(G, best, X);
  insert all edges (i, j) which have been removed;
  X := X − 1;
  if (X > number of current neighbors) then
  begin              comment right subtree;
    mark i as uncovered;
    for all neighbors j of i do
    begin
      mark j as covered; X := X + 1;
      remove from E all edges (j, k) incident to j;
    end;
    branch-and-bound(G, best, X);
    for all neighbors j of i do
      mark j as free; X := X − 1;
    insert all edges (j, k) which have been removed;
  end;
  mark i as free;
  return;
end
```

Example: Branch-and-bound algorithm

Here we consider again the graph from Fig. 12.4. During the first descent into the backtracking tree, the branch-and-bound algorithm operates exactly like the heuristic. Iteratively vertices of highest current degree are taken, covered, and the incident edges removed. The recursion stops the first time when the graph is covered. This situation is shown in Fig. 12.2, where the graph and the corresponding current backtracking tree are displayed. Since $X = 3$ vertices have been covered, $best := 3$.

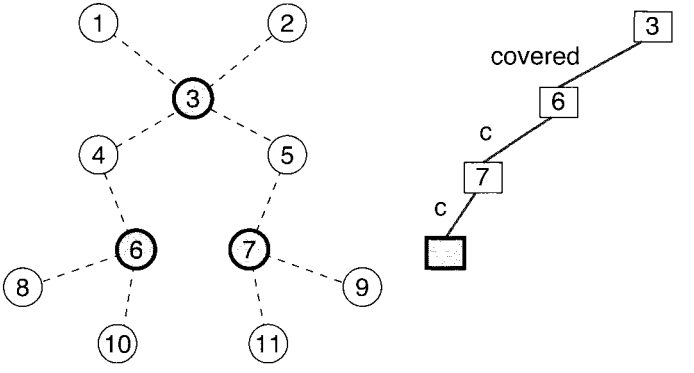


Figure 12.8: Example of the branch-and-bound algorithm. Result after the first full cover has been found. Left: graph, right: backtracking tree. In the graph, covered vertices are shown by bold and dark circles, covered edges indicated by dashed lines. The current node of the backtracking tree is highlighted as well, c=*covered*.

Then the algorithms returns to the preceding level of the backtracking tree. Vertex 7 is *uncovered*. Thus, since only full covers of the graph are desired, all its neighbors must be *covered*, namely vertices 5, 9 and 11. Again a cover of the whole graph has been obtained. Now $X = 4$ vertices have been covered, so no better optimum has been found, i.e. still $best := 3$.

For vertices 5, 9 and 11 only *covered* states had to be considered. Thus, the algorithm returns directly 3 levels in the backtracking tree. Then it returns one more level, since for vertex 7 both states have been considered. Next, vertex 6 is *uncovered* and its neighbors 4, 8 and 10 are covered ($X = 4$), see Fig. 12.2. Now three *uncovered* edges remain, i.e. the algorithm does not return immediately. Hence, the calculations for the bound are performed. $F = best - X = 3 - 4 = -1$ is obtained. This means no more vertices can be covered, i.e. $D = 0$ which is smaller than the number of *uncovered* edges ($= 3$). Therefore, the bound becomes active and the algorithm returns to the preceding level.

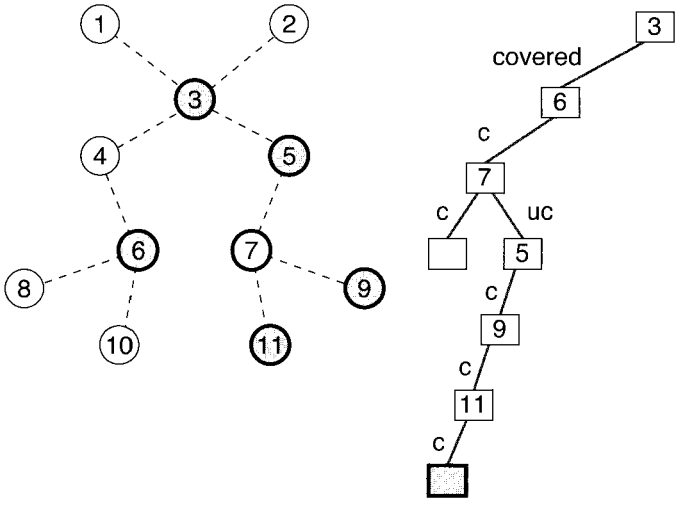


Figure 12.9: Example of the branch-and-bound algorithm. Result after the first backtracking step. Vertex 7 is *uncovered* (indicated by a bold, light circle), thus all its neighbors must be *covered*, (uc=*uncovered*).

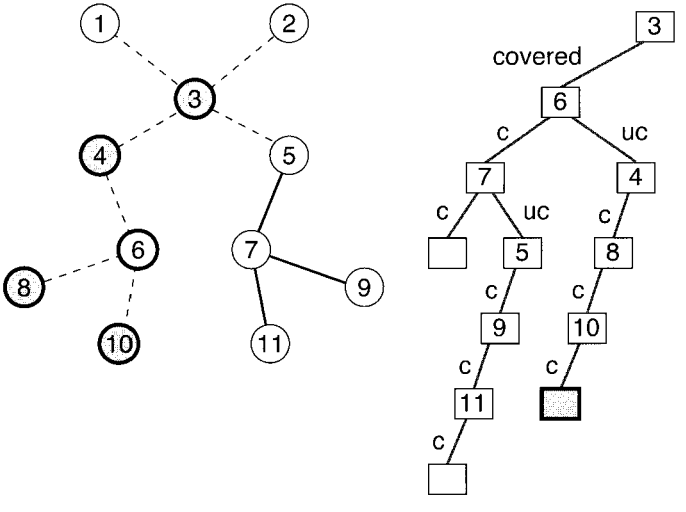


Figure 12.10: Example of the branch-and-bound algorithm. Result after the second backtracking step. Vertex 6 is *uncovered*, thus its neighbors 4, 8 and 10 must be *covered*.

Now the algorithm again reaches the top level of the backtracking tree. Vertex 3 is *uncovered* and all its neighbors (1, 2, 4 and 5) are *uncovered* ($X = 4$), see Fig. 12.2. Again, no VC has been found, i.e. the algorithm proceeds with the

calculation of the bound. Again, $F = best - X = 3 - 4 = -1$ is obtained, yielding $D = 0$. Since 4 uncovered edges remain, the bound becomes active again.

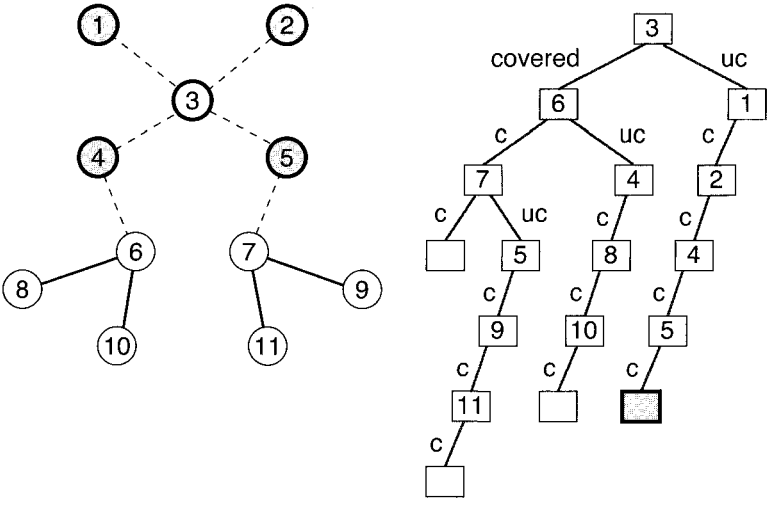


Figure 12.11: Example of the branch-and-bound algorithm. Final situation. Vertex 3 is *uncovered*, thus its neighbors 1, 2, 4 and 5 must be *covered*.

The algorithm returns again to the top level and has been finished. The minimum vertex cover has size $best = 3$. Please note that the backtracking tree only contains 17 nodes on 7 levels, while a full configuration tree would contain 12 levels, 2^{11} leaves and $2^{12} - 1 = 4097$ nodes. This shows clearly that by the branch-and-bound method the running time can be decreased considerably.

□

For the preceding example, for the calculation of the bounds $F \leq 0$ was always obtained. This is due to the small graph sizes, which can be treated within the scope of an example. With real applications, $F > 0$ occurs. Then, for every calculation of the bound, one has to access the F vertices of largest current connectivity. Therefore, it is favorable to implement the table of vertices as two arrays v_1, v_2 of sets of vertices. The arrays are indexed by the current degree of the vertices. The sets in the first array v_1 contain the F *free* vertices of the largest current degree, while the other array contains all other *free* vertices. Every time a vertex changes its degree, it is moved to another set, and eventually even changes the array. Also, in case the mark (*free/covered/uncovered*) of a vertex changes it may be entered in or removed from an array, possibly the smallest degree vertex of v_1 is moved to v_2 or vice versa. Since we are treating graphs of finite average connectivity, this implementation ensures that

the running time spent in each node of the graph is growing slower than linear in the graph size[1]. For the sake of clarity, we have omitted the update operation for both arrays from the algorithmic presentation.
The algorithm, as it has been presented, is suitable for solving the optimization problem, i.e. finding the smallest feasible size $X_c = Nx_c$ of a vertex cover, i.e. the minimum number of *covered* vertices needed to cover the graph fully. The algorithm can be easily modified to treat the problem, where the size $X = V_{vc}$ is given and a configuration with minimum energy is to be found. Then, in *best*, it is not the current smallest size of a vertex cover but the smallest number of uncovered edges (i.e. the energy) that is stored. The bound becomes active, if $F + D$ is larger than or equal to the current number of uncovered edges. Furthermore, when a vertex is *uncovered*, the step where all neighbors are *covered* cannot be applied, because the configuration of the lowest energy may not be a VC. On the other hand, if a VC has been found, i.e. all edges are covered, the algorithm can stop, because for sure no better solution can be obtained. But the algorithm can stop only for the case when *one* optimum has to be obtained. In case all minima have to be enumerated, the algorithm has to proceed and the bound becomes active only in the case if $F + D$ is strictly larger (not equal) to the current number of *uncovered* edges.
Although the branch-and-bound algorithm is very simple, in the regime $4 < c < 10$ random graphs up to size $N = 140$ can be treated. It is difficult to compare the branch-and-bound algorithm to more elaborate algorithms [14, 15], because they are usually tested on a different graph ensemble where each edge appears with a certain probability, independently of the graph size (high-connectivity regime). Nevertheless, in the computer-science literature usually graphs with up to 200 vertices are treated, which is slightly larger than the systems considered here. Nevertheless, the algorithm presented here has the advantage that it is easy to implement and its power is sufficient to study interesting problems. Some results are presented in the next section.

12.3 Results

First, the problem is considered where the energy is minimized for fixed values x. As stated in the first section, we know that for small values of x, the energy (12.1) is not zero [$e(0) = c$], i.e. no vertex covers with xN vertices covered exist. On the other hand, for large values of x, the random graphs are almost surely coverable, i.e. $e(x) = 0$. In Fig. 12.12 the average ground-state energy and the probability $P_{\mathrm{cov}}(x)$ that a graph is coverable with xN vertices is shown for different system sizes $N = 25, 50, 100$ ($c = 1.0$). The results were obtained using the branch-and-bound algorithm presented in the last section. The data are averages over 10^3 ($N = 100$) to 10^4 ($N = 25, 50$) samples. As expected, the value of $P_{\mathrm{cov}}(x)$ increases with the fraction of *covered* vertices. With growing graph sizes, the curves become steeper. This indicates that in the limit $N \to \infty$, which we are interested in, a sharp threshold $x_c \approx 0.39$ appears. Above x_c a graph is coverable with probability one, below it is

[1] Efficient implementations of *sets* require at most $O(\log S)$ time for the operations, where S is the size of a set.

almost surely not coverable. Thus, in the language of a physicist, a *phase transition* from an uncoverable phase to a coverable phase occurs. Please note that the value x_c of the critical threshold depends on the average density of vertices. The result for the phase boundary x_c as a function of c obtained from the simulations is shown later on.

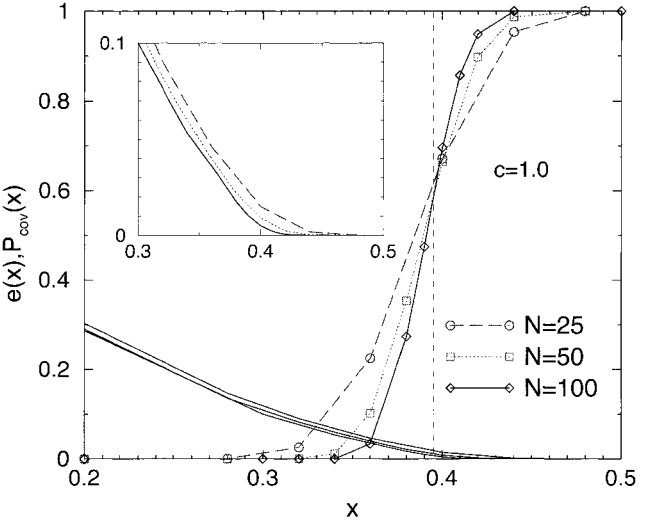


Figure 12.12: Probability $P_{\mathrm{cov}}(x)$ that a cover exists for a random graph ($c = 2$) as a function of the fraction x of *covered* vertices. The result is shown for three different system sizes $N = 25, 50, 100$ (averaged for $10^3 - 10^4$ samples). Lines are guides to the eyes only. In the left part, where the P_{cov} is zero, the energy e (see text) is displayed. The inset enlarges the result for the energy in the region $0.3 \leq x \leq 0.5$.

In Fig. 12.13 the median running time of the branch-and-bound algorithm is shown as a function of the fraction x of covered vertices. The running time is measured in terms of the number of nodes which are visited in the backtracking tree. Again graphs with $c = 1.0$ were considered and an average over the same realizations has been performed. A sharp peak can be observed near the transition x_c. This means, near the phase transition, the instances which are the hardest to solve can be found. Please note, that for values $x < x_c$, the running time still increases exponentially, as can been seen from the inset of Fig. 12.13. For values x considerably larger than the critical value x_c, the running time is linear. The reason is that the heuristic is already able to find a VC, i.e. the algorithm terminates after the first descent into the running tree[2].

Please note that in physics phase transitions are usually indicated by divergences in measurable quantities such as specific heat or magnetic susceptibilities. The peak

[2]The algorithm terminates after a full cover of the graph has been found.

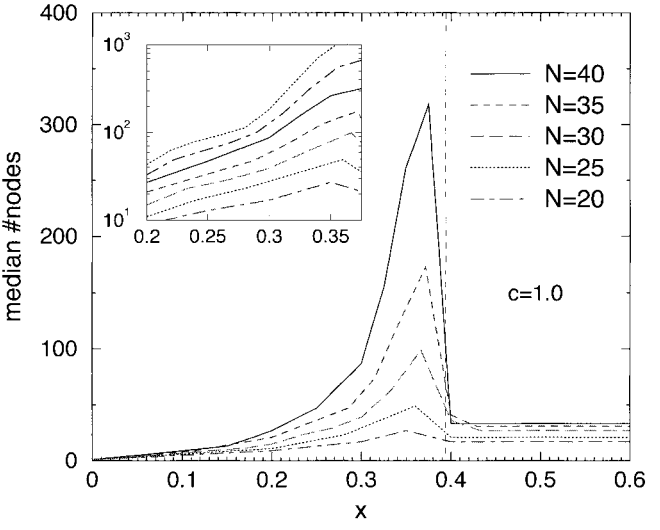


Figure 12.13: Time complexity of the vertex cover. Median number of nodes visited in the backtracking tree as a function of the fraction x of *covered* vertices for graph sizes $N = 20, 25, 30, 35, 40$ ($c = 1.0$). The inset shows the region below the threshold with logarithmic scale, including also data for $N = 45, 50$. The fact that in this representation the lines are equidistant shows that the time complexity grows exponentially with N.

appearing in the time complexity serves as a similar indicator, but is not really equivalent, because the time complexity diverges everywhere, only the rate of divergence is much stronger near the phase transition.

For the uncoverable region, the running time is also fast, but still exponential. This is due to the fact that a configuration with a minimum number of *uncovered* edges has to be obtained. If only the question whether a VC exists or not is to be answered, the algorithm can be easily changed[3], such that for small values of x again a polynomial running time will be obtained.

To calculate the phase diagram numerically, as presented in Fig. 12.3 for the analytical results, it is sufficient to calculate for each graph the size $X_c = Nx_c$ of a minimum vertex cover, as done by the version of the branch-and-bound algorithm or the divide-and-conquer method presented in the last chapter. To compare with the analytical result from Eq. (12.2) one has to perform the thermodynamic limit numerically. This can be achieved by calculating an average value $x_c(N)$ for different graph sizes N. The results for $c = 1.0$ are shown in the inset of Fig. 12.14. Then one fits a function

$$x_c(N) = x_c + aN^{-b} \tag{12.3}$$

[3]Set *best* := 0 initially.

to the data. The form of the function is purely heuristic, no exact justification exists. But in case you do not know anything about the finite-size behavior of a quantity, an algebraic ansatz of the form (12.3) is always a good guess. As can be seen from the inset, the fit matches well.

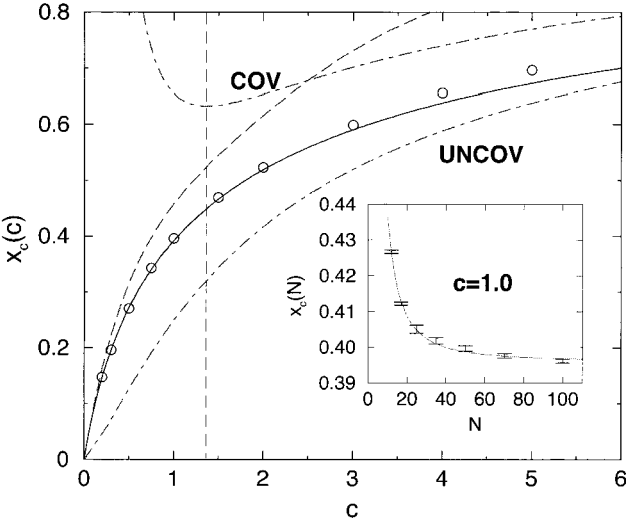


Figure 12.14: Phase diagram showing the fraction $x_c(c)$ of vertices in a minimal vertex cover as a function of the average connectivity c. For $x > x_c(c)$, almost all graphs have covers with xN vertices, while they have almost surely no cover for $x < x_c(c)$. The solid line shows the analytical result. The circles represent the results of the numerical simulations. Error bars are much smaller than symbol sizes. The upper bound of Harant is given by the dashed line, the bounds of Gazmuri by the dash-dotted lines. The vertical line is at $c = e/2 \approx 1.359$. Inset: all numerical values were calculated from finite-size scaling fits of $x_c(N, c)$ using functions $x_c(N) = x_c + aN^{-b}$. We show the data for $c = 1.0$ as an example.

This procedure has been performed for several concentrations c of the edges. The result is indicated in Fig. 12.14 by symbols. Obviously, the numerical data and the analytical result, which has been obtained by using methods from statistical physics, agree very well in the region $c < e/2 \approx 1.359$, as expected. But for larger connectivities of the graph agreement is also very good.

A stronger deviation between numerical and analytical results can be observed for the fraction b_c of the backbone vertices. Please remember that the backbone of a graph contains all vertices, which have in all degenerate minimum vertex covers (or all lowest energy configurations) the same state, i.e. they are either always *covered* or always *uncovered*. The numerical result can be obtained in a similar fashion as the threshold. For each graph G all minimum vertex covers are enumerated. All vertices having the

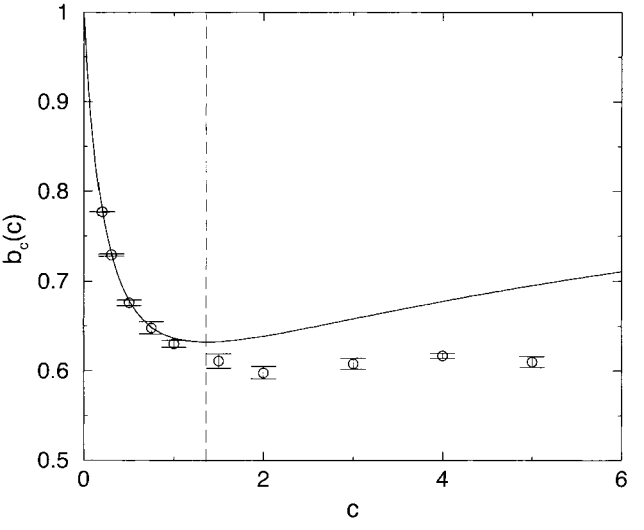


Figure 12.15: The total backbone size of minimal vertex covers as a function of c. The solid line shows the analytical result. Numerical data are represented by the error bars. They were obtained from finite-size scaling fits similar to the calculation for $x_c(c)$. The vertical line is at $c = e/2 \approx 1.359$ where the analytic results becomes not exact.

same state in all configurations belong to the backbone B. Then, the resulting fraction $b_c(G) = |B|/N$ of backbone vertices is averaged by considering different realizations of the random graphs, for one graph size N. The process is performed for different values of N. These data can be used to extrapolate to the thermodynamic limit via a finite-size scaling function. The result, as a function of different edge concentrations c, is displayed in Fig. 12.15 and compared with the analytical result. Again, a very good agreement can be observed for low values $c < e/2 \approx 1.359$, while for graphs having a stronger connectivity, larger deviations occur. Please note that for the case $c = 0.0$, where the graph has no edges, no vertex needs to be covered, meaning that all vertices belong to the backbone [$b_c(0) = 1$].
Many more results can be found in [16, 17]. In particular the critical concentration $c = e/2 \approx 1.359$ is related to the behavior of the subgraphs, consisting only of the non-backbone vertices. The analytical calculations are displayed in [17]. A calculation of the average running time for a simple algorithm, distinguishing the polynomial and the exponential regime, can be found in [18].

Bibliography

[1] B. Hayes, *Am. Scient.* **85**, 108 (1997)

[2] R. Monasson and R. Zecchina, *Phys. Rev.* **E56**, 1357 (1997)

[3] S. Mertens, *Phys. Rev. Lett.* **81**, 4281 (1998)

[4] T. Hogg, B.A. Huberman, and C. Williams (ed.), *Frontiers in Problem Solving: Phase Transitions and Complexity*, Artif. Intell. **81** (I+II) (1996)

[5] O. Dubois, R. Monasson, B. Selman, and R. Zecchina (ed.), *Theor. Comp. Sci.* **265**, (2001)

[6] B. Bollobas, *Random Graphs*, (Academic Press, New York 1985)

[7] P. G. Gazmuri, *Networks* **14**, 367 (1984)

[8] A. M. Frieze, *Discr. Math.* **81**, 171 (1990)

[9] M. Weigt and A. K. Hartmann, *Phys. Rev. Lett.* **84**, 6118 (2000)

[10] K. Mehlhorn and St. Näher, *The LEDA Platform of Combinatorial and Geometric Computing* (Cambridge University Press, Cambridge 1999);
see also http://www.mpi-sb.mpg.de/LEDA/leda.html

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

[12] E.L. Lawler and D.E. Wood, *Oper. Res.* **14**, 699 (1966)

[13] R.E. Tarjan and A.E. Trojanowski, *SIAM J. Comp.* **6**, 537 (1977)

[14] M. Shindo and E. Tomita, *Syst. Comp. Jpn.* **21**, 1 (1990)

[15] R. Lüling and B. Monien, in: *Sixth International Parallel Processing Symposium*, (IEEE Comput. Soc. Press, Los Alamitos, USA 1992)

[16] A.K. Hartmann and M. Weigt, *J. Theor. Comp. Sci.* **265**, 199 (2001)

[17] M. Weigt and A.K. Hartmann, *Phys. Rev. E* **63**, 056127 (2001)

[18] M. Weigt and A.K. Hartmann, *Phys. Rev. Lett.* **86**, 1658 (2001)

13 Practical Issues

Here practical aspects of conducting research via computer simulations are discussed. It is assumed that you are familiar with an operating system such as UNIX (e.g. Linux), a high-level programming language such as C, Fortran or Pascal and have some experience with at least small software projects.
Because of the limited space, usually only short introductions to the specific areas are given and references to more extensive literature are cited. All examples of code are in C/C++.
First, a short introduction to software engineering is given and several hints allowing the construction of efficient and reliable code are stated. In the second section a short introduction to object-oriented software development is presented. In particular, it is shown that this kind of programming style can be achieved with standard procedural languages such as C as well. Next, practical hints concerning the actual process of writing the code are given. In the fourth section macros are introduced. Then it is shown how the development of larger pieces of code can be organized with the help of so called *make files*. In the subsequent section the benefit of using libraries like *Numerical Recipes* or *LEDA* are explained and it is shown how you can build your own libraries. In the seventh section the generation of random numbers is covered while in the eighth section three very useful debugging tools are presented. Afterwards, programs to perform data analysis, curve fitting and finite-size scaling are explained. In the last section an introduction to information retrieval and literature search in the Internet and to the preparation of presentations and publications is given.

13.1 Software Engineering

When you are creating a program, you should never just start writing the code. In this way only tiny software projects such as scripts can be completed successfully. Otherwise your code will probably be very inflexible and contain several hidden errors which are very hard to find. If several people are involved in a project, it is obvious that a considerable amount of planning is necessary.
But even when you are programming alone, which is not unusual in physics, the first step you should undertake is just to sit down and think for a while. This will save you a lot of time and effort later on. To emphasize the need for structuring in the software development process, the art of writing good programs is usually called *software engineering*. There are many specialized books in this fields, see e.g. Refs.

[1, 2]. Here just the steps that should be undertaken to create a sophisticated software development process are stated. The following descriptions refer to the usual situation you find in physics: one or a few people are involved in the project. How to manage the development of big programs involving many developers is explained in literature.

- **Definition of the problem and solution strategies**
  You should write down which problem you would like to solve. Drawing diagrams is always helpful! Discuss your problem with others and tell them how you would like to solve it. In this context many questions may appear, here some examples are given:

  - What is the input you have to supply? In case you have only a few parameters, they can be passed to the program via options. In other cases, especially when chemical systems are to be simulated, many parameters have to be controlled and it may be advisable to use extra parameter files.
  - Which results do you want to obtain and which quantities do you have to analyze? Very often it is useful to write the raw results of your simulations, e.g. the positions of all atoms or the orientations of all spins of your system, to a configuration file. The physical results can be obtained by post-processing. Then, in case new questions arise, it is very easy to analyze the data again. When using configuration files, you should estimate the amount of data you generate. Is there enough space on your disk? It may be helpful, to include the compression of the data files directly in your programs[1].
  - Can you identify "objects" in your problem? Objects may be physical entities like atoms or molecules, but also internal structures like nodes in a tree or elements of tables. Seeing the system and the program as a hierarchical collection of objects usually makes the problem easier to understand. More on object-oriented development can be found in Sec. 13.2.
  - Is the program to be extended later on? Usually a code is "never" finished. You should foresee later extensions of the program and set up everything in a way it can be reused easily.
  - Do you have existing programs available which can be included into the software project? If you have implemented your previous projects in the above mentioned fashion, it is very likely that you can recycle some code. But this requires experience and is not very easy to achieve at the beginning. But over the years you will have a growing library of programs which enables you to finish future software projects much quicker.

    Has somebody else created a program which you can reuse? Sometimes you can rely on external code like libraries. Examples are the *Numerical Recipes* [3] and the *LEDA* library [4] which are covered in Sec. 13.5.
  - Which algorithms are known? Are you sure that you can solve the problem at all? Many other techniques have been invented already. You should always

[1] In C this can be achieved by calling `system("gzip -f <filename>")`; after the file has been written and closed.

search the literature for solutions which already exist. How searches can be simplified by using electronic data bases is covered more deeply in Sec. 13.9.

Sometimes it is necessary to invent new methods. This part of a project may be the most time consuming.

- **Designing data structures**
  Once you have identified the basic objects in your systems, you have to think about how to represent them in the code. Sometimes it is sufficient to define some *struct* types in C (or simple *classes* in C++). But usually you will need to design a large set of data structures, referencing each other in a complicated way.

  A sophisticated design of the data structures will lead to a better organized program, usually it will even run faster. For example, consider a set of vertices of a graph. Then assume that you have several lists L_i each containing elements referencing the vertices of degree i. When the graph is altered in your program and thus the degrees of the vertices change, it is sometimes necessary to remove a vertex from one list and insert it into another. In this case you will gain speed, when your vertices data structures also contain pointers to the positions where they are stored in the lists. Hence, removing and inserting vertices in the lists will take only a constant amount of time. Without these additional pointers, the insert and delete operations have to scan partially through the lists to locate the elements, leading to a linear time complexity of these operations.

  Again, you should perform the design of the data structures in a way, that later extensions are facilitated. For example when treating lattices of Ising spins, you should use data structures which are independent of the dimension or even of the structure of the lattice, an example is given in Sec. 13.4.1.

  When you are using external libraries, usually they have some data types included. The above mentioned LEDA library has many predefined data types like arrays, stacks, lists or graphs. You can have e.g. arrays of arbitrary objects, for example arrays of strings. Furthermore, it is possible to combine the data types in complicated ways, e.g. you can define a stack of graphs having strings attached to the vertices.

- **Defining small tasks**
  After setting up the basic data types, you should think about which basic and complex operations, i.e. which subroutines, you need to manipulate the objects of your simulation. Since you have already thought a lot about your problem, you have a good overview, which operations may occur. You should break down the final task "perform simulation" into small subtasks, this means you use a *top down* approach in the design process. It is not possible to write a program in a sequential way as one code. For the actual implementation, a *bottom up* approach is recommended. This means you should start with the most basic operations. Later on you can use them to create more complicated operations. As always, you should define the subroutines in a way that they can be applied in a flexible way and extensions are easy to perform.

But it is not necessary that you must identify all basic operations at the beginning. During the development of the code, new applications may arise, which lead to the need for further operations. Also it may be required to change or extend the data structures defined before. However, the more you think in advance, the less you need to change the program later on.

As an example, the problem of finding ground states in Ising spin glasses via simulated annealing is considered. Some of basic operations are:

- Set up the data structures for storing the realizations of the interactions and for storing the spin glass configurations.
- Create a random realization of the interactions.
- Initialize a random spin configuration.
- Calculate the energy of a spin in the local field of its neighbors.
- Calculate the total energy of a system.
- Calculate the energy changes associated with a spin flip.
- Execute a Monte Carlo step.
- Execute a whole annealing run.
- Calculate the magnetization.
- Save a realization and corresponding spin configurations in a file.

It is not necessary to define a corresponding subroutine for all operations. Sometimes they require only a few numbers of lines in the code, like the calculation of the energy of one spin in the example above. In this case, such operations can be written directly in the code, or a macro (see Sec. 13.4.1) can be used.

- **Distributing work**
  In case several people are involved in a project, the next step is to split up the work between the coworkers. If several types of objects appear in the program design, a natural approach is to make everyone responsible for one or several types of objects and the related operations. The code should be broken up into several modules (i.e. source files), such that every module is written by only one person. This makes the implementation easer and also helps testing the code (see below). Nevertheless, the partitioning of the work requires much care, since quite often some modules or data types depend on others. For this reason, the actual implementation of a data type should be hidden. This means that all interactions should be performed through exactly defined interfaces which do not depend on the internal representation, see also Sec. 13.2 on object-oriented programming.

  When several people are editing the same files, which is usually necessary later on, even when initially each file was created by only one person, then you should use a *source-code management system*. It prevents several people from performing changes on the same file in parallel, which would cause a lot of trouble. Additionally, a source-code management system enables you to keep track of all changes

made. An example of such a system is the *Revision Control System* (RCS), which is freely available through the *GNU project* [5] and part of the free operating system *Linux*.

- **Implementing the code**
  With good preparation, the actual implementation becomes only a small part of the software development process. General style rules, guaranteeing clear structured code, which can even be understood several months later, are explained in Sec. 13.3. You should use a different file, i.e. a different module, for each coherent unit of data structures and subroutines; when using an object oriented language you should define different classes (see Sec. 13.2). This rule should be obeyed for the case of a one-person project as well. Large software projects containing many modules are easily maintained via *makefiles* (see Sec. 13.4.2).

  Each subroutine and each module should be tested separately, before integrating many modules into one program. In the following some general hints concerning testing are presented.

- **Testing**
  When performing tests on single subroutines, standard cases usually are used. This is the reason why many errors become apparent much later. Then, because the modules have already been integrated into one single program, errors are much harder to localize. For this reason, you should always try to find special and rare cases as well when testing a subroutine. Consider for example a procedure which inserts an element into a list. Then not only inserting in the middle of the list, but also at the beginning, at the end and into an empty list must be tested. Also, it is strongly recommended to read your code carefully once again before considering it finished. In this way many bugs can be found easily which otherwise must be tracked down by intensive debugging.

  The actual debugging of the code can be performed by placing print instructions at selected positions in the code. But this approach is quite time consuming, because you have to modify and recompile your program several times. Therefore, it is advisable to use debugging tools like a *source-code debugger* and a program for checking the memory management. More about these tools can be found in Sec. 13.7. But usually you also need special operations which are not covered by an available tool. You should always write a procedure which prints out the current instance of the system that is simulated, e.g. the nodes and edges of a graph or the interaction constants of an Ising system. This facilitates the types of tests, which are described in the following.

  After the raw operation of the subroutines has been verified, more complex tests can be performed. When e.g. testing an optimization routine, you should compare the outcome of the calculation for a small system with the result which can be obtained by hand. If the outcome is different from the expected result, the small size of the test system allows you to follow the execution of the program step by step. For each operation you should think about the expected outcome and compare it with the result originating from the running program.

Furthermore, it is very useful to compare the outcome of different methods applied to the same problem. For example, you know that there must be something wrong, in case an approximation method finds a better value than your "exact" algorithm. Sometimes analytical solutions are available, at least for special cases. Another approach is to use invariants. For example, when performing a Molecular Dynamics simulation of an atomic/molecular system (or a galaxy), energy and momentum must be conserved; only numerical rounding errors should appear. These quantities can be recorded very easily. If they change in time there must be a bug in your code. In this case, usually the formulas for the energy and the force are not compatible or the integration subroutine has a bug.

You should test each procedure, directly after writing it. Many developers have experienced that the larger the interval between implementation and tests is, the lower the motivation becomes for performing tests, resulting in more undetected bugs.

The final stage of the testing process occurs when several modules are integrated into one large running program. In the case where you are writing the code alone, not many surprises should appear, if you have performed many tests on the single modules. If several people are involved in the project, at this stage many errors occur. But in any case, you should always remember: there is probably no program, unless very small, which is bug free. You should know the following important result from theoretical computer science [6]: it is impossible to invent a general method, which can prove automatically that a given program obeys a given specification. Thus, all tests must be designed to match the current code.

In case a program is changed or extended several times, you should always keep the old versions, because it is quite common that by editing new bugs are introduced. In that case, you can compare your new code with the older version. Please note that editors like emacs only keep the second latest version as backup, so you have to take care of this problem yourself unless you use a source-code management system, where you are lucky, because it keeps all older version automatically.

For C programmers, it is always advisable to apply the `-Wall` (warning level: all) option. Then several bugs already show up during the compiling process, for example the common mistake to use '=' in comparisons instead of '==', or the access to uninitialized variables[2].

In C++, some bugs can be detected by defining variables or parameter as `const`, when they are considered to stay unchanged in a block of code or subroutine. Here again, already the compiler will complain, if attempts to alter the value of such a variable are tried.

This part finishes with a warning: never try to save time when performing tests. Bugs which appear later on are much much harder to find and you will have to spend much more time than you have "saved" before.

[2]But this is not true for some C++ compilers when combining with option `-g`.

- **Writing documentation**
  This part of the software development process is very often disregarded, especially in the context of scientific research, where no direct customers exist. But even if you are using your own code, you should write good documentation. It should consist of at least three parts:

  - *Comments in the source code*: You should place comments at the beginning of each module, in front of each subroutine or each self-defined data structure, for blocks of the code and for selected lines. Additionally, meaningful names for the variables are crucial. Following these rules makes later changes and extension of the program much more straightforward. You will find in more hints on how a good programming style can be achieved Sec. 13.3.
  - *On-line help*: You should include a short description of the program, its parameters and its options in the main program. It should be printed, when the program is called with the wrong number/form of the parameters, or when the option `-help` is passed. Even when you are the author of the program, after it has grown larger it is quite hard to remember all options and usages.
  - *External documentation*: This part of the documentation process is important, when you would like to make the program available to other users or when it grows really complex. Writing good instructions is really a hard job. When you remember how often you have complained about the instructions for a video recorder or a word processor, you will understand why there is a high demand for good authors of documentation in industry.

- **Using the code**
  Also the actual performance of the simulation usually requires careful preparation. Several question have to be considered, for example:

  - How long will the different runs take? You should perform simulations of small systems and extrapolate to large system sizes.
  - Usually you have to average over different runs or over several realizations of the disorder. The system sizes should also be chosen in a way that the number of samples is large enough to reduce the statistical fluctuations. It is better to have a reliable result for a small system than to treat only a few instances of a large system. If your model exhibits self averaging, the larger the sample, the less the number of samples can be. But, unfortunately, usually the numerical effort grows stronger than the system size, so there will be a maximum system size which can be treated with satisfying accuracy. To estimate the accuracy, you should always calculate the statistical error bar $\sigma(A)$ for each quantity A[3].

    A good rule of a thumb is that each sample should take not more than 10 minutes. When you have many computers and much time available, you can attack larger problems as well.

[3]The error bar is $\sigma(A) = \sqrt{\mathrm{Var}(A)/(N-1)}$, where $\mathrm{Var}(A) = \frac{1}{N}\sum_{i=1}^{N} a_i^2 - (\frac{1}{N}\sum_{i=1}^{N} a_i)^2$ is the variance of the N values $a_1, \ldots, a_N$.

- Where to put the results? In many cases you have to investigate your model for different parameters. You should organize the directories where you put the data and the names of the files in such a way that even years later the former results can be found quickly. You should put a README file in each directory, explaining what it contains.

  If you want to start a sequence of several simulations, you can write a short script, which calls your program with different parameters within a loop.
- Logfiles are very helpful, where during each simulation some information about the ongoing processes are written automatically. Your program should put its version number and the parameters which have been used to start the simulation in the first line of each logfile. This allows a reconstruction of how the results have been obtained.

The steps given do not usually occur in linear order. It is quite common that after you have written a program and performed some simulations, you are not satisfied with the performance or new questions arise. Then you start to define new problems and the program will be extended. It may also be necessary to extend the data structures, when e.g. new attributes of the simulated models have to be included. It is also possible that a nasty bug is still hidden in the program, which is found later on during the actual simulations and becomes obvious by results which cannot be explained. In this case changes cannot be circumvented either.
In other words, the software development process is a *cycle* which is traversed several times. As a consequence, when planning your code, you should always keep this in mind and set up everything in a flexible way, so that extensions and code recycling can be performed easily.

13.2 Object-oriented Software Development

In recent years *object-oriented* programming languages like C++, Smalltalk or Eiffel became very popular. But, using an object-oriented language and developing the program in an object-oriented style are not necessarily the same, although they are compatible. For example, you can set up your whole project by applying object-oriented methods even when using a traditional *procedural* programming language like C, Pascal or Fortran. On the other hand, it is possible to write very traditional programs with modern object-oriented languages. They help to organize your programs in terms of objects, but you have the flexibility to do it in another way as well. In general, taking an object-oriented viewpoint facilitates the analysis of problems and the development of programs for solving the problems. Introductions to object-oriented software development can be found e.g. in Refs. [7, 8, 9]. Here just the main principles are explained:

- **Objects and methods**
  The real world is made of *objects* such as traffic-lights, books or computers. You can classify different objects according to some criteria into *classes*. This means

different chairs belong to the class "chairs". The objects of many classes can have internal *states*, e.g. a traffic-light can be red, yellow or green. The state of a computer is much more difficult to describe. Furthermore, objects are useful for the environment, because other objects interact via *operations* with the object. You (belonging to the class "human") can read the state of a traffic light, some central computer may set the state or even switch the traffic light off.

Similar to the real world, you can have objects in programs as well. The internal state of an object is given by the values of the variables describing the object. Also it is possible to interact with the objects by calling subroutines (called *methods* in this context) associated with the objects.

Objects and the related methods are seen as coherent units. This means you define within one *class definition* the way the objects look, i.e. the data structures, together with the methods which access/alter the content of the objects. The syntax of the class definition depends on the programming language you use. Since implementational details are not relevant here, the reader is referred to the literature.

When you take the viewpoint of a pure object-oriented programmer, then all programs can be organized as collections of objects calling methods of each other. This is derived from the structure the real world has: it is a large set of interacting objects. But for writing good programs it is as in real life, taking an orthodox position imposes too many restrictions. You should take the best of both worlds, the object-oriented and the procedural world, depending on the actual problem.

- **Data capsuling**
  When using a computer, you do not care about the implementation. When you press a key on the keyboard, you would like to see the result on the screen. You are not interested in how the key converts your pressing into an electrical signal, how this signal is sent to the input ports of the chips, how the algorithm treats the signal and so on.

  Similarly, a main principle of object-oriented programming is to hide the actual implementation of the objects. Access to them is only allowed via given interfaces, i.e. via methods. The internal data structures are hidden, this is called `private` in C++. The data capsuling has several advantages:

  - You do not have to remember the implementation of your objects. When using them later on, they just appear as a black box fulfilling some duties.
  - You can change the implementation later on without the need to change the rest of the program. Changes of the implementation may be useful e.g. when you want to increase the performance of the code or to include new features.
  - Furthermore, you can have *flexible data structures*: several different types of implementations may coexist. Which one is chosen depends on the requirements. An example are graphs which can be implemented via arrays, lists, hash tables or in other ways. In the case of sparse graphs, the list implementation has a better performance. When the graph is almost complete, the

array representation is favorable. Then you only have to provide the basic access methods, such as inserting/removing/testing vertices/edges and iterating over them, for the different internal representations. Therefore, higher-level algorithms like computing a spanning tree can be written in a simple way to work with all internal implementations. When using such a class, the user just has to specify the representation he wants, the rest of the program is independent of this choice.

- Last but not least, software debugging is made easier. Since you have only defined ways the data can be changed, undesired side-effects become less common. Also the memory management can be controlled easier.

For the sake of flexibility, convenience or speed it is possible to declare internal variables as `public`. In this case they can be accessed directly from outside.

- **Inheritance**

- inheritance This means lower level objects can be specializations of higher level objects. For example the class of (German) "ICE trains" is a subclass of "trains" which itself is a subclass of "vehicles".

  In computational physics, you may have a basic class of "atoms" containing mass, position and velocity, and built upon this a class of "charged atoms" by including the value of the charge. Then you can use the subroutines you have written for the uncharged atoms, like moving the particles or calculating correlation functions, for the charged atoms as well.

  A similar form of hierarchical organization of objects works the other way round: higher level objects can be defined in terms of lower level objects. For example a book is composed of many objects belonging to the class "page". Each page can be regarded as a collection of many "letter" objects.

  For the physical example above, when modeling chemical systems, you can have "atoms" as basic objects and use them to define "molecules". Another level up would be the "system" object, which is a collection of molecules.

- **Function/operator overloading**
  This inheritance of methods to lower level classes is an example of *operator overloading*. It just means that you can have methods for different classes having the same name, sometimes the same code applies to several classes. This applies also to classes, which are not connected by inheritance. For example you can define how to add integers, real numbers, complex numbers or larger objects like lists, graphs or documents. In language like C or Pascal you can define subroutines to add numbers and subroutines to add graphs as well, but they must have different names. In C++ you can define the operator "+" for all different classes. Hence, the operator-overloading mechanisms of object-oriented languages is just a tool to make the code more readable and clearer structured.

- **Software reuse**
  Once you have an idea of how to build a chair, you can do it several times. Because you have a blueprint, the tools and the experience, building another chair is an easy task.

  This is true for building programs as well: both data capsuling and inheritance facilitate the reuse of software. Once you have written your class for e.g. treating lists, you can include them in other programs as well. This is easy, because later on you do not have to care about the implementation. With a class designed in a flexible way, much time can be saved when realizing new software projects.

As mentioned before, for object-oriented programming you do not necessarily have to use an object-oriented language. It is true that they are helpful for the implementation and the resulting programs will look slightly more elegant and clear, but you can program everything with a language like C as well. In C an object-oriented style can be achieved very easily. As an example a class `histo` implementing histograms is outlined, which are needed for almost all types of computer simulations as evaluation and analysis tools.
First you have to think about the data you would like to store. That is the histogram itself, i.e. an array `table` of bins. Each bin just counts the number of events which fall into a small interval. To achieve a high degree of flexibility, the range and the number of bins must be variable. From this, the width `delta` of each bin can be calculated. For convenience `delta` is stored as well. To count the number of events which are outside the range of the table, the entries `low` and `high` are introduced. Furthermore, statistical quantities like mean and variance should be available quickly and with high accuracy. Thus, several summarized moments `sum` of the distribution are stored separately as well. Here the number of moments `_HISTO_NOM_` is defined as a macro, converting this macro to variable is straightforward. All together, this leads to the following C data structure:

```
#define _HISTO_NOM_      9          /* No. of (statistical) moments */

/* holds statistical informations for a set of numbers:   */
/* histogram, # of Numbers, sum of numbers, squares, ...  */
typedef struct
{
  double           from, to;   /* range of histogram              */
  double              delta;   /* width of bins                   */
  int               n_bask;    /* number of bins                  */
  double            *table;    /* bins                            */
  int             low, high;   /* No. of data out of range        */
  double   sum[_HISTO_NOM_];   /* sum of 1s, numbers, numbers^2 ...*/
} histo_t;
```

Here, the postfix `_t` is used to stress the fact that the name `histo_t` denotes a type. The bins are `double` variables, which allows for more general applications. Please

note that it is still possible to access the internal structures from outside, but it is not necessary and not recommended. In C++, you could prevent this by declaring the internal variables as `private`. Nevertheless, everything can be done via special subroutines. First of all one must be able to create and delete histograms, please note that some simple error-checking is included in the program:

```
/** creates a histo-element, where the empirical histogram **/
/** table covers the range ['from', 'to'] and is divided   **/
/** into 'n_bask' bins.                                    **/
/** RETURNS: pointer to his-Element, exit if no memory.    **/
histo_t *histo_new(double from, double to, int n_bask)
{
  histo_t *his;
  int t;

  his = (histo_t *) malloc(sizeof(histo_t));
  if(his == NULL)
  {
    fprintf(stderr, "out of memory in histo_new");
    exit(1)
  }
  if(to < from)
  {
    double tmp;
    tmp = to; to = from; from = tmp;
    fprintf(stderr, "WARNING: exchanging from, to in histo_new\n");
  }
  his->from = from;
  his->to = to;
  if( n_bask <= 0)
  {
    n_bask = 10;
    fprintf(stderr, "WARNING: setting n_bask=10 in histo_new()\n");
  }
  his->delta = (to-from)/(double) n_bask;
  his->n_bask = n_bask;
  his->low = 0;
  his->high = 0;
  for(t=0; t< _HISTO_NOM_ ; t++) /* initialize summarized moments */
    his->sum[t] = 0.0;
  his->table = (double *) malloc(n_bask*sizeof(double));
  if(his->table == NULL)
  {
    fprintf(stderr, "out of memory in histo_new");
    exit(1);
```

```
  }
  else
    for(t=0; t<n_bask; t++)
      his->table[t] = 0;
  }
  return(his);
}

/** Deletes a histogram 'his'     **/
void histo_delete(histo_t *his)
{
  free(his->table);
  free(his);
}
```

All histogram objects are created dynamically by calling `histo_new()`, this corresponds to a call of the *constructor* or `new` in C++. The objects are addressed via pointers. Whenever a method, i.e. a procedure in C, of the `histo` class is called, the first argument will always be a pointer to the corresponding histogram. This looks slightly less elegant than writing `histo.method()` in C++, but it is really the same. When avoiding direct access, the realization using C is perfectly equivalent to C++ or other object-oriented languages. Inheritance can be implemented, by including pointers to `histo_t` objects in other type definitions. When these higher level objects are created, a call to `histo_new()` must be included, while a call to `histo_delete()`, corresponding to the *destructor* in C++, is necessary, to implement a correct deletion of the more complex objects.
As a final example, the procedures for inserting an element into the table and calculating the mean are presented. It is easy to figure out how other subroutines for e.g. calculating the variance/higher moments or printing a histogram can be realized. The complete library can be obtained for free [10].

```
/** inserts a 'number' into a histogram 'his'. **/
void histo_insert(histo_t *his, double number)
{
  int t;
  double value;
  value = 1.0;
  for(t=0; t< _HISTO_NOM_; t++)
  {
    his->sum[t]+= value;;                    /* raw statistics */
    value *= number;
  }
  if(number < his->from)              /* insert into histogram */
    his->low++;
  else if(number > his->to)
    his->high++;
```

```
  else if(number == his->to)
    his->table[his->n_bask-1]++;
  else
    his->table[(int) floor( (number - his->from) / his->delta)]++;
}

/** RETURNS: Mean of Elements in 'his' (0.0 if his=empty) **/
double histo_mean(histo_t *his)
{
  if(his->sum[0] == 0)
    return(0.0);
  else
    return(his->sum[1] / his->sum[0]);
}
```

13.3 Programming Style

The code should be written in a style that enables the author, and other people as well, to understand and modify the program even years later. Here briefly some principles you should follow are stated. Just a general style of description is given. Everybody is free to choose his/her own style, as long as it is precise and consistent.

- Split your code into several modules. This has several advantages:
  - When you perform changes, you have to recompile only the modules which have been edited. Otherwise, if everything is contained in a long file, the whole program has to be recompiled each time again.
  - Subroutines which are related to each other can be collected in single modules. It is much easier to navigate in several short files than in one large program.
  - After one module has been finished and tested it can be used for other projects. Thus, software reuse is facilitated.
  - Distributing the work among several people is impossible if everything is written into one file. Furthermore, you should use a source-code management system (see Sec. 13.1) in case several people are involved in avoiding uncontrolled editing.
- To keep your program logically structured, you should always put data structures and implementations of the operations in separate files. In C/C++ this means you have to write the data structures in a header (`.h`) file and the code into a source code (`.c/ .cpp`) file.
- Try to find meaningful names for your variables and subroutines. Therefore, during the programming process it is much easier to remember their meanings, which helps a lot in avoiding bugs. Additionally, it is not necessary to look up

the meaning frequently. For local variables like loop counters, it is sufficient and more convenient to have short (e.g. one letter) names.

In the beginning this might seem to take additional time (writing e.g. 'kinetic_energy' for a variable instead of 'x10'). But several months after you have written the program, you will appreciate your effort, when you read the line

```
kinetic_energy += 0.5*atom[i].mass*atom[i].veloc*atom[i].veloc;
```

instead of

```
x10 += 0.5*x34[i].a*x34[i].b*x34[i].b;
```

- You should use proper indentation of your lines. This helps a great deal in recognizing the structure of a program. Many bugs are caused by misaligned braces forming a block of code. Furthermore, you should place at most one command per line of code. The reader will probably agree that

```
for(i=0; i<number_nodes; i++)
{
  degree[i] = 0;
  for(j=0; j<number_nodes; j++)
     if(edge[i][j] > 0)
       degree[i]++;
}
```

is much faster to understand than

```
for(i=0; i<number_nodes; i++) { degree[i] = 0; for(j=0;
j<number_nodes; j++)   if(edge[i][j] > 0)  degree[i]++; }
```

- Avoid jumping to other parts of a program via the "goto" command. This is bad style originating from programming in assembler or BASIC. In modern programming languages, for every logical programming construct there are corresponding commands. "Goto" commands make a program harder to understand and much harder to debug if it does not work as it should.

  In case you want to break out of a loop, you can use a while/until loop with a flag that indicates if the loop is to be stopped. In C, if you are lazy, you can use the commands `break` or `continue`.

- Do not use global variables. At first sight the use of global variables may seem tempting: you do not have to care about parameters for subroutines, everywhere the variables are accessible and everywhere they have the same name. Programming is done much faster.

  But later on you will have a bad time: many bugs are created by improper use of global variables. When you want to check for a definition of a variable you have to search the whole list of global variables, instead of just checking the parameter

list. Sometimes the range of validity of a global variable is overwritten by a local variable. Furthermore, software re-usage is almost impossible with global variables, because you always have to check *all* variables used in a module for conflicts and you are not allowed to employ the name for another object. When you want to pass an object to a subroutine via a global variable, you do not have the choice of how to name the object which is to be passed. Most important, when you have a look onto a subroutine after some months, you cannot see immediately which objects are changed in the subroutine, instead you will have to read the whole subroutine again. If you avoid this practice, you just have to look at the parameter list. Finally, when a renaming occurs, you have to change the name of a global variable everywhere in the whole program. Local variables can be changed with little effort.

- Finally, an issue of utmost importance: Do not be economical with comments in your source code! Most programs, which may appear logically structured when writing them, will be a source of great confusion when being read some weeks later. Every minute you spend on writing reasonable comments you will save later on several times over. You should consider different types of comments.

  – Module comments: At the beginning of each module you should state its name, what the module does, who wrote it and when it was written. It is a useful practice to include a version history, which lists the changes that have been performed. A module comment might look like this:

```
/*************************************************************/
/*** Functions for spin glasses.                           ***/
/*** 1. loading and saving of configurations               ***/
/*** 2. initialization                                     ***/
/*** 3. evaluation functions                               ***/
/***                                                       ***/
/*** A.K. Hartmann January 1996                            ***/
/*** Version 1.8    09.10.2000                             ***/
/***                                                       ***/
/*************************************************************/

/*** Vers. History:                                        ***/
/*** 1.0 feof-check in lsg_load...() included 02.03.96 ***/
/*** 2.0 comment for cs2html added            12.05.96 ***/
/*** 3.0 lsg_load_bond_n() added              03.03.97 ***/
/*** 4.0 lsg_invert_plane() added             12.08.98 ***/
/*** 5.0 lsg_write_gen() added                15.09.98 ***/
/*** 6.0 lsg_energy_B_hom() added             20.11.98 ***/
/*** 7.0 lsg_frac_frust() added               03.07.00 ***/
/*** 7.1 use new call-form of llist.c library 04.07.00 ***/
/***      -> no memory leak (through copy data)        ***/
/*** 8.0 lsg_mc_T() added                     23.08.00 ***/
```

- Type comments: For each data type (a `struct` in C or *class* in C++) which you define in a header file, you should attach several lines of comments describing the data type's structure and its application. For a class definition, also the methods which are available should be described. Furthermore, for a structure, each element should be explained. A nice arrangement of the comments makes everything more readable. An example of what such a comment may look like can be seen in Sec. 13.2 for the data type `histo_t`.

- Subroutine comments: For each subroutine, its purpose, the meaning of the input and output variables and the preconditions which have to be fulfilled before calling must be stated. In case you are lazy and do not write a *man* page, a comment atop of a subroutine is the only source of information, should you want to use the subroutine later on in another program.

  If you use some special mathematical methods or clever algorithms in the subroutine, you should always cite the source in the comment. This facilitates later on the understanding of how the methods works.

  The next example shows what the comment for a subroutine may look like:

```
/************************ mf_dinic1() ****************/
/** Calculated maximum flow using Dinics algorithm    **/
/** See: R.E.Tarjan, Data Structures and Network      **/
/** Algorithms, p.104f.                               **/
/**                                                   **/
/** PARAMETERS: (*)= return-parameter/altered var's   **/
/**        N: number of inner nodes (without s,t)     **/
/**      dim: dimension of lattice                    **/
/**     next: gives neighbors next[0..N][0..2*dim+1]  **/
/**        c: capacities  c[0..N][0..2*dim+1]         **/
/**   (*)  f: flow values f[0..N][0..2*dim+1]         **/
/** use_flow: 0-> flow set to zero before used.       **/
/**                                                   **/
/** RETURNS:                                          **/
/**          0 -> OK                                  **/
/*****************************************************/
int mf_dinic1(int N, int dim, int *next, int *c,
              int *f, int use_flow)
```

- Block comments: You should divide each subroutine, unless it is very short, into several logical blocks. A rule of thumb is that no block should be longer than the number of lines you can display in your editor window. Within one or two lines you should explain what is done in the block. Example:

```
/* go through all nodes except source s and sink t in  */
/* reversed topological order and set capacities       */
for(t2=num_nodes-2; t2>0; t2--)
```

– Line comments: They are the lowest level comments. Since you are using (hopefully) sound names for data types, variables and subroutines, many lines should be self explanatory. But in case the meaning is not obvious, you should add a small comment at the end of a line, for example:

```
C(t, SOURCE) = cap_s2t[t];      /* restore capacities */
```

Aligning all comments to the right makes a code easier to read. Please avoid unnecessary comments like

```
counter++;                              /* increase counter */
```

or unintelligible comments like

```
minimize_energy(spin, N, next, 5);   /* I try this one */
```

The line containing `C(t, SOURCE)` is an example of the application of a macro. This subject is covered in the following section.

13.4 Programming Tools

Programming languages and UNIX/Linux offer many concepts and tools which help you to perform large simulation projects. Here, three of them are presented: macros, which are explained first, *makefiles* and scripts.

13.4.1 Using Macros

Macros are shortcuts for code sequences in programming languages. Their primary purpose is to allow computer programs to be written more quickly. But the main benefit comes from the fact that a more flexible software development becomes possible. By using macros appropriately, programs become better structured, more generally applicable and less error-prone. Here it is explained how macros are defined and used in C, a detailed introduction can be found in C textbooks such as Ref. [11]. Other high-level programming languages exhibit similar features.
In C a macro is constructed via the `#define` directive. Macros are processed in the preprocessing stage of the compiler. This directive has the form

`#define` *name* *definition*

Each definition must be on one line, without other definitions or directives. If the definition extends over more than one line, each line except the last one has to be ended with the backslash \ symbol. The simplest form of a macro is a constant, e.g.

```
#define PI 3.1415926536
```

You can use the same sorts of names for macros as for variables. It is convention to use only upper-case letters for macros. A macro can be deleted via the `#undef` directive. When scanning the code, the preprocessor just replaces literally every occurrence of a macro by its definition. If you have for example the expression `2.0*PI*omega` in your

code, the preprocessor will convert it into `2.0*3.1415926536*omega`. You can use macros also in the definition of other macros. But macros are not replaced in strings, i.e. `printf("PI");` will print `PI` and not 3.1415926536 when the program is running. It is possible to test for the (non)existence of macros using the `#ifdef` and `#ifndef` directives. This allows for conditional compiling or for platform-independent code, such as e.g. in

```
#ifdef UNIX
  ...
#endif
#ifdef MSDOS
  ...
#endif
```

Please note that it is possible to supply definitions of macros to the compiler via the `-D` option, e.g. `gcc -o program program.c -DUNIX=1`. If a macro is used only for conditional #ifdef/#ifndef statements, an assignment like `=1` can be omitted, i.e. `-DUNIX` is sufficient.
When programs are divided into several modules, or when library functions are used, the definition of data types and functions are provided in header files (`.h` files). Each header file should be read by the compiler only once. When projects become more complex, many header files have to be managed, and it may become difficult to avoid multiple scanning of some header files. This can be prevented automatically by this simple construction using macros:

```
/** example .h file: myfile.h  **/

#ifndef _MYFILE_H_
#define _MYFILE_H_

   .... (rest of .h file)
   (may contain other #include directives)

#endif /* _MYFILE_H_ */
```

After the body of the header file has been read the first time during a compilation process, the macro `_MYFILE_H_` is defined, thus the body will never read be again.
So far, macros are just constants. You will benefit from their full power when using macros with arguments. They are given in braces after the name of the macro, such as e.g. in

```
#define  MIN(x,y)  ( (x)<(y) ? (x):(y) )
```

You do not have to worry more than usual about the names you choose for the arguments, there cannot be a conflict with other variables of the same name, because they are replaced by the expression you provide when a macro is used, e.g. `MIN(4*a, b-32)` will be expanded to `(4*a)<(b-32) ? (4*a):(b-32)`.

The arguments are used in braces () in the macro, because the comparison < must have the lowest priority, regardless which operators are included in the expressions that are supplied as actual arguments. Furthermore, you should take care of unexpected side effects. Macros do not behave like functions. For example when calling `MIN(a++,b++)` the variable `a` or `b` may be increased twice when the program is executed. Usually it is better to use inline functions (or sometimes templates in C++) in such cases. But there are many applications of macros, which cannot be replaced by incline functions, like in the following example, which closes this section.

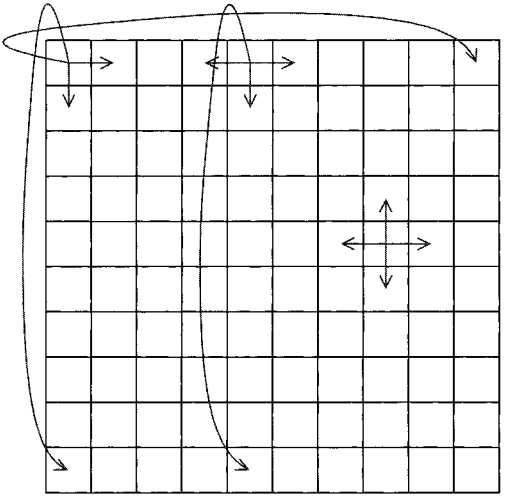


Figure 13.1: A square lattice of size 10×10 with periodical boundary conditions. The arrows indicate the neighbors of the spins.

The example illustrates how a program can be written in a clear way using macros, making the program less error-prone, and furthermore allowing for a broad applicability. A system of Ising spins is considered, that is a lattice where at each site i a particle σ_i is placed. Each particle can have only two states $\sigma_i = \pm 1$. It is assumed that all lattice sites are numbered from 1 to N. This is different from C arrays, which start at index 0, the benefit of starting with index 1 for the sites will become clear below. For the simplest version of the model only neighbors of spins are interacting. With a two-dimensional square lattice of size $N = L \times L$ a spin i, which is not at the boundary, interacts with spins $i+1$ ($+x$-direction), $i-1$ ($-x$-direction), $i+L$ ($+y$-direction) and $i-L$ ($-y$-direction). A spin at the boundary may interact with fewer neighbors when free boundary conditions are assumed. With periodic boundary conditions (pbc), all spins have exactly 4 neighbors. In this case, a spin at the boundary interacts also with the nearest mirror images, i.e. with the sites that are neighbors if you consider the system repeated in each direction. For a 10×10 system spin 5, which is in the first row, interacts with spins $5+1=6$, $5-1=4$, $5+10=15$ and through the pbc

with spin 95, see Fig. 13.1. The spin in the upper left corner, spin 1, interacts with spins 2, 11, 10 and 91. In a program pbc can be realized by performing all calculations *modulo* L (for the $\pm x$-directions) and modulo L^2 (for the $\pm y$-directions), respectively. This way of realizing the neighbor relations in a program has several disadvantages:

- You have to write the code everywhere where the neighbor relation is needed. This makes the source code larger and less clear.
- When switching to free boundary conditions, you have to include further code to check whether a spin is at the boundary.
- Your code works only for one lattice type. If you want to extend the program to lattices of higher dimension you have to rewrite the code or provide extra tests/calculations.
- Even more complicated would be an extension to different lattice structures such as triangle or face-center cubic. This would make the program look even more confusing.

An alternative is to write the program directly in a way it can cope with almost arbitrary lattice types. This can be achieved by setting up the neighbor relation in one special initialization subroutine (not discussed here) and storing it in an array `next[]`. Then, the code outside the subroutine remains the same for all lattice types and dimensions. Since the code should work for all possible lattice dimensions, the array `next` is one dimensional. It is assumed that each site has `num_n` neighbors. Then the neighbors of site `i` can be stored in `next[i*num_n]`, `next[i*num_n+1]`, ..., `next[i*num_n+num_n-1]`. Please note that the sites are numbered beginning with 1. This means, a system with `N` spins needs an array `NEXT` of size `(N+1)*num_n`. When using free boundary conditions, missing neighbors can be set to 0. The access to the array can be made easier using a macro `NEXT`:

```
#define  NEXT(i,r)  next[(i)*num_n + r]
```

`NEXT(i,r)` contains the neighbor of spin `i` in direction `r`. For e.g. a quadratic system, `r=0` is the $+x$-direction, `r=1` the $-x$-direction, `r=2` the $+y$-direction and `r=3` the $-y$-direction. However, which convention you use depends on you, but you should make sure you are consistent. For the case of a quadratic lattice, it is `num_n=4`. Please note that whenever the macro `NEXT` is used, there must be a variable `num_n` defined, which stores the number of neighbors. You could include `num_n` as a third parameter of the macro, but in this case a call of the macro looks slightly more confusing. Nevertheless, the way you define such a macro depends on your personal preferences.
Please note that the `NEXT` macro cannot be realized by an inline function, in case you want to set values directly like in `NEXT(i,0)=i+1`. Also, when using an inline function, you would have to include all parameters explicitly, i.e. `num_n` in the example. The last requirement could be circumvented by using global variables, but this is bad programming style as well.
When the system is an Ising spin glass, the sign and magnitude of the interaction may be different for each pair of spins. The interaction strengths can be stored in a similar

way to the neighbor relation, e.g. in an array `j[]`. The access can be simplified via the macro J:

```
#define  J(i,r)  j[(i)*num_n + r]
```

A subroutine for calculating the energy $H = \sum_{\langle i,j \rangle} J_{ij}\sigma_i\sigma_j$ may look as follows, please note that the parameter `N` denotes the number of spins and the values of the spins are stored in the array `sigma[]`:

```
double spinglass_energy(int N, int num_n, int *next, int *j,
                        short int *sigma)
{
  double energy = 0.0;
  int i, r;                                          /* counters */

  for(i=1; i<=N; i++)                /* loop over all lattice sites */
    for(r=0; r<num_n; r++)              /* loop over all neighbors */
      energy += J(i,r)*sigma[i]*sigma[NEXT(i,r)];

  return(energy/2);     /* each pair has appeared twice in the sum */
}
```

For this piece of code the comments explaining the parameters and the purpose of the code are just missing for convenience. In the actual program it should be included. The code for `spinglass_energy()` is very short and clear. It works for all kinds of lattices. Only the subroutine where the array `next[]` is set up has to be rewritten when implementing a different type of lattice. This is true for all kinds of code realizing e.g. a Monte Carlo scheme or the calculation of a physical quantity. For free boundary conditions, additionally `sigma[0]=0` must be assigned to be consistent with the convention that missing neighbors have the id 0. This is the reason, why the spin site numbering starts with index 1 while C arrays start with index 0.

13.4.2 *Make* Files

If your software project grows larger, it will consist of several source-code files. Usually, there are many dependencies between the different files, e.g. a data type defined in one header file can be used in several modules. Consequently, when changing one of your source files, it may be necessary to recompile several parts of the program. In case you do not want to recompile your files every time by hand, you can transfer this task to the *make* tool which can be found on UNIX operating systems. A complete description of the abilities of *make* can be found in Ref. [12]. You should look on the *man* page (type `man make`) or in the texinfo file [13] as well. For other operating systems or software development environments, similar tools exists. Please consult the manuals in case you are not working with a UNIX type of operating system.
The basic idea of *make* is that you keep a file which contains all dependencies between your source code files. Furthermore, it contains commands (e.g. the compiler command) which generate the resulting files called *targets*, i.e. the final program and/or

object (`.o`) files. Each pair of dependencies and commands is called *rule*. The file containing all rules of a project is called *makefile*, usually it is named `Makefile` and should be placed in the directory where the source files are stored.
A rule can be coded by two lines of the form

target : sources

<tab> *command(s)*

The first line contains the dependencies, the second one the commands. The command line must begin with a tabulator symbol `<tab>`. It is allowed to have several targets depending on the same sources. You can extend the lines with the backslash "\" at the end of each line. The command line is allowed to be left empty. An example of a dependency/command pair is

```
simulation.o: simulation.c simulation.h
<tab>   cc -c simulation.c
```

This means that the file `simulation.o` has to be compiled if either `simulation.c` or `simulation.h` have been changed. The *make* program is called by typing `make` on the command line of a UNIX shell. It uses the date of the last changes, which is stored along with each file, to determine whether a rebuild of some targets is necessary. Each time at least one of the source files are newer than the corresponding target files, the commands given after the <tab> are called. Specifically, the command is called, if the target file does not exist at all. In this special case, no source files have to be given after the colon in the first line of the rule.
It is also possible to generate meta rules, which e.g. tell how to treat all files which have a specific suffix. Standard rules, how to treat files ending for example with `.c` are already included, but can be changed for each file by stating a different rule. This subject is covered in the *man* page of *make*.
The make tool always tries to build only the first object of your *makefile*, unless enforced by the dependencies. Hence, if you have to build several independent object files `object1, object2, object3`, the whole compiling must be toggled by the first rule, thus your *makefile* should read like this

```
all: object1 object2 object3

object1:  <sources of object1>
<tab>    <command to generate object1>

object2: ...
<tab>    <command to generate object2>

object3 ...
<tab>    <command to generate object3>
```

It is not necessary to separate different rules by blank lines. Here it is just for better readability. If you want to rebuild just e.g. `object3`, you can call `make object3`. This

allows several independent targets to be combined into one *makefile*. When compiling programs via `make`, it is common to include the target "clean" in the *makefile* such that all objects files are removed when `make clean` is called. Thus, the next call of `make` (without further arguments) compiles the whole program again from scratch. The rule for 'clean' reads like

```
clean:
<tab>   rm -f *.o
```

Also iterated dependencies are allowed, for example

```
object1: object2

object2: object3
<tab> ...

object3: ...
<tab> ...
```

The order of the rules is not important, except that *make* always starts with the first target. Please note that the *make* tool is not just intended to manage the software development process and toggle compile commands. Any project where some output files depend on some input files in an arbitrary way can be controlled. For example you could control the setting of a book, where you have text-files, figures, a bibliography and an index as input files. The different chapters and finally the whole book are the target files.
Furthermore, it is possible to define variables, sometimes also called macros. They have the format

variable=definition

Also variables belonging to your environment like `$HOME` can be referenced in the *makefile*. The value of a variable can be used, similar to shells variables, by placing a $ sign in front of the name of the variable, but you have to embrace the name by (...) or {...}. There are some special variables, e.g. $@ holds the name of the target in each corresponding command line, here no braces are necessary. The variable `CC` is predefined to hold the compiling command, you can change it by including for example

```
CC=gcc
```

in the *makefile*. In the command part of a rule the compiler is called via `$(CC)`. Thus, you can change your compiler for the whole project very quickly by altering just one line of the *makefile*.
Finally, it will be shown what a typical *makefile* for a small software project might look like. The resulting program is called `simulation`. There are two additional modules `init.c`, `run.c` and the corresponding header `.h` files. In `datatypes.h` types are defined which are used in all modules. Additionally, an external precompiled object file `analysis.o` in the directory `$HOME/lib` is to be linked, the corresponding header

file is assumed to be stored in `$HOME/include`. For `init.o` and `run.o` no commands are given. In this case `make` applies the predefined standard command for files having `.o` as suffix, which reads like

```
<tab>   $(CC) $(CFLAGS) -c $@
```

where the variable `CFLAGS` may contain options passed to the compiler and is initially empty. The *makefile* looks like this, please note that lines beginning with "#" are comments.

```
#
# sample make file
#
OBJECTS=simulation.o init.o run.o
OBJECTSEXT=$(HOME)/lib/analysis.o
CC=gcc
CFLAGS=-g -Wall -I$(HOME)/include
LIBS=-lm

simulation: $(OBJECTS) $(OBJECTSEXT)
<tab>   $(CC) $(CFLAGS) -o $@ $(OBJECTS) $(OBJECTSEXT) $(LIBS)

$(OBJECTS): datatypes.h

clean:
<tab>   rm -f *.o
```

The first three lines are comments, then five variables `OBJECTS`, `OBJECTSEXT`, `CC`, `CFLAGS` and `LIBS` are assigned. The final part of the *makefile* are the rules.
Please note that sometimes bugs are introduced, if the *makefile* is incomplete. For example consider a header file which is included in several code files, but this is not mentioned in the *makefile*. Then, if you change e.g. a data type in the header file, some of the code files might not be compiled again, especially those you did not change. Thus the same objects files can be treated with different formats in your program, yielding bugs which seem hard to explain. Hence, in case you encounter mysterious bugs, a `make clean` might help. But most of the time, bugs which are hard to explain are due to errors in your memory management. How to track down those bugs is explained in Sec. 13.7.
The `make` tool exhibits many other features. For additional details, please consult the references given above.

13.4.3 Scripts

Scripts are even more general tools than *make* files. They are in fact small programs, but they are usually not compiled, i.e. they are quickly written but they run slowly. Scripts can be used to perform many administration tasks like backing up data, installing software or running simulation programs for many different parameters. Here

only an example concerning the last task is presented. For a general introduction to scripts, please refer to a book on UNIX/Linux.
Assume that you have a simulation program called `coversim21` which calculates vertex covers of graphs. In case you do not know what a vertex cover is, it does not matter, just regard it as one optimization problem characterized by some parameters. You want to run the program for a fixed graph size `L`, for a fixed concentration `c` of the edges, average over `num` realizations and write the results to a file, which contains a string `appendix` in its name to distinguish it from other output files. Furthermore, you want to iterate over different relative sizes `x`. Then you can use the following script `run.scr`:

```
#!/bin/bash
L=$1
c=$2
num=$3
appendix=$4
shift
shift
shift
shift
for x
do
  ${HOME}/cover/coversim21 -mag  $L $c $x $num > \
                        mag_${c}_${x}${appendix}.out
done
```

The first line starting with "#" is a comment line, but it has a special meaning. It tells the operating system the language in which the script is written. In this case it is for the *bash* shell, the absolute pathname of the shell is given. Each UNIX shell has its own script language, you can use all commands which are allowed in the shell. There are also more elaborate script languages like *perl* or *phyton*, but they are not covered here.
Scripts can have command line arguments, which are referred via `$1`, `$2`, `$2` etc., the name of the script itself is stored in `$0`. Thus, in the lines 2 to 5, four variables are assigned. In general, you can use the arguments everywhere in the script directly, i.e. it is not necessary to store them in other variables. It is done here because in the next four lines the arguments `$1` to `$4` are thrown away by four `shift` commands. Then, the argument which was on position five at the beginning is stored in the first argument. Argument zero, containing the script name, is not affected by the shift.
Next, the script enters a loop, given by "`for x; do ... done`". This construction means that iteratively all remaining arguments are assigned to the variable "`x`" and each time the body of the loop is executed. In this case, the simulation is started with some parameters and the output directed to a file. Please note that you can state the loop parameters explicitly like in "`for size in 10 20 40 80 160; do ... done`". The above script can be called for example by

```
run.scr 100 0.5 1000 testA 0.20 0.22 0.24 0.26 0.28 0.30
```

which means that the graph size is 100, the fraction of edges is 0.5, the number of realizations per run is 100, the string `testA` appears in the output file name and the simulation is performed for the relative sizes 0.20, 0.22, 0.24, 0.26, 0.28, 0.30.

13.5 Libraries

Libraries are collections of subroutines and data types, which can be used in other programs. There are libraries for numerical methods such as integration or solving differential equations, for storing, sorting and accessing data, for fancy data types like lists or trees, for generating colorful graphics and for thousands of other applications. Some can be obtained for free, while other, usually specialized libraries have to be purchased. The use of libraries speeds up the software development process enormously, because you do not have to implement every standard method by yourself. Hence, you should always check whether someone has done the jobs for you already, before starting to write a program. Here, two standard libraries are briefly presented, providing routines which are needed for most computer simulations.
Nevertheless, sometimes it is inevitable to implement some methods by yourself. In this case, after the code has been proven to be reliable and useful for some time, you can put it in a self-created library. How to create libraries is explained in the last part of this section.

13.5.1 Numerical Recipes

The *Numerical Recipes* (NR) [3] contain a huge number of subroutines to solve standard numerical problems. Among them are:

- solving linear equations
- performing interpolations
- evaluation and integration of functions
- solving nonlinear equations
- minimizing functions
- diagonalization of matrices
- Fourier transform
- solving ordinary and partial differential equations.

The algorithms included are all state of the art. There are several libraries dedicated to similar problems, e.g. the library of the *Numerical Algorithms Group* [14] or the subroutines which are included with the *Maple* software package [15].

To give you an impression how the subroutines can be used, just a short example is presented. Consider the case that a symmetrical matrix is given and that all eigenvalues are to be determined. For more information on the library the reader should consult Ref. [3]. There it is not only shown how the library can be applied, but also all algorithms are explained.
The program to calculate the eigenvalues reads as follows.

```
#include <stdio.h>
#include <stdlib.h>
#include "nrutil.h"
#include "nr.h"

int main(int argc, char *argv[])
{
  float **m, *d, *e;                          /* matrix, two vectors */
  long n =  10;                                    /* size of matrix */
  int i, j;                                          /* loop counter */

  m = matrix(1, n, 1, n);                          /* allocate matrix */
  for(i=1; i<=n; i++)                  /* initialize matrix randomly */
    for(j=i; j<=n; j++)
    {
        m[i][j] = drand48();
        m[j][i] = m[i][j];          /* matrix must be symmetric here */
    }

  d = vector(1,n);          /* contains diagonal elements          */
  e = vector(1,n);          /* contains off diagonal elements      */
  tred2(m, n, d, e);        /* convert symmetric m. -> tridiagonal */
  tqli(d, e, n, m);         /* calculate eigenvalues               */
  for(j=1; j<=n; j++)       /* print result stored now in array 'd'*/
    printf("ev %d = %f\n", j, d[j]);

  free_vector(e, 1, n);                          /* give memory back */
  free_vector(d, 1, n);
  free_matrix(m, 1, n, 1, n);
  return(0);
}
```

In the first part of the program, an $n \times n$ matrix is allocated via the subroutine `matrix()` which is provided by *Numerical Recipes*. It is standard to let a vector start with index 1, while in C usually a vector starts with index 0.
In the second part a matrix is initialized randomly. Since the following subroutines work only for symmetric real matrices, the matrix is initialized symmetrically. The *Numerical Recipes* also provide methods to diagonalize arbitrary matrices, for simplicity this special case is chosen here .

In the third part the main work is done by the *Numerical Recipes* subroutines `tred2()` and `tqli()`. First, the matrix is written in tridiagonal form by a Householder transformation (`tred2()`) and then the actual eigenvalues are calculated by calling `tqli(d, e, n, m)`. The eigenvalues are returned in the vector `d[]` and the eigenvectors in the matrix `m[][]` (not used here), which is overwritten. Finally the memory allocated for the matrix and the vectors is freed again.
This small example should be sufficient to show how simply the subroutines from the *Numerical Recipes* can be incorporated into a program. When you have a problem of this kind you should always consult the NR library first, before starting to write code by yourself.

13.5.2 LEDA

While the *Numerical Recipes* are dedicated to numerical problems, the *Library of Efficient Data types and Algorithms* (LEDA) [4] can help a great deal in writing efficient programs in general. It is written in C++, but it can be used by C style programmers as well via mixing C++ calls to LEDA subroutines within C code. LEDA contains many basic and advanced data types such as:

- strings
- numbers of arbitrary precision
- one- and two-dimensional arrays
- lists and similar objects like stacks or queues
- sets
- trees
- graphs (directed and undirected, also labeled)
- dictionaries, there you can store objects with arbitrary key words as indices
- data types for two and three dimensional geometries, like points, segments or spheres

For most data types, it is possible to create arbitrary complex structures by using templates. For example you can make lists of self defined structures or stacks of trees. The most efficient implementations known in literature so far are taken for all data structures. Usually, you can choose between different implementations, to match special requirements. For every data type, all necessary operations are included; e.g. for lists: creating, appending, splitting, printing and deleting lists as well as inserting, searching, sorting and deleting elements in a list, also iterating over all elements of a list. The major part of the library is dedicated to graphs and related algorithms. You will find for example subroutines to calculate strongly connected components, shortest paths, maximum flows, minimum cost flows and (minimum) matchings.

Here again, just a short example is given to illustrate how the library can be utilized and to show how easy LEDA can be used. A list of a self defined class `Mydatatype` is considered. Each element contains the data entries `info` and `flag`. In the first part of the program below, the class `Mydatatype` is partly defined. Please note that input and output stream operators `<</>>` must be provided to be able to create a list of `Mydatatype` elements, otherwise the program will not compile. In the main part of the program a list is defined via the LEDA data type `list`. Elements are inserted into the list with `append()`. Finally an iteration over all list elements is performed using the LEDA macro `forall`. The program `leda_test.cc` reads as follows:

```
#include <iostream.h>
#include <LEDA/list.h>

class Mydatatype                    // self defined example class
{
 public:
    int       info;                                  // user data 1
    short int flag;                                  // user data 2
    Mydatatype() {info=0; flag=0;};                  // constructor
    ~Mydatatype() {};                                // destructor
    friend ostream& operator<<(ostream& O, const Mydatatype& dt)
        { O << "info: " << dt.info << " flag: " << dt.flag << "\n";
          return(O);};                               // output operator
    friend istream& operator>>(istream &I, Mydatatype& dt)
        {return(I);};                                // dummy
};

int main(int argc, char *argv[])
{
  list<Mydatatype> l;          // list with elements of 'Mydatatype'
  Mydatatype element;
  int t;

  for(t=0; t<10; t++)          // create list
  {
    element.info = t;
    element.flag = t%2;
    l.append(element);
  }
  forall(element, l)           // iterate over all elements
    if(element.flag)           // print only 'even' elements
      cout << element;
  return(0);
}
```

The program has to be compiled with a C++ compiler. Depending on your system, you have to specify some compiler flags to include LEDA, please consult your systems documentation or the system administrator. The compile command may look like this:

```
g++ -I$LEDAROOT/incl -L$LEDAROOT -o leda_test leda_test.cc -lG -lL
```

The `-I` flag specifies where the compiler searches for header files like `LEDA/list.h`, the `-L` flag tells where the libraries (`-lG -lL`) are located. The environment variable `LEDAROOT` must point to the directory where LEDA is stored in your system.
Please note that using *Numerical Recipes* and LEDA together results in conflicts, since the objects `vector` and `matrix` are defined in both libraries. You can circumvent this problem by taking the source code of *Numerical Recipes* (here: `nrutil.c`, `nrutil.h`) and rename the subroutines `matrix()` and `vector()`, compile again and include `nrutil.o` directly in your program.
Here, it should be stressed: Before trying to write everything by yourself, you should check whether someone else has done it for you already. LEDA is a highly effective and very convenient tool. It will save you a lot of time and effort when you use it for your program development.

13.5.3 Creating your own Libraries

Although many useful libraries are available, sometimes you have to write some code by yourself. Over the years you will collect many subroutines, which – if properly designed – can be included in other programs, in which case it is convenient to put these subroutines in a library. Then you do not have to include the object file every time you compile one of your programs. If your self-created library is put in a standard search path, you can access it like a system library, you even do not have to remember where the object file is stored.
To create a library you must have an object file, e.g. `tasks.o`, and a header file `tasks.h` where all data types and function prototypes are defined. Furthermore, to facilitate the use of the library, you should write a *man* page, which is not necessary for technical reasons but results in a more convenient usage of your library, particularly should other people want to benefit from it. To learn how to write a *man* page you should consult `man man` and have a look at the source code of some *man* pages, they are stored e.g. in `/usr/man`.
A library is created with the UNIX command `ar`. To include `tasks.o` in your library `libmy.a` you have to enter

```
ar r libmy.a tasks.o
```

In a library several object files may be collected. The option "`r`" replaces the given object files, if they already belong to the library, otherwise they are added. If the library does not exist yet it is created. For more options, please refer to the *man* page of `ar`.
After including an object file, you have to update an internal object table of the library. This is done by

```
ar s libmy.a
```

Now you can compile a program `prog.c` using your library via

```
cc -o prog prog.c libmy.a
```

In case `libmy.a` contains several object files, it saves some typing by just writing `libmy.a`, furthermore you do not have to remember the names of all your object files. To make the handling of the library more comfortable, you can create a directory, e.g. `~/lib` and put your libraries there. Additionally, you should create the directory `~/include` where all personal header files can be collected. Then your compile command may look like this:

```
cc -o prog prog.c -I$HOME/include -L$HOME/lib -lmy
```

The option `-I` states the search path for additional header files, the `-L` option tells the linker where your libraries are stored and via `-lmy` the library `libmy.a` is actually included. Please note that the prefix `lib` and the postfix `.a` are omitted with the `-l` option. Finally, it should be pointed out, that the compiler command given above works in all directories, once you have set up the structure as explained. Hence, you do not have to remember directories or names of object files.

13.6 Random Numbers

For many simulations in physics, random numbers are necessary. Quite often the model itself exhibits random parameters which remain fixed throughout the simulation, one speaks of *quenched disorder*. A famous example are spin glasses. In this case one has to perform an average over different realizations of the disorder, to obtain physical quantities.
But even when the system which is treated is not random, very often random numbers are required by the algorithms, e.g. to realize a finite-temperature ensemble or when using randomized algorithms. In this section an introduction to the generation of random numbers is given. First it is explained how they can be generated at all on a computer. Then, different methods for obtaining numbers are explained, which obey a given distribution: the *inversion method*, the *Box-Müller method* and the *rejection method*. More comprehensive information about these and similar techniques can be found in Refs. [3, 16].
In this section it is assumed that you are familiar with the basic concepts of probability theory and statistics.

13.6.1 Generating Random Numbers

First, it should be pointed out that standard computers are deterministic machines. Thus, it is completely impossible to generate true random numbers, at least not without the help of the user. It is for example possible to measure the time interval between successive keystrokes, which is randomly distributed by nature. But they depend heavily on the current user and it is not possible to reproduce an experiment

in exactly the same way. This is the reason why *pseudo random numbers* are usually taken. They are generated by deterministic rules, but they look like and have many of the properties of true random numbers. One would like to have a random number generator `rand()`, such that each possible number has the same probability of occurrence. Each time `rand()` is called, a new random number is returned. Additionally, if two numbers r_i, r_k differ only slightly, the random numbers r_{i+1}, r_{k+1} returned by the respective subsequent calls should have a low correlation.
The simplest methods to generate pseudo random numbers are *linear congruential generators*. They generate a sequence $I_1, I_2, \ldots$ of integer numbers between 0 and $m-1$ by a recursive recipe:

$$I_{n+1} = (aI_n + c) \mathrm{mod} m \tag{13.1}$$

To generate random numbers r distributed in the interval $[0,1)$ one has to divide the current random number by m. It is desirable to obtain equally distributed values in the interval, i.e. a uniform distribution. Below, you will see, how random numbers obeying other distributions can be generated from uniformly distributed numbers.
The real art is to choose the parameters a, c, m in a way that "good" random numbers are obtained, where "good" means "with less correlations". In the past several results from simulations have been turned out to be wrong, because of the application of bad random number generators [17].

Example: Bad and good generators

To see what "bad generator" means, consider as an example the parameters $a = 12351, c = 1, m = 2^{15}$ and the seed value $I_0 = 1000$. 10000 random numbers are generated, by dividing each of them by m, they are distributed in the interval $[0,1)$. In Fig. 13.2 the distribution of the random numbers is shown.

The distribution looks rather flat, but by taking a closer look some regularities can be observed. These regularities can be studied by recording k-tuples of k successive random numbers $(x_i, x_{i+1}, \ldots, x_{i+k-1})$. A good random number generator, exhibiting no correlations, would fill up the k-dimensional space uniformly. Unfortunately, for linear congruential generators, instead the points lie on $(k-1)$-dimensional planes. It can be shown that there are *at most* of the order $m^{1/k}$ such planes. A bad generator has much fever planes. This is the case for the example studied above, see top part of Fig. 13.3

The result for $a = 123450$ is even worse, only 15 different "random" numbers are generated (with seed 1000), then the iteration reaches a fixed point (not shown in a figure).

If instead $a = 12349$ is chosen, the two-point correlations look like that shown in the bottom half of Fig. 13.3. Obviously, the behavior is much more irregular, but poor correlations may become visible for higher k-tuples. □

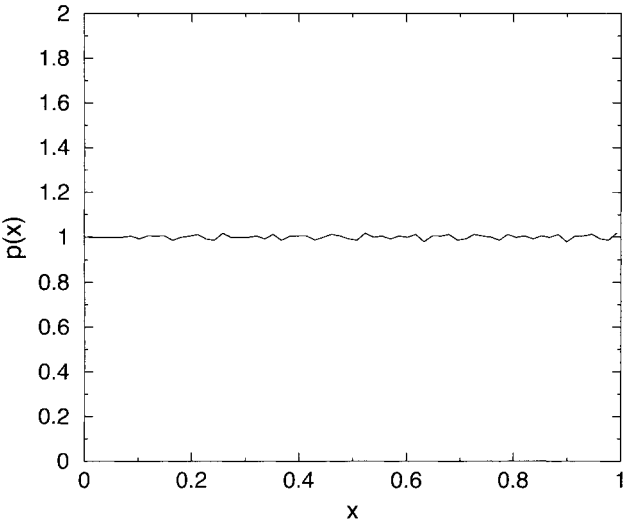


Figure 13.2: Distribution of random numbers in the interval $[0, 1)$. They are generated using a linear congruential generator with the parameters $a = 12351, c = 1, m = 2^{15}$.

A generator which has passed several theoretical test is $a = 7^5 = 16807$, $m = 2^{31} - 1$, $c = 0$. When implementing this generator you have to be careful, because during the calculation numbers are generated which do not fit into 32 bit. A clever implementation is presented in Ref. [3]. Finally, it should be stressed that this generator, like all linear congruential generators, has the low-order bits much less random than the high-order bits. For that reason, when you want to generate integer numbers in an interval [1,N], you should use

```
r = 1+(int) (N*(I_n)/m);
```

instead of using the modulo operation as with `r=1+(I_n % N);`.

So far it has been shown how random numbers can be generated which are distributed uniformly in the interval $[0, 1)$. In general, one is interested in obtaining random numbers which are distributed according to a given probability distribution with density $p(z)$. In the next sections, several techniques performing this task for continuous probability distributions are presented.

In case of discrete distributions, one has to create a table of the possible outcomes with their probabilities p_i. To draw a number, one has to draw a random number u which is uniformly distributed in $[0, 1)$ and take the entry j of the table such that the sum $\sum_{i=1}^{j} p_i$ of the preceding probabilities is larger than u, but $\sum_{i=1}^{j-1} p_i < u$. In the following, we concentrate on techniques for generating continuous random variables.

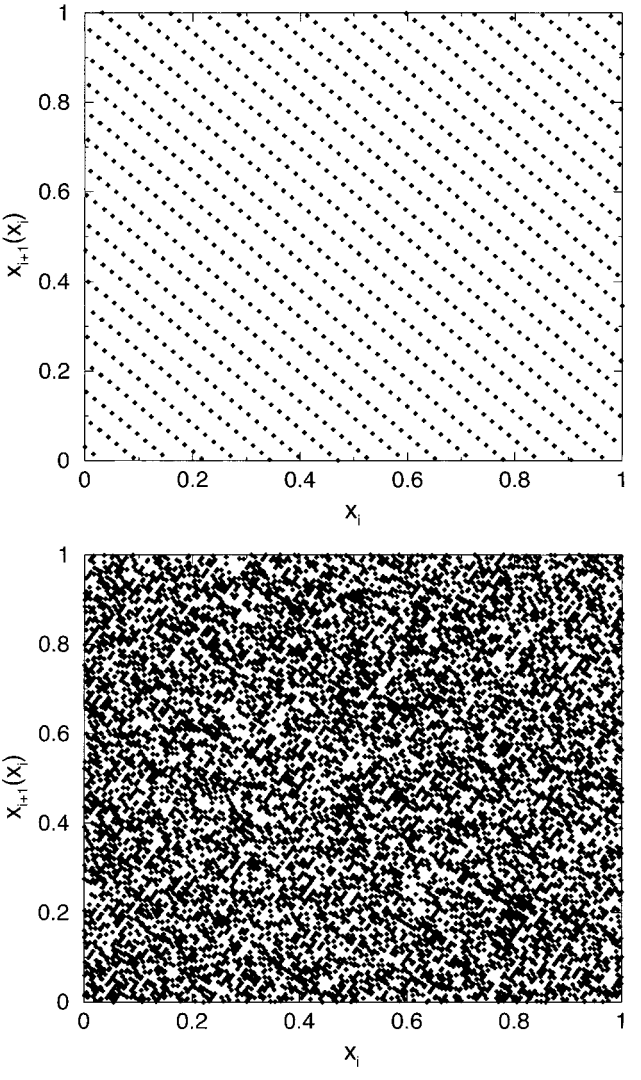


Figure 13.3: Two point correlations $x_{i+1}(x_i)$ between successive random numbers x_i, x_{i+1}. The top case is generated using a linear congruential generator with the parameters $a = 12351, c = 1, m = 2^{15}$, the bottom case has instead $a = 12349$.

13.6.2 Inversion Method

Given is a random number generator *drand*() which is assumed to generate random numbers U which are distributed uniformly in $[0, 1)$. The aim is to generate random

numbers Z with probability density $p(z)$. The corresponding distribution function is

$$P(z) \equiv \text{Prob}(Z \leq z) \equiv \int_{-\infty}^{z} dz' p(z') \tag{13.2}$$

The target is to find a function $g(X)$, such that after the transformation $Z = g(U)$, the values of Z are distributed according to (13.2). It is assumed that g can be inverted and is strongly monotonically increasing, then one obtains

$$P(z) = \text{Prob}(Z \leq z) = \text{Prob}(g(U) \leq z) = \text{Prob}(U \leq g^{-1}(z)) \tag{13.3}$$

Since the distribution function $F(u) = \text{Prob}(U \leq u)$ for a uniformly distributed variable is just $F(u) = u$ ($u \in [0,1]$), one obtains $P(z) = g^{-1}(z)$. Thus, one just has to choose $g(z) = P^{-1}(z)$ for the transformation function, in order to obtain random numbers, which are distributed according the probability distribution $P(z)$. Of course, this only works if P can be inverted.

Example: Exponential distribution

Let us consider the exponential distribution with parameter λ, with probability density

$$p(z) = \lambda \exp(-\lambda z) \tag{13.4}$$

and distribution function $P(z) = 1 - \exp(-\lambda z)$. Therefore, one can obtain exponentially distributed random numbers Z, by generating uniform distributed random numbers U and choosing $Z = -\ln(1-U)/\lambda$.

In Fig. 13.4 a histogram for 10^5 random numbers generated in this way and the exponential probability function for $\lambda = 1$ are shown with a logarithmically scaled y-axis. Only for larger values are deviations visible. They are due to statistical fluctuations since $p(z)$ is very small there.

For completeness, this example is finished by mentioning that by summing n independent exponentially distributed random numbers, the result is gamma distributed [16]. □

13.6.3 Rejection Method

As mentioned above, the inversion method works only when the distribution function P can be inverted. For distributions not fulfilling this condition, sometimes this problem can be overcome by drawing several random numbers and combining them in a clever way, see e.g. the next subsection.

The *rejection method*, which is presented in this section, works for random variables where the probability distribution $p(z)$ fits into a box $[x_0, x_1) \times [0, z_{\max})$, i.e. $p(z) = 0$ for $z \notin [x_0, x_1]$ and $p(z) \leq z_{\max}$. The basic idea of generating a random number distributed according to $p(z)$ is to generate random pairs (x, y), which are distributed

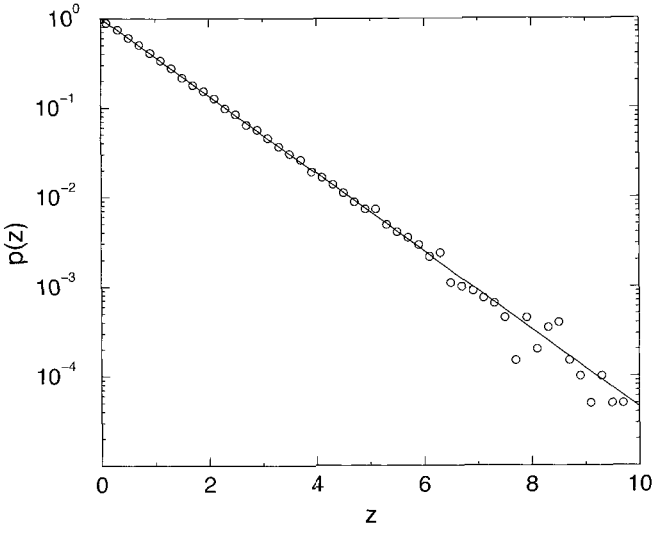


Figure 13.4: Histogram of random numbers generated according to an exponential distribution ($\lambda = 1$) compared with the probability density (straight line) in a logarithmic plot.

uniformly in $[x_0, x_1] \times [0, z_{\max}]$ and accept only those values x where $y \leq p(x)$ holds, i.e. the pairs which are located below $p(x)$, see Fig. 13.5. Therefore, the probability that x is drawn is proportional to $p(x)$, as desired. The algorithm for the rejection method is:

algorithm rejection_method($z_{\max}, p$)
begin
  $found$:= **false**;
  while not $found$ **do**
  begin
    u_1 := random number in $[0, 1)$;
    $x := x_0 + (x_1 - x_0) \times u_1$;
    u_2 := random number in $[0, 1)$;
    $y := z_{\max} \times u_2$;
    if $y \leq p(x)$ **then**
      $found$:= **true**;
  end;
  return(x);
end

The rejection method always works if the probability density is boxed, but it has the drawback that more random numbers have to be generated than can be used.

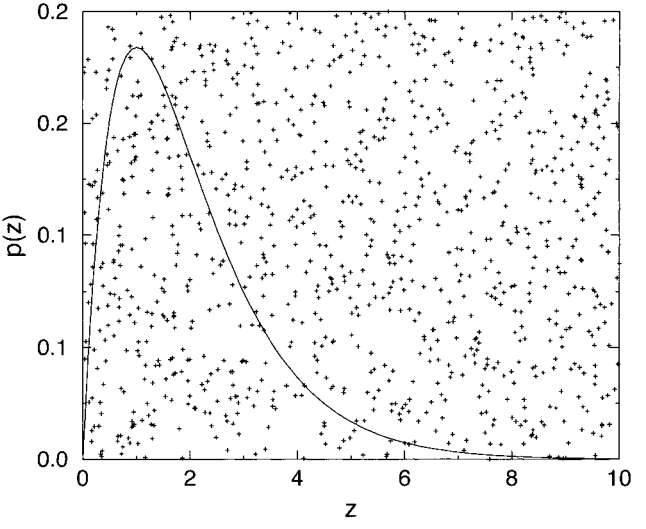


Figure 13.5: The rejection method: points (x, y) are scattered uniformly over a bounded rectangle. The probability that $y \leq p(x)$ is proportional to $p(x)$.

In case neither the distribution function can be inverted nor the probability fits into a box, special methods have to be applied. As an example a method for generating random numbers distributed according to a Gaussian distribution is considered. Other methods and examples of how different techniques can be combined, are collected in Ref. [16].

13.6.4 The Gaussian Distribution

The probability density for the Gaussian distribution with mean m and width σ is (see also Fig. 13.6)

$$p_G(z) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left(\frac{(z-m)^2}{2\sigma^2}\right) \tag{13.5}$$

It is, apart from uniform distributions, the most common distribution being applied in simulations.

Here, the case of a normal distribution ($m = 0$, $\sigma = 1$) is considered. If you want to realize the general case, you have to draw a normally distributed number z and then use $\sigma z + m$ which is distributed as desired.

Since the normal distribution extends over an infinite interval and cannot be inverted, the methods from above are not applicable. The simplest technique to generate random numbers distributed according to a normal distribution makes use of the central limit theorem. It tells us that any sum of N independently distributed random variables

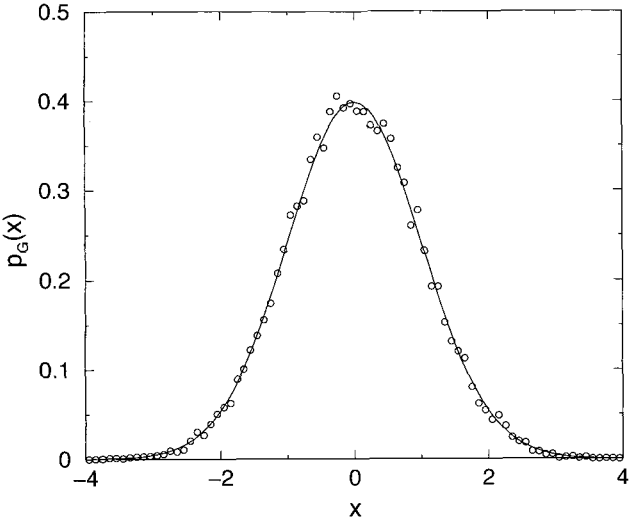


Figure 13.6: Gaussian distribution with zero mean and unit width. The circles represent a histogram obtained from 10^4 values drawn with the Box-Müller method.

u_i (with mean m and variance v) will converge to a Gaussian distribution with mean Nm and variance Nv. If again u_i is taken take to be uniformly distributed in $[0, 1)$ (which has mean $m = 0.5$ and variance $v = 1/12$), one can choose $N = 12$ and $Z = \sum_{i=1}^{12} u_i - 6$ will be distributed approximately normally. The drawback of this method is that 12 random numbers are needed to generate one final random number and that values larger than 6 never appear.
In contrast to this technique the *Box-Müller method* is exact. You need two uniformly in $[0, 1)$ distributed random variables U_1, U_2 to generate two independent normal variables N_1, N_2. This can be achieved by setting

$$N_1 = \sqrt{-2\log(1-u_1)}\cos(2\pi u_2)$$
$$N_2 = \sqrt{-2\log(1-u_1)}\sin(2\pi u_2)$$

A proof that N_1 and N_2 are indeed distributed according to (13.5) can be found in Refs. [3, 16], where also other methods for generating Gaussian random numbers, some even more efficient, are explained. A method which is based on the simulation of particles in a box is explained in Ref. [18]. In Fig. 13.6 a histogram of 10^4 random numbers drawn with the Box-Müller method is shown.

13.7 Tools for Testing

In Sec. 13.1 the importance of thorough testing has already been stressed. Here three useful tools are presented which significantly assist in facilitating the debugging pro-

cess. Please note again that the tools run under UNIX/Linux operating systems. Similar programs are available for other operating systems as well. The tools covered here are *gdb*, a source-code debugger, *ddd*, a graphic front-end to gdb, and *checkergcc*, which finds bugs resulting from bad memory management.

13.7.1 *gdb*

The *gdb* gnu debugger tool is a *source code debugger*. Its main purpose is that you can watch the execution of your code. You can stop the program at arbitrarily chosen points by setting *breakpoints* at lines or subroutines in the source code, inspect variables/data structures, change them and let the program continue (e.g. line by line). Here some examples for the most basic operations are given, detailed instructions can be obtained within the program via the `help` command.
As an example of how to debug, please consider the following little program `gdbtest.c`:

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
    int t, *array, sum = 0;

    array = (int *) malloc (100*sizeof(int));
    for(t=0; t<100; t++)
        array[t] = t;
    for(t=0; t<100; t++)
        sum += array[t];
    printf("sum= %d\n", sum);
    free(array);
    return(0);
}
```

When compiling the code you have to include the option `-g` to allow debugging:

```
cc -o gdbtest -g gdbtest.c
```

The debugger is invoked using `gdb` <*programname*>, i.e.

```
gdb gdbtest
```

Now you can enter commands, e.g. list the source code of the program via the `list` command, it is sufficient to enter just `l`. By default always ten lines at the current position are printed. Therefore, at the beginning the first ten lines are shown (the first line shows the input, the other lines state the answer of the debugger)

```
(gdb) l
1       #include <stdio.h>
2       #include <stdlib.h>
3
```

```
4       int main(int argc, char *argv[])
5       {
6           int t, *array, sum = 0;
7
8           array = (int *) malloc (100*sizeof(int));
9           for(t=0; t<100; t++)
10              array[t] = t;
```

When entering the command again the next ten lines are listed. Furthermore, you can refer to program lines of the code in the form `list` <*from*>, <*to*> or to subroutines by typing `list` <*name of subroutine*>. More information can be obtained by typing `help list`.
To let the execution stop at a specific line one can use the `break` command (abbreviation `b`). To stop the program *before* line 11 is executed, one enters

```
(gdb) b 11
Breakpoint 1 at 0x80484b0: file gdbtest.c, line 11.
```

Breakpoints can be removed via the `delete` command. All current breakpoints are displayed by entering `info break`.
To start the execution of the program, one enters `run` or just `r`. As requested before, the program will stop at line 11:

```
(gdb) r
Starting program: gdbtest

Breakpoint 1, main (argc=1, argv=0xbffff384) at gdbtest.c:11
11          for(t=0; t<100; t++)
```

Now you can inspect for example the content of variables via the `print` command:

```
(gdb) p array
$1 = (int *) 0x8049680
(gdb) p array[99]
$2 = 99
```

To display the content of a variable permanently, the `display` command is available. You can change the content of variables via the `set` command

```
(gdb) set array[99]=98
```

You can continue the program at each stage by typing `next`, then just the next source-code line is executed:

```
(gdb) n
12              sum += array[t];
```

Subroutines are regarded as one source-code line as well. If you want to debug the subroutine in a step-wise manner as well you have to enter the `step` command. By entering `continue`, the execution is continued until the next breakpoint, a severe error, or the end of the program is reached, please note the the output of the program appears in the *gdb* window as well:

```
(gdb) c
Continuing.
sum= 4949

Program exited normally.
```

As you can see, the final value (4949) the program prints is affected by the change of the variable `array[99]`.
The above given commands are sufficient for most of the standard debugging tasks. For more specialized cases *gdb* offers many other commands, please have a look at the documentation [5].

13.7.2 *ddd*

Some users may find graphical user interfaces more convenient. For this reason there exists a graphical front-end to the gdb, the *data display debugger (ddd)*. On UNIX operating systems it is just invoked by typing ddd (see also *man* page for options). Then a nice windows pops up, see Fig. 13.7. The lower part of the window is an ordinary *gdb* interface, several other windows are available. By typing `file` <*program*> you can load a program into the debugger. Then the source code is shown in the main window of the debugger. All *gdb* commands are available, the most important ones can be entered via menus or buttons using the mouse. For example to set a breakpoint it is sufficient to place the cursor in a source-code line in the main ddd window and click on the *break* button. A good feature is that the content of a variable is shown when moving the mouse onto it. For more details, please consult the online help of ddd.

13.7.3 *checkergcc*

Most program bugs are revealed by systematically running the program and cross-checking with the expected results. But other errors seem to appear in a rather irregular and unpredictable fashion. Sometimes a program runs without a problem, in other cases it crashes with a `Segmentation fault` at rather puzzling locations in the code. Very often a bad memory management is the cause of such a behavior. Writing beyond the boundaries of an array, reading uninitialized memory locations or addressing data which has been freed already are the most common bugs of this class. Since the operating system organizes the memory in a different way each time a program is run, it is rather unpredictable whether these errors become apparent or not. Furthermore it is very hard to track them down, because the effect of such errors most of the time becomes visible at positions different from where the error has occurred.
As an example, the case where it is written beyond the boundary of an array is considered. If in the heap, where all dynamically allocated memory is taken from, at the location behind the array another variable is stored, it will be overwritten in this case. Hence, the error becomes visible the next time the other variable is read. On

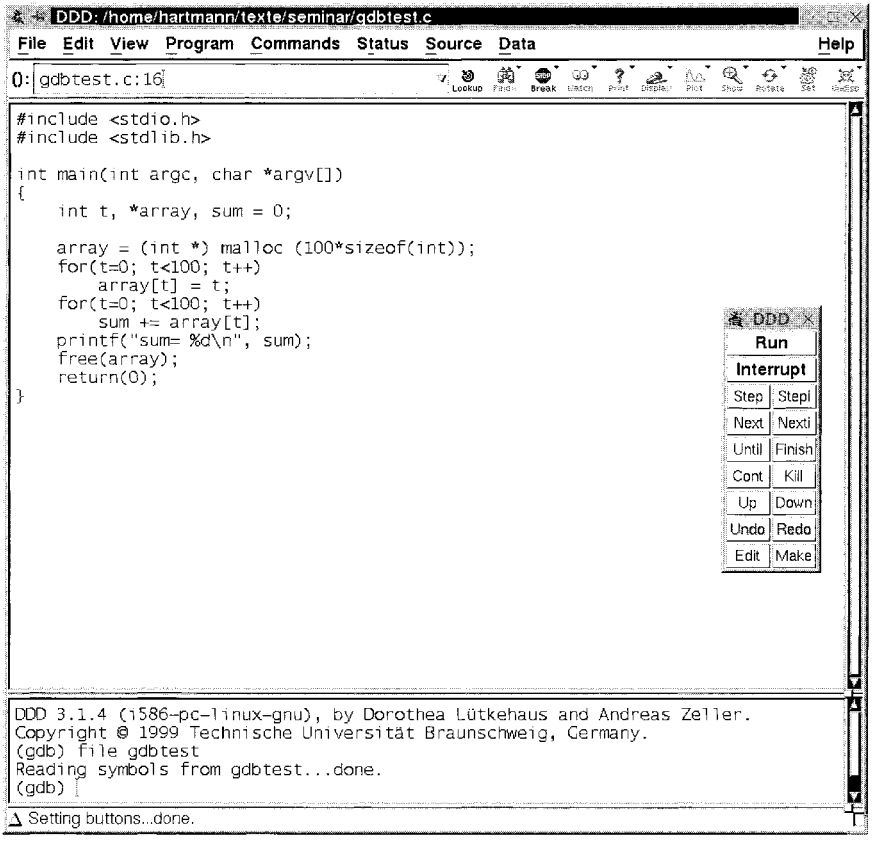


Figure 13.7: The data display debugger (ddd). In the main window the source code is shown. Commands can be invoked via a mouse or by entering them into the lower part of the window.

the other hand, if the memory block behind the array is not used, the program may run that time without any problems. Unfortunately, the programmer is not able to influence the memory management directly.

To detect such types of nasty bugs, one can take advantage of several tools. A list of free and commercial tools can be found in Ref. [19]. Here *checkergcc* is considered, which is a very convenient tool and freely available. It works under UNIX and is included by compiling everything with `checkergcc` instead of `cc` or `gcc`. Unfortunately, the current version does not have full support for C++, but you should try it on your own project. The checkergcc compiler replaces all memory allocations/deallocations and accesses by its own routines. Any access to non-authorized memory locations is reported, regardless of the positions of other variables in the memory area (heap).

As an example, the program from Sec. 13.7.1 is considered, which is slightly modified;

the memory block allocated for the array is now slightly too short (length 99 instead of 100):

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
    int t, *array, sum = 0;

    array = (int *) malloc (99*sizeof(int));
    for(t=0; t<100; t++)
        array[t] = t;
    for(t=0; t<100; t++)
        sum += array[t];
    printf("sum= %d\n", sum);
    free(array);
    return(0);
}
```

The program is compiled via

```
checkergcc -o gdbtest -g gdbtest.c
```

Starting the program produces the following output, the program terminates normally:

```
Sisko:seminar>gdbtest
Checker 0.9.9.1 (i686-pc-linux-gnu) Copyright (C) 1998 Tristan Gingold.
This program has been compiled with 'checkergcc' or 'checkerg++'.
Checker is a memory access detector.
Checker is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
General Public License for more details.
For more information, set CHECKEROPTS to '--help'
From Checker (pid:30448): 'gdbtest' is running

From Checker (pid:30448): (bvh) block bounds violation in the heap.
When Writing 4 byte(s) at address 0x0805fadc, inside the heap (sbrk).
0 byte(s) after a block (start: 0x805f950, length: 396, mdesc: 0x0).
The block was allocated from:
        pc=0x080554f9 in chkr_malloc at stubs-malloc.c:57
        pc=0x08048863 in main at gdbtest.c:8
        pc=0x080555a7 in this_main at stubs-main.c:13
        pc=0x40031c7e in __divdi3 at stubs/end-stubs.c:7
        pc=0x08048668 in *unknown* at *unknown*:0
Stack frames are:
        pc=0x080489c3 in main at gdbtest.c:10
```

```
        pc=0x080555a7 in this_main at stubs-main.c:13
        pc=0x40031c7e in __divdi3 at stubs/end-stubs.c:7
        pc=0x08048668 in *unknown* at *unknown*:0
From Checker (pid:30448): (bvh) block bounds violation in the heap.
When Reading 4 byte(s) at address 0x0805fadc, inside the heap (sbrk).
0 byte(s) after a block (start: 0x805f950, length: 396, mdesc: 0x0).
The block was allocated from:
        pc=0x00000063 in *unknown* at *unknown*:0
        pc=0x08048863 in main at gdbtest.c:8
        pc=0x080555a7 in this_main at stubs-main.c:13
        pc=0x40031c7e in __divdi3 at stubs/end-stubs.c:7
        pc=0x08048668 in *unknown* at *unknown*:0
Stack frames are:
        pc=0x08048c55 in main at gdbtest.c:12
        pc=0x080555a7 in this_main at stubs-main.c:13
        pc=0x40031c7e in __divdi3 at stubs/end-stubs.c:7
        pc=0x08048668 in *unknown* at *unknown*:0
```

Two errors are reported, each message starts with "`From checker`". Both errors consist of accesses to an array beyond the border (`block bound violation`). For each error both the location in the source code where the memory has been allocated and the location where the error occurred (`Stack frames`) are given. In both cases the error is concerned with what was allocated at line 8 (`pc=0x08048863 in main at gdbtest.c:8`). The bug appeared during the loops over the array, when the array is initialized (line 10) and read out (line 12).

Other common types of errors are memory leaks. They appear when a previously used block of memory has been forgotten to be freed again. Assume that this happens in a subroutine which is called frequently in a program. You can imagine that you will quickly run out of memory. Memory leaks are not detected using checkergcc by default. This kind of test can be turned on by setting a special environment variable `CHECKEROPTS`, which controls the behavior of `checkergcc`. To enable checking for memory leaks at the end of the execution, one has to set

```
export CHECKEROPTS="-D=end"
```

Let us assume that the bug from above is removed and instead the `free(array);` command at the end of the program is omitted. After compiling with `checkergcc`, running the program results in:

```
From Checker (pid:30900): 'gdbtest' is running

sum= 4950
Initialization of detector...
Searching in data
Searching in stack
Searching in registers
From Checker (pid:30900): (gar) garbage detector results.
```

```
There is 1 leak and 0 potential leak(s).
Leaks consume 400 bytes (0 KB) / 132451 KB.
( 0.00% of memory is leaked.)
Found 1 block(s) of size 400.
Block at ptr=0x805f8f0
        pc=0x08055499 in chkr_malloc at stubs-malloc.c:57
        pc=0x08048863 in main at gdbtest.c:8
        pc=0x08055547 in this_main at stubs-main.c:13
        pc=0x40031c7e in __divdi3 at stubs/end-stubs.c:7
        pc=0x08048668 in *unknown* at *unknown*:0
```

Obviously, the memory leak has been found. Further information on the various features of checkergcc can be found in Ref. [20]. A last hint: you should always test a program with a memory checker, even if everything seems to be fine.

13.8 Evaluating Data

To analyze and plot data, several commercial and non-commercial programs are available. Here three free programs are discussed, *gnuplot*, *xmgr* and *fsscale*. *Gnuplot* is small, fast, allows two- and three-dimensional curves to be generated and to fit arbitrary functions to the data. On the other hand *xmgr* is more flexible and produces better output. It is recommended that *gnuplot* is used for viewing and fitting data online, while *xmgr* is to be preferred for producing figures to be shown in talks or publications. The program fsscale has a special purpose. It is very convenient for performing finite-size scaling plots.
First, *gnuplot* and *xmgr* are introduced with respect to drawing figures. In the next subsection, data fitting is covered. Finally, it is shown how finite-size scaling plots can be created. In all three cases only very small examples can be presented. They should serve just as a motivation to study the documentation, then you will learn about the manifold potential the programs offer.

13.8.1 Data Plotting

The program *gnuplot* is invoked by entering `gnuplot` in a shell, for a complete manual see Ref. [13]. As always, our examples refer to a UNIX window system like X11, but the program is available for almost all operating systems. After startup, the prompt (e.g. `gnuplot>`) appears and the user can enter commands in textual form, results are shown in windows or are written into files. Before giving an example, it should be pointed out that gnuplot *scripts* can be generated by simply writing the commands into a file, e.g. `command.gp`, and calling `gnuplot command.gp`.
The typical case is that you have a data file of $x - y$ data and you want to plot the figure. Your file might look like this, it is the ground-state energy of a three-dimensional $\pm J$ spin glass as a function of the linear system size L. The filename is `sg_e0_L.dat`. The first column contains the L values, the second the energy values

and the third the standard error of the energy, please note that lines starting with "#" are comment lines which are ignored on reading:

```
# ground state energy of +-J spin glasses
# L     e_0    error
  3 -1.6710 0.0037
  4 -1.7341 0.0019
  5 -1.7603 0.0008
  6 -1.7726 0.0009
  8 -1.7809 0.0008
 10 -1.7823 0.0015
 12 -1.7852 0.0004
 14 -1.7866 0.0007
```

To plot the data enter

```
gnuplot> plot "sg_e0_L.dat" with yerrorbars
```

which can be abbreviated as `p "sg_e0_L.dat" w e`. Please do not forget the quotation marks around the file name. Next, a window pops up, showing the result, see Fig. 13.8.

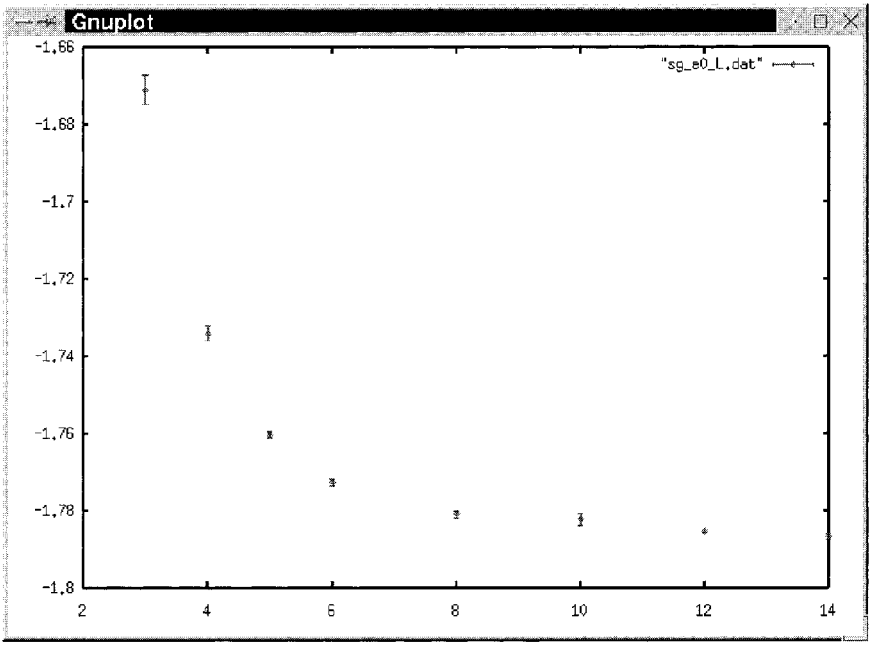


Figure 13.8: Gnuplot window showing the result of a plot command.

For the `plot` command many options and styles are available, e.g. `with lines` produces lines instead of symbols. It is possible to read files with multi columns via the

`using` option, e.g.

```
gnuplot> plot "test.dat" using 1:4:5 w e
```

displays the fourth column as a function of the first, with error bars given by the 5th column. Among other options, it is possible to redirect the output, for example to an encapsulated postscript file (by setting `set terminal postscript` and redirecting the output `set output "test.eps"`). Also several files can be combined into one figure. You can set axis labels of the figure by typing e.g. `set xlabel "L"`, which becomes active when the next plot command is executed. Online help on the plot command and its manifold options is available via entering `help plot`. Also three-dimensional plotting is possible using the `splot` command (enter `help splot` to obtain more information). For a general introduction you can type just `help`. Since *gnuplot* commands can be entered very quickly, you should use it for online viewing data and fitting (see Sec. 13.8.2).
The *xmgr* (x motiv graphic) program is much more powerful than *gnuplot* and produces nicer output, commands are issued by clicking on menus and buttons. On the other hand its handling is a little bit slower and the program has the tendency to fill your screen with windows. To create a similar plot to that above, you have to go (after staring it by typing `xmgr` into a shell) to the files menu and choose the read submenu and the sets subsubmenu. Then a file selection window will pop up and you can choose the data file to be loaded. The situation is shown in Fig. 13.9.
The *xmgr* program offers almost every feature you can imagine for two-dimensional data plots, including multiple plots, fits, many styles for lines, symbols, bar charts etc. Also you can create manifold types of labels or legends and it is possible to add elements like strings, lines or other geometrical objects in the plot. For more information, please consult the online help.

13.8.2 Curve Fitting

Both programs presented above, *gnuplot* and *xmgr*, offer fitting of arbitrary functions. It is advisable to use *gnuplot*, since it offers a higher flexibility for that purpose and gives you more information useful to estimate the quality of a fit.
As an example, let us suppose that you want to fit an algebraic function of the form $f(L) = e_\infty + aL^b$ to the data set of the file `sg_e0_L.dat` shown above. First, you have to define the function and supply some roughly (non-zero) estimations for the unknown parameters, please note that the exponential operator is denoted by `**` and the standard argument for a function definition is `x`, but this depends only on your choice:

```
gnuplot> f(x)=e+a*x**b
gnuplot> e=-1.8
gnuplot> a=1
gnuplot> b=-1
```

The actual fit is performed via the `fit` command. The program uses the nonlinear least-squares Marquardt-Levenberg algorithm [3], which allows a fit according to al-

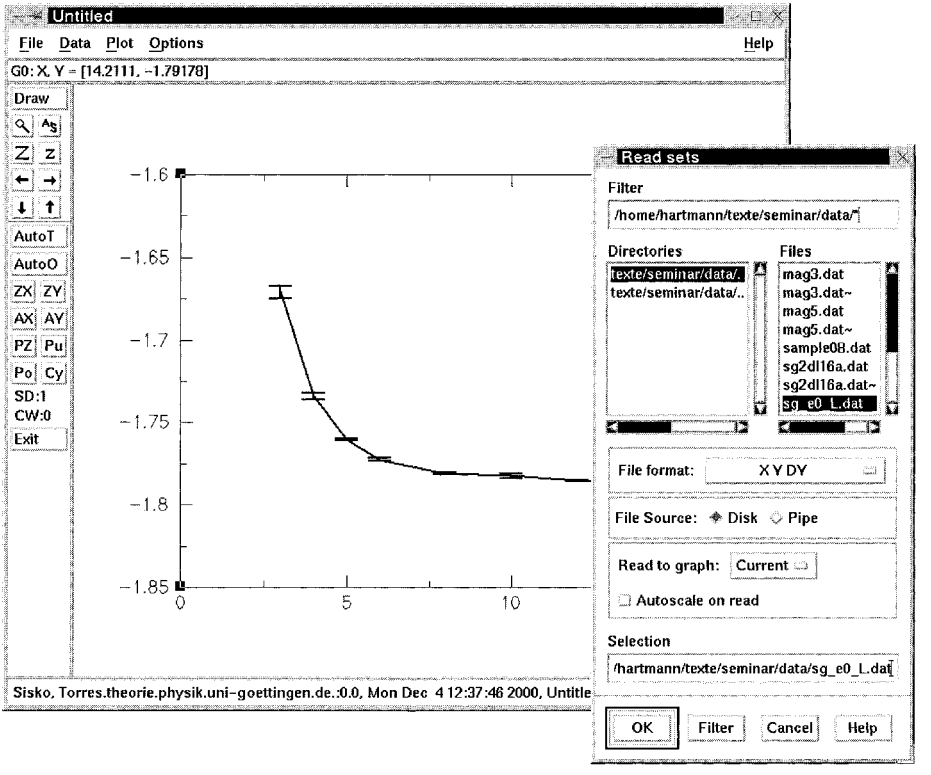


Figure 13.9: The *xmgr* program, just after a data file has been loaded, and the AS button has been pressed to adjust the figure range automatically.

most all arbitrary functions. To issue the command, you have to state the fit function, the data set and the parameters which are to be adjusted. For our example you enter:

```
gnuplot> fit f(x) "sg_e0_L.dat" via e,a,b
```

Then *gnuplot* writes log information to the output describing the fitting process. After the fit has converged it prints for the given example:

```
After 17 iterations the fit converged.
final sum of squares of residuals : 7.55104e-06
rel. change during last iteration : -2.42172e-09

degrees of freedom (ndf) : 5
rms of residuals      (stdfit) = sqrt(WSSR/ndf)      : 0.00122891
variance of residuals (reduced chisquare) = WSSR/ndf : 1.51021e-06
```

```
Final set of parameters            Asymptotic Standard Error
=======================            ==========================

e               = -1.78786         +/- 0.0008548    (0.04781%)
a               = 2.5425           +/- 0.2282       (8.976%)
b               = -2.80103         +/- 0.08265      (2.951%)


correlation matrix of the fit parameters:

                e      a      b
e                1.000
a                0.708  1.000
b               -0.766 -0.991  1.000
```

The most interesting lines are those where the results for your parameters along with the standard error are printed. Additionally, the quality of the fit can be estimated by the information provide in the three lines beginning with "`degree of freedom`". The first of these lines states the number of degrees of freedom, which is just the number of data points minus the number of parameters in the fit. The deviation of the fit function $f(x)$ from the data points $(x_i, y_i \pm \sigma_i)$ $(i = 1, \ldots, N)$ is given by $\chi^2 = \sum_{i=1}^{N} \left[\frac{y_i - f(x_i)}{\sigma_i}\right]^2$, which is denoted by WSSR in the *gnuplot* output. A measure of the quality of the fit is the probability Q that the value of χ^2 is worse than in the current fit, given the assumption that the datapoints y_i are Gaussian distributed with mean $f(x_i)$ and variance one [3]. The larger the value of Q, the better is the quality of the fit. To calculate Q you can use the little program `Q.c`

```
#include <stdio.h>
#include "nr.h"
int main(int argc, char **argv)
{
  float ndf, chi2_per_df;
  sscanf(argv[1], "%f", &ndf);
  sscanf(argv[2], "%f", &chi2_per_df);
  printf("Q=%e\n", gammq(0.5*ndf, 0.5*ndf*chi2_per_df));
  return(0);
}
```

which uses the `gammaq` function from *Numerical Recipes* [3]. The program is called in the form `Q <ndf> <WSSR/ndf>`, which can be taken from the *gnuplot* output.
To watch the result of the fit along with the original data, just enter

```
gnuplot> plot "sg_e0_L.dat" w e, f(x)
```

The result looks like that shown in Fig. 13.10
Please note that the convergence depends on the initial choice of the parameters. The algorithm may be trapped into a local minimum in case the parameters are too far

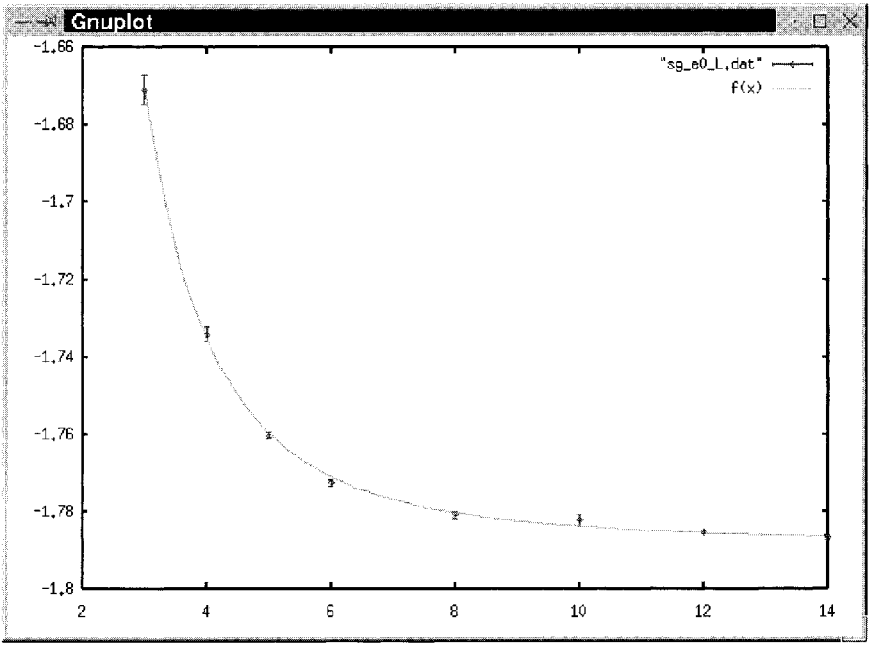


Figure 13.10: *Gnuplot* window showing the result of a fit command along with the input data.

away from the best values. Try the initial values `e=1`, `a=-3` and `b=1`! Furthermore, not all function parameters have to be subjected to the fitting. Alternatively, you can set some parameters to fixed values and omit them from the list at the end of the *fit* command. You should also know that in the example given above all data points enter into the result with the same weight. You can tell the algorithm to consider the error bars by typing `fit f(x) "sg_e0_L.dat" using 1:2:3 via a,b,c`. Then, data points with larger error bars have less influence on the results. More on how to use the `fit` command can be found out when entering `help fit`.

13.8.3 Finite-size Scaling

Statistical physics describes the behavior of systems with many particles. Usually, realistic system sizes cannot be simulated on current computers. To circumvent this problem, the technique of *finite-size scaling* has been invented, for an introduction see e.g. Ref. [21]. The basic idea is to simulate systems of different sizes and extrapolate to the large volume limit. Here it is shown how finite-size scaling can be performed with the help of *gnuplot* [13] or with the special-purpose program *fsscale* [22]
As an example, the average ground-state magnetization m of a three-dimensional $\pm J$ spin glass with fractions p of antiferromagnetic and $1-p$ of ferromagnetic bonds is

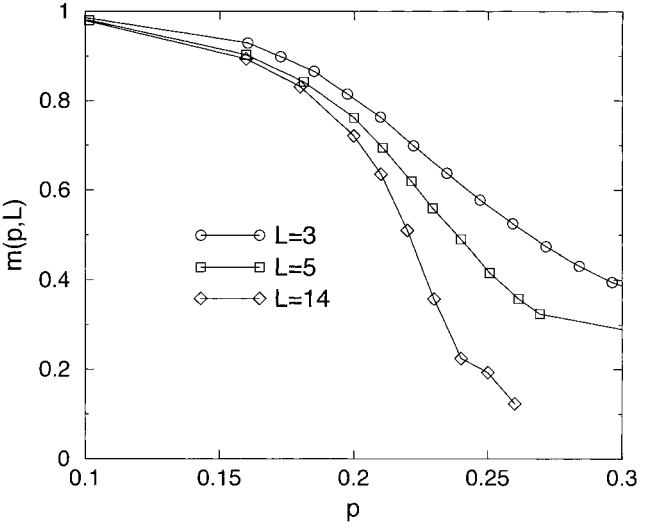


Figure 13.11: Average ground-state magnetization m of a three-dimensional $\pm J$ spin glass with fractions p of antiferromagnetic bonds. Lines are guides to the eyes only.

considered. For small values of p the system is expected to have a ferromagnetically ordered state. This can be observed in Fig. 13.11, where the results [23] for different system sizes $L = 3, 5, 14$ are shown.
The critical concentration p_c, where the magnetization m vanishes, and the critical behavior of m near the transition are to be obtained. From the theory of finite-size scaling, it is known that the average magnetization $m \equiv \langle M \rangle$ obeys the finite-size scaling form [24]

$$m(p, L) = L^{-\beta/\nu} \tilde{m}(L^{1/\nu}(p - p_c)) \tag{13.6}$$

where $\tilde{m}$ is a universal, i.e. non size-dependent, function. The exponent β characterizes the algebraic behavior of the magnetization near p_c, while the exponent ν describes the divergence of the correlation length when p_c is approached. From Eq. (13.6) you can see that when plotting $L^{\beta/\nu} m(p, L)$ against $L^{1/\nu}(p - p_c)$ with correct parameters β, ν the data points for different system sizes should collapse onto a single curve. A good collapse can be obtained by using the values $p_c = 0.222$, $\nu = 1.1$ and $\beta = 0.27$. The determination of p_c and the exponents can be performed via *gnuplot*. For that purpose you need a file `m_scaling.dat` with three columns, where the first column contains the system sizes L, the second the values of p and the third contains magnetization $m(p, L)$ for each data point. First, assume that you know the values for p_c, ν and β. In this case, the actual plot is done by entering:

```
gnuplot> b=0.27
```

```
gnuplot> n=1.1
gnuplot> pc=0.222
gnuplot> plot [-1:1] "m_scale.dat" u (($2-pc)*$1**(1/n)):($3*$1**(b/n))
```

The plot command makes use of the feature that with the u(sing) option you can transform the data of the input in an arbitrary way. For each data set, the variables \$1,\$2 and \$3 refer to the first, second and third columns, e.g. \$1**(1/n) raises the system size to the power $1/\nu$. The resulting plot is shown in Fig. 13.12. Near the transition $p - p_c \approx 0$ a good collapse of the data points can be observed.

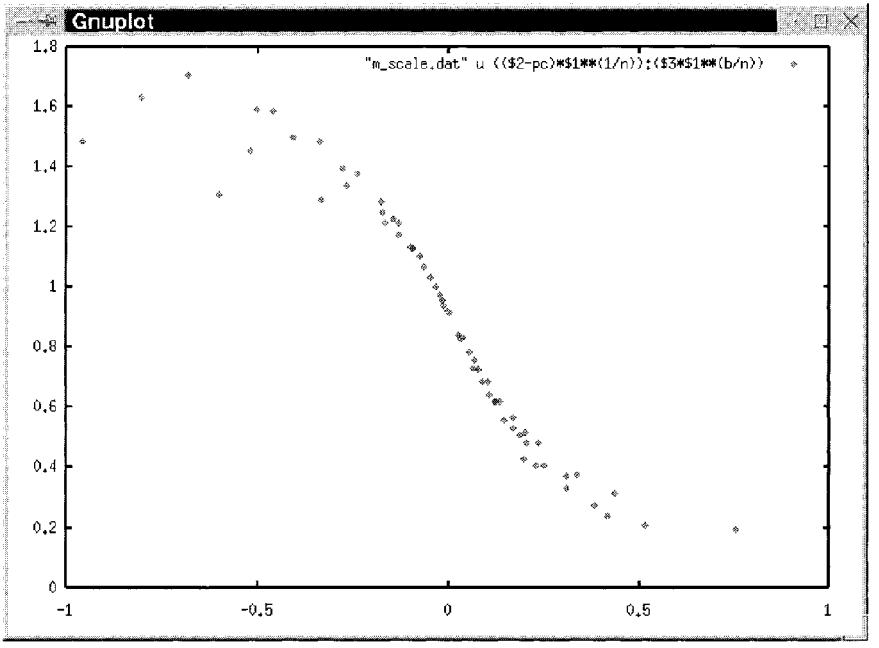


Figure 13.12: *Gnuplot* output of a finite-size scaling plot. The ground-state magnetization of a three-dimensional $\pm J$ spin glass as a function of the concentration p of the antiferromagnetic bonds is shown. For the fit, the parameters $p_c = 0.222, \beta = 0.27$ and $\nu = 1.1$ have been used.

In case you do not know the values of p_c, β, ν you can start with some estimated values, perform the plot, resulting probably in a bad collapse. Then you may alter the parameters iteratively and watch the resulting changes by plotting again. In this way you can converge to a set of parameters, where all data points show a satisfying collapse.

The process of determining the finite-size scaling parameters can be performed more conveniently by using the special purpose program *fsscale*. It can be obtained free of charge from [22]. This tool allows the scaling parameters to be changed interactively by pressing buttons on the keyboard, making a finite-size scaling fit very convenient

to perform. Several different scaling forms are available. To obtain more information, start the program, with `fsscale -help`. A sample screen-shot is shown in Fig. 13.13 Please note that the data have to be presented to *fsscale* in a file containing three columns, where the first column contains the system size, the second the x-value and the third the y-value. If you have only data files with more columns, you can use the standard UNIX tool *awk* to project out the relevant columns. For example, assume that your data file `results.dat` has 10 columns, and your are interested in columns 3, 8, and 9. Then you have to enter:

```
awk '{print $3,$8,$9}' results.dat > projected.dat
```

You can also use *awk* to perform calulations with the values in the columns, similar to *gnuplot*, as in

```
awk '{print $1+$2, 2.0*$7, $8*$1}' results.dat
```

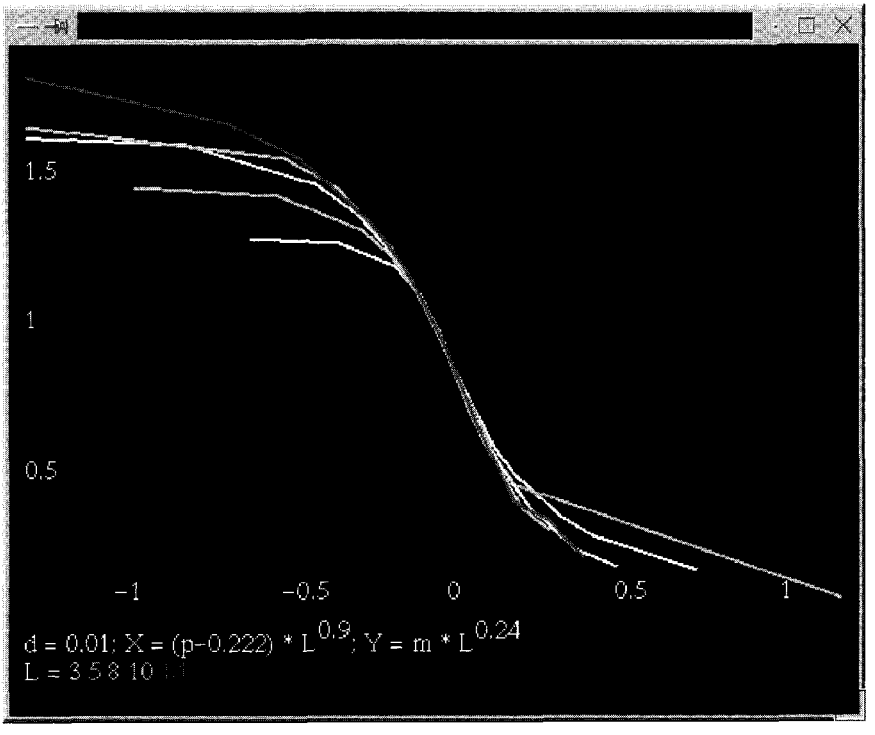


Figure 13.13: Screen-shot from a window running the *fsscale* tool.

13.9 Information Retrieval and Publishing

In this section some basic information regarding searching for literature and preparing your own presentations and publications is given.

13.9.1 Searching for Literature

Before contributing to the physical community and even publishing your results, you should be aware of what exists already. This prevents you from redoing something which has been done before by someone else. Furthermore, knowing previous results and many simulation techniques allows you to conduct your own research projects better. Unfortunately, much information cannot be found in textbooks. Thus, you must start to look at the literature. With modern techniques like CD-ROMs and the Internet this can be achieved very quickly. Within this section, it is assumed that you are familiar with the Internet and are able to use a browser. In the following list several sources of information are contained.

- **Your local (university) library**
  Although the amount of literature is limited from space constraints, you should always check your local library for suitable textbooks concerning your area of research. Also many old issues of scientific journals are yet not available through the Internet, thus you may have to copy some articles in the library.

- **Literature databases**
  In case you want to obtain all articles from a specific author or all articles on a certain subject, you should consult a literature database. In physics the *INSPEC* [25] database is the appropriate source of information. Unfortunately, the access is not free of charge. But usually your library should allow access to INSPEC, either via CD-ROMS or via the Internet. If your library/university does not offer an access you should complain.

  INSPEC frequently surveys almost all scientific journals in the areas of physics, electronics and computers. For each paper that appears, all bibliographic information along with the abstract are stored. You can search the database for example for author names, keywords (in the abstract or title), publication years or journals. Via INSPEC it is possible to keep track of recent developments happening in a certain field.

  There are many other specialized databases. You should consult the web page of your library, to find out to which of them you can access. Modern scientific work is not possible without regularly checking literature databases.

- **Preprint server**
  In the time of the Internet, speed of publication becomes increasingly important. Meanwhile, many researchers put their publications on the *Los Alamos Preprint server* [26], where they become available world wide at most 72 (usually 24) hours after submission. The database is free of charge and can be accessed from

almost everywhere via a browser. The preprint database is divided into several sections such as astrophysics (astro-ph), condensed matter (cond-mat) or quantum physics (quant-ph). Similar to a conventional literature database, you can search the database, eventually restricted to a section, for author names, publication years or keywords in the title/abstract. But furthermore, after you have found an interesting article, you can download it and print it immediately. File formats are *postscript* and *pdf*. The submission can also be in TEX/LATEX (see Sec. 13.9.2).

Please note that there is no editorial processing at all, that means you do not have any guarantee on the quality of a paper. If you like, you can submit a poem describing the beauty of your garden. Nevertheless, the aim of the server is to make important scientific results available very quickly. Thus, before submitting an article, you should be sure that it is correct and interesting, otherwise you might get a poor reputation.

The preprint server also offers access via email. It is possible to subscribe to a certain subject. Then every working day you will receive a list of all new papers which have been submitted. This is a very convenient way of keeping track of recent developments. But be careful, not everyone submits to the preprint server. Hence, you still have to read scientific journals regularly.

- **Scientific journals**
  Journals are the most important resources of information in science. Most of them allow access via the Internet, when your university or institute has subscribed to them. Some of the most important physical journals, which are available online, are published by (in alphabetical order)

  - the American Institute of Physics [27]
  - the American Physical Society [28]
  - Elsevier Science (Netherlands) [29]
  - the European Physical Society [30]
  - the Institute of Physics (Great Britain) [31]
  - Springer Science (Germany) [32]
  - Wiley-VCH (USA/Germany) [33]
  - World-Scientific (Singapore) [34]

- **Citation databases**
  In every scientific paper some other articles are cited. Sometimes it is interesting to get the reverse information, i.e. to obtain all papers which are citing a given article A. This can be useful, if one wants to learn about the most recent developments which are triggered by article A. In that case you have to access a *citation index*. For physics, probably the most important is the *Science Citation Index* (SCI) which can be accessed via the *Web of Science* [35]. You have to ask your system administrator or your librarian whether and how you can access it from your site.

The *American Physical Society* (APS) [28] also includes links to citing articles with the online versions of recent papers. If the citing article is available via the APS as well, you can immediately access the article from the Internet. This works not only for citing papers, but also for cited articles.

- **Phys Net**
  If you want to have access to the web pages of a certain physics department, you should go via your web browser to the *Phys Net* pages [36]. They offer a list of all physics departments in the world. Additionally, you will find lists of forthcoming conferences, job offers and many other useful links. Also, the home page of your department probably offers many interesting links to other web pages related to physics.

- **Web browsing**
  Except for the sources mentioned so far, nowadays much information is available on net. Many researchers present their work, their results and their publications on their home pages. Quite often talks or computer codes can also be downloaded.

  In case you cannot find a specific page through the *Phys Net* (see above), or you are interested in obtaining all web pages concerning a specific subject, you should ask a *search engine*. There are some very popular all purpose engines like *Yahoo* [37] or *Alta Vista* [38]. A very convenient way to start a query on several search engines in parallel is a *meta search engine*, e.g. *Metacrawler* [39]. To find out more, please contact a search engine.

13.9.2 Preparing Publications

In this section tools for two types of presenting your results are covered: via an article/report or in a talk. For writing papers, it is recommended that you use *TEX/LATEX*. Data plots can be produced using the programs explained in the last section. For drawing figures and making transparencies, the program *xfig* offers a large functionality. To create three-dimensional perspective images, the program *Povray* can be used. LaTeX, *xfig* and *Povray* are introduced in this section.
First, TeX/LaTeX is explained. It is a typesetting system rather than a word processor. The basic program is TeX, LaTeX is an extension to facilitate the application. In the area of theoretical computer science, the combination of TeX and LaTeX is a widespread standard. When submitting an article electronically to a scientific journal usually LaTeX has to be used. Unlike the conventional office packages, with LaTeX you do not see the text in the form it will be printed, i.e. LaTeX is not a WYSIWYG ("What you see is what you get") program. The text is entered in a conventional text editor (like *Emacs*) and all formatting is done via special commands. An introduction to the LaTeX language can be found e.g. in Refs. [40, 41]. Although you have to learn several commands, the use of LaTeX has several advantages:

- The quality of the typesetting is excellent. It is much better than self-made formats. You do not have to care about the layout. But still, you are free to change everything according to your requirements.

- Large projects do not give rise to any problems, in contrast to many commercial office programs. When treating a LaTeX text, your computer will never complain when your text is more than 300 pages or contains many huge post-script figures.
- Type setting of formulae is very convenient and fast. You do not have to care about sizes of indices of indices etc. Furthermore, in case you want for example to replace all α in your formulae with β, this can be done with a conventional replace, by replacing all `\alpha` strings by a `\beta` strings. For the case of an office system, please do not ask how to do this conveniently.
- There are many additional packages for enhanced styles such as letters, transparencies or books. The *bibtex* package is very convenient, which allows a nice literature database to be build up.
- Since you can use a conventional editor, the writing process is very fast. You do not have to wait for a huge packet to come up.
- On the other hand, if you still prefer a WYSIWYG ("what you see is what you get") system, there is a program called *lyx* [42] which operates like a conventional word processor but creates LaTeX files as output. Nevertheless, once you get used to LaTeX, you will never want to loose it.

Please note that this text was written entirely with LaTeX. Since LaTeX is a type setting language, you have to compile your text to create the actual output. Now, an example is given of what a LaTeX text looks like and how it can be compiled. This example will just give you an impression of how the system operates. For a complete reference, please consult the literature mentioned above.
The following file `example.tex` produces a text with different fonts and a formula:

```
\documentclass[12pt]{article}
\begin{document}
This is just a small sample text. You can write some words {\em
emphasized}\/, or in {\bf bold face}. Also different {\small sizes}
are possible.

An empty line generates a new paragraph. \LaTeX\  is very convenient
for writing formulae, e.g.
\begin{equation}
M_i(t) = \frac{1}{L^3} \int_V x_i \rho(\vec{x},t) d^3\vec{x}
\end{equation}
\end{document}
```

The first line introduces the type of the text (`article`, which is the standard) and the font size. You should note that all tex commands begin with a backslash (\), in case you want to write a backslash in your text, you have to enter `$\backslash$`. The actual text is written between the lines starting with `\begin{document}` and ending with `\end{document}`. You can observe some commands such as `\em`, `\bf` or `\small`. The { } braces are used to mark blocks of text. Mathematical formulae can be written

e.g. with `\begin{equation}` and `\end{equation}`. For the mathematical mode a huge number of commands exists. Here only examples for Greek letters (`\alpha`), subscripts (`x_i`), fractions (`\frac`), integrals (`\int`) and vectors (`\vec`) are given.
The text can be compiled by entering `latex example.tex`. This is the command for UNIX, but LaTeX exists for all operating systems. Please consult the documentation of your local installation.
The output of the compiling process is the file `example.dvi`, where "dvi" means "device independent". The `.dvi` file can be inspected on screen by a viewer via entering `xdvi example.dvi` or converted into a postscript file via typing `dvips -o example.ps example.dvi` and then transferred to a printer. On many systems it can be printed directly as well. The result will look like this:

> This is just a small sample text. You can write some words *emphasized*, or in **bold face**. Also different sizes are possible.
>
> An empty line generates a new paragraph. LaTeX is very convenient for writing formulae, e.g.

$$M_i(t) = \frac{1}{L^3} \int_V x_i \rho(\vec{x}, t) d^3\vec{x} \tag{13.7}$$

This example should be sufficient to give you an impression of what the philosophy of LaTeX is. Comprehensive instructions are beyond the scope of this section, please consult the literature [40, 41].
Under UNIX/Linux, the spell checker *ispell* is available. It allows a simple spell check to be performed. The tool is built on a dictionary, i.e. a huge list of known words. The program scans any given text, also a special LaTeX mode is available. Every time a word occurs, which is not contained in the list, *ispell* stops. Should similar words exist in the list, they are suggested. Now the user has to decide whether the word should be replaced, changed, accepted or even added to the dictionary. The whole text is treated in this way. Please note that many mistakes cannot be found in this way, especially when the misspelled word is equal to another word in the dictionary. However, at least *ispell* finds many spelling mistakes quickly and conveniently, so you should use the tool.
Most scientific texts do not only contain text, formulae and curves, but also schematic figures showing the models, algorithms or devices covered in the publication. A very convenient but also simple tool to create such figures is *xfig*. It is a window based vector-oriented drawing program. Among its features are the creation of simple objects like lines, arrows, polylines, splines, arcs as well as rectangles, circles and other closed, possibly filled, areas. Furthermore you can create text or include arbitrary (eps, jpg) pictures files. You may place the objects on different layers which allows complex sceneries to be created. Different simple objects can be combined into more complex objects. For editing you can move, copy, delete, rotate or scale objects. To give you an impression what *xfig* looks like, in Fig. 13.14 a screen-shot is shown, displaying *xfig* with the picture that is shown in Fig. 13.1. Again, for further help, please consult the online help function or the *man* pages.

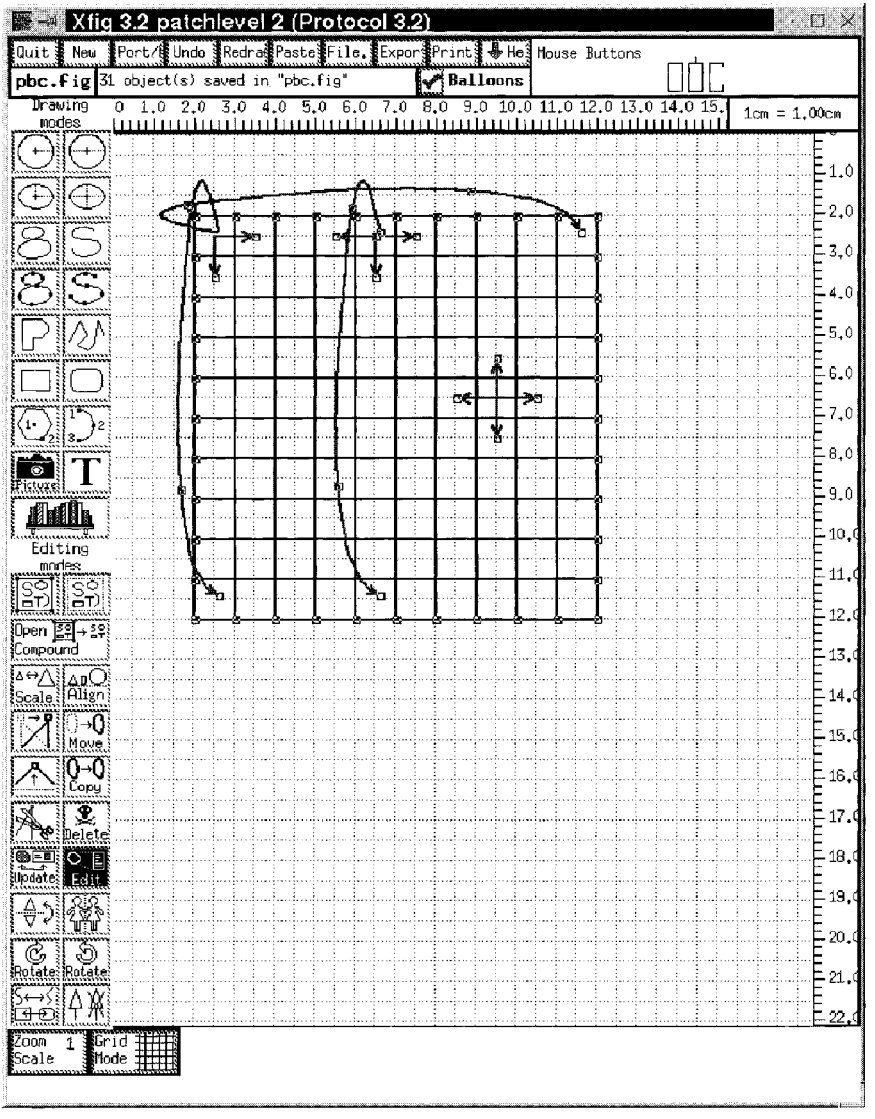


Figure 13.14: A sample screen-shot showing the *xfig* program.

The figures can be saved in the internal fig format, and exported in several file formats such as (encapsulated) *postscript*, LaTeX, *Jpeg*, *Tiff* or bitmap. The *xfig* program can be called in a way that it produces just an output file with a given fig input file. This is very convenient when you have larger projects where some small picture objects are contained in other pictures and you want to change the appearance of the small

objects in all other files. With the help of the *make* program pretty large projects can be realized.
Also, *xfig* is very convenient when creating transparencies for talks, which is the standard method of presenting results in physics. With different colors, text sizes and all the objects mentioned before, very clear transparencies can be created quickly. The possibility of including picture files, like *postscript* files which were created by a data plotting program such as *xmgr*, is very helpful. In the beginning it may seem that more effort is necessary than when creating the transparencies by hand. However, once you have a solid base of transparencies you can reuse many parts and preparing a talk may become a question of minutes. In particular, when your handwriting looks awful, the audience will be much obliged for transparencies prepared with *xfig*.
Last but not least, please note that *xfig* is vector oriented, but not pixel oriented. Therefore, you cannot treat pictures like jpg files (e.g. photos) and apply operations like smoothing, sharpening or filtering. For these purposes the package *gimp* is suitable. It is freely available again from GNU [5].
It is also possible to draw three-dimensional figures with *xfig*, but there is no special support for it. This means, *xfig* has only a two-dimensional coordinate system. A very convenient and powerful tool for making three-dimensional figures is *Povray* (Persistence Of Vision RAYtraycer). Here, again, only a short example is given, for a detailed documentation please refer to the home page [43], where the program can be downloaded for many operating systems free of charge.
Povray is, as can be realized from its name, a *raytracer*. This means you present a scene consisting of several objects to the program. These objects have characteristics like color, reflectivity or transparency. Furthermore the position of one or several light sources and a virtual camera have to be defined. The output of a raytracer is a photo-realistic picture of the scene, seen through the camera. The name "raytracer" originates from the fact that the program creates a picture by starting several rays of light at the light sources and traces their way through the scene, where they may be absorbed, reflected or refracted, until they hit the camera, disappear into infinity or become too weak. Hence, the creation of a picture may take a while, depending on the complexity of the scene.
A scene is described in a human readable file, it can be entered with any text editor. But for more complex scenes, special editors exist, which allow a scene to be created interactively. Also several tools for making animations are available on the Internet. Here, a simple example is given. The scene consists of three spheres connected by two cylinders, forming a molecule. Furthermore, a light source, a camera, an infinite plane and the background color are defined. Please note that a sphere is defined by its center and a radius and a cylinder by two end points and a radius. Additionally, for all objects color information has to be included, the center sphere is slightly transparent. The scene description file `test1.pov` reads as follows:

```
#include "colors.inc"

background { color White }
```

```
sphere {  <10, 2, 0>, 2
    pigment { Blue } }

cylinder { <10, 2, 0>,  <0, 2, 10>, 0.7
    pigment { color Red } }

sphere {  <0, 2, 10>, 4
    pigment { Green transmit 0.4} }

cylinder { <0, 2, 10>,  <-10, 2, 0>, 0.7
    pigment { Red } }

sphere {  <-10, 2, 0>, 2
    pigment { Blue } }

plane { <0, 1, 0>, -5
    pigment { checker color White, color Black}}

light_source { <10, 30, -3> color White}

camera {location <0, 8, -20>
    look_at  <0, 2,  10>
    aperture 0.4}
```

The creation of the picture is started by calling (here on a Linux system via command line) `x-povray +I test1.pov`. The resulting picture is shown in Fig. 13.15, please note the shadows on the plane.

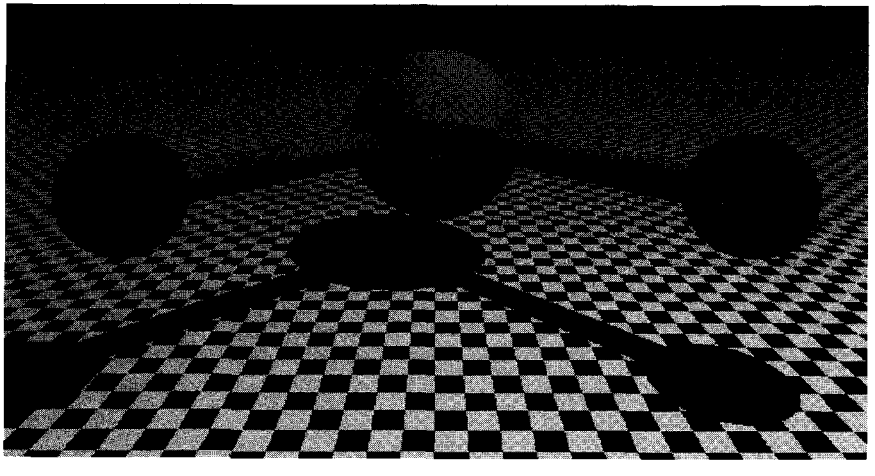


Figure 13.15: A sample scene created with *Povray*.

Povray is really powerful. You can create almost arbitrarily shaped objects, combine them into complex objects and impose many transformations. Also special effects like blurring or fog are available. All features of *Povray* are described in a 400 page manual. The use of *Povray* is widespread in the artists community. For scientists it is very convenient as well, because you can easily convert e.g. configuration files of molecules or three-dimensional domains of magnetic systems into nice looking perspective pictures. This can be accomplished by writing a small program which reads e.g your configuration file containing a list of positions of atoms and a list of links, and puts for every atom a sphere and for every link a cylinder into a *Povray* scene file. Finally the program must add suitable chosen light sources and a camera. Then, a three-dimensional pictures is created by calling *Povray*.

The tools described in this section, should allow all technical problems occurring in the process of preparing a publication (a "paper") to be solved. Once you have prepared it, you should give it to at least one other person, who should read it carefully. Probably he/she will find some errors or indicate passages which might be difficult to understand or that are misleading. You should always take such comments very seriously, because the average reader knows much less about your problem than you do.

After all necessary changes have been performed, and you and other readers are satisfied with the publication, you can submit it to a scientific journal. You should choose a journal which suits your paper. Where to submit, you should discuss with experienced researchers. It is not possible to give general advice on this issue. Nevertheless, technically the submission can be performed nowadays almost everywhere electronically. For a list of publishers of some important journals in physics, please see Sec. 13.9.1. Submitting one paper to several journals in parallel is not allowed. However, you should also consider submitting to the preprint server [25] as well to make your results quickly available to the physics community.

Nevertheless, although this text provides many useful hints concerning performing computer simulations, the main part of the work is still having good ideas and carefully conducting the actual research projects.

Bibliography

[1] I. Sommerville, *Software Engineering*, (Addisin-Wesley, Reading (MA) 1989)

[2] C. Ghezzi, M. Jazayeri, and D. Mandrioli, *Fundamentals of Software Engineering*, (Prentice Hall, London 1991)

[3] W.H. Press, S.A. Teukolsky, W.T. Vetterling, and B.P. Flannery, *Numerical Recipes in C* (Cambridge University Press, Cambridge 1995)

[4] K. Mehlhorn and St. Näher, *The LEDA Platform of Combinatorial and Geometric Computing* (Cambridge University Press, Cambridge 1999);
see also http://www.mpi-sb.mpg.de/LEDA/leda.html

[5] M. Loukides and A. Oram, *Programming with GNU Software*, (O'Reilly, London 1996);
see also http://www.gnu.org/manual

[6] H.R. Lewis and C.H. Papadimitriou, *Elements of the Theory of Computation*, (Prentice Hall, London 1981)

[7] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen, *Object-Oriented Modeling and Design*, (Prentice Hall, London 1991)

[8] R. Johnsonbaugh and M. Kalin, *Object Oriented Programming in C++*, (Macmillan, London 1994)

[9] J. Skansholm, *C++ from the Beginning*, (Addisin-Wesley, Reading (MA) 1997)

[10] Mail to hartmann@theorie.physik.uni-goettingen.de

[11] B.W. Kernighan and D.M. Ritchie, *The C Programming Language*, (Prentice Hall, London 1988)

[12] A. Oram and S. Talbott, *Managing Projects With Make*, (O'Reilly, London 1991)

[13] The programs and manuals can be found on http://www.gnu.org. For some there is a *texinfo file*. To read it, call the editor 'emacs' and type <crtl>+'h' and then 'i' to start the texinfo mode.

[14] J. Phillips, *The Nag Library: A Beginner's Guide* (Oxford University Press, Oxford 1987);
see also http://www.nag.com

[15] A. Heck, *Introduction to Maple*, (Springer-Verlag, New York 1996)

[16] B.J.T. Morgan, *Elements of Simulation*, (Cambridge University Press, Cambridge 1984)

[17] A.M. Ferrenberg, D.P. Landau and Y.J. Wong, *Phys. Rev. Lett.* **69**, 3382 (1992); I. Vattulainen, T. Ala-Nissila and K. Kankaala, *Phys. Rev. Lett.* **73**, 2513 (1994)

[18] J.F. Fernandez and C. Criado, *Phys. Rev. E* **60**, 3361 (1999)

[19] http://www.cs.colorado.edu/homes/zorn/public_html/MallocDebug.html

[20] The tool can be obtained under the gnu public license from http://www.gnu.org/software/checker/checker.html

[21] J. Cardy, *Scaling and Renormalization in Statistical Physics*, (Cambridge University Press, Cambridge 1996)

[22] The program *fsscale* is written by A. Hucht, please contact him via email: fred@thp.Uni-Duisburg.DE

[23] A.K. Hartmann, *Phys. Rev. B* **59** , 3617 (1999)

[24] K. Binder and D.W. Heermann, *Monte Carlo Simulations in Statistical Physics*, (Springer, Heidelberg 1988)

[25] http://www.inspec.org/publish/inspec/

[26] http://xxx.lanl.gov/

[27] http://www.aip.org/ojs/service.html

[28] http://publish.aps.org/

[29] http://www.elsevier.nl

[30] http://www.eps.org/publications.html

[31] http://www.iop.org/Journals/

[32] http://www.springer.de/

[33] http://www.wiley-vch.de/journals/index.html

[34] http://ejournals.wspc.com.sg/journals.html

[35] http://wos.isiglobalnet.com/

[36] http://physnet.uni-oldenburg.de/PhysNet/physnet.html

[37] http://www.yahoo.com/

[38] http://www.altavista.com/

[39] http://www.metacrawler.com/index.html

[40] L. Lamport and D. Bibby, *LaTeX : A Documentation Preparation System User's Guide and Reference Manual*, (Addison Wesley, Reading (MA) 1994)

[41] http://www.tug.org/

[42] http://www.lyx.org/

[43] http://www.povray.org/

Index

$\leq_p$ 23
1-SAT algorithm 21
2-CNF 20
3-CNF 20
3-SAT 22, 23

adjacency list 46
adjacency matrix 46
adjacent node 37
aging 187, 190
algorithm 9
  1-SAT 21
  approximation for 2d spin glasses 250
  bit-sequence 165
  branch-and-bound 194, 282–287
  branch-and-cut 194
  breadth-first search *59–61*, 104, 108, 115, 231–234
  build-cluster 196
  burning 54, 57
  certificate checking 22
  cluster 77
  conjugate gradient 170
  cycle canceling 142
  decision 20
  depth-first search 57–59
  Dijkstra's 63–67, 70
  Dinic's *107–115*
  divide-and-conquer 281–282
  ellipsoid 236
  extend-alternating-tree 233
  factorial 27
  fibonachi 32
  find 48
  for spin glasses 192–202, 208–219
  Ford-Fulkerson 102–107
  forest-fire 54, 57
  generalized Rosenbluth 263
  genetic *159–183*, 195, 199–202
  graph 48–49, 53–70
  greedy cover 278
  Hoshen-Kopelman 54–57
  Hungarian 235, *237–241*
  invasion 54, 70
  Kruskal's 69, *70*
  label-setting 63
  Leath 54, 57
  local optimization 170
  matching 194, *231–250*
  maximum-flow 143
  mergesort 30
  minimize-function 167
  minimum spanning tree 68–70
  minimum-cost-flow 139–147
  minimum-weight perfect matching 242–250
  Monte Carlo 77, 255–270
  negative-cycle-canceling 68, *141–143*
  N-line 135
  nondeterministic 23
  N queens 33
  optimization 5
  parallel tempering 260–262
  Prim's 69, *70*
  primal-dual 239
  prune-enriched Rosenbluth 262–270
  push-relabel 107
  quantum Monte Carlo 266
  random-walk 255
  sequential 27
  shortest path 61–68
  simplex 236

simulated annealing 194, *257–259*
sorting 29
steepest descent 170
successive-shortest-path *135*, 139
threshold accepting 256
wave *107–115*, 121
all ground states 115–121
all minimum cuts 115–121
α 85
Alta Vista 349
alternating path *229*, 230
alternating tree *231*
American Physical Society 349
amino acid 266
amorphous conductor 53
ancestor 39
AND operation 19
antiferromagnet 185, 250
antiferromagnetic interactions 4, 91
antiperiodic boundary conditions 205
applications of optimization methods 1–2
approximation algorithm for 2d spin glasses 250
APS 349
arbitrary precision 321
arc 37
array 295, 321, 334
array implementation of heaps 45
assignment 10, 19, 20
problem 235
astronomy 179
asymptotic behavior 17
augmenting path *103*, 105, 107, 230, 232–234, 242
theorem 231
autocorrelation function 78
awk 346

backbone *275*, 290
backtracking *32*, 242
backward wave 110
balance constraint 151
balanced tree 49
balanced vertex 109
ballistic search 208–219
base 242
basic graph algorithms 48–49
basic operations 296
behavior
asymptotic 17
β 54, 80, 81, 84, 123
BFS *59*, 104, 115, 231–234
biconnected component 59
bimodal distribution 88, *92*, 186
binary distribution 88
binary representation 161, 164
binary tree *40*, 279
Binder cumulant 85
Biot-Savat law 138
bipartite graph *229*, 231–241
bit-sequence procedure 165
bitmap 352
blocked vertex 110
blocking flow 107
blossom 242
shrinking 242, 245
Boltzmann
constant 74
distribution 74
entropy 75
bond 37, 61, 186
boolean
assignment 19, 20
formula 19
operation 19
variable 19
bottom up approach 295
bound 282
lower 276
upper 17, 276
boundary conditions 251
antiperiodic 205
periodic 205
branch-and-bound algorithm 194, 282–287
branch-and-cut algorithm 194
breadth-first search *59–61*, 104, 108, 115, 231–234
BS *212*
bucket 67, 70
Burgers charge 151
burning algorithm 54, 57

C programming language 293, 300
canonical ensemble 74
capacity 96, 102
constraint 43, 129, 140
of cut 97

capsuling 301
cases statement 10
CC 22
$CdCr_{1.7}In_{0.3}S_4$ 187
CEA *195–197*, 207, 209, 219
center of mass 179
central limit theorem 330
certificate checking algorithm 22
change-y procedure 247
chaos 252
checkerboard fashion 94, 96
checkergcc 334–338
$\chi(p)$ 81
Church's thesis 13
citation database 348
class 295, 297, *300*
    `histo` 303
    NP 22
    P 22
clause 19
clean 316
closed path 37
cluster 55, 81, 195
    algorithm 77
    mass 83
    percolating 82
    size 54
cluster-exact approximation *192–202*, 207, 209, 219
CNF 19
columnar defect 147
columnar disorder 137
combinatorial optimization problem 2
comment 12, 299, *308–310*, 318
complete tree 41
component
    biconnected 59
    connected 37, 57
    strongly connected 116
    strongly-connected *38*
computable function 15
condition 10
conductor 53
configuration file 294
conjugate gradient algorithm 170
conjunction 19
conjunctive normal form 19
connected component 37, 57
connected graph 37
connectivity percolation 53–61
conservation
    of energy 298
    of flow 43, 103
    of momentum 298
`const` 298
constraint 2
    capacity 43
constructor 305
contact matrix 270
continuous transition 78
convex function 136, 140
convex minimum-cost-flow problem 136–139
cooling 257
    protocol 259
correctness problem 16
correlation
    function 252
    length 54, 82, 84, 252
    time 78
correlation1function 84
costs 53, 63
    negative 132
    reduced 64, 133, 135, 143, 244
    residual 140
cover 50
    edge 50
    minimum edge 50
    vertex 51
critical
    behavior 80
    exponent 54, 80, 81
    point 78
crossover *161*
    operation 166, 173, 182
    triadic 199
crystal 257
curve fitting 340–343
cut
    capacity of 97
    minimum *99*, 101, 115
    odd 243
    optimality condition 69
    s-t 96
cutoff 175
cycle
    canceling algorithm 142
    Euler 50

Hamilton 50
negative 53, 68, 132

δ 81, 84
DAFF 91, *92–96*, 121–125, 192
data
analysis 338–346
capsuling 301
collapse 85, 123
structures 295, 301
data display debugger (*ddd*) 334
debugging 297, 332
tools 331–338
decidable problem 16
decision algorithm 20
deficit 143
node 145
`#define` 310
degeneracy 4, 91, 116, 122, 208, 228, 269
degree 37
of freedom 342
delta-peak 221
Δ_c 123
density 78
distribution 179
dependency 315
depth-first search 57–59
depth-first spanning tree 59
derived graph 245
descendant 39
destructor 305
detailed balance 76
d_f 54, 81
diagonalization technique 13–16
dictionary 321
differential equation 319
Dijkstra's algorithm 63–67, 70
diluted antiferromagnet 192
diluted antiferromagnet in a field 91, *92–96*, 121–125
diluted lattice 53
dimensionless ratio 85
dimer covering 229
Dinic's algorithm *107–115*
time complexity 107
dipolar interactions 189
directed graph *37*, 47, 96, 102
directed polymer 51, 61–63, 129
discontinuous transition 78
discrete optimization problem 2
disjunction 19
dislocation pair 151
disorder 187
columnar 137
frozen 3
quenched 3, 324
unbounded 150
disordered system 87–90
distance label 68
distributing work 296
distribution
bimodal 88, *92*, 186
binary 88
density 179
exponential 328
Gaussian 92, 186, 222, *330–331*
joint 89
normal 330
of overlaps 125, 191, 192, 219
divergence 133, 139
divide-and-conquer algorithm 281–282
divide-and-conquer principle 30, 31
documentation 299
domain 124
domain state 95
domain wall 154, 205
renormalization group analysis 154
Doppler shift 178
`-D` option 311
double linked list 45
DPRM 62
Droplet picture 191, 195, 219
DS 95
dual linear problem 236
duality 236
DWRG 154
dynamic programming 31
dynamic susceptibility 187

EA model 186
EC problem 50
edge *37*, 228
cover 50
free 228
head 38
incoming 38
inner 96
matched 228

outgoing 38
phantom 101
singly connected 59
tail 38
Edmonds theorem 242
Edwards-Anderson model 186
eigenvalue 320
elastic medium 96
electron 173
element of list 41
ellipsoid algorithm 236
energy 75
conservation 298
free 74
internal 74
enrichment 264
ensemble
canonical 74
microcanonical 74
entropy 206
Boltzmann 75
ground-state 192
enumeration problem 229
ergodic *209*
error bar 78, 299, 342
η 84
Euler cycle 50
even level 233
excess *109*, 143
excess node 145
excitation 154–155
excited state 221
expand-update procedure 247
expectation value 74
experimental results for spin glasses 187–190
exponent
α 85
β 54, 80, 81, 84, 123
critical 54
η 84
γ 81, 84
ν 123, 252
σ 81
stiffness 207
τ 81
Θ_S 207, 251
exponential distribution 328
external field 74
factorial 27
father 39
FC 95
feasible
flow 141
solution 2
vector 235
Fe_xAu_{1-x} 187, 188
FeF_2 92–93
$Fe_xZn_{1-x}F_2$ 92–93
$Fe_{1-x}Mn_xTiO_3$ 190
ferromagnet 3, 185, 204, 250
ferromagnetic interactions 4, 91
ferromagnetic phase 84
FH 95
Fibonachi numbers 30–32
field
cooling 95
external 74
heating 95
FIFO implementation 68
FIFO list 41
find operation 48
finite graph 37
finite-size scaling 81, 83–84, 203, 289, 343–346
first order transition 79
fit parameters 85
fitness *161*
function 180
fitting function 173
fitting 340–343
quality of 342
flow 42, 102, 105, 133
blocking 107
conservation 43, 103
feasible 141
infeasible 143
maximum 43, 106, 192
minimum-cost 43, 68
negative 43
pseudo 143
flow-augmentation lemma 145
fluctuations 75
flux 155
flux line 138
array 147–150
problem 147–150

for loop 10
Ford-Fulkerson algorithm 102–107
  time complexity 105
forest-fire algorithm 54, 57
formula
  boolean 19
  satisfiable 19
Fortran 293, 300
forward wave 110
four-dimensional spin glass 202
Fourier transform 319
fractal 54, 95
  dimension 81
free energy 74, 78
free spin 192, 210, 218
free vertex *228*, 286
frozen disorder 3
frustration *187*, 194, 227
fsscale 345–346
function
  computable 15
  convex 136
  partial 13

GA 159
galaxy 178
γ 54, 81, 84
gas 78
gauge transformation 95, 193
Gaussian distribution *92*, 186, *330–331*
$GdAl_2$ 189
gdb 332–334
GEATbx 164
generalized Rosenbluth method 263
genes 159
genetic algorithm *159–183*, 195, 199–202
  applications 163
  data structures 161
  example 164–171
  general scheme 163
  representation 160, 161, 164, 173, 179
Genetic and Evolutionary Algorithm Toolbox 164
geological structures 163
glass 87
global variable 307
GNU project 297
gnuplot 204, *338–340*
  finite-size scaling 344
  script 338
"go with the winners" strategy 266
gold 187
goto statement 11, 307
graph *37–38*, 96, 228, 295, 321
  algorithms 53–70
  algorithms in LEDA 321
  bipartite *229*, 231 241
  connected 37
  cover 50
  derived 245
  directed *37*, 47, 102
  finite 37
  implementation 46–48
  labeled *42*, 48
  network 42
  planar 227
  random 275
  reduced 117, 118
  representation 44–48
  residual 103
  sparse 46, 301
gravitational force 180
greedy cover algorithm 278
Green's function 139
ground state 4, 62, 89, 91, 94, 102, 116, 129, 139, 151, 152, 154, 160, 172, 192, 199, 202, 219, 220, 228, 250–252, 266
  degenerate 4
  energy 203, 219, 338
  landscape 192, 207, 219
  number of 124
ground states
  all 115–121
growth front 65, 70

halting problem 16
Hamilton cycle 50
Hamiltonian 62, 74, 80, 87, 92, 94, 98, 129, 133, 136, 138, 139, 186, 193, 196, 255
harmonic potential 175
h_c 89
HC problem 50
head of edge 38
head of list 41
header file 306, 311

heap 41, 66
  implementation 45
heap Dijkstra algorithm 67
heat bath 74
height of tree 39
`-help` option 299
heuristic 198
  traveling salesman problem *12–13*, 17–18
  vertex cover 277
hierarchy of calls 27, 29–31
histogram 303
Hoshen-Kopelman algorithm 54–57
Householder transformation 321
Hungarian algorithm 235, *237–241*
  time complexity 239
hyper scaling relation 82

`#ifdef` 311
`#ifndef` 311
implementation 297
  graphs 46–48
  heaps 45
  lists 44–45
  trees 45–46
importance sampling 76
incoming edge 38
individual 199
infeasible flow 143
inner edge 96
inner vertex 96
input 294
  size of 16
insert operation 41, 49
INSPEC 163, 347
integration 319
interaction
  antiferromagnetic 4, 91
  ferromagnetic 4, 91
  superexchange 92
interface 301
internal energy 74
interpolation 319
invasion algorithm 54, 70
inversion method 327–328
irreversibility 95
Ising model 73, 79, 84, 87, 92, 93, 185, 204
isolated node 37

ispell 351

joint distribution 89
Jpeg 352

Kruskal's algorithm 69, *70*
k-SAT *19–22*, 23, 273

label-correcting algorithm 67–68
label-setting algorithm 63
labeled graph *42*, 48
labeling 42
Lambert-W-function 276
laser 163
  pulses 175
LaTeX 349–351
lattice 81, 87
  diluted 53
  gauge theory 163
  square 312–314
leaf 39
Leath algorith 57
Leath algorithm 54
LEDA library 26, 47, 64, 294, 295, *321–323*
left son 40
left subtree 40
length of path 37
level
  even 233
  network 107
  odd 233
library 294, 295, *319–324*, 347
  create 323–324
  LEDA 26, 47, 64
LIFO list 41
linear congruential generators 325
linear equation 319
linear fitness ranking 182
linear program 235
Linux 293, 297, 310
liquid 78
list *41–42*, 295, 321
  adjacency 46
  double linked 45
  element 41
  FIFO 41
  head 41
  implementation 44–45

LIFO 41
operations 321
pointer implementation 44
tail 41
literal 19
literature databases 347
local field 195
local minimum 256
local optimization 162, 170
logfile 300
longitudinal excursion 149
loop
for 10
while 10
lower bound 276
LP 235

M-exposed vertex 228
machine
random access 13
Turing 13
macro 296, *310–314*
magnetization 74, 75, 84, 121, 250, 343
make 314–317
makefile 297, 314–317
man page 314, 323
Maple 319
Markov process 76
mass 179
mass of cluster 83
mass-balance constraint 139
master equation 76
matching *228–230*, 231
maximum-cardinality 228, 234
maximum-weight 229
non-bipartite
time complexity 242
perfect 228
weighted 229
matching algorithm 194, *231–250*
matching polytope theorem 244
mates 229
mathematical description
of an optimization problem 2
Matlab 164
matrix
adjacency 46
diagonalization 319
maximization problem 2
maximum flow 43, 103, 106, 192
maximum-cardinality matching 228, 234
time complexity 234
maximum-flow algorithm 143
maximum-flow problem 234
maximum-weight matching 229
mean-field model 191, 219
memory allocation 335
memory leak 337
mergesort 30
meta rule 315
meteorology 163
method *see* algorithm
methods for objects 300
MF 191
microcanonical ensemble 74
minimization problem 2
minimum cut *99*, 101, 115
minimum cuts
all 115–121
minimum edge cover 50
minimum spanning tree 69
algorithm 68–70
minimum vertex cover 275
minimum-cost flow 43, 68
minimum-cost path 63
minimum-cost-flow algorithm 139–147
minimum-cost-flow problem *139*, 235
minimum-weight perfect matching 235–250
algorithm 242–250
time complexity 249
miscellaneous statement 12
model
Edwards-Anderson 186
Ising 73, 79, 84, 87, 92, 93, 204
mean-field 191
random-field *92–96*, 192
Sherrington-Kirkpatrick 190, 220
module 297, 306
comment 308
Molecular Dynamics simulation 298
momentum conservation 298
monomer 263, 267
Monte Carlo
algorithm 77, 255–270
optimization 78
simulation 76, 94, 191, 207, 209–211, 252, 270

multi-canonical ensemble 260
mutation 159, *161*, 201
  operation 166, 173, 182

negation 19
negative costs 132
negative cycle 68, 132
negative flow 43
negative-cycle theorem 141
negative-cycle-canceling algorithm 68, *141–143*
  time complexity 143
neighbor node 37
neighbour configurations 210
network 42, 96–102, 129, 192
  level 107
  random fuse 96
  random resistor 61
  residual *103*, 105, 108, 134
neural network 186, 195
neutron scattering 187
N-line problem 129–135, 138
N-line algorithm 135
node *37*
  adjacent 37
  father 39
  isolated 37
  neighbor 37
  sink 42
  son 39
  source 42
node potential 132, 134, 143
nondeterministic algorithm 23
nondeterministic polynomial 22
nonlinear equation 319
normal distribution 330
NOT operation 19
NP 22
NP-complete *19–26*, 50
  graph problem 50–51
NP-hard *25*, 163, 185, 194, 220, 228, 273, 274
N-queens problem 32–33
ν 54, 82, 123, 252
nuclear reactions 163
number of ground states 124
number-partitioning problem 273
Numerical Recipes 294, *319–321*

O/Θ notation 17
object 294, *300–306*
object-oriented programming 300–306
odd cut 243
odd level 233
offset 136
offspring *161*, 199
one-dimensional quantum system 172–175
operation 295
  AND 19
  basic 296
  boolean 19
  crossover 166, 173, 182
  find 48
  insert 41, 49
  mutation 166, 173, 182
  NOT 19
  OR 19
  remove 41
operator overloading 302
optimization
  Monte Carlo 78
optimization algorithms 5
  applications of 1–2
optimization problem
  combinatorial 2
  discrete 2
  mathematical description 2
OR operation 19
orbital parameters of galaxies 178–183
order parameter 78, 84, 138
orthogonality conditions 237
outgoing edge 38
overflowing vertex 109
overlap 125, 191

P 22, 25, 50, 91
parallel tempering 260–262
paramagnet 80, 94
paramagnetic phase 84
parameters 294
parent 161, 199
partial function 13
partition function 74, 229
Pascal 293, 300
path 37
  alternating *229*, 230

augmenting *103*, 105, 107, 230, 232–234, 242
closed 37
length of 37
minimum cost 63
shortest 63, 107, 132
path-optimality condition 69
p_c 54, 81–84
pdf 348
percolating cluster 82
percolation 53–61, *81–84*, 275
perfect matching 228
periodic boundary conditions 205
perl 318
PERM 255
PERM procedure 265
phantom edge 101
phase
ferromagnetic 84
paramagnetic 84
phase diagram 78, 94, 276
phase transition 2, 4, *78–81*, 84, 136, 187, 273
continuous 78
discontinuous 78
first order 79
second order 78
Phys Net 349
phyton 318
pidgin Algol *9–12*, 13, 17
assignment 10
cases 10
comment 12
conditions 10
for loop 10
goto 11
miscellaneous statement 12
procedures 11
return 11
while loop 10
P_∞ 54
planar graph 227
plaquette 227
PM 94
point defect 151
pointer implementation
lists 44
trees 45
polymer 262
directed 51
polynomial 22
polynomial-time reducible 23
polytope 236
population *160*, 199
time evolution 170
post processing 209
postscript 340, 348, 351–353
potential lemma 144
Povray 353–355
$P(q)$ 125
preflow *109*
preprint server 347
pressure 79
Prim's algorithm 69, *70*
primal linear problem 236
primal-dual algorithm 239
principle
backtracking 32
divide-and-conquer 30, 31
priority queue 41, 66
`private` 301
probability density 330
problem
assignment 235
convex minimum-cost-flow 136–139
correctness 16
decidable 16
EC 50
flux-line 147–150
graph 50–51
halting 16
HC 50
maximum-flow 234
minimum-cost-flow *139*, 235
N-line 129–135, 138
NP-complete 50–51
N queens 32–33
number-partitioning 273
provable 16
recognition *16*, 23
satisfiability *19–22*, 23, 25, 273
sorting 28
traveling salesman *2–3*, 12–13, 25, 26, 42, 51, 195
vertex-cover 273, *274 291*
problem enumeration 229
procedural programming 300
procedure

bit-sequence 165
change-y 247
crossover 166
expand-update 247
extend-alternating-tree 233
local optimization 170
mutation 166
PERM 265
procedures 11
programming
dynamic 31
style 306–310
techniques 26–34
properties of shortest paths 64
protein folding 266–270
provable problem 16
prune-enriched Rosenbluth method 262–270
pseudo flow 143
pseudo node 242
`public` 302
push-relabel algorithm 107

q 125
quality of fit 342
quantum dots 172
quantum Monte Carlo algorithm 266
quantum system
one-dimensional 172–175
quenched disorder 3, 87, 92, 186, 324
queue 41, 321
implementation 44
priority 41

random access machine 13
random bond ferromagnet 87
random elastic medium 96
random fuse network 96
random graph 275
random number generator 324–331
random resistor network 61
random-bond ferromagnet 98
random-bond system 190
random-field Ising magnet 91, *92–96*, 99, 115, 121–125, 192, 260
random-walk algorithm 255
raytracer 353
$Rb_2Cu_{1-x}Co_xF_4$ 190
RCS 297
README 300
realization 92
reciprocal lattice 139
recognition problem 16, 23
recurrence equation 28–30
recursion 27, 30
reduced costs 64, 67, 133, 135, 143, 244
optimality theorem 144
reduced edge length 67
reduced graph 117, 118
rejection method 328–330
relative velocity 179
remove operation 41
renormalization-group theory 81
replica 212
reproduction 159
residual costs 140
residual graph *103*
residual network *103*, 105, 108, 134
return statement 11
reversed topological order 109
Revision Control System 297
RFIM *92–96*, 99, 115, 121–125, 192, 260
right son 40
right subtree 40
RKKY interaction 188
root 39
rough surface 136
roughness 149
running time 17
worst case 17
rutile 92

s-t cut 96
SAT *19–22*, 23, 25, 273
satisfiability problem *19–22*, 23, 25, 273
satisfiable formula 19
SAW 262
scaling function 85
scaling plot 85, 123
scaling relation 81, 85
SCC *38*, 116
Schrödinger equation 172
SCI 348
Science Citation Index 348
scientific journals 348
screening length 139
script 293, *317–319*
search

engine 349
in a tree 48–49
tree 39
second order transition 78
secondary structure 270
segmentation fault 334
self-averaging 88
self-avoiding walk 262
semiconductor 172
sequential algorithm 27
set 321
Sherrington-Kirkpatrick model 190, 220
shortest path 63, 107, 132
algorithm 61–68
properties of 64
Shubnikov phase 129
σ 81
simplex algorithm 236
simulated annealing 78, 194, *257–259*, 296
simulation
Molecular Dynamics 298
Monte Carlo 76, 94, 191, 207, 209–211, 252, 270
Sine-Gordon model 137
singly connected edges 59
sink 42, 96, 102
size
of cluster 54
of input 16
SK model 190
small tasks 295
social system 186
software
cycle 300
development 294
engineering 293–300
reuse 294, 303
solid-on-solid model *136–138*, 150–154
son 39
left 40
right 40
sorting 28
SOS *136–138*, 150–154
source 42, 96, 102
source-code debugger 297, 332
source-code management system 296
spanning forest 58
spanning tree 58
sparse graph 46, 301
specific heat 75, 85, 187, 288
spin 73, 80, 87, 91, 179
free 192, 210, 218
XY 76
spin glass 3–4, 25, 26, *185–192*, 296, 324, 338, 343
algorithms 192–202, 208–219
computational results 203–208, 219–222
experimental results 187–190
four-dimensional 202
theoretical approaches 190–192
three-dimensional 199, 202, 219
two-dimensional 194, 199, 202, 227, *250–252*
square lattice 312–314
stack 41, 295, 321
frame 337
implementation 44
staggered magnetization *95*
standard error bar 78
statement
assignment 10
cases 10
comment 12
for 10
goto 11
miscellaneous 12
return 11
statistical physics 73–90
steepest descent algorithm 170
step function 173
stiffness energy 205, 251
stiffness exponent 207
stock market 186
string 321
strongly-connected component *38*, 116
struct 295
style 306–310
subgraph 37
subtree 39
left 40
right 40
successive-shortest-path algorithm *135*, 139
time complexity 147
super-rough surface 137
superconductor 129, 137, 138

superexchange interaction 92, 189
surface
  rough 136
  super-rough 137
susceptibility 75, 84, 187, 188, 288
  dynamic 187
swap configurations 260
symmetric difference 231

tail of edge 38
tail of list 41
target 314
tasks 295
τ 81
T_c 78, 80, 84, 136
technique
  diagonalization 13–16
temperature 74, 79
  zero 75
testing 297
  tools 331–338
TEX 349
theorem
  augmenting path 231
  central limit 330
  Edmonds 242
  matching polytope 244
  negative-cycle 141
theoretical approaches for spin glasses 190–192
thermal expectation value 74
θ 155
theta point 264
θ-polymer 262
Θ_S 207, 251
three-dimensional spin glass 202
three-dimensional spin glasses 199, 219
threshold accepting 256
Tiff 352
time complexity 16–19
  Dinic's algorithm 107
  Ford-Fulkerson algorithm 105
  Hungarian algorithm 239
  maximum-cardinality matching 234
  minimum-weight perfect matching 249
  negative-cycle-canceling algorithm 143
  N-line algorithm 135
  non-bipartite matching 242
  successive-shortest-path algorithm 147
  vertex cover 288
  wave algorithm 107
time evolution of population 170
TiO_2 92
T_{irr} 95
top down approach 295
topological order 108
  reversed 109
`tqli()` 321
transfer matrix 194
transformation
  gauge 95
  network $\leftrightarrow$ RFIM 96–102
transition
  continuous 78
  discontinuous 78
  first order 79
  second order 78
transition probability 76
traveling salesman problem *2–3*, 12–13, 25, 26, 42, 51, 195
  heuristic *12–13*, 17–18
`tred2()` 321
tree *39–41*, 48, 321
  alternating 231
  balanced 49
  binary 40, 279
  complete 41
  depth-first spanning 59
  height 39
  implementation 45–46
  minimum spanning 69
  search 39
  spanning 58
  subtree 39
triadic crossover 199
truth table 20
TSP *2–3*, 12–13, 25, 26, 42, 51, 195
  heuristic *12–13*, 17–18
Turing machine 13
two-dimensional spin glass 194, 199, 202, 227, *250–252*
typical-case complexity 273

ultrametricity 191
unbounded disorder 150

units 179
universality 80, 87, 123, 124
  class 87
  in disordered systems 89
UNIX 293, 310, 335
upper bound 17, 276

valley 210, 221
  size 216–218, 221
vapor 78
variable 306
  boolean 19
  global 307
VC 273
vector potential 139, 154
vertex *37*, 228, 295
  adjacent 37
  balanced 109
  blocked 110
  covered 228
  degree 295
  exposed 228
  father 39
  free 228, 286
  isolated 37
  M-exposed 228
  matched 228
  neighbor 37
  overflowing 109
  sink 42
  son 39
  source 42
vertex cover 51, 273, *274–291*
  analytical result 276
  heuristic 277
  minimum 275
  results 287–291
  time complexity 288
  x_c 276, 289
vortex glass *138–139*, 154–155

`-Wall` option 298
water 78
wave algorithm *107–115*, 121
  time complexity 107
wave function 172
weakly fluctuating region 150
weighted matching *229*
while loop 10
worst case running time 17

X-ray data 163
xfig 351–353
ξ 54, 82, 84
xmgr 340
XY model 138
XY spin 76

Yahoo 349
yes-instance 22, 23
YFe_2 189

zero temperature 75