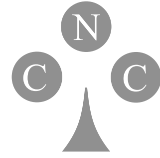
Natural Computing Series

Bastien Chopard · Marco Tomassini

An Introduction to Metaheuristics for Optimization

Natural Computing Series



Series Editors: G. Rozenberg
Th. Bäck A.E. Eiben J.N. Kok H.P. Spaink

Leiden Center for Natural Computing

Advisory Board: S. Amari G. Brassard K.A. De Jong C.C.A.M. Gielen T. Head L. Kari L. Landweber T. Martinetz Z. Michalewicz M.C. Mozer E. Oja G. Păun J. Reif H. Rubin A. Salomaa M. Schoenauer H.-P. Schwefel C. Torras D. Whitley E. Winfree J.M. Zurada

More information about this series at http://www.springer.com/series/4190

Bastien Chopard • Marco Tomassini

An Introduction to Metaheuristics for Optimization

Bastien Chopard
Département d'informatique
Université de Genève
Carouge, Switzerland

Marco Tomassini
Faculté des hautes études commerciales (HEC)
Université de Lausanne
Lausanne, Switzerland

ISSN 1619-7127
Natural Computing Series
ISBN 978-3-319-93072-5 ISBN 978-3-319-93073-2 (eBook)
https://doi.org/10.1007/978-3-319-93073-2

Library of Congress Control Number: 2018959278

© Springer Nature Switzerland AG 2018
This work is subject to copyright. All rights are reserved by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed.
The use of general descriptive names, registered names, trademarks, service marks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.
The publisher, the authors and the editors are safe to assume that the advice and information in this book are believed to be true and accurate at the date of publication. Neither the publisher nor the authors or the editors give a warranty, express or implied, with respect to the material contained herein or for any errors or omissions that may have been made. The publisher remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

This Springer imprint is published by the registered company Springer Nature Switzerland AG
The registered company address is: Gewerbestrasse 11, 6330 Cham, Switzerland

We would like to express our gratitude to our wives, Joanna and Anne, for their encouragement and patience during the writing of the book.

Preface

Heuristic methods are used when rigorous ones are either unknown or cannot be applied, typically because they would be too slow. A metaheuristic is a general optimization framework that is used to control an underlying problem-specific heuristic such that the method can be easily applied to different problems. In the last two decades metaheuristics have been successful for solving, or at least for obtaining satisfactory results in, the optimization of many difficult problems. However, these techniques, notwithstanding their common background and theoretical underpinnings, are rather varied and not easy to grasp for the beginner. Most previous books on the subject have been written for the specialist, with some exceptions, and therefore require knowledge that is not always available to undergraduates or scholars coming from other disciplines and wishing to apply the methods to their own problems.

The present book is an attempt to produce an accessible introduction to metaheuristics for optimization for exactly these kinds of readers. The book builds on notes written for full-semester lecture courses that both authors have been giving for about a decade in their respective universities to advanced undergraduate students in Computer Science and other technical disciplines. We realized during our teaching that there were almost no texts at the level we targeted in our lectures; in spite of the existence of several good books at an advanced level, many of those had prerequisites that were not matched by the typical students taking our courses, or were multi-author compilations that assumed a large body of previous knowledge. Thus, our motivation was to try to write a readable and concise introduction to the subject matter emphasizing principles and fundamental concepts rather than trying to be comprehensive. This choice, without renouncing rigor when needed, should be an advantage for the newcomer as many details are avoided that are unnecessary or even obtrusive at this level. Indeed, we are especially concerned with "how" and "why" metaheuristics do their job on difficult problems by explaining their functioning principles in simple terms and on simple examples and we do not try to fully describe real case studies, although we do mention relevant application fields and provide pointers to more advanced material. One feature that differentiates our approach is probably also due to our respective scientific origins: we are physicists who have done teaching and research on computer science and interdisciplinary fields and we would like

to bring a “computational science” and “complex systems” orientation to the book rather than an application-based one.

The book should be useful for advanced undergraduates in computer science and engineering, as well as for students and researchers from other disciplines looking for a concise and clear introduction to metaheuristic methods for optimization. The contents of the book reflect the choices explained above. Thus, we have given priority to well-known and well-established metaheuristics. After an introductory chapter devoted to standard computability and complexity ideas, we describe a concept that is fundamental in metaheuristics: the search space. After this basic knowledge we describe the main metaheuristics in succeeding chapters: Tabu search, Simulated Annealing, Ant Colony methods, and Particle Swarms. Chapter 7 contains an introduction to newer metaheuristics, such as Fireflies, which are not yet as well established but which could become important in the near future. Chapters 8 and 9 are devoted to Evolutionary Algorithms. Chapters 1-9 constitute the fundamental part of the book; altogether they present the basic notions that any student and practitioner should possess about metaheuristics. The following chapters are a bit more specialized but are still very accessible from the technical viewpoint. Chapter 10, which is a little more technical than the others, is rather unique in current computer science books at this level as it brings a statistical physics approach to computational complexity. This chapter can be skipped without consequences but the ensemble mean difficulty of a class of random problem instances is a valuable point of view when contrasted with the standard worst-case complexity approach. Finally, Chapters 11 and 12 are devoted to a more detailed statistical study of the performance of metaheuristics and of the structure of problem search spaces.

In keeping with our general philosophy of simplicity, we have deliberately chosen not to present multi-objective and constrained optimization, which are very important in practice but require a number of new concepts to be introduced. In the same vein, there are no explicit problems for the reader to solve in the book. Theoretical problems doable with pencil and paper would probably be inappropriate for the level of the book; on the other hand, numerical solutions to specific questions would certainly be very useful. Today there exist a number of excellent downloadable open software systems for several languages and environments that cover most of the methods presented in the book. The reader would be well advised to try out one or more of these and, to this end, we provide a list of suggestions in an appendix.

We would like to thank many colleagues and collaborators for comments and discussions. They have, in one way or another, contributed to our present understanding of metaheuristics. M. Tomassini acknowledges in particular P. Collard, M. Giacobini, G. Ochoa, L. Vanneschi, and S. Vérel for many stimulating discussions during our joint work. He also thanks his former Ph.D. student F. Daolio for his help with several figures and numerical computations in Chapters 8 and 12. We would also like to express our appreciation to the Springer staff, and in particular to Ronan Nugent, whose help and support were key during the whole process. B. Chopard thanks E. Taillard for hints and advice on the Traveling Salesman problem and simulated annealing. He also thanks R. Monasson and G. Semerjian for their feedback on the chapter on computational phase transitions.

September 2018
Bastien Chopard, Marco Tomassini

Contents

1 Problems, Algorithms, and Computational Complexity . . . 1
1.1 Computational Complexity . . . 1
1.2 Analysis of Algorithms and Their Complexity . . . 2
1.2.1 Operations and Data Size . . . 2
1.2.2 Worst-Case Complexity . . . 3
1.3 Decision Problems and Complexity Classes . . . 5
1.4 The Complexity of Optimization Problems . . . 7
1.5 Do We Need Metaheuristics? . . . 8

2 Search Space . . . 15
2.1 Search Space . . . 15
2.2 Examples . . . 16
2.2.1 Functions in $\mathbb{R}^n$ . . . 16
2.2.2 Linear Programming . . . 18
2.2.3 NK-Landscapes . . . 18
2.2.4 Permutation Space . . . 20
2.3 Metaheuristics and Heuristics . . . 22
2.4 Working Principle . . . 23
2.4.1 Fitness Landscapes . . . 24
2.4.2 Example . . . 25
2.5 Moves and Elementary Transformations . . . 26
2.6 Population Metaheuristics . . . 30
2.7 Fundamental Search Methods . . . 31
2.7.1 Random Search and Random Walk . . . 31
2.7.2 Iterative Improvement: Best, First, and Random . . . 32
2.7.3 Probabilistic Hill Climber . . . 32
2.7.4 Iterated Local Search . . . 33
2.7.5 Variable Neighborhood Search . . . 34
2.7.6 Fitness-Smoothing Methods . . . 34
2.7.7 Method with Noise . . . 36
2.8 Sampling Non-uniform Probabilities and Random Permutations . . . 37

2.8.1 Non-uniform Probability Sampling 37
2.8.2 Random Permutations 39

3 Tabu Search 43
3.1 Basic Principles 43
3.2 A Simple Example 44
3.3 Convergence 46
3.4 Tabu List, Banning Time, and Short- and Long-Term Memories 48
3.4.1 Principles 48
3.4.2 Examples 49
3.5 Guiding Parameters 50
3.6 Quadratic Assignment Problems 51
3.6.1 Problem Definition 51
3.6.2 QAP Solved with Tabu Search 53
3.6.3 The Problem NUG5 54

4 Simulated Annealing 59
4.1 Motivation 59
4.2 Principles of the Algorithm 60
4.3 Examples 62
4.4 Convergence 69
4.5 Illustration of the Convergence of Simulated Annealing 70
4.6 Practical Guide 74
4.7 The Method of Parallel Tempering 76

5 The Ant Colony Method 81
5.1 Motivation 81
5.2 Pheromones: a Natural Method of Optimization 82
5.3 Numerical Simulation of Ant Movements 85
5.4 Ant-Based Optimisation Algorithm 87
5.5 The “Ant System” and “Ant Colony System” Metaheuristics 92
5.5.1 The AS Algorithm Applied to the Traveling Salesman Problem 92
5.5.2 The “Ant Colony System” 94
5.5.3 Comments on the Solution of the TSP Problem 96

6 Particle Swarm Optimization 97
6.1 The PSO Method 97
6.2 Principles of the Method 97
6.3 How Does It Work? 99
6.4 Two-Dimensional Examples 100

7 Fireflies, Cuckoos, and Lévy Flights . . . 103
7.1 Introduction . . . 103
7.2 Central Limit Theorem and Lévy Distributions . . . 103
7.3 Firefly Algorithm . . . 107
7.4 Cuckoo Search . . . 109
7.4.1 Principle of the Method . . . 110
7.4.2 Example . . . 112

8 Evolutionary Algorithms: Foundations . . . 115
8.1 Introduction . . . 115
8.2 Genetic Algorithms . . . 116
8.2.1 The Metaphor . . . 116
8.2.2 Representation . . . 116
8.2.3 The Evolutionary Cycle . . . 117
8.2.4 First Example . . . 118
8.2.5 Second Exemple . . . 123
8.2.6 GSM Antenna Placement . . . 125
8.3 Theoretical Basis of Genetic Algorithms . . . 127
8.4 Evolution Strategies . . . 132

9 Evolutionary Algorithms: Extensions . . . 139
9.1 Introduction . . . 139
9.2 Selection . . . 139
9.2.1 Selection Methods and Reproduction Strategies . . . 140
9.2.2 Selection Intensity . . . 144
9.2.3 Analytical Calculation of Takeover Times . . . 145
9.3 Genetic Programming . . . 147
9.3.1 Representation . . . 148
9.3.2 Evaluation . . . 150
9.3.3 Genetic Operators . . . 150
9.3.4 An Example Application . . . 151
9.3.5 Some Concluding Considerations . . . 153
9.4 Linear Genetic Programming . . . 154
9.4.1 Example . . . 155
9.4.2 Control Structures . . . 157
9.5 Structured Populations . . . 158
9.6 Representation and Specialized Operators . . . 166
9.7 Hybrid Evolutionary Algorithms . . . 168

10 Phase Transitions in Combinatorial Optimization Problems . . . 171
10.1 Introduction . . . 171
10.2 The k-XORSAT Problem . . . 172
10.3 Statistical Model of k-XORSAT Problems . . . 173
10.4 Gaussian Elimination . . . 174
10.5 The Solution Space . . . 176

10.5.1 The **1**-XORSAT Case . . . 176
10.5.2 The **2**-XORSAT Case . . . 178
10.5.3 The **3**-XORSAT Case . . . 181
10.6 Behavior of a Search Metaheuristic . . . 183
10.6.1 The RWSAT Algorithm . . . 183
10.6.2 The Backtracking Algorithm . . . 186

11 Performance and Limitations of Metaheuristics . . . 191
11.1 Empirical Analysis of the Performance of a Metaheuristic . . . 191
11.1.1 Problem Choice . . . 192
11.1.2 Performance Measures . . . 193
11.1.3 Examples . . . 194
11.2 The "No Free Lunch" Theorems and Their Consequences . . . 200

12 Statistical Analysis of Search Spaces . . . 205
12.1 Fine Structure of a Search Space . . . 205
12.2 Global Measures . . . 206
12.3 Networks of Optima . . . 207
12.4 Local Measures . . . 209
12.4.1 Fitness-Distance Correlation . . . 209
12.4.2 Random Walks and Fitness Autocorrelation . . . 212

Appendices . . . 215

References . . . 219

Index . . . 223

1

Problems, Algorithms, and Computational Complexity

1.1 Computational Complexity

Metaheuristics are a family of algorithmic techniques that are useful for solving difficult problems. Roughly speaking, the difficulty or hardness of a problem is the quantity of computational resources needed to find the solution. When this quantity increases at a high rate with increasing problem size, in a way that will be defined precisely later, we are facing a difficult problem. The theory of the computational complexity of algorithmic problems is well known [34, 66] and, in this first chapter, we shall look at the basics and the main conclusions since these ideas are needed to understand the place of metaheuristics in this context.

By and large, computational problems can be divided into two categories: computable or decidable problems and non-computable or undecidable ones. Non-computable problems cannot be solved, in their general form, by any computational device whatever the computational resources at hand. One of the archetypal problems of this class is the *halting problem*: given a program and its input, will the program halt? There is no systematic way to answer this question for arbitrary programs and inputs. Computability is important in logic and mathematics but we shall ignore it in the following. On the other hand, for computable and decidable problems there are computational procedures that will give us the answer in finite time. These procedures are called *algorithms* and once one or more algorithms are known for a given problem, it is of interest to estimate the amount of work needed to obtain the result. Under the hypothesis that the computational device is a conventional computer, the relevant resources are the time needed to complete the computation and the memory space used. However, in theory the use of an electronic computer is by no means necessary: the "device" could be a mathematician equipped with pencils, an endless tape of paper, and a lot of patience. Indeed, the fundamental theoretical results have been established for an elementary automaton called a *Turing machine*; a modern computer is computationally equivalent to a Turing machine but it is much faster. In the present day memory space is usually not a problem and this is the reason why we are more interested in computing time as a measure of the cost of a computation.

© Springer Nature Switzerland AG 2018

B. Chopard, M. Tomassini, *An Introduction to Metaheuristics for Optimization*,
Natural Computing Series, https://doi.org/10.1007/978-3-319-93073-2_1

Several decades of research on computable problems have led to the conclusion that, to a first approximation, a problem may belong to one of two non-disjoint classes: either the class of problems for which the computation time is bounded by a polynomial of the size of the problem instance, or the class of problems for which a correct answer can be checked in such time. The former class is called P, which stands for *polynomial*, and the second has been called NP, which means *non-deterministic polynomial*.

Many important algorithmic problems belong to the class P; for example, searching for an element in a data structure, sorting a list, finding the shortest path between two vertices in a connected graph, finding a spanning tree of a graph, solving a system of linear equations, and many others. All these problems admit of *efficient* solutions, in the sense that there are algorithms for them that give the correct answer in time that is a polynomial function of the size of the problem instance considered. Here "instance" simply means a particular case of the problem at hand, e.g., a given graph or a given list of numbers. In addition, it is found in practice that the polynomial is of low degree, usually first, second, or third at most. It is clear that this kind of problem does not need metaheuristics or other approximate methods to be solved efficiently given that exact efficient algorithms already exist.

The situation is different for problems not belonging to the class P, which seem to require an amount of time to be solved that grows exponentially with the instance size, a time that becomes quickly impractical even for current-generation computers. There are many problems of this kind and a non-exhaustive list can be found in the classical book by Garey and Johnson [34]. Among these problems one can mention the satisfiability problem in logic, the existence of Hamiltonian cycles in graphs, graph partitioning, and many scheduling and sequencing problems. The important point is that many of these problems are not only of theoretical interest; rather, they arise naturally in several fields that are relevant in practice such as operations research and logistics. For these, the possibility of obtaining a perhaps not optimal but at least satisfying solution would be very valuable.

In the rest of the chapter we shall give an introduction to the issues of computational complexity and to the practical ways in which one can fight the inordinate growth of computing times.

1.2 Analysis of Algorithms and Their Complexity

1.2.1 Operations and Data Size

To establish the efficiency of an algorithm we need to know how much time it will take to solve one of a class of instances of a given problem. In complexity theory one makes use of the following simplification: the elementary operations a computer can perform such as sums, comparisons, products, and so on all have the same cost, say one time unit. This is obviously imprecise since, for instance, a division actually takes more machine cycles than a comparison but we will see later that this does not

change the main conclusions, only the actual computing times. Under this hypothesis, the time taken by an algorithm to return the answer is just the sum of the number of elementary operations executed until the program stops. This is called the *uniform cost* model and it is widely used in complexity studies. In the case of numerical algorithms such as finding the greatest common divisor, or primality testing, very large numbers may appear and the above hypothesis doesn't hold as the running time will depend on the number of digits, i.e., on the logarithm of the numbers. However, the uniform cost model can still be used provided that the numbers are sufficiently small such that they can be stored in a single computer word or memory location, which is always the case for the problems we deal with in this book.

The data structures that an algorithm needs to compute an answer may be of many different kinds depending on the problem at hand. Among the more common data structures we find lists and sequences, trees, graphs, sets, and arrays. The convention is to consider as an input size to a given algorithm the length of the data used. Whatever the structure at hand, ultimately its size can be measured in all cases by the number of bits needed to encode it.

1.2.2 Worst-Case Complexity

In computational complexity analysis one is not really interested in the exact time it takes to solve an instance of a problem on a given machine. This is useful information for practical purposes but the result cannot be generalized since it depends on the machine architecture, on the particular problem instance, and on software tools such as compilers. Instead, the emphasis is on the functional form $T(N)$ that the computation time takes as the input data size N grows. However, instances of the same size may generate different computational costs. For instance, in the linear search for a particular element x in an unsorted list, if the element is at the first place in the list, only one comparison operation will suffice to return the answer. On the other hand, if x doesn't belong to the list, the search will examine all the N elements before being able to answer that x is not in the list. The *worst case* complexity analysis approach always considers the case that will cause the algorithm to do the maximum work to solve the problem, thus providing an upper bound to $T(N)$.

Other possibilities exist. In *average-case* complexity analysis the emphasis is on the execution cost averaged over all the problem instances of size N, assuming a given distribution of the inputs. This approach seems more realistic but it quickly becomes mathematically difficult as soon as the distribution of the inputs becomes more complex that the uniform distribution in other words, it is limited by the probability assumptions that have been made about the input distribution. In the end, worst-case analysis is more widely used since it provides us with a guarantee that the given algorithm will do its work within the established bounds, and it will also be used here, except in the cases that will be explicitly mentioned in the text.

Let's focus our attention again on the behavior of the performance function $T(N)$. $T(N)$ must be a strictly increasing function of N since the computational effort cannot stay constant or decrease with increasing N. In fact, it turns out that only

a few functional forms appear in algorithm complexity analysis. Given such a function, we want to characterize its behavior when N grows without bounds; in other words, we are interested in the asymptotic behavior of $T(N)$. We do know that in the finite world of actual computing machines such a limit cannot really be reached but we can always take it in a formal sense. The result is customarily expressed through the "$\mathcal{O}''$ notation (for the technical details see a specialized text such as [26]) thus

$$T(N) = \mathcal{O}(f(N)) \tag{1.1}$$

This is interpreted in the following way: there exist positive constants k and N_0 such that $T(N) \leq k\ f(N),\ \forall N > N_0$, that is to say, for N large enough, i.e., asymptotically, $f(N)$ will be an upper bound of $T(N)$. Asymptotic expressions may hide multiplicative constants and lower-order terms. For instance, $0.5\ N^2 + 2\ N = \mathcal{O}(N^2)$. It is clear that expressions of this type cannot give us exact computation times but they can help us classify the relative performance of algorithms.

Recapitulating, the asymptotic interpretation of the computational effort of an algorithm gives us a tool to group algorithms into classes that are characterized by the same time growth behavior. Table 1.1 shows the growth of some typical $T(N)$ functions for increasing N. Clearly, there is an enormous difference between the very slow growth rate of a logarithmic function, or even of a linear function, as compared to an exponential function, and the gap increases extremely quickly with N. Moreover, for the functions in the lower part of the table, computing times are very large even for sizes as small as 50. The commonly accepted dividing line is between problems that can be solved by algorithms for which the running time is bounded by a polynomial function of the input size, and those for which running time is super-polynomial, such as an exponential or a factorial function. Clearly, a polynomial function of degree 50 would not be better than an exponential function in practice but it is found that the polynomial algorithms that arise in practice are of low degree, second or third at most. It is also to be noted that some functions on the polynomial side, such as $logN$ and $NlogN$, are not polynomial but they appear often and are certainly bounded by a polynomial. Likewise, $N!$ is not exponential but it dominates any polynomial function.

Function	Function Value			
$logN$	1	1.699	2	3
N	10	50	100	1,000
$NlogN$	23.026	765.2	460.52	6,907.75
N^2	100	2,500	10,000	10^6
N^3	1,000	125,000	10^6	10^9
2^N	1,024	1.126×10^{15}	1.27×10^{30}	1.05×10^{301}
10^N	10^2	10^{50}	10^{100}	$10^{1,000}$
$N!$	$3,628.8 \times 10^3$	3.041×10^{64}	10^{158}	4×10^{2567}

Table 1.1. Growth rate of some functions of the instance input size N

The horizontal line in Table 1.1 separates the "good" functions from the "bad" ones, in the sense that problems for which only exponentially bounded algorithms are known are said to be *intractable*, which does not mean that they cannot be solved, they would if we waited long enough, but the computation time can become so large that in practice we cannot afford to solve them exactly. Clearly, the frontier between tractable and intractable problems is a fuzzy and moving one and, thanks to advances in computer speed, what was considered at the limits of intractability twenty years ago would be tractable today. However, exponentially bounded algorithms only allow moderate increases in the size of the instances that can be solved exactly with increasing computer power. In contrast, algorithms whose running time is bounded by a polynomial function of N will fully benefit from computer performance increases.

1.3 Decision Problems and Complexity Classes

Using the notions presented above, we shall now give a summary of the classification of computational problems and their algorithms according to standard theoretical computer science. The interested reader will find a much more complete description in specialized books such as Papadimitriou's [66]. The theory is built around the concept of *decision problems*, i.e., problems that require a "yes" or "no" answer. More formally, $\mathcal{P}$ is a decision problem if the set of instances $I_{\mathcal{P}}$ of $\mathcal{P}$ can be partitioned into two sets: the set of "positive" instances $Y_{\mathcal{P}}$ and the set of "negative" instances $N_{\mathcal{P}}$. Algorithm $A_{\mathcal{P}}$ gives a correct solution to the problem if, for all instances $i \in Y_{\mathcal{P}}$ it yields "yes" as an answer, and for all instances $i \in N_{\mathcal{P}}$ it yields "no" as an answer.

The theory of computational complexity developed during the 1970s basically says that decision problems can be subdivided into two classes: the class P of those problems that can be solved in polynomial time with respect to the instance size, and the class NP of those for which a correct "yes" answer can be checked in polynomial time with respect to the instance size. The letter P stands for "polynomial", while NP are the initials of "non-deterministic polynomial". Essentially, this expression means that for a problem in this class although no polynomial-time bounded algorithm is known to solve it, if $x \in Y_{\mathcal{P}}$, i.e. x is a positive instance of $\mathcal{P}$ then it is possible to verify that the answer is indeed "yes" in polynomial time. The corresponding solution is called a *certificate*. Another equivalent interpretation makes use of non-deterministic Turing machines, hence the term non-deterministic in NP. We now describe a couple of examples of problems belonging, respectively, to the P and NP classes.

Example 1: Connection of a graph.

Let $G(V, E)$ be a graph with V representing the set of vertices and E the set of edges. G is connected if there is a path of finite length between any two arbitrary vertices $v_1, v_2 \in V$. In this case the decision problem $\mathcal{P}$ consists of answering the following question: is the graph G connected?

It is well known that the answer to the above question can be found in time $0(|V| + |E|)$, which is polynomial in the size of the graph, by the standard breadth-first search. Therefore, the problem is in P.

Example 2: Hamiltonian cycle of a graph.

The problem instance is a non-oriented graph $G(E, V)$ and the question is the following: does G contain a Hamiltonian cycle? A Hamiltonian cycle is a path that contains all the vertices of G and that visits each of them only once, before returning to the start vertex.
No polynomial-time algorithm is know for solving this problem, therefore the problem does not belong to P. However, if one is given a path $\{v_1, v_2, \ldots, v_N, v_1\}$ of G pretending to be a Hamiltonian cycle, it is easy to verify the claim. Indeed, there is only a polynomial amount of data to check, linear in this case, to see whether or not the given cycle is a Hamiltonian cycle. Therefore, the problem is in NP.

We thus see that $P \subseteq NP$ since any decision problem in P admits of a polynomial algorithm by definition and, at the same time, the solution can itself be checked in polynomial time. Thus far, researchers believe that $P \subset NP$, which means that is it is unlikely that somebody will find tractable algorithms for many hard problems.

To conclude this summary of computational complexity, we note that there exists a class of decision problems called NP-complete that play a major role in the theory. Problems in this class all belong to NP but they have an additional striking feature: any NP-complete problem can be reduced to another problem in this class in polynomial time. A reduction of problem $\mathcal{P}_a$ to problem $\mathcal{P}_b$ is a method to solve $\mathcal{P}_a$ using an algorithm for $\mathcal{P}_b$. The reduction is said to be polynomial if the transformation from $\mathcal{P}_a$ into $\mathcal{P}_b$ can be performed in polynomial time. In this sense all the NP-complete problems are equivalent. To show that that a problem $\mathcal{P}$ is NP-complete one must first show that it belongs to NP, meaning that it does not admit of a polynomial-time algorithm but that its positive instances can be validated in polynomial time. After that, one needs to find a polynomial-time reduction of the given problem to any other problem that is already known to belong to the NP-complete class. It is clear that it has been necessary to find a first prototypical NP-complete problem; this has been accomplished by S. Cook through the problem of the satisfiability of a Boolean formula [34, 66]. The important consequence of the theory is that if we could find a polynomial-time algorithm for any NP-complete problem, the consequence would be that all the other NP-complete problems would be solvable in polynomial time as well since they can all be transformed into each other by definition. A list of NP-complete problems can be found in [34]. Finally, we remark that there exist problems that are known not to be in NP. Thus, even if we find a polynomial-time algorithm for NP-complete problems, those problems outside the NP class would still be intractable.

Figure 1.1 graphically summarizes the relation between the complexity classes P, NP, and NP-complete according to present knowledge and assuming that $P \subset NP$. This view is essentially correct but ignores some details that can be found in [66] and in a simpler but extremely readable form in [64].

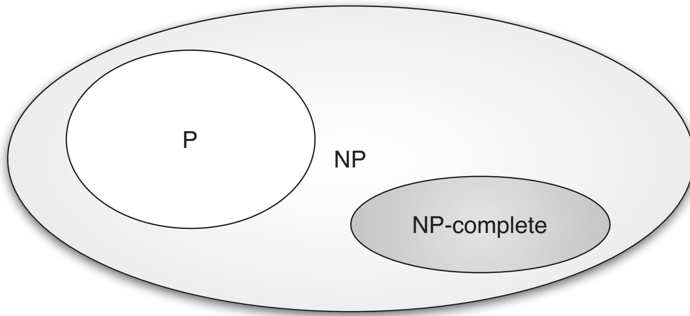


Fig. 1.1. Relations between complexity classes of decision problems assuming that $P \subset NP$

1.4 The Complexity of Optimization Problems

Optimization problems are very important in practice and a class of methodologies for solving them, namely metaheuristics, is the main subject of the present book. Understanding their relationship with the complexity classes briefly presented above is a necessary step in the path that we will take. To this end, we give here a few definitions and concepts about optimization problems especially in relation to their computational complexity. The optimization view will be further developed and deepened in the next chapter, where optimization problems will be examined from the point of view of the structure of the search spaces that characterize their instances.

Informally, an optimization problem consists of finding the solution, or solutions, that maximize or minimize a given criterion variously called the *objective function*, *cost function*, or *fitness function* (we will use these terms interchangeably here). A solution may be required to obey certain constraints, when it is usually called an *admissible or feasible* solution. Many important problems require the simultaneous optimization of more than one objective; this branch of optimization is called multi-objective optimization and it is very important in practical applications. However, it is a more technical and specialized field of study. Thus, for the sake of simplicity, we shall not take it up here and we refer the interested reader to the appropriate literature, e.g., [28].

In more formal terms, if $\mathbf{S}$ is a finite set of feasible solutions to a problem and f is a cost function $f : \mathbf{S} \rightarrow \mathbb{R}$, then an instance of an optimization problem $\mathcal{P}$ asks for $x \in \mathbf{S}$ such that [1]:

$$f(x) \geq f(s), \; \forall s \in \mathbf{S} \tag{1.2}$$

The optimization problem is the set $I_{\mathcal{P}}$ of all the instances of $\mathcal{P}$.

Computational complexity ideas, as presented in the previous sections, strictly apply only to decision problems. Optimization problems, due to their importance in many application areas, are very relevant and therefore it is important to understand

[1] Here we assume maximization. If minimization is implied instead, we have $f(x) \leq f(s), \; \forall s \in \mathbf{S}$. The nature and properties of $\mathbf{S}$ will be explained in Chapter 2.

how complexity theory can be extended from decision to optimization problems. The main ideas are as follows. Once the NP-complete class has been defined, one may ask whether there are other problems at least as difficult as those that have been proved to be NP-complete. A problem is called NP-hard if, in terms of complexity, it is at least as difficult to solve as any problem in NP, apart from a polynomial-time reduction. Thus, if an optimization problem $\mathcal{P}^o$ reduces to an NP-complete one $\mathcal{P}$, then $\mathcal{P}^o$ is NP-hard. In this way, one can show that, in the sense of their running times, NP-complete problems are contained in the more general NP-hard class. Many NP-hard problems are optimization problems, which are not decision problems, and thus cannot be NP-complete, but their solution is at least as hard as that of NP-complete problems.

Let us look at a simple example of this relationship between a difficult optimization problem and its decision form. Consider again a slightly different version of the Hamiltonian cycle decision problem:

Given the undirected graph $G(E, V)$, we ask the following question: does G possess a Hamiltonian cycle of length $L \leq k$?

The optimization version of the problem goes under the name of "Euclidean symmetric traveling salesman problem", or TSP for short, and can be defined thus:

Given the undirected graph $G(E, V)$ with the set of vertices $V = \{v_1, \ldots, v_n\}$ and the $n \times n$ matrix of distances $d_{ij} \in \mathbb{Z}^+$, find a permutation $\Pi(V)$ of V such that the corresponding tour length $L(\Pi) = \sum_{i=1}^{n-1} d_{v_i, v_i+1} + d_{v_n, v_1}$ is minimal.

We already know that the Hamiltonian cycle problem belongs to NP, and it has also been shown to be NP-complete. But the optimization version is at least as hard as the decision problem for, once an optimal solution of length L has been found, the decision version only asks us to compare L to k. In conclusion, although the TSP cannot belong to NP because it is not a decision problem, it is at least as difficult to solve since its solution implies the solution of the corresponding decision problem. Indeed, having found the shortest tour, we are sure that the length we compare to k is the minimal one and therefore the decision problem is solved as well since, if L turns out to be larger than k, no other tour exists that gives a "yes" answer. The preceding qualitative and intuitive ideas can be put into a rigorous form but doing so would lead us out of the scope of the book. The interested reader will find the corresponding technical details in a specialized book such as Papadimitriou's [66].

1.5 Do We Need Metaheuristics?

From the previous pages, the reader might have got the impression that searching for solutions to NP-hard optimization problems is a hopeless enterprise. Indeed, these problems require quickly increasing computing times as the instance sizes we want to solve grow, and at some point they become effectively impractical. The point is that

the algorithms we know for solving them are essentially just complete enumeration of the admissible solutions, whose number grows exponentially or worse with size. In other words, we lack any shortcut that could save us checking most of the solutions, which is so effective for problems in P like sorting. In sorting a list of N numbers we can leverage an ordering principle that limits the work to be done to $N \log N$, as compared to checking all the $N!$ possible solutions. Nevertheless, the fact that large hard problems are solved daily in many fields should encourage us to have a more positive attitude toward these issues. In the following pages, we shall briefly review a number of ideas that all help relieve the computational burden generated by these algorithms.

Special cases of difficult problems.

We start by recalling that theoretical results on computational complexity have been established mainly according to the worst-case scenario. In practice, however, many instances of a hard problem might have a structure that makes them easier to solve in spite of the fact that the worst case is hard. For example, the satisfiability problem with two variables per clause (2-SAT) can be solved in polynomial time while the general case belongs to NP and MaxSat, the optimization version, is NP-hard. An important case is the simplex algorithm for linear programming problems, which has exponential complexity in the worst case, but in practice it is fast and can be routinely applied with success to very large problems. For another example, rather large instances of the knapsack problem, which is NP-hard, can be solved by using partial enumeration and dynamic programming. This is good news but the cases are particular and cannot be generalized to the bulk of the hard problems that are at the core of many real-life applications.

Brute-force computation.

We have already remarked that the constant increase in computer power, at least up to a point where fundamental physical factors impede further progress, might allow us to just use the simplest and most direct method to solve a difficult problem: just generate and test all admissible solutions. A few decades ago this was possible only for small problem instances but today TSP instances with tens of thousands of cities have been solved using partial enumeration and problem knowledge. Clearly, to accomplish this, smart enumeration algorithms and powerful computers are required [25]. But there will always be limits to what can be done even with the fastest computers. For instance, suppose that an algorithm with complexity $\propto 2^N$ takes an hour to solve an instance of size S with a present-day computer; then, with a hundred-fold increase in performance, the instance we can solve in one hour is only of size $S + 6.64$, and it is of size $S + 9.97$ with a computer $1,000$ times faster [34]. As another example, a brute-force attack to find the public key by factoring a large prime in an RSA encryption system is doable with current technology only if the integers used are less than $1,000$ bits in length [26].

Approximation algorithms.

A reasonable way of tackling a difficult problem is perhaps just to satisfy oneself with a good, but not necessarily globally optimal solution. After all, in many practical circumstances when limited time is available to get an answer, it is usually preferable to get a quick satisfactory solution rather than to wait for much longer to get a marginally better one. Thus, if we need to schedule a TSP tour for tomorrow, it is probably better to compute a good tour in ten minutes rather than to wait for a week to obtain an optimum that is only 10% better. *Approximation algorithms* [26] give us just that: they take polynomial time in the size of the problem instance, and thus are faster, and provide us with a certified guarantee that the solution will be nearly optimal up to a certain constant factor. There are approximation algorithms for many NP-hard problems, but sometimes the approximation is almost as difficult as the original as the polynomial may be of high degree, and they are usually difficult to program. It has also been proved that some NP-hard problems are "inapproximable" and thus for them we are left with the original complexity result [66]. In conclusion, approximation algorithms may be very useful when they exist and run sufficiently fast but they cannot be considered to be a general solution to solve difficult problems.

Parallel computing.

We have already given some attention to the fact that improved computation technology has the potential to make tractable what was once considered to be intractable, up to a point. One might be tempted to extend the idea and reason that if a single machine has made such a progress, what could a large number of similar machines working together accomplish? In practice, it is a matter of having many computation streams active at the same time by connecting in some suitable way a number of single machines or processors. Indeed, today even standard laptops are actually multi-processor machines since parallelism in various forms is already present at the chip level, and connecting together several such machines can provide huge computing power. These kinds of architectures are now popular and affordable owing to the diminishing costs of processors, the introduction of cheap graphics processing units, and the high performance of communication networks. These ideas have been fully explored in the last three decades with excellent results. There are however some limitations, both technological and theoretical. In the first place, many algorithms are not easy to "parallelize," i.e., to restructure in such a way that they can be run on parallel hardware with high efficiency. This is mainly due to synchronization phases between the computing streams and communication between the processor memories by message passing, or access to a common shared memory. When these factors are taken into account, one realizes that the ideal speedup, which is n for n processors working in parallel, is almost never achieved. Nevertheless, computer time savings can be substantial and parallel computing is used routinely in many computer tasks.

However, there exist principled reasons leading experts to say that parallel computing can help but cannot fundamentally change the easy/hard frontier delineated previously in this chapter. Let us briefly recall the concept of a non-deterministic

Turing machine, which was mentioned when the class NP of decision problems was introduced. In problems of this class, positive solutions, i.e., those providing a "yes" answer to a question can be checked in polynomial time by definition. Another way of finding a solution is to use a non-deterministic Turing machine (NDTM), an unrealistic but useful theoretical device that spawns computation paths as needed until it either accepts its input or it doesn't. Its computation can be seen as a tree of decisions in which all the paths are followed simultaneously [66]. Thus, a problem $\mathcal{P}$ belongs to NP if for any positive instance of $\mathcal{P}$ such an NDTM gives a "yes" answer in polynomial time in the size of the instance. If we could use a parallel machine to simulate the branching of decisions of the NDTM by allocating a new processor each time there is a new decision to evaluate, we would obtain a physical analogue of the NDTM. But for a problem in NP the number of decisions to make increases exponentially if we want the NDTM to find the answer in polynomial time, which would imply a number of processors that grows exponentially as well! Such a machine is clearly infeasible if we think about physical space, not to speak of the interconnection requirements. The conclusion is clear: parallelism can help to save computer time in many ways and it is well adapted to most of the metaheuristics that will be presented later in the book, however parallelism *per se* cannot fundamentally change the frontier between easy and hard problems. To further explore these issues, the reader is referred to, e.g., [14] for the algorithmic aspects, and to [66] for the complexity theory part.

Linear programming relaxation.

In many cases the optimization task is to maximize or minimize a linear function of non-negative real variables $x_1, \ldots, x_n$ subject to M linear constraints. This is called a *linear programming problem* and can be formulated thus:

$$\text{maximize} \quad \sum_{i=1}^{n} c_i x_i \tag{1.3}$$

$$\text{subject to} \quad \sum_{i=1}^{n} a_{ji} x_i \leq b_j, \quad j = 1, \ldots, M \tag{1.4}$$

$$\text{and} \quad x_j \geq 0, \quad j = 1, \ldots, n \tag{1.5}$$

This type of problem occurs in a variety of practical situations and many hard combinatorial optimization problems can be expressed as linear programming problems using integer variables. The corresponding linear program with integrity constraints does not admit of a polynomial-time algorithmn and, in fact, it is NP-complete [34].

Let us consider the *knapsack problem*, which is NP-hard, as an example.

There are n items with utility $p_j,\ j = 1, \ldots, n$ and "volume" $w_j,\ j = 1, \ldots, n$. The binary variable x_j is equal to 1 if the object j is included, otherwise its value is 0. The solution we look for is a string of objects x that maximizes the objective function $\sum_{j=1}^{n} p_j\ x_j$, with the constraints $\sum_{j=1}^{n} w_j\ x_j \leq c$, where $c > 0$ is the

knapsack capacity. In other words, what is required is to fill a knapsack of capacity c with items having the largest possible total utility.
The knapsack problem is equivalent to the following $0/1$ integer linear programming problem:

$$\text{maximize} \quad \sum_{j=1}^{n} p_j x_j \tag{1.6}$$

$$\text{subject to} \quad \sum_{j=1}^{n} w_j x_j \leq c \tag{1.7}$$

We have seen above that integer linear programming is NP-complete; therefore, as expected, the formal transformation of the knapsack problem into a $0/1$ integer linear programming form does not change the problem complexity. However, through this reformulation it is now possible to "relax" the problem to an ordinary linear programming form by replacing integer variables by real ones $0 \leq x_i \leq 1$. Now we can solve the standard linear programming problem with the simplex method, which is usually fast enough, obtaining the optimal real variables x_i, and to round them to the closest integer. This rather old technique can be used on some occasions but it is not without problems. To start with, many problems do not have obvious reductions to integer linear programming. And there are other difficulties as well. Although a purely integer solution to the relaxed problem is also an optimal solution for the original problem, most often some or all of the variables of the solution vector are fractional. In this case the optimal solution lies in the interior of the feasible region and rounding them to integers or searching in their neighborhoods is not always an effective strategy. Nevertheless, the method at least gives a lower bound, in the case of minimization, to the optimal cost of the original integer programming problem. Relaxation to linear programming and its shortcomings are discussed in detail in the book by Papadimitriou and Steiglitz [67].

Randomized algorithms.

Randomized algorithms try to exploit non-determinism to find a solution to intractable problems in reasonable time. In contrast to standard deterministic algorithms, randomized algorithms do not provide correctness guarantees for any allowable input. However, we can make the probability of error extremely small, which makes the algorithms almost infallible while, at the same time, they run in reasonable, i.e., polynomial, time with respect to their intractable deterministic counterparts. A well-known example of such an algorithm is primality testing of very large numbers, a very important feature of cryptographic systems. Although a polynomial algorithm for this task has been found, and thus the problem is in P, the algorithm is inefficient and it is not used in practice; a randomized version is preferred instead. If the randomized algorithm has a prime number as input the answer is guaranteed to be "yes". If the input number is composite, the answer will almost certainly be negative and the probability that the answer is "yes" when it should be "no" can be

made as small as $1/2^{200}$ or smaller by adding just a polynomial amount of computing time. On the whole, randomized algorithms for difficult problems are very practical and useful when they exist, but there are not many of them around and therefore, at least for the time being, they cannot be considered a general solution to intractability. Good sources for learning more about randomized algorithms are [26, 66].

Quantum computation.

While the ways we have described until now for alleviating the intractability of hard problems are being used all the time, with some of them being rather old and well known, the approach briefly described in this section is fundamentally different and much more speculative in character. The main idea is to harness the quantum mechanical properties of matter to speed up computation; it has been in existence for some time, perhaps since the suggestions of physicists Feynman and Benioff during the eighties [39]. The main concept is to work with quantum bits, dubbed *qubits*, instead of ordinary bits for storage and computation. Qubits can represent 0, 1, or 0 and 1 at the same time thanks to the quantum mechanical property of superposition, which means than with n qubits available one can in principle process 2^n states simultaneously. It is unfortunately impossible to explain the principles of quantum computation without introducing a number of difficult physical concepts, which would be inappropriate for a book such as this one. We just observe that the quantum computation approach has the potential for turning an exponential-time algorithm into a feasible polynomial one, as has been demonstrated in theory for a few algorithms such as Shor's algorithm for the integer factorization problem, for which no polynomial-time algorithm is known in standard computation. A completely different problem is whether a quantum computer can actually be built. At the time of writing, only very small systems have been capable of operating with at most a few tens of qubits. Maintaining the coherence and reliability of such machines involves huge technical problems and exploitable quantum computers are not yet in sight although physicists and engineers have made important advances. A very readable introduction to quantum computation can be found in [64] and [18] offers a layman a glimpse into the future of this exciting field.

Metaheuristics.

All the alternative approaches that we have seen so far offer, in different ways, the possibility of limiting the impact of the intractability of hard optimization problems. Some of them give up strict global optimality in exchange for quickly obtained good enough solutions, a sacrifice that is often fully acceptable in large real-life applications. When easily available, these approaches are perfectly appropriate and should be used without hesitation if their implementation is not too difficult, especially when they are able to provide solutions of guaranteed quality. However, the main drawback of all of them is to be found in their lack of generality and, sometimes, in the difficulty of their implementation.

We now finally arrive at the family of methodologies collectively called *metaheuristics*, which are the main subject of the present book. The underlying idea is

the following: we are willing to reduce somewhat our requirements about the quality of solutions that can be found, in exchange for a flexibility in problem formulation and implementation that cannot be obtained with more specialized techniques. In the most general terms, metaheuristics are approximation algorithms that provide good or acceptable solutions within an acceptable computing time but which do not give formal guarantees about the quality of the solutions, not to speak of global optimality. Among the well-known and -established metaheuristics one might mention simulated annealing, evolutionary algorithms, ant colony method, and particle swarms, all of which will be presented in detail later in the book. The names of these methods make it clear that they are often inspired by the observation of some natural complex process that they try to harness in an abstract way to the end of solving some difficult problem. An advantage of metaheuristics is that they are flexible enough to include problem knowledge when available and they can deal with the complex objective functions that are often found in real-world applications.

Another advantage of many metaheuristics is that they can be used in conjunction with more rigorous methods through a process that we might call "hybridization" thus improving their performance when needed. Furthermore, and in contrast with many standard algorithms, most metaheuristics can be parallelized quite easily and to good effect. Because of the reasons just explained we do think that metaheuristics are, on the whole, a sensible, general, and efficient approach for solving or approximately solving difficult optimization problems. All these notions will be taken up and explained in detail starting with the next chapter. Our approach in this book is didactic and should be effective for newcomers to the field and for readers coming from other disciplines. We introduce new concepts step by step using simple examples of the workings of the different metaheuristics. Our introductory book should provide a basic and clear understanding of the mechanisms at work behind metaheuristics. A more comprehensive treatment can be found, for example, in [78], which also includes multi-objective optimization and many implementation details, and in [74] which, in addition to the basic material, also presents interesting real-life case studies. A good introduction at the level of the present text with an emphasis on evolutionary computing and implementation can be found in Luke's book [58].

2 Search Space

2.1 Search Space

In this chapter we take up again the notion of an optimization problem from the point of view of the structure of the search space associated with its instances, an idea that has been only mentioned in the previous chapter. These concepts are a prerequisite to understand all the metaheuristics that will follow in later chapters. For the sake of clarity, let us repeat how the optimization of a given function is defined. In optimization the goal is to find one or more solutions $\mathbf{x}$ to a problem which is often defined by its objective function f and possibly by the *constraints* that a solution must obey.

For example, the problem might ask for a point in the $\mathbf{x} = (x_1, x_2)$ plane that minimizes the function $f(\mathbf{x}) = x_1^2 + x_2^2$, subject to the constraint $x_1 + x_2 = 3$. We shall say that the values of $\mathbf{x}$ that satisfy the constraints are the feasible, or admissible, solutions of the problem. Among these solutions, those that maximize (or minimize) f are called *optimal solutions*. Sometimes we shall also call a solution $\mathbf{x}$ a *configuration* in the case of problems for which the word configuration is adequate.

Mathematically, the formulation of an optimization problem requires the specification of a *search space* $\mathbf{S}$ such that the elements $\mathbf{x} \in \mathbf{S}$ are the admissible solutions of the problem. To obtain the solution(s) of a maximization problem we must explore $\mathbf{S}$ and find the $\mathbf{x}$ that satisfy

$$f(\mathbf{x}) \geq f(\mathbf{y}), \quad \forall \mathbf{y} \in \mathbf{S}$$

The analogous conditions for a minimization problem are

$$f(\mathbf{x}) \leq f(\mathbf{y}), \quad \forall \mathbf{y} \in \mathbf{S}$$

These relations show that what we are looking for here are the global, as opposed to local, optima, i.e., the $\mathbf{x}$ for which $f(\mathbf{x})$ is maximal or minimal over the whole space $\mathbf{S}$.

The search space $\mathbf{S}$ and the objective function f are the two essential components in the formulation of an optimization problem. When we choose them, we define the

© Springer Nature Switzerland AG 2018

B. Chopard, M. Tomassini, *An Introduction to Metaheuristics for Optimization*,
Natural Computing Series, https://doi.org/10.1007/978-3-319-93073-2_2

problem and a coding of the admissible solutions **x**. It is important to note that, for a given problem, the coding is not unique, as we will see later, and this means that the structure of the search space may be different for the same problem depending on our choice.

Oftentimes **x** is a quantity specified by n degrees of freedom such as an n-dimensional vector of real numbers, integers, or Booleans:

$$\mathbf{x} = (x_1, x_2, \ldots, x_n)$$

The size of an optimization problem is defined as the number n of degrees of freedom it has. This value is not the same thing as the size of the corresponding search space $|\mathbf{S}|$, which is the number of elements, i.e., solutions or configurations, that it contains. The latter can be infinite or uncountable, for instance when the x_i are real numbers. In this case we shall speak of a *continuous optimization problem* and its difficulty will be characterized using the n degrees of freedom of the admissible solutions **x**. If, on the other hand, the x_i belong to a discrete countable space then we have a *combinatorial optimization problem*, several examples of which were given in Chapter 1. In this case the size of the search space is finite but it can contain a number of solutions that is exponential in the number n of degrees of freedom. To complete the main definitions let us recall that the objective function, also dubbed a cost function or a fitness function, can sometimes be called an *energy function*, by analogy with some fundamental problems in physics that call for the minimization of the associated energy.

2.2 Examples

Here we are going to present some typical classes of optimization problems that are important both in theory and in practical applications and, for each one of them, we will give the general formulation and a description of the relevant search spaces.

2.2.1 Functions in $\mathbb{R}^n$

To find the optima of a function $f : \mathbb{R}^n \to \mathbb{R}$, assumed continuous and differentiable in the domain of interest, one must look for the points at which all first-order partial derivatives vanish, that is the solutions of

$$\nabla f = 0$$

where ∇ denotes the gradient.

Then, if the second-order derivatives have the right properties we can conclude that the point in question is an extremum. Usually, numerical methods must be used to find the zeroes of derivatives. Now, in most cases where the objective function f is not a simple one, the optima found in this way are only local optima and in principle one must find all of them to locate the global one. Also, it is possible that

the global optimum is on the boundary of the domain, in case it is bounded. Clearly, for a non-convex function f, the problem of finding its optima may be difficult.

Note also that constraints of the form $g_i(\mathbf{x}) = 0$ can be added to the problem of finding the optima of f. The way to solve the problem is to use the standard Lagrange multipliers method. In short, one has to solve

$$\nabla f - \sum_i \lambda_i \nabla g_i = 0 \tag{2.1}$$

for real values λ_i to be determined. These quantities λ_i are called *Lagrange multipliers*. This method may not look very intuitive. A simple example is illustrated in $\mathbb{R}^2$, for $f(x,y) = ax^2 + by^2$ and one single constraint $g(x,y) = y - Ax - B$ (see also Fig. 2.1). The explanation is the following: the optimum we are looking for must be on the curve $g(x,y) = 0$ as this is the constraint. The optimal solution (x^*, y^*) is then on a contour line of f which is tangent to the constraint. Otherwise, by "walking" along $g = 0$, one would find arbitrarily close to (x^*, y^*), a point for which f is smaller or larger than $f(x^*, y^*)$. Since f and g are tangent for (x^*, y^*) their gradient must be co-linear. Thus, there must exist a real value λ such that $\nabla f = \lambda \nabla g$ at (x^*, y^*). Equation (2.1) and the fact that $g(x^*, y^*) = 0$, together give enough conditions to find λ and (x^*, y^*). We are not going to discuss this classical approach

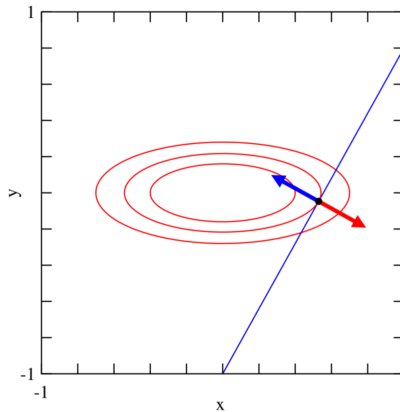


Fig. 2.1. The optimum (actually here a minimum) of $f(x,y) = ax^2 + by^2$ with constraint $g(x,y) = y - Ax - B$, with $a = 2$, $b = 5$, $A = 1.8$ and $B = -1$. The red curves are contour lines of f and the red arrow is proportional to ∇f at the given (x,y). The blue line shows the condition $g(x,y) = 0$ and the blue arrow is the gradient of g at the given point. The optimum is the black point, at which the gradients of f and g are parallel

further as it can found in many mathematical books at the undergraduate level. We remark, however, that metaheuristics are often the only practical way to solve real-world mathematical optimization problems when the objective function f is highly

multimodal, noisy, non-differentiable, or even not known in analytical form. In Chapter 8 we will mention examples of continuous, differentiable, multimodal functions whose global optimum is easily obtained with a genetic algorithm (see for instance Figure 8.4).

2.2.2 Linear Programming

Linear programming was briefly introduced in Chapter 1. Its formulation, which is repeated here for convenience, describes the following optimization problems: for given c_i, b_j, a_{ji}, find positive and real $x_1, \ldots, x_n$ such that

$$z = \sum_{i=1}^{n} c_i x_i \tag{2.2}$$

is a maximum and obeys the M constraints

$$\sum_{i=1}^{n} a_{ji} x_i \leq b_j, \qquad j = 1 \ldots M \tag{2.3}$$

The search space in linear programming is in principle $\mathbb{R}^n$ but in practice it is the set of vertices of a convex polygon, and the simplex algorithm, although theoretically it can take exponential time in the worst case, generally finds the solution in polynomial time even for large instances. This kind of problem is well known and the standard solvers are very effective. Therefore, metaheuristics are not needed at all in this case. However, you might remember from Chapter 1 that in the special case in which the x_i are $0/1$ or just integers the general problem is known to be hard.

2.2.3 NK-Landscapes

NK-landscapes have been proposed by S. Kauffman as an abstract model of genetic regulation and interaction [47]. They belong to the class of NP-hard problems as they allow one to solve the satisfaction problems. NK problems include for instance the problem of energy minimization in physical systems known as "spin glasses."

To give an intuitive and more concrete idea of what NK problems are, let us present the following example, based on an economics metaphor. We consider N persons or agents, labeled with an index i, $i = 1, \ldots, N$. Each agent i acts according to two possible strategies, denoted $x_i = 0$ or $x_i = 1$. The success of an agent depends on the strategy it chooses and the type of relation it has (competition or collaboration) with the other persons it interacts with. If we assume that each agent i depends on K other agents, we may define a function $f_i(x_i, \ldots)$ which gives the profit resulting from the chosen strategy and that of the connected agents. The total profit of the N agents is then $f = \sum f_i$. The problem of finding the strategies (the values of all x_i) that maximize the global profit is typically an NK problem.

More formally, NK problems are specified as a string $(x_1, x_2, ..., x_N)$ of N Boolean variables, each of which is linked to K other variables. The system can be visualized as a graph with N vertices, each of which has degree K.

In the framework of optimization problems, NK-landscapes generate search spaces that are harder the larger K is for a given N. The problem calls for the optimization of a fitness function

$$f(x_1, \ldots, x_N) = \sum_{i=1}^{N} f_i(x_{j_1(i)}, \ldots, x_{j_K(i)}) \tag{2.4}$$

with $x_i \in \{0, 1\}$ and given local fitnesses f_i, often built randomly, with values chosen uniformly in $[0, 1]$. It is thus an optimization problem with the N-dimensional hypercube $\{0, 1\}^N$ as a search space. Note here that we defined K as the number of arguments of the functions f_i, which is not the usual definition[1]. Here, $K = 0$ corresponds to constant functions f_i, a case which is not included in the usual definition.

As an example of an abstract NK problem, let us consider the family of problems with $K = 3$ and f defined by coupling between genes (variables) and next neighbor genes in the string $(x_1, x_2, ..., x_N)$ (coupling with randomly located genes is also customary):

$$f(x_1, \ldots, x_N) = \sum_{i=2}^{N-1} h(x_{i-1}, x_i, x_{i+1}) \tag{2.5}$$

where $h = f_i$ is a known function of three Boolean variables with values in $\mathbb{R}$ that does not depend on i.

If the objective is to maximize f, the problem difficulty depends on h. For instance, if $h(1, 1, 1)$ is the maximum then the optimal solution will clearly be $x = (1, 1, 1, \ldots, 1)$. On the other hand, if $h(1, 1, 0)$ is the maximum then the string $x = (110110110110...)$ will not necessarily be a global optimum of f; the result will depend on the values of $h(1, 0, 1)$ and $h(0, 1, 1)$.

This example shows that in NK-landscapes the search for the global optimum is made difficult by the correlations between variables induced by the couplings. In other words, the problem is not separable for $K > 1$ and we cannot search for the optimum one variable at a time. This feature of NK-landscapes allows one to tune the problem difficulty from easy with $K = 1$ to hard with $K = N$. The number of local optima increases exponentially with increasing K and the corresponding search spaces go from smooth to rugged and highly multimodal. Because of these properties, NK-landscapes are often used to test local search methods and metaheuristics.

However, these search spaces are often artificial and randomly constructed and thus they are not always representative of real-world problems. Nevertheless, we shall use them often in this book because of their didactic value.

In contrast to the case presented above with $K = 3$, in the following example $K = 1$ and the variables are fully independent of each other

$$f(x_1, \ldots, x_N) = \sum_{i=1}^{N} h(x_i) \tag{2.6}$$

[1] Usually, NK problems are defined with $f(x_1, \ldots, x_N) = \sum_{i=1}^{N} f_i(x_i, x_{j_1(i)}, \ldots, x_{j_K(i)})$, which corresponds to $K + 1$ in our definition.

where

$$h(x) = \begin{cases} 1 & \text{if } x = 1 \\ 0 & \text{otherwise} \end{cases} \quad (2.7)$$

and the global maximum is clearly $x = (11111\ldots)$. This problem, in which the objective is to maximize the number of "ones" in a binary string, is commonly called "*MaxOne*" and it will be used again in Section 2.4 of this chapter and in Chapter 8 on evolutionary algorithms. The solution is clearly obvious for a human being but it is interesting for a "blind" solver that cannot see the higher-level context.

2.2.4 Permutation Space

An important search space in combinatorial optimization is the *permutation space* of n objects. This search space is key in the TSP problem,which was introduced in Chapter 1 and will reappear in Chapters 4, 5, and 11, as well as in many other difficult combinatorial problems. In the TSP the salesman is looking for the shortest route that allows him to visit each and every town once. For example, if the salesman starts at city O and must visit towns A, B, and C, he must consider the following six possible tours

$$OABCO \quad OACBO \quad OBACO \quad OBCAO \quad OCABO \quad OCBAO$$

and choose the one that has minimum length. But here we put emphasis on the fact that the six possible cycles are exactly the permutations of the symbols A, B, and C. For a general $n \geq 1$ the number of permutations is

$$n! = n(n-1)(n-2)\ldots 1$$

which translates into $n!$ admissible solutions for the search space of the TSP with n cities. The size of such a search space grows exponentially with n, since Stirling's formula gives us

$$n! \approx \exp[n(\ln n - 1)]$$

A permutation of a set of n elements $a_i, i = 1, \ldots, n$, is a linear ordering of the elements and can be represented by a list $(a_{i_1}, a_{i_2}, \ldots, a_{i_n})$ where i_k is the index of the element at place k. For example, the expression

$$(a_2, a_4, a_1, a_3)$$

describes a permutation of four objects in which the element a_2 is at position 1, object a_4 is at position 2, object a_1 is at position 3, and object a_3 is at position 4. A shorter notation can also be used to specify the same permutation: $(2, 4, 1, 3)$, and in the general case, $(i_1, i_2, \ldots, i_n)$.

In the above representation the positions are the first-class quantities. We can however choose another representation where the permutation is defined by indicating, for each object, the position it will occupy. The permutation we so obtain is then just the inverse of the previous one. To illustrate the differences between these two

representations, let us consider again the case of a traveling salesman who must visit the following towns: Geneva, Lausanne, Bern, Zurich, and Lugano. A tour of these towns can be described according to the order in which they are visited, for instance

```
towns=(Geneva, Bern,  Zurich, Lugano, Lausanne)
```

The very same tour can also be expressed by indicating, for each town, at which step of the tour it is reached. In our example, we would write

```
step={Bern:2,Geneva:1,Lausanne:5,Lugano:4,Zurich:3}
```

The reader might have noticed that we have used here the syntax of the Python programming language to define the two representations, with the data structures that best match their meaning. In the first case, a *list* is appropriate as it reflects the ordering of the towns imposed by the tour. In the second case, a *dictionary* is used because the data structure is accessed through the town names. The above list defines a mapping from the set $\{1, 2, 3, 4, 5\}$ to the space of names, whereas the dictionary specifies a mapping from the space of names to the integers.

Obviously, one has

```
step[towns[i]]==i
```

for all integers `i`. And, for all towns `v`

```
 towns[step[v]]=v
```

since each of these two representations is the inverse of the other.

In many cases, the space of names is actually replaced by a set of numbers. The two representations are no longer easy to distinguish and care should be taken to avoid confusion when specifying transformations on a permutation. Such transformations will be discussed in a more general way in Section 2.5 because they are an essential element of metaheuristics.

We will for instance consider transformations denoted (i, j), which, by definition, exchange items i and j in a permutation. However, such a transformation (which will also be called a *move*) has different results in the two representations. In the first case one exchanges the towns visited at steps i and j of the given tour. In the second case, one exchanges the town named i with the town named j.

In our example, if we name our five towns with a number corresponding to the order in which they are specified in the data structure `step`, the transformation $(2, 3)$ amounts to exchanging `Geneva` and `Lausanne`, which then gives the following tour

```
step={Bern:2,Geneva:5,Lausanne:1,Lugano:4,Zurich:3}
```

In the other representation, one exchanges the town visited at step 2 with that visited at step 3. This gives the following new tour

```
towns=(Geneva, Zurich, Bern, Lugano, Lausanne)
```

We clearly see from this example that transformation $(2, 3)$ produces different tours, depending on which representation it is applied to. It is therefore critical not to mistakenly mix these two representations while solving a problem.

Finally, note that we can also represent a permutation of n objects with a function "successor", $s[a]$, which indicates which object follows a at the next position of the permutation. In our example, the permutation

```
towns=(Geneva, Zurich, Bern, Lugano, Lausanne)
```

can also be expressed as

```
s[Geneva]=Zurich, s[Zurich]=Bern, s[Bern]=Lugano, etc.
```

We leave it to the reader to think of how to implement a given transformation with this representation.

2.3 Metaheuristics and Heuristics

Exhaustive search and its variants is often the only way we have to find a globally optimal solution in the search space generated by a given instance of a hard problem. However, in many cases, even for discrete problems, the size is too large to enumerate all the feasible solutions. For example, the search space for Boolean problems of n variables is $\mathbf{S} = \{0, 1\}^n$ and it contains 2^n possible solutions, but for n larger than a few tens of variables the computing times quickly become excessive.

Given this situation, what we are looking for is an "intelligent" way to traverse the search space that ideally avoids sampling most uninteresting points and that allows us to find an at least satisfying solution to the problem in reasonable time. Such a methodology will of necessity be imperfect but, if we are clever enough, the hope is that the solution will be of very good quality, in any event much better that a randomly chosen one, and that the computational effort will be low with respect to other approaches.

A *heuristic* is a method of exploration that exploits some specific aspects of the problem at hand and only applies to it. For example, when solving a linear programming problem by the simplex algorithm, a heuristic is often used for choosing so-called entering and leaving variables.

A *metaheuristic* is a general exploration method, often stochastic, that applies in the same way to many different problems. Examples are tabu search, simulated annealing, ant colony, and evolutionary algorithms.

For the sake of clarity, it should be added at this point that the two terms are often used interchangeably in the literature, but the modern trend is to call the general methodology a metaheuristic and we shall follow this convention here.
Metaheuristics are characterized by the following properties:

- They don't make any hypothesis on the mathematical properties of the objective function such as continuity or derivability. The only requirement is that $f(\mathbf{x})$ can be computed for all $\mathbf{x} \in \mathbf{S}$.

- They use a few parameters to guide the exploration. The values of these parameters have an influence on the quality of the solutions found and the speed of convergence. However, the optimal values of the parameters are generally unknown and they are set either empirically, based on previous knowledge, or as a result of a learning process.
- A starting point for the search process must be specified. Usually, but not always, the initial solution is chosen randomly.
- A stopping condition must also be built into the search. This is normally based either on CPU time or number of evaluations, or when the fitness has ceased to improve for a given number of iterations.
- They are generally easy to implement and can usually be parallelized efficiently.

In all cases, a metaheuristic traverses the search space trying to combine two actions: *intensification* and *diversification*, also called *exploitation* and *exploration* respectively. In an intensification phase the search explores the neighborhood of an already promising solution in the search space. During diversification a metaheuristic tries to visit regions of the search space not already seen.

2.4 Working Principle

Most metaheuristics are based on common algorithmic principles, although this is not always made explicit in the definition of the different metaheuristics. The main ingredients are the following:

- For a given problem we define a search space $\mathbf{S}$ and an objective function $f : \mathbf{S} \to \mathbb{R}$.
- For each solution $\mathbf{x} \in \mathbf{S}$, we define a *neighborhood* $V(\mathbf{x})$ which is the set of solutions $\mathbf{y}$ that one can reach from $\mathbf{x}$ in one step and includes $\mathbf{x}$ itself. The neighborhood is usually specified in an implicit manner by a set of *transformations* T_i that generate $\mathbf{y}$ starting from $\mathbf{x}$: if $\mathbf{y} \in V(\mathbf{x})$ then there is an i for which $\mathbf{y} = T_i(\mathbf{x})$. The transformations are also called *moves*.
- We specify an *exploration operator* U such that the application of the operator to the current point (solution) $\mathbf{x}_0$ generates the next point to explore in the search trajectory. Operator U makes use of the fitness values in the neighborhood to generate the next solution to explore and is often stochastic. ,
- The exploration process

  $$\mathbf{x}_0 \to \mathbf{x}_1 \in V(\mathbf{x}_0) \to \mathbf{x}_2 \in V(\mathbf{x}_1) \to \dots$$

  continues until a suitably chosen stopping criterion is met. The result is then either the last $\mathbf{x}_n$ in the trajectory, or the $\mathbf{x}_i$ found along the trajectory that produces the best fitness value.

The efficiency of the search process depends, among other things, on the choice of $V(\mathbf{x})$ as larger neighborhoods allow one to explore more alternative solutions

but also require more time. It also depends on the coding of the solutions and on the choice of U, given that these choices determine the ability of a metaheuristic to select a promising neighbor.

Note finally that the exploration process described above, consisting of moving through the search space from neighbor to neighbor, is often referred to as *local search*. The term "local" indicates that the neighborhood is usually small with respect to the size of the search space $\mathbf{S}$. For instance the successor of the current solution may be taken from a subspace of $\mathbf{S}$ of size $\mathcal{O}(n^2)$ whereas $\mathbf{S}$ contains $\mathcal{O}(\exp(n))$ possible solutions.

The limited size of the neighborhood is obviously important if all the neighbors are evaluated to find the next exploration point. But, when the operator U builds one successor without exploring the entire neighborhood, the size of the neighborhood no longer matters. We will see many examples in this book where this happens. For instance, in Chapter 8, we will see that a *mutation* can generate any possible solution from any other one. In this case, we may no longer speak of a "local search."

To complete this discussion, let us note that backtracking and branch-and-bound methods [66] are not considered to be metaheuristics since they systematically explore the search space, which makes them expensive in computer time when applied to large problems. They work by partitioning the space into a search tree and growing solutions iteratively, eliminating many cases by determining that the pruned solutions exceed a given bound for branch-and-bound, and reverting to an earlier search point when further search is unprofitable in backtracking. In any event, these techniques share some similarities with metaheuristics as they can be applied in a generic manner to many different problems.

2.4.1 Fitness Landscapes

A *fitness landscape* or *energy landscape* is a representation of f that preserves the neighborhood topology. Basically, it is the search space with a fitness value for each configuration and a neighborhood relationship. Except in very simple cases, this abstract space becomes difficult or impossible to visualize as soon as the neighborhood is large enough. However, the "shape" of that space, in a statistical sense, will reflect the problem instance difficulty: either almost flat or very rugged landscapes will intuitively correspond to difficult problems, while single-peak or few-peaks landscapes will in general correspond to easy problems (wells will be considered, instead of peaks, for minimization). It must also be noted that the optimization of mathematical functions gives rise to "almost" continuous energy landscapes, at the level of approximation of the computer's numerical accuracy. In contrast, the search spaces generated by combinatorial optimization problem instances are of a discrete nature. Figure 2.2 illustrates a rugged energy landscape in which the configurations are represented as discrete points in a one-dimensional space. A more direct representation would require a binary hypercube with 2^{10} points, which would be very hard to draw in a two-dimensional figure.

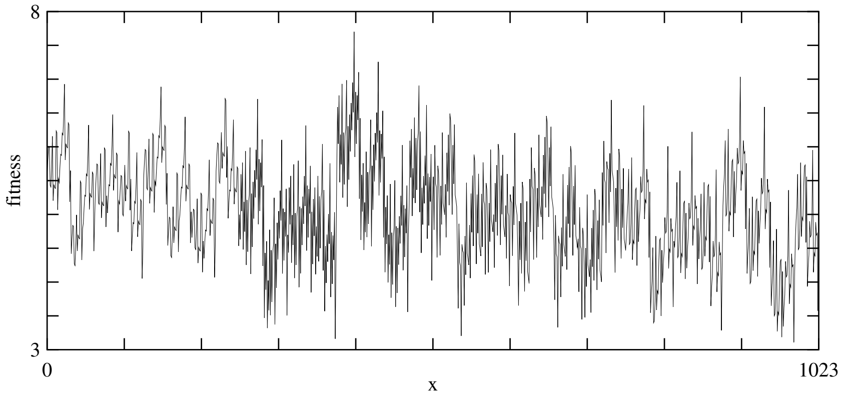


Fig. 2.2. Fitness landscape of an NK instance with $N = 10$, $K = 3$ and the local fitnesses f_i, $i = 1, \ldots, N$ randomly generated. The topology here is the one obtained by interpreting the binary bit string representation of each search point as an unsigned decimal number. Two points **x** and **y** are neighbors if $\mathbf{x} = \mathbf{y} + 1$ or $\mathbf{x} = \mathbf{y} - 1$

2.4.2 Example

The *MaxOne* problem nicely illustrates the importance of the choice of the search space and of the neighborhood. In this problem we seek to maximize

$$f(x_1, \ldots, x_n) = \sum_{i=1}^{n} x_i \tag{2.8}$$

which has the obvious solution $\mathbf{x} = (x_1, \ldots, x_n) = (1, 1, \ldots, 1)$. Actually, it is a problem belonging to the NK family with $K = 1$.

We can see the search space as the set of integers from 0 to $2^n - 1$ defined by the binary representation $(x_1, x_2, \ldots, x_n)$. Then the natural neighbors of a number **x** are the numbers $\mathbf{x} - 1$ and $\mathbf{x} + 1$.

The search space of the problem can thus conveniently be represented by the graphics of Fig. 2.3 for $n = 3$ in this example. Let's assume now that the chosen search metaheuristic is such that the U operator always selects the neighbor having the highest fitness or, in case of equality, the left neighbor. If the starting point of the search is $\mathbf{x} = 0$ we shall have the following search trajectory:

$$0 \to 1 \to 2 \to 3 \to 2 \to 3 \ \to \ldots \tag{2.9}$$

And the search process will get stuck at a local optimum.

But we could choose a search space with a different neighborhood: the n-dimensional binary hypercube, which is also in a bijective relation with the integers from 0 to $2^n - 1$. The vertices of the hypercube are the n-bit strings and the

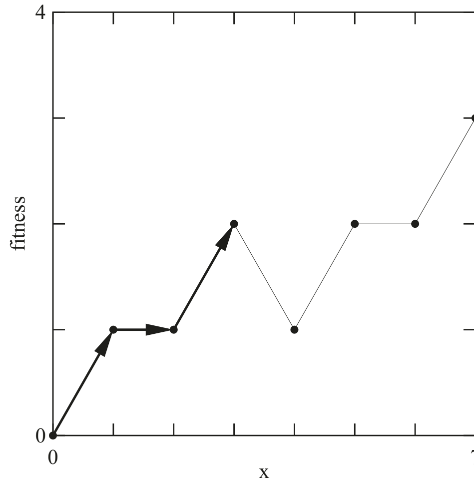


Fig. 2.3. Fitness landscape of the *MaxOne* problem for $n = 3$ and a "closest neighbors" neighborhood. Arrows show the search trajectory starting at **x** $= 0$ using a search operator that always chooses the neighbor with the highest fitness or the left neighbor in case of equality

neighbors of a vertex **x** are the bit strings **x'** that only differ by one bit from **x**. This representation of the *MaxOne* search space is depicted in Fig. 2.4.

With the same U operator as above, the search will now find the global optimum **x** $= (1, 1, 1)$, for instance along the following search trajectory:

$$000 \rightarrow 001 \rightarrow 101 \rightarrow 111 \tag{2.10}$$

which is shown in the figure by arrows. Other successful search trajectories are clearly possible. The point we want to emphasize here using this simple and unrealistic example is that the choice of a problem representation can heavily influence the efficiency of a search process.

It must be noted that problems that would be extremely simple to solve for a human may turn out to be difficult for a "blind" metaheuristic, for example, maximizing the product

$$f(x_1, \ldots, x_n) = x_1 x_2 \ldots x_n$$

where $x_i \in \{0, 1\}$. The solution is obviously $x = (1, 1, 1, \ldots, 1, 1)$ but the fitness landscape is everywhere flat except at the maximum (this situation is customarily called searching for a needle in a haystack). Thus, if no knowledge is available about the problem, the search will be essentially random.

2.5 Moves and Elementary Transformations

The chosen neighborhood, as we have seen above, is a fundamental component of a metaheuristic as it defines the set of possible successors of the current position

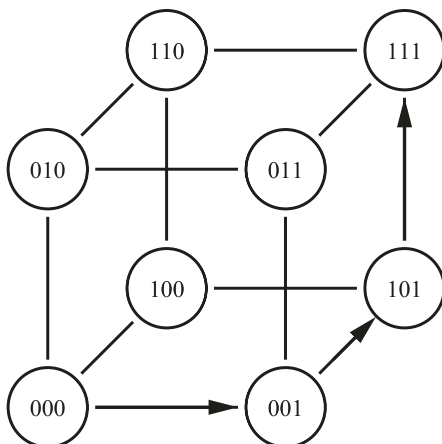


Fig. 2.4. Search space of the *MaxOne* problem when possible solutions are represented by an n-dimensional hypercube with $n = 3$ and the standard one-bit flip neighborhood. Using a move operator U that always selects the best neighbor there are several trajectories from $\mathbf{x} = (0, 0, 0)$ to the optimum $\mathbf{x} = (1, 1, 1)$, all of the same length, one of which is shown in the figure

in the exploration trajectory. A large neighborhood offers more choices and gives a more extensive vision of the fitness landscape around the current position. The drawback is that exploring larger neighborhoods to find the next configuration requires more computer time. Thus, the neighborhood size is an important parameter of a metaheuristic and, unfortunately, there is no principled way of choosing it since it strongly depends on the problem at hand.

We are now going to explain how to specify the points or configurations $\mathbf{y}$ of the search space $\mathbf{S}$ that are contained in the neighborhood $V(\mathbf{x})$ of a given configuration $\mathbf{x}$. The usual way in which this is done is to prescribe a sequence of ℓ moves m_i, $i = 1, \ldots, \ell$, which altogether will define the neighborhood of any point $\mathbf{x} \in \mathbf{S}$. Formally this can be written thus:

$$V(\mathbf{x}) = \{\mathbf{y} \in \mathbf{S} | \mathbf{y} = m_i(\mathbf{x}), i = 1, \ldots \ell\}$$

where the m_i's are given.

For example, in the discrete cartesian space $\mathbb{Z}^2$, the moves that generate the nearest neighbors of each point are the displacements along the four main directions *north, east, south, west*. Thus, for a point $\mathbf{x} = (x_1, x_2)$, the point $\mathbf{y} = \mathrm{north}(\mathbf{x})$ is $\mathbf{y} = (x_1, x_2 + 1)$, in other words, the point situated north of $\mathbf{x}$. Of course, we could also consider other moves if we wanted a larger neighborhood; for instance, we could add *north-east, south-east, south-west*, and *north-west* to the list of moves.

Let us now explain how the notion of move, or transformation, can be defined in order to generate a suitable neighborhood in the space of permutations. This space was introduced in Section 2.2.4 and it corresponds to the number of different orderings of n objects. This space is very important as it arises often in combinatorial optimization problems. To illustrate the issue, let's consider a permutation $\mathbf{x} \in \mathbf{S}$ and assume $n = 5$. A suitable neighborhood can be obtained, for example, by the transposition of two elements. Denoting by (i, j), $j > i$ the transposition of the element at position i with the one at position j, the ten moves m that generate the neighbors of a permutation of five objects are

$$m \in \{(1,2),(1,3),(1,4),(1,5),(2,3),(2,4),(2,5),(3,4),(3,5),(4,5)\}$$

For instance, the permutation $(a_3, a_2, a_4, a_5, a_1)$ has $(a_2, a_3, a_4, a_5, a_1)$ as a neighbor, which is obtained by applying the move $(1, 2)$, i.e., transposing the first two elements. Such a choice for the neighborhood of a given permutation ensures that an arbitrary point $\mathbf{y}$ in the search space can be reached from any other configuration $\mathbf{x}$ in a finite number of moves. Actually, it can be proved that any permutation can be decomposed into a sequence of transpositions of pairs of elements.

For arbitrary n, transpositions are of the form (i, j) with $i = 1, 2, \ldots, n-1$ and $j = i+1, i+2, \ldots, n$. Their number is $(n-1) + (n-2) + \ldots + 1 = n(n-1)/2$ and thus the neighborhood of a point in $\mathbf{S}$ contains $\mathcal{O}(n^2)$ elements, which can be a large number if the exploration operator U must evaluate all those neighbors to find the best one. Another way of generating a neighborhood in the permutation space is described in Section 3.6.

The choice of the moves may have an impact on the quality of the search. It is likely that some moves are better adapted to a given problem than others. We will see in Chapter 4 that this actually happens with the TSP problem.

To illustrate this point, let us consider the permutation

$$p = (p_0, p_1, \ldots, p_9) = (v_0, v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8, v_9)$$

that corresponds to a tour of 10 cities v_i, as shown in Figure 2.5. Let us now apply two different types of moves to p. First, we consider the transposition introduced previously, namely the exchange of, for instance, the towns visited at steps 2 and 6. The result of this move is also illustrated in Figure 2.5 and corresponds to $p_2 = v_6$ and $p_6 = v_2$. It should be noted that the numbers indicated in the figure represent the "names" of the towns and that the order of the visits is given by the arrows shown on the links connecting successive towns. Initially, the order of the visits is given by the names of the towns.

We can observe that the move $(2, 6)$ as just performed has an important impact on the tour. Four links of the original path have been removed, and four new links have been created. This modification can lead to a shorter tour (as is the case in our example), but it substantially modifies the topology of the tour.

We can however consider other moves that modify the path in a more progressive way. The moves, often referred to as *2-Opt*, only remove two sections of the path and create two new ones. Figure 2.6 shows such a move for the situation presented

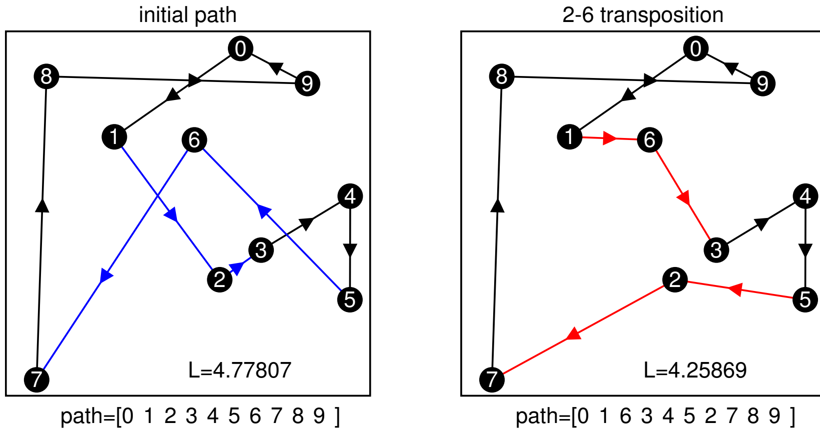


Fig. 2.5. Example of a TSP tour of 10 cities v_ℓ labeled with numbers ℓ between 0 and 9. By applying the transposition move $(2, 6)$, four sections of the original tour disappear (blue links) and are replaced by four new links (in red)

in Figure 2.5. This move swaps two towns, but also reverses the travel order between them. There is less impact on the topology of the path, but more modifications to the permutation vector $(p_0, \ldots, p_9)$ coding for the tour. Here we use the notation (i, j)-2-Opt to indicate the move that replaces edges $i \to (i+1)$ and $j \to (j+1)$ of the tour with edges $i \to j$ and $(i+1) \to (j+1)$.

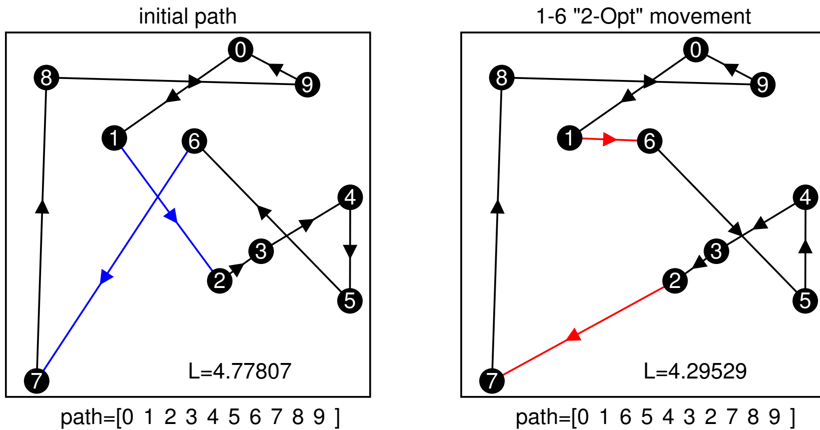


Fig. 2.6. Example of a TSP tour of 10 cities numbered from 0 to 9 on which a 2-Opt move is applied. The edges that are removed are shown in blue and the new ones in red

The terminology 2-Opt comes from an algorithm proposed by G.A. Croes in 1958, to solve instances of the TSP problem. When a path contains the crossing of an edge $v_{p_i} \to v_{p_{i+1}}$ with another edge $v_{p_j} \to v_{p_{j+1}}$, as shown in blue color in Figure 2.7, the length of the tour can easily be reduced by replacing these two edges with the red ones, namely $v_{p_i} \to v_{p_j}$ and $v_{p_{i+1}} \to v_{p_{j+1}}$. As noticed previously, this also requires cities v_{p_j} to $v_{p_{i+1}}$ to be traversed in the reverse order.

This transformation guarantees a shorter path due to the famous *triangle inequality*, which here reads as

$$|v_{p_i} - v_{p_{i+1}}| + |v_{p_j} - v_{p_{j+1}}| \geq |v_{p_i} - v_{p_j}| + |v_{p_{i+1}} - v_{p_{j+1}}| \tag{2.11}$$

A path is called *2-Opt* if it is optimal under any such transformation that replaces edges $v_{p_i} \to v_{p_{i+1}}$ and $v_{p_j} \to v_{p_{j+1}}$ by edges $v_{p_i} \to v_{p_j}$ and $v_{p_{i+1}} \to v_{p_{j+1}}$. Such an optimization can be performed in time $\mathcal{O}(n^2)$ with an algorithm that considers all pairs (i, j) of an n-town TSP tour and applies the (i, j)-2-Opt move whenever inequality (2.11) is true.

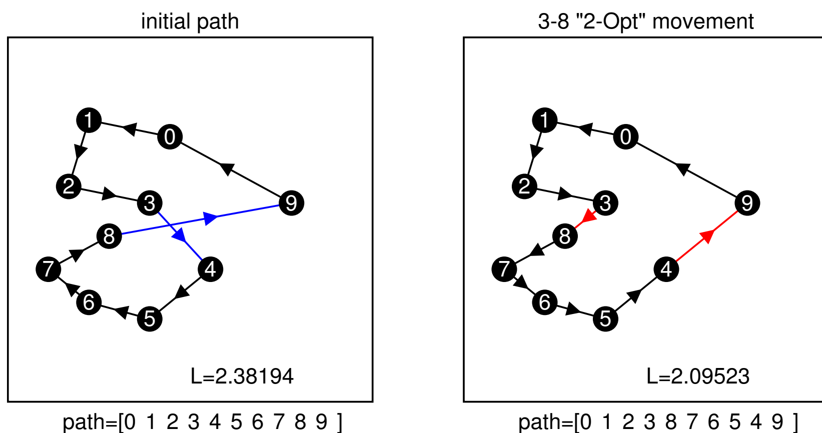


Fig. 2.7. Path with a crossing. Its length can be reduced with a 2-Opt move: blue edges are removed and replaced by the red ones

Note that k-Opt algorithms can also be considered. They ensure that a path is optimal under the destruction of any set of k links and the creation of k new links, with the constraint that the new tour is still a TSP tour. By extension, it is now usual to designate one such transformation by the word "k-Opt", even though one single move does not make the path k-optimal.

2.6 Population Metaheuristics

Instead of just considering a single point of the search space at a time, we could take a whole population of solutions evolving in each time step. Figure 2.8 schematically

shows the search trajectory in the case of a single individual metaheuristic (left), and in the case of a population metaheuristic (right). The basic principles remain the same: in the population case the search space becomes the cartesian product $\mathbf{S}^N = \mathbf{S} \times \mathbf{S} \times \ldots \times \mathbf{S}$ where N is the number of individuals in the population and $\mathbf{S}$ the search space for a single individual. The neighborhood of a population P is the set of populations P_i that can be built from the the individuals of P by using transformations to be defined. However, in population-based metaheuristics one does not generate all the possible neighboring populations, as this would be too time-consuming. Instead, only one successor population is generated according to a stochastic process. The idea is that using a set of solutions allows one to exploit correlations and synergies among the population members, one example being the recombination of successful features from several candidate solutions. We shall see examples of population-based metaheuristics in Chapters 8, 5, and 6.

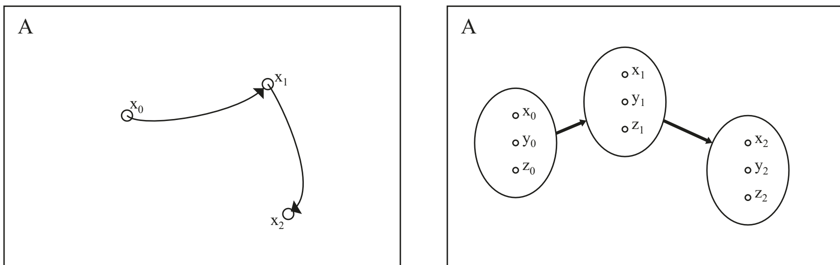


Fig. 2.8. Three stages of a single point metaheuristic are illustrated in the left image. The right image schematically shows three stages of a population-based maetaheuristic. In this example population P_0 contains the three individuals of $\mathbf{S}$ called $\mathbf{x}_0$, $\mathbf{y}_0$, and $\mathbf{z}_0$

2.7 Fundamental Search Methods

In the following we are going to give examples of the search operator U for some widely used elementary metaheuristics. These simple search methods are important because they form the basis for more elaborate operators that will be presented in later chapters. In what follows we assume that the search space and the neighborhood are given.

2.7.1 Random Search and Random Walk

The simplest search method is *random search* where the next point to test in the search is chosen uniformly at random in the whole search space $\mathbf{S}$. Usually, one keeps the solution having the best fitness after having performed a prescribed number of steps. Of course this kind of search offers a very low probability of falling on the

global optimum, or at least on a high-fitness solution, given that the probability of such an avent is equal to $1/|\mathbf{S}|$, a very small one indeed if $|\mathbf{S}|$ is large.

If we restrict the random search to the neighborhood of the current solution then we have what is called a *random walk* in search space. Analogously to unrestricted random search, this exploration method has a very low probability of finding the optimum in reasonable time. An example of random walk search is found in Chapter 3, in Figure 3.2. We take up random walks again in Chapter 7.

2.7.2 Iterative Improvement: Best, First, and Random

A more intelligent local search approach is given by the metaheuristic called *iterative best improvement* or hill climbing. In this case the U operator always chooses the best fitness neighbor of the current solution as the next point in the search trajectory. Clearly, this search method will get stuck at a local optimum unless the search space is unimodal with a single maximum or minimum. If the search space has a plateau of solutions with equal fitness values then the search will also get stuck unless we slightly change the acceptance rule and admit new solutions with the same objective function value. This method is often used with a *restart* device, i.e., if the search gets stuck then it is restarted from another randomly chosen point. While this can work in some cases, we shall see that better metaheuristics are available for overcoming getting stuck at a local optimum.

A variation of iterative best improvement is *iterative first improvement*, which checks the neighborhood of the current solution in a given order and returns the first point that improves on the current point's fitness. This technique is less greedy than best improvement and evaluates fewer points on average, but it also gets stuck at a local optimum.

2.7.3 Probabilistic Hill Climber

In order to prevent the search becoming stuck at a local optimum, one can consider probabilistic moves. This is the case of *randomized iterative improvements* in which improving moves are normally performed as above, but worsening moves can also be accepted with some prescribed probability. Since worsening moves are allowed, the search can escape a local optimum. The terminating condition is then based on a predefined number of search steps. The ability to accept moves that deteriorate the fitness is an important element of several metaheuristics, as discussed in Chapters 3 and 4.

The *probabilistic hill climber* is an example of a simple metaheuristic using randomized moves. It achieves a compromise between best improvement and first improvement metaheuristics. The next search point $\mathbf{x}' \in V(\mathbf{x})$ is selected with a probability $p(\mathbf{x}')$ that is proportional to the fitness $f(\mathbf{x}')$: thus, the higher the fitness of the candidate solution, the higher the probability that this solution will be chosen as the next search point. If $f(\mathbf{x})$ is positive for all $\mathbf{x}$ in the search space then, for a maximization problem, we can define $p(\mathbf{x})$ as follows:

$$p(\mathbf{x}') = f(\mathbf{x}')/\sum_{\mathbf{y}\in V(\mathbf{x})} f(\mathbf{y})$$

In this way, the best candidate solutions around $\mathbf{x}$ will be chosen with high probability but it will also be possible to escape from a local optimum with positive probability. Section 2.8 offers a description of how to implement such a probabilistic choice. Note also that this fitness-proportionate selection is popular in evolutionary algorithms for replacing a population with a new one (see Chapter 8).

2.7.4 Iterated Local Search

Iterated local search, ILS for short, is a simple but efficient metaheuristic that is useful for avoiding premature convergence towards a local optimum. The intuitive idea behind ILS is straightforward. To start with, we assume the existence of a local search metaheuristic `localSearch(x)` which, for any candidate solution $\mathbf{x}$ in $\mathbf{S}$, returns a local optimum $\mathbf{x}'$. The set of points $\mathbf{x}$ that share this property is called the *attraction basin* of $\mathbf{x}'$. Conceptually, one could replace the $\mathbf{S}$ space by a smaller one called $\mathbf{S}'$ that only contains the local optima configurations $\mathbf{x}'$. The search of $\mathbf{S}'$ for a global optimum would thus be facilitated. This vision requires that the local optima of $\mathbf{S}$ are well defined and thus `localSearch(x)` must be deterministic, always returning the same optimum when applied to a given configuration $\mathbf{x}'$. Ideally, we should have a neighborhood topology on $\mathbf{S}'$ that should allow us to reapply the local search on the restricted search space. At least formally, such a neighborhood relationship does exist since two optima $\mathbf{x}'$ and $\mathbf{y}'$ can be considered neighbors if their corresponding basins of attraction are neighbors in $\mathbf{S}$. However, determining the neighborhood $V(\mathbf{x}')$ in $\mathbf{S}'$ is often impossible in practice. We are thus led to a less formal implementation which can nevertheless give good results and does not require `localSearch(x)` to be deterministic. The main idea is to perform "basin hopping", i.e., to go from a local optimum $\mathbf{x}'$ to another $\mathbf{y}'$ passing through an intermediate configuration $\mathbf{y} \in \mathbf{S}$ obtained by a random perturbation of $\mathbf{x}' \in \mathbf{S}'$. The local optimum $\mathbf{y}'$ obtained by applying `localSearch` to $\mathbf{y}$ is considered to be a neighbor of $\mathbf{x}'$. The following pseudo-code describes the implementation of the ILS algorithm:

```
x=initialCondition(S)
x'=localSearch(x)
while(not end condition):
  y=perturbation(x')
  y'=localSearch(y)
  x'=acceptance(x',y')
```

Figure 2.9 schematically illustrates the principles of the method. The elements of $\mathbf{S}'$ are the local maxima of the fitness landscape of the whole search space $\mathbf{S}$, which is the blue curve. The local maximum $\mathbf{x}'$ is the current solution obtained from an initial random solution after a `localSearch()` phase. The perturbation operation `perturbation(x')` is symbolized by the red arrow and produces a

new configuration $\mathbf{y}$. Applying again the procedure `localSearch()` to $\mathbf{y}$ we get a solution $\mathbf{y}' \in \mathbf{S}'$. Whether or not to accept the new solution $\mathbf{y}'$ is set by `acceptance(x',y')`. If the new solution is rejected the search starts again from $\mathbf{x}'$. The process is iterated until a termination condition is satisfied. The difficult

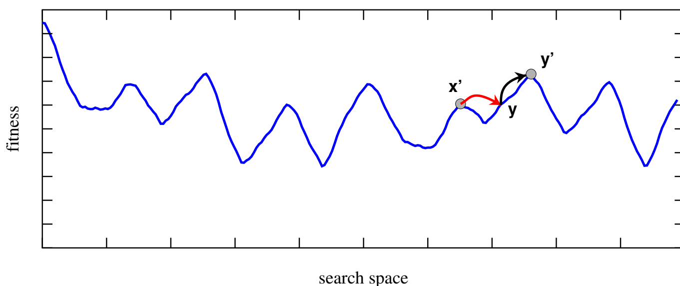


Fig. 2.9. Schematic illustration of iterated local search (ILS)

point in this metaheuristic is the choice of the perturbation procedure. Too strong a perturbation will lead the search away from a potentially interesting search region, reducing it to a hill climbing with random restarts, while too weak a perturbation will be unable to kick the search out of the original basin, causing a cycle if the local search and the perturbation are deterministic. The interested reader will find more details on ILS in [57].

2.7.5 Variable Neighborhood Search

This search strategy utilizes a family of neighborhoods $V_1(\mathbf{x})$, $V_2(\mathbf{x}), \ldots, V_k(\mathbf{x})$, usually ordered by increasing size, for each $\mathbf{x} \in \mathbf{S}$. Let's consider a metaheuristic with a search operator U that returns the best neighbor $\mathbf{y}$ of $\mathbf{x}$ in $V_1(\mathbf{x})$. If the fitness of the new solution $\mathbf{y}$ does not improve on the fitness of $\mathbf{x}$ then U is applied on the following neighborhood $V_2(\mathbf{x})$ and so on, until we find a neighborhood V_ℓ within which a solution is found that improves the fitness. This point then becomes the current solution and the process is iterated until a stop condition is met. If no improvement has been found on any of the k neighborhoods the search stops and returns $\mathbf{x}$. The advantage of using a hierarchy of neighborhoods is that one hopes that most of the time using V_1 will suffice. The search explores more points in the solution space only when needed.

2.7.6 Fitness-Smoothing Methods

Fitness smoothing is a strategy that consists of modifying the original fitness function in order to reduce its ruggedness, thus making it easier to explore with a given

metaheuristic. While the fitness is modified for the sake of limiting the number of unwanted local optima, the search space itself remains the same. For instance, consider the following example in which the average objective function is defined as

$$\bar{f} = \frac{1}{|\mathbf{S}|} \sum_{\mathbf{x} \in \mathbf{S}} f(\mathbf{x}) \tag{2.12}$$

A smoothed fitness f_λ with a coefficient λ can be written as

$$f_\lambda(\mathbf{x}) = \bar{f} + \lambda(f(\mathbf{x}) - \bar{f}) \tag{2.13}$$

When $\lambda = 0$ the landscape is flat, while we recover the original $f(\mathbf{x})$ with $\lambda = 1$. This is illustrated in Figure 2.10 in which the fitness of an NK problem has been smoothed with $\lambda = 0.4$ (red curve) and $\lambda = 0.8$ (blue curve). In addition, a local classical smoothing is applied to the $\lambda = 0.4$ and $\lambda = 0.8$ curves to eliminate small fitness variations.

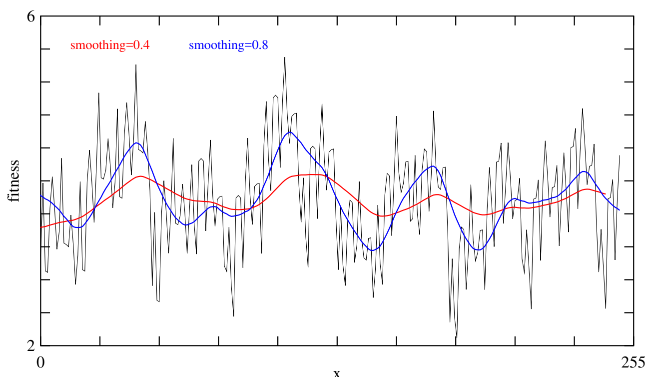


Fig. 2.10. Fitness smoothing in an NK problem with $N = 8$ and $K = 3$. The non-smoothed landscape is represented in black. The smoothed landscapes with a coefficient of 0.4 and 0.8 are shown in red and blue respectively

Clearly the smoothed problem does not have exactly the same solution as the original one but the strategy is to solve a series of problems with landscapes that become harder and harder. In fact, in the figure one can see that the maximum of the red curve is a good starting point for finding the maximum of the blue curve, which, in turn, having bracketed a good search region, should make it easier to find the maximum of the original function.

The approach can be abstractly described by the following pseudo-code:

```
x=initialCondition
for lambda=0 to 1:
    x=optimal(f,lambda,x)
```

The problem with the above description, which was only intended to illustrate the principles of the method, is that computing the mean fitness $\bar{f}$ is tantamount to enumerating all the points in $\mathbf{S}$, which would give us the answer directly but is undoable for problems generating an exponential-size search space. However, it is sometimes possible to just smooth a component of the fitness function, rather than the fitness itself. For example, in the traveling salesman problem one might smooth the distances d_{ij} between cities i and j

$$d_{ij}(\lambda) = \bar{d} + \lambda(d_{ij} - \bar{d}) \tag{2.14}$$

where the average distance $\bar{d}$ is computed as

$$\bar{d} = \frac{1}{n(n-1)} \sum_{i \neq j} d_{ij} \tag{2.15}$$

This computation has complexity $\mathcal{O}(n^2)$, much lower than $\mathcal{O}(n!)$, which is the original problem instance complexity. With this choice for $d_{ij}(\lambda)$ we have that all cycles through the n cities are of identical length if we take $\lambda = 0$, and we could take this approximation as the first step in our increasingly rugged sequence of landscapes that are obtained when λ progressively tends to 1, i.e., to the original fitness landscape.

2.7.7 Method with Noise

This is another approach that, like the smoothing method, perturbs the original landscape, this time by adding random noise to the objective function, with the hope of avoiding local optima that would cause premature convergence of the search. In practice, one starts with a given perturbation noise which is iteratively reduced towards the original, unperturbed fitness function. One way of doing this is to add a uniformly distributed random component in the interval $[-r, r]$ to the fitness function f:

$$f_r(\mathbf{x}) = f(\mathbf{x}) + \xi \tag{2.16}$$

The problem is first solved for $f_{r_{max}}$ followed by an iterative decrease of r towards 0. The process is described by the following pseudo-code:

```
x=initialCondition

for r=r_max to 0:
    x=optimal(f,r,x)
```

Reference [78] provides further information on this methodology. In Chapter 4 we shall describe a popular metaheuristic called *simulated annealing* which successfully exploits a similar idea.

2.8 Sampling Non-uniform Probabilities and Random Permutations

Most metaheuristics have an important stochastic component and, because of that, they must generate and use random numbers of various kinds to generate initial conditions or to make probabilistic choices during the search. Although the usual scientific computer programming environments do provide many tools for generating such values according to several probability distributions, it is still important to understand the main ideas if one wants to implement a given metaheuristic. In this section we consider two cases that arise often: generating arbitrarily distributed random numbers, and generating random permutations.

2.8.1 Non-uniform Probability Sampling

From a programming point of view, it is important to understand how one can select a random event (for instance, a neighbor of the current solution) according to a given probability distribution function. This problem will constantly reappear throughout the book, for example in Chapter 5 to route artificial ants through a graph, or in Chapter 8 to select solutions proportionally to their fitness.

All computer languages offer functions for generating pseudo-random "real" numbers uniformly distributed in the interval $[0, 1[$. Starting from such equiprobable random numbers, one can turn them into non-uniform ones according to an arbitrary probability distribution by one of several different techniques, as illustrated below.

The simplest, but frequently needed case consists of executing an action `A` with probability p or rejecting it with probability $1 - p$. This is easily done according to the following pseudo-code:

```
r=uniform(0,1)
if r<p: execute A
```

This works because the probability that r is less than p is just p, if r is drawn uniformly between 0 and 1.

When there are more than two choices the algorithm becomes a little more complicated. Figure 2.11 visually shows how one can choose event i whose probability is p_i given that $\sum_i p_i = 1$. In the example there are six possibilities, each with a different probability. Once r has been drawn, the chosen event will be the one corresponding to the largest i for which $\sum_{j=1}^{i-1} p_j < r$. In the limit case for which r is less than p_1 the solution is $i = 1$.

The algorithm can be described by the following pseudo-code:

```
r=uniform(0,1)
s=0
i=1
for p in [p1,p2,...,p6]:
    s+=p
    if r<s: return i
    i+=1
```

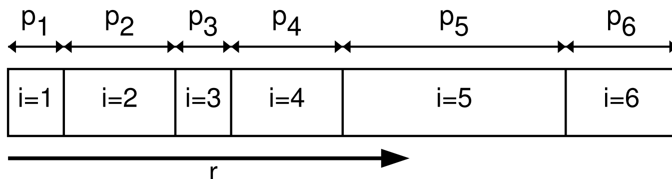


Fig. 2.11. Example of the selection of an event among six possible events, each with given probabilities. In the figure, event $i = 5$ would be chosen

Sampling a discrete distribution is an important problem in many applications of computational science. The code above, although simple, is not the best one from the efficiency point of view. The interested reader is referred to the web site [48] for optimized versions of the algorithm.

In the case of continuous random variables the above method can be reformulated in a similar way. Let us assume that we want to generate a random number s in a given interval $[a, b[$, and distributed according to the probability density $p_s(s)$. Such a value s can be obtained from a random number r drawn from a uniform distribution over $[0, 1[$, as follows

$$r = \int_a^s p_s(t)dt \tag{2.17}$$

To understand this relation, we notice that it implies that the probability that s is found between s_0 and $s_1 > s_0$ is the same as the probability that r is between $r(s_0)$ and $r(s1)$. The latter is obviously $r(s_1) - r(s_0)$ since r is uniformly distributed between 0 and 1. But

$$r(s_1) - r(s_0) = \int_a^{s_1} p_s(t)dt - \int_a^{s_0} p_s(t)dt = \int_{s_0}^{s_1} p_s(t)dt \tag{2.18}$$

The last term of the above equation is the very definition of the probability that s is between s_0 and s_1, thus showing the correctness of (2.17).

In Chapter 7, we will discuss a random process called Lévy flight. It is characterized by a probability distribution

$$p_s(s) = (\alpha - 1)s^{-\alpha} \qquad \alpha > 1, \qquad s \in [1, \infty[\tag{2.19}$$

From relation (2.17) one can easily simulate such a process using uniformly distributed random numbers r. For this distribution p_s, one can compute the integral analytically and obtain

$$r = (\alpha - 1) \int_1^s dt\, t^{-\alpha} = 1 - s^{1-\alpha} \tag{2.20}$$

Therefore s can be computed from r as

$$s = (1 - r)^{\frac{1}{1-\alpha}} \tag{2.21}$$

For other distributions p_s, for instance Gaussian distributions, one cannot compute the integral (2.17) analytically. However, there exist numerous algorithms that can be used to produce Gaussian numbers from a uniform distribution. We refer the interested reader to the specialized literature. But we remind the reader that many software libraries, such as the module `random` in Python, contain many functions for generating random numbers distributed according to discrete and continuous probability distributions such as `random.randint()`, `random.normalvariate()`, `random.expovariate()`, `random.paretovariate()`, as well as several others. For C++, the library `boost` offers the same possibilities. At this stage, it is good to remember that metaheuristics perform simple calculations, but many are needed. Therefore the choice of the programming language should be adapted to the size of the problem to be solved. Clearly Python decreases the programming effort, but its performance may not match that of C++.

To finish this overview of how probability distributions are modified due to a change of variable, let us mention the following result. If a random variable x is distributed according to $f(x)$, and if we define another random variable y through the relation $y = g(x)$, then y is distributed according to

$$p(y) = f(h(y))h'(y) \tag{2.22}$$

where h is the inverse function of g, namely $h(y) = x$, and h' is the derivative of h. We leave it to the reader to check that relation (2.21) gives that s is indeed distributed as $(\alpha-1)s^{-\alpha}$ when $f(r) = 1$. For further discussion on how to sample non-uniform probability distributions we refer the reader to Knuth's book [51].

2.8.2 Random Permutations

We already remarked in this chapter (see Section 2.2.4) that many combinatorial optimization problems can be formulated in a permutation search space. Thus, because of their importance, it is useful to know how to generate random permutations of n elements, for example the numbers from 0 to $n-1$. Random permutations of this kind will be used in Chapters 3 and 4.

Intuitively, any permutation can be built by imagining that the n objects are in a bag and that we draw them randomly and with uniform probability one by one. From an algorithmic point of view, a random permutation can be efficiently generated by the method of Fisher and Yates, also called *KnutShuffle* [51]. With this method, a random permutation among the possible $n!$ is produced in time $\mathcal{O}(n)$ and all permutations are equiprobable.

The method can be understood by using induction: we assume that it is correct for a problem of size i and then we show that it is also correct for $i+1$. Suppose that the algorithm has been applied to the first i elements of the list that we want to shuffle randomly. This means that, at this stage, the algorithm has generated a sequence of length i corresponding to one possible permutation of the first i objects. Now we add the $i+1$,th object at the end of the list and we randomly exchange it with any element of the list, including the element $i+1$ itself. There are thus $i+1$ possible

results, all different and equiprobable. Since we assumed that at stage i we were able to generate the $i!$ possible permutations all with the same probability, we now see that at stage $i+1$ for each of these we get $i+1$ additional possibilities, which shows that in the end we can create $(i+1)!$ equiprobable random permutations. Extending this reasoning up to $i = n$, a random permutation among the $n!$ possible will be obtained with n successive random exchanges, giving rise to time complexity $\mathcal{O}(n)$.

To illustrate, we give an implementation of the algorithm using the Python language in the following program excerpt, where the elements to be randomly shuffled are contained in the list `listOfObjects`.

```
import random

listOfObjects=["a", "b", "c", "d", "e", "f"]
n=len(listOfObjects)

permutation=listOfObjects[:]

for i in range(n):
    j=random.randint(0,i)
    permutation[i]=permutation[j]
    permutation[j]=listOfObjects[i]
```

where the function `randint(0,1)` returns a random integer in the closed interval $[0, 1]$. Furthermore, since the modifications in the loop over i only affect elements with an index less than or equal to i, we do not need to define a `permutation` array. The permutation can be performed in-place in the list `listOfObjects` by saving the current value of `listOfObjects` in a temporary variable `tmp`:

```
for i in range(n):
    j=random.randint(0,i)
    tmp=listOfObjects[i]
    listOfObjects[i]=listOfObjects[j]
    listOfObjects[j]=tmp
```

In fact, there is a built-in function `random.shuffle(permutation)` in the Python package `random` that accomplishes exactly this task using the *Knuth Shuffle* algorithm. For an implementation using another language, it is important to be able to generate good-quality uniformly distributed random integers j between 0 and i for arbitrary i. For instance, it would be wrong to draw random numbers between 0 and M for a given M and to calculate their modulo $i+1$. If M is not a multiple of $i+1$, the numbers between 0 and $m-1$, where $m = M \mod (i+1)$, would be overrepresented.

The simplest way is probably to consider a standard good pseudo-random number generator that returns a real r in $[0, 1[$. One can now divide the interval $[0, 1[$ into $i+1$ equal portions and choose j as the index of the sub-interval to which

r belongs. Since all the intervals have the same size, it is immediate to see that `j=int(r*(i+1))`, where `int` returns the integer part of a number.

3

Tabu Search

3.1 Basic Principles

Tabu search was proposed in 1986 by F. Glover [36]. This metaheuristic gives good results on combinatorial optimization problems such as quadratic assignment. The principles of tabu search are simple and are based on the methodology discussed in Section 2.4:

- The problem search space is explored by going from neighbor to neighbor, $\mathbf{x}_n \rightarrow \mathbf{x}_{n+1} \in V(\mathbf{x}_n)$, starting from an arbitrary admissible initial solution $\mathbf{x}_0$.
- The search operator selects $\mathbf{x}_{n+1} \in V(\mathbf{x}_n)$ as the *non-tabu* $\mathbf{x}_{n+1}$ that locally optimizes fitness and this choice is independent of the fitness of the current solution $f(\mathbf{x}_n)$. If there is more than one candidate solution respecting the constraints and having the same fitness, one is chosen randomly.
- From the previous point, the specificity of tabu search is the existence of forbidden, or tabu, solutions. The idea is to prevent the search from going back to previously explored configurations $\mathbf{x}$. The tabu configurations are thus those that have already been sampled. Actually, the situation is a bit more subtle, as we will see in the sequel.
- The "tabu" attribute is not a permanent one. There is a short-term memory that causes tabu points to become again non-tabu after a certain number of iterations. It is also possible to introduce a long-term memory that maintains statistical information on all the solutions that have been visited during the whole search. The long-term memory is useful to avoid systematic biases in the way the solution space is searched.
- The tabu method implements a list to enumerate the forbidden points and movements. This memory is continuously updated during the search by suppressing old forbidden configurations and by adding new ones.

In summary, the tabu metaheuristic can be expressed by the flow diagram of Figure 3.1. We remark that the simplicity of the method is only apparent, as some heuristics are needed to build and maintain the tabu list. We will come back to these points in Section 3.4.

© Springer Nature Switzerland AG 2018

B. Chopard, M. Tomassini, *An Introduction to Metaheuristics for Optimization*,
Natural Computing Series, https://doi.org/10.1007/978-3-319-93073-2_3

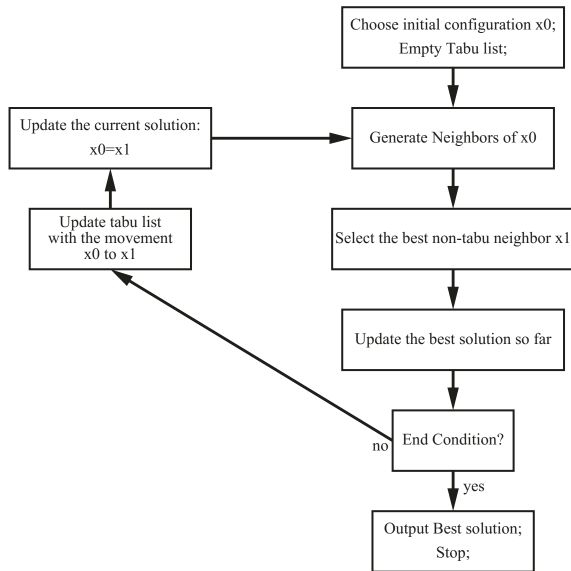


Fig. 3.1. The flow diagram of tabu search

3.2 A Simple Example

As a simple illustration of how tabu search works, let us consider the example of the objective function $f(x, y)$ shown in the left part of Figure 3.2. The search space is a subspace of $\mathbb{Z}^2$,

$$\mathbf{S} = \{0, 1, \dots, 19\} \times \{0, 1, \dots, 19\}$$

that is, a domain of size 20×20.

The goal is to find the global maximum of f, which is located at $P_1 = (10, 10)$. However, there is a local maximum at $P_2 = (6, 6)$, which should make finding the global one slightly more difficult.

The neighborhood $V(x, y)$ of a point (x, y) will be defined as being formed by the four closest points in the grid

$$V(x, y) = \{(x-1, y), (x+1, y), (x, y-1), (x, y+1)\} \tag{3.1}$$

Clearly, for points belonging to the borders and corners of the grid the neighborhood will be smaller since some neighbors are missing in these cases. For instance, $V_{(0,0)} = \{(1, 0), (0, 1)\}$.

In the right image of Figure 3.2 a random walk search of 50 iterations is shown starting from the initial solution $(7, 8)$ (in blue on the image). By chance, the exploration finds the second maximum at $P_2 = (6, 6)$. However, the region explored

during the 50 steps contains only 29 distinct points, since the random walk often re-samples points that have already been seen. We remark that for this toy problem an exhaustive search would take $20 \times 20 = 400$ iterations.

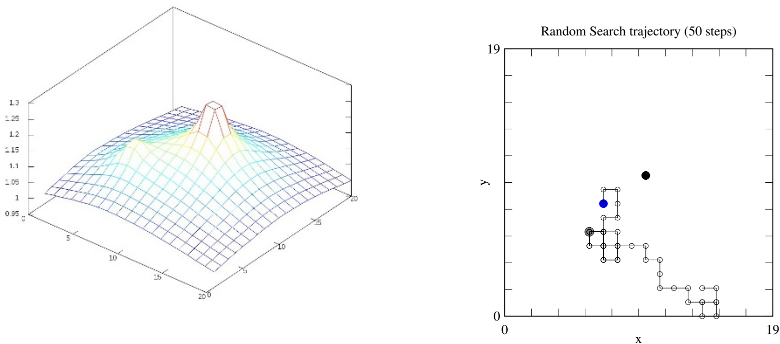


Fig. 3.2. Left image: an example of a discrete fitness function defined on a two-dimensional space of size 20×20. The global maximum is at $P_1 = (10, 10)$, and there is a second local maximum at $P_2 = (6, 6)$. Right image: example of a 50-step random walk search trajectory following the neighborhood $V(x, y)$ defined in equation (3.1). The black point is the global maximum P_1; the grey point is the second-highest maximum P_2. The blue point represents the initial configuration of the search

In contrast with random walk, in tabu search configurations that have already been visited are forbidden. In practice, the method saves the last M points visited and declares them tabu. In each iteration a new point is added to the tabu list and, if the list has reached the maximum size M, the oldest entry is suppressed to make space for the newest point. This form of memory is called *short-term* because the information on tabu points evolves during time and is not conserved for the whole search duration.

Figure 3.3 illustrates the exploration that results from using tabu search starting from the initial solution $(0, 0)$ using a tabu list of size $M = 4$ (left) and $M = 10$ (right). In both cases, the search quickly reaches the second maximum P_2 since tabu behaves here as a strict hill climber that always chooses the best configuration in the neighborhood of the current one. However, once there, the search trajectory is forced to visit points of lower fitness since the local maximum has already been visited. As a consequence, the trajectory will remain around the local optimum, trying not to loose "height". With a memory M of size four, after four iterations around P_2, the trajectory will visit the latter again, since this configuration is no longer in the tabu list. However, the search will not be able to extract itself from the basin of attraction of P_2 and will never reach the global optimum. On the other hand, with a memory size $M = 10$, the trajectory will be forced to get away from P_2 and will be able to reach the basin of attraction of P_1. From there, it will be easy to finally find the

global optimum. If the iterations continue, the trajectory will perform walks of length 10 around P_1. With respect to random walk, we remark that the number of unique points visited with tabu search is larger. When the search stops, the algorithm will return point P_1 since it is the best solution found during the search.

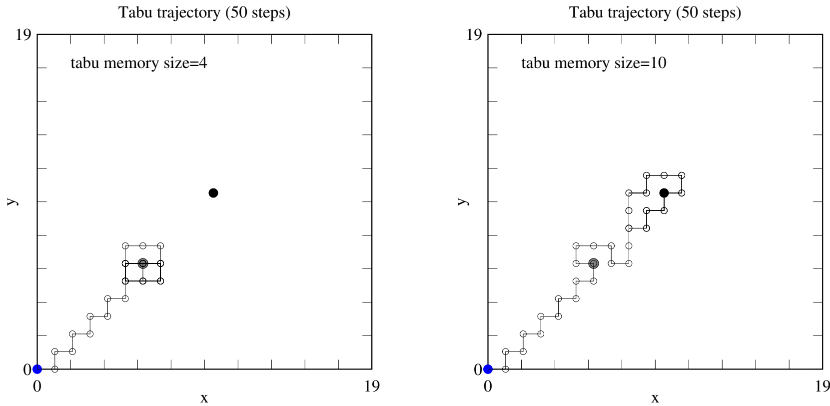


Fig. 3.3. Tabu search trajectory with a tabu list of length 4 (left), and 10 (right)

If the tabu list size is further increased, a larger portion of the search space is explored, as illustrated in Figure 3.4. The space is visited in a systematic fashion and, if there were another local maximum, it would have been found. However, we also see that after 167 iterations the trajectory reaches a point (in red in the figure) whose neighbors are all tabu. This means that the search will abruptly get stuck at this point, preventing the algorithm from visiting further search space points that could potentially be interesting (there are $400 - 167 = 233$ points that have not yet been visited).

3.3 Convergence

By *convergence* of the search process we refer to the question of knowing whether tabu search is able to find the global optimum in a given fitness landscape. Convergence depends on a number of factors and, in particular, on the way the tabu list is managed. The following result can be proved:

If the search space $\mathbf{S}$ is finite and if the neighborhood is symmetric ($s \in V(t)$ implies that $t \in V(s)$), and if any $s' \in \mathbf{S}$ can be reached from any $s \in \mathbf{S}$ in a finite number of steps,
then:
a tabu search that stores all visited points, and also *is allowed to revisit the oldest point in the list*, will visit the whole search space and thus will find the optimum with certainty.

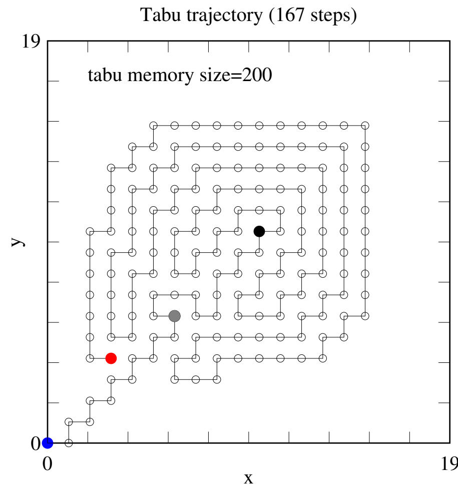


Fig. 3.4. Tabu search trajectory with a memory size of 200 visited points. The red point is the position reached after 167 steps. This situation represents a blockage since all neighbors of the red point are in the tabu list

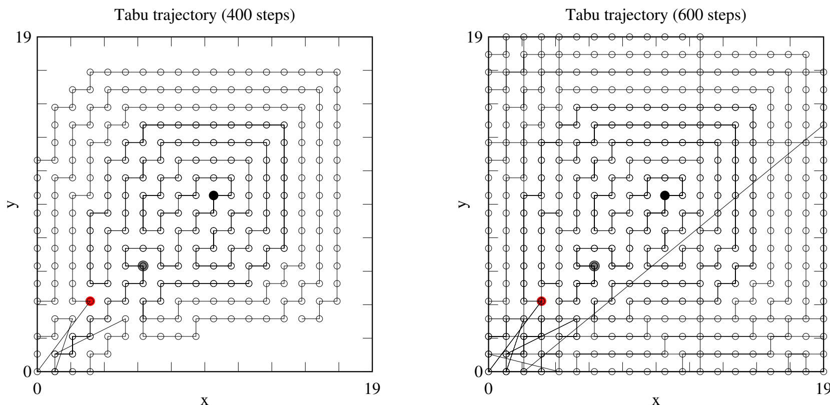


Fig. 3.5. Tabu search trajectory in the case of the example of Section 3.2. When the search gets stuck, it is restarted from the oldest tabu point in the tabu list. If this point has no non-tabu neighbors, the search is restarted from the second oldest point, and so on. The diagonal lines show the jumps needed to find a suitable new starting point. Here we see that, after 600 iterations, the space has not yet been wholly visited

We thus see that this theoretical result implies a memory of size $|\mathbf{S}|$ which is clearly unfeasible in practice for real problems. Moreover, this result doesn't say anything about the time needed for such an exploration; one can only hope that the

search will be more efficient than a random walk. Reference [37] suggests that this time can be exponential in the size of the search space $|\mathbf{S}|$, that is, even longer than what is required for a systematic exhaustive search.

With realistic implementations of tabu search having finite, and not too large memory requirements, the example of Section 3.2 shows that the search process may fail to converge to the global optimum and that the exploration may enter cycles in the search space.

Restarting the search from the oldest tabu point in the tabu list may avoid getting stuck, as shown in Figure 3.4. This strategy is depicted in Figure 3.5. When a point is reached where all the neighbors are tabu, the search is restarted from the oldest tabu point if it has some non-tabu neighbors, otherwise the next oldest point is used, and so on. On the figure, these jumps are indicated by the grey lines. At the beginning the oldest tabu point $(0, 0)$ still has non-tabu neighbors to be visited. However, this point will quickly become a blocking configuration too. At this point, it will be necessary to go forward in the tabu list in order to find a point from which the search can be restarted. Potentially, the process might require more iterations than there are elements in $\mathbf{S}$ because of the extra work needed to find a good restarting point. Figure 3.5 illustrates such a situation after 400 and 600 iterations, with an arbitrarily large memory. Here we see that 600 iterations are not yet enough to explore the whole search space, which only contains 400 points in our example, since there are eight points left. An exhaustive search would have been better in this case.

3.4 Tabu List, Banning Time, and Short- and Long-Term Memories

3.4.1 Principles

As we saw above, the goal of the tabu list is to avoid resampling configurations that have already been visited, thus facilitating a more efficient exploration of the search space. We will now discuss how to implement and manage the tabu list, and we will also describe the short- and long-term memory mechanisms.

In Section 3.2 the visited solutions are directly placed on the tabu list. In practice, a list of size M can be efficiently implemented using a circular buffer, as illustrated in Figure 3.6. In this way, once the buffer becomes full, all the M configurations visited last can be memorized by just deleting the oldest one, which is at the rear of the queue, and adding the newest one to the front.

Usually, it is more convenient to store items in the tabu list that are related to the visited solutions, rather than the solutions themselves. In general the stored items are

- *attributes* belonging to the visited solutions such as fitness values.
- the *moves* that have been used to explore the search space. The notion of move, or transformation, has been introduced in Section 2.5.

When storing solution attributes or moves, a circular buffer as schematized in Figure 3.6 can be used, thus preventing the search from using a solution possessing an attribute that has already been encountered in the previous M iterations.

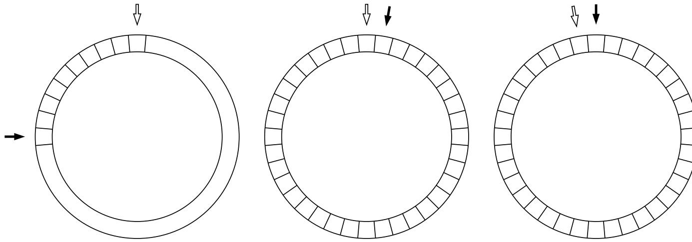


Fig. 3.6. A tabu list storing the M last-visited solutions can be implemented as an array of size M with an index i pointing to the oldest element of the queue (white arrow), and a variable n that stores the number of elements in the queue. Initially, $i_0 = 0$. While $n < M$, the new elements are added at position $i_0 + n$ (black arrow) followed by the operation $n = n + 1$. When $n = M$, i_0 is incremented and a new element is placed at $i_0 + n \bmod M$

Often, a different approach will be chosen that takes into account the concept of a *banning time*, also called *tabu tenure*. The idea can be applied to tabu lists containing solution properties, as well as to lists containing forbidden moves. This can be formulated thus:

> If a move m has been used in iteration t, or the attribute h characterizes the current solution at time step t, then the inverse move of m, or the attribute h, will be forbidden until iteration $t + k$, where k is the banning time.

The tenure's duration k, as well as the size of the tabu list, are guiding parameters of the metaheuristic. The k value is often chosen randomly in each iteration according to a specific probability distribution.

3.4.2 Examples

Below we will look at the tabu list concept and tenure times in simple situations, as well as at the notion of long-term memory as opposed to short-term memory. A more realistic example will be presented in Section 3.6.

Example 1:

Reference [31] suggests the following example: the tabu list is implemented as an array T of size M and it uses a positive integer function $h(s)$ which is defined at all points s of the search space **S**. This function defines the attribute used to build the tabu list. Let s_t be the current solution in iteration t. The attribute $h(s_t)$ is transformed into an index i in T through a hashing function. For example, we could have $i = h(s_t) \bmod M$. Next, the value $t + k$ is stored at $T[i]$, where k is the chosen tenure time, i.e., the duration of the prohibition for this attribute. A configuration s will thus be tabu in iteration t' if $t' < T[h(s) \bmod M]$. In other words, with this mechanism, the points having the attribute $h(s_t)$ will be forbidden during the next k iterations.

Example 2:

Let us consider a random walk in $\mathbb{Z}^2$ with a neighborhood defined by the four moves `north`, `east`, `south`, and `west`. The tabu list might, for example, be represented by the array below, with all zeroes initially in both columns :

	tenure time (short-term memory)	move frequency (long-term memory)
`north`	0	0.5
`east`	0	0.5
`south`	2	0
`west`	4	0

The interpretation is as follows: let us suppose that in the first iteration $t = 1$ the move `north` has been selected. The inverse move `south` is now tabu, to avoid going back to the same configuration for a time $k = 1$ iterations. Therefore, $t+k = 2$ is entered under the "tenure time" column at line `south`. In the next iteration $t = 2$ the move `south` is tabu since t is not strictly greater than 2. Let us suppose that the move `east` is now chosen. If the banning time of the inverse move is now $k = 2$, the entry `west` of the tabu list will be $2 + 2 = 4$. In the the iteration $t = 3$, the move `west` will thus be forbidden. On the other hand, the moves `north`, `east`, and `south` are now possible. The latter is no longer tabu because now $t > 2$. It is easy to see that if the tenure time k were large, the exploration would be biased towards `north` and `east` since, each time these moves are chosen, they extend the tenure times of the inverse moves `south` and `west`. For this reason, a second column is added to the tabu list, representing the long-term memory of the exploration process. Its entries are the frequencies of each move from the very beginning of the search. In the present case, after two iterations, two moves have each been chosen once, which gives a frequency $1/2$ for `north` and `east`. Equipped with this information, it would now be simple to allow a tabu move if the move has not been performed in a long enough time interval.

3.5 Guiding Parameters

Like most metaheuristics, tabu search needs a number of parameters to be suitably chosen to guide the search and make it as efficient as possible, in particular those that specify the behavior of the tabu list. We now look at the impact of those parameters on the way the search space is visited.

To start with, it is easy to see that the larger the number of forbidden moves, the larger the probability of choosing as the next configuration $\mathbf{x} \in \mathbf{S}$ a point of low fitness, leading to more exploration of the search space. This is the *diversification* aspect of the search. On the other hand, the shorter the tabu list, the larger the probability of choosing a high-fitness point in the neighborhood: this is the *intensification* aspect of the search.

In general, tabu search also implements *aspiration criteria* according to which a solution can be included in the allowed set even if it is currently forbidden. For example, this might be the case when a point is found which is in the tabu list but whose fitness is better than the current solution.

The tabu list content's evolution is managed according to the concepts of short-term and long-term memories, of which we have already seen a simple example. This is an aspect that is rather technical, as it should reflect detailed knowledge about the tabu metaheuristic and the problem at hand. There is thus no established procedure that is equally valid in all cases. This freedom translates into the choice of parameters such as the list size M or the probability distribution used to define the tenure time.

Long-term memory collects information on the search trajectory during the whole search process. This information can be used to more efficiently guide the choice of the next configuration. For example, it has been observed that in the short term it is useful to select the same move several times, whereas this would not work in the long term. Using the long-term frequency of attributes or moves, one can penalize moves that have been performed too often in the long term, for example by adding a penalizing contribution to the fitness, which will help diversify the search by performing moves that have seldom been used.

In practice, tabu search will combine, or alternate, short- and long-term memories to update the tabu list according to control parameters that are defined for each case, and possibly even on the fly, in order to attain a good compromise between diversification and intensification.

3.6 Quadratic Assignment Problems

3.6.1 Problem Definition

The Quadratic Assignment Problem (QAP) belongs to a class of important combinatorial optimization problems in which, given a number of *objects* and *locations*, *flow values* between the objects, and *distances* between the locations, the goal is to assign all objects to different locations in such a way that the sum of the products of the flow values between pairs of objects and the distances between the respective locations is minimal. QAP is NP-hard and the instances arising in applications are very difficult to solve in practice. In fact, only small-size instances with a few tens of locations can be solved by exact methods and there are no good approximation algorithms. Metaheuristics are the only feasible approach for larger problems.
Formally, the problem can be expressed thus:

Let us consider n objects and n possible locations for the objects. Let f_{ij} be the values of the flows between the pairs of objects i and j, and let d_{rs} be the distances between locations r and s. The goal is to find an optimal placement $i \to r_i$ of the n objects such that the following objective function is minimized:

$$f = \sum_{i,j} f_{ij} d_{r_i r_j} \tag{3.2}$$

where r_i denotes the position chosen for object i. The search space **S** is all permutations of the n objects.

An example of a quadratic assignment problem is illustrated in Figure 3.7. The squares A, B, C, and D in the figure can be seen as the possible locations of some electronic component on a chip. The numbers 0, 1, 2, and 3 then correspond to the type of component placed at those locations. Two of the $4!$ possible permutations of the objects 0, 1, 2, and 3 on the locations A, B, C, D are shown in the figure.

The distances $d_{rs} = d_{sr}$ between any two locations r and s are defined by their position in space. Here we assume $d_{AB} = 4$, $d_{AC} = 3$, $d_{AD} = 5$, $d_{BC} = 5$, $d_{BD} = 3$, and $d_{CD} = 4$. The lines joining the locations represent the flows f_{ij} whose values are given by the number of lines. In the example we have the following flow values: $f_{01} = f_{03} = 1$, $f_{23} = 3$, $f_{13} = 2$, $f_{02} = f_{12} = 0$, with $f_{ij} = f_{ji}$.

In another context, one might also imagine that the squares are buildings and the flows f_{ij} stand for the stream of people between the buildings according to their function such as supermarket, post office, and so on. In both cases, what we look for is the placement that minimizes wire length or the total distance traveled.

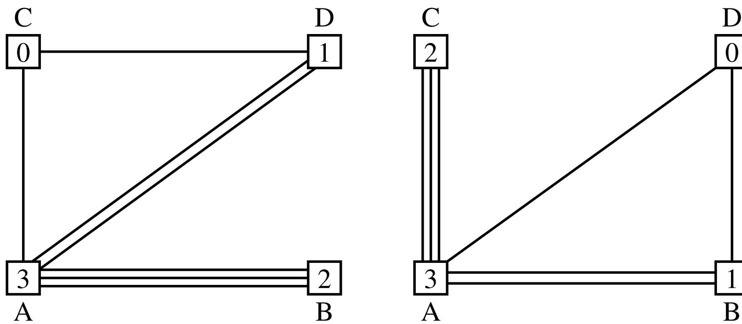


Fig. 3.7. Schematic illustration of a quadratic assignment problem. The squares represent locations and the numbers stand for objects. The figure shows two possible configurations, with the right solution having a better objective function value than the left one

For the configuration shown in the left image of Figure 3.7, the fitness value is

$$\begin{aligned} f &= \sum_{i,j} f_{ij} d_{r_i r_j} \\ &= f_{01} d_{CD} + f_{03} d_{CA} + f_{13} d_{DA} + f_{23} d_{BA} \\ &= 1 \times 4 + 1 \times 3 + 2 \times 5 + 3 \times 4 = 29 \end{aligned} \tag{3.3}$$

For the configuration depicted in the right image the fitness is:

$$\begin{aligned} f &= f_{01} d_{DB} + f_{03} d_{DA} + f_{13} d_{BA} + f_{23} d_{CA} \\ &= 1 \times 3 + 1 \times 5 + 2 \times 4 + 3 \times 3 = 25 \end{aligned} \tag{3.4}$$

We thus have that the permutation $0 \to C$, $1 \to D$, $2 \to B$, and $3 \to A$ is worse that the permutation $0 \to D$, $1 \to B$, $2 \to C$, and $3 \to A$. It is possible to conclude by inspection that the latter represents the global minimum, although it is not unique as the symmetric configurations lead to the same objective function value.

A particular case of the QAP problem is the well-know traveling salesman problem (TSP) which has been introduced in Chapters 1 and 2 and will be treated in detail in Chapter 5. In this case the locations are the cities to be visited, the objects are the visiting order, and the flows are $f_{i,i+1} = 1$ and zero otherwise.

3.6.2 QAP Solved with Tabu Search

Choice of neighborhood

As already observed above, the search space of a quadratic assignment problem of size n is the set of permutations of n objects. We are thus going to describe the admissible solutions through permutations.

$$\mathbf{p} = (i_1, i_2, \ldots i_n)$$

where i_k is the index of the object positioned at location k.

There is more than one way to define the neighborhood $V(\mathbf{p})$ through basic moves. For example

- By the exchange of two contiguous objects

$$(i_1, i_2, \ldots i_k, i_{k+1}, \ldots i_n) \to (i_1, i_2, \ldots i_{k+1}, i_k, \ldots i_n)$$

  Such a neighborhood is thus defined by $n - 1$ possible moves.
- By the exchange of two arbitrary objects

$$(i_1, i_2, \ldots i_k \ldots i_\ell, \ldots i_n) \to (i_1, i_2, \ldots i_\ell, \ldots i_k, \ldots i_n)$$

  In this case the corresponding neighborhood contains $n(n-1)/2$ elements.

The behavior of tabu search will depend on the particular neighborhood chosen. In practice, it has been observed that the second neighborhood type above gives good results. Although in principle it requires a lengthier evaluation since its size is quadratic in n, while the former is linear in n, we shall see below that the time can be made shorter by using incremental evaluation.

Evaluation of the neighborhood

The computational burden for evaluating the fitness of the neighbors of a given configuration grows as a function of the number of moves m_i that define the neighborhood. In the tabu method, the search operator U selects the non-tabu neighbor having the best fitness. Therefore, starting from the current solution $\mathbf{p}$, we must compute

$$\Delta_i = f(m_i(\mathbf{p})) - f(\mathbf{p})$$

for all the moves m_i. If the move m_i is a transposition, only two objects change their places and most terms in the fitness function $f = \sum_{kl} f_{i_k i_l} d_{kl}$ stay the same. In this case it can be shown that, instead of having to compute $\mathcal{O}(n^2)$ terms, it is enough to evaluate only $\mathcal{O}(n)$ of them to obtain Δ_i.

The tabu list

Here we consider the case in which the allowed moves, called (r, s), correspond to the exchange of the two objects at positions r and s

$$(\ldots i_r \ldots i_s \ldots) \overset{(r,s)}{\rightarrow} (\ldots i_s \ldots i_r \ldots)$$

The tabu moves are then defined as being the inverse moves of those that have been just accepted. However, the tabu list structure is slightly more complex than what we have seen in Section 3.4. In the short-term memory context, the move that would switch back object i at location r and object j at position s is tabu for the next k iterations, where k is a randomly chosen tenure time.

The tabu list takes the form of an $n \times n$ matrix of elements T_{ir} whose values are the iteration step numbers t in which the element i most recently left the site r plus the tenure time k.

As a consequence, the move (r, s) will be forbidden if $T_{i_s r}$ and $T_{i_r s}$ both contain a value that is larger than the current iteration count.

3.6.3 The Problem NUG5

The example in this section is borrowed from reference [31]. The term $NUG5$ refers to a QAP benchmark problem class that is contained in the Quadratic assignment problem library (QAPLIB)[1]. It is a problem of size $n = 5$ defined by its flow and distance matrices F and D with elements f_{ij} and d_{rs}

$$F = \begin{pmatrix} 0 & 5 & 2 & 4 & 1 \\ 5 & 0 & 3 & 0 & 2 \\ 2 & 3 & 0 & 0 & 0 \\ 4 & 0 & 0 & 0 & 5 \\ 1 & 2 & 0 & 5 & 0 \end{pmatrix} \qquad D = \begin{pmatrix} 0 & 1 & 1 & 2 & 3 \\ 1 & 0 & 2 & 1 & 2 \\ 1 & 2 & 0 & 1 & 2 \\ 2 & 1 & 1 & 0 & 1 \\ 3 & 2 & 2 & 1 & 0 \end{pmatrix} \tag{3.5}$$

The search process starts with an initial solution represented by the permutation

$$\mathbf{p}_1 = (2, 4, 1, 5, 3)$$

whose fitness $f(\mathbf{p}_1) = 72$ can be easily computed from the F and D matrices. The tabu list is initialized to all zeroes: $T_{ir} = 0, \forall i, r$.

[1] The QAPLIB is to be found at `http://anjos.mgi.polymtl.ca/qaplib/inst.html`.

In iteration 1, the neighbors of $\mathbf{p}_1$ can be represented as a function of the 10 allowed moves m_i and, for each of them, we find the fitness variation. This is shown in tabular form as follows:

m_i	(1,2)	(1,3)	(1,4)	(1,5)	(2,3)	(2,4)	(2,5)	(3,4)	(3,5)	(4,5)
Δ_i	2	-12	-12	2	0	-10	-12	4	8	6

We see that the three moves $(1, 3)$, $(1, 4)$, and $(2, 5)$ are those that give the best improvement in the objective function. We choose one at random among the three, say $m_2 = (1, 3)$.

As the current solution is $\mathbf{p}_1 = (2, 4, 1, 5, 3)$, the m_2 move produces a new configuration

$$\mathbf{p}_2 = (1, 4, 2, 5, 3)$$

whose fitness is $f(\mathbf{p}_2) = f(\mathbf{p}_1) + \Delta(1, 3) = 72 - 12 = 60$. To update the tabu list, we note that, as a consequence of the move, object 1 has left location 3 and object 2 has left location 1, which will cause matrix elements T_{13} and T_{21} to be updated. To do that, as suggested in reference [31], a tenure time k will be drawn at random with uniform probability in the interval

$$k \in [0.9 \times n, 1.1 \times n + 4]$$

The bounds have been chosen empirically and belong to the guiding parameters that characterize heuristic methods.

Assume that $k = 9$ has been drawn. Since we are in iteration 1, replacing object 1 at position 3 and object 2 at position 1 will be forbidden during the next $k+1 = 10$ iterations. The tabu matrix is thus

$$T = \begin{pmatrix} 0 & 0 & 10 & 0 & 0 \\ 10 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix} \tag{3.6}$$

In iteration 2, we obtain the following table for the possible moves

m_i	(1,2)	(1,3)	(1,4)	(1,5)	(2,3)	(2,4)	(2,5)	(3,4)	(3,5)	(4,5)
Δ_i	14	12	-8	10	0	10	8	12	12	6

Move $(1, 3)$ is tabu since it is the inverse of the last accepted move. Move $m_3 = (1, 4)$ is chosen as it is the only one that improves fitness. We get the new configuration

$$\mathbf{p}_3 = (5, 4, 2, 1, 3)$$

whose fitness is $f(\mathbf{p}_3) = 60 - 8 = 52$. This move affects the elements T_{11} and T_{54}. Assuming that now the random draw gives $k = 6$, the tenure time will extend from the next iteration to iteration $2 + 6 = 8$, giving

$$T = \begin{pmatrix} 8 & 0 & 10 & 0 & 0 \\ 10 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 8 & 0 \end{pmatrix} \tag{3.7}$$

In iteration 3, we have the following possibilities

m_i	(1,2)	(1,3)	(1,4)	(1,5)	(2,3)	(2,4)	(2,5)	(3,4)	(3,5)	(4,5)
Δ_i	24	10	*8*	10	0	22	20	8	8	14

We remark that, at this stage, the $(1,4)$ move is tabu but move $(1,3)$ is no longer tabu since it would place object 5 at location 3 and $T_{53} = 0$ (object 5 has never been at position 3). The minimal cost move is $m_5 = (2,3)$ although it doesn't improve fitness since $\Delta_5 = 0$. It is nevertheless chosen as the next point in the search trajectory:

$$\mathbf{p}_4 = (5, 2, 4, 1, 3)$$

giving rise to an unchanged fitness $f(\mathbf{p}_4) = 52$ with respect to the current solution.

Assuming now that the random drawing gives $k = 8$, we obtain the new tabu list

$$T = \begin{pmatrix} 8 & 0 & 10 & 0 & 0 \\ 10 & 0 & 11 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 11 & 0 & 0 & 0 \\ 0 & 0 & 0 & 8 & 0 \end{pmatrix} \tag{3.8}$$

which reflects the fact that object 2 has left position 3 and object 4 has left position 2. These two moves thus become forbidden until iteration $11 = 3 + 8$.

In iteration 4, the moves and costs table is

m_i	(1,2)	(1,3)	(1,4)	(1,5)	(2,3)	(2,4)	(2,5)	(3,4)	(3,5)	(4,5)
Δ_i	24	10	*8*	10	*0*	8	8	22	20	14

There are now two tabu moves, $m_3 = (1,4)$ and $m_5 = (2,3)$. For m_5 this is obvious, as it would undo the previous move. Move m_3 remains tabu as it would replace objects 5 and 1 at locations that they have already occupied. The allowed moves are $m_6 = (2,4)$ and $m_7 = (2,5)$, which both *worsen* fitness by 8 units. In the fourth iteration we are thus witnessing a deterioration of the current objective function value. Let's assume that $(2,4)$ is chosen as the next move, giving

$$\mathbf{p}_5 = (5, 1, 4, 2, 3)$$

with fitness $f(\mathbf{p}_5) = 60 = 52 + 8$.

With a random choice $k = 5$, the elements T_{22} (object 2 leaving position 2) and T_{14} (object 1 leaving position 4) take the value $9 = 4 + 5$ and the tabu list is

$$T = \begin{pmatrix} 8 & 0 & 10 & 9 & 0 \\ 10 & 9 & 11 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 11 & 0 & 0 & 0 \\ 0 & 0 & 0 & 8 & 0 \end{pmatrix} \tag{3.9}$$

The above process is repeated until a given stopping criterion is attained.

In iteration 5, the chosen move is $(1,3)$, which corresponds to an improvement in fitness of 10 units. We have thus

$$\mathbf{p}_6 = (4, 1, 5, 2, 3)$$

with fitness $f(\mathbf{p}_6 = 50)$. This configuration is the best among all those explored so far and it turns out that it is also the global optimum in the present problem. We thus see that accepting a fitness degradation has been beneficial for reaching better quality regions in the search space.

To conclude this chapter, we emphasize that tabu search has proved to be a successful metaheuristic for solving hard combinatorial optimization problems (see, for instance [41] for a number of applications of the algorithm). To provide worsening moves that allow the search to escape from local optima, tabu's specificity is to rely on search history instead of using random move mechanisms as in most other metaheuristics. However, successful implementations of tabu search require insight and problem knowledge. Among the crucial aspects for the success of tabu search we mention the choice of a suitable neighborhood, and clever use of short-term and long-term memories.

4

Simulated Annealing

4.1 Motivation

The method of *simulated annealing* (SA) draws its inspiration from the physical process of metallurgy and uses terminology that comes from that field. When a metal is heated to a sufficiently high temperature, its atoms undergo disordered movements of large amplitude. If one now cools the metal down progressively, the atoms reduce their movement and tend to stabilize around fixed positions in a regular crystal structure with minimal energy. In this state, in which internal structural constraints are minimized, ductility is improved and the metal becomes easier to work. This slow cooling process is called *annealing* by metallurgists and it is to be contrasted with the *quenching* process, which consists of a rapid cooling down of the metal or alloy. Quenching causes the cooled metal to be more fragile, but also harder and more resistant to wear and vibration. In this case, the resulting atomic structure corresponds to a local energy minimum whose value is higher than the one corresponding to the arrangement produced by annealing. Figure 4.1 illustrates this process. Note finally that in practice metallurgists often used a process called *tempering* by which one alternates heating and cooling phases to obtain the desired physical properties. This term will be reused in Section 4.7 to describe an extension of the simulated annealing algorithm.

We can intuitively understand this process in the following way: at high temperature, atoms undergo large random movements thereby exploring a large number of possible configurations. Since in nature the energy of systems tends to be minimized, low-energy configurations will be preferred, but, at this stage, higher energy configurations remain accessible thanks to the thermal energy transferred to the system. In this way, at high temperature the system is allowed to explore a large number of accessible states. During the exploration, the system might find itself in a low-energy state by chance. If the energy barrier to leave this state is high, then the system will stay there longer on average. As temperature decreases, the system will be more and more constrained to exploit low-amplitude movements and, finally, it will “freeze” into a low-energy minimum that may be, but is not guaranteed to be, the global one.

© Springer Nature Switzerland AG 2018

B. Chopard, M. Tomassini, *An Introduction to Metaheuristics for Optimization*,
Natural Computing Series, https://doi.org/10.1007/978-3-319-93073-2_4

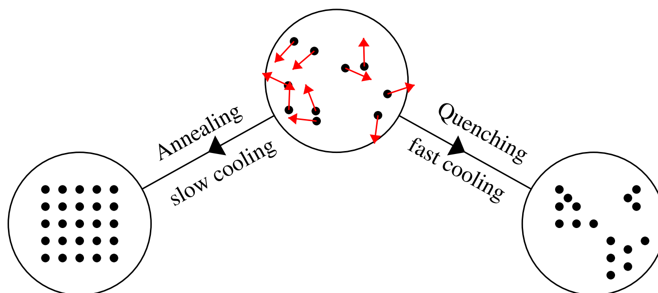


Fig. 4.1. Illustration of the metallurgical processes of annealing and quenching. The upper disk represents a sample at high temperature, in which atoms move fast, in a random way. If the sample is cooled down slowly (annealing), the atoms reach the organized state of minimal energy. But if the cooling is too fast (quenching), the atoms get trapped in alternating ordered and disordered regions, which is only a local minimum of energy

In 1983, Kirkpatrick et al. [50], taking inspiration from the physical annealing process, had the idea of using an algorithm they called *simulated annealing* to search for the global minimum of a spin glass system, which can be shown to be a difficult combinatorial optimization problem. In the following years, simulated annealing has been successfully used in a large number of optimization problems unrelated to physics.

4.2 Principles of the Algorithm

The simulated annealing method is used to search for the minimum of a given objective function, often called the *energy* E, by analogy to the physical origins of the method. The algorithm follows the basic principles of all metaheuristics. The process begins by choosing an arbitrary admissible initial solution, also called the initial configuration. Furthermore, an initial "temperature" must also be defined, following a methodology that will be described in Section 4.6.

Next, the moves that allow the current configuration to reach its neighbors must also be defined. These moves are also called *elementary transformations*. The algorithm doesn't test all the neighbors of the current configuration; instead, a random move is selected among the allowed ones. If the move leads to a lower energy value, then the new configuration is accepted and becomes the current solution. But the original feature of SA is that even moves that lead to an increase of the energy can be accepted with positive probability. This probability of accepting moves that worsen the fitness are computed from the energy variation ΔE before and after the given move:

$$\Delta E = E_{new} - E_{current}$$

The probability p of accepting the new configuration is defined by the exponential

$$p = \min(1, e^{-(\Delta E/T)}) \tag{4.1}$$

This relation is called the Metropolis rule for historical reasons, and its graphical representation is indicated in Figure 4.2. As the figure shows, the rule says that for $\Delta E \leq 0$, the acceptance probability is one, as the exponential is larger than one in this case. In other words, a solution that is better than the current one will always be accepted. On the other hand, if $\Delta E > 0$, which means that the fitness of the new configuration is less good, the new configuration will nonetheless be accepted with probability $p < 1$ computed according to equation (4.1). Thus, a move that worsens the fitness can still be accepted. It is also clear that the larger ΔE is, the smaller p will be and, for a given ΔE, p becomes larger with increasing temperature T. As a consequence, at high temperatures worsening moves are more likely to be accepted, making it possible to overcome fitness barriers, providing exploration capabilities, and preventing the search being stuck in local minima. In contrast, as the temperature is progressively lowered, the configurations will tend to converge towards a local minimum, thus exploiting a good region of the search space. Indeed, in the limit for $T \to 0$, $p \to 0$, and no new configuration with $\Delta E > 0$ is accepted.

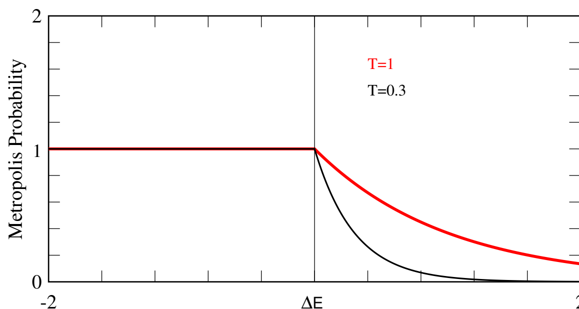


Fig. 4.2. Acceptance probability function according to equation (4.1) for two different temperatures T

The choice of the Metropolis rule for the acceptance probability is not arbitrary. The corresponding stochastic process that generates changes and that accepts them with probability $p = e^{-(\Delta E/T)}$ samples the system configurations according to a well-defined probability distribution p that is known in equilibrium statistical mechanics as the Maxwell-Boltzmann distribution. It is for this reason that the Metropolis rule is so widely used in the so-called Monte Carlo physical simulation methods.

A fundamental aspect of simulated annealing is the fact that the temperature is progressively decreased during the search. The details of this process are specified by a *temperature schedule*, also called a *cooling schedule*, and can be defined in different ways. For instance, the temperature can be decreased at each iteration following

a given law. In practice, it is more often preferred to lower the temperature in stages: after a given number of steps at a constant temperature the search reaches a stationary value of the energy that fluctuates around a given average value that doesn't change any more. At this point, the temperature is decreased to allow the system to achieve convergence to a lower energy state. Finally, after several stages in which the temperature has been decreased, there are no possible fitness improvements; a state is reached that is to be considered the final one, and the algorithm stops. Figure 4.3 summarizes the different stages of the simulated annealing algorithm.

Another interpretation of equation (4.1) can be obtained by taking logarithms and writing it as

$$\Delta E = -T \ln(p) \tag{4.2}$$

for positive ΔE.

This is the amplitude of a worsening energy difference that can be accepted with probability p. For example, an energy barrier of $\Delta E = 0.69T$ will be overcome with probability $1/2$. If we were able to estimate the energy variations in the fitness landscape, this would allow us to determine the temperature that would be needed to traverse the energy barriers with a given probability. In Section 4.6 we will use this idea to compute a suitable initial temperature for a simulated annealing run.

The behavior of simulated annealing is illustrated in Figure 4.4. Two search trajectories generated as described in the flowchart of Figure 4.3 are shown in the figure. The energy landscape is unidimensional with several local optima. Both search trajectories find the global minimum but with paths of different lengths. The grey part of the figure shows the amplitude ΔE of the energy differences that are accepted with probability $p = 1/2$ according to the Metropolis criterion. Initially, this amplitude is chosen to be rather large in order to easily traverse and sample the whole search space. However, as exploration progresses, this amplitude decreases, stage by stage, following the chosen temperature schedule. At the end of the process only small amplitude variations are possible and search converges, one hopes to the global minimum.

4.3 Examples

To illustrate the SA method, here we will look at the traveling salesman problem, or TSP, already introduced in Chapter 1, Section 1.4 and Chapter 2, Section 2.2.4. This problem will be taken up again with a different metaheuristic in Chapter 5.

The problem instance considered here is extremely simple by modern standards but it will be useful to illustrate the workings of simulated annealing. The points (cities) are uniformly distributed on a circle of radius $r = 1$, as shown in Figure 4.5. The goal is to find the shortest path that goes through all the cities once and back to the starting point. Given the placement of the cities in this case, it is easy to see that the solution is a 30-vertex polygon whose vertices are the cities and whose length is close to $2\pi r$. However, there exist *a priori* 30! possible paths through these points. The SA metaheuristic will start the exploration of this search space from an initial

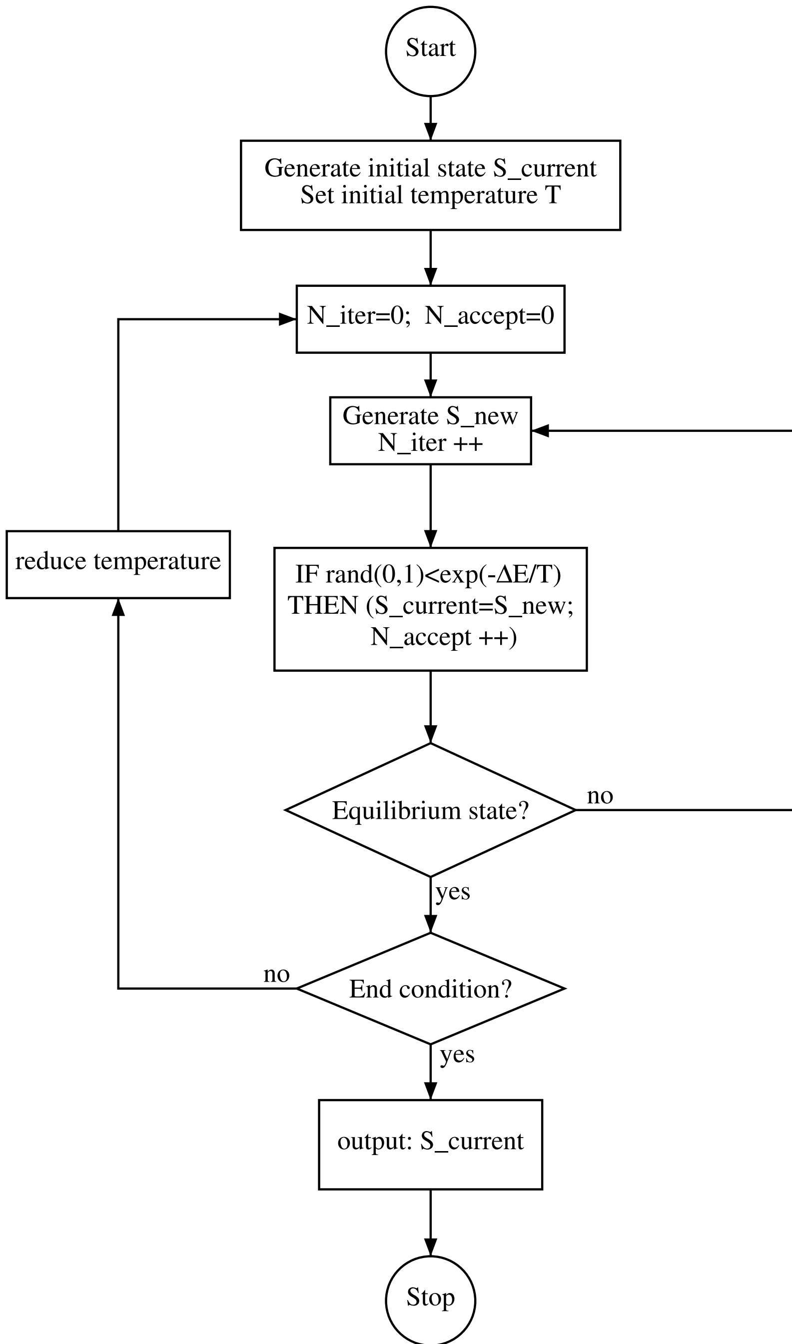


Fig. 4.3. Flowchart for the simulated annealing algorithm

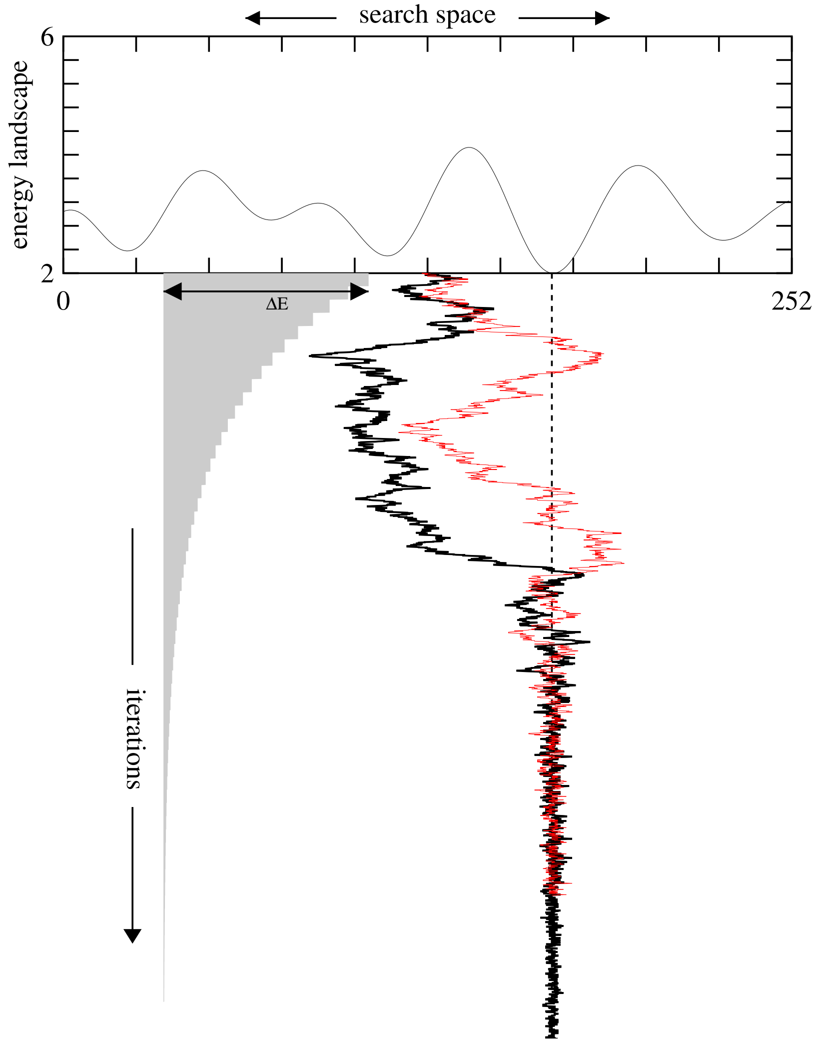


Fig. 4.4. An example of two simulated annealing trajectories (in red and black respectively) of a search for the global minimum of the multimodal fitness landscape shown on top of the figure. The grey area indicates the energy variation amplitude ΔE that is accepted by the Metropolis rule with probability 1/2

random tour going through these points. In the present case it is the tour at the upper left corner of Figure 4.5 whose length, i.e., fitness, is $f = 42.879$.

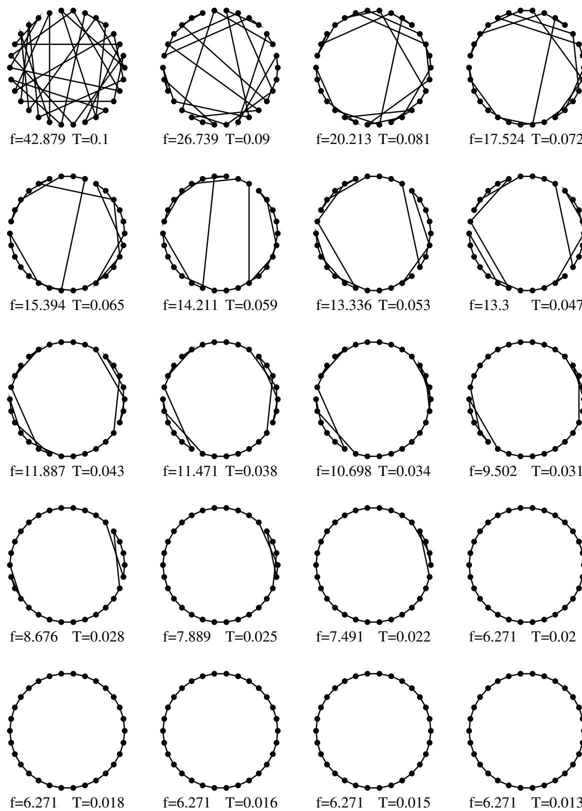


Fig. 4.5. Simulated annealing iterations for finding the shortest tour for a salesman wishing to visit $n = 30$ cities uniformly distributed on a circle. The figure shows the configuration obtained at each temperature step, the corresponding temperature value, and the tour's length

The initial temperature is empirically chosen to be $T_0 = 0.1$. In this example, fewer than 5,000 iterations were needed to find the minimum-length tour starting from the given initial condition. An iteration corresponds to the exchange of two cities in the tour. If this move shortens the tour, then it is accepted. Otherwise, it may be accepted as well, but with a probability that decreases with decreasing temperature and that also decreases the more the move degrades the tour length.

Figure 4.5 shows 20 steps of the search, each one corresponding to a new, and lower, temperature step. The new temperature at step $k + 1$ is obtained from the temperature at step k by the relation $T_{k+1} = 0.9T_k$. In this example, it has been

empirically decided to change the temperature step after 20 accepted moves at a given temperature.

We also note that during the last five temperature steps there has been no improvement in tour length. This is an indication that the search is trapped in a minimum of the energy function. In the present case, it happens to be the globally optimal solution, which is easy to verify thanks to the specific geometric features of the problem. As we have already remarked several times, this need not always be the case as metaheuristics do not give guarantees of the quality of the solution they find.

To experiment a bit, the algorithm was rerun with a higher initial temperature $T_0 = 0.5$, and longer stages at the same temperature, 200 accepted moves instead of 20. The optimal solution was obtained again but required about 55,000 iterations in total. The optimal solution was already obtained at temperature stage $T_{15} = 0.114$ but it disappeared in the following temperature stages. Indeed, a worse fitness value of $f = 6.685$ was reached at $T_{16} = 0.092$. However, the optimal value was found again later.

The choices that have to be made before running the algorithm amount to setting the values of several parameters: initial temperature T_0, number of accepted moves before changing the temperature, how to decrease the latter, and the stopping criterion. These are the *guiding parameters* of the metaheuristic and there is no principled way to satisfactorily set their values for all problems. We shall try to better understand their role later in the chapter, and in Section 4.6 we will give some rules of thumb for their choice.

We said in Section 2.5 that the kind of move used to explore a permutation space such as the one generated by a TSP problem, can influence the quality of the solutions obtained through a given metaheuristic. We now illustrate this by considering SA for the solution of a TSP problem with 500 cities randomly distributed on a square domain of side 2 (in arbitrary length units), a larger but still moderate problem size by modern standards. In Figure 4.6 we show on the left a randomly chosen tour as the initial solution to start the simulated annealing search.

In the middle image of Figure 4.6 the moves used to go from the current solution to the next one are transpositions of two randomly chosen cities, as explained in chapter 2. The best tour found after SA convergence is clearly not optimal, as there are crossings that could be eliminated in order to shorten the total tour length.

In the same Figure 4.6, right image, the same problem is solved with SA but this time using *2-Opt* moves, instead of transpositions. This kind of move was explained in Chapter 2. The final result is clearly better, both visually and in terms of tour length: $L = 34.628$ here compared to $L = 50.703$ using transpositions. Apart from the choice of moves, the runs have exactly the same initial conditions and the same parameters: an initial temperature of $T_0 = 0.5$ and the same temperature schedule. The latter is defined by $T_{k+1} = 0.95T_k$ and stages at the same temperature were of length $20 \times n = 10{,}000$ accepted moves, or $150 \times n = 75{,}000$ tried moves, whichever comes first. The search was stopped when there was no fitness improvement during three successive temperature stages. The results suggest that *2-Opt* moves give better results in this problem compared to transpositions. That said, it is still possible to

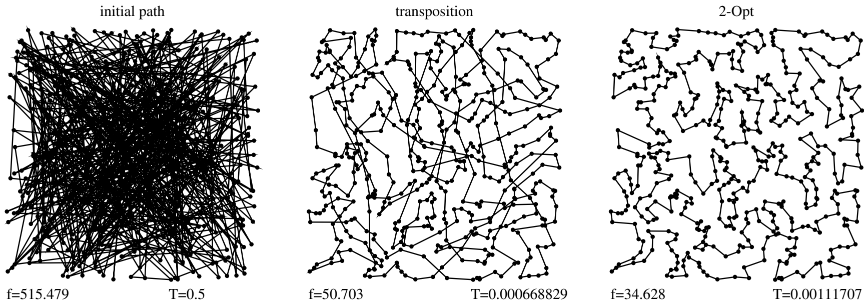


Fig. 4.6. Simulated annealing results on a TSP problem with 500 cities randomly distributed in the plane. On the left, the initial tour with its length f, and the initial temperature T. In the middle image, the solution found by SA using *transposition* moves. On the right, the solution found using moves of type *2-Opt*, with the same temperature schedule and the same initial conditions. It is apparent that *2-Opt* moves lead to a better quality solution

improve the results with transpositions by fine-tuning the temperature schedule, but at the expense of longer computing times.

In Chapter 5 we shall see that it is possible to find the optimal solution to this problem by using the *Concorde* algorithm, a state-of-the-art specialized program for solving large-scale TSP problems. As shown in Figure 4.7 the optimal tour in this problem has length $L^* = 33.0015$. Thus, the result obtained with SA using *2-Opt* moves is just less than 5% worse. While *Concorde* takes about 40 seconds on a laptop to solve the problem to optimality, the tour found by simulated annealing (Figure 4.6, right image) only took 1 second on the same laptop computer.

When considering moves of the *2-Opt* type, the variation Δf in tour length is easy to compute. For a move (i, j)-*2-Opt*, as described in Section 2.5, we have

$$\Delta f = d_{i,j} + d_{i+1,j+1} - d_{i,i+1} - d_{j,j+1}$$

where $d_{i,j}$ is the distance between the cities visited in steps i and j. If the move is accepted, the order of travel between cities $i + 1$ and j must be reversed. Still, it is possible to deal with relatively large problems in reasonable time: about 10 seconds for 5,000 cities and about 40 seconds for 10,000 cities on a standard laptop. The optimal tours found in these two cases are depicted in Figure 4.8.

The previous discussion has shown that optimizing the guiding parameters of the search is not an easy matter. Problem knowledge would surely help, for example in the choice of the best adapted move operator. However, we have seen that even a relatively naive and conservative choice may lead to satisfactory results.

In the TSP problem discussed above the moves are more or less complex city exchanges. However, in other applications the suitable moves can be very different. For instance, in the graph layout problem, one looks for a placement of the graph vertices that minimizes the number of edge crossings. In this case, the moves might be random displacements of a graph vertex. Such an approach is illustrated in Fig-

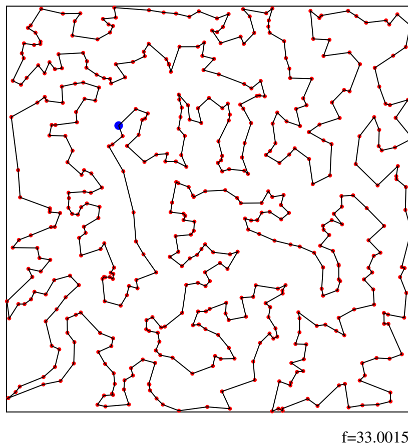


Fig. 4.7. Optimal tour obtained with the *Concorde* algorithm (see Chapter 5 for details) for the 500 cities problem of Figure 4.6. The minimal length is $L = 33.0015$

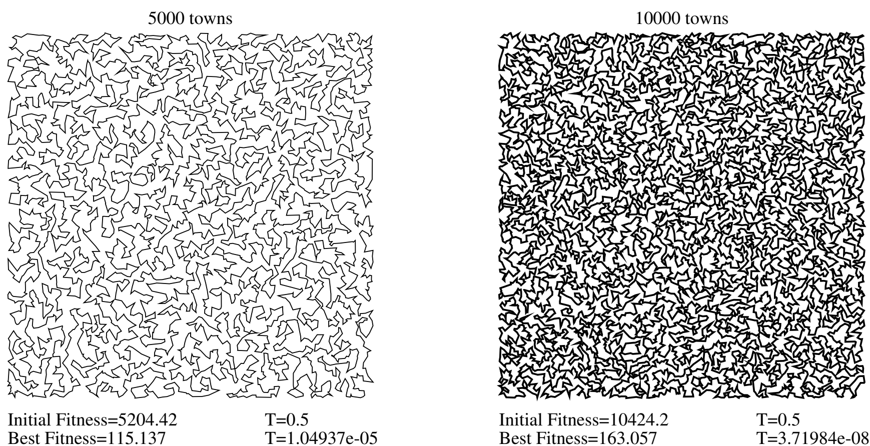


Fig. 4.8. Simulated annealing results for a TSP problem with 5,000 and 10,000 cities respectively. The SA parameters are the same as those used to generate Figure 4.6

ure 4.9. In this example the graph edges are given and simulated annealing looks for the minimum of the following fitness function:

$$f = \sum_{i,j} A_{ij}(d_{ij} - d_0)^2 + \sum_{i,j} \frac{V_0}{d_{ij}} \tag{4.3}$$

Here A_{ij} is the graph adjacency matrix and d_{ij} is the distance between vertices i and j. The quantities d_0 and V_0 are constants. This objective function corresponds to the criterion that pairs of vertices must be as far apart as possible, which minimizes

the second term, but connected vertices should ideally be at a distance d_0, thus minimizing the first term. This fitness function is appropriate for the graph considered in the example. We also note that the f function defined in equation (4.3) can be minimized by using straightforward classical methods, without help from metaheuristics.

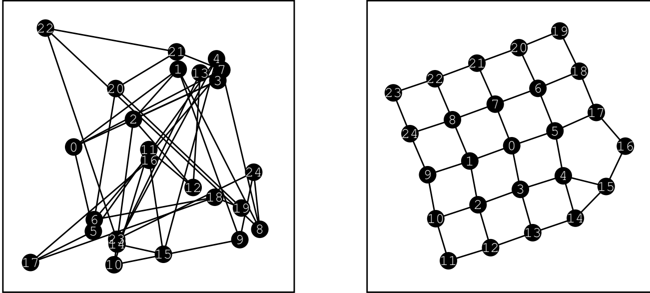


Fig. 4.9. Solution to a graph layout problem obtained with simulated annealing. On the left, the initial graph layout. On the right, the solution found by simulated annealing

4.4 Convergence

An important question in all metaheuristics is to know whether they will be able to find the global optimum of a problem after a sufficient number of iterations, or whether they will get stuck in a local optimum. We speak of "convergence" when we describe the long-term behavior of a metaheuristic.

For simulated annealing, theoretical analyses based on Markov chains have shown that, under certain conditions, SA converges *in probability* to the global optimum. This means that SA obtains a solution arbitrarily close to the global optimum with probability arbitrarily close to 1. The conditions that must be satisfied are:

- The temperature must not be decreased too quickly during the search process.
- The elementary transformations (moves) must be reversible: if we go from a configuration A to a configuration B through an elementary transformation, then it should be possible to go from B to A as well.
- Any feasible state of the system must be reachable from any other state in a finite number of moves.

The function $T(t)$ that dictates how the temperature T is progressively decreased is the temperature schedule, as we might recall from above. It can be shown that if T doesn't decrease faster than $C/\log(t)$ for large time step t then convergence is assured. C is a constant whose value is related to the "depth" of the "energy wells" of

the problem, which amounts to the amplitude of fitness variations in the search space. Of course, this quantity is not known *a priori* because it would require a previous exploration of the space. Another reason that renders such a temperature schedule impractical is the slowness of the process owing to the logarithmic dependence.

In conclusion, simulated annealing is based on a more solid mathematical theory than several other metaheuristics. In spite of this, when applied in practice, the algorithm requires good empirical choices for the parameters if it is to be efficient, since their values affect both the computing time and the quality of the results.

4.5 Illustration of the Convergence of Simulated Annealing

Let us consider the search space $\mathbf{S} = \{1, 2, \ldots, n_x\} \times \{1, 2, \ldots, n_y\}$, with a neighborhood of the point $(x, y) \in \mathbf{S}$ formed by the following points:

$$V(x, y) = \{(x + 1, y), (x - 1), y), (x, y + 1), (x, y - 1)\} \tag{4.4}$$

The fitness, or energy, $E(x, y)$ of the problem is indicated in Figure 4.10, in which $n_x = n_y = 10$ has been chosen. The goal is to find the global minimum, which is at $(x_m, y_m) = (9, 2)$. We also see that the space is multimodal and the energy has several minima. The most important one after the global minimum is the minimum at $(6, 7)$. This local minimum has the potential for attracting search trajectories since its basin is rather large.

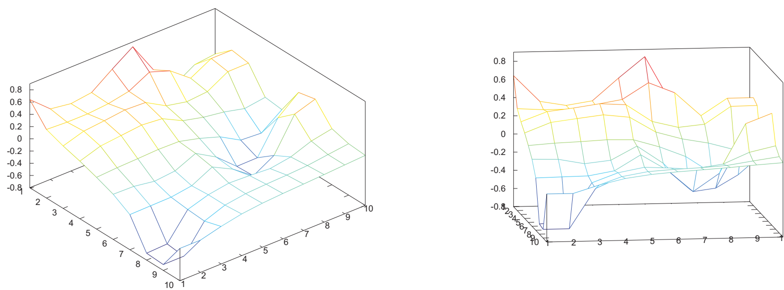


Fig. 4.10. Example of an energy landscape $E(x, y)$, which is a subset of $\mathbf{Z}^2$. The left and right images correspond to two different views of the same landscape $E(x, y)$

This search space contains only $n_x \times n_y = 100$ possible solutions and an exhaustive search would be trivial to conduct. However, we use this problem here for didactic reasons, in order to study in detail how simulated annealing behaves when the search space $\mathbf{S}$ is sufficiently small to allow the process to be followed in detail.

In order to analyze the behavior of SA, we denote the elements of $\mathbf{S}$ by i, where i takes the values from 1 to $|\mathbf{S}| = n_x \times n_y = 100$. The relationship between i and the spatial coordinates (x, y) is

$$i = n_x(y-1) + x$$
$$x = \text{mod}(i-1, n_x) + 1 \qquad y = \text{int}((i-1)/n_x) + 1 \tag{4.5}$$

Let $P(t, i)$ be the probability that the current SA configuration in iteration t is i. The probability that the current solution becomes j at the next iteration is given by the joint probability $P(t+1, j, t, i)$ of having j at time step $t+1$ and i at time step t, summed over all the i's. Therefore

$$P(t+1, j) = \sum_i P(t+1, j, t, i) = \sum_i P(t, i) W_{ij}(t) \tag{4.6}$$

where $W_{ij}(t)$ is called the transition probability from state i to state j. For all possible state transitions, this is a matrix of size $(n_x \times n_y)^2 = 100 \times 100$ which is defined by the neighborhood and the Metropolis rule.

For the neighborhood (4.4) and the relation (4.5), at most four other states j are accessible from state i. If i belongs to the border of the domain $\mathbf{S}$ there are three neighbors, and only two if it is a corner. Let $k_{out}(i)$ denote the number of i's neighbors, E_i the fitness of state or configuration i, and $P_{metro}(E_i, E_j, T)$ the probability according to the Metropolis rule of accepting a move from a state with energy E_i to a state with energy E_j at temperature T. Since the temperature depends on the iteration number, we thus have

$$W_{ij}(t) = \begin{cases} 0 & \text{if } j \text{ is not a neighbor of } i \\ \frac{1}{k_{out}(i)} P_{metro}(E_i, E_j, T(t)) & \text{if } j \text{ is a neighbor of } i \end{cases} \tag{4.7}$$

Finally, we denote by

$$W_{ii}(t) = 1 - \sum_{j=j \text{ is a neighbor of } i} W_{ij}(t)$$

the probability that simulated annealing rejects the chosen move and thus no change of state takes place.

For our problem, the transition matrix is given in Figure 4.11 (left image) for temperature $T = 2$.

From equation (4.6) we can compute $P(t, k)$ as a function of $P(0, \ell)$. We first remark that using (4.6) twice we obtain

$$\begin{aligned} P(t, k) &= \sum_j P(t-1, j) W_{jk}(t-1) \\ &= \sum_j \sum_i P(t-2, i) W_{ij}(t-2) W_{jk}(t-1) \\ &= \sum_i P(t-2, i) \left[W(t-2) W(t-1) \right]_{ik} \end{aligned} \tag{4.8}$$

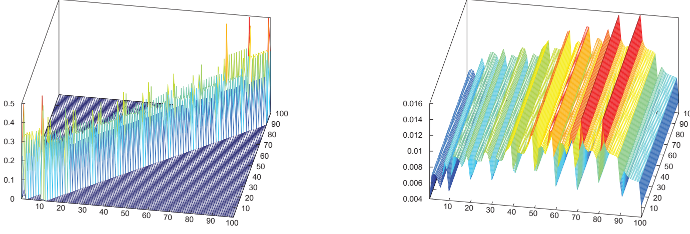


Fig. 4.11. On the left, the SA transition matrix corresponding to the energy landscape of Figure 4.10, for temperature $T = 2$ and with states numbered according to equation (4.5). On the right, the transition matrix after 500 iterations (W^{500}) at the same temperature $T = 2$. It is seen that simulated annealing visits the whole search space

where the term $W(t-2)W(t-1)$ is the matrix product of $W(t-2)$ and $W(t-1)$. By iterating this process we get

$$P(t,k) = \sum_{\ell} P(0,\ell) \left[\Pi_{t'=0}^{t-1} W(t')\right]_{\ell k} = \sum_{\ell} P(0,\ell) \left[W(0,t-1)\right]_{\ell k} \tag{4.9}$$

which means that the transition matrix that makes the system go from iteration 0 to iteration t is the product $W(0,t-1)$

$$W(0,t-1) = W(0)W(1)\dots W(t-1)$$

of the transition matrices at each stage.

For the example of Figure 4.10, and a given temperature schedule $T(t)$, we can numerically compute the SA evolution at time t thanks to formula (4.9). The hope is that for t large enough, $P(t,k)$ is independent of $P(0,\ell)$ and $P(t,k)$ is zero for all k except for the global minimum.

Indeed, as shown in Figure 4.12, we find numerically that for large enough $T(0)$, the rows of $W(0,t-1)$ all have identical entries; thus, for instance,

$$[W(0,t-1)]_{\ell k} = [W(0,t-1)]_{1k}$$

and thus

$$\begin{aligned} P(t,k) &= \sum_{\ell} P(0,\ell) \left[W(0,t-1)\right]_{\ell k} \\ &= \sum_{\ell} P(0,\ell) \left[W(0,t-1)\right]_{1k} \\ &= \left[W(0,t-1)\right]_{1k} \sum_{\ell} P(0,\ell) \end{aligned} \tag{4.10}$$

which shows that

$$P(t,k) = [W(0,t-1)]_{1k}$$

where the index 1 has been arbitrarily chosen among the $n_x \times n_y$ possible values. On the other hand, if the initial temperature is too low, some regions of the search space will be more or less accessible depending on the starting point. It follows that $P(t)$ depends on $P(0)$, where $P(0)$ and $P(t)$ are the probability distributions at time 0 and time t respectively. The consequence is that it is important to start the simulated annealing search with a sufficiently high temperature.

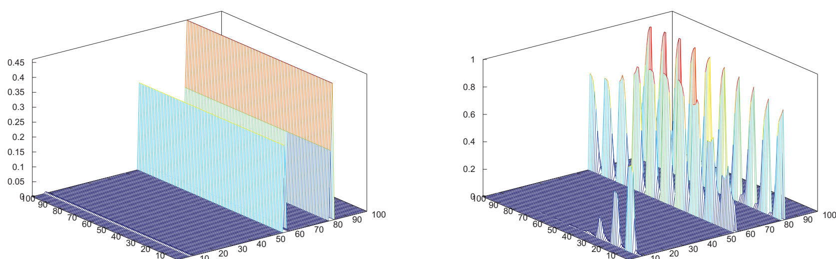


Fig. 4.12. Transition matrix $W(0,t-1)$ for two different values of the initial temperature after $t = 1{,}500$ iterations. On the left, $T(0) = 2$. Each matrix column has identical entries, which means that equilibrium has been reached and the probability distribution is independent of the starting state. On the right, $T(0) = 0.1$ and $P(t)$ depends on $P(0)$

When $P(t)$ does not depend on $P(0)$, we can represent $P(t,i)$ in the space **S**, taking into account that i and (x,y) are related by equations (4.5). This allows us to visualize the effects of a too-fast temperature schedule. In Figure 4.13 we show two simulated annealing runs with different temperature schedules. On the left, $T(t+1) = 0.998T(t)$ and we see that the global minimum at $(x,y) = (9,2)$ has a significantly higher probability of being found than the second deepest one at $(x,y) = (6,7)$. On the other hand, the evolution depicted in the right image with temperature schedule $T(t+1) = 0.95T(t)$ is too fast-paced. As a result, the two main minima are attainable with similar probabilities and some secondary minima are still present. This example shows numerically that choosing the right temperature schedule is very important in simulated annealing if we want to increase the likelihood of finding the global optimum.

Finally, Figure 4.14 shows the time evolution of $P(t)$ with a temperature schedule $T(t+1) = 0.999T(t)$. We can see that the global minimum is more likely to be selected and the width of the peaks decreases. However, even for times $t > 8{,}000$, the second deepest minimum still has a non-zero probability of being reached since, for large t, $T(t) = 0.999^t T(0)$ decreases more rapidly than the theoretical prescription $T(t) = C/\log(t)$. We see here that, as we already remarked, the slow temperature decrease that guarantees convergence of the process to the global optimum is not

acceptable in terms of computing time. The 8,000 iterations of the example mean a much larger computational effort than a straightforward exhaustive search.

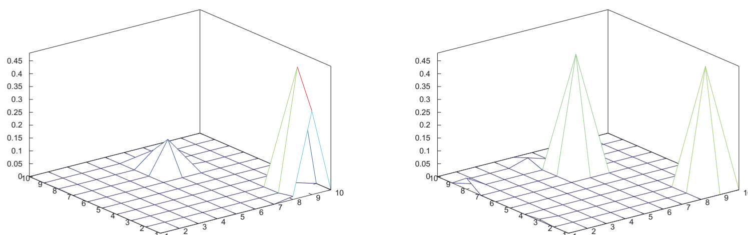


Fig. 4.13. On the left, the probability $P(t)$ of finding the current solution by simulated annealing after 3,000 iterations with $T(0) = 2$ and a temperature schedule $T(t + 1) = 0.998T(t)$, giving $T(3{,}000) = 0.0049$. On the right image, the temperature schedule is $T(t + 1) = 0.95T(t)$ and $T(3{,}000) = 2.96 \times 10^{-7}$

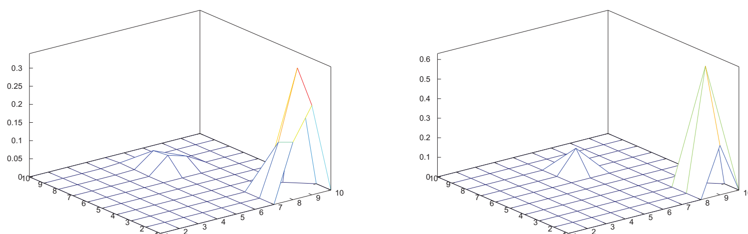


Fig. 4.14. Probability $P(t)$ of finding the current solution at point (x, y) after 4,000 iterations (left) and 8,000 iterations (right), with $T(0) = 2$ and $T(t+1) = 0.999T(t)$. Note the different vertical scales in the two images

4.6 Practical Guide

In this section we give a series of recommendations borrowed from reference [31] that should help implement simulated annealing for practical applications.

Problem coding.

This consists of choosing the associated search space and the data structures that allow us to describe and code the admissible configurations of the problem. It is also

necessary to define the elementary transformations and their effect on the chosen data structure. The choice of the representation must also be such that the variations ΔE following a move can be quickly computed. The possible constraints of the problem must be either easily translated into restrictions on the acceptable moves, or implemented by adding a suitable positive penalizing term to the fitness.

Choice of the initial temperature.

In order to start a simulated annealing search, an initial temperature T_0 must be specified. The following heuristic is useful to determine a suitable value for T_0:

- Perform 100 elementary transformations randomly starting from the initial configuration.
- Compute $\langle \Delta E \rangle$, that is, the average of the energy variations observed in these 100 moves.
- Choose an initial acceptance probability p_0 for worsening moves according to the assumed quality of the initial solution. Typically, $p_0 = 0.5$ if the quality is assumed to be average, and $p_0 = 0.2$ if it is assumed to be good.

  After that, T_0 can be computed such that

$$\exp\left(-\frac{\langle \Delta E \rangle}{T_0}\right) = p_0$$

which means that the temperature is high enough to allow the system to traverse energy barriers of size $\langle \Delta E \rangle$ with probability p_0. This idea is illustrated in Figure 4.15.

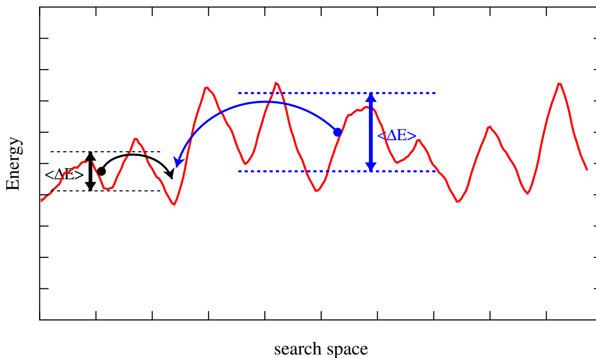


Fig. 4.15. Illustration of the choice of the initial temperature T_0. T_0 must be such that energy barriers separating the attraction basins of the initial condition and the optimal configuration can be overcome with a sufficiently high probability. The barrier heights depend on the quality of the initial solution

Temperature stages.

The temperature is modified when an equilibrium state is reached. In practice, we assume that an equilibrium state has been attained if $12N$ elementary transformations have been accepted over a total quantity of $100N$ tried moves. N is the number of degrees of freedom of the problem, i.e., the number of variables that define the solution.

Temperature schedule.

When the equilibrium as defined above has been reached, the system goes to another temperature stage by decreasing T according to a *geometric law*

$$T_{k+1} = 0.9T_k$$

where k is the stage number.

Termination condition.

The stopping condition of a simulated annealing is typically the following: if during the last three successive temperature stages the energy E didn't improve then the process is halted.

Validation.

In order to check that the optimal value found is sufficiently reliable, we can run SA again with a different initial condition, or with a different sequence of pseudorandom numbers. The final solution may change because it is not necessarily unique, but the optimal fitness value must be close from one execution to the next.

By applying this advice to the example with $N = 30$ we discussed in Section 4.3, we would typically have $T_0 = 0.65$ (assuming $p_0 = 0.2$), with temperature stage changes after $12 \times 30 = 360$ acceptances or $100 \times 30 = 3{,}000$ tried moves. In the example a rapid convergence was obtained with $T_0 = 0.1$ and with more frequent temperature changes (after 20 acceptances). The example also showed that starting from a higher $T_0 = 0.5$ and with temperature stage changes after 200 accepted moves also led to the global optimum, only using more computing time.

The values of the guiding parameters given in this section are "generous" enough to allow us to tackle more difficult practical problems than the very simple one treated in Section 4.3.

4.7 The Method of Parallel Tempering

The *parallel tempering* method finds its origin in the numerical simulation of protein folding [35]. The idea consists of considering several Monte Carlo simulations of proteins at similar temperatures. If the temperatures are close enough to each

other, the probability distributions will overlap to some extent and it will be possible to exchange two configurations obtained in two simulations at different but close temperatures. This approach improves the sampling of the configuration space and prevents the system from getting stuck in a local energy minimum.

The idea can be adapted to the metaheuristics framework in the form of a parallel simulated annealing scheme. To this end, we consider M replicas of the problem, each at a different temperature T_i where i is the replica index. In parallel tempering, differently from SA, T_i is held constant during the search process. However, some adjustments may occur during the run, as explained later.

The name "parallel tempering" comes from the process called *tempering*, which consists of heating and cooling some material several times. The process is notably used in the production of chocolate.

The key idea in parallel tempering is that a constant-temperature annealing at high temperature will sample the search space in a coarse-grained fashion, while a simultaneous process at low temperature will explore a reduced region but runs the risk of being blocked into a local fitness optimum. To take advantage of these two behaviors, which clearly correspond to diversification and intensification, the current solutions of the two processes are exchanged from time to time. This procedure is illustrated in Figure 4.16. Roughly speaking, instead of running a single simulated annealing with a given cooling schedule, we now have several of them that together cover the whole temperature range. Thus, the temperature variation here is between systems, rather than during time in a single system.

More precisely, the exchange of two configurations C_i and C_j is possible only if T_i and T_j are close enough to each other ($j = i \pm 1$), as suggested by Figure 4.17. The two energy configurations E_i and E_j are exchanged with probability

$$p = \min(1, e^{-\Delta_{ij}}) \tag{4.11}$$

where Δ_{ij} is defined as

$$\Delta_{ij} = (E_i - E_j)\left(\frac{1}{T_j} - \frac{1}{T_i}\right) \tag{4.12}$$

This relationship is symmetric with respect to the exchange of i and j. Moreover, we see that if the system at high temperature, say system j, has a current configuration of energy E_j lower than the system at lower temperature T_i, then $E_i - E_j > 0$, $1/T_j - 1/T_i < 0$, $\Delta_{ij} < 0$, and the exchange probability p equals one. This means that the natural tendency is for good solutions to go from a system at high temperature to a system at low temperature. The reverse is also possible, but with a lower probability, the more so the higher the energy variation and the higher the temperature difference between T_i and T_j.

Parallel tempering amounts to the execution of M different simulated annealing processes, each one at a given constant temperature. If the SAs are run in parallel on M processors, there is almost no time overhead due to process communication to exchange solutions as this phase requires little time. With respect to the execution of M independent standard SAs of which we keep the best result found, it has been

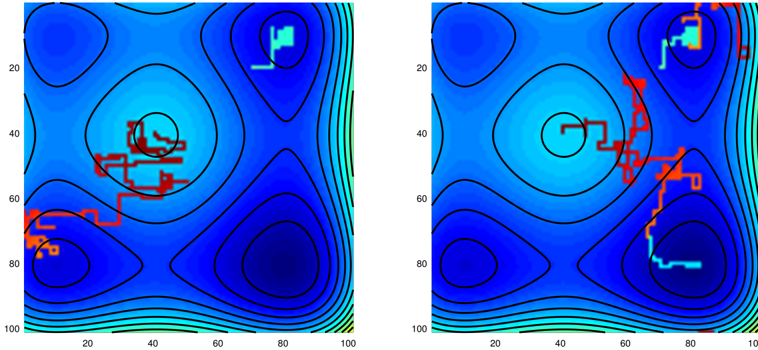


Fig. 4.16. Illustration of the parallel tempering principle. The energy landscape is represented by the blue tones and the level curves. The energy minimum is in the dark zone at the bottom right. The other dark zone at the top right of the image is a local minimum. The light blue trajectory corresponds to an annealing at low temperature, while the red one represents a high-temperature annealing. In the left image, the low-temperature annealing is stuck at a local minimum, while the high-temperature process explores the space widely but without finding a better energy region than the low-temperature one. There are no exchanges of trajectories. On the other hand, in the right image, the red trajectory enters a low-energy zone by chance. Now, after configuration exchange, the light blue annealing quickly reaches the global minimum. Because of its higher temperature, the red annealing doesn't get stuck in the local minimum

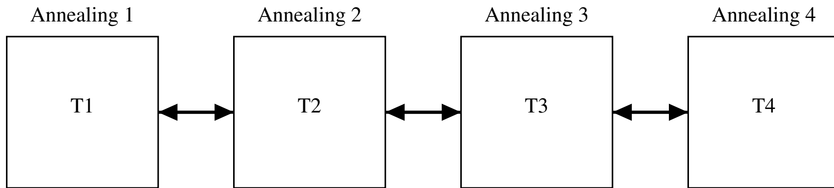


Fig. 4.17. Parallel tempering amounts to running M annealing processes in parallel, each one at constant temperature $T_1 < T_2 < \ldots < T_M$. The configuration exchanges take place between processes at similar temperatures, as suggested in the image by the arrows between the $M = 4$ replicas

observed that, in general, parallel tempering converges to a good solution faster. The reason is the communication of information between the systems, which happens in a manner that conceptually resembles the exchange of information between populations in a multipopulation evolutionary system (see Chapter 9).

Figure 4.18 shows a possible scenario of the exchange of configurations between neighboring systems (in the sense of their annealing temperature). There are, however, several guiding parameters that must be set in a suitable way:

- The number M of replicas is often chosen to be $M = \sqrt{N}$, where N is the problem size, i.e., the number of degrees of freedom.

- The frequency of configuration exchanges is often based on the annealing process state, and it is done when the two replicas have reached a stationary state.
- The number of different temperatures is chosen such that the SA at the highest temperature samples the space widely enough, together with the requirement that the temperature levels are close enough.

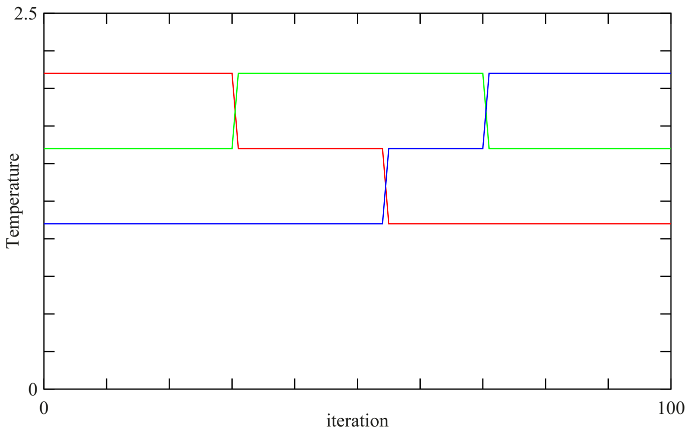


Fig. 4.18. Graphical illustration of configuration exchanges between SA processes at similar temperatures. The colors allow us to follow the exchanges between parallel processes

Finally, let's note that the replicas' temperatures can be adjusted if the exchange rates between neighbors are judged to be either too rare or too frequent. For example, if this rate goes below 0.5%, all temperatures are decreased by an amount $\Delta T = 0.1(T_{i+1} - T_i)$. In the opposite case, for example a frequency larger than 2%, all temperatures are increased by ΔT.

5

The Ant Colony Method

5.1 Motivation

The metaheuristic called the *ant colony* method has been inspired by entomology, the science of insect behavior. An interesting observation is that ants are apparently capable of solving what we would call optimization problems, such as finding a short path between the nest and a source of food. This result will be discussed in detail in Section 5.2. The ants' ability to collectively find optimal or nearly optimal solutions for their development and survival is witnessed by their biomass, which is estimated to be similar to that of humans. This also means that, being much lighter than humans, their number must be huge; it has indeed been estimated to be around 10^{16} on Earth.

Ants, in spite of their individually limited capabilities, seem to be able to collaborate in solving problems that are out of reach for any single ant. We speak in this case of the *emergence* of a recognizable collective behavior in which the whole is more than the sum of the individual parts. The term *swarm intelligence* is also used, especially in connection with other insects such as bees. Moreover, in certain tasks that ants perform, such as the construction of a cemetery for dead ants [19], there seems to be no centralized control but, despite the short-sighted local vision of each single insect, a global coherence does emerge. This kind of phenomenon is typical of *complex systems* and one also speaks of *self-organization* in this context.

The absence of a centralized control is called *heterarchy* in biology, as opposed to a hierarchy in which an increasing degree of global knowledge is assumed as we climb towards the top of the organizational structure. The self-organization of ant populations is the key to robustness and flexibility of the problem-solving processes. In particular, the system appears to be highly fault-tolerant, as it continues to function with almost no disruption even when ants disappear or do a wrong action, and it quickly adapts to a change of a problem's constraints.

The above considerations, together with the parallel nature of the ants' actions, led Dorigo, Maniezzo, and Colorni [30] to propose in 1992 an optimization algorithm inspired by the ants' ability to solve a global problem with only a local appreciation of it. The key ingredient of the idea is the use of artificial *pheromone* (see below)

© Springer Nature Switzerland AG 2018

B. Chopard, M. Tomassini, *An Introduction to Metaheuristics for Optimization*,
Natural Computing Series, https://doi.org/10.1007/978-3-319-93073-2_5

to mark promising solutions, thus stimulating other ants to follow and explore the corresponding region.

5.2 Pheromones: a Natural Method of Optimization

To carry out a collective task, ants must be able to communicate in some way. Entomologists have known for a long time that ants deposit a chemical substance called pheromone when they move. These pheromone trails are then used to attract, or guide, other ants along the trail. This process of indirect coordination is known in biology as *stigmergy* or *chemotaxis*: ants that perceive the smell of pheromones in their environment orient their movements towards the region were the presence of pheromone is high. Pheromones do not last forever however; after a certain time they evaporate thus, "erasing" a path that is not reinforced by the continuous passage of other ants.

The efficiency of such a communication tool for finding the global optimal solution to a problem is illustrated by the following experiment carried out by Goss et al. in 1989 [38] with true ants (Linepithema humile, Argentine ant).

Figure 5.1 defines the experimental setting. A food source is connected to an ant nest by tubes, which provide the ants with paths of different lengths. What was observed is that, at the beginning, ants leaving their nest in search of food distribute randomly and nearly equally on all possible paths. However, after a certain adaptation time, almost all ants tend to follow the same trail, which also happens to be the shortest one. The histogram of Figure 5.2 shows that between 80 and 100% of the ants end up finding the shortest path in about 90% of the experiments. In about 10% of the experiments only 60 to 80% of the ants follow the optimal trail.

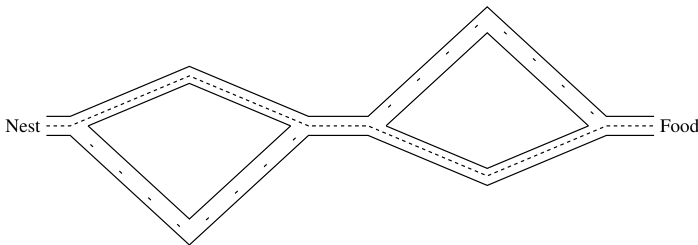


Fig. 5.1. Schematic representation of Goss et al.'s experiment to show the ants' ability to find the shortest path between the nest and a food source. Dots stand for the presence of ants along the trails. The image suggest that, even if most ants are using the shortest path, there are always some that go the wrong way

A possible explanation of these results could be the following: at the beginning there is no pheromone on the paths connecting the nest and the food. Ants have no signals to rely upon and thus they choose the branches randomly. But while moving

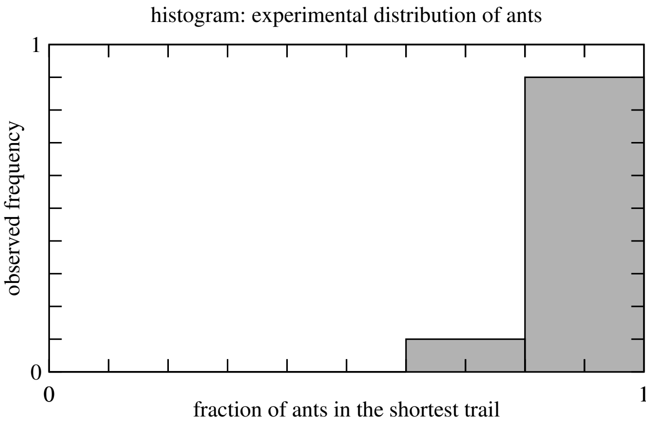


Fig. 5.2. Results of Goss et al.'s experiment. The histogram gives the observed frequency of ants having found the shortest path over several repetitions of the experiment

they start deposing pheromones to mark the path. The ants that have by chance chosen the shortest path will be the first to reach the food and to carry it back to the nest. They will return using the same path, which is the only one that is heavily marked. By doing so, they will depose more pheromone, thus amplifying the attractiveness of the path.

The ants that arrive at the destination later following other, possibly longer trails will discover the shortest path on their way back to the nest because it will be more heavily marked. By taking that trail, they will depose more pheromone, thus strengthening the path still further. After a while, most of the ants will have converged on the optimal path.

This plausible scenario has been confirmed by a simpler experiment that can be described by a mathematical model using the amplification of the pheromone trail. The experiment was performed in 1990 by Deneubourg et al. [29], according to the schema of Figure 5.3. In this case, the nest and the food are accessible through two paths of the same length. Ants thus have no advantage in choosing one path rather than the other. However, here too, after a transient period during which ants use the two branches equally, almost all of them end up choosing one of the two paths.

The results of a typical experiment are shown in Figure 5.4, in which the percentage of occupation of the two paths is given as a function of time. It appears that, after a few minutes hesitating over the direction to take, a decisive fluctuation causes the ants to follow the upper trail, and the pheromone reinforcement that follows is enough to attract the other ants to this path. In this experiment, it is the upper trail that ends up being chosen, but it is clear that the argument is also valid for the other trail and in other experiments the latter will be chosen instead. This experiment shows that the final choice is indeed a collective effect and does not only depend on the length of the path. If the latter was the only crucial factor, then both branches should be used during the experiment.

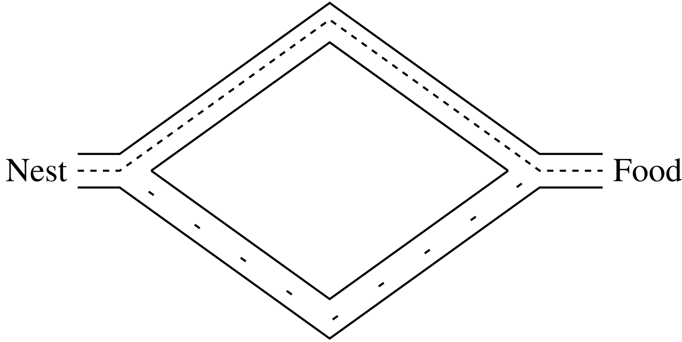


Fig. 5.3. Schematic setting of the Deneubourg et al.'s experiment. Two paths of identical length are available to the ants. One of them ends up being chosen

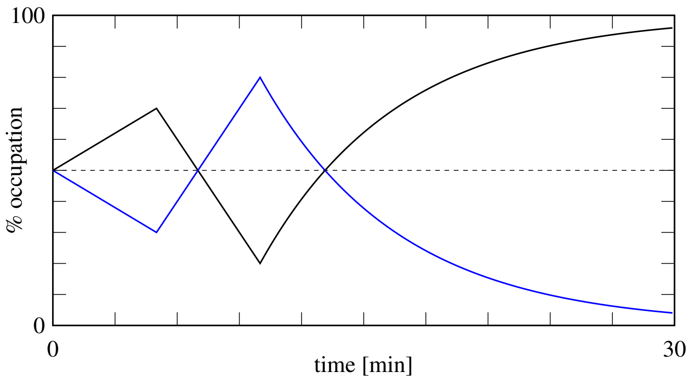


Fig. 5.4. Percentage of occupation of the two paths by the ants during the experiment. The blue curve corresponds to the lower path in Figure 5.3, the black curve represents the upper one. The sum of the values on the two curves at any time is 100% since ants are necessarily on one path or the other

Deneubourg et al. proposed a mathematical model of their experiment [29]. They assumed that the probability of choosing one of the two branches depends on the quantity of pheromones that have been deposited by all the ants that have gone through each one of them since the beginning of the experiment.

Let m be the number of ants that have already transited through the system by one or the other of the two paths. We denote by U_m and L_m the number of ants that have chosen the upper branch and the lower branch respectively. It is assumed that the probability $P_U(m+1)$ that the ant $(m+1)$ chooses the upper branch is

$$P_U(m+1) = \frac{(U_m + k)^h}{(U_m + k)^h + (L_m + k)^h} \tag{5.1}$$

where k and h are parameters to be specified. The probability $P_L(m+1)$ according to which the ant chooses the lower branch is symmetrically given by

$$P_L(m+1) = \frac{(L_m + k)^h}{(U_m + k)^h + (L_m + k)^h} \tag{5.2}$$

and, of course, $P_U(m+1)+P_L(m+1) = 1$. These expressions suggest that ants will choose their path as a function of the fraction of pheromone that has been deposited on each path since the beginning of the experiment.

Deneubourg et al.'s measures show that the above formulas represent the observed behavior quite well by choosing $k = 20$ and $h = 2$.

A possible interpretation of the parameters k and h might be the following. If U_m and L_m are small with respect to k, the probability of choosing the upper branch is close to 1/2. Therefore, k is of the order of the number of ants that must initially run through the system before the pheromone track becomes sufficiently selective. The h coefficient indicates the ants' sensitivity to the quantity of pheromone deposited. Since $h \neq 1$ the sensitivity is non-linear with respect to the quantity of pheromone.

5.3 Numerical Simulation of Ant Movements

The results reported in the previous section suggest that an ant chooses its way in a probabilistic fashion, and that the probability depends on the amount of pheromone deposited on each possible path by the ants that have gone through the system previously. Equations (5.1) and (5.2) are a quantitative attempt to estimate these probabilities.

We might nevertheless ask whether these hypotheses are really sufficient to explain the observation that ants find the shortest paths between their nest and food sources. The hypothesis was made that the shortest path emerges because the first ant to reach the food is the one that, by chance, used the shortest path. This same ant will also be the first one to return to the nest, thus reinforcing its own path. This shortest path will then become more attractive to the ants that will arrive later at the food source.

To check whether these ideas are sufficient in themselves to explain the phenomenon, and in the absence of a rigorous mathematical model, we can numerically simulate the behavior of artificial ants obeying the proposed principles. Here we will consider a *discrete-event simulation* in which the evolution of a system is modeled as a discrete sequence of events in time. If we consider the situation schematized in Fig. 5.5, the typical events are the following: (1) an ant leaves the nest; (2) an ant reaches the food either from the upper path or from the lower one; (3) an ant leaves the food and travels back to the nest; (4) an ant reaches the nest either from the upper path or from the lower path.

Each event is characterized by the time at which it takes place and by an associated action. For instance, the action associated with the event "leave the nest" calls for (a) choosing a path according to the quantity of pheromone already present on the

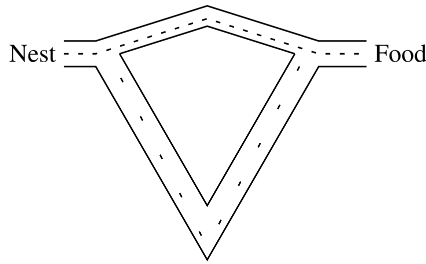


Fig. 5.5. Geometry of the paths considered in the discrete-event simulation of the ants' movement. The upper branch is half as long as the lower branch

two branches (probabilities (5.1) and (5.2) will be used for this choice); (b) adding some pheromone to the chosen branch; (c) going along the path and creating a new event "arrived at the nest by the upper/lower branch," with a total time given by the present time plus the time needed to go along the path. This time will be twice as large for the lower path than for the upper path.

In a discrete-event simulation, the next event to take place is the one whose time is the smallest among all the events not yet realized. In this way, the events caused by the ants choosing the shortest path will take place before those associated with the ants choosing the longest path. One thus hopes that this mechanism will be sufficient to asymmetrically strengthen the concentration of pheromone on the shorter path, in order to definitely bias the ants' decision towards the optimal solution.

Figures 5.6 and 5.7 present the results of the simulation with a total of 500 ants, which leave the nest at a rate of 2 ants/second, choose one of the two paths, reach the food, and go back to the nest. We assume that the short path takes $\Delta t = 5$ seconds to be traveled, whereas the long one takes $\Delta t = 10$ seconds. As explained before, at each branch, a direction is chosen probabilistically according to the quantity of pheromone placed at the path entrance. The choice of the branch creates a new event in the simulation, which consists of exiting the chosen branch after a time Δt.

Figure 5.6 shows, for a particular simulation, the number of ants, as a function of time, that are either on the short path or on the long path. Ants leave the nest at $t = 0$, one every half second, and their number increases in both branches. After approximately 30 seconds, the shorter path becomes more attractive and, as a consequence, the density of ants on the long path strongly decreases. After about 250 seconds all 500 ants have returned to the nest and the occupation of the paths goes to zero. The dashed lines indicate the number of ants that would be on each path if it were the only one. For example, consider the shorter path, which takes 5 seconds to traverse. Given that two ants leave the nest every second, there would then be 10 ants on average in each direction, thus a total of 20 ants, on the shorter path. Similarly, for the long path, we would expect to find 40 ants. However, the simulation shows an average o less than five, in agreement with the fact that most ants take the other path after a transitional period.

Although the previous simulation shows quite convincingly that the shorter path is indeed selected, this is not the case when the numerical experiment is repeated. Figure 5.7 shows the histogram, averaged over 100 runs, of the fraction of ants that have taken the shorter path, over the entire simulation time. The results are less clear-cut than in Goss et al.'s experiment (see Figure 5.2); however, the asymmetry favoring the shorter path is statistically clearly visible.

The following parameter values have been used in the simulations: $k = 30$ and $h = 2$ for computing the probabilities (5.1) and (5.2), and a pheromone evaporation rate of 0.01/s.

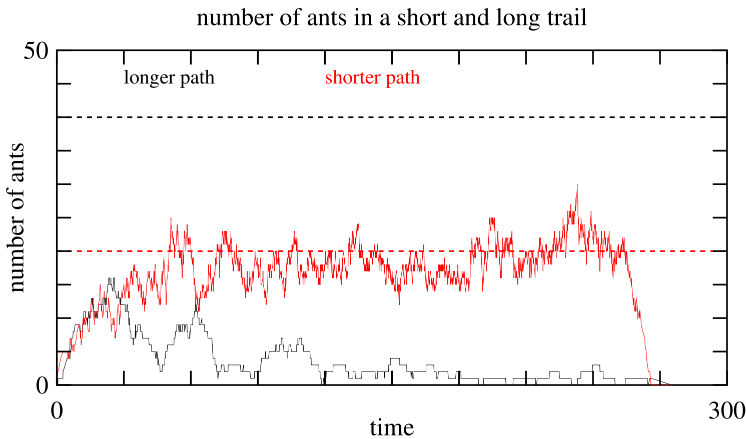


Fig. 5.6. Results of an instance of the discrete-event simulation where ants clearly find the shortest path. Dotted lines show the expected number of ants in each branch if it were the only one available

5.4 Ant-Based Optimisation Algorithm

In Section 5.5 we shall give the classical formulation of "ant" algorithms for combinatorial optimization as applied to the traveling salesman problem. For the time being, we present a simpler version that explores a one-dimensional search space $\mathbf{S} = \{0, 1, \ldots, 2^k - 1\}$, where k is an integer that defines the problem size. Each element $\mathbf{x} \in \mathbf{S}$ is a k-bit string $\mathbf{x} = x_0 x_1 \ldots x_{k-1}$. For example, for $k = 3$ we would have

$$\mathbf{S} = \{000, 001, 010, 011, 100, 101, 110, 111\}.$$

This space can be represented as a binary tree in which, at each level ℓ of the tree, we can choose 0 or 1 as the value of x_ℓ. A solution $\mathbf{x} \in \mathbf{S}$ is thus a path from the root of the tree to a leaf, as illustrated in Figure 5.8 for $k = 6$. In this figure we

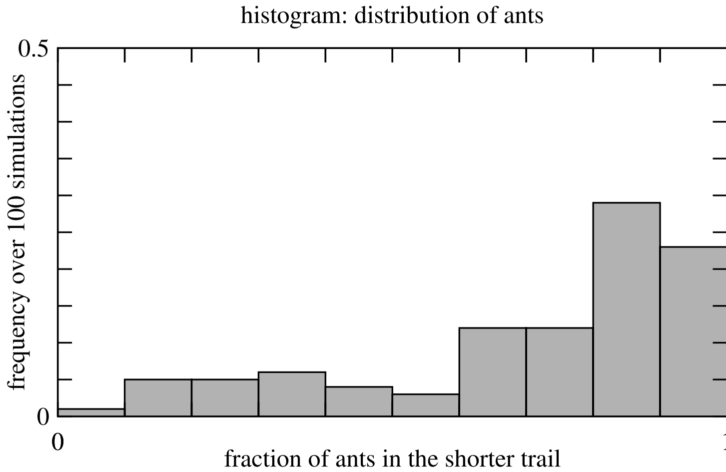


Fig. 5.7. Fraction of ants that find the shorter path averaged over 100 simulations. We see that, although this is not always the case, in more than 70% of the cases, more than half of the ants have found the shorter path

see in grey the 64 possible paths, each one corresponding to a particular solution in $\mathbf{S} = \{0, \ldots 63\}$. The heavier paths in black are six particular paths $\mathbf{x}_1 = 000110$, $\mathbf{x}_2 = 001000$, $\mathbf{x}_3 = 001110$, $\mathbf{x}_4 = 100011$, $\mathbf{x}_5 = 100101$, and $\mathbf{x}_6 = 110110$.

We then make the hypothesis that these paths are the result of the movements of artificial ants, starting from the root and reaching the leaves, where some food is supposed to be found. We can also imagine, for the sake of the example, that the quality of the food at the leaves is defined by a fitness function f which depends on the particular "site" in $\mathbf{S}$. In the example of Figure 5.8, the fitness function has been arbitrarily chosen to be

$$f(\mathbf{x}) = 1 + \frac{\mathbf{x}}{63} \cos\left(2\pi \frac{\mathbf{x}}{63}\right)$$

and it is graphically represented at the bottom of the figure.

We can now propose the following "ants" algorithm to explore $\mathbf{S}$ and search for the maximum of f, which is placed at $\mathbf{x} = 111111 = 63$ in this example, where the function value is $f(63) = 2$.

At each iteration, we let m ants explore the 2^k possible paths in the tree. Initially, we assume that all paths are equiprobable. This means that there are no pheromone traces on any trail yet. Thus, when it arrives at a branching point, an ant chooses the left or the right branch uniformly at random. When it arrives at leaf of the tree, it observes the fitness associated with the particular leaf it has reached. This value is interpreted as a pheromone quantity, which is then added to each branch of the particular path that has been followed.

From the ants' point of view, we may imagine that they deposit the pheromone on each path segment on their way back to the tree root, following in reverse the

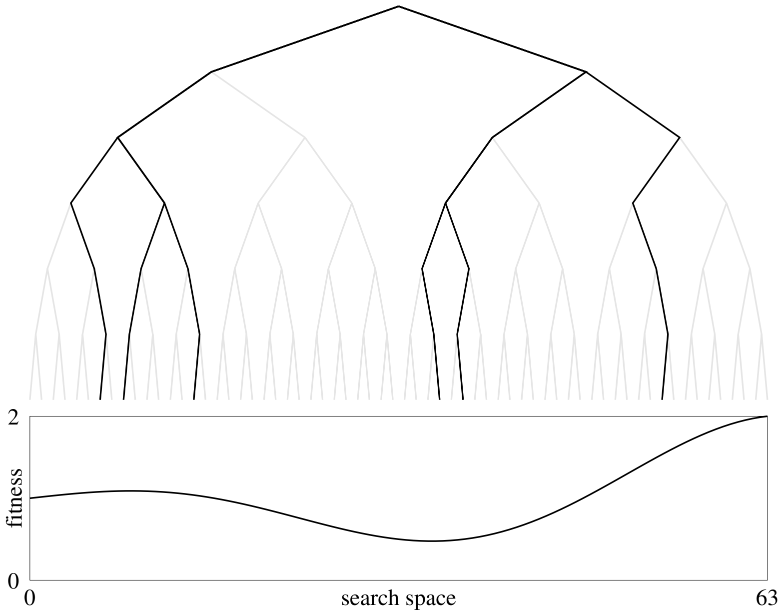


Fig. 5.8. Top: binary tree to explore the search space **S**. Bottom: fitness values associated with each point in **S**

same path used to find the leaf. From the programming point of view it is easier to carry out this reinforcement phase in a global fashion, without asking whether ants really do it. The important point is to mark the paths proportionally to their quality. It is also important to note that path segments can be "shared" by several ants traveling to different destinations; in this case, the segment will be enhanced by each ant that is using it, in an additive way.

At the next iteration, m ants will again travel from the root to the leaves, but this time they will find a pheromone trace that will guide them at each branching point. The decision to follow a particular branch can be implemented in a number of ways. If τ_{left} and τ_{right} are the current quantities of pheromone in the left and right branches respectively, an ant will choose the left branch with probability $p_{left} = \tau_{left}/(\tau_{left} + \tau_{right})$ or the right branch with probability $p_{right} = 1 - p_{left}$. To avoid a branch that has never been traversed by any ant having $\tau = 0$, which would make its probability of being chosen to be zero when compared to a branch having $\tau \neq 0$, all pheromones are initialized to a constant value τ_0, making the choice of a left or right branch in the first iteration equally likely, as required.

Another possibility for guiding the ants along the pheromone trails is to choose with probability q the better of the two branches, and to apply the method just described with probability $1 - q$. It is this last version that has been used to produce the trajectories shown in Figure 5.9. The four iterations that follow the one depicted in Figure 5.8 are indicated. The grey level of each tree node stands for the quantity

of pheromone on the incident branch: the darker the node, the more pheromone deposited. Now it is clearly seen that the ants' movement is no longer random; rather, it is biased towards the paths that lead to high-fitness regions. That said, statistically speaking, other paths are also explored, and this exploration can potentially lead to even better regions of the search space.

In this algorithm the parameters have been set empirically by trial and error as follows: $\tau_0 = 0.1$, $q = 0.8$, and a pheromone evaporation rate equal to a half of the pheromone deposited in each iteration. Furthermore, the best path found in each iteration is rewarded with a quantity equal to twice its fitness value.

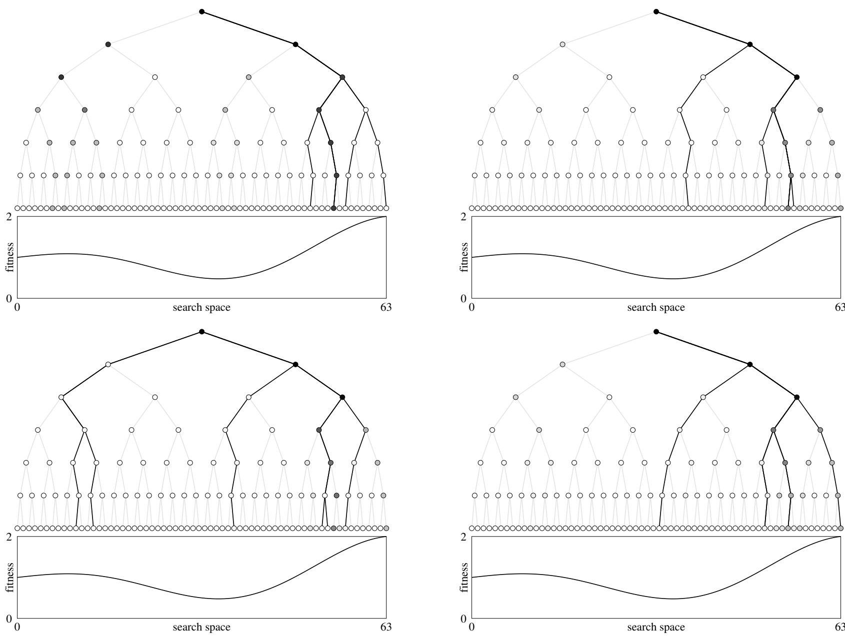


Fig. 5.9. Iterations number 2, 3, 4, and 5 in the exploration of the search space $\mathbf{S}$ by six ants starting from the pheromone trail created at iteration 1, shown in Figure 5.8

In Figure 5.9 it can be seen that the optimal solution $\mathbf{x} = 111111$ has been found in iterations 2 and 5 but it disappeared in iterations 3 and 4. This shows that it is fundamental to keep track of the best-so-far solution at all times since there is no guarantee that it will be present in the last iteration. As in other metaheuristics, there is also the question of how to set the termination condition of the algorithm. Here it was decided to stop after six iterations. The reason for such a choice is to limit the computational effort to a value smaller than the exhaustive enumeration of $\mathbf{S}$, which contains 64 points. With six ants and six iterations, the maximum number of points explored is 36.

In spite of the reduced computational effort, the optimal solution has been obtained. Was this due to chance? Repeating the search several times may shed some light on the issue. For 10 independent runs of the algorithm we found the performances reported in Table 5.1. We see that seven times out of ten the optimal solution has been found. This ratio is called the *success rate* or *success probability*. It is also apparent from Table 5.1 that the success rate increases if we are ready to give up something in terms of solution quality; in nine executions out of ten the best fitness is within 3% of the optimal fitness.

computational effort	$6 \times 6 = 36$
success rate	7/10
success rate within 3% of the optimum	9/10

Table 5.1. Performances of the simple "ants" algorithm on a problem of size $k = 6$. Values are averages over 10 executions, each with six ants and six iterations

It is interesting to compare the 70% success rate obtained with this simple algorithm to a random search. The probability of finding the global optimum in a random trial is $p = 1/64$. In 36 random trials, the probability of finding it at least once is $1-(1-p)^{36} = 0.43$, that is, one minus the probability of never finding it in 36 trials. As expected, even a simple metaheuristic turns out to be better than random search. The concepts briefly introduced here to discuss the performance of metaheuristics will be discussed in detail in Chapter 11.

To conclude the analysis of this toy problem, Figure 5.10 gives the empirical probability, or frequency, of each of the 64 possible paths after 100 iterations with 6,400 ants. The computational effort is evidently disproportionate with respect to the problem difficulty. But the goal here is to illustrate the convergence of the probability of attaining any given point in the search space. We see that the two maxima are clearly visible, but with a probability that is higher for the global optimum. There are also non-optimal paths whose probability is non-vanishing. It is unlikely that they would completely disappear if we increased the number of iterations since there would always be ants depositing some pheromone on these paths. However, with a probability of about 0.3 of reaching the global optimum (see Figure 5.10), the probability that at least one ant among the 6,400 will find it is essentially equal to one.

We remark that the simple "ants" algorithm is a population-based metaheuristic since there are m possible solutions at each iteration. Thus, formally, the search space is the cartesian product $\mathbf{S}^m$. The next iteration is a new set of m solutions that can be considered as a neighbor of the previous set, if one is to be faithful to the basic metaheuristic concept of going from neighbor to neighbor in the search. In the present case, the neighbor is generated by a stochastic process based on the attractiveness of the paths starting from the m current solutions.

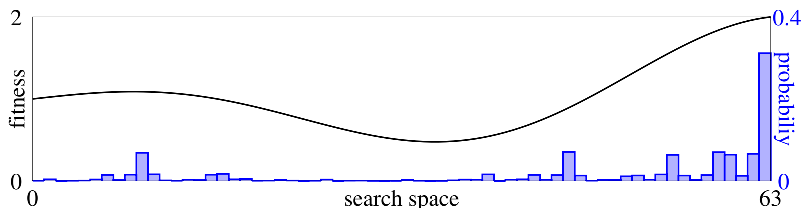


Fig. 5.10. Fitness function of the problem and probability of each of the 64 possible paths for exploring **S**. This empirical probability is obtained with 6,400 ants and corresponds to the paths chosen after 100 iterations of the algorithm

5.5 The "Ant System" and "Ant Colony System" Metaheuristics

The ant behavior described in Section 5.2 inspired Dorigo et al. to propose an optimization metaheuristic they called *Ant System* (AS) [30]. The metaheuristic was first formulated in the context of the traveling salesman problem, followed by quadratic assignment problems. Successively, the AS algorithm has been modified in its details to improve on some of its weaknesses. The resulting metaheuristic was called *Ant Colony System* (ACS). As is often the case with metaheuristics, there exist several variants and there is little in the way of theoretical arguments that would explain why one particular version performs better than another on a class of problems. In the next section, we will follow the description of the AS and ACS algorithms given in [31].

5.5.1 The AS Algorithm Applied to the Traveling Salesman Problem

The traveling salesman problem (*TSP* for short) calls for the shortest tour that passes exactly once through each of n cities and goes back to the starting city. This problem has already been mentioned several times, especially in Chapter 2.

The AS algorithm for solving the TSP can be formulated as follows. For the n cities, we consider the distances d_{ij} between each pair of cities i and j. Distances can be defined as being the usual Euclidean distances or in several other ways, depending on the type of problem. From the d_{ij} values, a quantity η_{ij} called *visibility* can be defined through the expression

$$\eta_{ij} = \frac{1}{d_{ij}} \tag{5.3}$$

Thus, a city j that is close to city i will be considered "visible", while a more distant one won't.

We consider m virtual ants. In each iteration, the ants will explore a possible path that goes through the n cities, according to the TSP definition. The m ants choose their path as a function of the quantity τ_{ij} of virtual pheromones that has been deposited on the route joining city i to city j as well as the visibility of city

j seen from city i. The quantity τ_{ij} is called the track *intensity* and it defines the attractiveness of edge (i, j).

With respect to the simple ant algorithm described in Section 5.4, the main difference here is that ants are given supplementary information, the visibility of the next town, which they use to guide their search.

After each iteration, the ants generate m new admissible solutions to the problem. The search space $\mathbf{S}$ is thus the cartesian product

$$\mathbf{S} = \underbrace{\mathbf{S}_n \times \ldots \times \mathbf{S}_n}_{m \text{ times}}$$

where $\mathbf{S}_n$ is the set of permutations of n objects.

As explained in the previous section, the neighborhood of the current solution, which consists of m possible tours, is not explicitly represented. Instead, the neighbor that will be chosen in the search space is built using a stochastic method based on the aggregated quality of the m paths obtained in the previous iteration.

In more concrete terms, let us consider the way in which ant k builds, in iteration $t + 1$, the k-th tour out of m tours of the new current solution. Let us suppose that ant k has reached city i after having visited a certain number of other cities. Let the set J denote the set of cities not yet visited by the ant. Among these cities, the ant will choose as the next stage in the tour city $j \in J$ with probability p_{ij}, which depends on the visibility η_{ij} of city j and on the *current intensity* $\tau_{ij}(t)$ of edge (i, j). Specifically, p_{ij} is defined as

$$p_{ij} = \begin{cases} \frac{[\tau_{ij}(t)]^\alpha [\eta_{ij}]^\beta}{\sum_{\ell \in J} [\tau_{i\ell}(t)]^\alpha [\eta_{i\ell}]^\beta} & \text{if } j \in J \\ 0 & \text{otherwise} \end{cases} \tag{5.4}$$

where α and β are control parameters whose values must be chosen in a suitable way. The above formula computes the transition probabilities, giving a weight to the j selection with respect to all the choices ℓ that are still available. With this construction, one also guarantees that an ant cannot visit the same city twice in the tour it builds in iteration $t + 1$.

It can be seen that a high value of the α parameter will attribute more weight to edges whose pheromone intensity is high. In this way, potentially promising edges will be favored in the exploration. This is tantamount to *intensifying* the good solutions already found. In contrast, a high β will favor local geographic information, independent of the importance that edge (i, j) may have in the global tour. This factor helps the *diversification* by trying new paths.

To complete the algorithm description, we must still say how the pheromone track is updated in iteration $t + 1$. Once the m ants have generated their respective tours T_k, $k = 1, \ldots m$, the pheromone intensity τ_{ij} on each edge is updated as follows:

$$\tau_{ij}(t + 1) = (1 - \rho)\tau_{ij}(t) + \Delta\tau_{ij}(t + 1) \tag{5.5}$$

where $\Delta\tau_{ij}(t+1)$ is the pheromone contribution resulting from the global quality of the m new tours and $\rho \in [0,1]$ is an *evaporation* factor that allows us to deconstruct a sub-optimal solution; such a solution can result from a chance process and may lead to a premature search convergence without evaporation.
The quantity $\Delta\tau_{ij}$ is obtained as the sum of the contributions of the m ants. We set

$$\Delta\tau_{ij}(t+1) = \sum_{k=1}^{m} \Delta\tau_{ij}^{(k)} \tag{5.6}$$

with

$$\Delta\tau_{ij}^{(k)} = \begin{cases} \frac{Q}{L_k} & \text{if } (i,j) \in T_k \\ 0 & \text{otherwise} \end{cases} \tag{5.7}$$

where Q is a new control parameter and L_k is the length of tour T_k associated with ant k. We see that only the edges effectively used by the ants are strengthened ((i,j) must belong to T_k). The enhancement is larger for edges that have been used by many ants and for edges that are part of a short tour, i.e., small L_k.

It is important to attribute a non-zero initial value to τ_{ij} in order for the probabilities p_{ij} to be correctly defined at iteration $t = 1$. In practice, a small pheromone value is chosen

$$\tau_{ij}(0) = \frac{1}{nL}$$

where L is an estimate of the TSP tour length, which can be obtained, for example, by running a greedy algorithm.

Suitable values of the parameters m, α, β, ρ, and Q that give satisfactory results for solving the TSP problem with the AS metaheuristic have been found empirically:

$$m = n, \quad \alpha = 1, \quad \beta = 5, \quad \rho = 0.5, \quad Q = 1$$

Finally, we note that in the AS algorithm the m ants build the m tours in iteration t in an independent manner. In fact, ant k chooses its path on the basis of the pheromone deposited in iteration $t-1$ and not as a function of the paths already chosen by the first $k-1$ ants in iteration t. As a consequence, the algorithm is easy to parallelize with a thread for each of the m ants.

5.5.2 The "Ant Colony System"

The algorithm of the previous section has been applied with success to the benchmark "Oliver30," a 30-city TSP [65]. The AS was able to find a better solution than the best one known at the time, which had been obtained with a genetic algorithm, with a computing time comparable to or shorter than that required by other methods. However, when tackling a larger problem, AS was unable to find the known optimum within the time allowed by the benchmark, although convergence to a good optimum was fast.

This stimulated several researchers to propose variants of the AS algorithm to improve its performance. The *Ant Colony System* (ACS algorithm modifies the AS in two ways: (1) the choice of the next city to visit, and (2) the pheromone track update. In practice, these two points are redefined in ACS as follows:

(1)

A new parameter q_0, whose value is between 0 and 1, is introduced such that in iteration $t+1$ ant k, having reached city i, chooses the next city j according to the rule

$$j = \begin{cases} \text{argmax}_{\ell \in J}[\tau_{i\ell}(t)\eta_{i\ell}^{\beta}] & \text{with probability } q_0 \\ u & \text{with probability } 1-q_0 \end{cases} \tag{5.8}$$

where

$$\text{argmax}_x f(x)$$

is, by definition, the argument that maximizes the function f.

The quantity $u \in J$ is a randomly chosen city in the set J of allowed cities which is drawn with probability p_{iu}

$$p_{iu} = \frac{\tau_{iu}(t)\eta_{iu}^{\beta}}{\sum_{\ell \in J} \tau_{i\ell}(t)\eta_{i\ell}^{\beta}} \tag{5.9}$$

Thus, with probability q_0 the algorithm *exploits* the known information as it chooses the best edge available. Otherwise, with probability $1-q_0$, the *exploration* of new paths is privileged.

(2)

The amount of pheromone deposited on the graph edges now evolves both locally and globally going from iteration t to $t+1$. After the passage of m ants, a local provision of pheromone is supplied. Each ant deposits a quantity of pheromone $\phi\tau_0$ on each edge (i,j) that it has traversed; at the same time, a fraction ϕ of the already present pheromone evaporates. We thus have

$$\tau'_{ij} = (1-\phi)\tau_{ij}(t) + \phi\tau_0 \tag{5.10}$$

Next, an amount of pheromone $\Delta\tau = 1/L_{min}$ is added only to the edges (i,j) of the *best* tour T_{best} among the m tours of iteration $t+1$. Simultaneously, a fraction ρ of pheromone evaporates from the edges of the best tour. Due to the global pheromone update, the τ'_{ij} obtained in the local pheromone update phase are corrected thus:

$$\tau_{ij}(t+1) = \begin{cases} (1-\rho)\tau'_{ij} + \rho\Delta\tau & \text{if } (i,j) \text{ belongs to the best tour} \\ \tau'_{ij} & \text{otherwise} \end{cases} \tag{5.11}$$

The aim here is to reinforce the best path the length of which is L_{min}.

5.5.3 Comments on the Solution of the TSP Problem

The traveling salesman problem has a long history of research and many algorithms have been developed to try to solve the largest instances possible. Since this emblematic problem has a large number of applications in many areas, the results of this research have often been applied to other domains. Metaheuristics are just one approach among the many that have been proposed.

Since the early 2000s, the *Concorde* algorithm, which is a TSP specialized code, allows us to find the optimal solution of problems with a large number of cities[1].

The algorithm, much more complex than any metaheuristic approach, is described in the book of Applegate et al. [7]. It is based on a linear programming formulation combined with the so-called *cutting-planes* method and uses *branch-and-bound* techniques. The program comprises about 130,000 lines of C code.

Concorde computing time can be significant for large difficult instances. For example, in 2006, a TSP instance with 85,900 cities needed 136 CPU years to be solved[2]. But computing time does not depend only on problem size; some city configurations are more difficult to solve than others. For instance, in [6] the authors mention the case of a 225-city problem (`ts225`) that took 439 seconds to solve, while another 1,002-city problem (`pr1002`) only took 95 seconds.

Let us remind the reader that *Concorde* is essentially based on linear programming, for which the simplex algorithm can take an exponential amount of time in the problem size in the worst case. Although polynomially bounded algorithms for linear programming do exist, they are often less efficient in practice than the simplex algorithm and the latter remains the method of choice. As a consequence, *Concorde* does not question the fact that TSP is NP-hard. Indeed, in many practical cases, notably many of those contained in the TSPLIB library [83], we do not face city positions that generate the most difficult instances.

In [3], the authors propose techniques for generating TSP instances that are difficult for *Concorde* to solve. The results indicate that for unusual city placements, such as those based on Lindenmayer systems (*L-Systems*) [70], which have a fractal structure, the computing time of *Concorde* may become unacceptable.

The use of a metaheuristic is thus justified for such pathological problems or, more generally, for large problems if a good solution obtained in reasonable time is considered sufficient. In Chapter 4 we saw that a standard simulated annealing approach was able to find a solution to a 500-city problem in less than one second, i.e., at least 30 times faster than *Concorde*, with a precision of 5%. The same SA solves a 10,000-city problem in 40 seconds with a precision of the same order. Moreover, there exist specialized metaheuristics for the TSP, offering much better performance than those presented in this book, both in computing time and solution quality. For example, some researchers work on methods that can deal with one million cities in one minute, with 5% precision [77].

[1] See, for instance, the sites `http://www.math.uwaterloo.ca/tsp/index.html` and `http://www.math.uwaterloo.ca/tsp/world/countries.html`

[2] `https://en.wikipedia.org/wiki/Travelling_salesman_problem`

6

Particle Swarm Optimization

6.1 The PSO Method

Inspired by animal behavior, Eberhart and Kennedy [49, 22] proposed in 1995 an optimization method called *Particle Swarm Optimization* (PSO). In this approach, a swarm of particles simultaneously explore a problem's search space with the goal of finding the global optimum configuration.

6.2 Principles of the Method

In PSO the *position* $\mathbf{x}_i$ of each particle i corresponds to a possible solution to the problem, with fitness $f(\mathbf{x}_i)$. In each iteration of the search algorithm the particles move as a function of their *velocity* $\mathbf{v}_i$. It is thus necessary that the structure of the search space allows such movement. For example, searching for the optimum of a continuous function in $\mathbb{R}^n$ offers such a possibility.

The particles' movement is similar to a flock of birds or a school of fish, or to a swarm of insects. In these examples, it is assumed that the animals move by following the individual in the group that knows the path to the optimum, perhaps a source of food. In addition, however, the individuals also follow their instinct and integrate the knowledge they have about the optimum into their movements.

In the PSO method two quantities $\mathbf{x}_i^{best}(t)$ and $\mathbf{B}(t)$ have to be defined and updated in each iteration. The first one, $\mathbf{x}_i^{best}(t)$, which is often called *particle-best*, corresponds to the best fitness point visited by particle i since the beginning of the search. The second quantity, $\mathbf{B}(t)$, called *global-best*, is the best fitness point reached by the population as a whole up to time step t:

$$\mathbf{B}(t) = \text{argmax}_{\mathbf{x}_i^{best}} f(\mathbf{x}_i^{best}(t))$$

In certain variants of PSO the *global-best* position $\mathbf{B}(t)$ is defined with respect to a sub-population to which a given individual belongs. The subgroup can be defined

© Springer Nature Switzerland AG 2018

B. Chopard, M. Tomassini, *An Introduction to Metaheuristics for Optimization*,
Natural Computing Series, https://doi.org/10.1007/978-3-319-93073-2_6

by a neighborhood relationship, either geographical or social. In this case, $\mathbf{B}$ will depend on i.

Therefore, as illustrated in Figure 6.1, the particles' movement in PSO is determined by three contributions. In the first place, there is a term accounting for the "inertia" of the particles: this term tends to keep them on their present trajectory. Second, they are attracted towards $\mathbf{B}(t)$, the global best. And third, they are also attracted towards their best fitness point $\mathbf{x}_i^{best}(t)$.

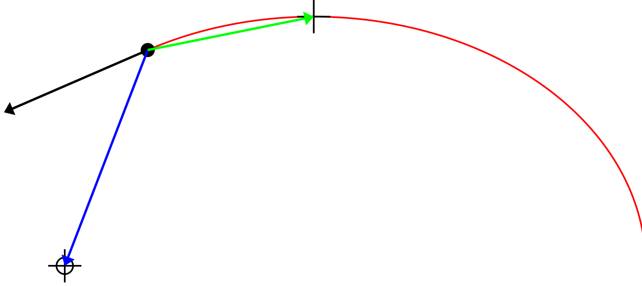


Fig. 6.1. The three forces acting on a PSO particle. In red, the particle's trajectory; in black, its present direction of movement; in blue, the attraction toward the *global-best*, and in green, the attraction towards the *particle-best*

Mathematically, the movement of a particle from one iteration to the next is described by the following formulas:

$$\begin{aligned} \mathbf{v}_i(t+1) &= \omega \mathbf{v}_i(t) + c_1 r_1(t+1)[\mathbf{x}_i^{best}(t) - \mathbf{x}_i(t)] \\ &\quad + c_2 r_2(t+1)[\mathbf{B}(t) - \mathbf{x}_i(t)] \\ \mathbf{x}_i(t+1) &= \mathbf{x}_i(t) + \mathbf{v}_i(t+1) \end{aligned} \tag{6.1}$$

where ω, c_1 and c_2 are constants to be specified, and r_1 and r_2 are pseudo-random numbers uniformly distributed in the interval $[0, 1]$. We remark that a different random number is used for each velocity component.

The c_1 parameter is called the *cognitive coefficient* since it reflects the individual's own "perception," and c_2 is called the *social coefficient*, since it takes into account the group's behavior. For example, $c_1 \approx c_2 \approx 2$ can be chosen. The ω parameter is the *inertia* constant, whose value is in general chosen as being slightly less than one.

Besides formulas (6.1), one must also impose the constraints that each velocity component must not be allowed to become arbitrarily large in absolute value. To this end, a $\mathbf{v}_{max}$ cutoff is prescribed. In the same way, the positions $\mathbf{x}_i$ are constrained to lie in a finite domain having diameter x_{max}.

In the initialization phase of the algorithm the particles are distributed in a uniform manner in the search domain and are given zero initial velocity. The relations above make it clear that it is necessary to work in a search space in which the arithmetic operations of sum and product make sense. If the problem variables are Boolean it is possible to temporarily work in real numbers and then round the results. The method can also be extended to combinatorial problems [56], although this is not the natural frame for this approach, which is clearly geared towards mathematical optimization.

Similarly to the ant colony method, PSO is a population-based metaheuristic. In each iteration, n candidate solutions are generated, one per particle, and the set of solutions is used to construct the next generation. PSO is characterized by rapid convergence speed. Its problem-solving capabilities are comparable to those of other metaheuristics such as ant colonies and evolutionary algorithms, with the advantage of simpler implementation and tuning. There have been several applications of the method [68, 75, 1], and it has proved very competitive in the field of optimization of difficult continuous functions.

6.3 How Does It Work?

In order to intuitively understand how and why PSO can find an optimum in a given fitness landscape, perhaps the global one, we shall consider a toy example. Figure 6.2 illustrates a PSO with two particles in a one-dimensional space ($x \in [-1, 8]$) with a simple fitness function $f(x)$ that is to be maximized. We find that, after a sufficient number of iterations, the two particules have traveled towards the maximum of f, as they should.

According to the general PSO equations (6.1), here the following system must be solved for $i = 1, 2$:

$$
\begin{aligned}
v_i(t+1) &= 0.9v_i(t) + [b_i(t) - x_i(t)] + [B(t) - x_i(t)] \\
x_i(t+1) &= x_i(t) + 0.2v_i(t+1)
\end{aligned}
$$

where t is the iteration number of the process, $b_i(t)$ is the *particle-best*, and $B(t)$ is the *global-best*.

Initially the two particles are at rest, randomly placed in the search space. Their $b_i(0)$ are thus their respective positions $x_i(0)$. The *global-best* $B(0)$ corresponds to the position of the "best" particle, represented here by the black one.

- The grey particle is attracted towards the black particle but the latter, being already the *global-best*, doesn't move.
- Since the grey particle is increasing its fitness, its *local-best* continues to be its current position, which doesn't modify its attraction towards the black particle.
- Thanks to its momentum, the grey particle will overtake the black one and will reach a better fitness value.
- In this way, the grey particle becomes the new *global-best*, slows down progressively and stops.

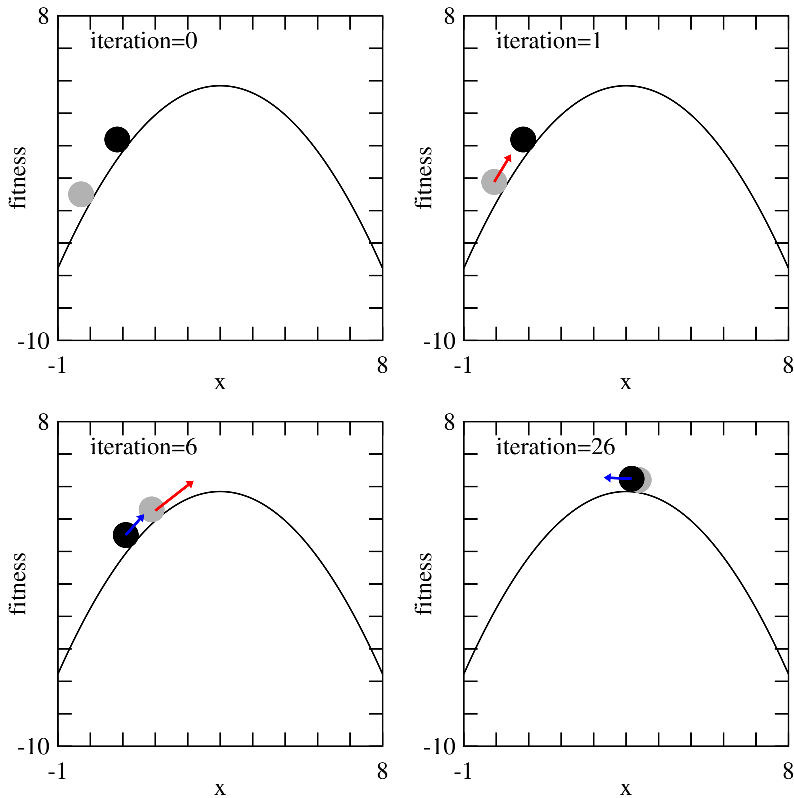


Fig. 6.2. PSO example with two particles in the one-dimensional space $x \in [-1, 8]$ with a parabolic fitness function of which the maximum is sought. To better illustrate the evolution, particles are shown here moving on the fitness curve; actually, they only move along the x axis

- Once the grey particle has passed it, the black particle starts moving towards the grey particle.

6.4 Two-Dimensional Examples

In this section we look at two examples of PSO in which several particles explore a relatively complex subspace of $\mathbb{R}^2$. For the sake of the numerical simulation, the continuous space has been discretized as a grid of 80×60 points. It is interesting to observe the trajectory of the moving particles and their approach to the global maximum. The problem is simple enough for an exhaustive search to be applied since there are only $80 \times 60 = 4{,}800$ points in the search space.

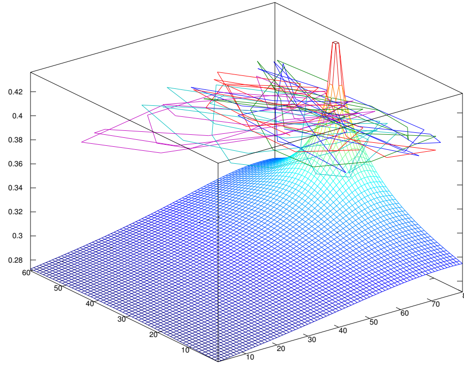


Fig. 6.3. An example of PSO in 2D on a single-maximum fitness landscape

The example, illustrated in Figure 6.3, has the following properties:

- Global optimum at: (75, 36); fitness value at the global optimum: 0.436,
- With five particles, 50 iterations, the best solution found in a single run (Figure 6.3) was

$$\mathbf{B} = (75, 39) \qquad f(\mathbf{B}) = 0.384$$

- With 20 particles, 100 iterations, the optimal solution was found in each run.
- We note that the particles are grouped around the maximum at the end.
- In this example $r_1 = r_2 = 1$ (see eq. 6.1), and the particles are reflected by the domain borders.

Figure 6.4 gives an example of a more difficult search space with several maxima. The global optimum is at $(22, 7)$, with a fitness value of 0.754. With 10 particles and 200 iterations the best solution found by PSO in one run was

$$\mathbf{B} = (23, 7) \qquad f(\mathbf{B}) = 0.74$$

This is very close to the global optimum. The computational effort can be estimated as the product of the number of particles times the number of iterations, that is $10 \times 200 = 2{,}000$, which is less than half the effort needed for an exhaustive search of the 4,800 points in the space.

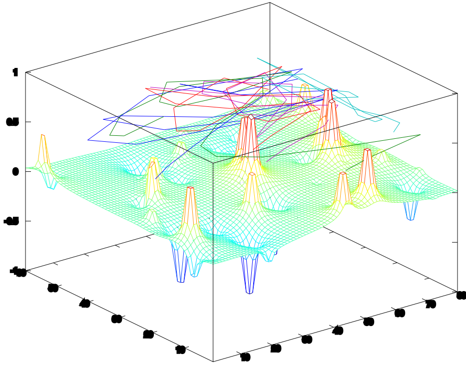


Fig. 6.4. An example of PSO in 2D with a multimodal fitness landscape

7

Fireflies, Cuckoos, and Lévy Flights

7.1 Introduction

In this chapter we shall present some newer metaheuristics for optimization that are loosely based on analogies with the biological world. In contrast with the metaheuristics described in the previous chapters, these algorithms have been around for a comparatively short time and it is difficult to know whether they are going to be as successful as more classical methods. Be that as it may, these metaheuristics contain some new elements that make them worth knowing. Some versions make use of *Lévy flights*, a probabilistic concept that, besides being useful in search, is interesting in itself too. Before getting into the main subject matter, we shall devote some space to an elementary introduction to some probability and stochastic processes concepts that will be needed later and that are of general interest in the fields of metaheuristics and computational science.

7.2 Central Limit Theorem and Lévy Distributions

The central limit theorem (CLT) is one of the most fundamental results in probability theory. Let (X_i) be a sequence of identically distributed and independent random variables, with mean m and finite variance σ^2, and let $S_n = \sum_{i=1}^{n} X_i$ be their sum. The theorem says that the sum

$$\frac{S_n - nm}{\sigma\sqrt{n}}$$

converges in law to the standard reduced normal distribution

$$\frac{S_n - nm}{\sigma\sqrt{n}} \to \mathcal{N}(0,1), \quad n \to \infty$$

A proof can be found in [15]. The theorem also holds under weaker conditions such as random variables that are not identically distributed, provided they have finite

© Springer Nature Switzerland AG 2018

B. Chopard, M. Tomassini, *An Introduction to Metaheuristics for Optimization*,
Natural Computing Series, https://doi.org/10.1007/978-3-319-93073-2_7

variances of the same order of magnitude, and for weakly correlated variables. Besides, although the result is an asymptotic one, in most cases $n \sim 30$ is sufficient for convergence to take place.

The importance of the central limit theorem cannot be overemphasized: it provides a convincing explanation for the appearance of the normal distribution in many important phenomena in which the sum of many elementary independent actions leads to a regular global behavior. Examples of phenomena that are ruled by the normal distribution are diffusion and Brownian motion, errors in measurement, the number of molecules in a gas in a container at a given temperature, white noise in signals, and many others.

However, sometimes the conditions required for the application of the CLT are not met, in particular for variables having a distribution with infinite mean or infinite variance. Power-law probability distributions of the type

$$P(x) \sim |x|^{-\alpha}$$

provide a common example. One may ask the question whether a result similar to the CLT exists for this class of distributions. The answer is positive and has been found by the mathematician P. Lévy [55]. Mathematical details, which are not elementary, can be found, for example, in Boccara's book [15]. Here we just summarize the main results. Lévy distributions are a class of probability distributions with the property of being "attractors" for sums of random variables with diverging variance. Power-laws are a particular case of this class; therefore, if $X_1, X_2, \ldots, X_n$ are distributed according to a power-law with infinite variance, their sum asymptotically converges to a Lévy law. One can also say that such random variables are "stable" under addition, exactly like the Gaussian case, except that now the attractor is a Lévy distribution, not the normal. The invariance property under addition is very useful in many disciplines that study rare events and large statistical deviations.

In order to illustrate the above ideas, Figure 7.1 shows an ordinary random walk and a Lévy walk in the unidimensional case.

In both cases the starting position is at the origin. In the standard random walk (magenta curve) 1 is added or subtracted to the current position in each time step with the same probability $p = 1/2$. On the y-axis we represent the current value of the sum $X(0)+X(1)+\ldots+X(t)$. In the case of a Lévy walk (black curve) each random step is drawn from the distribution $P(x) = \alpha x^{-(1+\alpha)}$ with $x > 1$ and $\alpha = 1.5$ (we refer the reader to Section 2.8 for the way to generate such values). The sign of the term to be added to the current position is positive or negative with the same probability $p = 1/2$. One sees clearly that while the random walk has a limited excursion, the Lévy walk shows a similar behavior except that some large fluctuations appear from time to time. The difference is even more striking in two dimensions. Figure 7.2 (upper image) shows an example of a random walk in the plane in which the new position is determined by drawing a random number from a normal distribution with zero mean and a given σ. The corresponding stochastic process is called "Brownian" and it was intensively studied at the end of the nineteenth century by Perrin, Bachelier, Einstein, and others.

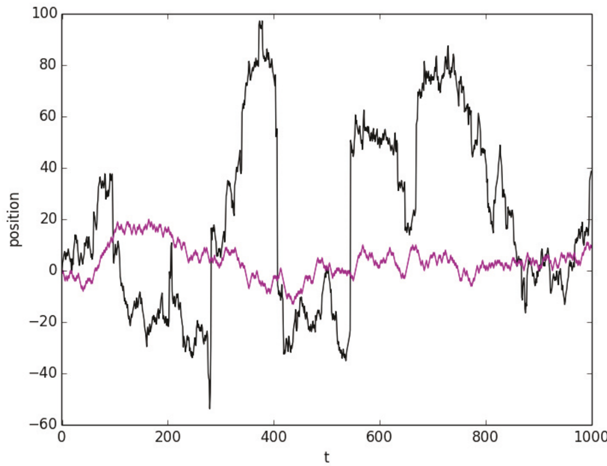


Fig. 7.1. The magenta curve represents on the y-axis the sum of random displacements on the line at time step t. The result is a random walk in one dimension. Displacements with respect to the current position on the line are either 1 or -1 with the same probability $1/2$. The black curve depicts the same sum but this time the single displacements are drawn from a power-law distribution with exponent $\alpha = 1.5$. The corresponding stochastic process is a Lévy walk

In both cases the starting point is the origin at $(0,0)$ and the process is stopped after $2{,}000$ time steps. Brownian motion is a Gaussian process: being drawn from a normal distribution, the random changes in direction are of similar size as the fluctuations around the mean are limited. According to the central limit theorem, the distribution of the sum of all those displacements is itself normal. The corresponding Lévy process is depicted in the lower image of figure 7.2. Here the random displacements are drawn from a power-law with exponent $\alpha = 1.8$. In the case of Lévy flights the process is very different: the probability of large movements is smaller but not negligible as in the Gaussian case. As a consequence, the rare events dominate the global behavior to a large extent.

The general properties of Lévy flights have been known for a long time but it is only recently that their usefulness in understanding a number of natural phenomena has been fully recognized. A nice summary can be found in [12], which is a short review of the subject that mentions the main references. In particular, it has been found that the movement patterns of several animal species do not follow a random walk; rather, they perform Lévy flights in which the displacement lengths d follow a power-law $P(d) \sim d^{-\alpha}$, as illustrated in the right image of Figure 7.2. These patterns of movement seem to provide an optimal foraging strategy if the exponent is suitably chosen, and this fact has found confirmation in the data coming from experimental field observations.

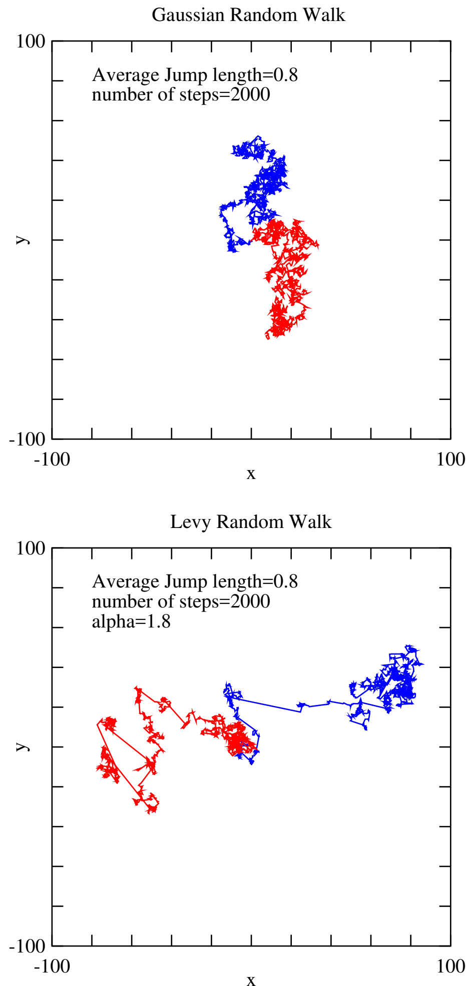


Fig. 7.2. A typical Brownian process is shown on the top panel, whereas a Lévy fly (see text) is illustrated on the bottom panel. The red and blue lines correspond to independent trajectories, showing that the typical features of these two stochastic processes are present in all instances. In this example we chose the parameters of the Brownian motion so that both random walks have the same average jump length

The results of this research have not gone undetected in the metaheuristics community and Lévy flights are now employed in single-trajectory searches, as well as in population-based search such as PSO. They help avoid search stagnation and allow us to overcome being trapped in local optima. There are many uses for Lévy flights in optimization metaheuristics but reasons of space prevent us from dealing with the

issue more deeply. In the rest of the chapter we shall see how these processes can be harnessed in two new metaheuristics: firefly and cuckoo search. The reader will find a more detailed description of those new metaheuristics in [87].

Note that there exist several other recent metaheuristics based on the collective behavior of insects or animals. As an example we just mention here the bee colony method [46, 87], the antlion optimization method (ALO) [61] and the Grey Wolf Optimizer (GWO) [62]. They will not be further discussed here for reasons of space.

7.3 Firefly Algorithm

This new metaheuristic for optimization has been proposed by Xin-She Yang [87]. The main ideas come from the biological behavior of fireflies but the algorithm is also clearly inspired by PSO methods. The metaheuristic is geared towards continuous mathematical optimization problems but there exist versions for discrete problems too.

Fireflies are insects that emit light to attract mates and prey. The degree of attraction is proportional to the intensity of the light source, and the metaphorical exploitation of this phenomenon is at the heart of the metaheuristic, as explained below.

The *Firefly* metaheuristic considers a colony of n virtual fireflies, identified by an index i between 1 and n. The fireflies are initially randomly distributed on a given search space $\mathbf{S}$. At iteration t of the search, firefly i occupies position $x_i(t) \in \mathbf{S}$ (for example $\mathbf{S} \subset \mathbb{R}^d$).

A given objective function f is assumed to be defined on the search space and the goal is to maximize (or minimize) f. To this end, each firefly i emits light with intensity I_i, which depends on the fitness of the search point occupied by i. Typically, $I_i(t)$ is set as follows:

$$I_i(t) = f(x_i(t)) \tag{7.1}$$

The algorithm has the following steps: at each iteration t it cycles over all firefly pairs (i, j) with $1 \leq i \leq n$ and $1 \leq j \leq n$ and compares their light intensities. If $I_i < I_j$ then firefly i moves towards firefly j according to the *perceived attractiveness* of j, a quantity that is defined below. Note that firefly i's position is immediately updated, which means that i might move several times, for example after comparison between the intensities of i and j and between i and k.

The strength of attraction A_{ij} of firefly i towards firefly j, which has more luminosity, corresponds to the intensity perceived by i. It is defined thus:

$$A_{ij} = \beta_0 \exp(-\gamma r_{ij}^2) \tag{7.2}$$

where r_{ij} is the distance, Euclidean or of another type, between x_i and x_j. Attraction thus decreases exponentially with increasing distance between two fireflies, an effect that simulates the fact that perceived intensity becomes weaker as the distance increases. The quantities β_0 and γ are suitable free parameters. Writing the argument of the exponential as $(r_{ij}/\gamma^{-1/2})^2$ one sees that $\gamma^{-1/2}$ provides a distance

scale. Typically, $\gamma \in [0.01, 100]$ is chosen depending on the unities of the x_i's. The β_0 parameter gives the attractiveness A_{ij} at zero distance and it is typically set at $\beta_0 = 1$.

The displacement of firefly i towards firefly j is defined by an attractive part determined by the relative light intensities, and a random part. The updated position x'_i is given by the expression

$$\mathbf{x}'_i = \mathbf{x}_i + \beta_0 \exp(-\gamma r_{ij}^2)(\mathbf{x}_j - \mathbf{x}_i) + \alpha\,(\text{rand}_d - \frac{1}{2}) \tag{7.3}$$

where rand_d is a d-dimensional random vector whose components belong to $[0, 1[$. The α parameter is typically chosen as $\alpha \in [0, 1]$ and the product is understood to be performed componentwise.

The passage from the random noise represented by the third term in equation 7.3 to Lévy flights is straightforward: it is enough to replace the random uniform draw of displacements with a draw from a power law distribution:

$$\mathbf{x}'_i = \mathbf{x}_i + \beta_0 \exp(-\gamma r_{ij}^2)(\mathbf{x}_j - \mathbf{x}_i) + \alpha\,\text{sgn}(\text{rand}_d - \frac{1}{2})\ \text{Levy}_d \tag{7.4}$$

where the term $rand_d - \frac{1}{2}$ now gives the sign of the displacement, the magnitude of which is determined by the random vector Levy_d whose components are drawn from the power law. Note that here the product $\text{sgn}(\text{rand}_d - \frac{1}{2})\text{Levy}_d$ corresponds to an elementwise multiplication.

The parameter α controls the degree of randomness and, consequently, the degree of diversification or intensification of the search. The introduction of Lévy flights allows more diversification with respect to uniform or Gaussian noise, which, according to the originators of the method, is an advantage for the optimisation of high-dimensional multimodal functions [86].

The algorithm just discussed can be described by the following pseudo-code:

iteration = 0
Initialize the firefly population $\mathbf{x}_i, i = 1, \ldots, n$
The light intensity I_i at $\mathbf{x}_i$ is given by $f(\mathbf{x}_i)$
while *iteration* $<$ *Max* **do**
  for $i = 1$ to n
    for $j = 1$ to n
      if $I_i < I_j$ **then**
        firefly i moves towards j according to (7.3) or (7.4)
        update distances r_{ij} and the intensity I_i
        update attractiveness according to new distances (eq. 7.2)
    end for
  end for
  rank fireflies by fitness and update the current best
  iteration = *iteration* + 1
end while

Figure 7.3 depicts the first two iterations of the algorithm without noise, i.e., with $\alpha = 0$. We illustrate this in a simplified environment in order to make the dynamics easier to understand. The objective function we are seeking to maximize here is $f(x, y) = -(x - 5)^2 - (y - 3)^2$, which has its global optimum at $\mathbf{x} = (5, 3)$, in the middle of the contour levels shown in the figure. The left image corresponds to the first iteration. We see the firefly $i = 0$ moving under the influence of firefly $j = 1$, and then towards firefly $j = 2$. At this point firefly 0 has attained a better fitness than that of firefly $j = 3$ and is not attracted to it. Following the loop, now firefly $i = 1$ moves towards the new position of firefly $j = 0$, reaching a better fitness than fireflies $j = 2$ and $j = 3$ and thus stopping there for this iteration. It is now the turn of fireflies $i = 2$ and $i = 3$ to move depending on the current position of the fireflies $j \neq i$.

The right image of Figure 7.3 shows the second iteration of the outer loop of the algorithm. Firefly $i = 0$ first moves towards firefly $j = 1$, then it is attracted by $j = 2$, and finally towards firefly $j = 3$. Following this, fireflies $i = 1$, $i = 2$, and $i = 3$ move in turn. With no noise and with $\beta = 1$, all the fireflies quickly converge to a point $(x, y) = (4.77, 3.23)$ close to the current positions of fireflies 0 and 3, which is suboptimal but close enough to the optimum.

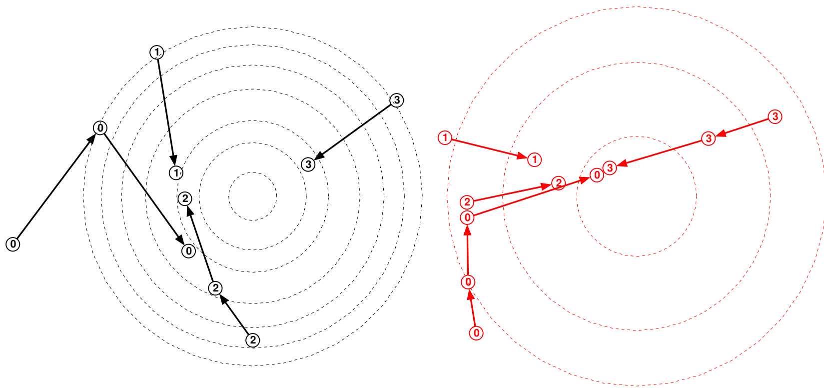


Fig. 7.3. Two stages of the dynamics of four fireflies in a search space described by the dashed contour levels. The global optimum is at the middle of the drawing. The left image shows the first iteration of the outer loop of the algorithm. The right part is the continuation of the left part, with a zooming factor to better illustrate the movement of the fireflies towards the optimum

7.4 Cuckoo Search

We end this chapter on the new metaheuristics with a brief description of a method recently introduced by Yang and Deb [88] and called by them *Cuckoo search*. This

search method was inspired by the behavior of certain bird species, generally called "cuckoos," whose members lay their eggs in the nests of other hosts birds and develop strategies to make the "host" parents take care of their offspring. The behavior described is opportunistic and parasitic and, while sometimes it goes undetected, it elicits a number of reactions on the part of the cheated birds such as recognizing and eliminating the stranger's eggs. This can give rise to a kind of "arms race" in which better and better ways are employed to defend oneself from parasites on one hand, and to improve the mimicry strategies on the other.

7.4.1 Principle of the Method

The cuckoo metaheuristic uses some of these behaviors in an abstract way. Here is a summary of how it is implemented:

- Each cuckoo lays one egg (i.e., a solution) at a time and leaves it in a randomly chosen nest.
- A fraction of the nests containing the better solutions are carried over to the next generation.
- The number of available nests is fixed and there is a probability p_a that the host discovers the intruder egg. In this case, it either leaves the nest and builds another elsewhere, or it disposes of the egg. This phase is simulated by replacing a fraction p_a of the nests among those that contain the worst solutions by new nests chosen at random with a Lévy flight.

The algorithm belongs to the family of population-based metaheuristics. In more detail, it begins by generating an initial population of n host nests containing the initial solutions $\mathbf{x}_i$, which are randomly chosen. Then, each solution's fitness $f(\mathbf{x}_i)$ is evaluated. After that a loop is entered that, as always, is executed until a predefined stopping criterion is met. In the loop a new solution $\mathbf{x}_i$ is generated, a cuckoo's "egg," by performing a Lévy flight starting from an arbitrary nest. The new solution is evaluated and compared with the one contained in a random nest. If the new solution is better than the one already in the nest, it replaces the previous one. The last part of the algorithm consists of substituting a fraction p_a of the n nests containing the worst solutions and of building an equivalent number of new nests containing solutions found by Lévy flights. Metaphorically speaking, this phase corresponds to the discovery of the intruder's eggs and their elimination or the abandoning of the nest by the host. The new solutions are evaluated and ranked, the current best is updated, and a fraction of the best solutions are kept for the next generation.

When a new solution $\mathbf{x}'_i$ is generated through a Lévy flight, it is obtained as follows:

$$\mathbf{x}'_i = \mathbf{x}_i + \alpha \operatorname{sgn}(\text{rand}_d - \frac{1}{2}) \text{ Lévy}_d \tag{7.5}$$

where $\alpha > 0$ is a problem-dependent parameter that dictates the scale of the displacements and the product $\alpha \operatorname{sgn}(\text{rand}_d - \frac{1}{2}) \times \text{Lévy}_d$ is to be taken componentwise, as usual. It is worth noting that, with respect to other metaheuristics, the number of free parameters is smaller. In practice, it seems that only α is relevant, while p_a does not

affect the behavior of the search to a noticeable extent. As we have seen previously in this chapter, Lévy flights are such that most moves will be short around the current solution, giving a local flavor to the search. From time to time, though, longer moves will appear, just as illustrated in Figure 7.2, thus providing a global search and diversification component.

The following pseudo-code describes the cuckoo search algorithm, assuming maximization of the objective function:

```
Initialize the nest population x_i, i = 1,...,n
while stopping criterion not reached do
    choose a random nest i and generate a new solution through a Lévy flight
    evaluate fitness f_i of new solution
    choose a random nest j among the n available

    if f_i > f_j then
        replace j with the new solution
    end if

    a fraction p_a of the worst nests are abandoned and
    the same number of nests with new solutions
    are generated randomly through Lévy flights

    solutions are ranked by fitness and the best current
    solution is stored

    keep the best solutions for the next iteration

end while
```

According to the authors [88], their simple metaheuristic gives better results in the search for the global optimum on a series of standard benchmark functions when compared with an evolutionary algorithm and even with respect to PSO, which is considered highly efficient in difficult function optimization. Nevertheless, we shall see in Chapter 12 that it is dangerous to generalize from a few successful cases. There also exist modified versions of the algorithm which try to increase the convergence speed by progressively reducing the flight amplitude by lowering the parameter α as a good solution is approached. This technique is reminiscent of the temperature schedule used in simulated annealing (see Chapter 4). Furthermore, similarly to what happens in PSO methods, one can create an information exchange between eggs thus relieving the independence of the searches in the original method described above. The results so far are encouraging but it is still too early to judge the general efficiency of the approach in solving a larger spectrum of problems, especially those arising in the real world.

7.4.2 Example

In this section we propose a simple example of the cuckoo method, the search for the maximum of the fitness function f from $[0, 1] \to \mathbb{R}$, shown in Figure 7.4.

The code of the method can be formulated using the syntax of the Python programming language. Here the variable `nest` contains the number of nests, that is the number of solutions x_i (eggs) that are considered at each step of the exploration. These eggs are stored in a list called `solution`, randomly initialized with the Python uniform random generator `random()`. In the main loop, which here contains a maximum number of iterations `maxIter`, a first nest `i` is chosen and its solution is modified with a Lévy increment. If this value x'_i is better than the solution contained in another randomly chosen nest `j`, x_j is discarded and replaced by x'_i. The Lévy flight used here has exponent 1.5 and the amplitude `a` is chosen arbitrarily with a value 0.1.

```
iter=0
solution=[random() for i in range(nest)]
while(iter<maxIter):
    iter+=1
    i=randint(0,nest-1)      # selection of a nest
    x=solution[i]+ a*levy() # random improvement of the "egg"
    x=x-int(x)               # wrap  solution in [0,1]
    j=randint(0,nest-1)      # selection of a recipient nest
    if fitness(x)>fitness(solution[j]):solution[j]=x
    solution.sort(key=fitness)
    solution[0:nest/4]=[random() for i in range(nest/4)]
best=solution[nest-1]
```

Figure 7.4 shows the first six iterations of the cuckoo search with four nests, numbered from 0 to 3. Each image, except that corresponding to the initial state (iteration 0), contains the indices `i` and `j` of the chosen nests. In addition, the variable `accept` in the figure indicates whether the replacement of x_j by x'_i was accepted or not.

The current solutions x_i (the eggs) are shown as black disks, with the corresponding nest index. Note that the solutions are sorted at the end of each iteration. Thus, nest number 3 always contains the best of the four current solutions.

Figure 7.5 shows iterations 11 to 16. The optimal solution ($x^* = 0.4133$, $f(x^*) = 187.785$) is obtained with a precision of about 5% ($x = 0.399$, $f(x) = 180.660$). Figure 7.6 displays the evolution of the three best solutions throughout the iterations. The fourth solution here is random, and it is not shown in the figure. The curves increase monotonically because of the sorting operation performed at each step. As a result of this sorting, the non-randomly renewed solutions can only increase their fitness.

A more detailed analysis that would include steps 6 to 10 reveals that the cuckoo strategy (namely the replacement of x_j with x'_i) was beneficial only four times out of the 16 iterations. Therefore, most of the improvement is the result of the new, randomly generated, solution in nest $i = 0$. To better understand this element of

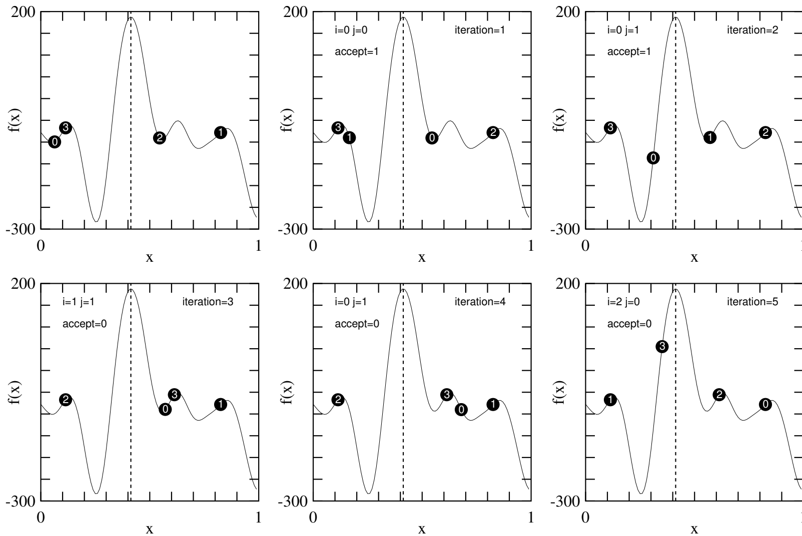


Fig. 7.4. The first six steps of the cuckoo search, with four solutions (nests). Indices i and j indicate the choices of "improvement" and "replacement" nests, as defined in the above code

the method, Figure 7.6 shows in blue the evolution of the best nest when the other three are randomly repopulated at each iteration. In this case, the number of accepted modifications increases to seven. One also sees that values close to the optimal solution are quickly reached. The best solution $x = 0.4137$, $f(x) = 187.780$ is found at iteration 15, now with a precision of less than 0.1%. The computational effort of the method can be estimated with the number of fitness evaluations. At each iteration one has to evaluate the fitness of the m new random solutions, as well as the fitness of the modified solution x'_i. In the case $m = 1$, i.e., only nest $i = 0$ is repopulated at each step, the computational effort is 2 per iteration. In the second case, $m = 3$, nests $i \in \{0, 1, 2\}$ are regenerated, and the computational effort is 4 per iteration.

To get the same order of accuracy, Figure 7.6 indicates that 15 iterations are needed in the first case, (upper black curve) whereas five iterations are enough in the second case (blue curve). The corresponding computational effort is then 30 and 20, respectively. This hints again at the importance of introducing enough new random solutions at each steps.

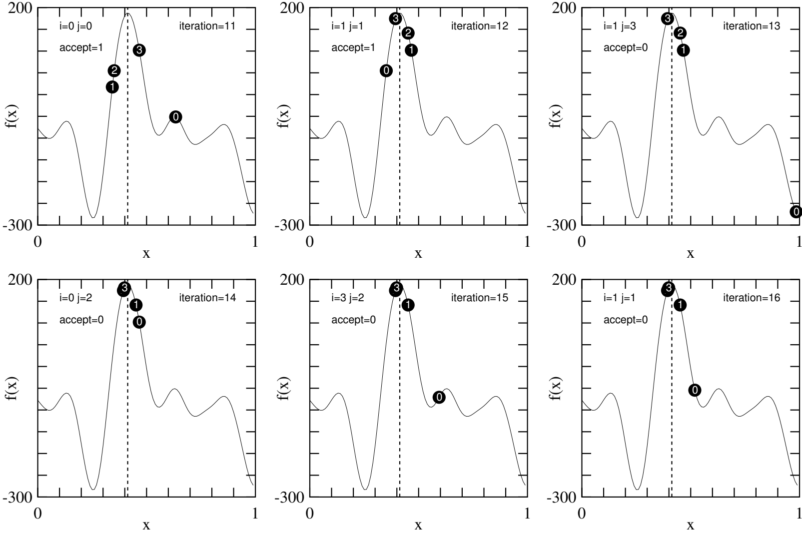


Fig. 7.5. Steps 11 to 16 of the cuckoo search described in Figure 7.4

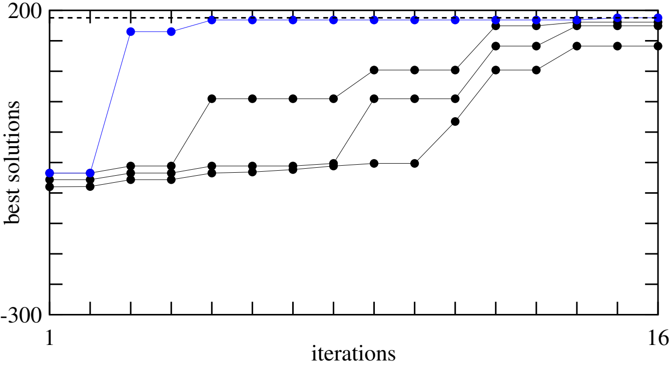


Fig. 7.6. The black curves display the evolution of the three best solutions during the first 16 iterations of the cuckoo search, where only nest $i = 0$ is randomly repopulated. In blue, the best solution is shown for the case where the other three solutions are randomly recreated at each iteration

8

Evolutionary Algorithms: Foundations

8.1 Introduction

Evolutionary algorithms (EAs) are a set of optimization and machine learning techniques that find their inspiration in the biological processes of evolution established by Darwin [27] and other scientists in the ninenteenth century. Starting from a population of individuals that represent admissible solutions to a given problem through a suitable coding, these metaheuristics leverage the principles of variation by mutation, and recombination, and of selection of the best-performing individuals in a given environment. By iterating this process the system finds increasingly good solutions and generally solves the problem satisfactorily.

A brief history of the field will be useful to understand where these techniques come from and how they evolved. The first ideas were conceived in the United States by J. Holland [40] and L. Fogel and coworkers [33]. Holland's method is known as *genetic algorithms* while Fogel's approach is known as *evolutionary programming*. Approximately at the same time and independently Ingo Rechenberg, while working at the Technical University in Berlin, started to develop related evolution-inspired methods that were called *evolution strategies* [71]. In spite of their common origin in the abstract imitation of natural evolutionary phenomena, these strands were different at the beginning and evolved separately for some time. However, as time passed and researchers started to have knowledge of the work of their peers, the different techniques influenced each other and gave birth to the family of metaheuristics that are collectively known today as *evolutionary algorithms* (EAs). In fact, although the original conceptions were different, the fundamental idea of using evolution to find good solutions to difficult problems was common to all the techniques proposed. Today EAs are a rich class of population-based metaheuristics that can profitably be used in the optimization of hard problems and also for machine learning purposes. Here, in line with the main theme of the book, we shall limit ourselves to optimization applications, although it is sometimes difficult to distinguish between optimization and learning in this context. For pedagogical reasons, since evolutionary methods are certainly more complex than most other metaheuristics, we believe that it is useful to first present them in their original form and within the frame of

© Springer Nature Switzerland AG 2018

B. Chopard, M. Tomassini, *An Introduction to Metaheuristics for Optimization*,
Natural Computing Series, https://doi.org/10.1007/978-3-319-93073-2_8

their historical development. Thus, we shall first describe genetic algorithms and evolution strategies in some detail in this chapter. Other techniques such as genetic programming will be introduced, together with further extensions, in Chapter 9.

8.2 Genetic Algorithms

We shall begin by giving a qualitative description of the structure and the functioning of Holland's genetic algorithms (GAs) with binary coding of solutions, one of the best-known evolutionary techniques. The idea is to go through a couple of simple examples in detail, to introduce the main concepts and the general evolutionary mechanisms, without unnecessary complications.

8.2.1 The Metaphor

Let us dig deeper into the biological inspiration of genetic algorithms. The metaphor consists in considering an optimization problem as the environment in which simulated evolution takes place. In this view, a set of admissible solutions to the problem is identified with the individuals of a population, and the degree of adaptation of an individual to its environment represents the fitness of the corresponding solution. The other necessary ingredients are a source of variation such that individuals (solutions) may undergo some changes and, possibly, the production of new individuals from pairs or groups of existing individuals. The last ingredient is, most importantly, selection. Selection operates on the individuals of the population according to their fitness: those having better fitness are more likely to be reproduced while the worst ones are more likely to disappear, in a such a way that the size of the population is kept constant. It is customary to not submit the best or a small number of best solutions to selection in order to keep the best current individuals in the population representing the next generation. This is called *elitism*.

8.2.2 Representation

In EAs the admissible solutions to a given problem, i.e., the individual members of the population, are represented by suitable data structures. These structures may be of various types, as we will see later on. For the time being, let's stick to the simplest representation of all: binary coding. Binary strings are universal in the sense that any finite alphabet of symbols can be coded by a suitable number of binary digits, which in turn means that any finite data structure can be represented in binary. We shall indeed use binary coding for our first examples, although from the efficiency point of view other choices would be better in some cases. Classical genetic algorithms were characterized by Holland's choice to use binary-coded individuals.

8.2.3 The Evolutionary Cycle

A genetic algorithm begins by establishing a population of individuals, which are binary-coded solutions to a problem we want to solve. The size of the population is small with respect to the number of admissible solutions, from some tens to a few thousands of individuals. This initial population is usually chosen at random but it is also possible to "seed" it with solutions found with another approach or determined by a human.

The algorithm then enters a loop and a new population will be produced in each iteration starting from the previous populations and applying a certain number of stochastic operators to it. Such an iteration is usually dubbed a *generation*.

The first operator to be applied is called *selection*. The goal of selection is to simulate the Darwinian law of the survival of the more adapted individuals. Historically, the first selection method in genetic algorithms was *fitness-proportionate selection* but other methods have been introduced subsequently and we will describe them in detail in the next chapter. Selection proportional to fitness works by choosing population members with probability proportional to their fitness, which of course must be evaluated previously. In a population of size n, selection is repeated n times with replacement. In other words, individuals are selected one at a time, saved in an intermediate population, and a copy is replaced in the original population from which the draw is being made. In the intermediate population high-fitness individuals may appear more than once, while low-fitness individuals may never be selected and thus disappear from the population.

The intermediate population contains the "parents" from which the new population will be generated thanks to further genetic operators, the original ones being *mutation* and *crossover*.

For crossover one first forms pairs from the intermediate population; pairs can be taken simply in the order in which they have been previously selected. Next, pairs of individuals are mixed or recombined with a certain probability called the *crossover probability*, which is usually about 0.5. In the simplest case, crossing them over means choosing at random an interior point in the strings and then, for example, exchanging the sections to the right of the cut point between the two individuals. In the end, we are left with two new individuals that contain "genetic material" of both parents, imitating sexual reproduction in biology. This type of recombination is called one-point crossover; there exist more complicated crossovers that will be introduced in due course.

After crossver, one can possibly apply the mutation operator to the individuals of the new population. Mutation applies to single individuals and its purpose is to simulate transcription and other errors, or noise from outside, both of which are known to occur in biological reproduction. In genetic algorithms mutation is typically applied with lower probability than crossover. In binary strings it simply consists in flipping bits with the given mutation probability.

The following schema in which $P(t)$ stands for a whole population at time step, or generation, t illustrates the cycle we have just described.

$$P(t) \xrightarrow{selection} P'(t) \xrightarrow{crossover} P''(t) \xrightarrow{mutation} P'''(t) \equiv P(t+1)$$

The loop selection/crossover/mutation terminates according to different criteria chosen by the user. The more commons ones are the following:

- a predetermined number of generations has been reached;
- a satisfactory solution to the problem has been found;
- fitness has ceased to improve during a predetermined number of generations.

The evolutionary cycle just described can be represented by the following pseudo-code:

generation = 0
Initialize population
while *exit criterion not reached* **do**
  generation = *generation* + 1
  Compute the fitness of all individuals
  Select individuals
  Crossover
  Mutation
end while

The above evolutionary algorithm schema is called *generational*. As the name implies, the entire population is replaced by the offspring before the next generation begins, in a synchronized manner. In other words, generations do not overlap. It is also possible to have generations overlap by producing some offspring, thus changing only a fraction of the population, and having them compete for survival with the parents. These *steady-state evolutionary algorithms* are also used but are perhaps less common and more difficult to understand than generational systems. For these reasons, we will stick to generational EAs in the rest of the book, with the exception of some evolution strategies to be studied later in this chapter. For some discussion of the issues see, e.g., [60].

8.2.4 First Example

This first example lacks realism but it is simple enough to usefully illustrate the mechanisms implemented by genetic algorithms, as described in an abstract manner in the previous sections. Indeed, we shall see that powerful and flexible optimization techniques can result from the application of biological concepts to problem solving. The example is based on the *MaxOne*, problem which was introduced in Chapter 2.

We recall that in this problem the objective is the maximization of the number of 1s in a binary string of length l. We know that the problem is trivial for an intelligent agent but we also know that the algorithm has no information whatsoever about the "high-level" goal, it only sees zeros and ones and has to blindly find a way to maximize the 1s, i.e., to find the string $(1)^l$. Moreover, if we take $l = 100$, the search space is of size 2^{100}, a big number indeed when one has no clue as to how to proceed.

Let's start by defining the fitness $f(s)$ of a string s: it will simply be the number of 1s in s and this is coherent with the fact that the more 1s in the string, the closer we are to the sought solution.

Let us start with a population of n randomly chosen binary strings of length l. For the sake of illustration, we shall take $l = 10$ and $n = 6$ although these numbers are much too small for a real genetic algorithm run.

$$\begin{aligned} s_1 &= 1111010101 \; f(s_1) = 7 \\ s_2 &= 0111000101 \; f(s_2) = 5 \\ s_3 &= 1110110101 \; f(s_3) = 7 \\ s_4 &= 0100010011 \; f(s_4) = 4 \\ s_5 &= 1110111101 \; f(s_5) = 8 \\ s_6 &= 0100110000 \; f(s_6) = 3 \end{aligned} \tag{8.1}$$

The selection operator is next applied to each member of the population. To implement fitness-proportionate selection, we first compute the total fitness of the population, which is 34, for an average fitness of $34/6 = 5.666$. The probability with which an individual is selected is computed as the ratio between its own fitness and the total population fitness. For example, the probability of selecting s_1 is $7/34 = 0.2059$, while s_6 will be selected with probability $3/34 = 0.088$. Selection is done n times, where $n = 6$ is the population size. Whenever an individual is selected, it is placed in an intermediate population and a copy is replaced in the original population. Let us assume that the result of selection is the following:

$$\begin{aligned} s'_1 &= 1111010101 \; (s_1) \\ s'_2 &= 1110110101 \; (s_3) \\ s'_3 &= 1110111101 \; (s_5) \\ s'_4 &= 0111000101 \; (s_2) \\ s'_5 &= 0100010011 \; (s_4) \\ s'_6 &= 1110111101 \; (s_5) \end{aligned} \tag{8.2}$$

We remark that string s_5 has been selected twice, while s_6 has not be selected and it is thus bound to disappear. This behavior is normal: selection tends to concentrate search on solutions that are better than average. After a number of generations, this phenomenon causes an homogenization of the population. However, the progressive loss of diversity is partially thwarted by the other genetic operators, especially mutation.

Until now what has happened is that individuals of better quality have enjoyed more chance of finding themselves in the next population. Clearly, this is not enough to create novelty: with selection alone, only an already existing solution in the population may come to dominate. The required variation is provided by the operators of crossover and mutation. To apply crossover we first form pairs of strings in the order from s'_1 to s'_6. Next, the pairs s'_1 and s'_2, s'_3 and s'_4, and s'_5 and s'_6 will be recombined with probability of crossover 0.6. For the sake of illustration, let's assume that, after drawing the random numbers, only the pairs (s'_1, s'_2) and (s'_5, s'_6) undergo crossover. For each pair to be recombined we draw another random number between 1 and 9 to

determine the crossover point; for example 2 for the first pair of strings and 5 for the second. For the first pair, this will give us

$$\begin{aligned} s'_1 &= 11 \cdot 11010101 \\ s'_2 &= 11 \cdot 10110101 \end{aligned} \tag{8.3}$$

before crossover, and

$$\begin{aligned} s''_1 &= 11 \cdot 10110101 \\ s''_2 &= 11 \cdot 11010101 \end{aligned} \tag{8.4}$$

after crossover. By chance, in this case no new strings will be produced as the offspring are identical to the parents. For the other pair (s'_5, s'_6) we will have

$$\begin{aligned} s'_5 &= 01000 \cdot 10011 \\ s'_6 &= 11101 \cdot 11101 \end{aligned} \tag{8.5}$$

before crossover, and

$$\begin{aligned} s''_5 &= 01000 \cdot 11101 \\ s''_6 &= 11101 \cdot 10011 \end{aligned} \tag{8.6}$$

after crossover. This time the offspring are new individuals.

The last phase for the production of a new population makes use of the random mutation of one or more "genes", here represented by binary digits, in the single individuals that have been included in the intermediate population. For each string and for each bit in the string we allow the inversion of the bit with a low probability, e.g., 0.1. Over the total 60 binary digits of our population we would thus expect that about six will be flipped. Of course in a such small population fluctuations will be high but this will be less of a problem in the large populations that are used in practice. In the end, the result might be the following, where the bits to be flipped have a bar on them:

$$\begin{aligned} s''_1 &= 11101\bar{1}0101 \\ s''_2 &= 1111\bar{0}1010\bar{1} \\ s''_3 &= 11101\bar{1}11\bar{0}1 \\ s''_4 &= 0111000101 \\ s''_5 &= 0100011101 \\ s''_6 &= 11101100\bar{1}1 \end{aligned} \tag{8.7}$$

Looking carefully at the result we see that four out of the six mutations turn a 1 into a 0, which will cause the fitness of the corresponding individual to decrease in the present problem. This is not illogical: at this stage, selection will have built strings that tend to have more 1s than 0s on average because better solutions are more likely to be chosen. Crossover may weaken this phenomenon to some extent but 1s should still prevail. So, mutations should be more likely to produce a negative effect. The interplay between selection, crossover, and mutation will be studied in a quantitative way in Section 8.3.

After mutation, our population looks like this:

$$\begin{array}{ll} s_1''' = 1110100101 & f(s_1''') = 6 \\ s_2''' = 1111110100 & f(s_2''') = 7 \\ s_3''' = 1110101111 & f(s_3''') = 8 \\ s_4''' = 0111000101 & f(s_4''') = 5 \\ s_5''' = 0100011101 & f(s_5''') = 5 \\ s_6''' = 1110110001 & f(s_6''') = 6 \end{array} \quad (8.8)$$

Thus, in just one generation the total population fitness has gone from 34 to 37, an improvement of about 9%, while the mean fitness has increased from 5.666 to 6.166. According to the algorithm, the process may now be iterated a number of times until a given termination criterion is reached. However, the mechanism remains the same and the description of the workings of a single generation should be enough for the sake of the example.

In a more realistic way, let us consider now the evolution of a larger population of 100 binary strings each of length $l = 128$ and let's suppose that the algorithm stops when either it finds the optimal solution, or when 100 generations have elapsed. Crossover and mutation probabilities are 0.8 and 0.05 respectively and fitness-proportionate selection is used. Figure 8.1 shows the evolution of the fitness of the best individual and the average fitness for one particular execution of the genetic algorithm. In this case the optimal solution with fitness 128 has been found in fewer than 100 iterations. We remark that the average fitness in the final phase is also close to 128, which means that the whole population is close to optimality. Another typical trait of genetic algorithms, and of evolutionary algorithms in general, is the fast increase of the fitness in the first part of the process, while improvements become slower later on. Indeed, as the population becomes fitter it is also more difficult to further improve the solutions and the search tends to stagnate as time goes by. Finally, it is to be remarked that the *MaxOne* problem is an easy one for genetic algorithms as fitness improvements are cumulative: each time a 1 is added to the string there is a fitness improvement that can only be undone by an unlucky crossover or a mutation. On the whole, strings with many 1s will tend to proliferate in the population. This is far from being the case for harder problems.

Figure 8.2 refers to the same run as above and it illustrates another aspect of the population evolution that is different, but related, to the fitness curves. The graphics here depict two different measures of the *diversity* of the individuals in the population: entropy and fitness variance. Without getting into too many details that belong to elementary statistics, we clearly see that diversity is maximal at the beginning because of the random initialization of the population. During the execution diversity, by either measure, tends to decrease under the effect of selection, which causes good individuals to be replicated in the population and is only slightly moderated by crossover and mutation. But even those sources of noise cannot thwart the curse of the algorithm and diversity becomes minimal at convergence since good solutions are very similar.

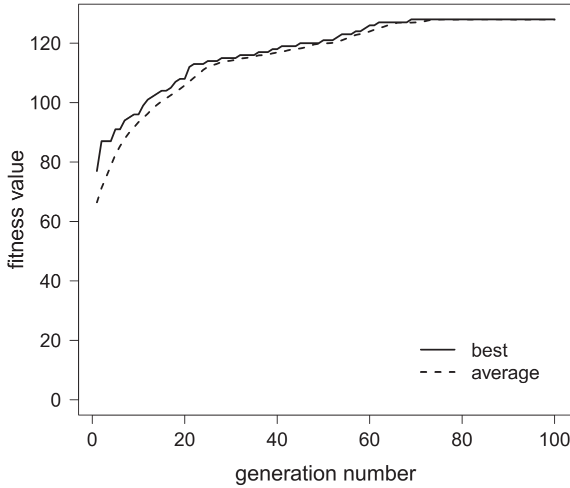


Fig. 8.1. Evolution of the average population fitness and of the best individual fitness as a function of generation number. The curves represent a particular execution of the genetic algorithm on the *MaxOne* problem but are representative of the behavior of an evolutionary algorithm

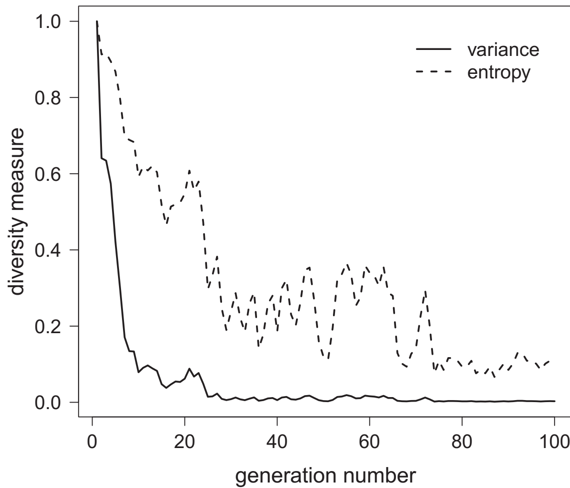


Fig. 8.2. Evolution of population diversity in terms of fitness variance and fitness entropy for the same genetic algorithm run as in the previous figure

8.2.5 Second Exemple

In this section we shall introduce a second example of the use of genetic algorithms for optimization. This time the example falls within the more classical continuous real-valued function optimization domain. The problem is again very simple and it is possible to solve it by hand but it is still interesting to see how it is treated in a genetic algorithm context, thus preparing the ground for more realistic cases.

We remind the reader (see Chapter 2) that the non-constrained minimization of a function $f(\mathbf{x})$ in a given domain $D \in \mathbb{R}^n$ of its real variables can be expressed in the following way: find $\mathbf{x}^*$ such that

$$f(\mathbf{x}^*) = \min\{f(\mathbf{x}) \mid \forall \mathbf{x} \in D\}$$

where $\mathbf{x} = (x_1, x_2, \ldots, x_n)^T$.
Let us consider the following function (see Figure 8.3):

$$f(x) = - \mid x \sin(\sqrt{\mid x \mid}) \mid + C.$$

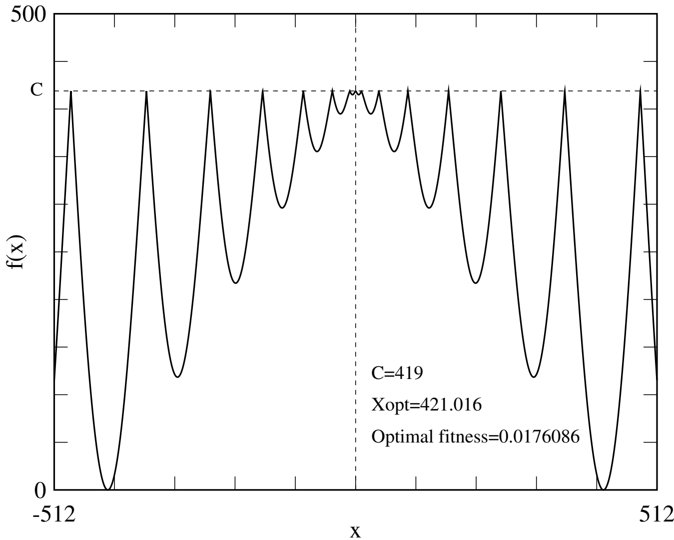


Fig. 8.3. Graph of $f(x), x \in [-512, 512]$

We are asked to find x^* in the interval $D = [-512, 512]$ such that $f(x^*)$ takes its globally minimal value in this interval. Since $f(x)$ is symmetric, it will suffice to

consider the positive portion of the interval and zero. Here are the genetic algorithm ingredients that are needed to solve the problem. An admissible solution is a real value x in the interval $[0, 512]$. The initial population is of size 50 and the candidate solutions, i.e., the individuals in the population, are randomly chosen in the interval $[0, 512]$. An admissible solution will be coded as a binary string of suitable length; therefore, in contrast with the *MaxOne* problem where the solutions were intrinsically binary, we will need to decode the strings from binary to real numbers in order to be able to evaluate the function.

In the computer memory, only a finite number of reals can actually be represented because of the finite computer word length. The string length gives the attainable precision in representing reals: the longer the string, the higher the precision.[1] For example, if strings have a length of ten bits, then $1{,}024$ values will be available to cover the interval $[0, 512]$, which gives a granularity of 0.5 meaning that we will be able to sample points that are 0.5 apart. The strings (0000000000) and (1111111111) represent the extremities of the interval, i.e., 0.0 and 512.0 respectively, all the other strings will correspond to an interior point.

The genetic algorithm mechanism is identical to the previous case and corresponds to the pseudo-code presented in Section 8.2. The following Table 8.1 shows the evolution of the best fitness and the average fitness as a function of the generation number for a particular run of the algorithm.

Generation	Best	Average
0	104.30	268.70
3	52.67	78.61
9	0.00179	32.71
18	0.00179	14.32
26	0.00179	5.83
36	0.00179	2.72
50	0.00179	1.77
69	0.00179	0.15

Table 8.1. Evolution of best and average fitness for a particular run of an EA

The average fitness as well as the fitness of the best solution found are high at the beginning, but very quickly the population improves under the effect of the genetic operators, and the optimal solution is already found in generation nine, within the limits of the available precision. Average fitness continues to improve past this point, a sign that the population becomes more and more homogeneous. We point out that, because of the stochastic nature of evolutionary algorithms, performance may vary

[1] In actual practice, real numbers are represented in the computer according to the IEEE floating-point formats, which comprise a sign bit, a mantissa, and an exponent. Modern evolutionary algorithms take advantage of this standard coding, as we shall see below.

from one execution to the next unless we use the same seed and the same pseudo-random number generator across executions. A better indicator of the efficiency of the algorithm would be the average performance over a sufficient number of runs, a subject that will be treated in detail in Chapter 12.

In this example the probability of getting stuck in one of the suboptimal local minima is negligible, but this need not be the case in more difficult problems. Ideally, the algorithm should strive to find a compromise between the exploitation of promising regions of the search space by searching locally, and the exploration of other parts of the space where better solutions might be found. This is a manifestation of the diversification/intensification compromise we found in Chapter 2.

For the optimization of mathematical real-valued functions the naive binary representation used in the example for pedagogical purposes is not efficient enough. Indeed, let's assume that the function to be optimized is defined in a 20-dimensional space, which is current in standard benchmark problems, and let's also assume that we use 20 binary digits for each coordinate. This gives us strings of length $20 \times 20 = 400$ to represent a point $\mathbf{x}$ in space. The size of the search space is thus 2^{400}, which is huge. Crossover and mutation are not likely to be efficient in such a gigantic space and the search will be slow. For real-valued functions modern evolutionary algorithms work directly with machine-defined floating-point numbers; they are much more efficient because this choice significantly limits the scope of the variation operators and prevents the search from wandering in the space. This coding choice needs specialized operators that take into account the real-valued format but it's worth the effort. We shall take up the subject again in Section 8.4 of this chapter on evolution strategies and more information is available in the literature, see, e.g., [9, 60].

It turns out that genetic algorithms and evolutionary algorithms in general are, together with PSO, among the best known metaheuristics for the optimization of highly multimodal functions of several variables. The two-dimensional functions represented in Figures 8.4 and 8.5 illustrate the idea, considering that functions similar to these, but with many more variables, are often used as benchmarks to evaluate the quality of a given optimization method. One of the reasons that make evolutionary algorithms interesting in this context is their capability to jump out of a local optimum thanks to crossover and mutation, a feat that is in general not achievable with classical mathematical optimization methodologies. An additional advantage is that they don't need derivatives in their workings, and can thus also be used with discontinuous and time-varying functions.

8.2.6 GSM Antenna Placement

To complete the rather idealized examples presented above, in this section we briefly describe a real-world optimization problem solved with the help of a GA. The objective is to find the optimal placement of GSM antennas of a provider of mobile telephone services in a town [20]. The problem calls for covering a whole urban zone such that the intensity of the signal at each point is sufficient to guarantee a good telephone communication. The technical difficulty consists in doing that while

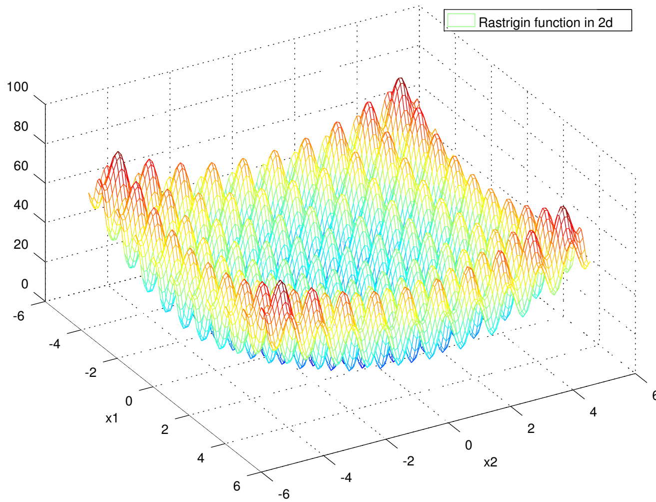


Fig. 8.4. Rastrigin function in two dimensions

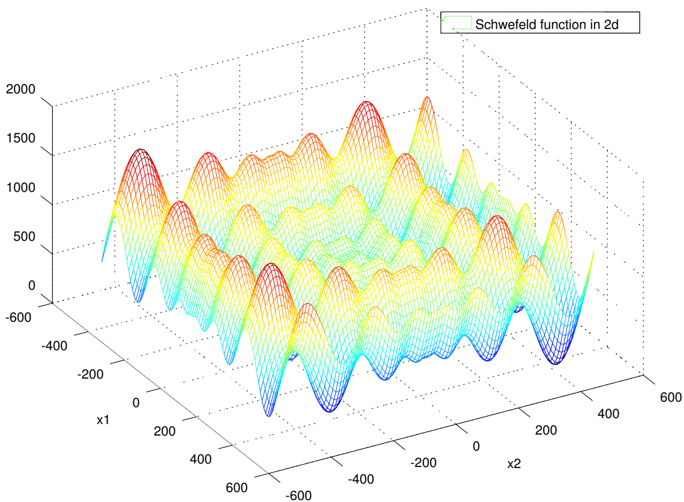


Fig. 8.5. Schwefel function in two dimensions

obeying a number of physical and cost constraints. Waves are absorbed and reflected by the buildings in an urban environment and a zone, called a *microcell*, is created around each antenna such that the intensity of the signal there is higher than a given threshold. The size of a microcell should not be too large, in order for the number

of potential users in that zone to be adequate with respect to the capabilities of the telephonic relays associated with the microcell.

Figure 8.6 provides an example of an antenna layout (black spots) and of their corresponding microcells (colored areas). It should be pointed out that the set of possible placements is limited *a priori* by technical and legal factors.

The fitness function for this problem is too technical and complex for a meaningful expression to be given here; the interested reader is referred to [20] for a detailed description. Roughly speaking, the algorithm must find the placement of a given number of antennas such that the number of non-covered streets is minimized and the adjacent microcells are of maximal size, including sufficient microcell overlap to allow mobile users to move from one zone to the next without service disruption.

To compute the fitness function one has to simulate the wave propagation emitted by each antenna, and the effect of the buildings and other obstacles must be calculated. This kind of computation can be difficult and needs appropriate computing power to be done. The result is an intensity map similar to Figure 8.6. Starting from these data, we must evaluate the quality of the corresponding coverage according to the given fitness function.

From the point of view of the genetic algorithm implementation, which is the most interesting aspect here, one must first define an individual of the population. In the solution chosen here, each individual is a geographical map on which a choice of antenna layout has been made. Mutation is then performed by moving a randomly chosen antenna from its present position to another possible point. Crossover of a pair of individuals consists of merging two adjacent pieces of the corresponding maps. Intuitively, the hope is that crossover should be able to recombine two pieces that are partially good, but still representing sub-optimal individuals, into a new better map.

8.3 Theoretical Basis of Genetic Algorithms

Now that we are somewhat familiar with the workings of a GA, it is time to take a look at their theoretical basis, which was established by Holland [40]. Holland's analysis makes use of the concept of *schemas* and their evolution; in the next section we provide an introduction to these important ideas and their role in understanding how a genetic algorithm works.

Evolution of schemas in the population.

The concept of a schema is very important in the analysis of classical, binary-coded genetic algorithms. From a population point of view, a schema is a subset S of the set of all the 2^l binary strings of length l. Formally, it is a pattern that consists of l symbols belonging to the set $\{0, 1, \star\}$. The symbol $\star$ plays the role of a "wild card," that is, it matches both a 0 and a 1. A schema thus defines a family of binary strings, all the strings that have a 0 or a 1 at positions marked as 0 or 1 respectively, and all the possible combinations of 0 and 1 at the positions marked as $\star$. For instance, the schema

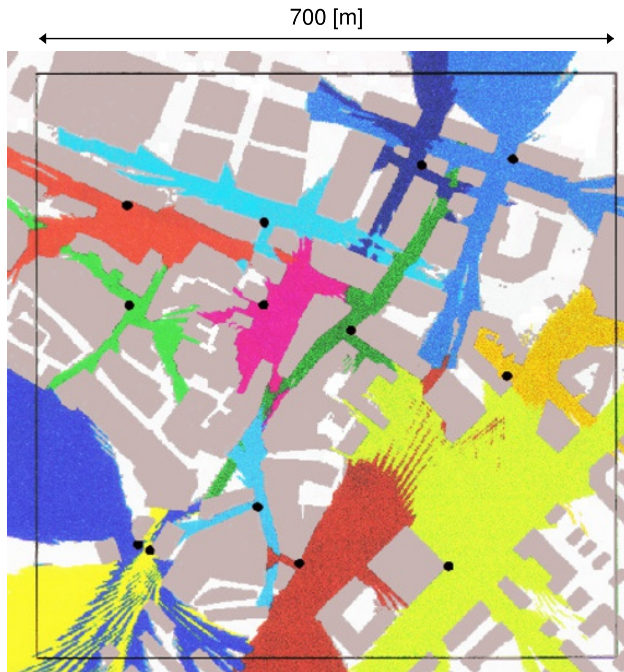


Fig. 8.6. An example of GSM antenna placement in an urban environment, and of the range of the waves emitted

$$S = (0 \star 1 \star) \tag{8.9}$$

represents the following family of binary strings:

$$\begin{array}{c}(0010)\\(0011)\\(0110)\\(0111)\end{array} \tag{8.10}$$

There are 3^l different schemas of length l. A schema S can also be seen as a *hyperplane* in binary space whose dimension and orientation in the l-dimensional binary space depends on the number and positions of the $\star$ symbols in the string. We now give some useful definitions for schemas.

The *order* $o(S)$ of a schema S is defined as the number of fixed positions (0 or 1) in the string that represents it. For example, for the schema (8.9) $o(S) = 2$. The cardinality of a schema S depends on its order according to the following expression: $|S| = 2^{l-o(S)}$.

The *defining length* $\delta(S)$ of a schema S is the distance between the first and the last fixed positions in S. For example, for the schema defined in equation (8.9), $\delta(S) = 3 - 1 = 2$. The defining length of a schema can be understood as a measure of the "compactness" of the information contained in it.

The goal of Holland's analysis is the description of the dynamics of schemas when the population of binary strings evolves under the effects of the operators of the standard genetic algorithm, i.e., selection, crossover, and mutation. We will start by describing the role of selection alone.

Let $N_t(S)$ be the number of instances of schema S in the population at time t and let $f_t(S)$ be the average fitness at time t of all strings represented by schema S; $\bar{f}_t$ denotes the average fitness of the whole population. Under the effect of fitness-proportionate selection, the number of instances of S at time $t+1$ is given by the following recurrence:

$$N_{t+1}(S) = N_t(S)\,\frac{f_t(S)}{\bar{f}_t} \tag{8.11}$$

Assuming that S defines a set of strings with an average fitness higher than the mean fitness of the population, we have that

$$f_t(S) = \bar{f}_t + c\,\bar{f}_t, \quad c > 0 \tag{8.12}$$

By replacing $f_t(S)$ in equation (8.11) with the expression in equation (8.12) we get:

$$N_{t+1}(S) = N_t(S)\,\frac{\bar{f}_t + c\,\bar{f}_t}{\bar{f}_t} = N_t(S)\,(1+c). \tag{8.13}$$

This recurrence is of the type $x_{t+1} = kx_t$ with k constant, one of the simplest. The closed solution is easy to find: if x_0 is the initial value of x then we have:

$$x_1 = kx_0,\; x_2 = kx_1 = k(kx_0) = k^2x_0, \ldots,\; x_t = kx_{t-1} = \ldots = k^t x_0$$

Therefore, replacing k by $(1+c)$ and x_t by $N_t(S)$ we are led to

$$N_t(S) = N_0(S)\,(1+c)^t \tag{8.14}$$

where $N_0(S)$ is the fraction of strings belonging to S in the initial population. The last equation says that schemas that are better than the average reproduce exponentially in the population under the action of fitness-proportionate selection. Instead, the frequency of schemas with a fitness lower than the average ($c < 0$) will decay exponentially given that in this case the factor $(1+c)$ is less than one.

Selection alone would simply make the dynamics converge on the best individuals in the population but this is not how a genetic algorithm works for, otherwise, it would be unable to find still unknown solutions, possibly better than those already contained in the initial population. Thus, we must investigate the effect of the variation operators that provide for novelty: crossover and mutation. Owing to the fact that they modify individuals, crossover and mutation have an adverse effect on the rate of growth of the best individuals when they disrupt or diminish the number of instances of a good schema S.

To investigate these effects, let $P_{sc}[S]$ be the survival probability of a schema S with respect to crossover and let $P_{sm}[S]$ be the survival probability of S with

respect to mutation. The "survival" of a schema S means that a crossover or mutation operation generates a string of the same family S.

Let us first investigate the effects of crossover. The defining length of a schema plays an important role in its survival. If the crossover point falls in the interior of the portion defined by $\delta(S)$ the schema may be destroyed. But the crossover point is chosen with uniform probability among the $l-1$ possible points; it follows that the probability that the chosen point fragments the schema is given by $\frac{\delta(S)}{l-1}$, and, consequently, the probability of survival is $1-\frac{\delta(S)}{l-1}$. Now, crossover is applied with probability p_{cross}, which gives in the end

$$P_{sc}[S] = 1 - p_{cross}\,\frac{\delta(S)}{l-1}. \tag{8.15}$$

In the mutation case, the probability that a fixed position of a schema is altered is p_{mut} and thus the probability that it stays the same is $1-p_{mut}$. But there are $o(S)$ fixed positions in a schema S and each position may mutate independently, which gives the following probability for the survival of a schema, i.e., for the probability that no bit in a fixed position is changed:

$$P_{sm}[S] = (1-p_{mut})^{o(S)} \approx 1 - o(S)\,p_{mut} \tag{8.16}$$

where it has been taken into account that $p_{mut} \ll 1$.

We can now combine equation (8.11) describing the growth of schema frequency with the equations that give the probability of schema survival when we add crossover and mutation (eqs. 8.15 and 8.16). The result is the following expression, which is customarily called the *schema theorem* of genetic algorithms:

$$N_{t+1}(S) \geq N_t(S)\,\frac{f_t(S)}{\bar{f}_t(S)}\left(1 - p_{cross}\frac{\delta(S)}{l-1} - o(S)p_{mut}\right) \tag{8.17}$$

The last inequality is a lower bound on the rate of growth of schemas and it is interpreted in the following way: short, high-fitness schemas of low order have a tendency to increase exponentially in the population. The growth is limited by the effects of crossover and mutation, with the former being more important since usually $p_{cross} \gg p_{mut}$ in genetic algorithms. The result is a lower bound because crossover and mutation are not only destructive operators; from time to time they may create new instances of a schema S.

The building-block hypothesis and deceptive functions.

The results of the previous section are due originally to Holland but equivalent or similar results have been obtained more recently for individual representations other than binary. These results thus extend the important concept of growth of good individuals in evolving populations to other evolutionary algorithms, also in the presence of more complex genetic operators.

One of the consequences of the growth equation for schemas is the so-called "building block hypothesis." The idea is that a GA increases the frequency in the population of high-fitness, short-defining-length, low-order schemas, which are dubbed

"building blocks" and which are recombined through crossover into solutions of increasing order and increasing fitness. The building-block hypothesis is interesting but its importance has probably been overemphasized. As an example, let us consider again the *MaxOne* problem. It is easy to see that building blocks do exist in this case: they are just pieces of strings with 1s that are adjacent or close to one another. With positive probability, crossover will act on pairs of strings containing more 1s on opposite sides of the crossover. This will produce offspring with more 1s than either parent, thus effectively increasing the fitness of the new solutions. However, this behavior, which is typical of additive problems, is not shared by many other problem types. For an extreme example of this, let us introduce *deceptive functions*, which are problems that have been contrived to expose the inability of a GA to steer the search toward a global optimum. The simplest case arises when for a schema S, $\gamma^* \in S$ but $f(S) < f(\bar{S})$, where γ^* is the optimal solution and $\bar{S}$ is the complement of S. This situation easily leads the algorithm to mistakenly take the wrong direction, and that is why such functions are called deceptive. The basic example of this behavior is given by a *trap* function. A trap is a piecewise linear function that divides the space into two distinct regions, a smaller one that leads to the global optimum, and the second leading to a local optimum (see Figure 8.7). The trap function is defined as follows:

$$t(u(s)) = \begin{cases} \frac{a}{z}(z - u(s)), & u(s) \in [0, z] \\ \frac{b}{l-z}(u(s) - z), & u(s) \in [z, l] \end{cases}$$

where $u(s)$ is called "unitation," that is, the number of 1s in the string s. It is intuitively clear that, unless the search starts in the region between z and l, evolution is much more likely to steer the search towards the suboptimal maximum a, which has a much larger basin of attraction, and the hardness of the function increases with increasing z. In conclusion, although deceptive functions are not the norm in real-world problems, the example shows that the building-block hypothesis might be overoptimistic in many cases. For details on this advanced subject the reader is referred to the specialized literature.

Convergence in probability.

The successive application of the genetic operators of selection, crossover, and mutation to an initial population X_0 is a discrete stochastic process that generates a sequence $\{X_t\}_{t=0,1,2,...}$ of populations (states) because the genetic operators contain random components. The finite set of states of this process is the set of all possible finite-size populations. In the case of the generational genetic algorithm as described previously, the population $X(t)$ at time step t only depends on the previous population X_{t-1}. This implies that the stochastic process is of Markovian type and it has been shown that it converges with probability one to the global optimum of the associated problem. Convergence is guaranteed under weak conditions: it must be possible to choose any individual as a parent with positive probability, reproduction must include elitism in the sense that the best current solution goes unchanged into the next generation, and any admissible solution must be obtained with positive probability by the variation operators to ensure ergodicity. This result is theoretically useful since few metaheuristics are able to offer such a proof of convergence. However,

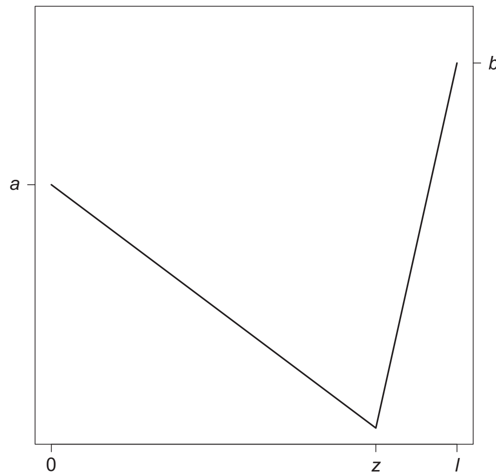


Fig. 8.7. The trap function described in the text. The abscissa represents the number of 1s in the string. The corresponding fitness is shown on the y-axis

it is not very relevant in practice since it doesn't tell us anything about the speed at which the optimal solution will be found. This time can be very long indeed because of the fluctuations of the random path to the solution. In the end, the situation can be even worse than complete enumeration, which takes an exponential, but known, time to solution. Of course, these considerations are not very important for practitioners who, in general, are ready to sacrifice some marginal fitness improvement in exchange for a quickly obtained good enough solution.

8.4 Evolution Strategies

Together with genetic algorithms, the other two original methods based on evolutionary ideas are evolution strategies and evolutionary programming. For reasons of space, and also because they have been adopted more widely over the years, we shall limit ourselves to the description of evolution strategies, a family of metaheuristics that is very useful in optimizing continuous real-valued functions with or without constraints.

As we said in the introduction to this chapter, during the 1960s optimization methods based on random changes of intermediate solutions were being designed and implemented at the Technical University of Berlin and applied to engineering hydrodynamics problems. In 1965, H.-P. Schwefel implemented the method on a computer in the framework of his Ph.D. work and called it a *two-membered evolution strategy*. The algorithm works as follows. A vector $\mathbf{x}$ of n real objective variables describing the parameters of the problem at hand is formed and a mutation drawn from

a normal distribution $\mathcal{N}(0, \sigma^2)$, with zero mean and the same variance σ^2 for all variables, is applied to each and every variable. The choice of a normal distribution is motivated by the observation that small variations are more common in nature than large ones. The new individual is evaluated with respect to the problem; if its objective function is higher (assuming maximization), then the new individual replaces the original one, otherwise another mutation is tried. The method was named Evolution Strategy $(1+1)$ or $(1+1)$-ES by Schwefel, where $(1+1)$ simply means that there is one "parent" individual from which one generates an "offspring" by the kind of mutation described above. The process is iterated until a given stopping condition is met, according to the following pseudo-code:

t = 0
$\mathbf{x}_t = (x_1, x_2, \ldots, x_n)_t$
while *not termination condition* **do**
    draw z_i *from* $\mathcal{N}(0, \sigma^2), \quad \forall i \in \{1, \ldots, n\}$
    $x_i^{'} = x_i + z_i, \quad \forall i \in \{1, \ldots, n\}$
    if $f(\mathbf{x}_t) \leq f(\mathbf{x}_t^{'})$ **then** $\mathbf{x}_{t+1} = \mathbf{x}'_t$
    else $\mathbf{x}_{t+1} = \mathbf{x}_t$
    t = t+1
end while

This first version of ES was not really an evolutionary algorithm in the sense we have given to the term since the concept of a population was absent. However, populations have been incorporated in later versions of ES, giving birth to the evolution strategies called $(\mu + \lambda)$-ES and (μ, λ)-ES. The main difference between the two types is in the selection method used to form the new population in each generation. In contrast with GAs and other EAs, evolution strategies make use of deterministic selection methods instead of probabilistic ones.

Let μ be the constant population size. Strategy $(\mu + \lambda)$-ES selects the μ best individuals for the next generation starting from a population formed by the union of μ parents and λ offspring. On the other hand, in (μ, λ)-ES the μ survivors are chosen among the offspring only, which implies $\lambda > \mu$. Strategies $(\mu + \lambda)$ do guarantee elitism but they have disadvantages in multimodal functions and can interfere in the auto-adaptation of strategy parameters, an important feature of modern ES. For these reasons, in today's ES the (μ, λ) selection/reproduction is the preferred one.

Let us now go into more detail on the concepts and notations involved in using evolution strategies as they are rather different from what we know in genetic algorithms, in spite of the common evolutionary inspiration. First, we explain the individual representation. In an l-dimensional parameter space solutions are represented by vectors **x** with l real components. However, the strategy itself manipulates l-dimensional vectors that are composed of three parts, ($\mathbf{x}$, $\boldsymbol{\sigma}^2$, $\boldsymbol{\alpha}$). $\boldsymbol{\sigma}^2$ may contain up to l variances σ^2: either all the variances are the same for all x_i, or there are l distinct variances. The third vector $\boldsymbol{\alpha}$ may contain up to $l(l-1)/2$ "rotation angles" α_{ij}. The variances and the α_{ij} are the parameters of the strategy but the rotation angles can be omitted in the simpler versions.

Mutation.

Mutation has always been the main variation operator in evolution strategies. It is simple enough in its original version, which we have seen above in two-membered strategies, but it has become more efficient, and also more complex, over the years. Therefore, we are going to discuss it in some detail. In the simplest case, the vector **x** is mutated by drawing random deviates from the same normal distribution with zero mean and standard deviation $\sigma = \sqrt{\sigma^2}$ and adding them to the components of vector **x**:

$$x_i^{'} = x_i + \mathcal{N}(0, \sigma^{'}), \quad i = 1, \ldots, l \tag{8.18}$$

It is important to point out that the variance itself is subject to evolution, hence the new standard deviation $\sigma^{'}$ in equation (8.18). We shall come back to variance evolution below. For the time being, we just remark that in equation (8.18) the mutation of σ takes place before the mutation of the parameter vector components, which means that the new components are computed with the new variance.

A more efficient approach to the mutation of the objective vector components consists of using different standard deviations for each component. The motivation for this lies in the observation that, in general, fitness landscapes are not isotropic, and thus mutations should be of different amplitudes along different directions to take this feature into account. This is schematically depicted in Figure 8.8. With a single variance for both coordinates, the new mutated point is bound to lie inside a circle around the original point, while two different variances will cause the mutated points to lie inside an ellipse. For the time being, we shall make the assumption that the l mutations are uncorrelated. In this case equations (8.19) describe the mutation mechanism of a vector **x**.

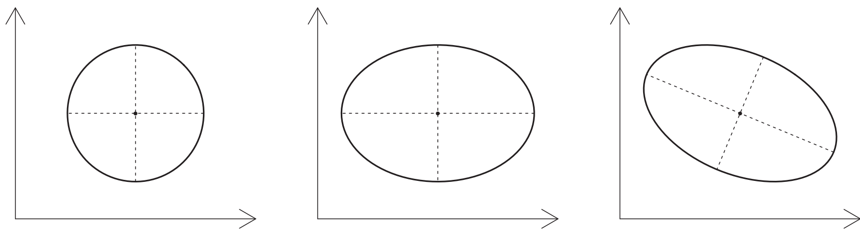


Fig. 8.8. Schematic illustration of mutation distributions with respect to a two-dimensional normal distribution. On the left the variance is the same for both coordinates. In the middle image variances are different for x and y. In the right image variances are different and covariance is introduced, which causes a rotation of the ellipsoid

$$x_i^{'} = x_i + \mathcal{N}_i(0, \sigma^{'}), \quad i = 1, \ldots, l, \tag{8.19}$$

where $\mathcal{N}_i(0, \sigma^{'})$ stands for a distinct normal deviate drawn for each variable of the problem.

Let us come back now to the variance auto-adaptation mechanism. In the simpler case in which mutations are drawn with a single variance for all variables (eq. 8.18), σ is mutated in each iteration by multiplying it by a term $e^{\mathcal{T}}$, where $\mathcal{T}$ is a random variable drawn from a standard normal distribution $\mathcal{N}(0, \tau)$ with 0 mean and standard deviation τ. Thus we have $\mathcal{N}(0, \tau) = \tau \cdot \mathcal{N}(0, 1)$. If the deviations $\sigma^{'}$ are too small their influence on the optimization process will be almost negligible. For this reason, it is customary to impose a minimal threshold ϵ on the size of mutations; if $\sigma^{'} < \epsilon$, then we set $\sigma^{'} = \epsilon$. The standard deviation τ is a parameter to be set externally, and it is usually chosen to be inversely proportional to the square root of the problem dimension l.

Recalling equation (8.18), the complete equations describing the mutation mechanism read

$$\sigma^{'} = \sigma \cdot \exp(\tau \cdot \mathcal{N}(0, 1)) \tag{8.20}$$

$$x_i^{'} = x_i + \sigma^{'} \cdot \mathcal{N}_i(0, 1), \quad i = 1, \ldots, l \tag{8.21}$$

The multiple-variance case, one for each problem dimension (eq. 8.19), can be treated in a similar manner except that each coordinate gets a specific variation. We are led in this case to the following equations describing the evolution of the variances and of the corresponding objective variables:

$$\sigma_i^{'} = \sigma_i \cdot \exp(\tau \cdot \mathcal{N}_i(0, 1) + \tau^{'} \cdot \mathcal{N}(0, 1)), \quad i = 1, \ldots, l \tag{8.22}$$

$$x_i^{'} = x_i + \sigma_i^{'} \cdot \mathcal{N}_i(0, 1), \quad i = 1, \ldots, l \tag{8.23}$$

Equation (8.22) is technically correct since the sum of two normally distributed variables is itself normally distributed. Conceptually, the term $e^{\tau^{'} \cdot \mathcal{N}(0,1)}$, which is common to all σ, provides a mutability evolution shared by all variables, while the term $e^{\tau \cdot \mathcal{N}_i(0,1)}$ is variable-specific and gives the necessary flexibility for the use of different mutation strategies in different directions.

With the above explanations under our belt, we are now able to present the most general formulation of evolution strategies with the auto-adaptation of all the strategy parameters $(\mathbf{x}, \sigma, \alpha)$ [9]. So far, we have used uncorrelated mutations, meaning that each coordinate-specific mutation is independent of the others. In two dimensions, this allows mutations to spread according to ellipses orthogonal to the coordinate axes, as depicted in Figure 8.8 (middle picture), instead of being limited to a circle (left picture), and the idea can easily be generalized to l dimensions. To get even more flexibility in positioning the mutation zones in order to cope with different problem

landscapes, we can also introduce the ellipsoid rotation angles α_{ij}, as schematically shown in the right picture of Figure 8.8. These angles are related to the variances and covariances of the joint normal distribution of l variables with probability density

$$p(\mathbf{x}) = \sqrt{\frac{\det \mathbf{C}^{-1}}{(2\pi)^l}} e^{-\frac{1}{2}\mathbf{x}^T\mathbf{C}^{-1}\mathbf{x}}, \tag{8.24}$$

where $\mathbf{C} = (c_{ij})$ is the covariance matrix of $p(\mathbf{x})$. In this matrix, the c_{ii} are the variances σ_i^2 and the off-diagonal elements c_{ij} with $i \neq j$ represent the covariances. In the previous uncorrelated case matrix $\mathbf{C}$ is diagonal, i.e., $c_{ij} = 0, i \neq j$. The l variances and $l(l-1)/2$ covariances (the matrix is symmetric) needed for parameter evolution are drawn from this general distribution, and the link between covariances and rotation angles α_{ij} is given by the expression

$$\tan(2\alpha_{ij}) = \frac{2c_{ij}}{\sigma_i^2 - \sigma_j^2} \tag{8.25}$$

Clearly, $\alpha_{ij} = 0$ for uncorrelated variables x_i and x_j since $c_{ij} = 0$ in this case. Finally, we can summarize the most general adaptation mechanism of ES parameters according to the following steps:

1. update the standard deviations according to the auto-adaptive lognormal method;
2. perturb the rotation angles according to a normally distributed variation;
3. perturb the objective vector by using the mutated variances and rotation angles.

This translates into the following equations:

$$\sigma_i^{'} = \sigma_i \cdot e^{(\tau \cdot \mathcal{N}_i(0,1) + \tau^{'} \cdot \mathcal{N}(0,1))} \tag{8.26}$$

$$\alpha_j^{'} = \alpha_j + \beta \cdot \mathcal{N}(0,1) \tag{8.27}$$

$$\mathbf{x}' = \mathbf{x} + \mathcal{N}(\mathbf{0}, \mathbf{C}') \tag{8.28}$$

Here $\mathcal{N}(\mathbf{0}, \mathbf{C}')$ is a random vector drawn from the joint normal distribution (equation 8.24) with $\mathbf{0}$ mean and covariance matrix $\mathbf{C}^{'}$. The latter is obtained from the $\sigma_i^{'}$ and the $\alpha_j^{'}$ previously computed. Notice that we have passed from a matrix notation for the α_{ij} to a vector notation thanks to the correspondence between $(i,j) \in \{1,\ldots,l-1\} \times \{1,\ldots,l(l-1)/2\}$ and the interval $\{1,\ldots,l(l-1)/2\}$ [9]. The suggested value for β is ≈ 0.087.

We conclude by saying that the usage of the normal distribution to generate perturbations is traditional and widely used because of its well-known properties and because it is likely to generate small perturbations. However, different probability distributions can be used if the problem needs a special treatment, without changing the essence of the methodology.

Recombination.

Recombination is the production of one or more offspring individuals starting from two or more parent individuals. While recombination, under the name of crossover, has always been a fundamental operator in genetic algorithms, it has been introduced later in evolution strategies, which are based on a sophisticated mutation mechanism. There exist a few recombination types in ES but the two most commonly used forms are *discrete recombination* and *intermediate recombination*, both taking two parent individuals and producing a single offspring. In discrete recombination the offspring parameter values are randomly chosen from one parent or the other with probability $1/2$. Thus, if x_i and y_i are the components of parents **x** and **y**, the components of the product of their recombination **z** will be

$$z_i = x_i \quad \text{or} \quad y_i, \quad i = 1, \ldots, l$$

In intermediate recombination the parents' components are linearly combined:

$$z_i = \alpha\, x_i + (1 - \alpha)\, y_i, \quad i = 1, \ldots, l$$

Often $\alpha = 0.5$, which corresponds to the average value.

The above recombination mechanisms are the commonest ones in ES but more than two parents are sometimes used too. Another recombination method, called *global recombination*, takes a randomly chosen parent in the population and, for each component of the offspring, a new individual is randomly chosen from the same population. In this technique, the offspring components can be either obtained by discrete or intermediate recombination.

To conclude this section, we note that usually not only the objective variables of the problem but also the other strategy parameters are submitted to recombination, possibly using different methods. For instance, discrete recombination is often used for the objective variables and intermediate recombination for variances and rotation angles. The interested reader will find more details in, e.g., [9].

An overview of theoretical results in evolution strategies.

It is out of the question to go into the details of this mathematically highly developed field. Here we can only give a glimpse of the various results that have been obtained over the years by stating a classical result, but the reader will find a good introduction to the theory of ES in [9] and in the original literature.

One of the first remarkable theoretical results is the so-called "$1/5$ success rule" proposed by I. Rechenberg in 1973 for the two-membered strategy $(1 + 1)$-ES. Rechenberg derived this rule for two specific basic objective functions: the "corridor" function and the "sphere" function. By using the convergence rate expressions, he was able to derive the optimal value of the standard deviation of the mutation operator and the maximal convergence rate. The rule states that the ratio between the improving mutations and their total number should be $1/5$. Thus, if the ratio is larger σ should be increased, while it should be decreased if it is smaller.

This rule is historically interesting but only applies to old-style $(1 + 1)$-ES and not really to modern ES using a population and parameter evolution. More recently, theoreticians have obtained more general and rigorous results about the convergence in probability for $(\mu + \lambda)$-ES and more. Again, we refer the reader to the book [9] for a good introduction to the field.

9

Evolutionary Algorithms: Extensions

9.1 Introduction

In this chapter we give a more detailed description of genetic operators, and in particular of selection. We will also introduce a more advanced evolutionary method called Genetic Programming, which makes use of complex, variable-sized individual representations. Finally, we will discuss the possibilities that arise when there is a topological structure in the evolving populations.

Formally, we can make a distinction between variation operators and selection/reproduction. The classical variation operators are crossover and mutation. Their role is the creation of new individuals in order to maintain some diversity in the population. These operators depend on the particular representation that has been chosen. For example, in the last chapter we saw the binary representation used in classical genetic algorithms, and the real-number-based representation typical of evolution strategies. The corresponding genetic operators, although they are inspired by the same biological idea of reinjecting novelty into the population, are implemented in different ways that are suited to the representation being used. We will see that the same happens in genetic programming, a more recent evolutionary algorithm that will be introduced later in the chapter, and also when we must deal with combinatorial optimization problems, for which specialized individual representations are needed. Selection is different from this point of view. All a selection method needs is a fitness value to work with. It follows that, in a way, all selection methods are interchangeable and independent of other parts of evolutionary algorithms, in the sense that different selection methods can be "plugged" into the same evolutionary algorithm according to the user's criteria. Because of their fundamental role, selection methods deserve a detailed treatment and this is what we are going to do in the next section.

9.2 Selection

As we have recalled several times already, the goal of selection is to favor the more adequate individuals in the population, and this in turn means that selection leads the

© Springer Nature Switzerland AG 2018

B. Chopard, M. Tomassini, *An Introduction to Metaheuristics for Optimization*,
Natural Computing Series, https://doi.org/10.1007/978-3-319-93073-2_9

algorithm to focus the search on promising regions of the space. Again, we draw the attention of the reader to the distinction between exploitation/intensification of good solutions and exploration/diversification, a compromise that is common to all metaheuristics. Exploration by itself is unlikely to discover very good solutions in large search spaces. Its limiting case is random search, which is extremely inefficient. Conversely, when exploitation is maximal the search degenerates into hill climbing, which, considered alone, is only useful for monomodal functions, a trivial case that does not arise in practice in real applications. Therefore, in order for an evolutionary algorithm to function correctly, the available computational resources must be allocated in such a way as to obtain a good compromise between these two extreme tendencies. Admittedly, this goal is easy to state but difficult to attain in practice. However, selection may help to steer the search in the direction sought: weak selection will favor exploration, while more intense selection will favor exploitation of the best available individuals. In what follows we shall present the more common selection methods and their characteristics, and we will define the intensity of a selection method in a more rigorous way in order to compare them.

9.2.1 Selection Methods and Reproduction Strategies

Selection methods can be classified according to a few different criteria; here we will divide them into *deterministic* and *stochastic* methods. Deterministic methods attribute to an individual a probability of survival of zero or one; in other words, the individual either survives or it is eliminated from the population with certainty. This type of selection method is typical in evolution strategies and we refer the reader to the last chapter for the details. Stochastic methods, on the other hand, attribute a positive probability of survival to any individual and this probability will be higher, the higher the quality of the solution that the individual represents. According to some researchers, stochastic selection is to be preferred over deterministic selection as it allows the survival of relatively weak individuals that would otherwise be eliminated. This improves population diversity and may help avoid too much exploitation.

In the framework of generational evolutionary algorithms, as we have described them here, it is useful to consider three populations: the current population in iteration t, $P(t)$, the intermediate population $P^{'}(t)$, and the next-generation population $P(t+1)$. This situation was schematized in the previous chapter in Section 8.2.3, in which the crossover and mutation phases, which we need not take into account here, were also present. Selection operates on $P(t)$ and the selected individuals go into the intermediate population $P^{'}(t)$ one by one until the original population size n is reached[1]. As we have already remarked in the previous chapter, it may well be that some good individuals are selected more than once and some weak ones could disappear altogether. The process of forming a population with the selected individuals is called *reproduction*. Following this, the variation operators are applied to the individuals in the intermediate population to produce the next generation population

[1] Evolutionary algorithms with variable-size populations have been sometimes used but they are rather uncommon.

$P(t+1)$ and, after fitness evaluation, the cycle starts again until a termination condition is reached.

Below, we now present in some detail the stochastic selection methods more commonly used starting with fitness-proportionate selection.

Proportionate selection.

This is the original GA selection method proposed by J. Holland that we already met a few times in the previous chapter. It will be analyzed in more detail here. In this method, the expected fraction of times a given individual is selected for reproduction is given by its fitness divided by the total fitness of the population. The corresponding probability p_i of selecting individual i whose fitness is f_i, is given by the following expression:

$$p_i = \frac{f_i}{\sum_{j=1}^{n} f_j}$$

In the program codes that implement EAs with fitness-proportionate selection these probabilities are computed numerically by using the so-called "roulette wheel" method. On this biased virtual roulette, each individual gets a sector of the circle whose area is proportional to the individual's fitness. The roulette wheel is "spun" n times, where n is the population size, and in each spin the individual whose sector receives the ball will be selected. The computer code that simulates this special roulette uses an algorithm analogous to the general one presented in Chapter 2, Section 2.8.1. It is rewritten here in EA style for the reader's convenience:

1. compute the fitness f_i of each individual $i = 1 \dots n$
2. compute the cumulated fitness of the population $S = \sum_{j=1}^{n} f_j$
3. compute the probability p_i for an individual i to be selected: $p_i = f_i/S$
4. compute the cumulated probability P_i for each individual i: $P_i = \sum_{j=1}^{n} p_j$

- repeat n times:
  1. draw a pseudo-random number $r \in [0, 1]$
  2. if $r < P_1$ then select individual 1, otherwise select the ith individual such that $P_{i-1} < r \leq P_i$

This method is straightforward but it can suffer from sampling errors because of the high variance with which individuals are selected in relatively small populations. Countermeasures to this effect have been documented in the literature, for example *stochastic universal sampling*, whose description is beyond our scope. For more details see, e.g., [9]. However, there can be other problems when using fitness-proportionate selection. In rare cases, the occasional presence in the population of an individual with much better fitness than the rest, a so-called *superindividual*, may cause the roulette wheel method to select it too often, leading to a uniform population and to premature convergence.

Another more common and more serious problem with this selection method is that it directly employs the fitness values of the individuals. To start with, fitness cannot be negative otherwise probabilities would be undefined. Next, even for positive fitness values, minimization values cannot be treated directly, they must first be transformed into equivalent maximization ones using the fact that $\max(f) = -(\min(-f))$. Finally, even if we can solve all of the above problems, it remains the fact that, as evolution progresses, the population tends to lose its diversity to a growing extent with time. This means that the fitness values associated with the individuals become more and more similar (see, e.g., Figure 8.1 and Table 8.1 in Chapter 8). In the roulette wheel analogy this would produce circular sectors of very similar size; thus selection probabilities would also be similar, leading to an almost uniform random choice with a consequent loss of selection pressure. To overcome this problem, researchers have proposed to transform the fitness function as time goes by (fitness scaling) in various ways such that fitness differences are amplified, thus allowing selection to do its job. These methods are described in the specialized literature, see, e.g., [60]. Fortunately, most of the problems inherent to fitness-proportionate selection can be avoided by using the alternative methods described below.

Ranking selection.

In this method, individuals are sorted by rank from rank 1, attributed to the individual with the best fitness, down to rank n. The probability of selection of an individual is then calculated as a function of its rank, higher-ranking individuals being more likely to be selected. The remarkable thing with this method is that the actual fitness values do not play a role anymore; only the rank counts. This method maintains a constant selection pressure as long as the individuals have different fitnesses and thus avoids most of the problems caused by the fitness-proportionate method. The original ranking method is *linear ranking*. It attributes selection probabilities according to a linear function of the rank:

$$p_i = \frac{1}{n}\left[\beta - 2(\beta - 1)\frac{i-1}{n-1}\right], \qquad i = 1, \ldots, n \tag{9.1}$$

where index i refers to the rank of the considered individual and $1.0 < \beta \leq 2.0$ is a parameter representing the expected number of copies of the best-ranked individual. The intensity of selection can be influenced by the β parameter: the higher it is, the higher the selection pressure.

The value of p_i as a function of the rank i is reported in Figure 9.1 for $n = 20$ and $\beta = 1.8$. The horizontal dashed line corresponds to $\beta = 1$, which causes the probability of choosing any individual to be uniform and equal to $1/n$, as the reader can readily check by substituting values in equation (9.1). When β becomes larger than 1, the best-ranked individuals see their selection probability increase, while the low-ranked ones have lower probabilities. The difference increases with growing β up to its maximum value $\beta = 2$.

Non-linear ranking schemes have also been proposed as a means of increasing the selection pressure with respect to linear ranking.

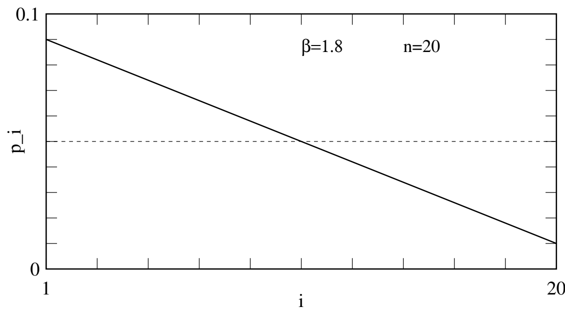


Fig. 9.1. The probability p_i of choosing the ith-ranked individual in linear-ranking selection. The case with $n = 20$ and $\beta = 1.8$ is shown

Tournament selection.

This method is probably the simplest one in theoretical terms and it is also easy to implement. The main advantage is that there is no need to make hypotheses on the fitness distribution in the population, the only condition is that individual fitnesses be comparable. For example, if the EA goal is to evolve game strategies, we can just compare two strategies by simulating the game and decide which one is better as a function of the result. As another example, it is rather difficult to precisely define the fitness of an autonomous robot when it performs a given task but, thanks to suitable measures, roboticists will be able to attribute an overall fitness to two different paths and compare them to decide which one is better. Other examples come from evolutionary art, or evolutionary design, in which humans look at given designs, compare them, and decide which ones must be kept.

The simplest form of tournament selection is a binary tournament. Two individuals are randomly drawn from the population with uniform probability and their fitnesses are compared. The "winner", i.e., the individual that has the better fitness, is copied into the intermediate population for reproduction, and it is replaced in the original population. Since the extraction is done with replacement, an individual may be chosen more than once. The process is repeated n times until the intermediate population has reached the constant initial size n.

The method can be generalized to tournaments with k participants, where $k \ll n$ and typically of the order of two to ten. The induced selection pressure is directly proportional to the tournament size k. The stochastic aspect of this method comes primarily from the random draw of the tournament participants. The winner can be chosen deterministically, as above, or with a certain probability, which has the effect of lowering the selection pressure.

9.2.2 Selection Intensity

We have already alluded several times to the idea of selection intensity without really defining the concept in a clear manner. We do this now by using the concept of *takeover time* that characterizes the intensity of selection induced by the different selection mechanisms described above. The takeover time τ is defined as being the time needed for the best individual to conquer the whole population under the effect of selection alone. At the beginning there are two types of individuals in the population: one individual with a higher fitness, and $n-1$ individuals with a lower fitness. The application of the selection operator to such a population causes the best individual to increase its frequency in the population by replacing the less good individuals, until the whole population is constituted by copies of the high-fitness individual. The idea is that short τ characterizes strong selection, while longer τ is typical of weaker selection methods. The theoretical derivation of the takeover times is tricky (but see Section 9.2.3 for examples of how this can be done). The general results are that those times are $\mathcal{O}(\log n)$, where n is the population size. The proportionality constants may differ according to the particular method used. Thus, we know that, in general, ranking and tournament selection are more intense than fitness-proportionate selection. With respect to evolution strategies, without giving technical details, it has been found that the selection pressure associated with $(\mu + \lambda)$-ES and (μ, λ)-ES is even stronger than for ranking and tournament selection. This implies that evolution strategies must rely on variation operators to maintain sufficient diversity in the population, since the selection methods are rather drastic and they tend to exploit the current solutions. The interested reader will find details in the book [9].

The theoretical predictions can easily be tested by numerical simulations. The growth curves of the best individual are of the "logistic" type since the problem is formally analogous to the growth of a population in a limited capacity environment. This situation, in the continuum approximation, is described by the Verhulst differential equation

$$\frac{dm}{dt} = rm(1 - m/n),$$

where r is the growth rate, m is the current number of copies of the best individual, and n is the maximal capacity, i.e., the number of individuals that can be sustained. The solution of this equation, denoting by m_0 the number of copies of the best individual at time 0, is

$$m(t) = \frac{m_0 n e^{rt}}{[n + m_0(e^{rt} - 1)]}$$

which explains the term logistic function. There is an exponential increase of the population at the beginning but, after the inflection point, the growth rate decreases and asymptotically tends to the carrying capacity n since for $t \to \infty$, $m(t) \to n$.

Figure 9.2 shows the results of numerical simulations; it reports the average of one hundred numerical runs showing such a behavior for binary tournament selection and a population size of 1,024.

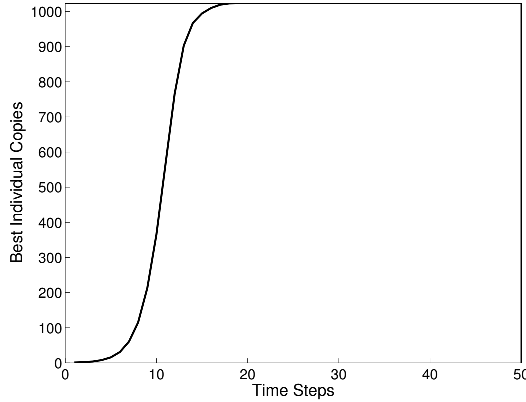


Fig. 9.2. Takeover time curve for binary tournament selection in a population of 1,024 individuals. The curve is the average of 100 executions

9.2.3 Analytical Calculation of Takeover Times

In this section we give an analytical proof of the general results mentioned in the previous section, i.e., that the number of copies of the best individual is bounded by a logistic curve and that the takeover time is logarithmic in the population size, at least in fitness-proportionate selection and tournament selection. This section can be omitted on a first reading.

Fitness-proportionate selection

Let $m(t)$ be the average number of copies at time t of the best individual in a population $P(t)$ of size n. In fitness-proportionate selection (all fitnesses are assumed positive and fitness is maximized) we have

$$m(t+1) = n\frac{m(t)f_1}{F_{tot}(t)} \geq n\frac{m(t)f_1}{m(t)f_1 + (n - m(t))f_2} \tag{9.2}$$

where $F_{tot}(t) > 0$ is the total fitness of population $P(t)$, $f_1 > 0$ is the fitness of the best individual, and f_2 is the fitness of the second-best individual: $0 < f_2 < f_1$. The above inequality is justified since

$$F_{tot}(t) \leq m(t)f_1 + (n - m(t))f_2 \tag{9.3}$$

To simplify, we take $m(t) = m$. We can now write

$$\begin{aligned} m(t+1) &\geq m + n\frac{mf_1}{mf_1 + (n-m)f_2} - m\frac{mf_1 + (n-m)f_2}{mf_1 + (n-m)f_2} \\ &= m + \frac{nm(f_1 - f_2) - m^2(f_1 - f_2)}{mf_1 + (n-m)f_2} \end{aligned} \tag{9.4}$$

We set

$$\Delta = f_1 - f_2 \tag{9.5}$$

with $\Delta > 0$ and $\Delta/f_1 < 1$ since $0 < f_2 < f_1$. The equation becomes

$$m(t+1) \geq m + \frac{nm\Delta(1 - m/n)}{nf_1 - (n-m)\Delta} \tag{9.6}$$

$$= m + m\frac{\Delta}{f_1}\left(1 - \frac{m}{n}\right)\left(\frac{1}{1 - \left(1 - \frac{m}{n}\right)\frac{\Delta}{f_1}}\right) \tag{9.7}$$

Given that $m \leq n$ and $\Delta/f_1 \leq 1$,

$$\left(\frac{1}{1 - \left(1 - \frac{m}{n}\frac{\Delta}{f_1}\right)}\right) \geq 1 \tag{9.8}$$

We thus get

$$m(t+1) - m(t) \geq m\frac{\Delta}{f_1}\left(1 - \frac{m}{n}\right) \tag{9.9}$$

We can solve this equation by reformulating it as the following differential equation

$$\dot{m} = m\frac{\Delta}{f_1}\left(1 - \frac{m}{n}\right) \tag{9.10}$$

whose solution is (with $r = \Delta/f_1$)

$$m(t) = \frac{n}{1 + \frac{n-m_0}{m_0}e^{-rt}} \tag{9.11}$$

where $m_0 = m(0) = 1$. With

$$\tau = \frac{\ln(n-1)}{r} \tag{9.12}$$

we obtain that

$$m(\tau) = \frac{n}{2} \tag{9.13}$$

which shows that the takeover time of the discrete process is

$$\tau \leq \mathcal{O}(\ln n) \tag{9.14}$$

The above derivations are illustrated by the numerical simulation results depicted in Figure 9.3. In this figure, we consider a population size of $n = 100$. The fitnesses of each individual are randomly chosen to lie between 0 and $f_2 = 100$. The fitness of the individual $n/2$ is then readjusted to $f_1 > 100$ in order to guarantee that there is only one best individual. We observe that the number of copies of the best individual grows faster than what is predicted by the logistic equation (9.11), as stated in equation (9.9).

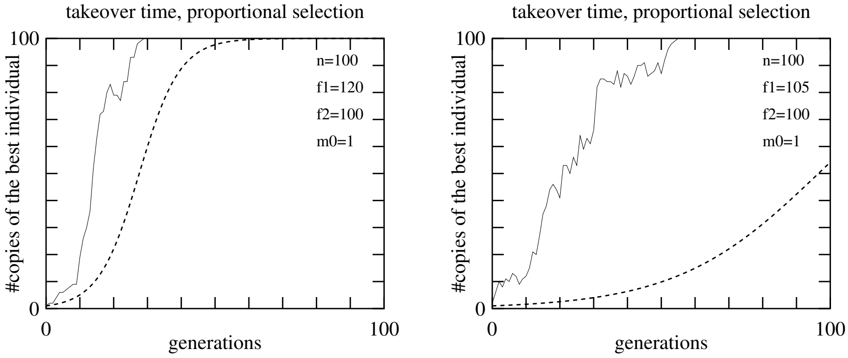


Fig. 9.3. Numerical simulation of the takeover time for fitness-proportionate selection for different values of f_1 and f_2. The dashed line is the solution of equation (9.11)

Tournament selection

In the case of tournament selection with k individuals, the probability of not drawing the best individual k times is $(1 - m/n)^k$, where m is the current number of copies of the best individual and n the population size.

Thus, the average number of copies of the best individual in iteration $t + 1$ is

$$\begin{aligned} m(t+1) &= n \left(1 - \left(1 - \frac{m}{n}\right)^k\right) \\ &\geq n \left(1 - \left(1 - \frac{m}{n}\right)^2\right) \end{aligned} \tag{9.15}$$

The inequality results from $k \geq 2$, which implies $(1 - m/n)^k \leq (1 - m/n)^2$. The equation becomes

$$m(t+1) \geq n \left(2\frac{m}{n} - \frac{m^2}{n^2}\right) \tag{9.16}$$

and, after rearranging and recalling that $m = m(t)$ this gives

$$m(t+1) - m(t) \geq m \left(1 - \frac{m}{n}\right) \tag{9.17}$$

which is again a logistic equation with a takeover time that is logarithmic in n.

9.3 Genetic Programming

Genetic programming (GP) is a more recent evolutionary approach to problem solving that was essentially introduced by J. Koza and others towards the end of the

1980s [52]. The basic idea in genetic programming is to make a population of programs evolve with the goal of finding a program that solves, exactly or approximately, a predefined task. At first sight the idea would seem almost impossible to put into practice. There are indeed many questions that seem to have no satisfactory answer. How are we going to represent a program in the population? And what do we mean exactly by program mutation and program crossover? If we use an ordinary procedural programming language such as Java or C++, it is indeed difficult to imagine such genetic operations. The random unrestricted mutation of a program piece would almost certainly introduce syntax errors or, in the unlikely case that the resulting program is syntactically correct, it would hardly compute something meaningful. However, a program can be expressed in computationally equivalent forms that do not suffer from such problems or, at least, that can be more easily manipulated. Functional programming in particular is suitable for artificial evolution of programs and J. Koza used LISP in his first experiences with GP. However, even with an adequate syntactic form, unrestricted evolution of programs would have to search a huge space and it would be unlikely to find interesting results. It is enough to try to imagine how such a system would find a working compiler or operating system program: very unlikely indeed, in a reasonable amount of time. But a complex program such as a text-processing system or a compiler is certainly best produced by using solid principles of algorithmics and software engineering. In this case, designers can make use of modules and abstractions that encapsulate functionalities and can combine them in meaningful ways. It would be almost hopeless to find such abstractions and their hierarchies and structures by evolution alone. Koza clearly realized these limitations and proposed to work with a restricted set of operations and data structures that are task-specific. In this reduced framework, GP has become a powerful method of problem solving by program evolution when the problem at hand is well defined and has a restricted scope. In this sense, GP can be seen as a general method for machine learning rather than a straight optimization technique. However, many automated learning problems can be seen from an optimization point of view and GP is a powerful and flexible way of approximately solving them. In our opinion, it is thus perfectly justified to present the basis of the methodology here.

9.3.1 Representation

The choice of individual representation in GP is more difficult compared to other evolutionary algorithms. When faced with a problem to solve by evolving a suitable program, the first step for the user is to define two sets of objects: the *Terminal* set T and the *Function* set F. The function set contains the functions that are considered useful *a priori* for solving the problem at hand, and the terminal set is constituted by the variables and the constants that characterize the problem. The F and T sets must obey the *closure* property: each function must accept as arguments all the value types returned by the other functions, and all the values in T. The space of the possible programs is then formed by all the function compositions possible from the elements of F and T, a huge and potentially infinite space, but we shall see that practical constraints will be effective in bounding it.

A simple example, which is unrealistic but good enough for illustrating the ideas, is arithmetic expressions. Suppose that we wish to use GP to generate and evolve arbitrary arithmetic expressions with the usual operations and four variables at most. In this case the sets F and T may be defined as follows:

$$F = \{+, -, *, /\}$$

and

$$T = \{\mathtt{A}, \mathtt{B}, \mathtt{C}, \mathtt{D}\}$$

The following programs, for instance, would be valid in such a restricted language universe: `(+ (* A B) (/ C D))`, and `(* (- (+ A C) B) A)`.

A second example, equally simple but more useful in practice, is Boolean logic. In this case, a reasonable function set might be the following:

$$F = \{\mathtt{AND}, \mathtt{OR}, \mathtt{NOT}\}$$

together with a set of terminals that includes some Boolean variables and constants:

$$T = \{\mathtt{b}_1, \mathtt{b}_2, \mathtt{b}_3, \ldots, \mathtt{TRUE}, \mathtt{FALSE}\}$$

It is important to point out that genetic programs need not be expressed directly in a functional form such as a subset of LISP S-expressions; it is equally possible to use the corresponding parse tree form. Thus, in Figure 9.4 the expressions above are illustrated in equivalent tree form, where the internal nodes are functions and the leaves are terminals. Today, most GP software systems are written in conventional languages such as C++ or Java and use dynamic memory allocation to build and manage the trees. Trees are combinatorial structures whose size tends to increase exponentially fast, causing increasing evaluation times. This practical limitation has led researchers to bound the tree depth in order to avoid too-heavy computations. Finally, trees are not the only possibility: linear genomes (see section 9.4 in this chapter), graphs, and grammars have also been used.

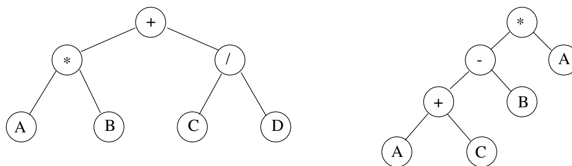


Fig. 9.4. Two trees that are equivalent to the S-expressions in the text

While we have just seen that the individual representation is fundamentally different in GP with respect to other evolutionary algorithms, the evolutionary cycle itself is identical to the GA pseudo-code shown at the beginning of Chapter 8. Once

suitable sets F and T for the problem have been found, an initial population of programs is randomly created. The fitness of a program is simply the score of its evaluation (see below), and suitable genetic operators are then applied to the population members.

9.3.2 Evaluation

Attributing a fitness score to a GP individual, that is, evaluating the corresponding program, is different from evaluating the fitness of a fixed-length bit string or a vector of reals. In GP, the program is executed and its performance evaluated: this value corresponds to its fitness. Formally, the individual coding and its evaluation take place in the same space, the space of programs generated by the F and T sets. A program can be evaluated exactly if all the possible inputs to the program are known and if this number is not too large; otherwise, an approximate evaluation can be done. The first case may be represented by a Boolean problem with k variables. The number of possible inputs is 2^k and, if k is not too large, all the possible cases can be tested. In other situations we might use a restrained number of representative test cases. Once the test cases have been defined, the performance $f(P_i)$ of a program P_i can be computed as the sum of the differences between the expected outputs G_k and the actual outputs g_k of the program P_i:

$$f(P_i) = \sum_{k=1}^{N} \parallel g_k - G_k \parallel \tag{9.18}$$

Consequently, the fitness of a program is maximal when the performance $f(P_i) = 0$.

9.3.3 Genetic Operators

Using the tree representation for the programs in the population, there exist different forms for the genetic operators of crossover and mutation. The following two are the original ones introduced by Koza and still in wide use. Mutation of a program is implemented by selecting a random subtree and replacing it with another tree randomly generated from the problem's F and T sets. The process is schematized in Figure 9.5.

Crossover is performed by starting from two parent trees and selecting a random link in each parent. The two subtrees thus defined are then exchanged (see Figure 9.6). The links to be cut are usually chosen with non-uniform probability, so as to favor the links that are closer to the root because the latter are obviously less numerous than the links that are lower in the tree, which would be more likely to be chosen with uniform probability.

Finally, we remind the reader that the operators presented here are adapted to the tree representation of programs. If other representations are used, such as linear, graphs, or grammars, different operators must be designed that are suited to the adopted representation.

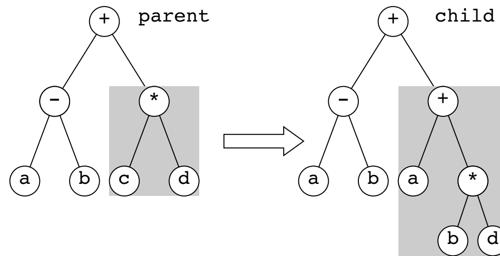


Fig. 9.5. Example of mutation of a tree representing an arithmetic expression

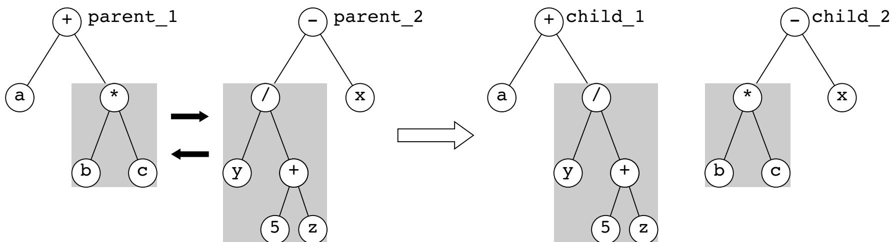


Fig. 9.6. Example of crossover of two program trees

9.3.4 An Example Application

Genetic programming has been applied with great success to all kinds of problems in many fields from analogical and digital circuit design, to game programming, non-linear systems, and autonomous robotics just to name a few. The interested reader will find more information on the many uses of this powerful technique in Koza's review article [54]. However, genetic programming is a more complex technique than standard evolutionary algorithms. The first step, which consists of choosing the appropriate primitives for the problem, is a delicate point. The problem itself usually suggests what could be reasonable functions and terminals but the choice has an influence on the results. In fact, choosing the right functions and terminals is still a trial-and-error process; a good idea is to start small and to possibly add new primitives if needed.

To give an example of the use of GP on a real problem, we now briefly summarize its application to a problem in the financial field. The application consists of developing *trading models*, which are decision support systems, in the field of foreign exchange markets [21]. The model is built around a system of rules that combine relevant market indicators, and its output is a signal of the type "buy," "sell" or "hold" as an explicit recommendation of the action to be taken. For example, a very simple trading model could have the following flavor:

$$\texttt{IF } |\texttt{I}| > \texttt{K THEN G} := \texttt{SIGN(I) ELSE G} = 0$$

`I` is an *indicator* that models the current market trend and that can be technically complex. Indicators of this kind are based on price time series and can be calculated in advance. `K` is a threshold value imposed by the human trader. The result `G` is a signal that can take the values 0, 1, or -1, which correspond, respectively, to a neutral position (hold), to sell, or to buy the currency in question. Real models are more complex but the ideas are the same.

For this application the following functions were chosen:

$$F = \{\texttt{AND}, \texttt{OR}, \texttt{NOT}, \texttt{IF}\}$$

The terminals are

$$T = \{+1, -1, 0, \texttt{I}_\texttt{i}\},$$

where the $\texttt{I}_\texttt{i}$ are indicators computed as moving averages of price time series. The logical operations have been redefined in order to function correctly in this context. Thus, the function `OR` returns the sign of the sum of its arguments, function `NOT` returns the inverse of the decision of its argument, and function `AND` returns the neutral signal 0 when one of its arguments is 0, otherwise it returns the `OR` of its arguments. The function `IF` takes three arguments and returns the second if the first is `TRUE`, otherwise it returns the third argument.

The fitness of a trading model is its performance measure. Performance quantifies the *return* provided by the application of the trading model, but it also takes into account the associated *risk*. The return is the total gain accumulated over all the transactions in the sequence of test cases. The technical details are relatively complex and can be found in the original works. Here we just recall that the fitness function for a trading model tm_i is defined as

$$f(tm_i) = \langle R \rangle - \frac{C}{2}\sigma^2$$

In the previous expression $\langle R \rangle$ is the annualized average gain, $C = 0.1$ is a risk aversion constant, and σ^2 is the variance of the accumulated gain during the considered period. A desirable quality for a trading model is *robustness*, which in this context means that the model should not overfit the data and that it should be able to generalize correctly. In the present application this is obtained by using several time series belonging to different currency exchange rates.

The GP system that was implemented for the problem is a standard one, with tree mutation and crossover as defined above. Selection is done by tournament and the best trading model of the current generation goes without change into the next-generation population. Because of the lengthy indicator computation, the application was executed on a computer cluster with a maximal time corresponding to 100 generations as a termination criterion. The population size was 100 and the algorithm was executed 10 times on the training data. The best models resulting from each execution have been tested on unseen data to evaluate their generalization capabilities. For the sake of illustration, we remark that one of the best trading models in terms

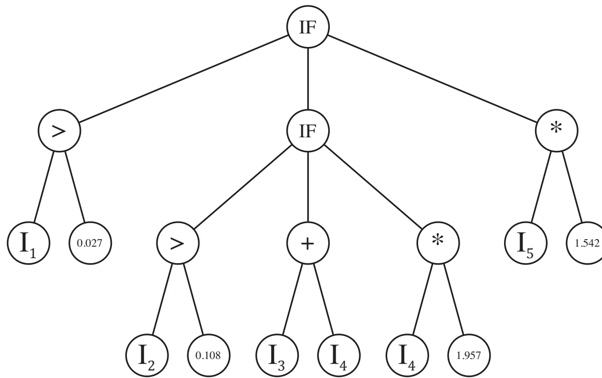


Fig. 9.7. A decision tree evolved by genetic programming in the field of trading models

of performance that has been evolved by the GP system corresponds to the tree in Figure 9.7. In this tree, I_1, I_2, I_3, and I_4 are complex indicators based on moving averages of price time series of different lengths. The constants are suitable empirical values dictated by experience.

9.3.5 Some Concluding Considerations

In this section we try to summarize some aspects of GP that appear when the system is used in practice. A common problem in GP is the inordinate growth of the trees representing the programs in the evolving population. This growth, also called "bloat" in the field, is due to the application of genetic operators, especially crossover. Program bloat has unpleasant consequences such as stagnation in the fitness evolution and a slowdown in the evaluation of trees as they become larger and larger. Some simple measures may be effective at fighting bloat. The first thing to do is to limit the maximum tree depth from the beginning, a feature that all GP systems possess. Another approach consists of introducing a penalty term in the fitness function that depends on program length, in such a way that longer programs see their fitness reduced.

Genetic programming is at its best on well-defined problems that give rise to relatively short programs. We have seen that this also depends on the choice of the terminal and function sets. GP can be extended to more complex problems but in this case a hierarchical principle becomes necessary. This can be accomplished by encapsulating subprograms and pieces of code that seem to play the role of building blocks in the evolution of good solutions. The inspiration here comes from standard software engineering techniques and, to dig deeper into the subject, we refer the reader to the specialized literature, starting with the books [53, 10].

To conclude, we observe that GP has opened up a whole new chapter in evolutionary problem solving and design. The theory of GP turns out to be more difficult than that of ordinary EA and it would be out of scope to present it here, even in

summarized form. We can just say that, among other things, results that generalize schema theories to GP have been obtained.

In the next section we present another style of genetic programming called *linear genetic programming*. Although the main principles are the same as in tree-based GP, linear genetic programming possesses some particularities that make it worth a separate discussion.

9.4 Linear Genetic Programming

In this section we present a different and much less well-known approach to genetic programming than the tree-based version. This version is based on a linear representation of a program, much closer to standard procedural languages than the functional form. The interest of the approach lies in its simplicity and in the fact that programs are of fixed size; besides, control structures such as branching and looping are easier to construct. Such a program is constituted by a set of instructions that are executed sequentially on one side, and a stack that can contain an arbitrary number of numerical values on the other. The program instructions take their arguments from the stack and push their results onto the same stack. The final result is also to be found on the stack, for example as the most recent value on top of it.

We are now going to describe this process in more detail with inspiration coming from other stack-based languages such as Forth or Postscript. Let us first consider the program variables that were called terminals in tree-based GP. For instance, the variable `A` is considered here as an instruction that places the value of `A` on the stack. The program that computes $\frac{(A+B)\times C}{\sqrt{2}}$, where A, B, and C are three variables, is given by

```
A B ADD C MUL 2 sqrt DIV
```

where `ADD` is an instruction that takes the two top elements on the stack and pushes their sum onto the stack. The instructions `MUL` and `DIV` do the same for multiplication and division respectively. The instruction `sqrt` takes the element on top of the stack and returns its square root to the stack.

Clearly, to guarantee program execution and avoid faults and exceptions, the behavior of instructions must be defined when the number of parameters is insufficient, or when values are inconsistent, for example taking the square root of a negative number. Here such exceptions are simply ignored and the operation is not executed. Thus, for instance, instruction `2 ADD` will leave the stack unchanged with the value 2 on top of it.

An execution engine for linear GP programs is easy to build, for example using the Python language, by making a correspondence through a dictionary, i.e., a hash table, between each program identifier and a preprogrammed function that uses the stack.

With linear programs, the genetic operations of mutation and crossover are very similar to those presented in Chapter 8 for genetic algorithms. For example, one

can use one-point crossover, as shown below for the two programs `P1` and `P2`. The symbol | marks the crossing point and is randomly chosen.

```
P1: A B ADD | C MUL 2 sqrt DIV
P2: 1 B SUB | A A MUL ADD 3
```

After crossover, the following two programs will result:

```
P1′: A B ADD A A MUL ADD 3
P2′: 1 B SUB C MUL 2 sqrt DIV
```

We can verify that `P1′` leaves two values on the stack: $A + B + A^2$ and 3, while `P2′` leaves only one value: $(1 - B) * C/(\sqrt{2})$.

A mutation is generated by randomly exchanging an instruction or a variable with another instruction or variable and the probabilities of choosing a variable or an instruction may be different.

9.4.1 Example

As an illustration of the workings of linear genetic programming, we will let the system find the Boolean function f of three variables x_0, x_1, x_2 given by the truth table

x_0	x_1	x_2	$f(x_0, x_1, x_2)$
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1

By simple inspection, we can see that the sought function is

$$f(x_0, x_1, x_2) = x_0 \text{ and } x_1 \text{ and } x_2$$

but we would like to have GP discover it by itself. Let's consider the terminal set

$$T = \{X_0, X_1, X_2\}$$

and the function set

$$F = \{\text{AND}, \text{OR}, \text{NOT}\}$$

which is clearly sufficient to express any Boolean function.

For this example, we consider programs consisting of only five instructions. The output of the program is to be found as the top element of the stack after its execution. With six different possible choices for each instruction the number of possible

different programs is $6^5 = 7{,}776$. Clearly, an exhaustive search would be possible in this case but the method cannot be extended to large program space sizes.

Let's suppose a small population of $n = 12$ linear programs of length five, initially chosen at random. We let them evolve during 50 generations according to the standard cycle: selection, crossover, and mutation. A program's fitness function is the number of f values of the function table above that are correctly computed. The training set is given by the eight possible combinations of the variable triplet (x_0, x_1, x_2), therefore the optimal program has a fitness of eight.

Below, we see a population of 12 programs that has been randomly generated at time 0:

```
Initial population:
['OR', 'NOT', 'X2', 'OR', 'X0'] fitness= 5
['X1', 'X2', 'OR', 'X0', 'OR'] fitness= 2
['OR', 'X2', 'OR', 'X0', 'X0'] fitness= 5
['OR', 'X2', 'X2', 'X1', 'X1'] fitness= 5
['X2', 'X0', 'X0', 'OR', 'X0'] fitness= 5
['OR', 'OR', 'AND', 'X1', 'X0'] fitness= 5
['X0', 'X0', 'X1', 'X2', 'X0'] fitness= 5
['X0', 'OR', 'X2', 'X0', 'OR'] fitness= 3
['AND', 'X2', 'NOT', 'OR', 'X0'] fitness= 5
['OR', 'AND', 'X2', 'X1', 'AND'] fitness= 7
['X0', 'AND', 'AND', 'X0', 'NOT'] fitness= 3
['X2', 'X2', 'X2', 'X0', 'X0'] fitness= 5
```

After 50 generations, a typical population looks like this:

```
Final population:
['X2', 'OR', 'X1', 'X0', 'AND'] fitness= 7
['X2', 'X2', 'NOT', 'X2', 'AND'] fitness= 7
['X1', 'X2', 'AND', 'X0', 'AND'] fitness= 8
['X2', 'OR', 'X1', 'X0', 'AND'] fitness= 7
['X2', 'OR', 'X1', 'X0', 'AND'] fitness= 7
['X2', 'OR', 'X1', 'X0', 'AND'] fitness= 7
['X2', 'OR', 'X0', 'OR', 'AND'] fitness= 3
['X2', 'OR', 'NOT', 'X2', 'AND'] fitness= 7
['X2', 'OR', 'AND', 'X0', 'AND'] fitness= 7
['X2', 'OR', 'AND', 'X0', 'AND'] fitness= 7
['X2', 'X0', 'X0', 'OR', 'AND'] fitness= 7
['X2', 'OR', 'NOT', 'X2', 'AND'] fitness= 7
```

With a computational effort of $50 \times 12 = 600$ it has been possible to find an optimal program having a fitness of eight, that is the program `P=[X1 X2 AND X0 AND]`. This is faster than exhaustive search which, on average, would have taken $7{,}776/2$ evaluations. We must point out, however, that with the chosen parameters, linear GP doesn't find an optimal program each time it is run.

The following observation is interesting: the function (x_0 and x_1) or x_2 is, strangely enough, more difficult to find with the GP search. However, any program that terminates with x_2 has a fitness of 7, a very good fitness that makes evolution pressure weak.

9.4.2 Control Structures

An interesting particularity of the linear representation is the possibility of easily defining branches and loops in a genetic program. Here, we will limit ourselves to indicating a robust syntax for implementing such fundamental control structures.

Branching

The branching instructions `IF` and `ENDIF` can be introduced thus:

- `IF` pops the element a at the top of the stack and compares it to 0.
- If $a \geq 0$, the instructions that follow the `IF` are executed.
- If $a < 0$ no further instructions are executed until `ENDIF` is found. For this, it is enough to add to the execution engine an indicator with values 0 or 1, which say whether or not the current instruction must be executed. The instruction `ENDIF` resets the indicator to 1. If an `ENDIF` is not found, all the instructions following the `IF` will be ignored if $a < 0$, but the program will still be executable.
- `IF` instructions can be nested and will work correctly provided that the pairs `IF`–`ENDIF` are balanced. If this is not the case, perhaps after a crossover operation, the program will still be executable.

Loop

To implement the loop control structure it is necessary to introduce the instructions `LOOP` and `END-LOOP`. We must also add a control stack, besides the data stack. Each element of the control stack contains two values: the position in the program of the most recent `LOOP` instruction, and the number of remaining iterations. This works in the following way:

- When a `LOOP` instruction is found at position i in the program, the top element a of the data stack is used to specify the number of iterations. The tuple (a, i) is pushed onto the control stack. If the number of iterations is undefined or negative `LOOP` does nothing.
- When an `END-LOOP` is found the number of iterations a is decremented on the control stack and the control flow returns to position i except when $a = 0$, in which case the tuple (a, i) is suppressed in the control stack.
- Loops can be nested by pushing pairs (a, i) onto the control stack. The `END-LOOP` always acts on the pair (a, i) currently on the top of the stack.
- This program construction is robust even when the pairs `LOOP-END-LOOP` are not balanced.

9.5 Structured Populations

Up to now it has been implicitly assumed that the populations used in evolutionary algorithms do not possess any particular structure. In other words, any individual can interact with any other individual in the population. Such populations are called *panmictic* or *well mixed* and they are the norm both in biology and in EA. However, observation of animal and vegetal species in Nature shows that often population structures reflect the geographical nature of the world and cannot always considered to be well mixed. Indeed, Darwin himself based some of his fundamental observations on the fact that certain species had some different traits on different islands of the Galápagos archipelago and attributed this to the isolation caused by distance. Many biological phenomena can be better explained when one assumes that there is some structure and locality in the populations such as migration, diffusion, territoriality, and so on. This was thus the source of inspiration but we would like to point out that EAs are artificial systems and therefore do not need to obey the constraints by which natural systems are limited. Any suitable structure can be used and, in the end, if assuming a structure makes an EA more efficient or more capable of dealing with certain problems, there is no need for any other justification, although there is no doubt that analogous biological phenomena have played an important role in inspiring these changes.

There are many ways of attributing a structure to an EA population, but simple solutions are often the best ones. Thus, two main types of structure have been used in the literature: *multipopulations*, also called *islands*, and *cellular populations*. To have a uniform notation for these populations we shall represent them as simple graphs $G(V, E)$, where V is the set of vertices and E is the set of edges, or arcs, if the graph is an oriented one. In multipopulations, each vertex is a whole subpopulation, while in cellular populations a vertex is a single individual. Let's start by describing multipopulations first. They are constituted by a number of populations that may communicate with each other, as schematically depicted in Figure 9.8.

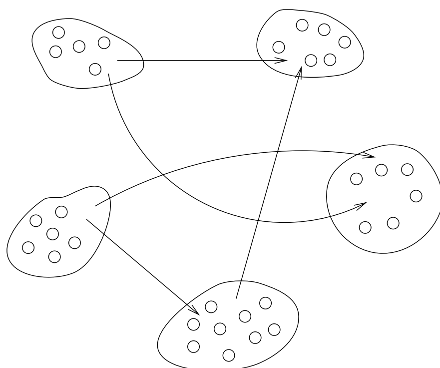


Fig. 9.8. An example structure of communicating populations

The system is clearly hierarchical since within a subpopulation the structure is well mixed, and this can be represented as a complete graph, since each individual may interact with any other member of the population.

In contrast, cellular populations enjoy a high degree of locality. As said above, each vertex in the corresponding graph represents a single individual, and a given individual can only interact with the individuals that are directly connected to it or, in other terms, only with the individuals belonging to a small neighborhood around the focal one. Cellular graphs can have any structure but most of the time they are regular, such as meshes or rings, as shown in Figure 9.9. Actually, grids are usually taken with cyclic boundary conditions, which means that vertices at the border are connected to the opposite side vertices (not shown in the figure). This transforms the grid into a torus, which is a regular graph.

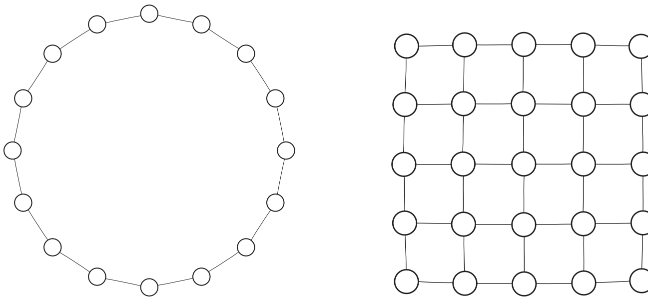


Fig. 9.9. Two possible structures for cellular populations

Now, if these structured populations didn't have any effect on the progression of an evolutionary algorithm, they would only be a scientific curiosity. But this is not the case, as we shall see in the following sections. Indeed, these new population structures cause an EA to change its behavior with respect to well-mixed populations in ways that can be exploited to improve its performance. We will start by describing the multipopulation model in more detail below.

Multipopulations.

These structures are rather close to the standard single panmictic populations. In fact, within each well-mixed subpopulation evolution takes place as usual. The only difference is that now islands (subpopulations) may exchange individuals with each other from time to time. These exchanges are sometimes called "migrations," by analogy with biological populations. With respect to the fully centralized standard evolutionary algorithm, there are several new parameters to be set in a suitable way:

- the number of subpopulations and their size;
- the communication topology between the subpopulations;
- the number and type of migrant individuals;

- the frequency of migration.

A detailed discussion of the above parameters would lead us out of the scope of the book. Their values are difficult to justify theoretically but many empirical studies have provided "ranges" that seem to work well in many cases. The number of subpopulations depends on the problem difficulty. The idea is to determine an appropriate single population size and then to distribute those many individuals among a sufficient number of communicating populations. The main danger is to use too many small subpopulations since this would negatively impact the search.

With respect to the communication pattern between the populations several solutions are available. The interconnection can range from the complete graph, in which each island communicates with all the other islands, to the very simple ring topology in which subpopulations are connected in a ring shape and communicate only with adjacent subpopulations. After many studies and experiments, it appears that the precise choice of the communication topology only has a marginal importance for the performance. As a consequence, a ring or a random graph structure is often used because of their simplicity.

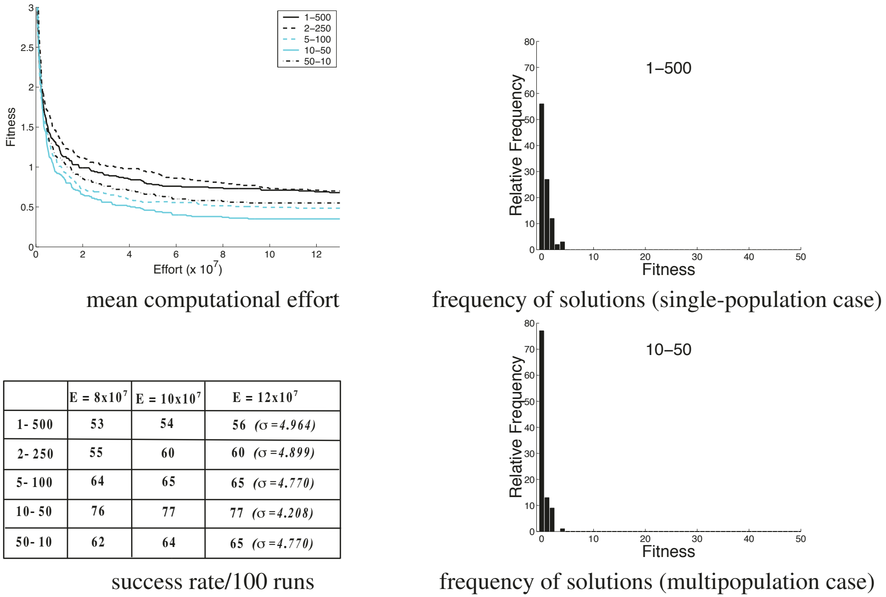
The number and quality of migrant individuals, and the replacement method used in the destination subpopulation are important parameters. In practice, it has been found that the "right" number of individuals to send out is between 5% and 10% of the subpopulation size. How to choose those individuals? The usual solution is to send the best individuals of the origin population and, at the destination island, to replace an equal number of the worst individuals with the new ones. Sometimes another policy is used that consists of replacing the same number of randomly chosen local individuals. In any case, it is customary to keep the subpopulations' sizes equal and constant.

For the last parameter, the migration frequency, experience indicates that about ten generations is a good choice. In other words, populations evolve independently running a standard EA during about ten iterations, after which the fixed number of migrants is sent and received between the populations in a synchronized manner, according to a pre-established communication pattern. The exchange can also be asynchronous but this requires more care in setting up the distributed programming environment. In some variants, the EA parameters for each subpopulation may be different, or the subpopulations may be arranged in a hierarchical fashion, but these more complicated settings are seldom used.

Now that we have described the main elements that influence the design of a distributed EA, we might ask whether the added complexity is worth the effort. In general, we can say that the practical experience accumulated clearly indicates that the answer is positive. To illustrate this, we compare the results obtained on a standard difficult benchmark problem in GP, the even-parity problem [2], using a single population and several multipopulations with the same total size of individuals.

The graphics in Figure 9.10 show some performance measures on the even-parity problem with a single population of 500 individuals and with multipopulations in

[2] The Boolean even parity problem takes a bit string as input and returns True if there is an even number of 1s in the string and False otherwise.



	E = 8×10^7	E = 10×10^7	E = 12×10^7
1-500	53	54	56 ($\sigma = 4.964$)
2-250	55	60	60 ($\sigma = 4.899$)
5-100	64	65	65 ($\sigma = 4.770$)
10-50	76	77	77 ($\sigma = 4.208$)
50-10	62	64	65 ($\sigma = 4.770$)

success rate/100 runs

Fig. 9.10. Computational effort and quality of solutions found for the even-parity problem with genetic programming in the single-population case and with a multipopulation structure

which the same number of individuals is distributed into 2, 5, 10, and 50 subpopulations connected in a ring structure. We will come back to performance evaluation in much more detail in Chapter 12. For the time being, let us note that the upper left image reports the mean fitness over 100 runs of the algorithm for this problem as a function of the computational effort expended. The optimal fitness is zero. The table at the left in the lower part of the figure gives the success rate, i.e., the number of times the algorithm found the global optimum over 100 runs. Finally, the histograms on the right of the panel show the frequency of solutions having a given fitness for the single population (upper image), and for a multipopulation system with 10 subpopulations of 50 individuals each (lower image), when using the maximum allowed computational effort, which is 12×10^7.

The mean-fitness curves clearly show that the multipopulation systems perform better than the single population, except when there are 50 populations of size 10, in which case the performance is worse or similar depending on the effort. In this case the size of the subpopulations is too small to allow for the rapid evolution of good individuals. This is confirmed by the success rates in the table and by comparing the single population with a multipopulation system with 10 subpopulations (right images), which seems to be the best compromise. This example is not an unusual case: the research published to date confirms that multipopulations perform better than, or at least equally well as, single well-mixed populations. The drawback of multipopulation EAs is that they are slightly more difficult to implement than centralized

ones. On the other hand, they also have the advantage that they can be executed on distributed networked systems in parallel, as the overhead caused by the communication phases is not too heavy given the low migration frequencies normally used.

What are the reasons behind the better performance of multipopulation systems? It is difficult to give a clear-cut answer as the systems are very hard to analyze rigorously, but in qualitative terms it appears that the systematic migration of good individuals from other populations has a positive effect on the diversity of the target population. Indeed, we have seen that a characteristic of evolution is the loss of population diversity due to selection. The multipopulation system helps fight this tendency by periodically reinjecting new individuals. This phenomenon can be appreciated in Figure 9.11, jagged curve, where migrants are sent and received every ten generations.

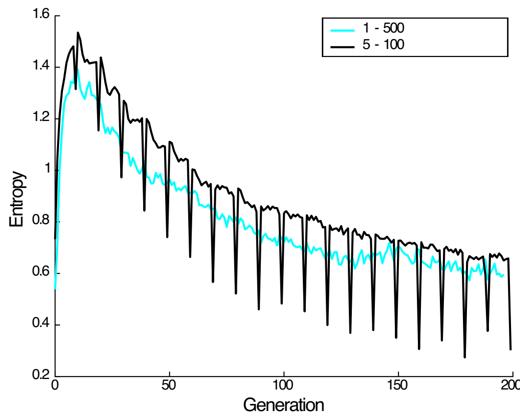


Fig. 9.11. Population diversity measured as fitness entropy for the even-parity problem. Turquoise curve: a well-mixed population of size 500. Black curve: five populations of 100 individuals each

Cellular evolutionary algorithms.

As shown in Figure 9.9, cellular populations bring about a major change in the population structure. While multipopulation evolutionary algorithms are, after all, similar to standard EAs, cellular evolutionary algorithms show a rather different evolution when compared to well-mixed EAs. The reasons are to be found in the way genetic operators work which, is now *local* instead of *global*.

The situation is easy to understand from Figure 9.12 where an individual i and its neighborhood $V(i)$ have been highlighted. The neighborhood shown is not the only one possible but the idea is that $V(i)$ has a small cardinality with respect to the population size: $|V(i)| \ll n$. Within these populations, selection, crossover, and mutation are restricted to take place only in the neighborhood of a given focal individual. To

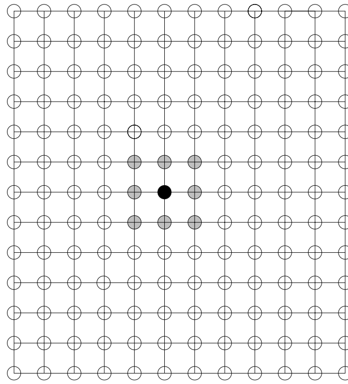


Fig. 9.12. A population structured as a mesh. An individual and its immediate neighborhood are highlighted

be more precise, let us describe tournament selection, the selection method of choice in cellular populations. For example, the central individual might play a tournament with all the neighbors and be replaced by the winner, if the latter is not itself, or we might randomly draw a neighbor and let it play a binary tournament with the central individual. Crossover might then be performed between the central individual and a randomly chosen neighbor, followed by mutation of the offspring with a certain probability. Clearly, what we describe for a single individual will have to take place for all the individuals in the population. This, in turn, can be done synchronously or asynchronously. In synchronous updating an intermediate grid is kept in which the new individuals are stored at their positions as soon as they are generated. Evolution takes place sequentially using the current-generation individuals and their fitness values. When all the individuals have been updated, the intermediate grid becomes the new one for the next generation. In asynchronous updating, an updating order has to be decided after which individuals are updated in that order and are integrated into the population as soon as they are produced. In the following we will assume synchronous updating, which is more common and easier to understand.

To better understand the nature of evolution in cellular populations, we shall briefly discuss the takeover times generated by binary tournament selection in cellular populations as compared to panmictic ones. In the case of a cellular population with a ring structure and with a neighborhood formed by the central individual and its immediate right and left neighbors (see Figure 9.13), the maximum propagation speed of the best individual depends linearly on time. This is easy to understand considering that at time step 1 only the immediate two neighbors can be replaced in the best case; at time step 2 the two new neighbors at the extremity of the propagating front will be replaced, and so forth. In this case, which is an upper bound on the growth speed, it is easy to write the recurrence (9.19) for the number $N(t)$ of individuals conquered by the best one at time step t:

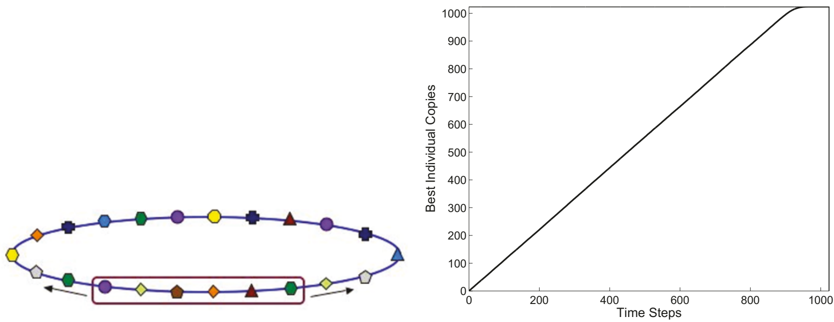


Fig. 9.13. Propagation speed of the best individual on a ring of size $1{,}024$ using binary tournament selection. The neighborhood includes an individual and its right and left neighbors at distance one

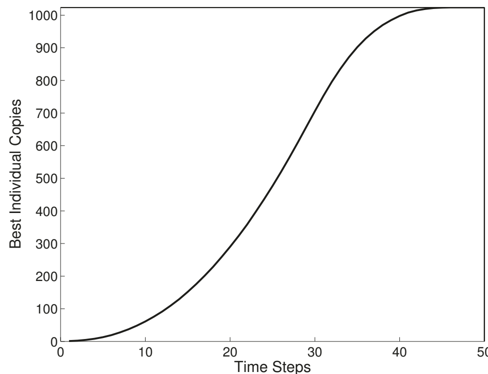


Fig. 9.14. Propagation speed of the best individual on a torus. Selection is by binary tournament and the population size is $1{,}024$

$$\begin{cases} N(0) = 1, \\ N(t) = N(t-1) + 2r, \end{cases} \tag{9.19}$$

where $r = 1$ is the radius of the neighborhood. This recurrence can be given in the closed form $N(t) = 1 + 2rt$ which shows the linear nature of the growth.
The two-dimensional case is treated in a similar way and gives rise to the curve of Figure 9.14 for binary tournament selection. In spite of its "S" shape, this curve is not a logistic. Indeed, the growth speed is bounded by a quadratic expression. Before the inflection point, the curve is convex, while after that point the population starts to saturate and the trend is reversed, following a concave curve. This behavior can be understood from Figure 9.15 in which neighbors are replaced deterministically, i.e., with probability one, by the best individual in each time step. We see in the figure

that the perimeter of the propagating region grows linearly, which means that the enclosed area, which is proportional to the population size, grows at quadratic speed.

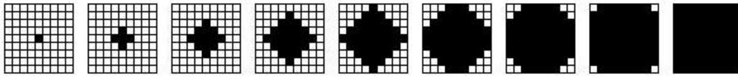


Fig. 9.15. Illustration of the maximum growth rate of the best individual in a grid-structured population

We thus come to the conclusion that the selection pressure is lower on rings than on grids, and both are lower with respect to well-mixed populations. Numerical simulations do indeed confirm these results, as can be seen in Figure 9.16. This result offers a rather flexible method for regulating the selection pressure by varying the ratio of the lattice diameter to the neighborhood diameter and the adjustment can also be made dynamically adaptive and performed on the fly.

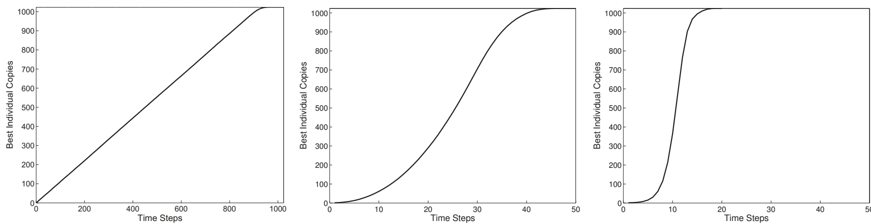


Fig. 9.16. Numerical simulation of the growth speed of the best individual in a ring (left), a torus (middle), and a panmictic population, all of the same size

In practice, cellular evolutionary algorithms have been mainly used with populations structured as meshes, because rings, although easy to deal with, have a very slow evolution speed. It is also worth noting that cellular structures are very well suited for parallel computation, either by decomposing the grid and having different machines in a cluster compute each piece independently, or by using modern graphical processing units (GPUs) to compute each new cell in parallel in data-parallel style. The first solution only requires the boundary of the subgrid to be communicated to other processors at the end of each generation step, while GPUs allow an even simpler implementation and have an excellent cost/performance ratio. To conclude this part, we remind the reader that much more detailed descriptions of multipopulation and cellular evolutionary algorithms can be found in [79, 4].

9.6 Representation and Specialized Operators

The efficiency of an EA depends on several factors, but one of the most important is the individual representation adopted. We have already seen in the last chapter that a good encoding for the real numbers can make a big difference in the efficiency of the optimization of mathematical functions. We have also hinted at the fact that a critical choice in GP is the determination of the primitives that are going to be used. And the idea can be extended, in a different way, to other optimization domains, especially combinatorial optimization. Thus, it is always worth trying to use an adapted data structure to represent the individuals. For example, as far as possible, the representation and the genetic operators should be matched in such a way that they can only produce admissible solutions. This is sometimes a difficult goal to achieve, especially in combinatorial optimization where several choices are equally good *a priori*. In fact, the problem that arises if the representation and the operators are not matched correctly is that many solutions produced during the evolution turn out to be non-admissible. There exist some remedies to this problem:

- the fitness of non-admissible solutions can be penalized in order to drastically decrease their probability of being selected;
- the non-admissible solutions can be "repaired";
- an encoding of solutions and genetic operators that can only produce admissible solutions can be used.

Penalizing individuals that do not respect the problem constraints is a simple, general, and useful technique but it can be costly in terms of computing, and making viable non-viable solutions is even more expensive. The third solution is more difficult to implement but it is surely the best. To illustrate some of the issues that may arise when the representation is not adequate we shall make use of the traveling salesman problem (*TSP*) once more. In this problem, computing the fitness of a tour is trivial but producing viable tours through genetic operators is more difficult. Indeed, what follows applies to the *TSP* but similar considerations could be made on many other combinatorial optimization problems.

A tour on the set of cities $\{1, 2, \ldots, N\}$ can be represented as a permutation of these numbers. Let us take $N = 8$ for the sake of simplicity and consider the following two tours:

$$1 - 2 - 4 - 3 - 8 - 5 - 6 - 7$$

$$1 - 3 - 4 - 8 - 2 - 5 - 7 - 6$$

If we try to cross them over by using one-point crossover, the following situation might present itself:

$$1 - 2 - 4 \mid 3 - 8 - 5 - 6 - 7$$

$$1-3-4 \mid 8-2-5-7-6$$

in which case we would get the clearly non-admissible solution

$$1-2-4-8-2-5-7-6$$

Therefore, standard crossover cannot work with the permutation representation since offspring will be non-admissible in most cases. However, one can search for other non-standard genetic operators that do not have this drawback. For instance, an operator that does work with the permutation representation is the random displacement of a city in the list:

$$1-2-4-3-8-5-6-7 \rightarrow 1-4-3-8-2-5-6-7$$

Another popular operator that always generates a valid tour selects a subtour and reverses the sequence of cities in the subtour:

$$1-2-3-|4-5-6|-7-8 \rightarrow 1-2-3-|6-5-4|-7-8$$

If we really want to use crossover on two tours T_1 and T_2 represented by permutations of cities, the following method produces a viable offspring:

$$T_1 = 1-4-|3-5-2|-6-7-8$$

$$T_2 = 1-3-2-6-4-5-7-8$$

The section between the random cut points in parent T_1 is transferred to the offspring T_3; then the cities of parent T_2 that are not already present in the offspring are inserted in succession:

$$T_3 = 3-5-2-1-6-4-7-8$$

One can also make use of representations other than permutations. For example, the *adjacency representation* encodes a tour as a list of N cities. In the list, city k is at position i if there is a connection between city k and city i. For example, the following individual

$$(2\ 4\ 8\ 3\ 6\ 1\ 5\ 7)$$

corresponds to the tour

$$1-2-4-3-8-7-5-6$$

Standard crossover does not work with this representation but there are other ways of recombining tours. For example, to obtain a new tour starting from two parent tours, one can take a random edge from the first parent, then another from

the second parent, and so on, alternating parents. If an edge introduces a cycle in a partial tour, it is replaced by a random edge from the remaining edges that does not introduce a cycle.

The examples presented above were selected to illustrate the problems that may arise from a bad choice of a representation and some ways of fixing them. We hope that the reader has been convinced by these few examples that the issue is an important one in EAs and that it must be taken into account when implementing an algorithm. In the end, however, even if the representation and the associated variation operators are correct in the sense of always producing admissible solutions, it does not mean that the EA will solve the problem efficiently. In fact EAs are probably not the first choice for solving *TSP* problems. Other metaheuristics and approaches based on more classical techniques have proved to be superior in general.

9.7 Hybrid Evolutionary Algorithms

Evolutionary algorithms are very general and flexible and can be applied to many different search and optimization problems. However, this generality is paid for in terms of efficiency and precision. It appears that EAs are very good at finding good regions of the search space but they have some trouble rapidly converging toward the best solutions. This behavior is partly due to the stochastic nature of EAs and, intuitively, calls for adding a local search phase around good solutions in order not to lose them by jumping far away with crossover or mutation. This algorithmic schema has indeed been frequently used, configuring what can be called a *hybrid evolutionary algorithm*. To give an example, in the *MaxOne* problem of the previous chapter, when we approach a good solution a random bit mutation makes it difficult to progress, i.e., mutating a 0 into a 1, given that the opposite mutation has the same probability. Clearly, hill climbing in which a mutation is accepted only if it increases the fitness would be more efficient at this point. The example is not really well chosen because, from the start, hill climbing is a better search method in this problem, but the idea is to illustrate how local search may help an EA to find good solutions. The general schema of a hybrid evolutionary algorithm can be described by the following pseudo-code:

generation = 0
Initialize population
while *not termination condition* **do**
  generation = *generation* + 1
  Compute the fitness of each individual
  Select individuals
  Apply genetic operators to selected individuals
  Perform local search
end while

After the local search phase, the new solutions may replace the solutions from which local search started or not. In the first case, the algorithm is called "Lamar-

ckian," meaning that the solutions obtained with local search are incorporated in the population, symbolically keeping the "new genes" that have been obtained by learning. Of course, modern molecular biology denies the possibility of genetically transmitting acquired traits, but EAs need not be faithful to biology and can include this kind of evolution. In the second case, we take into account the fitness of the new solutions in terms of optimization obtained by local search, for example by keeping track of a few current best solutions, but they are not incorporated into the population. Other forms of hybridization are possible too. For example, if we could somehow obtain solutions of rather good quality by another quick method, we could start from those in the EA, instead of generating them randomly; or we could seed the initial population with a fraction of those solutions. This might speed up convergence of the EA but there is also the danger of premature convergence toward a local optimum due to early exploitation.

10

Phase Transitions in Combinatorial Optimization Problems

10.1 Introduction

The goal of this chapter, which is based on reference [63], is to better characterize the nature of the search space of particular problems where the number of optimal solutions, and the difficulty of finding them, varies as a function of a parameter that can be modified at will. According to the value of the parameter, the system goes from a situation in which there are many solutions to the problem to a situation in which, suddenly, there are no solutions at all. This type of behavior is typical of *phase transitions* in physics and the term has been adopted in the computational field by analogy with the physical world. In fact, there are many natural phenomena in which some quantity varies wildly, either continuously or discontinuously, at critical points as a function of an external parameter. For instance, the volume of a macroscopic sample of water varies discontinuously when the temperature goes from positive to negative, during the phase transition from the liquid phase to the solid phase, that is, when water freezes. We will also see that the phase transition concept applies as well to the behavior of a given metaheuristic itself. For small size N problems the solution is quickly found in time $\mathcal{O}(N)$, but if the size increases the complexity may quickly become exponential.

This chapter has a different flavor from the rest of the book as it provides a view of complexity that comes from a statistical mechanics approach applied to computational problems. Although it may turn out to be slightly more difficult to read for computer scientists, and although it applies mostly to randomly generated problem instances, we think that it is worth the effort since it provides a statistical view of problem difficulty that, in some sense, is complementary to the classical worst-case complexity analysis summarized in Chapter 1.

The problem we are going to consider in this chapter to illustrate this kind of behavior is the *satisfiability* problem (SAT) in Boolean logic. In this problem, the objective is to find the value of the variables in a Boolean formula that satisfy a number of given logical conditions (constraints), that is, that make the formula evaluate to True. The following expression is an example of a formula in *conjunctive normal form*, that is, a collection of clauses related by the logical $\wedge$ (*AND*) operation, each

© Springer Nature Switzerland AG 2018
B. Chopard, M. Tomassini, *An Introduction to Metaheuristics for Optimization*,
Natural Computing Series, https://doi.org/10.1007/978-3-319-93073-2_10

of which is the disjunction $\vee$ (*OR*) of several literals, where each literal is either a Boolean variable v or the negation of a variable $\bar{v}$:

$$(x_1 \vee x_3) \wedge (x_3 \vee \bar{x}_2) \wedge (x_2 \vee \bar{x}_1).$$

For instance, the assignments $x_1, x_2, x_3 = (0, 1, 1)$, and $x_1, x_2, x_3 = (0, 0, 1)$ both make the formula to evaluate to 1 (True) and thus the formula is satisfiable. However, it may happen that such a problem has no solution if there are too many constraints, in which case the formula is said to be *unsatisfiable*. In this case one can ask how many constraints are satisfied, transforming the decision problem into an optimization one which is called MAXSAT. SAT is the prototypical NP-complete problem having exponential complexity (see chapter 1) except when there are only two variables per clause (2-SAT). The interested reader is referred to [76, 59, 17, 43, 2, 23] for further discussions. In what follows we are going to study in detail a particular case of the satisfiability class of problems to illustrate the kind of phenomena that may appear.

10.2 The k-XORSAT Problem

The *XORSAT* problem is a particular satisfiability problem in which the Boolean variables are combined only using the logical operator *XOR* or, in an equivalent fashion, by addition modulo 2. In this arithmetic modulo 2, the constraints, also called clauses, are built with the *XOR* operation and can be expressed by linear equations. For instance, the problem below is a *XORSAT* problem with three constraints:

$$\begin{cases} x_1 + x_2 + x_3 = 1 \\ \quad x_2 + x_4 = 0 \\ \quad x_1 + x_4 = 1 \end{cases} \tag{10.1}$$

where the $+$ sign stands for the addition modulo 2, i.e., the *XOR* logical operation. The variables x_i are Boolean, meaning that $x_i \in \{0, 1\}$. Such a system is said to be *satisfiable* if it has one or more solutions. In the present case, one finds by inspection that the only two solutions are:

$$(x_1, x_2, x_3, x_4) = \begin{cases} (1, 0, 0, 0) \\ (0, 1, 0, 1) \end{cases} \tag{10.2}$$

It is easy to contrive a problem that has no solution. For example

$$\begin{cases} x_1 + x_2 + x_3 = 1 \\ x_1 + x_2 + x_3 = 0 \end{cases} \tag{10.3}$$

cannot be satisfied since the two constraints contradict each other. In this case the problem is unsatisfiable and the maximum number of satisfied constraints is one.

In general, a *XORSAT* problem can be solved by using Gaussian elimination (in modulo 2 arithmetic) in time $\mathcal{O}(N^3)$, where N is the number of variables and also

the number of equations in the system. In contrast with *SAT*, *XORSAT* problems are thus not hard to solve.

In what follows we shall limit ourselves to the particular case of *k-XORSAT* problems. Let us denote by N the number of variables $x_i \in \{0, 1\}$, $i = 1, \ldots, N$, and let M be the number of equations (or constraints) to be satisfied. If the M equations all contain exactly k variables among the N possible ones, we say that the problem is of type *k-XORSAT*.

10.3 Statistical Model of *k*-XORSAT Problems

Our goal is the analysis of the solution space of the *k-XORSAT* problems. Our approach is statistical in nature and considers a large population of problems, all having the same N, M, and k. For each problem in the set, we shall numerically compute whether it is satisfiable or not using Gaussian elimination. In the limit of a very large number of problems we will thus be able to estimate the probability $P_{SAT}(N, M, k)$ that a randomly generated instance is satisfiable. This probability will be evaluated according to the standard frequency interpretation, that is, it will be the fraction of problems that have at least one solution divided by the total number of generated problems.

We introduce a new variable α defined as the ratio of the number of clauses to the number of variables:

$$\alpha = \frac{M}{N} \tag{10.4}$$

and we consider the probability of satisfaction P_{SAT} as a function of N and α, $P_{SAT}(N, \alpha)$ for fixed k. It is intuitive to expect that this probability should be a decreasing function of α since increasing the number of constraints should make it more difficult to find a solution. In fact the behavior, shown in Figure 10.1, is more surprising than that: the drop becomes sharper as the number of variables N increases and, in the limit for $N \to \infty$ and for $k \geq 2$, a phase transition occurs at a critical value $\alpha_c(k) \approx 0.9179$ of the control parameter α. For $\alpha < \alpha_c$ all the systems are satisfiable, while they all become unsatisfiable past this point.

To generate a random sample of *k-XORSAT* problems for given N, α, and k we can proceed in two different ways.

In the first method we choose uniformly at random k distinct indices among the N and a Boolean ν, thus generating the equation

$$x_{i_1} + x_{i_2} + \ldots + x_{i_k} = \nu \tag{10.5}$$

where $i_1, i_2, \ldots, i_k$ are the k randomly chosen indices. The procedure is repeated M times to obtain an instance of the problem with M constraints. Clearly, there is the risk that several equations will be the same if the same sequence of indices is drawn more than once, or even contradictory if, in addition, the second member ν is different. This is the method that has been used to produce the curves in Figure 10.1.

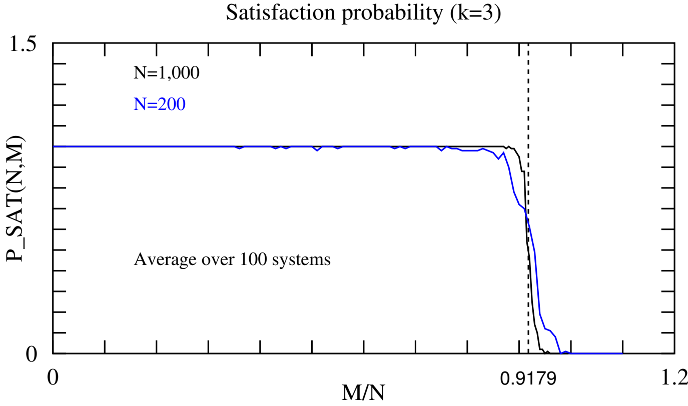


Fig. 10.1. Typical phase transition behavior for the satisfaction probability as a function of the clauses to variables ratio $\alpha = N/M$. As the number N of variables x_i increases and α reaches the critical value 0.9179, the curve P_{SAT} goes in an increasingly steeper way from 1, where formulas are typically easily satisfiable, to 0, where there are no solutions. The curves have been obtained through Gaussian elimination for $k = 3$, $N = 1{,}000$ and $N = 200$ and they represent averages over 100 randomly generated problems

Another approach for generating the statistical sample is to consider all the possible equations with k variables among the N. Moreover, since there are two ways of choosing the right-hand side of an equation (0 or 1), the number of possible equations is given by

$$H = 2\binom{N}{k} \tag{10.6}$$

Each one of these H equations is then selected with probability $p = M/H$ which in the end will give $pH = M$ equations in the system on average.

For both sampling methods proposed, the process is repeated a sufficient number of times $\mathcal{N} \gg 1$ in order to correctly sample the space of possible problems. The probability P_{SAT} is defined according to these ways of generating the *k-XORSAT* problems.

10.4 Gaussian Elimination

As a reminder, and in order to introduce a number of notions that will be used in the sequel, this section illustrates the well known Gaussian elimination procedure for solving systems of linear equations. Given the form of a *XORSAT* problem, Gaussian elimination is well suited for the task, as we saw above.

As an example, consider the 3-*XORSAT* system below, which has been randomly generated and contains $N = 10$ variables and $M = 8$ equations.

$$\begin{pmatrix} 0\,0\,0\,1\,1\,0\,0\,1\,0\,0 \\ 0\,0\,0\,0\,1\,1\,0\,0\,1\,0 \\ 0\,0\,0\,0\,0\,1\,1\,0\,0\,1 \\ 1\,1\,0\,0\,0\,0\,0\,1\,0\,0 \\ 1\,0\,1\,0\,1\,0\,0\,0\,0\,0 \\ 0\,0\,0\,1\,0\,0\,1\,1\,0\,0 \\ 0\,0\,1\,0\,0\,0\,0\,1\,1\,0 \\ 1\,1\,0\,0\,0\,0\,0\,0\,0\,1 \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \\ x_8 \\ x_9 \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 0 \\ 1 \\ 1 \end{pmatrix} \tag{10.7}$$

Gaussian elimination consists of performing a series of operations on the matrix of coefficients in such a way that the initial system of equations is transformed into an equivalent one. The latter is obtained by replacing certain equations with a linear combination of the others. We proceed in the following way: we consider the column j_1 that contains the first 1 on the first matrix line. We add this line to all the other lines that have a 1 in the same column position j_1 and we do the same on the right-hand side of the equation. Since we work here in modulo 2 arithmetic, the result of the operation is that column j_1 contains only one 1, on line 1. We repeat the same procedure for each of the remaining lines and, as a result, a different column with only one 1 is created for each line. Iterating this procedure on the above system, one gets the following transformed system:

$$\begin{pmatrix} 0\,0\,0\,1\,0\,0\,1\,0\,0\,1 \\ 0\,0\,0\,0\,1\,0\,1\,0\,0\,0 \\ 0\,0\,0\,0\,0\,1\,1\,0\,0\,1 \\ 1\,0\,0\,0\,0\,0\,1\,0\,0\,0 \\ 0\,1\,0\,0\,0\,0\,1\,0\,0\,1 \\ 0\,0\,0\,0\,0\,0\,0\,0\,1\,1 \\ 0\,0\,1\,0\,0\,0\,0\,0\,0\,0 \\ 0\,0\,0\,0\,0\,0\,0\,1\,0\,1 \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \\ x_8 \\ x_9 \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \\ 1 \\ 0 \\ 1 \\ 1 \\ 0 \\ 0 \end{pmatrix} \tag{10.8}$$

This system can also be written as

$$\begin{aligned} x_3 &= x_6 + x_9 \\ x_4 &= x_6 + 1 \\ x_5 &= x_6 + x_9 + 1 \\ x_0 &= x_6 \\ x_1 &= x_6 + x_9 + 1 \\ x_8 &= x_9 + 1 \\ x_2 &= 0 \\ x_7 &= x_9 \end{aligned} \tag{10.9}$$

where we have kept the ordering given by the original equations instead of numbering them according to the variable number. From the above formulation the solution of the system is apparent. There are eight *constrained variables*, that is, x_3, x_4, x_5, x_0, x_1, x_8, x_2, and x_7, whose values are determined by the two *free variables* x_6 and x_9. These free variables can take any value in $\{0, 1\}$, giving here four possible solutions. The system is thus clearly satisfiable. Interpreting variables as bit strings, the four solutions are generated by $x_6x_9 \in \{00, 01, 10, 11\}$. For $x_6x_9 = 10$, the solution is $x_3x_4x_5x_0x_1x_8x_2x_7 = 10010100$. The set of solutions can be graphically represented by placing the numbers x_6x_9 expressed in decimal on the x axis and the corresponding number $x_3x_4x_5x_0x_1x_8x_2x_7$ on the y axis, also in decimal notation. In the present case we obtain

$$\begin{aligned}
x_6x_9 &= 0 \rightarrow x_3x_4x_5x_0x_1x_8x_2x_7 = 108 \\
x_6x_9 &= 1 \rightarrow x_3x_4x_5x_0x_1x_8x_2x_7 = 193 \\
x_6x_9 &= 2 \rightarrow x_3x_4x_5x_0x_1x_8x_2x_7 = 148 \\
x_6x_9 &= 3 \ \rightarrow x_3x_4x_5x_0x_1x_8x_2x_7 = 57
\end{aligned} \tag{10.10}$$

This representation will be used in Figure 10.7.

In Gaussian elimination one does not always obtain a matrix in which all lines and columns are non-zero. A column j with all zeroes indicates that the x_j variable does not appear in the system. A line i with all zeroes indicates that equation i is a linear combination of other lines, if the right-hand side is also zero. We then get a "$0 = 0$" equation, which gives no information. On the other hand, if the right-hand side is different from zero we get an equation of the type "$0 = 1$," which indicates a non-satisfiable system. The number of "$0 = 1$" equations obtained after Gaussian elimination gives the minimal "energy" of the system.

10.5 The Solution Space

In this section we look at the *k-XORSAT* problem's solution space for $k = 1, 2$, and 3. For these k values it is possible to obtain analytical results that show the presence of the phase transition mentioned at the beginning of the chapter [63]. Here we shall only outline the main results, avoiding complicated mathematical derivations; the reader wishing to know the details is referred to [63].

10.5.1 The 1-XORSAT Case

This simple case is not interesting in itself as a satisfiability problem but it allows us to illustrate the probability concepts introduced in this chapter in a straightforward way. Here is an example of a 1-$XORSAT$ problem

$$\begin{cases} x_{i_1} = \nu_1 \\ \dots \\ x_{i_M} = \nu_M \end{cases} \tag{10.11}$$

with $i_k \in \{1, 2, \dots, N\}$. A system like this one is satisfiable if no x_i appears more than once in the constraints list. If it appears more than once, then the right-hand sides must be equal, otherwise we face a contradiction.

The probability P that we never draw the same variable twice when we randomly draw M equations is equal to

$$P = 1 \times \left(1 - \frac{1}{N}\right) \times \left(1 - \frac{2}{N}\right) \times \dots \times \left(1 - \frac{M-1}{N}\right) = \Pi_{i=0}^{M-1}\left(1 - \frac{i}{N}\right) \tag{10.12}$$

Indeed, the first variable can be chosen in N different ways and so its probability is $N/N = 1$. For the second draw, since one variable has already been chosen, there are $N-1$ possible choices out of N, which gives a probability $(N-1)/N = 1-1/N$. For the third draw, since two variables have already been chosen, the probability is $(N-2)/N = 1 - 2/N$, and so on until all the M variables have been drawn. An alternative view is that for the first draw any choice is possible; for the second draw the probability of not choosing the variable drawn first is the total probability minus the probability corresponding to the draw of a single variable out of N, i.e., $1-1/N$, and so on. Interestingly, this problem is equivalent to the *birthday* problem, that is, the one that calls for finding the probability that, among M persons, no two persons have their birthday the same day, with $N = 365$ days.

The probability P_{SAT} that the system (10.11) has a solution is strictly greater than P since there are cases in which we draw the same variable and the same right-hand side ν twice. Therefore

$$P_{SAT} > P$$

P can be estimated by taking logarithms in equation 10.12

$$\log P = \sum_{i=0}^{M-1} \log\left(1 - \frac{i}{N}\right) \tag{10.13}$$

But $\log(1-x) = -x + x^2/2 + \dots$, which gives

$$\log P = -\sum_{i=0}^{M-1} \frac{i}{N} + \sum_{i=0}^{M-1} \frac{i^2}{2N^2} + \dots \tag{10.14}$$

Now, for N sufficiently large with respect to M, we have $M/N \ll 1$ and the dominant term in the sum will be

$$\log P = -\frac{M(M-1)}{2N} + \mathcal{O}\left(\frac{M^3}{N^2}\right) \tag{10.15}$$

because $\sum_{i=0}^{M-1} i = \frac{M(M-1)}{2}$ by the arithmetic summation formula. Therefore

$$P_{SAT} > P \approx \exp\left(-\frac{M(M-1)}{2N}\right) \approx \exp\left(-\frac{\alpha^2 N}{2}\right) \tag{10.16}$$

where $\alpha = M/N$. From the last equation we see that in order to have a constant satisfaction probability, M^2/N has to be constant. In other words, if we increase the number of variables by a factor of 100, then an increase by a factor of 10 in the number of equations will give rise to the same probability of satisfaction, i.e., to the same average problem difficulty. Figure 10.2 depicts the behavior of P as a function of α for $N = 10$ and $N = 50$. We see that P tends to zero more rapidly with increasing N but its value at $\alpha = 0$ is non-vanishing. There is a phase transition for a critical value $\alpha_c(N)$ that tends to zero when N tends to infinity. The phase transition becomes crisp only in this limit; for finite-size systems, the transition between the *satisfiable* and *unsatisfiable* phases is smoother, as hinted at in the figure. As a consequence, there is an interval of α values for which 1-$XORSAT$ problems have an intermediate probability of being satisfiable.

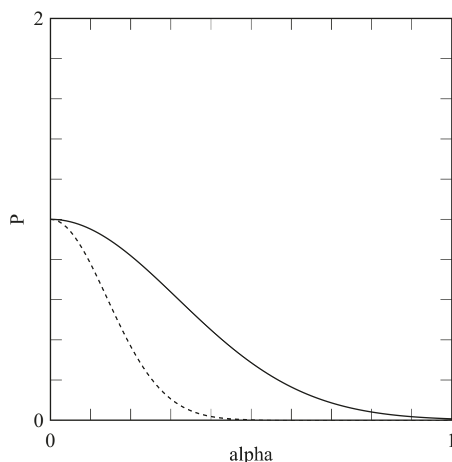


Fig. 10.2. Estimated satisfaction probability for 1-$XORSAT$ as a function of α from equation (10.16). The continuous line corresponds to $N = 10$, the dotted line is for $N = 50$

10.5.2 The 2-XORSAT Case

We now look at the 2-$XORSAT$ case. The M equations that make up the system are of the following form:

$$x_{i_1} + x_{i_2} = \nu \tag{10.17}$$

where $i_1, i_2 \in \{1, \ldots, N\}$ and $i_1 \neq i_2$.

We can represent such a system by a graph that has the N variables as vertices. If two vertices, i.e., two variables, appear in the same equation they are joined by an edge with a label given by the ν value. Thus, the graph corresponding to a 2-$XORSAT$ problem will have $M = \alpha N$ edges, that is as many edges as there are equations.

To illustrate, the system

$$\begin{cases} x_1 + x_3 = 0 \\ x_2 + x_3 = 1 \\ x_3 + x_4 = 0 \end{cases} \tag{10.18}$$

can be represented by the graph of Figure 10.3.

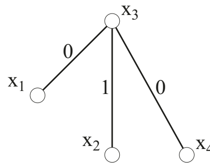


Fig. 10.3. Graph representation of the 2-$XORSAT$ system in equation (10.18)

If the graph thus obtained is a tree, as in Figure 10.3, or a forest, that is a set of disjoint trees, it is easy to see that the system is satisfiable. Indeed, it suffices to arbitrarily choose the value of any leaf of the tree, and to attribute the following variable values depending on the edge value: if the edge is labeled 0, the two incident vertices will have the same value; if its value is 1, the opposite value. This procedure is illustrated in Figure 10.4.

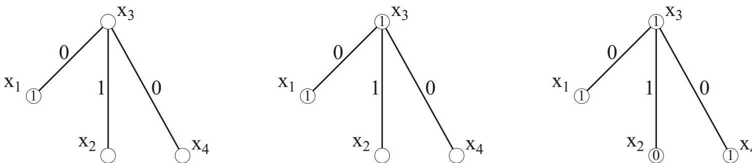


Fig. 10.4. Step-by-step construction of the solution by assigning an arbitrary value to a tree leaf

On the other hand, if the system of equations gives rise to a graph that contains cycles, satisfiability is not guaranteed. For an example, see Figure 10.5, which corresponds to the following system

$$\begin{cases} x_1 + x_2 = 1 \\ x_2 + x_3 = 1 \\ x_3 + x_4 = 0 \\ x_4 + x_1 = 0 \end{cases} \tag{10.19}$$

In this example there are two solutions. If one starts by attributing an arbitrary 0/1 value to a vertex belonging to the cycle and then one travels along the cycle, the previous value is reversed if the arc is labeled 1, while it is the same if the label is 0. Therefore, when the loop is closed, the solution is consistent if the number of edges labeled 1 traversed is even. This is the case in Figure 10.5 and the cyclic path gives

$$x_2 = \bar{x_1} \quad x_3 = \bar{x_2} = x_1 \quad x_4 = x_3 = x_1 \quad x_1 = x_4 = x_1 \tag{10.20}$$

which is consistent for any value of x_1.

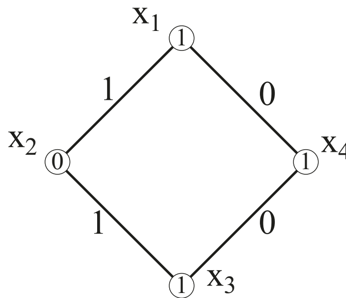


Fig. 10.5. A 2-$XORSAT$ example that gives rise to a cycle

Given that the number of 1 labels on a cycle is either even or odd, the probability that a cycle is satisfiable is $1/2$. Therefore, the question of the satisfiability of a 2-$XORSAT$ problem boils down to the estimation of the probability that a random graph has cycles with an odd number of edges of value 1. This depends on the number of edges in the graph. The more arcs there are, the greater the risk that there is a non-satisfiable cycle.

In the above graphical construction of 2-$XORSAT$ systems the probability of choosing any given variable is $1/N$. Likewise, the probability that the same variable is at the other end of the edge is also $1/N$. Thus, with M randomly drawn edges, the average degree of each vertex in the graph is $M(1/N + 1/N) = 2\alpha$.

Random graphs have been investigated by Erdös and Renyi among others and their properties are well known (see, e.g., [16] and references therein). Some of the most important ones can be summarized thus:

- If $2\alpha < 1$ the 2-$XORSAT$ system graph is essentially composed of independent trees. This is understandable for, if $2\alpha < 1$, there is on average less than one edge per vertex and thus a low probability of having a cycle.

- if $2\alpha > 1$, the graph possesses a *giant component* that contains $\mathcal{O}(N)$ vertices. There are many cycles and the probability that there exists a solution decreases with increasing N.

From these observations we can conclude that the satisfiability of a problem changes for a critical value of $\alpha_c = 1/2$. In fact, if α is small the problem is almost surely satisfiable. When α approaches $1/2$ P_{SAT} slowly decreases at first, and then more quickly in the vicinity of $\alpha = 1/2$. For $\alpha > 1/2$ the probability that the problem is satisfiable tends to 0 as $N \to \infty$. The relation P_{SAT} as a function of α is illustrated in Figure 10.6 (see [63] for the mathematical details).

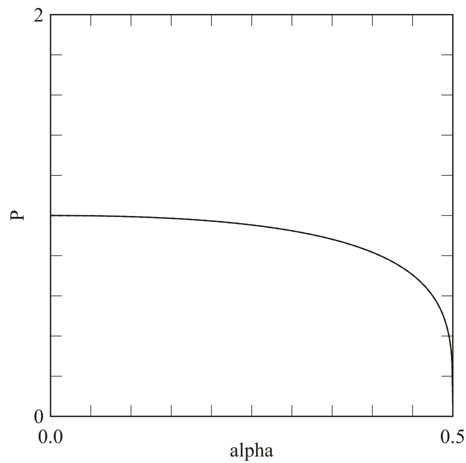


Fig. 10.6. Probability of satisfiability of a 2-$XORSAT$ problem as a function of α in the limit for $N \to \infty$. See [63] for details

10.5.3 The 3-XORSAT Case

3-$XORSAT$ problems are constituted by M equations of the form

$$x_{i_1} + x_{i_2} + x_{i_3} = \nu$$

where each equation contains three distinct variables among the N total variables.

If some variables only appear in one equation in the system of equations the problem can be simplified. For instance, if x_0 only appears in the equation

$$x_0 + x_i + x_j = \nu$$

we can conclude that

$$x_0 = \nu + x_i + x_j$$

thus eliminating the variable x_0 and the corresponding equation. The process is an iterative one in the sense that after a variable and equation elimination, other variables may appear in only one equation and be eliminated as well. The following example shows this:

$$\begin{cases} x_1 + x_2 + x_3 = \nu_1 \\ x_2 + x_4 + x_5 = \nu_2 \\ x_3 + x_4 + x_5 = \nu_3 \end{cases} \tag{10.21}$$

The x_1 variable is eliminated first by solving $x_1 = \nu_1 + x_2 + x_3$. We are then left with two equations

$$\begin{cases} x_2 + x_4 + x_5 = \nu_2 \\ x_3 + x_4 + x_5 = \nu_3 \end{cases} \tag{10.22}$$

in which we can eliminate x_2 and x_3. As a result, there are no equations left and we conclude that the problem is satisfiable; in fact, we can choose any values for x_4 and x_5.

In a general way, through this iterative elimination procedure the original problem S is reduced to a smaller problem S'. In the limit of large N one can show [63] that with high probability S' is empty if

$$\alpha < \alpha_d = 0.8184 \tag{10.23}$$

and therefore the problem is satisfiable. For

$$\alpha_d \leq \alpha < \alpha_c = 0.9179 \tag{10.24}$$

it turns out that S' is not empty but solutions may still exist. In contrast, for $\alpha > \alpha_c$ there are no solutions and the problem becomes unsatisfiable with high probability for $N \to \infty$. There is thus a hardness phase transition for $\alpha = \alpha_c = 0.9179$, as shown in Figure 10.1.

In addition to showing the phase transition from states in which solutions exist with high probability to states in which the system is unsatisfiable, it is also of interest to investigate the structure of the search space, that is, how solutions are distributed in the N-dimensional hypercube describing the set of all possible x_i values. What is found (see [63]) is that, for $\alpha < \alpha_d = 0.8184$, solutions are uniformly distributed in the hypercube. In contrast, for $\alpha_d \leq \alpha < \alpha_c = 0.9179$, solutions are clustered without connections between the clusters. There is thus another kind of transition, this time in the solution space structure: instead of having solutions that become increasingly rare in a homogeneous fashion with increasing α, we find for $\alpha = \alpha_d$ a discontinuous rarefaction of solutions. This is illustrated in Figure 10.7. In this figure, two 3-$XORSAT$ systems are considered, with $N = 50$ and $M = 30$ and $M = 40$, which gives $\alpha = 0.6$ and $\alpha = 0.8$ respectively. These two linear systems have been solved by Gaussian elimination as explained in Section 10.4. Figure 10.7 shows all the solutions of these two systems. On the x-axis all the values of the bit

string $x_{i_1} x_{i_2} \dots x_{i_n}$ are reported in decimal, the x_{i_ℓ} being the free variables. On the y-axis the constrained variables $x_{j_1} x_{j_2} \dots x_{j_m}$ are reported, also in decimal format. We refer the reader to Section 10.4 for an example of this type of representation. Here, what we wanted to point out is the phase transition in the structure of the solution space.

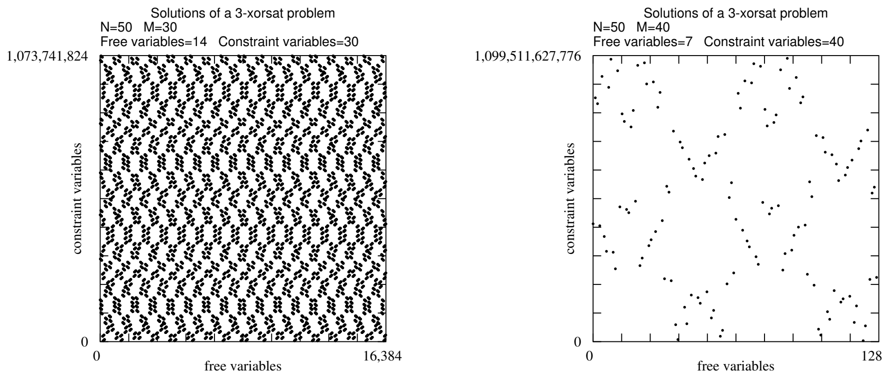


Fig. 10.7. An example of structural phase transition in the solution space of a 3-$XORSAT$ problem for the different difficulty regimes: in the left image $\alpha < \alpha_d$, in the right image $\alpha_d \approx \alpha < \alpha_c$. One can see a transition from a situation with solutions homogeneously distributed in space to a situation in which solutions tend to form clusters on a global scale. For $\alpha > \alpha_c$ the system becomes unsatisfiable and there are no solutions at all in the space

10.6 Behavior of a Search Metaheuristic

In this section we will look at the behavior of a metaheuristic for solving 3-$XORSAT$ problems. We recall that 3-$XORSAT$ problems can be solved by Gaussian elimination, which takes polynomial time. Therefore, there is no need to use a metaheuristic to know whether a given problem instance is satisfiable or not. However, if a problem is non-satisfiable, an optimization metaheuristic will tend to minimize the number of non-satisfied constraints. It is interesting to watch the behavior of such a metaheuristic according to the problem difficulty as specified by the α parameter value. As one would expect, when solutions become rare, the metaheuristic will be less and less effective at finding them. Indeed, we shall see that there can be a phase transition linear/exponential in the mean time complexity.

10.6.1 The RWSAT Algorithm

The algorithm called “Random Walk SAT” (RWSAT) was proposed in 1991 by C. Papadimitriou to solve 3-$XORSAT$ problems using a simple metaheuristic. The

idea is to choose a random variable assignment of the N variables and, while there are unsatisfied equations, pick any unsatisfied equation and flip a random variable in it. This will satisfy the corresponding equation but other equations that were previously satisfied may become unsatisfied. The process is repeated until either all the equations are satisfied, or a prescribed number of iterations t has been reached.

The algorithm can easily be described in pseudo-code form. We first define a quantity E which equals the number of non-satisfied equations. The objective of the algorithm is to minimize E. In a physical interpretation, E would represent a kind of system energy which corresponds to the cost of having unsatisfied equations. The optimal situation obviously corresponds to no unsatisfied equation, i.e., to $E = 0$. The pseudo-code of the algorithm follows:

```
initialize all N variables at random
compute E
t=0
while(E>0 and t<max)
   choose at random a non-satisfied equation
   choose at random one of its k variables
   change the value of that variable to its complement
   compute E
   t=t+1
end while
print E, t
```

One might ask how long it would take to conclude that the problem is unsatisfiable (UNSAT) if $E > 0$ and that the result is not simply due to bad luck. Actually, as we know from previous chapters, nothing guarantees a priori that, if a solution exists, this metaheuristic will find it in finite time. However, it can be shown that if we iterate the algorithm T times, each time with `max`$= 3N$, if no solution has been found during the T repetitions, the probability that the problem is satisfiable is

$$P_{SAT} \leq e^{-T\left(\frac{3}{4}\right)^N} \tag{10.25}$$

We now look at the behavior of RWSAT for different values of $\alpha = M/N$. In particular, we want to investigate how $e = E/M$ varies as a function of the iteration number t. In Figures 10.8 and 10.9 (blue curve) we see that for small α the energy decreases toward $E = 0$ with increasing t (blue curve), meaning that a variable assignment that satisfies the formula has been found.

For higher values of α (Figure 10.9, red curve), E starts decreasing in a similar way but the solution condition $E = 0$ is not attained. Instead, $E(t)/M$ oscillates about a plateau value $e_p > 0$. Only a larger random fluctuation may make $E(t)$ go to zero. But it is also seen in Figure 10.9 that the amplitude of the fluctuations decreases with increasing N, which implies that the probability that RWSAT produces a large enough fluctuation to reach $E = 0$ tends to zero for $N \to \infty$.

The behavior described above can be quantified by measuring the average time $\langle T_{res} \rangle$ the RWSAT algorithm needs to solve a sample of randomly generated 3-

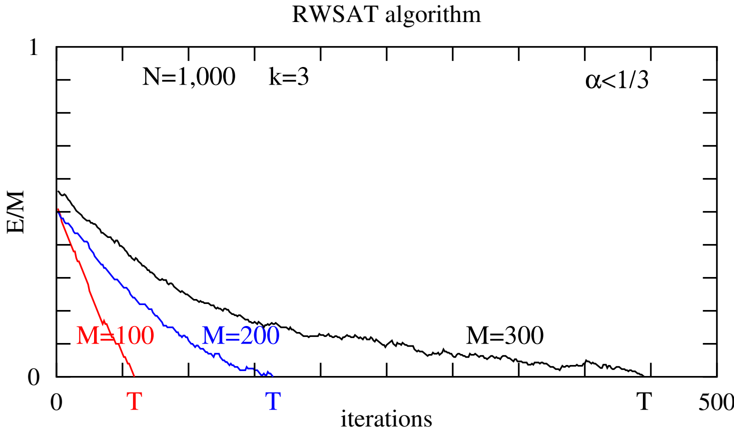


Fig. 10.8. RWSAT behavior on a 3-$XORSAT$ problem with $\alpha = 0.1$, $\alpha = 0.2$, and $\alpha = 0.3$. The indicated T value when the energy goes to zero gives the resolution time for each problem

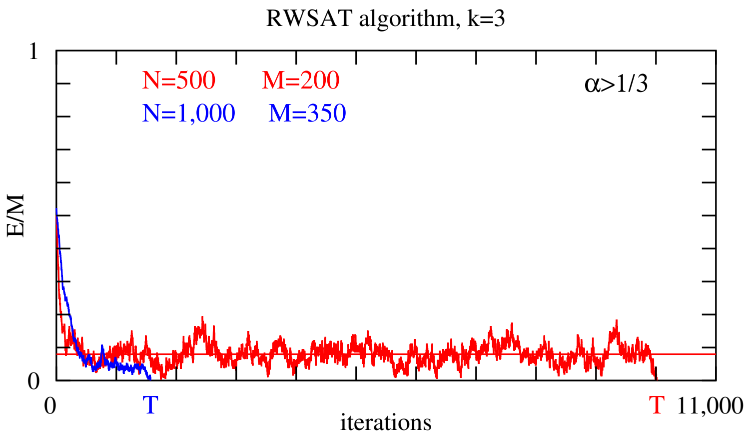


Fig. 10.9. Typical behavior of RWSAT on a 3-$XORSAT$ problem with $\alpha > 1/3$. Blue curve: $\alpha = 0.35$; red curve: $\alpha = 0.40$. The indicated T values give the respective resolution times

$XORSAT$ instances, time being counted here as the number of iterations. For a sample size of 100 systems, we obtain the typical behavior shown in Figure 10.10. For a given α close to $1/3$ one sees that the resolution time increases in a dramatic way, probably exponentially, since simulations become prohibitively lengthy and cannot be run in reasonable time.

In reference [63] the above behavior is described by introducing a new critical α value called α_E:

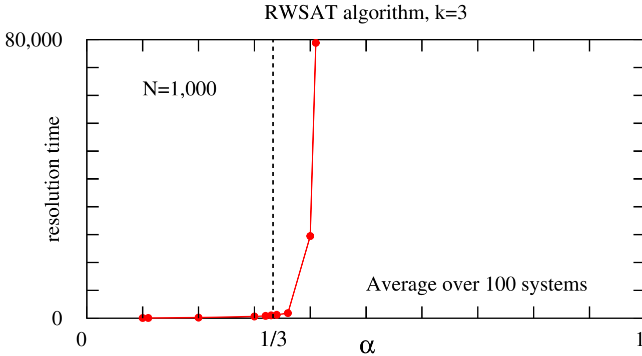


Fig. 10.10. RWSAT average solution times on 3-$XORSAT$ problem instances as a function of α for instance size $N = 1{,}000$. The average is taken over 100 randomly generated instances. Note the sudden increase of solution time when $\alpha > 1/3$

- If $\alpha < \alpha_E = 0.33$ then $e_p = 0$ and solution time increases linearly with the problem size N: $\langle T_{res} \rangle = N t_{res}$, where t_{res} is a factor that increases with α. This linear behavior is illustrated in the left part of Figure 10.11.
- If $\alpha_E \leq \alpha < \alpha_c$ then $e_p > 0$ and solution time becomes exponential in N: $\langle T_{res} \rangle = A \exp(N \tau_{res})$, where A is a coefficient and τ_{res} is a factor that grows with α, as seen in the right part of Figure 10.11.

We therefore have a new phase transition, this time for the time complexity of the algorithm.

The slopes of the straight lines

$$\log(\langle T_{res} \rangle) = \log(A) + N\tau_{res}$$

shown in Figure 10.11 suggest here that $A(\alpha = 0.4) = 10^{2.43}$, $\tau_{res}(\alpha = 0.4) = 0.0018$ and $A(\alpha = 0.5) = 10^{2.02}$, $\tau_{res}(\alpha = 0.4) = 0.015$.

10.6.2 The Backtracking Algorithm

The reason why the RWSAT metaheuristic finds it increasingly difficult to discover a solution as α increases is related to the rarefaction of solutions in the search space, and also to the fact that an increasing number of clauses contain the same variables, creating conflicts when a variable is modified by RWSAT.

It is interesting to consider another well-known algorithm for the k-$XORSAT$ problem: backtracking. Backtracking is an exact algorithm, not a metaheuristic. It systematically explores the search space and always finds the existing solutions if the problem is small enough to allow such an exhaustive exploration. Backtracking is a general algorithmic method that applies to many problems of different natures. The specificity of backtracking is in the stopping criterion: if a partial solution has no chance of success, we must back up and reconsider the earlier choices, to continue the

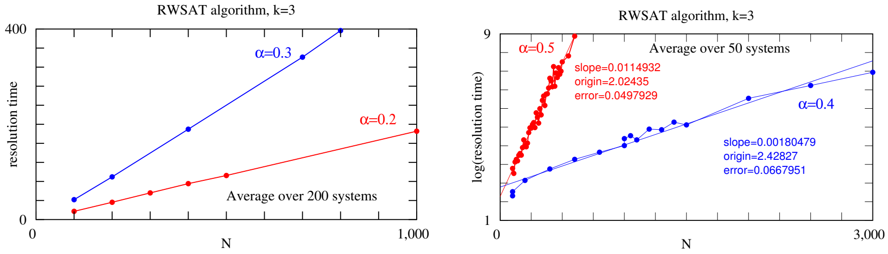


Fig. 10.11. Mean solution times for RWSAT. Measured points are shown, as well as their linear fits. Left image: illustration of RWSAT's $\mathcal{O}(N)$ mean complexity growth for $\alpha < 1/3$. Right image: beyond $\alpha = 1/3$ RWSAT's mean complexity increases exponentially, in agreement with $\langle T_{res} \rangle = \exp(N\tau_{res})$. To better appreciate the exponential behavior, $\log_{10}\langle T_{res} \rangle$ is represented as a function of N, giving a straight line in this semi-logarithmic plot

process until either a solution is found, or all the possibilities have been exhausted. We shall see by numerical experimentation that if α goes beyond a threshold the size of the space that has to be searched grows abruptly.

The 3-$XORSAT$ problem can be solved by backtracking in the following way. The binary variables x_i are progressively assigned according to a binary tree that is traversed in depth-first fashion. For instance, one can start by attributing the value 0 or 1 to x_1. Suppose we choose 1 and then go on to the assignment of the value of x_2. Here too two values are possible. Let's assume that 1 is chosen, i.e., $x_2 = 1$. The process continues with the remaining variables and, in each step, we check that the equations for which the variable values are known are satisfied. If we face a contradiction, we change the last assigned variable from 1 to 0. If, again, there is a contradiction, we back up one more step and so on until we find a partial variable assignment $x_1, \ldots, x_i$ that is compatible with the system equations. At this point, we assign the following variable x_{i+1} starting with the value 1 and continue the process in a recursive manner. We note that, in each step, before checking the termination criterion, we can also assign all the variables that belong to equations where the other two variables are known. The backtracking algorithm finds a solution only if the equations are satisfiable; otherwise it can be concluded that the problem is unsatisfiable. Thus, the algorithm is a *complete* one, always providing the exact answer.

The performance of the algorithm can be measured by the number of search tree nodes that the backtracking procedure generates before either finding a solution, i.e., a complete assignment of x_i, $i = 1, \ldots, N$ that verifies the M equations, or concluding that no solution exists. Again, we observe a threshold behavior typical of phase transitions. Beyond a critical value α_E[1], the backtracking algorithm spends a time which is exponential in N to solve the problem, while for $\alpha < \alpha_E$ this time is linear [63]:

[1] This value is not the same as the one found for RWSAT.

$$\langle T_{res} \rangle = \begin{cases} Nt & \text{if } \alpha < \alpha_E \\ \exp(N\tau) & \text{if } \alpha > \alpha_E \end{cases} \tag{10.26}$$

This behavior is illustrated in Figure 10.12, which shows the number of nodes explored by backtracking as a function of α. We see that the maximum times are found around $\alpha = \alpha_c$, beyond which the 3-$XORSAT$ problems become unsatisfiable in the limit for $N \to \infty$. For $\alpha > \alpha_c$ we note that the time taken by the backtracking algorithm decreases. This is simply due to the fact that in this region, the larger α is, the sooner the algorithm will realize that there are no chances of finding a solution and the stopping function will be activated, avoiding the generation of many search tree nodes. We note also that in the non-satisfiable phase $\tau \propto 1/\alpha$.

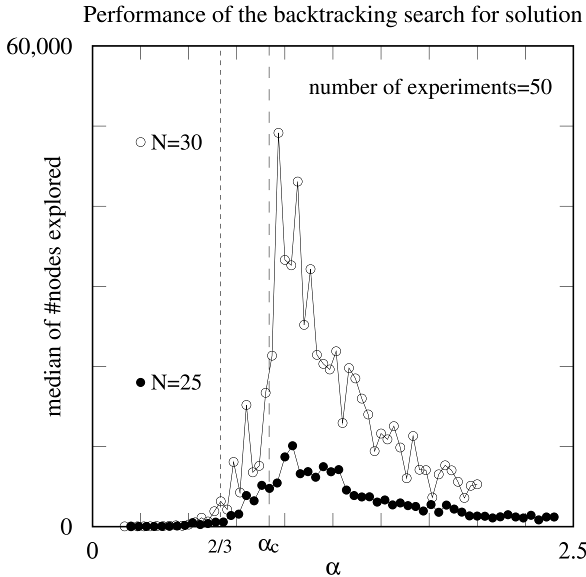


Fig. 10.12. Median value of the number of nodes explored by the backtracking algorithm to solve 3-$XORSAT$ problems with N variables and $\alpha = M/N$, where M is the number of equations. The algorithm terminates either when a solution is found or when all possibilities have been exhausted

Another way to understand the reason for the computational phase transition is given by the analysis of the path taken on the search tree by the backtracking algorithm. Figure 10.13 shows that in the linear phase the algorithm does little or no backtracking: only a small part of the tree is explored since there are many solutions and it is easy to find one of them. In contrast, in the exponential-complexity phase a large part of the whole tree, which contains 2^N nodes in total, must be generated before finding a solution or deciding that there is none.

The α_E value depends of the way the variables x_i are assigned at each tree level. If the assignment is done in the order specified by the variables' indices then in reference [63] it is shown that

$$\alpha_E = 2/3$$

But if one is more clever and first assigns the variables that appear in the equations that still contain two free variables, then it is found numerically that

$$\alpha_E = 0.7507$$

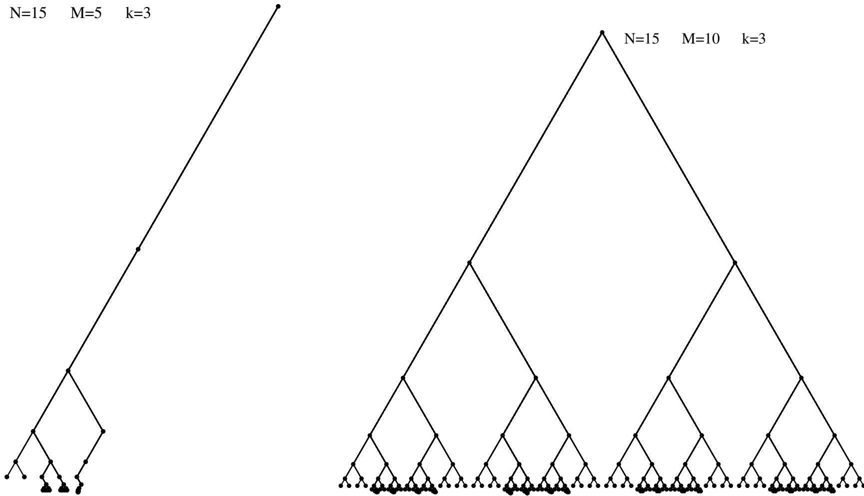


Fig. 10.13. Typical backtracking path on the search tree as a function of α. Left side: a small value of $\alpha = 1/3$; right side: larger value of $\alpha = 2/3$. There is a phase transition between a regime in which a solution is found with a short exploration, and a regime in which a significant fraction of the search tree must be explored

From the computational point of view, we thus remark that algorithms such as backtracking and a metaheuristic such as RWSAT also show a phase transition in their ability to solve a 3-$XORSAT$ problem. The "easy" phase is characterized by a linear computing time in the problem size, while the "hard" phase requires a time that is exponential in the problem size. The α values that define these transitions are lower than the theoretical values α_c and α_d, and they have a strong dependence on the algorithm considered.

11

Performance and Limitations of Metaheuristics

11.1 Empirical Analysis of the Performance of a Metaheuristic

In contrast with exact algorithms whose worst-case time complexity is known (see Chapter 1), metaheuristics do not provide that kind of bound. They can be very effective on a given instance of a problem and, at the same time, show long running times on another without finding a satisfactory solution. On the other hand, for example, the selection sort algorithm could spend different amount of time on an already sorted list, and on a list sorted in the opposite order, but we know that, on any list permutation, its time complexity function $T(n)$ will be bounded by a second-degree polynomial and the list will be sorted correctly. However, for hard problems of increasing size such guarantees are useless in practice since the problems become intractable. This was exactly why we looked at metaheuristics as a generally efficient way of tackling hard problems. As we remarked in Chapter 1, more rigorous methods giving performance guarantees do exist for difficult problems but they are not as general and easy to apply.

In the case of metaheuristics, the hope is clearly to get polynomial-time-bounded computations for difficult problems, but we cannot be sure that this will be the case; moreover, we have no guarantee that the solution found will be globally optimal either, or at least of high quality. The computing time can vary depending on the problem, the particular instance at hand, the chosen metaheuristic and its parameters. In addition, almost all the metaheuristics illustrated in this book employ random choices, which means that the computing time, as well as the solution quality, are actually random variables. The stochastic nature of most metaheuristics makes their rigorous analysis difficult. We have seen that this kind of analysis is possible in certain cases such as evolutionary algorithms or simulated annealing [8], but the results, being in general asymptotic, are of little help.

Given these considerations, researchers have been led to take into account empirical methods to measure the performance of a metaheuristic and to compare metaheuristics with each other. The approach is essentially based on standard statistical methods and the goal is to be able to ensure that the results are statistically significant. In what follows we assume that the parameters that characterize a given metaheuris-

© Springer Nature Switzerland AG 2018
B. Chopard, M. Tomassini, *An Introduction to Metaheuristics for Optimization*,
Natural Computing Series, https://doi.org/10.1007/978-3-319-93073-2_11

tic have been chosen and that they do not change during the measurement. As we have seen in the previous chapters, correctly setting these parameters is very important for the efficiency of the method, for example the initial temperature in simulated annealing (see Chapter 4), or the mutation rate in an EA (Chapter 8). Usually, these parameters are either set by using standard values that worked on other problems or the algorithm is run a few times and suitable values are found. To simplify and unify the treatment, here we shall assume that this choice has already been made. In a similar vein, we will ignore more sophisticated metaheuristics in which parameters are dynamically changed online as a result of learning during the search.
To test a given metaheuristic on one or more classes of problems, or to compare two metaheuristics with each other, the computational experimental methodology is much the same and goes through the following phases:

- Choice of problems and of their instances;
- Choice of performance measures and statistical analysis of the results;
- Graphical or tabular presentation of results and their discussion.

11.1.1 Problem Choice

There are fundamentally two broad classes of problems to choose from: real-world instances that come from operations research, engineering, and the sciences, and "synthetic" problems, which are those that are artificially constructed with the goal of testing different aspects of search. The approach is essentially similar in both cases; however, given the didactic orientation of our book, we shall make reference only to constructive and benchmark problems in the rest of the chapter.

Problem-based benchmark suites are a good choice because they allow one to conceive of problems with different features. Benchmark functions for continuous optimization are very widely used because of the practical importance of the problem in engineering, economics, and the sciences. These benchmarks must contain diverse functions so as to test for different characteristics such as multimodality, separability, nonlinearity, increasing dimensions, and several others. A recent informative review on test functions for mathematical optimization can be found in [44]. For reasons of space, in the rest of the chapter we will limit ourselves to combinatorial optimization test problems. In this case, the important features that are offered in standard repositories of problem instances such as SATLIB or TSPLIB are a range of instance sizes, the amount of constrainedness, and the way in which the problem variables are chosen. On the last point, and roughly speaking, there are two types of instances in benchmark suites for discrete problems: artificially built instances and randomly generated instances. Artificial instances can be very useful: they can incorporate certain characteristics of real-world instances, and they can also help expose particular aspects that are difficult to find in real-life problems. Randomly generated instances are very frequently used too, for example in SAT problems or TSP problems. They

have the advantage that many instances can be generated in an unbiased way, which is good for statistical analysis; on the other hand, deviations from randomness are very common in combinatorial problems and thus these instances might have little to do with naturally occurring problems. In any case, as we discussed above for global mathematical optimization, we should use sufficiently varied sets of test functions, including at least both random and structured instances, once the parameters of the metaheuristics have been set.

11.1.2 Performance Measures

Among the most interesting data to measure about the search behavior of a metaheuristic we mention the *computational effort*, that is, the computing time, and the *solution quality* that can be obtained with a given computational effort. Computing time can be measured in two ways: either as the physical time elapsed, which is easily recorded through calls to the operating system, or as a relative quantity such as the number of operations executed. In optimization, an even simpler often-used quantity is the number of objective function evaluations. Using the clock time is useful for practitioners who are only interested in the behavior of a solver on a given computer system for a restricted class of problems. However, there are several drawbacks. Clock times depend on the processor used and also on other hardware and software details such as memory, cache, operating system, languages, and compilers. This makes comparisons across different systems difficult, if not impossible. On the other hand, the number of function evaluations is system-independent, which makes it useful for comparisons. A possible drawback of this measure is that it becomes inadequate if the problem has a time-varying fitness function, or when the objective function evaluation only accounts for a small fraction of the total computing time, giving results that cannot reliably be generalized. In spite of some limitations, fitness evaluation counts are widely used in performance evaluation measures because of their simplicity and independence from the computing system.

We remind the reader at this point that the most important metaheuristics belong to the class of *Las Vegas* algorithms, which are guaranteed to only return correct solutions but whose running time may vary across different runs for the same input. Such an algorithm may run arbitrarily long without finding a global solution. Consequently, the running times, as well as the solution quality, are random variables. To measure them in a statistically meaningful way, the algorithm must be run many times on each given instance under the same conditions in order to compute reliable average values. In other words, as in any statistical application, the sample size must be large enough. In the performance evaluation domain it is considered that at least 100 executions are needed.

Several metrics have been suggested to characterize the performance of a metaheuristic and an established common methodology is still missing. However, there is a general agreement on a number of fundamental measures. Here we shall present two very common ones: the empirical distribution of the probability of solving a given problem instance as a function of the computational effort, and the empirical distribution of the obtained solutions. The *success rate*, which is the number of

times that the algorithm has found the globally optimal solution divided by the total number of runs, is a simple metric that can be derived from the previous two and it is often used. Clearly, it is defined only for problems of which the globally optimal solution is known, which is the case for most benchmark problems.

11.1.3 Examples

We have already mentioned some results that are in the spirit of the present chapter in Chapter 5 on ant colony methods, and in Chapter 9 on GP multipopulations. In order to further illustrate the above concepts, in this section we present a couple of simple case studies. We focus our discussion on two particular metaheuristics: simulated annealing (see Chapter 4) as applied to the TSP problem, and a genetic algorithm on NK problems. The latter class of problems has been already used a few times in the book and is defined in Section 2.2.3.

From the perspective of performance evaluation, it is interesting to find the mean computing time that a metaheuristic needs to solve problem instances of a given class when the size N of the instance increases. Indeed, the main motivation behind the adoption of metaheuristics is the hope that they will solve the problem in polynomial time in N, whereas deterministic algorithms, essentially complete enumeration, require exponential time on such hard problems. It would certainly be reassuring to verify that we may actually obtain satisfactory solutions to the problem in reasonable time, otherwise one might question the usefulness of metaheuristics.

We recall here that the RWSAT metaheuristic presented in Chapter 10 has $\mathcal{O}(N)$ complexity for "easy" problems, that is, those problems for which the ratio α of the number of clauses to the number of variables is small. However, the complexity suddenly becomes exponential $\mathcal{O}(\exp(N))$ when $\alpha > \alpha_c$, where α_c is the critical point at which the computational phase transition occurs. It is thus natural to investigate the complexity behavior of simulated annealing and of genetic algorithms. But, as we have already pointed out above, owing to their stochastic character, we can only define the time complexity of metaheuristics in a statistical way. Another distinctive point is that metaheuristics, being unable to guarantee convergence to the optimal solution in bounded time, must have a built-in stopping criterion. For example, we might allow a maximum of m iterations during which there are no improvements to the best fitness found and then stop. We might therefore measure the mean time to termination as a function of the instance size N. In this case we will not be able to obtain any precise information about the quality of the found solution, only the expected computational effort to obtain an answer.

Figure 11.1 shows the results obtained with simulated annealing on TSP instances with a number of cities N between 20 and 50,000. For each N value, N cities are first randomly placed on a square of given size and then an SA run is started using the parameters proposed in Section 4.6. The movements we consider here are of type *2-Opt*. The search stops when, during the last three temperature levels, there was no improvement of the current best solution. A priori, we have no indication of the quality of the solution. However, we remember from Section 4.3 that the solution found was within 5% of the exact optimum obtained with the *Concorde* algorithm.

In Figure 11.1 we can see that the time to obtain a solution for the TSP problem with random city placement grows almost linearly in the range $N = 20$ to $N = 2{,}000$. The curve can be fitted by the following second-degree polynomial

$$T(N) = 7{,}100 \times N^{1.14} \qquad (11.1)$$

Here $T(N)$ is measured as the number of iterations of the SA run until convergence. It corresponds to the total number of accepted and rejected configurations, that is the number of fitness evaluations[1]. For N in the range 5,000-50,000, the computational effort, $T(N)$, is more important[2]. We obtain the following relation

$$T(N) = 35.5 \times N^{1.78} < \mathcal{O}(N^2) \qquad (11.2)$$

Thus, there is a change of complexity regime between small and large problems. However, for these values of N, the complexity is less than quadratic.

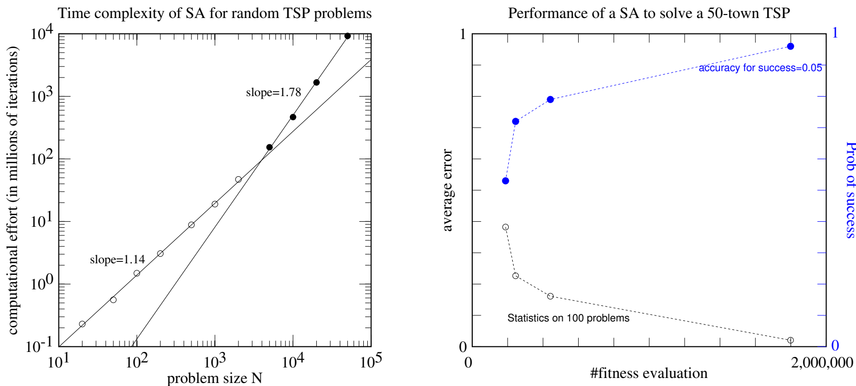


Fig. 11.1. Left image: average time complexity for simulated annealing in solving a TSP problem with N cities randomly placed in a square of size 2×2. Right image: simulated annealing performance on a TSP problem with 50 cities of which the optimal solution is known. The SA computational effort is varied by changing the temperature schedule

Now we consider the SA performance from the point of view presented in Section 11.1.2. The goal here is to determine the quality of the solution found with a given computational effort. First of all, we should explain how to vary the effort of a metaheuristic. This is easy to do by simply changing the termination condition. For simulated annealing, it is also possible to change some parameter of the algorithm, for example the temperature schedule, that is the rate at which the temperature T is decreased. We remember that practical experience suggests that $T_{k+1} = 0.9T_k$. However, we might replace 0.9 by 0.85 for faster convergence and less precision,

[1] The actual computational time varies from 0.03 to 4.3 seconds with a standard laptop.

[2] The CPU time varies from 11 to 1,000 seconds on a laptop.

or take 0.95 or 0.99 for slower convergence and better solution quality. This is the approach adopted here.

The numerical experiment is performed on a benchmark problem of which the globally optimal solution is known, which allows us to compare the error of the tour returned by SA at a given computational effort with respect to the length of the optimal tour. The problem is of size 50 cities distributed on a circle of radius one. The optimal tour is a polygon with 50 sides and length $L = 6.27905$. The results are averages over 100 problem instances that differ in their initial conditions and in the sequence of random numbers generated.

Figure 11.1 (right) shows two indicators of performance as a function of the computational effort measured as the number of function evaluations. The black curve plots the mean error of the solution found with respect to the optimal tour of length $L = 6.27905$ over 100 SA runs. It is easy to see that the precision of the answer improves if we devote more computational resources to simulated annealing.

The points on the blue curve are estimates of the success rate, or of the probability P_ϵ of success on this problem if we require a precision of $\epsilon = 0.05$. The P_ϵ value is obtained as follows for each computational effort:

$$P_\epsilon = \frac{\text{number of solutions with an error less than } \epsilon}{\text{number of repetitions}} \tag{11.3}$$

One can see here that the probability of a correct answer at the $\epsilon = 5\%$ level tends to 1 for a computational effort exceeding 2×10^6 iterations. This means that almost all the 100 runs have found a solution having this precision. To appreciate the magnitude of the computational effort, we may recall here that the search space size for this problem is $50! \approx 1.7 \times 10^{63}$.

Clearly, if we increased the required precision by taking a smaller ϵ, the success rate would correspondingly decrease. Also, while the performance measured is specific to this particular problem, that is, all the cities lie on a circle, the observed behavior can be considered qualitatively general. We should thus expect that finding the optimum would be increasingly difficult, without adding computational resources, if we are more demanding on the quality of the solution we want to achieve.

This is true in general whatever the metaheuristic examined. To see this, we shall now consider the performance of a genetic algorithm in solving problems in the NK class. As in the TSP case above, we describe first the empirical average time complexity of a GA for an NK problem with varying N and constant $K = 5$.

For each N value, 500 NK problem instances are randomly generated. Periodic boundary conditions are used in the bit strings representing configurations of the system. The fitness of a bit string $\mathbf{x} = x_0 x_1 \ldots x_{N-1}$ of length N is given by

$$f(\mathbf{x}) = \sum_{i=0}^{N-1} h(x_{i-2}, x_{i-1}, x_i, x_{i+1}, x_{i+2}) \tag{11.4}$$

where h is a table with 32 entries (2^K in the general case), randomly chosen among the integers 0 to 10. This allows us to generate landscapes that are sufficiently difficult but not too hard. For each of the 500 generated instances the optimal solution

is found by exhaustive enumeration, that is, by evaluating all the 2^N possible solutions. The goal here is clearly to have an absolute reference for the evaluation of the performance of the GA on this problem.

The chosen GA has a population size of 100 individuals, one-point crossover with crossover probability 0.6, and a mutation probability of 0.01 for each of the N bits of $\mathbf{x}$. The best individual of each generation goes unchanged to the next generation, where it replaces the worst one. We allow a maximum number of 80 generations and we compute the computational effort as the number of function evaluations.

After each generation ℓ a check is made to see whether the known exact solution has been found. If this is the case, the computational effort for the instance at hand is recorded as $100 \times \ell$. If the best solution has not been found, the GA iteration continues. The maximum computational effort is thus $100 \times 80 = 8{,}000$. If the solution has not been found after 80 generations, we shall say that the GA has failed, which will allow the computation of an empirical failure probability at the end. If the empirical failure probability is denoted by p_f, the corresponding success probability is $1 - p_f$. The motivation for introducing a failure probability is the observation that, if the exact solution is not found in a reasonable number of generations, it will be unlikely to be found later. In fact, if we increase the number of allowed generations beyond 80, the probability of failure doesn't change significantly. For this reason, it is necessary to separate solvable instances from those that are not in order not to bias the computational effort; otherwise the latter would be influenced by the choice of the maximum number of allowed generations, which can be arbitrarily large.

Figure 11.2 depicts the results that have been obtained. Here the number of fitness evaluations is the average value over all the problem instance that have been solved optimally within a computing time corresponding to at most 80 generations. We see that this time is indeed small when compared with the upper limit of 8000 evaluations.

We also remark that the empirical average computational complexity grows essentially linearly, which is encouraging, given that NK landscapes have an exponentially increasing complexity. On the other hand, it is also seen in the figure that the failure probability increases with N, and there are more and more problems that the GA cannot solve exactly.

We now characterize the GA performance in a slightly different way on the same class of problems. Thus, we keep the same GA parameters and the same problem instance generation as above but we change the termination condition into a stagnation criterion instead of a hard limit on the number of generations. The condition now becomes the following: if during m consecutive generations the best fitness has not been improved, the GA stops. We then save the best solution found up to this point and the number of generations elapsed. As before, the baseline for comparing the results is the exhaustive search for the optimal solutions for each N.

After the 500 repetitions for each value of N, we can compute and plot the average computational effort, given that we know how many generations were needed in each run. We can also compute the mean relative error with respect to the known optimum, and the number of times the best solution found was within a precision interval ϵ around the exact solution. Finally, we can vary the computational effort by

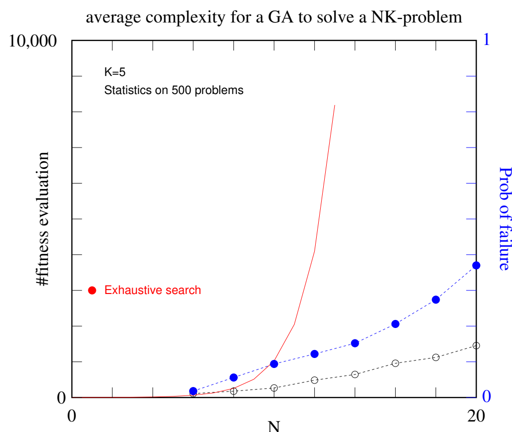


Fig. 11.2. Black curve: average computational complexity of a genetic algorithm in terms of function evaluations for solving NK problems with N between 8 and 20 and $K = 5$. Each point is the average of 500 randomly generated instances of the corresponding problem. Blue curve: fraction of problems for which the global optimum has not been found within the allowed time limit corresponding to 8,000 evaluations. Red curve: number of function evaluations required by exhaustive search of the 2^N possible solutions

varying the value of m. The results presented in Figure 11.3 are for m between 3 and 15, $N = 18$, two values of K, $K = 5$ and $K = 7$, and two values of the precision, $\epsilon = 0.1$ and $\epsilon = 0.02$.

As expected, the average error decreases with increasing computational effort; the solution quality is improved by using more generations, and the problems with $K = 5$ are easier than those with $K = 7$.

The previous results make it clear that there is a compromise to be found between the computational effort expended and the quality of the solutions we would like to obtain. High-quality solutions require more computational effort, as seen in the figure. It should be said that in the present case the solution quality can be compared with the ideal baseline optimal solution, which is known. Now, very often the globally optimal solution is unknown, typically for real-life problems or for very large benchmark and constructive problems, for instance NK landscapes with, say, $N = 100$ and $K = 80$. In this case, the best we can do is to compare the obtained solutions to the best solution known, even if we don't know whether the latter is globally optimal or not. For some problems, one can get a reliable approximate value for theoretical lower bounds on solution quality by using Lagrangian relaxation or integer programming relaxation (see Chapter 1). In these cases, the solutions found by the metaheuristic can be compared with those bounds.

Quite often performance measures similar to the ones just described are obtained in the framework of comparative studies between two or more different metaheuris-

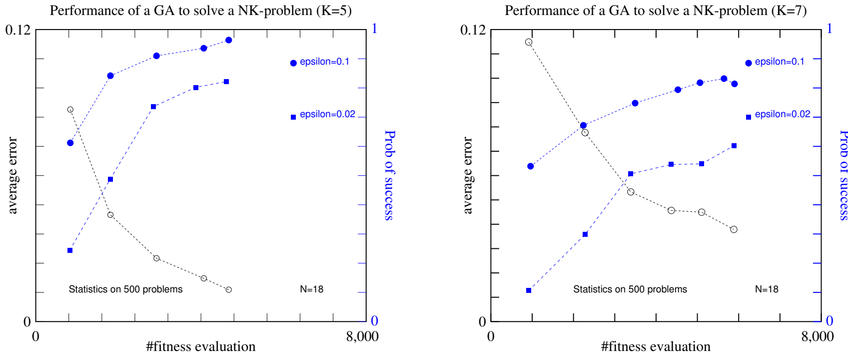


Fig. 11.3. Performance curves for the GA described in the text for solving NK problems as a function of the computational effort. The left graphic corresponds to $K = 5$ and the right curve is for $K = 7$. On both panels the probability of success is in blue for two values of the precision, $\epsilon = 0.1$ and 0.02. The black curves give the average relative error

tics with the goal of establishing the superiority of one of them over the others. This kind of approach can be useful when it comes to a particular problem or a well-defined class of problems that are of special interest for the user. However, as we shall see in the next section, it is in principle impossible to establish the definitive superiority of a stochastic metaheuristic with respect to others. This doesn't prevent researchers from trying to apply robust statistical methods when comparing metaheuristics with each other. The approach is analogous to what we have just seen applied to a single metaheuristic. However, when comparing algorithms, one must be able to establish the statistical significance of the observed differences in performance. In general, since samples are usually not normally distributed, non-parametric statistical tests are used such as the *Wilcoxon*, *Mann-Whitney*, and the *Kolmogorov-Smirnov* or *Chi-squared* tests for significant differences in the empirical distributions [69].

To recapitulate, performance evaluation in the metaheuristic field is a necessary and useful step. The examples illustrated here were chosen for their simplicity and familiarity to the reader rather than their importance, in order to bring the main message home without unnecessary complication. The message is twofold: on the one hand, we discussed some common metrics that are useful for characterizing the performance of a metaheuristic and, on the other hand, using those measures, we showed that two metaheuristics on a couple of difficult but not extremely hard versions of the problems require much less computational resources than exhaustive enumeration to obtain very good solutions. Of course, these conclusions cannot immediately be generalized to other problems and other metaheuristics without studying their performance behavior, but at least the examples suggest that the metaheuristics approach to hard problem solving is a reasonable one.

The field of performance measures and their statistics is varied and complex; here we have offered an introduction to this important subject but, to avoid making the text more cumbersome, several subjects have been ignored. Among these, we might cite the *robustness* of a metaheuristic and the parallel and distributed implementations of metaheuristics and their associated performance measures. The robustness of a method refers to its ability to perform well on a wide variety of input instances of a problem class and/or on different problems. Concerning parallel and distributed metaheuristics, it is too vast a subject to be tackled here. The reader wishing to pursue the study of the issues presented in this chapter is referred to the specialized literature, e.g., [41, 13] for details and extensions of performance evaluation and statistical analysis, and [78, 32] for parallel and distributed implementations and their performance.

11.2 The "No Free Lunch" Theorems and Their Consequences

We hope that the reader is convinced at this point that metaheuristics, without being a silver bullet, do nevertheless provide in practice a flexible, general, and relatively easy approach to hard optimization problems. Metaheuristics work through some kind of "intelligent" sampling of the solution space, both for methods in which the search follows a trajectory in space, such as simulated annealing, as well as for population-based methods such as evolutionary algorithms or particle swarms, for example.

In the previous sections of this chapter we discussed a number of approaches for evaluating the performance of a metaheuristic on a given problem or class of problems, and for comparing the effectiveness of different algorithms on a problem or a set of problems. Especially when comparisons between metaheuristics are called for, a number of questions arise naturally. Can we really compare the performance of different metaheuristics on a problem class? What if we include different problem types or problem classes in the comparison? Is there a principled way of choosing the best-adapted metaheuristic on a given problem? All these questions are legitimate because metaheuristics are general problem-solving methods that can be applied to many different problems, not specialized exact algorithms such as those used for sorting or searching. Thus, it has very frequently been the case in the literature that different metaheuristics or differently parameterized versions of the same metaheuristic are compared using a given suitable set of test functions. As we explained in Section 11.1.1, there are several well-known sets of test functions that typically contain a few tens of functions chosen according to various important criteria.

On the basis of numerical experiments, often using only a handful of test functions, one can obtain performance measures of two or more metaheuristics by following the methodology explained in the previous sections. Often the authors of such studies quickly extrapolate the results to unseen cases and sometimes affirm the superiority of one search method over others. Implicitly, their conclusion is that if method $A(f_i)$ applied to instance f_i of a given test function has a better performance

than method $B(f_i)$ on the same instance, and the result is the same for all or the majority of n test functions $\{f_1, f_2, \ldots, f_n\}$, with n typically between 5 and 10, then the result is probably generalizable to many other cases. But experience shows that one can reach different conclusions according to the particular metaheuristics used, their parameterization, and the details of the test functions. In this way, rather sterile discussions on the superiority of this or that method have often appeared in the literature. However, in 1997 Wolpert and Macready's work [85] on "no free lunch theorems" (NFL) showed that, under certain general conditions, it is impossible to design a "best" general optimization method.

In Wolpert and Macready's article the colloquial expression "no free lunch," which means that nothing can be acquired without a corresponding effort or cost, is employed to express the fact that no metaheuristic can perform better than another on all possible problems. More precisely, here is how the ideas contained in the NFL theorems might be enunciated in a nutshell:

For all performance measures, no algorithm is better than another when they are compared on all possible discrete functions.

Or, equivalently:

The average behavior of any two search methods on all possible discrete functions is identical.

The latter formulation implies that if method A is better than method B on a set of problems, then there must exist another set of problems on which B outperforms A. In particular, and perhaps surprisingly, on the finite set of discrete problems, random search has the same average performance as any more sophisticated metaheuristic. For example, let's consider a deterministic local search such as best improvement (see Chapter 2). For reasons that will become clear below, let's assume that the search can restart from an arbitrary configuration when it reaches a local optimum. Such a metaheuristic, though simple, should provide at least reasonably good results on many conceivable functions. Let's consider now a local search that always chooses a random neighbor along its trajectory in the search space. Perhaps contrary to intuition, although on many functions best improvement would perform better, there must be other functions on which the random walk search outperforms hill climbing since both must have the same performance on average. It goes without saying that many of the functions that would favor random walk search are essentially random functions, or functions of a particular nature which are not important in problems that present themselves in real applications. However, those functions do exist and they influence the results from a statistical point of view.

We now summarize the theoretical framework under which the NFL theorems have been established without going into too much mathematical detail. The interested reader will find the full discussion in the original work [85].

- The theory considers search spaces $\mathbf{S}$ of size $|\mathbf{S}|$, which can be very large but always finite. This restricts the context to combinatorial optimization problems (see Chapters 1 and 2). On these spaces the objective functions $f : \mathbf{S} \rightarrow Y$ are defined, with Y being a finite set. Then the space $F = Y^{\mathbf{S}}$ contains all possible functions and has size $|Y|^{|\mathbf{S}|}$, which is in general very large but still finite.
- The point of view adopted is that of *black box optimization*, which means that the algorithm has no knowledge of the problem apart from the fact that, given any candidate solution, it can obtain the objective function value of that solution. Wolpert and Macready use the number of function evaluations as the basic measure of the performance of a given search method. In addition, to avoid unbounded growth of this number, only a finite number m of *distinct* function evaluations is taken into account. That is to say the search space points are never resampled. The preceding scenario can easily be applied to all common metaheuristics provided we ignore the possibly resampled points.

Under the previous rather mild and general conditions, Wolpert and Macready establish the following result by using probability and information theory techniques:

$$\sum_f P(d_m^y|f, m, A_1) = \sum_f P(d_m^y|f, m, A_2)$$

In the previous expression $P(d_m^y|f, m, a)$ is the conditional probability of obtaining a given sample d_m^y of size m from function f, corresponding to the sampled search space points d_m^x, when algorithm A is iterated m times. Summations are performed on all functions $f \in F$ and A_1 and A_2 are two particular search algorithms. In other words, the expression means that the probability of generating a particular sequence of function values is the same for all algorithms when it is averaged over all functions, or, equivalently, that $P(d_m^y|f, m, A)$ is independent of algorithm A when the probability is averaged over all objective functions f.

Now, if $\Phi(d_m^y)$ is any sampling-based performance measure, the average of $\Phi(d_m^y)$ is independent of A as well:

$$\sum_f \Phi(d_m^y|f, m, A_1) = \sum_f \Phi(d_m^y|f, m, A_2)$$

meaning that no algorithm can outperform any other algorithm when their performance is averaged over all possible functions. Wolpert and Macready show that the results are also valid for all sampling-based performance measures $\Phi(d_m^y)$, and that they also apply to stochastic algorithms and to time-varying objective functions.

What are the lessons to be learned from the NFL theorems? The most important positive contribution is that the theorems imply that it cannot be said any longer that algorithm A is better than algorithm B without also specifying the class of problems for which this is true. This means that no search method that is based on sampling and without specific knowledge of the search space can claim to be superior to any other in general. It is also apparent that performance results claimed for a given benchmark

suite or for specific problems do not necessarily translate into similar performance on other problems. As we saw above, this behavior is the result of the existence of a majority of random functions in the set of all possible functions. However, the interesting functions in practice are not random, which means that the common metaheuristics will in general be more effective on the problems researchers are normally confronted with.

Moreover, the NFL theorems hold in the black box scenario only. If the user possesses problem knowledge beyond that this knowledge can, and should, be used in the search algorithm in order to make it more efficient. This is what often happens in real applications such as scheduling or assignment in which problem knowledge is put to good use in the algorithms to solve them. Thus, the conclusions reached in the NFL theorems are not likely to stop the search for better metaheuristics, but at least we now know that some discipline and self-restraint must be observed in analyzing and transferring performance results based on a limited number of problems.

The above results trigger a few considerations that are mainly of interest for the practitioner. Since it is impossible to prove the superiority of a particular metaheuristic in the absence of specific problem knowledge, why not use simpler and easy to implement metaheuristics first when tackling a new problem? This approach will save time and does not prevent one from switching to a more sophisticated method if the need arises.

In the same vein, we now briefly describe a useful and relatively new approach to problem solving that somehow exploits the fact that different algorithms perform better on different groups of functions, and also assumes a context similar to the black box scenario. In this case, since we do not know how to choose a suitable algorithm A_i in a small set $\{A_1, A_2, \ldots, A_k\}$ the idea is to use all of them. This leads to the idea of an *algorithm portfolio*, in which several algorithms are combined into a portfolio and executed sequentially or in parallel to solve a given difficult problem. In certain cases, the portfolio approach may be more advantageous than the traditional method. The idea comes from the field of randomized algorithms but it is useful in general [42]. Another way of implementing the approach is to select and activate the algorithms in the portfolio dynamically during the search, perhaps as a consequence of some statistical measures of the search space that are generated on the fly during the search. Clearly, the portfolio composition as well as the decision of which algorithm to use at which time are themselves difficult problems but some ideas have been proposed to make these choices automatic or semi-automatic. A deeper description of this interesting approach would lead us beyond our scope in this book and we refer the reader to the specialized literature for further details.

12

Statistical Analysis of Search Spaces

12.1 Fine Structure of a Search Space

In contrast to the classical theoretical computational complexity point of view summarized in Chapter 1 according to which a given problem belongs to a certain complexity class, the common practice in the metaheuristics community is to consider the specific search space of a given problem instance or class of problem instances (see Chapter 2). This is natural to the extent that metaheuristics can be seen as clever techniques that exploit the search space structure of a problem instance in order to find a quality solution in reasonable time. And it is not in contradiction with the fact that a problem may be classified as being hard in general as, in practice, not all instances will be equally difficult to solve, as we have learned in the chapter on phase transitions in computational hardness, where we have seen that intractable problems may possess easy-to-solve instances under some conditions. It therefore becomes important in the field of metaheuristics to be able to build tools that allow us to obtain quantitative measures of the main features of a search space. This analysis has a double value: it may provide indirect quantitative information about the problem difficulty, and it can be helpful, at least in principle, for improving the efficiency of the metaheuristics used to solve the problem. The subject is relatively advanced but we have decided to provide at least a glimpse of it for it helps to understand the factors that may affect the behavior of a metaheuristic. The prerequisite to benefit from reading the present chapter is a good understanding of the ideas presented in Chapter 2. For more detailed treatments, the reader is referred to the following books [41, 78, 73].

We briefly remind the reader that the search space of a problem[1] (see Chapter 2) is essentially constituted by the set of admissible solutions to the problem, a move

[1] To simplify the writing, the term "problem" will be informally interpreted as referring to a particular instance of a problem or to the general problem itself depending on the context. In the same way, the terms "search space" and "fitness landscape" will sometimes be used interchangeably.

© Springer Nature Switzerland AG 2018

B. Chopard, M. Tomassini, *An Introduction to Metaheuristics for Optimization*,
Natural Computing Series, https://doi.org/10.1007/978-3-319-93073-2_12

operator that generates a neighborhood for any solution, and a fitness measure that associates a generally real value to each solution.

There exist a number of useful quantities that can be computed from a knowledge of the search space. Without pretending to be exhaustive, we shall now describe some of the most important and commonly used. It is important to point out that all these measures rely on sampling the search space since the size of the latter grows exponentially with problem size and quickly becomes impossible to enumerate. Clearly, in those cases, if we could enumerate all the points, then we could also directly solve the problem. Let's start with indicators that globally characterize the fitness landscape.

12.2 Global Measures

Two important global features of a search space are the *number of local optima* and the associated *size of the basins of attraction*. For example, Table 12.1 shows the number of optima for randomly generated instances of NK problems, which were introduced in Chapter 2, with $N = 18$ and increasing values of $K < N$. Given the relatively small size of the problem, it has been possible to exhaustively enumerate all the optima in this particular case and thus the table entries are exact in this sense, although they represent an average over 30 randomly drawn instances for each K because of the statistical nature of NK landscapes. In general, complete enumeration is out of the question for computational reasons except for small problems, which means that this statistic must normally be computed on a suitable sample of the whole search space.

Table 12.1. Average number $\bar{n}$ of optima of NK landscapes with $N = 18$ as a function of K. Values are averages over 30 randomly generated instances for each value of K. Standard deviations in parentheses

	$N = 18$
K	$\bar{n}$
2	50(25)
4	330(72)
6	994(73)
8	2,093(70)
10	3,619(61)
12	5,657(59)
14	8,352(60)
16	11,797(63)
17	13,795(77)

We remark that the number of optima rapidly increases with K, making the landscape more and more rugged and difficult to search, a well-known result [47]. Some

researchers have suggested that in the search spaces of hard problems the number of optima increases exponentially with the problem size, a hypothesis that has been empirically verified in many cases but for which no rigorous theoretical background is available yet. Anyway, it seems likely that a large number of optima constitutes a hindrance in searching for the global one.

What is the shape of the attraction basins related to these optima? We recall that an attraction basin is the set of solutions such that executing strict hill climbing from any of them will lead the search to end in the same local optimum. Figure 12.1 shows the complementary empirical cumulative distribution of the number of basins having a given size for NK landscapes with the same N and K as in Table 12.1. Note the logarithmic scale of the axes.

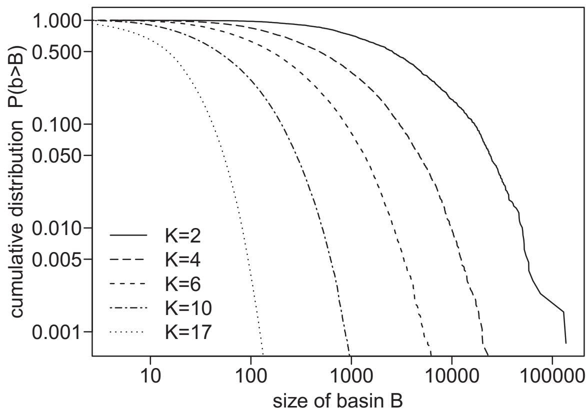


Fig. 12.1. Cumulative complementary distribution of basin size in NK landscapes for $N = 18$ and different values of K. Curves represent averages over 30 independently generated random instances for each K

The figure shows that the number of basins having a given basin size decreases exponentially and the drop is larger for higher K, in other words the size of basins decreases when K increases. This is in agreement with what we know about these search spaces: there are many more basins for large K but they are smaller. For low K the basins are more extended and this too is coherent with the intuitions we have developed about the difficulty of searching these landscapes.

12.3 Networks of Optima

Another recently introduced global description of a problem's search space is *networks of optima* [82]. Optima networks arise from a particular vision of a search space. Instead of considering all the solutions $s \in \mathbf{S}$, this approach consists of extracting either all the optima by an exhaustive search if the search space size allows

it, or a representative fraction of them by a sampling process if this is not possible. At the same time, connections among the optima are also determined. In the original formulation of the optima network graph there is an arc e_{ij} between two optima i and j if there is at least one transition from the attraction basin of i to the basin of attraction of j; if this is the case, the two basins communicate. The arc e_{ij} will have a normalized weight w_{ij} which is proportional to the number of transitions, the weight representing then a probability of transition from the basin of i to the basin of j. The arcs are oriented and need not be symmetric, i.e., it can be that $w_{ij} \neq w_{ji}$. This construction gives rise to an oriented weighted graph $G(V, E(w))$ in which V is the set of optima and E is the set of links (transitions) between the optima. G is a complex network that can be studied with the tools that have been developed in the last fifteen years in this field [11]. Among the features that can be studied we mention the number of vertices, the number of links, the average length of shortest paths, the graph diameter, the clustering coefficient, and several others. The idea is that these graph statistics might provide interesting information on the search space and its difficulty. To illustrate the idea, Figure 12.2 is a graphic representation of the optima and the associated transitions between optima basins for a relatively small instance of an NK landscape.

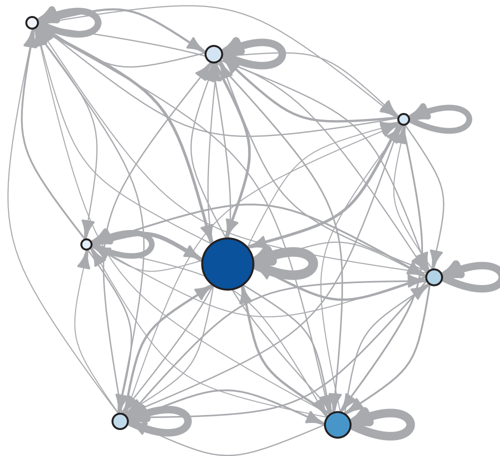


Fig. 12.2. An optima network for an NK instance with $N = 18$ and $K = 2$. The size of nodes is proportional to the size of the corresponding basins of attraction and the thickness of the oriented links is proportional to the probability of the transitions. We can see that most transitions are intra-basin

The concept of the graph of local optima and their transitions, although recent, has already been useful in studying the relation between problem classes and the difficulty of their search spaces. For example, it has been remarked that the correlation between the average distances from the local optima to the global one is a good indicator of problem hardness: the longer the distances, the harder the problem is. The transition frequency between optima (i.e., between basins thereof) also provides useful information: a high probability of transition out of a basin indicates that the problem is easier since a local search is more likely to be able to jump out of it. On the other hand, if most transitions are intra-basin and only a few lead outside the search will be more difficult. Similarly, a weak incoming transition probability indicates that a basin is difficult to reach. Another useful statistic that can be computed on these networks is the fitness correlation between adjacent optima: a large positive value suggests that the problem is likely to be easy. Here it has only been possible to offer a glimpse of the usefulness of a complex network view in understanding the nature of search spaces. A deeper discussion of the subject would lead us too far away but the reader is referred to [73], a collective book that probably contains the most up-to-date discussion on fitness landscapes in general.

12.4 Local Measures

In the previous sections some global quantities characterizing a fitness landscape have been presented. Here we shall take a more local view of a landscape, similarly to the way in which many metaheuristics work, e.g., by testing solutions in the landscape and using the values found in previous iterations to make choices influencing the future of the search process. A simple example of this is straight hill climbing. The two measures we are going to discuss are *fitness-distance correlation* and *fitness autocorrelation*, both of which have proven useful in understanding how local features of the landscape give information about the problem difficulty and how they influence the search. Our description will stick to the basics; the interested reader will find more information in, e.g., [41].

12.4.1 Fitness-Distance Correlation

This measure has been proposed by T. Jones [45] who conceived it mainly as a way of classifying the difficulty of landscapes for search. It is based on the intuition that there should be a negative correlation between the fitness value of a solution and the distance of the solution from the global optimum in the case of maximization, and a positive one for minimization. In other words, it assumes that if we move towards the maximum in a search, then the shorter the distance to the maximum, the higher the fitness should be. A mono-modal function is a trivial example in which, effectively, the function value decreases (increases) if we move away from the global maximum (minimum). To compute the Fitness-Distance Correlation (*FDC*) one has to sample a sufficient number n of solutions s in the $\mathbf{S}$ space. For each sampled value in the series $\{s_1, s_2, \ldots, s_n\}$ we must compute the associated fitness values $F =$

$\{f(s_1), f(s_2), \ldots, f(s_n)\}$, and their distances $D = \{d_1, d_2, \ldots, d_n\}$ to the global optimum, which is assumed to be known. The *FDC* is a number between -1 and 1 given by the following expression, which is just the standard Pearson correlation coefficient:

$$FDC = \frac{C_{FD}}{\sigma_F \sigma_D}$$

where

$$C_{FD} = \frac{1}{n}\sum_{i=1}^{n}(f_i - \langle f \rangle)(d_i - \langle d \rangle)$$

is the covariance of F and D, and $\langle f \rangle$, $\langle d \rangle$, σ_F, σ_D are their averages and standard deviations respectively.

It will not have escaped the attentive reader that there is a problematic aspect in the *FDC* calculation: how can we compute the distances to the global optimum given that the latter is exactly what we would like to find? The objection is a valid one but, when the global optimum is unknown, it may still be useful to replace it with the best known solution. Thus, according to the *FDC* value, Jones proposed the following classification of problems (assuming maximization):

- *Misleading* ($FDC \geq 0.15$), fitness increases when the distance to the optimum increases.
- *Difficult* ($-0.15 < FDC < 0.15$), there is no or little correlation between fitness and distance.
- *Straightforward* ($FDC \leq -0.15$), fitness increases when the distance to the optimum decreases.

According to Jones and other authors, *FDC* is a rather good indicator of the difficulty of a problem as seen by a metaheuristic based on local search and even for more complex population-based methods, such as evolutionary algorithms using crossover, which is more puzzling. In any case, we should not forget that *FDC*, being an index obtained by a sampling process, is subject to sampling errors. Indeed, it has been shown that it is possible to contrive problems that exploit these weaknesses and that make *FDC* malfunction, giving incorrect results [5]. However, when *FDC* is applied to "naturally" occurring problems it seems that the results are satisfactorily reasonable if one takes its limitations into account.

The following example is taken from [81], in which *FDC* is employed in genetic programming. The studied function is analogous to the *trap* function defined in Chapter 8 (see Figure 8.7) except that, instead of binary strings, individuals are coded as trees and unitation is replaced by a suitable notion of distance in trees. Before showing the results of the *FDC* sampling, it is necessary to define a difficulty measure in order to compare the prediction with the observed behavior. The empirical measure used here is simply the fraction of times that the global optimum has been found in 100 executions for each pair of values of the parameters of the trap function (see also Chapter 11).

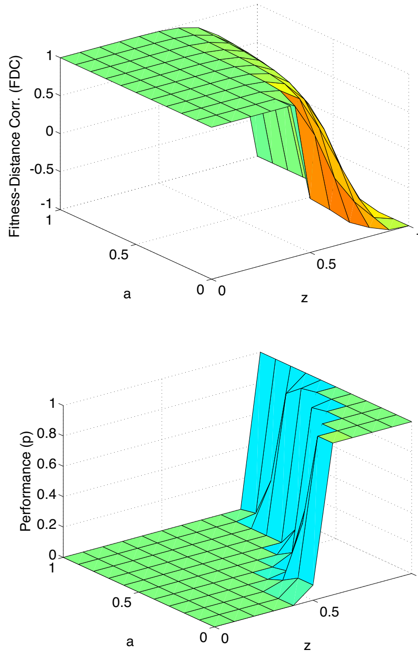


Fig. 12.3. Left: fitness-distance correlation values for a set of trap functions with parameters a and z, compared with genetic programming performance on the same functions with mutation only (right)

The images of Figure 12.3 clearly indicate that there is a very good agreement between difficulty as measured by performance and the *FDC* for a given trap, as a function of the a and z parameters of the latter. Indeed, in regions where performance is almost zero *FDC* approaches one, while it becomes negative or does not give indications for the "easy" traps. It is important to point out that the above results have been obtained with a special kind of genetic programming without the crossover operator and with a particular mutation operator that preserves certain properties in relation to tree distance in the solution space. When crossover is introduced, the agreement between *FDC* and performance is less good but the correlation still goes in the right direction.

A more qualitative and descriptive method than computing the *FDC* consists of drawing a scatterplot of sampled fitness values against the distance to the global optimum if it is known, or to the best solution found. To illustrate the point, let us use once more the NK landscapes.

Figure 12.4 shows the scatterplots for two cases: $K = 0$ (left image) and $K = 17$ (right image) for $N = 18$. Even without computing a regression straight line, it is visually clear that the case $K = 0$, which is easy, gives rise to a negative correlation,

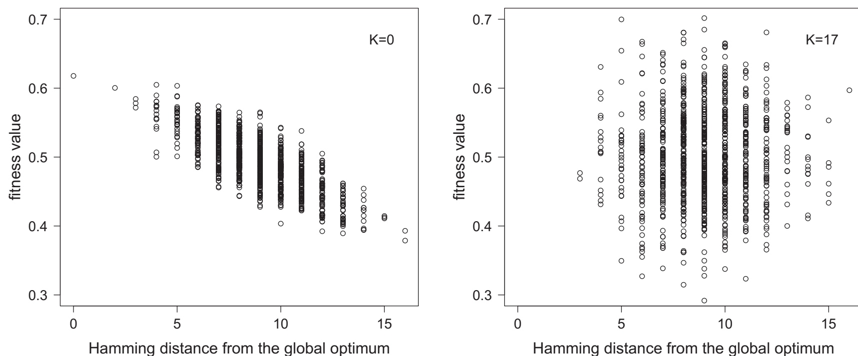


Fig. 12.4. Scatterplots of fitness/distance for two landscapes NK with $N = 18$ and $K = 0$ (left) and $K = 17$ (right). Samples are of size 1,000 and have been chosen uniformly at random in the corresponding search spaces

while the much more difficult case $K = 17$, which corresponds to an almost random and maximally rugged landscape, shows little or no correlation between fitness and distance to the global optimum. It must be said that things are seldom so clear-cut and one usually needs to compute several statistics on the search space to better understand its nature.

12.4.2 Random Walks and Fitness Autocorrelation

As we have seen in Chapter 3, random walks are not an efficient method for searching a fitness landscape. However, they can be used to collect interesting information about the search space. Let's choose a starting solution s_0 in $\mathbf{S}$, then a random walk of length l is a sequence $\{s_0, s_1, \ldots, s_l\}$ such that $s_i \in V(s_{i-1}), (i = 1, \ldots, l)$, where s_i is chosen with uniform probability in the neighborhood $V(s_{i-1})$. This kind of random walk may provide useful information about the search space and it is the basis of several single-trajectory metaheuristics such as simulated annealing. Random walks give better results as a sampling method if the search space is isotropic, i.e., if it has the same properties in all directions. Starting from such a random walk, one can compute the *fitness autocorrelation function* along the walk. The autocorrelation function is defined as follows [84]:

$$\rho(d) = \frac{\langle (f(s_t) - \langle f(s_t) \rangle)(f(s_{t+d}) - \langle f(s_t) \rangle) \rangle}{\langle f(s_t)^2 \rangle - \langle f(s_t) \rangle^2}$$

in which $f(s_t)$ is the value of the fitness of solution s sampled at step t, $f(s_{t+d})$ is another fitness values shifted by a distance d from s, $\langle . \rangle$ is the expected-value operator, and the denominator is the variance of the process. Thus, for instance, $\rho(1)$ only takes into account a solution's fitness and the fitness values of solutions at distance one. The autocorrelation function normalized by the variance falls between -1 and

1. To compute $\rho(d)$, the averages must be calculated for all points in $\mathbf{S}$ and for all pairs of points at distance d, which is too demanding except for small search spaces. In practice, the autocorrelation function can be approximated by the quantity $r(d)$, which is computed on a sample of length l obtained by performing a random walk:

$$r(d) = \frac{\sum_{t=1}^{l-d} (f(s_t) - \langle f \rangle)(f(s_{t+d}) - \langle f \rangle)}{\sum_{t=1}^{l} (f(s_t) - \langle f \rangle)^2}$$

We have seen that the difficulty of a problem is often associated with the ruggedness of the corresponding search space. The intuition is that in a "smooth" landscape fitness changes moderately when going from one solution to a neighboring one.

In the limiting case of a "flat" or almost flat landscape we speak of a *neutral* landscape [72]. Neutrality is a widespread phenomenon in difficult fitness landscapes such as those generated by hard instances of many combinatorial optimization problems. For example, a high degree of neutrality has been found in the search spaces of problems such as SAT and graph coloring. The presence of large neutral regions in the landscape, also called *neutral networks*, has a remarkable influence on search algorithms. Thus, hill climbers cannot exploit the fitness "gradient" and the search becomes a time-consuming random drift with, from time to time, the discovery of a better solution that allows the search to extract itself from the neutral region. There exist search techniques more adapted to neutral landscapes. For details, we refer the reader to the relevant literature, e.g., [24, 80].

On the contrary, a landscape in which fitness variations are abrupt at a short distance will be defined as being "rugged." In general terms, a rugged search space will be more difficult to explore because the numerous wells and peaks will easily cause the search to get stuck at local optima[2]. By the way, the terms "wells," "peaks," and the whole metaphor of a landscape being a kind of topographic object similar to a mountain chain only makes sense for three-dimensional continuous functions. It is almost impossible to picture, or even to imagine, for higher dimensions. And it is wrong for discrete combinatorial spaces, where there are only isolated points (solutions) and their fitnesses, which might be depicted as bars "sticking out" from solutions with heights proportional to the solution's fitness. Nevertheless, the metaphor of a continuous landscape with peaks, wells, and valleys continues to be a useful one if it is taken with a grain of salt.

Returning to the fitness autocorrelation function, we can say that it is a quantitative tool for characterizing the amount of ruggedness of a landscape and therefore, at least indirectly, for obtaining information about the difficulty of a problem. As an example, let's again use NK landscapes. Figure 12.5 shows the empirical fitness autocorrelation coefficient computed on $1{,}000$ steps of a random walk on NK landscapes with $N = 18$ and several values of K. It is apparent that for $K = 0$ the fitness autocorrelation decreases slowly with distance since the landscape is smooth. For small values of K the behavior is similar but as K increases the autocorrelation

[2] Be aware, however, that a flat landscape with a single narrow peak is also very difficult to search (or a "golf course" situation for minimization).

drops more quickly and for $K = 17$, where the landscape is random, ruggedness is maximal and the autocorrelation drops to zero even for solutions at distance one.

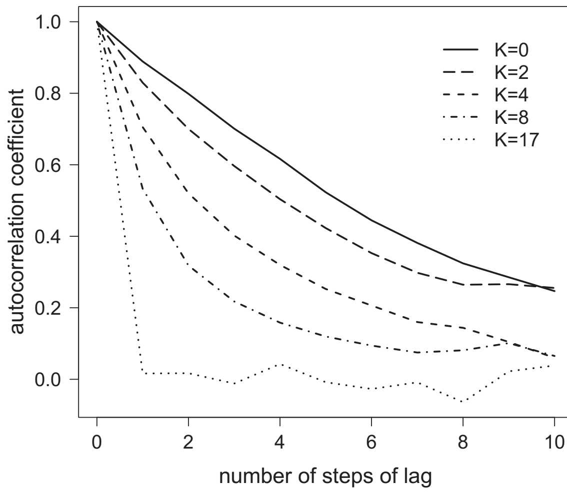


Fig. 12.5. Fitness autocorrelation coefficient as a function of distance for NK landscapes with $N = 18$ and variable K

Appendices

© Springer Nature Switzerland AG 2018

B. Chopard, M. Tomassini, *An Introduction to Metaheuristics for Optimization*,
Natural Computing Series, https://doi.org/10.1007/978-3-319-93073-2

Tools and Recommended Software

The following lists a few useful websites pertaining to the metaheuristics referenced in this book. At the time of writing, these sites are active and maintained.

- ECJ is a Java environment for evolutionary programming that comprises the main metaheuristics, including genetic programming and the parallelization of the algorithms
  https://cs.gmu.edu/~eclab/projects/ecj/
- Optimization with particle swarms (PSO):
  http://www.particleswarm.info/Programs.html
  is a website that contains information about open PSO software
- CMA-ES: Modern evolution strategies. It is maintained by N. Hansen
  http://cma.gforge.inria.fr/cmaes_sourcecode_page.html
- Open Beagle: an integrated system for genetic programming
  https://github.com/chgagne/beagle
- ACO Iridia: a software system providing programs that use ant colony strategies on different types of problems
  http://iridia.ulb.ac.be/~mdorigo/ACO/aco-code
- METSlib is an open, public domain optimization tool written in C++. It includes tabu search, various forms of local search, and simulated annealing
  https://projects.coin-or.org/metslib
- Paradiseo is a complete software system that contains all the main metaheuristics, including evolutionary computation and multi-objective optimization; parallelization tools are also provided
  http://paradiseo.gforge.inria.fr

References

1. A. Abraham, H. Guo, and H. Liu. Swarm intelligence: Foundations, perspectives and applications, swarm intelligent systems. In N. Nedjah and L. Mourelle, editors, *Studies in Computational Intelligence*, pages 3–25. Springer Verlag, 2006.
2. D. Achlioptas and M. Molloy. The solution space geometry of random linear equations. *Random Structures and Algorithms*, 46:197–231, 2015. arXiv:1107.5550.
3. F. Ahammed and P. Moscato. Evolving L-Systems as an intelligent design approach to find classes of difficult-to-solve traveling salesman problem instances. In Di Chio C. et al. (eds), editor, *Applications of Evolutionary Computation*, volume 6624 of *LNCS*. Springer, 2011.
4. E. Alba and B. Dorronsoro. *Cellular genetic algorithms*. Springer, Berlin, 2009.
5. L. Altenberg. Fitness distance correlation analysis: an instructive counterexemple. In T. Bäck, editor, *Seventh International Conference on Genetic Algorithms*, pages 57–64. Morgan Kaufmann, 1997.
6. D. Applegate, R. Bixby, V. Chvátal, and W. J. Cook. On the solution of traveling salesman problems. *Documenta Mathematica*, Extra Volume ICM III:645–656, 1998.
7. D. L. Applegate, R. E. Bixby, V. Chvàtal, and W. J. Cook. *The Traveling Salesman Problem*. Princeton University Press, 2007. ISBN-10: 0-691-12993-2.
8. A. Auger and T. B. Doerr (Eds.). *Theory of Randomized Search Heuristics*. World Scientific, Singapore, 2011.
9. T. Bäck. *Evolutionary Algorithms in Theory and Practice*. Oxford University Press, Oxford, 1996.
10. W. Banzhaf, P. Nordin, R. E. Keller, and F. D. Francone. *Genetic programming, An Introduction*. Morgan Kaufmann, San Francisco, CA, 1998.
11. A.-L. Barábasi. *Network Science*. Cambridge University Press, Cambridge, UK, 2016.
12. A. Baronchelli and F. Radicchi. Lévy flights in human behavior and cognition. *Chaos, Solitons, and Fractals*, 56:101–105, 2013.
13. T. Bartz-Beielstein, M. Chiarandini, L. Paquete, and M. Preuss (Eds.). *Experimental Methods for the Analysis of Optimization Algorithms*. Springer, Berlin, 2010.
14. D. Bertsekas and J. N. Tsitsiklis. *Parallel and Distributed Computation*. Prentice-Hall, Englewood Cliffs, NJ, 1989.
15. N. Boccara. *Modeling Complex Systems*. Springer, Berlin, 2010.
16. B. Bollobaś. *Modern Graph Theory*. Springer, Berlin, 1998.
17. A. Braunstein, M. Leone, F. Ricci-Tersenghi, and R. Zecchina. Complexity transitions in global algorithms for sparse linear systems over finite fields. *J. of Physics A: Mathematical and General*, 35(35):7559–7574, 2002. arXiv:cond-mat/0203613.

© Springer Nature Switzerland AG 2018

B. Chopard, M. Tomassini, *An Introduction to Metaheuristics for Optimization*,
Natural Computing Series, https://doi.org/10.1007/978-3-319-93073-2

18. E. Cartlidge. Quantum computing: How close are we? *Optics and Photonics News*, pages 32–37, 2016.
19. B. Chopard, P. Albuquerque, and M. Martin. Formation of an ant cemetery: Swarm intelligence or statistical accident? *FGCS*, 18:951–959, 2002.
20. B. Chopard, Y. Baggi, P. Luthi, and J.F. Wagen. Wave propagation and optimal antenna layout using a genetic algorithm. *Speedup*, 11(2):42–47, November 1997. TelePar Conference, EPFL, 1997.
21. B. Chopard, O. Pictet, and M. Tomassini. Parallel and distributed evolutionary computation for financial applications. *Parallel Algorithms and Applications*, 15:15–36, 2000.
22. M. Clerc. *Particle Swarm Optimization*. Wiley-ISTE, London, UK, 2006.
23. A. Coja-Oghlan, A. Haqshenas, and S. Hetterich. WALKSAT stalls well below the satisfiability threshold. Technical report, arXiv:1608.00346, 2016.
24. P. Collard, S. Vérel, and M. Clergue. How to use the scuba diving metaphor to solve problem with neutrality? In R. L. de Mantaras and L. Saitta, editors, *ECAI 2004*, pages 166–170. IOS Press, 2004.
25. W. J. Cook. *In Pursuit of the Traveling Salesman*. Princeton University Press, Princeton, NJ, 2012.
26. T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, third edition, 2009.
27. C. Darwin. *On the Origin of Species*. John Murray, London, 1859.
28. K. Deb. *Multi-Objective Optimization using Evolutionary Algorithms*. J. Wiley, Chichester, 2001.
29. J. L. Deneubourg, S. Aron, S. Goss, and J. M. Pasteels. The self-organizing exploratory pattern of the argentine ant. *Journal of Insect Behavior*, 3(2):159–168, 1990.
30. M. Dorigo, V. Maniezzo, and A. Colorni. Ant system: optimization by a colony of cooperating agents. *Systems, Man, and Cybernetics, Part B: Cybernetics, IEEE Transactions on*, 26(1):29–41, 1996.
31. J. Dréo, A. Pétrowski, P. Siarry, and E. Taillard. *Métaheuristiques pour l'optimisation difficile*. Eyrolles, 2003.
32. E. Alba (Ed.). *Parallel Metaheuristics*. J. Wiley, Hoboken, NJ, 2005.
33. L. J. Fogel, A. J. Owens, and M. J. Walsh. *Artificial Intelligence through Simulated Evolution*. J. Wiley, New York, 1966.
34. M. R. Garey and D. S. Johnson. *Computers and Intractability*. Freeman, San Francisco, CA, 1979.
35. C. J. Geyer. Computing science and statistics: Proceedings of the 23rd symposium on the interface. *American Statistical Association, New York*, page 156, 1991.
36. F. Glover. Tabu search-Part I. *ORSA Journal on computing*, 1(3):190–206, 1989.
37. F. Glover and S. Hanafi. Finite convergence of Tabu search. In *MIC 2001 - 4th Metaheuristics International Conference*, pages 333–336, 2001.
38. S. Goss, S. Aron, J. L. Deneubourg, and J. M. Pasteels. Self-organized shortcuts in the argentine ant. *Naturwissenschaften*, 76(12):579–581, 1989.
39. A. J. G. Hey, editor. *Feynman and Computation*. Perseus Books, Reading, Massachusetts, 1999.
40. J. H. Holland. *Adaptation in Natural and Artificial Systems*. The University of Michigan Press, Ann Arbor, Michigan, 1975.
41. H. H. Hoos and T. Stützle. *Stochastic Local Search: Foundations and Applications*. Morgan Kaufmann, San Francisco, CA, 2005.
42. B. A. Huberman, R. L. Lukose, and T. Hogg. An economics approach to hard computational problems. *Science*, 275(5296):51–54, 1997.

43. M. Ibrahimi, Y. Kanoria, M. Kraning, and A. Montanari. The set of solutions of random XORSAT formulae. *Annals of Applied Probability*, 25(5):2743–2808, 2015. arXiv:1107.5377.
44. M. Jamil and X.-S. Yang. A literature survey of benchmark functions for global optimization problems. *Int. Journal of Mathematical Modelling and Numerical Optimization*, 4:150–194, 2013.
45. T. Jones. *Evolutionary Algorithms, Fitness Landscapes and Search*. PhD thesis, University of New Mexico, Albuquerque, 1995.
46. D. Karaboga and B. Akay. A comparative study of artificial bee colony algorithm. *Applied mathematics and computation*, 214(1):108–132, 2009.
47. S. A. Kauffman. *The Origins of Order*. Oxford University Press, New York, 1993.
48. http://www.keithschwarz.com/darts-dice-coins/.
49. J. Kennedy and R. C. Eberhart. *Swarm Intelligence*. Morgan Kaufmann, San Francisco, CA, 2001.
50. S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, 1983.
51. Donald E. Knuth. *Seminumerical Algorithms*, volume 2 of *The Art of Computer Programming*. Addison-Wesley, Reading, Massachusetts, third edition, 1998.
52. J. R. Koza. *Genetic Programming*. The MIT Press, Cambridge, Massachusetts, 1992.
53. J. R. Koza. *Genetic Programming II*. The MIT Press, Cambridge, Massachusetts, 1994.
54. J. R. Koza. Human-competitive results produced by genetic programming. *Genetic Programming and Evolvable Machines*, 11:251–284, 2010.
55. P. Lévy. *Théorie de l'addition des variables aléatoires*. Gauthier-Villars, Paris, 1937.
56. H. Liu, A. Abraham, and J. Zhang. A particle swarm approach to quadratic assignment problems. In A. Saad et al., editor, *Soft Computing in Industrial Applications*, volume 39, pages 213–222. Springer, Berlin, 2007.
57. H. R. Lourenço, O. C. Martin, and T. Stützle. Iterated local search. In F. Glover and G. A. Kochenberger, editors, *Handbook of Metaheuristics*, volume 57 of *International Series in Operations Research & Management Science*, pages 320–353. Springer US, 2003. http://arxiv.org/pdf/math/0102188.pdf.
58. S. Luke. *Essentials of Metaheuristics*. Lulu, available at http://cs.gmu.edu/ sean/book/metaheuristics, 2013. second edition.
59. M. Mezard, F. Ricci-Tersenghi, and R. Zecchina. Alternative solutions to diluted p-spin models and XORSAT problems. *J. Stat. Phys.*, 111:505, 2003. arXiv:cond-mat/0207140.
60. Z. Michalewicz. *Genetic Algorithms + Data Structures = Evolution Programs (Third Edition)*. Springer, Berlin, Heidelberg, 1996.
61. S. Mirjalili. The ant lion optimizer. *Advances in Engineering Software*, 83:80–98, 2015.
62. S. Mirjalili, S. M. Mirjalili, and A. Lewis. Grey wolf optimizer. *Advances in Engineering Software*, 69:46–61, 2014.
63. R. Monasson. Introduction to phase transitions in random optimization problems. Technical report, Summer School on Complex Systems, Les Houches, 2006.
64. C. Moore and S. Mertens. *The Nature of Computation*. Oxford University Press, Oxford, UK, 2011.
65. I. M. Oliver, D. J. Smith, and J. R. C. Holland. A study of permutation crossover operators on the travelling salesman problem. In *Proceedings of the Second International Conference on Genetic algorithms and Their Application*, pages 224–230, 1987.
66. C. H. Papadimitriou. *Computational Complexity*. Addison-Wesley, Reading, MA, 1994.
67. C. H. Papadimitriou and K. Steiglitz. *Combinatorial Optimization: Algorithms and Complexity*. Prentice-Hall, Englewood Cliffs, NJ, 1982.

68. K.E. Parsopoulos and M.N. Vrahatis. Recent approaches to global optimization problems through particle swarm optimization. *Natural Computing 1*, 1:235–306, 2002.
69. W. H. Press, S. A. Teukolski, W. T. Vetterling, and B. P. Flannery. *Numerical Recipes: The Art of Scientific Computing*. Cambridge University Press, 2007. third edition.
70. P. Prusinkiewicz and A. Lindenmayer. *The Algorithmic Beauty of Plants*. Springer-Verlag, 1990.
71. I. Rechenberg. *Evolutionsstrategie*. Frommann-Holzboog Verlag, Stuttgart, 1973.
72. C. M. Reidys and P. F. Stadler. Neutrality in fitness landscapes. *Applied Mathematics and Computation*, 117:321–350, 2001.
73. H. Richter and A. P. Engelbrecht (Eds.). *Recent Advances in the Theory and Application of Fitness Landscapes*. Series in Emergency, Complexity, and Computation. Springer, Berlin, 2014.
74. F. Rothlauf. *Design of Modern Heuristics*. Springer, Berlin, 2011.
75. J.F. Schute and A.A. Groenwold. A study of global optimization using particle swarms. *Journal of Global Optimization*, 31:93–108, 2005.
76. G. Semerjian and R. Monasson. Relaxation and metastability in a local search procedure for the random satisfiability problem. *Phys. Rev. E*, 67:066103, 2003.
77. E. Taillard. Private communication.
78. E. Talbi. *Metaheuristics: From Design to Implementation*. Wiley, Hoboken, NJ, 2009.
79. M. Tomassini. *Spatially Structured Evolutionary Algorithms*. Springer, Berlin, Heidelberg, 2005.
80. M. Tomassini. Lévy flights in neutral fitness landscapes. *Physica A*, 448:163–171, 2016.
81. M. Tomassini, L. Vanneschi, P. Collard, and M. Clergue. A study of fitness distance correlation as a difficulty measure in genetic programming. *Evolutionary Computation*, 13:213–239, 2005.
82. M. Tomassini, S. Vérel, and G. Ochoa. Complex networks analysis of combinatorial spaces: the NK landscape case. *Phys. Rev. E*, 6:066114, 2008.
83. http://elib.zib.de/pub/mp-testdata/tsp/tsplib/tsplib.html.
84. E. D. Weinberger. Correlated and uncorrelated fitness landscapes and how to tell the difference. *Biol. Cybern.*, 63:325–336, 1990.
85. D. Wolpert and W. G. Macready. No free lunch theorems for optimization. *IEEE Trans. Evol. Comp.*, 1(1):67–82, 1997.
86. X.-S. Yang. Firefly algorithm, Lévy flights and global optimization. In M. Bremer at al., editor, *Research and Development in Intelligent Systems XXVI*, pages 209–218. Springer, Berlin, 2010.
87. X.-S. Yang. *Nature-Inspired Metaheuristic Algorithms*. Luniver Press, 2010.
88. X.-S. Yang and S. Deb. Engineering optimisation by cuckoo search. *International Journal of Mathematical Modelling and Numerical Optimization*, 1:330–343, 2010.

Index

1-*XORSAT*, 176
2-XORSAT, 178
NK landscapes, 18, 206, 211, 212
NK problem, 196
NP-complete problems, 6
NP-hard problems, 8
QAP, 51
TSP, 20, 53, 62
k-XORSAT problem, 172
MaxOne problem, 25
MaxOne problem, 19
2-Opt move, 28, 66
3-XORSAT, 181

acceptance procedure, 33
ACS, 94
adjacency representation, 167
admissible solution, 16, 166
algorithm portfolio, 203
analytic takeover times, 145
annealing, simulated, 59
ant colony, 81
Ant Colony System, 92, 94
Ant System, 92
antlion optimization, metaheuristic, 107
ants, 81
ants trail, 82
approximation algorithms, 9
AS, 92
asymptotic complexity, 4
attraction basin, 33
attraction basin, size, 206
average case complexity, 3
average fitness, 121

backtracking, 24
backtracking algorithm for SAT, 186
basin of attraction, 33
bees, metaheuristic, 107
benchmark problems, 94, 192
binary hypercube, 26
binary tournament selection, 143
birthday problem, 177
black box optimization, 201
branch-and-bound, 24
Brownian motion, 104
brute force, 9
building-block hypothesis, 130

cellular evolutionary algorithms, 162
cellular populations, 158
central limit theorem, 103
class NP, 1
class P, 1
combinatorial optimization problem, 16
complete algorithm, 187
complex system, 81
complexity classes, 5
computational complexity, 1
computational effort, 193
computational phase transitions, 171
Concorde algorithm for *TSP*, 67, 96
conjunctive normal form, 171
constraint satisfaction, 171
continuous optimization problem, 16
convergence, 46
convergence in probability, 131

© Springer Nature Switzerland AG 2018

B. Chopard, M. Tomassini, *An Introduction to Metaheuristics for Optimization*,
Natural Computing Series, https://doi.org/10.1007/978-3-319-93073-2

convergence, simulated annealing, 69
convergence, tabu, 46
cooling schedule, 61
crossover, 117, 127, 129
crossover in genetic programming, 150
crossover in linear genetic programming, 154
crossover probability, 117
cuckoo metaheuristic, 109
cuckoo search, 109

deceptive function, 130
deceptive problem, 130
decision problems, 5
decision trees, 151
discrete optimization problem, 16
discrete recombination, 137
discrete-event simulation, 85
diversity, 121

efficient algorithm, 2
elementary operations, 2
elitism, 116
empirical performance measures, 198
empirical solution probability, 194
energy, 60
energy landscape, 24
evaporation factor, 93
evolution of schemas, 127
evolution strategies, 115, 132
  covariance matrix, 136
  mutation, 133
  parameters adaptation, 134
  recombination, 137
  rotation angles, 136
  theoretical results, 137
  two-membered, 132
evolutionary algorithms, 115
  generational, 118
  steady state, 118
evolutionary cycle, 117
evolutionary programming, 115

feasible solutions, 16
fine-grained evolutionary algorithms, 162
fireflies, 107
firefly method, 103
firefly, attractiveness, 107
fitness autocorrelation, 212
fitness evaluation in genetic programming, 150
fitness landscape, 24
fitness penalty, 166
fitness scaling, 141
fitness smoothing, 34
fitness, average, 121
fitness-distance correlation, 209
fitness-proportionate selection, 32, 117, 141, 145
function optimization, 123
fundamental search methods, 31

Gaussian elimination, 174
generational evolutionary algorithm, 118
genetic algorithms, 115, 196
  theory, 127
genetic operators, 139, 166
genetic programming, 147
  crossover, 150
  evaluation, 150
  function set, 148
  linear, 154
  mutation, 150
  representation, 148
  terminal set, 148
giant component, 180
global landscape measures, 206
global recombination, 137
graph connectedness, 5
graph layout problem, 67
grey wolf optimizer, 107
GSM antenna placement, 125
guiding parameters, 66

Hamiltonian cycle, 8
Hamiltonian tour of a graph, 6
heterarchy, 81
heuristic methods, 13
heuristics, 22
hill climbing, 32
hybrid evolutionary algorithms, 168
hyperplane, 128

integer linear programming, 11
intermediate recombination, 137
iterated local search, 33
iterative best improvement, 32
iterative first improvement, 32

knapsack problem, 11
Knuth Shuffle, 40

Lévy distributions, 103
Lévy flight, 38, 104, 110
Lévy walk, 104
Lagrange multipliers, 17
landscape
  neutral, 213
  rugged, 213
  smooth, 213
Las Vegas algorithm, 193
Lindenmayer systems, 96
linear genetic programming, 154
linear programming, 18
linear programming relaxation, 11
linear ranking, 142
list, tabu, 48
local genetic operators, 162
local landscape measures, 209
local optima, number, 206
local search, 24, 33
long-term memory, 43, 48

mathematical optimization, 16
MaxOne problem, 118
memory, long-term, 43
memory, short-term, 43
metaheuristics, 8, 13, 22
  antlion, 107
  bee colony, 107
  cuckoo search, 109
  firefly, 107
  grey wolf, 107
  performance comparison, 198
  performance measures, 191
Metropolis rule, 61
Monte Carlo simulation, 61
moves, 23, 26
multipopulations, 158, 159
mutation, 117, 127, 129
mutation in evolution strategies, 133
mutation in genetic programming, 150
mutation in linear genetic programming, 154
mutation probability, 117

needle in a haystack, 26
neighborhood, 23, 26
neighborhoods in QAP, 53
network of optima, 207
neutral networks, 213
neutrality, 213
no free lunch theorems, 200
non-admissible solution, 166
non-parametric tests, 198
non-tabu configuration, 43
non-uniform probability sampling, 37

optima network, 207
optimal foraging, 105
optimization, 7, 15
optimization problems, 7

panmictic populations, 158
parallel computing, 10
parallel tempering, 76
parallelism, 10
parameters adaptation in evolution strategies, 134
particle swarm optimization, 97
perceived attractiveness, fireflies, 107
performance, 91
performance measures, 193
performance of metaheuristics, 191
permutation, 20, 27
permutation space, 20
permutations, 166
perturbation procedure, 33
phase transitions, 171
pheromone, 82
polynomial reduction, 6
polynomial time algorithm, 5
population metaheuristics, 30
population, diversity, 121
premature convergence, 141
probabilistic hill climber, 32
problem difficulty, 1
proportionate selection, 32, 141
PSO, 97
PSO, global-best configuration, 97
PSO, particle-best configuration, 97

quadratic assignment problem, 51
quantum computation, 13
qubits, 13

random graph, 180
random permutations, 39

random search, 31
random walk, 31, 104, 212
Random walk SAT algorithm, 183
randomized algorithms, 12
randomized iterative improvement, 32
ranking selection, 142
real function optimization, 123
recombination, 137
representation, binary, 116
reproduction, 141
RWSAT, 194
RWSAT algorithm, 183

sampling errors, 141
sampling non-uniform probabilities, 37
SAT, 171
satisfiability problem, 171
satisfiable formula, 171
schema, 127
  cardinality, 128
  defining length, 128
  evolution, 127
  order, 128
search space, 15, 205
  statistics, 205
search stagnation, 197
selection, 117, 139
  deterministic, 133, 140
  fitness-proportionate, 141
  probabilistic, 133
  ranking, 142
  stochastic, 133, 140
  tournament, 143
selection in evolution strategies, 133
selection intensity, 144
selection methods, 140
selection pressure, 144, 165
self-organization, 81
short-term memory, 43, 48
simplex algorithm, 18
simulated annealing, 59
  practical guide, 74
simulated annealing and *TSP*, 194
simulated annealing convergence, 69
solution quality, 193
special cases of difficult problems, 9
specialized operators, 166
spin glasses, 18
statistics of search spaces, 205
steady-state evolutionary algorithm, 118
stigmergy, 82
stochastic universal sampling, 141
structured populations, 158
success probability, 91
success rate, 91, 193
super-polynomial time algorithm, 5
superindividual, 141
swarm intelligence, 81

tabu
  aspiration criteria, 50
  banning time, 48
  convergence, 46
  guiding parameters, 50
  memory, 48
tabu list, 43
tabu search, 43
tabu tenure, 48
takeover time, 144, 165
temperature schedule, 61
termination condition, 197
test problems, 192
theory of genetic algorithms, 127
tournament selection, 143, 147
track intensity, 93
trading models, 151
transformations, 23, 26
transposition, 27, 66
trap function, 131
traveling salesman, 20, 92, 96
traveling salesman problem, 8, 36, 53, 62, 166
tree crossover, 150
TSP, 92, 96
TSPLIB, 96

UNSAT, 184
unsatisfiable formula, 171
unsatisfiable problem, 184

variable neighborhood search, 34

well mixed populations, 158
worst case complexity, 3

XORSAT problem, 172