METAHEURISTICS

METAHEURISTICS

FROM DESIGN TO IMPLEMENTATION

El-Ghazali Talbi
University of Lille – CNRS – INRIA

WILEY
A JOHN WILEY & SONS, INC., PUBLICATION

Published by John Wiley & Sons, Inc., Hoboken, New Jersey
Published simultaneously in Canada.

Library of Congress Cataloging-in-Publication Data:

Talbi, El-Ghazali, 1965-
Metaheuristics : from design to implementation / El-ghazali Talbi.
p. cm.
Includes bibliographical references and index.
ISBN 978-0-470-27858-1 (cloth)
1. Mathematical optimization. 2. Heuristic programming. 3. Problem solving–Data processing. 4. Computer algorithms. I. Title.
QA402.5.T39 2009
519.6–dc22

2009017331

Printed in the United States of America

10 9 8 7 6 5 4 3 2 1

To my wife Keltoum, my daughter Besma, my parents and sisters.

CONTENTS

PREFACE

IMPORTANCE OF THIS BOOK

Applications of optimization are countless. Every process has a potential to be optimized. There is no company that is not involved in solving optimization problems. Indeed, many challenging applications in science and industry can be formulated as optimization problems. Optimization occurs in the minimization of time, cost, and risk or the maximization of profit, quality, and efficiency. For instance, there are many possible ways to design a network to optimize the cost and the quality of service; there are many ways to schedule a production to optimize the time; there are many ways to predict a 3D structure of a protein to optimize the potential energy, and so on.

A large number of real-life optimization problems in science, engineering, economics, and business are complex and difficult to solve. They cannot be solved in an exact manner within a reasonable amount of time. Using approximate algorithms is the main alternative to solve this class of problems.

Approximate algorithms can further be decomposed into two classes: specific heuristics and metaheuristics. Specific heuristics are problem dependent; they are designed and applicable to a particular problem. This book deals with metaheuristics that represent more general approximate algorithms applicable to a large variety of optimization problems. They can be tailored to solve any optimization problem. Metaheuristics solve instances of problems that are believed to be hard in general, by exploring the usually large solution search space of these instances. These algorithms achieve this by reducing the effective size of the space and by exploring that space efficiently. Metaheuristics serve three main purposes: solving problems faster, solving large problems, and obtaining robust algorithms. Moreover, they are simple to design and implement, and are very flexible.

Metaheuristics are a branch of optimization in computer science and applied mathematics that are related to algorithms and computational complexity theory. The past 20 years have witnessed the development of numerous metaheuristics in various communities that sit at the intersection of several fields, including artificial intelligence, computational intelligence, soft computing, mathematical programming, and operations research. Most of the metaheuristics mimic natural metaphors to solve complex optimization problems (e.g., evolution of species, annealing process, ant colony, particle swarm, immune system, bee colony, and wasp swarm).

Metaheuristics are more and more popular in different research areas and industries. One of the indicators of this situation is the huge number of sessions, workshops, and conferences dealing with the design and application of metaheuristics. For

example, in the biannual EMO conference on evolutionary multiobjective optimization, there one more or less 100 papers and 200 participants. This is a subset family of metaheuristics (evolutionary algorithms) applied to a subset class of problems (multiobjective problems)! In practice, metaheuristics are raising a large interest in diverse technologies, industries, and services since they proved to be efficient algorithms in solving a wide range of complex real-life optimization problems in different domains: logistics, bioinformatics and computational biology, engineering design, networking, environment, transportation, data mining, finance, business, and so on. For instance, companies are faced with an increasingly complex environment, an economic pressure, and customer demands. Optimization plays an important role in the imperative cost reduction and fast product development.

PURPOSE OF THIS BOOK

The main goal of this book is to provide a unified view of metaheuristics. It presents the main design questions and search components for all families of metaheuristics. Not only the design aspect of metaheuristics but also their implementation using a software framework are presented. This will encourage the reuse of both the design and the code of existing search components with a high level of transparency regarding the target applications and architectures.

The book provides a complete background that enables readers to design and implement powerful metaheuristics to solve complex optimization problems in a diverse range of application domains. Readers learn to solve large-scale problems quickly and efficiently. Numerous real-world examples of problems and solutions demonstrate how metaheuristics are applied in such fields as telecommunication, logistics and transportation, bioinformatics, engineering design, scheduling, and so on. In this book, the key search components of metaheuristics are considered as a toolbox for

- Designing efficient metaheuristics for optimization problems (e.g., combinatorial optimization, continuous optimization).
- Designing efficient metaheuristics for multiobjective optimization problems.
- Designing hybrid, parallel, and distributed metaheuristics.
- Implementing metaheuristics on sequential and parallel machines.

AUDIENCE

For a practicing engineer, a researcher, or a student, this book provides not only the materiel for all metaheuristics but also the guidance and practical tools for solving complex optimization problems.

One of the main audience of this book is **advanced undergraduate and graduate students** in computer science, operations research, applied mathematics, control,

business and management, engineering, and so on. Many undergraduate courses on optimization throughout the world would be interested in the contents thanks to the introductory part of the book and the additional information on Internet resources.

In addition, the **postgraduate** courses related to optimization and complex problem solving will be a direct target of the book. Metaheuristics are present in more and more postgraduate studies (computer science, business and management, mathematical programming, engineering, control, etc.).

The intended audience is also **researchers** in different disciplines. Researchers in computer science and operations research are developing new optimization algorithms. Many researchers in different application domains are also concerned with the use of metaheuristics to solve their problems.

Many **engineers** are also dealing with optimization in their problem solving. The purpose of the book is to help engineers to use metaheuristics for solving real-world optimization problems in various domains of application. The application part of the book will deal with many important and strategic domains such as computational biology, telecommunication, engineering design, data mining and machine learning, transportation and logistics, production systems, and so on.

The prerequisite knowledge the readers need to have is a basic background in algorithms. For the implementation part, basic background in programming with C++ will be a plus.

OUTLINE

The book is organized in the following six different chapters (Fig. P.1):

- **Common concepts for metaheuristics:** First, this chapter justifies the existence of the book. The main concepts of optimization models, complexity of

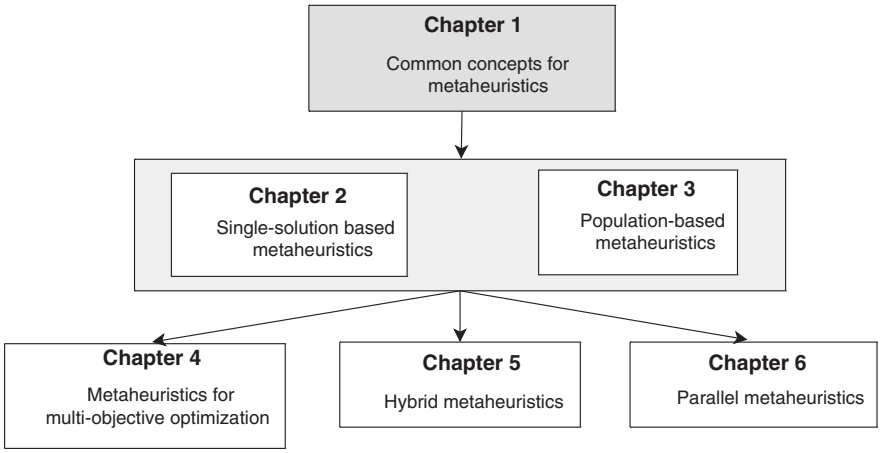

FIGURE P.1 Relationship between the different chapters of the book. The graph presents the dependencies between the chapters.

algorithms, and optimization methods are outlined. Then, the chapter exposes the common and basic concepts for metaheuristics (e.g., representation, objective function, and constraint handling). These concepts are used in designing *any* metaheuristic. The encoding (or representation) of a solution and its associated objective function are one of the most important features in metaheuristics. They will define the structure of the problem in which metaheuristics will search "good" solutions. Other important common issues of metaheuristics are detailed: performance evaluation and parameter tuning. Finally, the software engineering aspect dealing with frameworks for metaheuristics is presented.

- **Single-solution based metaheuristics:** In chapter 2, the focus is on the design and implementation of single-solution based metaheuristics such as local search, tabu search, simulated annealing, threshold accepting, variable neighborhood search, iterated local search, guided local search, GRASP, and so on. The common concepts of this class of metaheuristics are outlined (e.g., neighborhood, initial solution, and incremental evaluation). For each metaheuristic, the specific design and implementation of its search components are exposed. The parameters of each metaheuristic are analyzed and the relationship between different algorithms is addressed. Moreover, the convergence aspect of the introduced metaheuristics and many illustrative examples of their implementation are presented.
- **Population-based metaheuristics:** Chapter 3 concerns the design and implementation of population-based metaheuristics such as evolutionary algorithms (genetic algorithms, evolution strategies, genetic programming, evolutionary programming, estimation of distribution algorithms, differential evolution, and coevolutionary algorithms), swarm intelligence-based methods (e.g., ant colonies, particle swarm optimization), scatter search, bee colony, artificial immune systems, and so on. The common and specific search concepts of this class of metaheuristics are outlined. Many illustrative examples for their design and implementation are also presented.
- **Metaheuristics for multiobjective optimization:** In Chapter 4, the design of metaheuristics for multiobjective optimization problems is addressed according to their specific search components (fitness assignment, diversity preservation, and elitism). Many examples of multiobjective problems and popular metaheuristics are illustrated. The performance evaluation aspect is also revisited for this class of metaheuristics.
- **Hybrid metaheuristics:** Chapter 5 deals with combining metaheuristics with mathematical programming, constraint programming, and machine learning approaches. A general classification, which provides a unifying view, is defined to deal with the numerous hybridization schemes: low-level and high-level hybrids, relay and teamwork hybrids, global and partial hybrids, general and specialist hybrids, and homogeneous and heterogeneous hybrids. Both monoobjective and multiobjective hybrid optimizations are addressed.
- **Parallel and distributed metaheuristics:** Parallel and distributed metaheuristics for monoobjective and multiobjective optimization are detailed in Chapter 6.

The unified parallel models for mataheuristics (algorithmic level, iteration level, solution level) are analyzed in terms of design. The main concepts of parallel architectures and parallel programming paradigms, which interfere with the implementation of parallel metaheuristics, are also outlined.

Each chapter ends with a summary of the most important points. The evolving web site `http://paradiseo.gforge.inria.fr` contains the main material (tutorials, practical exercises, and problem-solving environment for some optimization problems).

Many search concepts are illustrated in this book: more than 170 examples and 169 exercises are provided. Appendix introduces the main concepts of the UML (Unified Modeling Language) notations and C++ concepts for an easy understanding and use of the ParadisEO framework for metaheuristics.

ACKNOWLEDGMENTS

Thanks go to

- My former and actual PhD students: V. Bachelet, M. Basseur, J.-C. Boisson, H. Bouziri, J. Brongniart, S. Cahon, Z. Hafidi, L. Jourdan, N. Jozefowiez, D. Kebbal, M. Khabzaoui, A. Khanafer, J. Lemesre, A. Liefooghe, T.-V. Luong, M. Mehdi, H. Meunier, M. Mezmaz, A. Tantar, E. Tantar, and B. Weinberg.
- Former and actual members of my research team OPAC and the INRIA DOLPHIN team project: C. Canape, F. Clautiaux, B. Derbel, C. Dhaenens, G. Even, M. Fatene, J. Humeau, T. Legrand, N. Melab, O. Schütze, and J. Tavares. A special thank to C. Dhaenens and N. Melab for their patience in reading some chapters of this book.
- Former and current collaborators: E. Alba, P. Bessière, P. Bouvry, D. Duvivier, C. Fonlupt, J.-M. Geib, K. Mellouli, T. Muntean, A.J. Nebro, P. Preux, M. Rahoual, D. Robillard, O. Roux, F. Semet, and A. Zomaya.

This book was written during my world scientific tour! This has influenced my inspiration. In fact, it was written during numerous international visits I made to attend seminars, training courses, and conferences in the past 2 years: Madrid, Algiers, Sydney, Dresden, Qatar, Saragossa, Tokyo, Dagstuhl, Gran Canaria, Amsterdam, Luxembourg, Barcelona, Pragua, Vienna, Hawaii, Porto, Montreal, Tunis, Dubai, Orlando, Rio de Janeiro, Warsaw, Hong Kong, Auckland, Singapour, Los Angeles, Tampa, Miami, Boston, Malaga, and (I will not forget it) Lille in France!

Finally, I would like to thank the team at John Wiley & Sons who gave me excellent support throughout this project.

GLOSSARY

ACO	Ant colony optimization
ADA	Annealed demon algorithm
aiNET	Artificial immune network
AIS	Artificial immune system
AMS	Adaptive multistart
ARMA	Autoregression moving average
ART	Adaptive reasoning technique
A-Teams	Asynchronous teams algorithm
BA	Bee algorithm
B&B	Branch and bound algorithm
BC	Bee colony
BDA	Bounded demon algorithm
BOA	Bayesian optimization algorithm
BOCTP	Biobjective covering tour problem
BOFSP	Biobjective flow-shop scheduling problem
CA	Cultural algorithms
CC-UMA	Cache coherent uniform memory access machine
CC-NUMA	Cache coherent nonuniform memory access machine
CEA	Coevolutionary algorithms
CIGAR	Case-injected genetic algorithm
CLONALG	Clonal selection algorithm
CLUMPS	Cluster of SMP machines
CMA	Covariance matrix adaptation
CMA-ES	CMA-evolution strategy
CMST	Capacitated minimum spanning tree problem
COSEARCH	Cooperative search algorithm
COW	Cluster of workstations
CP	Constraint programming
CPP	Clique partitioning problem
CSP	Constraint satisfaction problem
CTP	Covering tour problem

CTSP	Colorful traveling salesman problem
CVRP	Capacitated vehicle routing problem
CX	Cycle crossover
DA	Demon algorithm
DACE	Design and analysis of computer experiments
DE	Differential algorithm
DM	Data mining
DOE	Design of experiments
DP	Dynamic programming
EA	Evolutionary algorithm
EC	Evolutionary computation
EDA	Estimation of distribution algorithm
EMNA	Estimation of multivariate normal algorithm
EMO	Evolutionary multicriterion optimization
EO	Evolving objects
EP	Evolutionary programming
ES	Evolution strategy
FDC	Fitness distance correlation
FPGA	Field programming gate arrays
FPTAS	Fully polynomial-time approximation scheme
FSP	Flow-shop scheduling problem
GA	Genetic algorithms
GAP	Generalized assignment problem
GBP	Graph bipartitioning problem
GCP	Graph coloring problem
GDA	Great deluge algorithm
GLS	Guided local search algorithm
GP	Genetic programming
GPP	Graph partitioning problem
GPS	Global positioning system
GPSO	Geometric particle swarm optimization
GPU	Graphical processing unit
GRASP	Greedy adaptive search procedure
GVNS	General variable neighborhood search
HMSTP	Hop-constrained minimum spanning tree problem
HTC	High-throughput computing
HPC	High-performance computing
HRH	High-level relay hybrid

HTH	High-level teamwork hybrid
IBEA	Indicator-based evolutionary algorithm
ILP	Integer linear programming
ILS	Iterative local search
IP	Integer program
JSP	Job-shop scheduling problem
LAN	Local area network
LOP	Linear ordering problem
LP	Linear programming
LRH	Low-level relay hybrid
LS	Local search
LTH	Low-level teamwork hybrid
MBO	Marriage in honeybees optimization
MCDM	Multicriteria decision making
MDO	Multidisciplinary design optimization
MIMD	Multiple instruction streams—multiple data stream
MIP	Mixed integer programming
MISD	Multiple instruction streams—single data stream
MLS	Multistart local search
MLST	Minimum label spanning tree problem
MO	Moving objects
MOEO	Multiobjective evolving objects
MOEA	Multiobjective evolutionary algorithm
MOGA	Multiobjective genetic algorithm
MOP	Multiobjective optimization
MOSA	Multiobjective simulated annealing algorithm
MOTS	Multiobjective tabu search
MP	Mathematical programming
MPI	Message passing interface
MPP	Massively parallel processing machine
MSTP	Minimum spanning Tree Problem
NFL	No free lunch theorem
NLP	Nonlinear continuous optimization problem
NM	Noisy method
NOW	Network of workstations
NSA	Negative selection algorithm
NSGA	Nondominated sorting genetic algorithm
OBA	Old bachelor accepting algorithm

OX	Order crossover
PAES	Pareto archived evolution strategy
ParadisEO	Parallel and distributed evolving objects
PBIL	Population-based incremental learning algorithm
PCS	Parent centric crossover
PEO	Parallel Evolving objects
P-metaheuristic	Population-based metaheuristic
PMX	Partially mapped crossover
POPMUSIC	Partial optimization metaheuristic under special intensification conditions
PR	Path relinking
PSO	Particle swarm optimization
PTAS	Polynomial-time approximation scheme
PVM	Parallel virtual machine
QAP	Quadratic assignment problem
RADA	Randomized annealed demon algorithm
RBDA	Randomized bounded demon algorithm
RCL	Restricted candidate list
RMI	Remote method invocation
RPC	Remote procedural call
RRT	Record-to-record travel algorithm
SA	Simulated annealing
SAL	Smoothing algorithm
SAT	Satisfiability problems
SCP	Set covering problem
SCS	Shortest common supersequence problem
SDMCCP	Subset disjoint minimum cost cycle problem
SIMD	Single instruction stream—multiple data stream
SISD	Single instruction stream—single data stream
SM	Smoothing method
S-metaheuristic	Single-solution based metaheuristic
SMP	Symmetric multiprocessors
SMTWTP	Single-machine total-weighted tardiness problem
SPEA	Strength Pareto evolutionary algorithm
SPX	Simplex crossover
SS	Scatter search
SUS	Stochastic universal sampling
SVNS	Skewed variable neighborhood search
TA	Threshold accepting

TAPAS	Target aiming Pareto search
TS	Tabu search
TSP	Traveling salesman problem
UMDA	Univariate marginal distribution algorithm
UNDX	Unimodal normal distribution crossover
VEGA	Vector evaluated genetic algorithm
VIP	Vote-inherit-promote protocol
VND	Variable neighborhood descent
VNDS	Variable neighborhood decomposition search
VNS	Variable neighborhood search
VRP	Vehicle routing problem
WAN	Wide area network

CHAPTER 1

Common Concepts for Metaheuristics

Computing optimal solutions is intractable for many optimization problems of industrial and scientific importance. In practice, we are usually satisfied with "good" solutions, which are obtained by heuristic or metaheuristic algorithms. Metaheuristics represent a family of approximate[1] optimization techniques that gained a lot of popularity in the past two decades. They are among the most promising and successful techniques. Metaheuristics provide "acceptable" solutions in a reasonable time for solving hard and complex problems in science and engineering. This explains the significant growth of interest in metaheuristic domain. Unlike exact optimization algorithms, metaheuristics do not guarantee the optimality of the obtained solutions. Instead of approximation algorithms, metaheuristics do not define how close are the obtained solutions from the optimal ones.

The word *heuristic* has its origin in the old Greek word *heuriskein*, which means the art of discovering new strategies (rules) to solve problems. The suffix *meta*, also a Greek word, means "upper level methodology." The term *metaheuristic* was introduced by F. Glover in the paper [322]. Metaheuristic search methods can be defined as upper level general methodologies (templates) that can be used as guiding strategies in designing underlying heuristics to solve specific optimization problems.

This chapter is organized as follows. Section 1.1 discusses the diverse classical optimization models that can be used to formulate and solve optimization problems. It also introduces the basic concepts of algorithm and problem complexities. Some illustrative easy and hard optimization problems are given. Section 1.2 presents other models for optimization that are not static and deterministic. Those problems are characterized by dynamicity, uncertainty, or multiperiodicity. Then, Section 1.3 outlines the main families of optimization methods: exact versus approximate algorithms, metaheuristic versus approximation algorithms, iterative versus greedy algorithms, single-solution based metaheuristics versus population-based metaheuristics. Finally, an important question one might ask is answered: "when use metaheuristics?" Once the basic material for optimization problems and algorithms are presented, important common concepts of metaheuristics are introduced in Section 1.4, such as the

[1]There is a difference between approximate algorithms and approximation algorithms (see Section 1.3.2).

Metaheuristics: From Design to Implementation, by El-Ghazali Talbi

representation of solutions and the guiding objective function. Then, Sections 1.5, 1.6, and 1.7 present successively three important topics common to all metaheuristics: constraint handling, parameter tuning, and performance evaluation. Finally in Section 1.8, the software framework aspect of metaheuristics is discussed and the ParadisEO framework, which is used to implement the designed metaheuristics, is detailed.

1.1 OPTIMIZATION MODELS

As scientists, engineers, and managers, we always have to take decisions. Decision making is everywhere. As the world becomes more and more complex and competitive, decision making must be tackled in a rational and optimal way. Decision making consists in the following steps (Fig. 1.1):

- **Formulate the problem:** In this first step, a decision problem is identified. Then, an initial statement of the problem is made. This formulation may be imprecise. The internal and external factors and the objective(s) of the problem are outlined. Many decision makers may be involved in formulating the problem.
- **Model the problem:** In this important step, an abstract mathematical model is built for the problem. The modeler can be inspired by similar models in the literature. This will reduce the problem to well-studied optimization models. Usually, models we are solving are simplifications of the reality. They involve approximations and sometimes they skip processes that are complex to represent in a mathematical model. An interesting question may occur: why solve exactly real-life optimization problems that are fuzzy by nature?
- **Optimize the problem:** Once the problem is modeled, the solving procedure generates a "good" solution for the problem. The solution may be optimal or suboptimal. Let us notice that we are finding a solution for an abstract model of the problem and not for the originally formulated real-life problem. Therefore, the obtained solution performances are indicative when the model is an accurate one. The algorithm designer can reuse state-of-the-art algorithms on

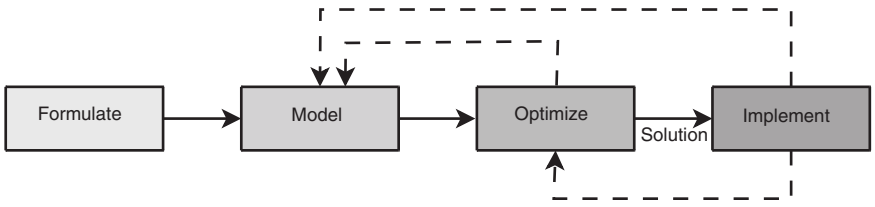

FIGURE 1.1 The classical process in decision making: formulate, model, solve, and implement. In practice, this process may be iterated to improve the optimization model or algorithm until an acceptable solution is found. Like life cycles in software engineering, the life cycle of optimization models and algorithms may be linear, spiral, or cascade.

similar problems or integrate the knowledge of this specific application into the algorithm.

- **Implement a solution:** The obtained solution is tested practically by the decision maker and is implemented if it is "acceptable." Some practical knowledge may be introduced in the solution to be implemented. If the solution is unacceptable, the model and/or the optimization algorithm has to be improved and the decision-making process is repeated.

1.1.1 Classical Optimization Models

As mentioned, optimization problems are encountered in many domains: science, engineering, management, and business. An optimization problem may be defined by the couple (S, f), where S represents the set of feasible solutions[2], and $f : S \longrightarrow \mathbb{R}$ the objective function[3] to optimize. The objective function assigns to every solution $s \in S$ of the search space a real number indicating its worth. The objective function f allows to define a total order relation between any pair of solutions in the search space.

Definition 1.1 Global optimum. *A solution $s^* \in S$ is a global optimum if it has a better objective function[4] than all solutions of the search space, that is, $\forall s \in S$, $f(s^*) \leq f(s)$.*

Hence, the main goal in solving an optimization problem is to find a global optimal solution s^*. Many global optimal solutions may exist for a given problem. Hence, to get more alternatives, the problem may also be defined as finding all global optimal solutions.

Different families of optimization models are used in practice to formulate and solve decision-making problems (Fig. 1.2). The most successful models are based on *mathematical programming* and *constraint programming*.

A commonly used model in mathematical programming is *linear programming* (LP), which can be formulated as follows:

$$\text{Min } c \cdot x$$

subject to

$$A \cdot x \geq b$$
$$x \geq 0$$

[2] A solution is also referred to as a configuration or a state. The set S is named search space, configuration space, or state space.

[3] Sometimes named cost, utility, or fitness function.

[4] We suppose without loss of generality a minimization problem. Maximizing an objective function f is equivalent to minimizing $-f$.

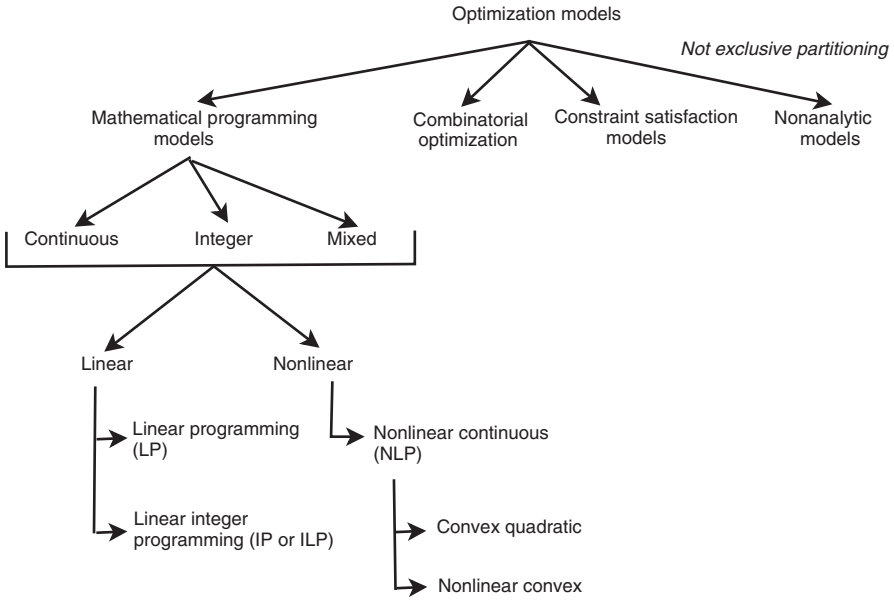

FIGURE 1.2 Classical optimization models. The different classes are possibly overlapping.

where x is a vector of continuous decision variables, and c and b (resp. A) are constant vectors (resp. matrix) of coefficients.

In a linear programming optimization problem, both the objective function $c \cdot x$ to be optimized and the constraints $A \cdot x \leq b$ are linear functions. Linear programming is one of the most satisfactory models of solving optimization problems[5]. Indeed, for continuous linear optimization problems[6], efficient exact algorithms such as the simplex-type method [174] or interior point methods exist [444]. The efficiency of the algorithms is due to the fact that the feasible region of the problem is a convex set and the objective function is a convex function. Then, the global optimum solution is necessarily a node of the polytope representing the feasible region (see Fig. 1.3). Moreover, any local optima[7] solution is a global optimum. In general, there is no reason to use metaheuristics to solve LP continuous problems.

Example 1.1 Linear programming model in decision making. A given company synthesizes two products Prod_1 and Prod_2 based on two kinds of raw materials M_1 and M_2. The objective consists in finding the most profitable product mix. Table 1.1 presents the daily available raw materials for M_1 and M_2, and for each product Prod_i the used amount of raw materials and the profit. The decision variables are x_1 and x_2 that

[5]LP models were developed during the second world war to solve logistic problems. Their use was kept secret until 1947.

[6]The decision variables are real values.

[7]See Definition 2.4 for the concept of local optimality.

TABLE 1.1 Data Associated with the Production Problem

	Usage for $Prod_1$	Usage for $Prod_2$	Material Availability
M_1	6	4	24
M_2	1	2	6
Profit per unit	€ 5	€ 4	

represent, respectively, the amounts of $Prod_1$ and $Prod_2$. The objective function consists in maximizing the profit.

The model of this problem may be formulated as an LP mathematical program:

$$\text{Max profit } = 5x_1 + 4x_2$$

subject to the constraints

$$6x_1 + 4x_2 \leq 24$$
$$1x_1 + 2x_2 \leq 6$$
$$x_1, x_2 \geq 0$$

Figure 1.3 illustrates the graphical interpretation of the model. Each constraint can be represented by a line. The objective function is an infinity of parallel lines. The optimum solution will always lie at an extreme point. The optimal solution is ($x_1 = 3$, $x_2 = 1.5$) with a profit of € 21.

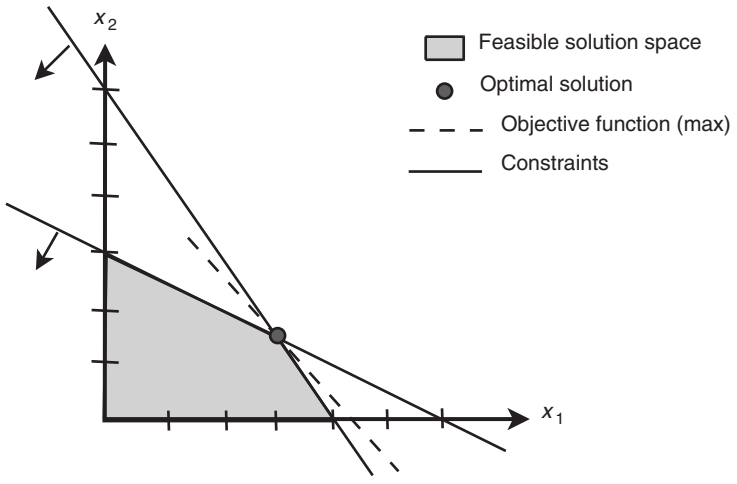

FIGURE 1.3 Graphical illustration of the LP model and its resolution.

Nonlinear programming models (NLP)[8] deal with mathematical programming problems where the objective function and/or the constraints are nonlinear [72]. A continuous nonlinear optimization problem consists in minimizing a function $f : S \subset \mathbb{R}^n \longrightarrow \mathbb{R}$ in a continuous domain. Nonlinear continuous models are, however, much more difficult to solve, even if there are many modeling possibilities that may be used to linearize a model: linearizing a product of variables [321], logical conditions, ordered set of variables, and so on [31]. Linearization techniques introduce in general extra variables and constraints in the model and in some cases some degree of approximation [319].

For NLP optimization models, specific simplex-inspired heuristics such as the Nelder and Mead algorithm may be used [578]. For quadratic and convex continuous problems, some efficient exact algorithms can be used to solve small or moderate problems [583]. Unfortunately, some problem properties such as high dimensionality, multimodality, epistasis (parameter interaction), and nondifferentiability render those traditional approaches impotent. Metaheuristics are good candidates for this class of problems to solve moderate and large instances.

Continuous optimization[9] theory in terms of optimization algorithms is more developed than discrete optimization. However, there are many real-life applications that must be modeled with discrete variables. Continuous models are inappropriate for those problems. Indeed, in many practical optimization problems, the resources are indivisible (machines, people, etc.). In an *integer program* (IP)[10] optimization model, the decision variables are discrete [579].

When the decision variables are both discrete and continuous, we are dealing with *mixed integer programming problems* (MIP). Hence, MIP models generalize LP and IP models. Solving MIP problems has improved dramatically of late with the use of advanced optimization techniques such as relaxations and decomposition approaches, and cutting plane algorithms (see Section 5.2.1). For IP and MIP models, enumerative algorithms such as branch and bound may be used for small instances. The size is not the only indicator of the complexity of the problem, but also its structure. Metaheuristics are one of the competing algorithms for this class of problems to obtain good solutions for instances considered too complex to be solved in an exact manner. Metaheuristics can also be used to generate good lower or upper bounds for exact algorithms and improve their efficiency. Notice that there are some easy problems, such as *network flow problems*, where linear programming automatically generates integer values. Hence, both integer programming approaches and metaheuristics are not useful to solve those classes of problems.

A more general class of IP problems is *combinatorial optimization* problems. This class of problems is characterized by discrete decision variables and a finite search space. However, the objective function and constraints may take any form [597].

[8] Also referred to as global optimization.

[9] Also called real parameter optimization.

[10] IP models denote implicitly linear models (integer linear programming: ILP).

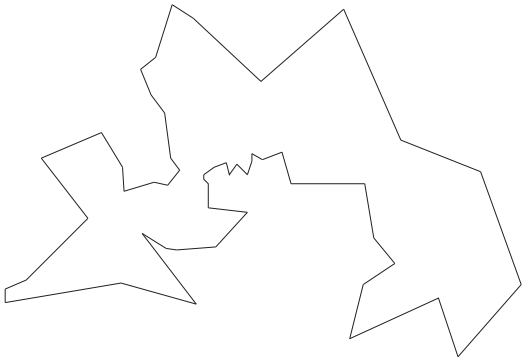

FIGURE 1.4 TSP instance with 52 cities.

The popularity of combinatorial optimization problems stems from the fact that in many real-world problems, the objective function and constraints are of different nature (nonlinear, nonanalytic, black box, etc.) whereas the search space is finite.

Example 1.2 Traveling salesman problem. Perhaps the most popular combinatorial optimization problem is the traveling salesman problem (TSP). It can be formulated as follows: given n cities and a distance matrix $d_{n,n}$, where each element d_{ij} represents the distance between the cities i and j, find a tour that minimizes the total distance. A tour visits each city exactly once (Hamiltonian cycle) (Figs. 1.4 and 1.5). The size of the search space is $n!$ Table 1.2 shows the combinatorial explosion of the number of solutions regarding the number of cities. Unfortunately, enumerating exhaustively all possible solutions is impractical for moderate and large instances.

Another common approach to model decision and optimization problems is *constraint programming* (CP), a programming paradigm that integrates richer modeling tools than the linear expressions of MIP models. A model is composed of a set of variables. Every variable has a finite domain of values. In the model, symbolic and

TABLE 1.2 Effect of the Number of Cities on the Size of the Search Space

Number of Cities n	Size of the Search Space
5	120
10	3, 628, 800
75	2.5×10^{109}

FIGURE 1.5 TSP instance with 24,978 cities.

mathematical constraints related to variables may be expressed. Global constraints represent constraints that refer to a set of variables. Hence, the CP paradigm models the properties of the desired solution. The declarative models in CP are flexible and are in general more compact than in MIP models.

Example 1.3 Assignment problem within constraint programming. The goal is to assign n objects $\{o_1, o_2, \ldots, o_n\}$ to n locations $\{l_1, l_2, \ldots, l_n\}$ where each object is placed on a different location. Using constraint programming techniques, the model will be the following:

$$\texttt{all\_different}(y_1, y_2, \ldots, y_n)$$

where y_i represents the index of the location to which the object o_i is assigned. The global constraint $\texttt{all\_different}(y_1, y_2, \ldots, y_n)$ specifies that all variables must be different. If this problem is modeled using an IP model, one has to introduce the following decision variables:

$$x_{ij} = \begin{cases} 1 & \text{if object } o_i \text{ is assigned to location } l_j \\ 0 & \text{otherwise} \end{cases}$$

Hence, much more variables (n^2 instead of n) are declared.

However, it does not mean that solving the problem will be more efficient within constraint programming than using mathematical programming. Solving the problem is another story. The advantage of MIP solvers is that they use relaxations of the problem to prune the search tree, while in CP they use constraint propagation techniques to

reduce the variable domains (see Section 5.3.1). The efficiency of the solvers depends mainly on the structure of the target problem and its associated model. The modeling step of the problem is then very important. In general, CP techniques are less suitable for problems with a large number of feasible solutions, such as the assignment problem shown in the previous example. They are usually used for "tight" constrained problems such as timetabling and scheduling problems.

There are often many ways to formulate mathematically an optimization problem. The efficiency obtained in solving a given model may depend on the formulation used. This is why a lot of research is directed on the reformulation of optimization problems. It is sometimes interesting to increase the number of integer variables and constraints. For a more comprehensive study of mathematical programming approaches (resp. constraint programming techniques), the reader may refer to Refs [37,300,686] (resp. [34,287,664]).

For many problems arising in practical applications, one cannot expect the availability of analytical optimization models. For instance, in some applications one has to resort to simulation or physical models to evaluate the objective function. Mathematical programming and constraint programming approaches require an explicit mathematical formulation that is impossible to derive in problems where simulation is relevant [288].

1.1.2 Complexity Theory

This section deals with some results on intractability of problem solving. Our focus is the complexity of decidable problems. *Undecidable* problems[11] could never have any algorithm to solve them even with unlimited time and space resources [730]. A popular example of undecidable problems is the *halting problem* [782].

1.1.2.1 Complexity of Algorithms

An algorithm needs two important resources to solve a problem: time and space. The time complexity of an algorithm is the number of steps required to solve a problem of size n. The complexity is generally defined in terms of the worst-case analysis.

The goal in the determination of the computational complexity of an algorithm is not to obtain an exact step count but an asymptotic bound on the step count. The Big-O notation makes use of asymptotic analysis. It is one of the most popular notations in the analysis of algorithms.

Definition 1.2 Big-O notation. *An algorithm has a complexity* $f(n) = O(g(n))$ *if there exist positive constants* n_0 *and* c *such that* $\forall n > n_0$, $f(n) \leq c \cdot g(n)$.

In this case, the function $f(n)$ is upper bounded by the function $g(n)$. The Big-O notation can be used to compute the time or the space complexity of an algorithm.

[11] Also called *noncomputable problems*.

Definition 1.3 Polynomial-time algorithm. *An algorithm is a polynomial-time algorithm if its complexity is* $O(p(n))$*, where* $p(n)$ *is a polynomial function of* n.

A polynomial function of degree k can be defined as follows:

$$p(n) = a_k \cdot n^k + \cdots + a_j \cdot n^j + \cdots + a_1 \cdot n + a_0$$

where $a_k > 0$ and $a_j \geq 0, \forall 1 \leq j \leq k-1$. The corresponding algorithm has a polynomial complexity of $O(n^k)$.

Example 1.4 Complexity of shortest path algorithms. Given a connected graph $G = (V, E)$, where V represents the set of nodes and E the set of edges. Let $D = (d_{ij})$ be a distance matrix where d_{ij} is the distance between the nodes i and j (we assume $d_{ij} = d_{ji} > 0$). The shortest path problem consists in finding the path from a source node i to a destination node j. A path $\pi(i, j)$ from i to j can be defined as a sequence $(i, i_1, i_2, \ldots, i_k, j)$, such that $(i, i_1) \in E, (i_k, j) \in E, (i_l, i_{l+1}) \in E, \forall 1 \leq l \leq k-1$. The length of a path $\pi(i, j)$ is the sum of the weights of its edges:

$$\text{length}(\pi(i, j)) = d_{ii_1} + d_{i_k j} + \sum_{l=1}^{k-1} d_{i_l i_{l+1}}$$

Let us consider the well-known Dijkstra algorithm to compute the shortest path between two nodes i and j [211]. It works by constructing a shortest path tree from the initial node to every other node in the graph. For each node of the graph, we have to consider all its neighbors. In the worst-case analysis, the number of neighbors for a node is in the order of n. The Dijkstra algorithm requires $O(n^2)$ running time where n represents the number of nodes of the graph. Then, the algorithm requires no more than a quadratic number of steps to find the shortest path. It is a polynomial-time algorithm.

Definition 1.4 Exponential-time algorithm. *An algorithm is an exponential-time algorithm if its complexity is* $O(c^n)$*, where* c *is a real constant strictly superior to* 1.

Table 1.3 illustrates how the search time of an algorithm grows with the size of the problem using different time complexities of an optimization algorithm. The table shows clearly the combinatorial explosion of exponential complexities compared to polynomial ones. In practice, one cannot wait some centuries to solve a problem. The problem shown in the last line of the table needs the age of universe to solve it in an exact manner using exhaustive search.

Two other notations are used to analyze algorithms: the Big-Ω and the Big-Θ notations.

Definition 1.5 Big-Ω notation. *An algorithm has a complexity* $f(n) = \Omega(g(n))$ *if there exist positive constants* n_0 *and* c *such that* $\forall n > n_0$, $f(n) \geq c \cdot g(n)$*. The complexity of the algorithm* $f(n)$ *is lower bounded by the function* $g(n)$.

TABLE 1.3 Search Time of an Algorithm as a Function of the Problem Size Using Different Complexities (from [299])

Complexity	Size = 10	Size = 20	Size = 30	Size = 40	Size = 50
$O(x)$	0.00001 s	0.00002 s	0.00003 s	0.00004 s	0.00005 s
$O(x^2)$	0.0001 s	0.0004 s	0.0009 s	0.0016 s	0.0025 s
$O(x^5)$	0.1 s	0.32 s	24.3 s	1.7 mn	5.2 mn
$O(2^x)$	0.001 s	1.0 s	17.9 mn	12.7 days	35.7 years
$O(3^x)$	0.059 s	58.0 mn	6.5 years	3855 centuries	2×10^8 centuries

Definition 1.6 Big-Θ notation. *An algorithm has a complexity* $f(n) = \Theta(g(n))$ *if there exist positive constants* n_0, c_1, *and* c_2 *such that* $\forall n > n_0, c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$. *The complexity of the algorithm* $f(n)$ *is lower bounded by the function* $g(n)$.

It is easier to find first the Big-O complexity of an algorithm, then derive successively the Big-Ω and Big-Θ complexities. The Big-Θ notation defines the exact bound (lower and upper) on the time complexity of an algorithm.

The asymptotic analysis of algorithms characterizes the growth rate of their time complexity as a function of the problem size (scalability issues). It allows a theoretical comparison of different algorithms in terms of the worst-case complexity. It does not specify the practical run time of the algorithm for a given instance of the problem. Indeed, the run time of an algorithm depends on the input data. For a more complete analysis, one can also derive the average-case complexities, which is a more difficult task.

1.1.2.2 Complexity of Problems

The complexity of a problem is equivalent to the complexity of the best algorithm solving that problem. A problem is *tractable* (or easy) if there exists a polynomial-time algorithm to solve it. A problem is *intractable* (or difficult) if no polynomial-time algorithm exists to solve the problem.

The complexity theory of problems deals with *decision problems*. A decision problem always has a yes or no answer.

Example 1.5 Prime number decision problem. A popular decision problem consists in answering the following question: is a given number Q a prime number? It will return yes if the number Q is a prime one, otherwise the no answer is returned.

An optimization problem can always be reduced to a decision problem.

Example 1.6 Optimization versus decision problem. The optimization problem associated with the traveling salesman problem is "find the optimal Hamiltonian tour that optimizes the total distance," whereas the decision problem is "given an integer D, is there a Hamiltonian tour with a distance less than or equal to D?"

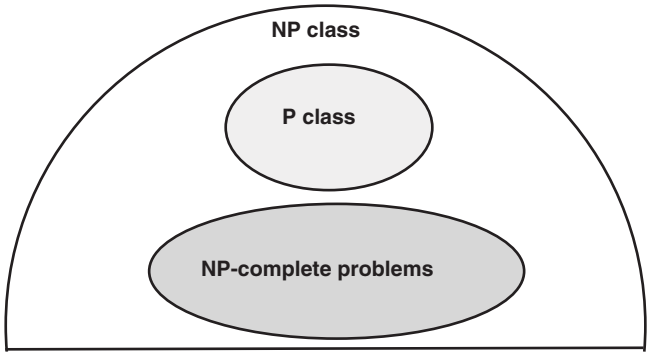

FIGURE 1.6 Complexity classes of decision problems.

An important aspect of computational theory is to categorize problems into complexity classes. A complexity class represents the set of all problems that can be solved using a given amount of computational resources. There are two important classes of problems: P and NP (Fig. 1.6).

The complexity class P represents the set of all decision problems that can be solved by a deterministic machine in polynomial time. A (deterministic) algorithm is polynomial for a decision problem A if its *worst*[12] complexity is bounded by a polynomial function $p(n)$ where n represents the size of the input instance I. Hence, the class P represents the family of problems where a known polynomial-time algorithm exists to solve the problem. Problems belonging to the class P are then relatively "easy" to solve.

Example 1.7 Some problems of class P. Some classical problems belonging to class P are minimum spanning tree, shortest path problems, maximum flow network, maximum bipartite matching, and linear programming continuous models[13]. In the book of Garey and Johnson, a more exhaustive list of easy and hard class P problems can be found [299].

The complexity class NP represents the set of all decision problems that can be solved by a nondeterministic algorithm[14] in polynomial time. A nondeterministic algorithm contains one or more choice points in which multiple different continuations are possible without any specification of which one will be taken. It uses the primitives: `choice` that proposes a solution (oracle), `check` that verifies in polynomial time if a solution proposal (certificate) gives a positive or negative answer, `success` when the algorithm answers yes after the check application, and `fail` when the algorithm

[12] We take into account the worst-case performance and not the average one.
[13] Linear programming continuous problems belong to class P, whereas one of the most efficient algorithms to solve LP programs, the simplex algorithm, has an exponential complexity.
[14] In computer science, the term algorithm stands for a deterministic algorithm.

does not respond "yes." Then, if the `choice` primitive proposes a solution that gives a "yes" answer and the oracle has the capacity to do it, then the computing complexity is polynomial.

Example 1.8 Nondeterministic algorithm for the 0–1 knapsack problem. The 0–1 knapsack decision problem can be defined as follows. Given a set of N objects. Each object O has a specified weight and a specified value. Given a capacity, which is the maximum total weight of the knapsack, and a quota, which is the minimum total value that one wants to get. The 0–1 knapsack decision problem consists in finding a subset of the objects whose total weight is at most equal to the capacity and whose total value is at least equal to the specified quota.

Let us consider the following nondeterministic algorithm to solve the knapsack decision problem:

Algorithm 1.1 Nondeterministic algorithm for the knapsack problem.

```
Input OS : set of objects ; QUOTA : number ; CAPACITY : number.
Output S : set of objects ; FOUND : boolean.
  S = empty ; total_value = 0 ; total_weight = 0 ; FOUND = false ;
  Pick an order L over the objects ;
  Loop
    Choose an object O in L ; Add O to S ;
    total_value = total_value + O.value ;
    total_weight = total_weight + O.weight ;
    If total_weight > CAPACITY Then fail
    Else If total_value ≥ QUOTA
         FOUND = true ;
         succeed ;
    Endif Endif
    Delete all objects up to O from L ;
  Endloop
```

The question whether P = NP[15] is one of the most important open questions due to the wide impact the answer would have on computational complexity theory. Obviously, for each problem in P we have a nondeterministic algorithm solving it. Then, P $\subseteq$ NP (Fig. 1.6). However, the following conjecture P $\subset$ NP is still an open question.

A decision problem A is *reduced polynomially* to a decision problem B if, for all input instances I_A for A, one can always construct an input instance I_B for B in polynomial-time function to the size $L(I_A)$ of the input I_A, such that I_A is a positive instance of A if and only if I_B is a positive instance of B.

[15]The question is one of the millennium problems with a prize of US$ 1,000,000 for a first-found solution.

A decision problem $A \in$ NP is *NP-complete* if *all* other problems of class NP are reduced polynomially to the problem A. Figure 1.6 shows the relationship between P, NP, and NP-complete problems. If a polynomial deterministic algorithm exists to solve an NP-complete problem, then all problems of class NP may be solved in polynomial time.

NP-hard problems are optimization problems whose associated decision problems are NP-complete. Most of the real-world optimization problems are NP-hard for which provably efficient algorithms do not exist. They require exponential time (unless P = NP) to be solved in optimality. Metaheuristics constitute an important alternative to solve this class of problems.

Example 1.9 Some NP-hard problems. Cook (1971) was the first to prove that the satisfiability problem (SAT) is NP-complete. The other NP-complete problems are at least as hard as the SAT problem. Many academic popular problems are NP-hard among them:

- Sequencing and scheduling problems such as flow-shop scheduling, job-shop scheduling, or open-shop scheduling.
- Assignment and location problems such as quadratic assignment problem (QAP), generalized assignment problem (GAP), location facility, and the p-median problem.
- Grouping problems such as data clustering, graph partitioning, and graph coloring.
- Routing and covering problems such as vehicle routing problems (VRP), set covering problem (SCP), Steiner tree problem, and covering tour problem (CTP).
- Knapsack and packing/cutting problems, and so on.

Many of those optimization problems (and others) will be introduced in the book in a progressive manner to illustrate the design of search components of metaheuristics. Those optimization problems are canonical models that can be applied to different real-life applications. Integer programming models belong in general to the NP-complete class. Unlike LP models, IP problems are difficult to solve because the feasible region is not a convex set.

Example 1.10 Still open problems. Some problems have not yet been proved to be NP-hard. A popular example is the graph isomorphism problem that determines if two graphs are isomorphic. Whether the problem is in P or NP-complete is an open question. More examples may be found in Ref. [299].

1.2 OTHER MODELS FOR OPTIMIZATION

The rest of the book focuses mainly on solving static and deterministic problems. Demand is growing to solve real-world optimization problems where the data are noisy or the objective function is changing dynamically. Finding robust solutions for some design problems is another important challenge in optimization. A transformation to

deterministic and static problems is often proposed to solve such problems. Moreover, some adaptations may be proposed for metaheuristics in terms of intensification and diversification of the search to tackle this class of problems [414]. Chapter 4 deals with another class of optimization problems characterized by multiple objectives: the multiobjective optimization problems (MOP) class.

1.2.1 Optimization Under Uncertainty

In many concrete optimization problems, the input data are subject to noise. There are various sources of noise. For instance, the use of a stochastic simulator or an inherently noisy measurement device such as sensors will introduce an additive noise in the objective function. For a given solution x in the search space, a noisy objective function can be defined mathematically as follows:

$$f_{\text{noisy}}(x) = \int_{-\infty}^{+\infty} [f(x) + z]p(z)\mathrm{d}z$$

where $p(z)$ represents the probability distribution of the additive noise z. The additive noise z is mostly assumed to be normally distributed $N(0, \sigma)$ with zero mean and a σ variance [414]. Non-Gaussian noise can also be considered, such as the Cauchy distribution. For the same solution x, different values of the objective function f_{noisy} may be obtained by multiple evaluations. Unlike dynamic optimization, the function f_{noisy} is time invariant.

In practice, the objective function f_{noisy} is often approximated by the function f'_{noisy}, which is defined by the mean value on a given number of samples:

$$f'_{\text{noisy}}(x) = \frac{\sum_{i=1}^{N} [f(x) + z_i]}{N}$$

where z_i represents the noise associated with the sample i and N is the number of samples.

Example 1.11 Uncertainty in routing and scheduling problems. Uncertainty may be present in different components of routing problems. In vehicle routing problems, stochastic demands or stochastic transportation times between locations may be considered as sources of uncertainty. In scheduling problems, uncertainty can occur from many sources such as variable processing and release times or due date variations.

The simplest approach to handle uncertainty is to estimate the mean value of each parameter and solve a deterministic problem. The domain of *stochastic programming* has the goal to solve some limited range of optimization problems under uncertainty [442,674]. Hence, metaheuristics for solving deterministic optimization problems can help solve problems with uncertainty.

1.2.2 Dynamic Optimization

Dynamic Optimization problems represent an important challenge in many real-life applications. The input elements of the problem change over time. In dynamic optimization problems, the objective function is deterministic at a given time but varies over the time [414]:

$$f_{\text{dynamic}}(x) = f_t(x)$$

where t represents the time at which the objective function is evaluated. In that case, the optimal solution of the problem changes as well. Unlike optimization with uncertainty, the function f is deterministic. At a given time, the multiple evaluations of the objective function always give the same values.

Example 1.12 Dynamic routing problems. In many routing problems such as traveling salesman and vehicle routing problems, the properties of the input graph can change over time concurrently with the search process. For the TSP, some cities may be added or deleted during the tour. For the VRP, one can expect a new demand (new customer) to be handled in the problem. A solution might be regarded as a global optimal solution at a given time and may not be optimal in the next time.

The main issues in solving dynamic optimization problems are [91,564]

- Detect the change in the environment when it occurs. For most of real-life problems, the change is smooth rather than radical.
- Respond to the change in the environment to track the new global optimal solution. Hence, the search process must adapt quickly to the change of the objective function. The goal is to track dynamically the changing optimal solution as close as possible. The main challenge is to reuse information on previous searches to adapt to the problem change instead of re-solving the problem from scratch. Some forecasting strategies may also be used to predict the future scenarios.

The main question in designing a metaheuristic for dynamic optimization problems is what information during the search must be memorized and how this information will be used to guide the search and maintain adaptability to changes [91].

1.2.2.1 Multiperiodic Optimization In multiperiodic problems, the input data change periodically. It is a class of dynamic optimization problems where the change is known *a priori*. So, one has to take into account the planning horizon in optimizing those models. In general, static models taking into account the whole horizon are used to tackle this class of problems.

Example 1.13 Multiperiodic planning problem. An example of multiperiodic problems may be the planning of mobile telecommunication networks. One can design

the network by taking into account all the periods. For instance, each period is characterized by a given traffic or new incoming technology. In designing the network at a given period, the telecommunication operator must take into account the future evolutions to make the implemented solution more flexible for the future periods. Optimizing the static models in sequence for each period may produce a solution that is not optimal over the whole planning horizon. For instance, the optimal planning for a period i may not be well adapted to a future period $i + k$ with a higher traffic in a given region. A multiperiodic model must integrate all the data associated with all periods to find the sequence of optimal solutions over the successive periods.

1.2.3 Robust Optimization

In many optimization problems, the decision variables or the environmental variables are perturbed or subject to change after a final solution has been obtained and implemented for the problem. Hence, in solving the problem we have to take into account that a solution should be acceptable with respect to slight changes of the decision variable values. The term *robust* qualifies those solutions. Robust optimization may be seen as a specific kind of problem with uncertainties.

Example 1.14 Robustness in multidisciplinary design optimization and engineering design. Robust optimization is of great importance in many domains such as in engineering design. The growing interest is driven by engineering demands to produce extremely robust solutions. Indeed, in this class of problems, the implemented solution must be insensitive to small variation in the design parameters. This variation may be caused by production tolerances, or parameter drifts during operation [74]. Another important application of robust optimization is in multidisciplinary design optimization, where multiple teams associated with different disciplines design a complex system by independently optimizing subsystems. For complexity reasons (time and/or cost), each team will optimize its own subsystem without a full and precise information on the output of other subsystems.

There are many possible ways to deal with robustness. The most used measure is to optimize the expected objective function given a probability distribution of the variation. The expected objective function to optimize in searching robust solutions may be formulated as follows:

$$f_{\text{robust}}(x) = \int_{-\infty}^{+\infty} f(x + \delta)p(\delta)\mathrm{d}\delta$$

where $p(\delta)$ represents the probability distribution of the decision variable disturbance. In general, the distribution takes a normal distribution. Usually, this effective objective function is not available. Hence, it is approximated, for instance, by a Monte Carlo

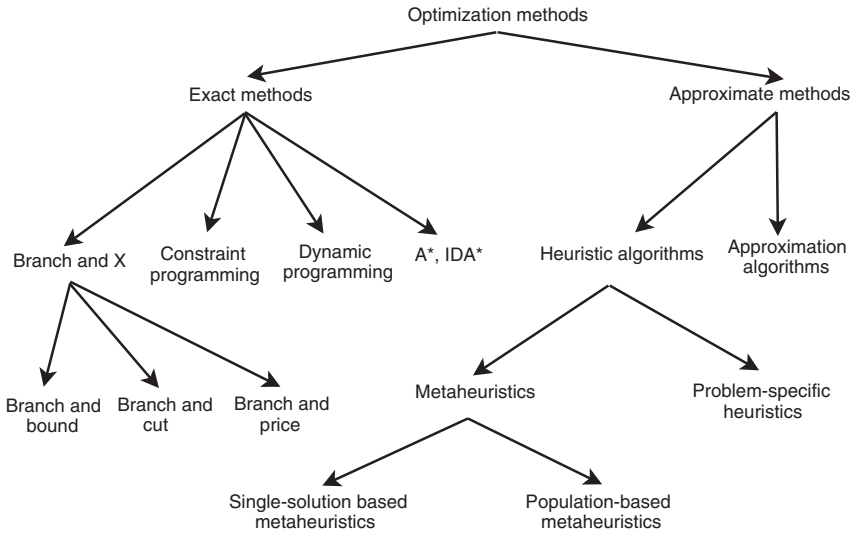

FIGURE 1.7 Classical optimization methods.

integration:

$$f'_{\text{robust}}(x) = \frac{\sum_{i=1}^{N} f(x+\delta_i)}{N}$$

Robust optimization has to find a trade-off between the quality of solutions and their robustness in terms of decision variable disturbance. This problem may be formulated as a multiobjective optimization problem (see Chapter 4) [415]. Unlike optimization under uncertainty, the objective function in robust optimization is considered as deterministic.

The introduced different variants of optimization models are not exclusive. For instance, many practical optimization problems include uncertainty as well as robustness and/or multiperiodicity. Thus, uncertainty, robustness, and dynamic issues must be jointly considered to solve this class of problems.

1.3 OPTIMIZATION METHODS

Following the complexity of the problem, it may be solved by an exact method or an approximate method (Fig. 1.7). Exact methods[16] obtain optimal solutions and guarantee their optimality. For NP-complete problems, exact algorithms are nonpolynomial-time algorithms (unless P = NP). Approximate (or heuristic) methods generate high-quality solutions in a reasonable time for practical use, but there is no guarantee of finding a global optimal solution.

[16]In the artificial intelligence community, those algorithms are also named *complete* algorithms.

1.3.1 Exact Methods

In the class of exact methods one can find the following classical algorithms: dynamic programming, branch and X family of algorithms (branch and bound, branch and cut, branch and price) developed in the operations research community, constraint programming, and A* family of search algorithms (A*, IDA*—iterative deepening algorithms) [473] developed in the artificial intelligence community [673]. Those enumerative methods may be viewed as tree search algorithms. The search is carried out over the whole interesting search space, and the problem is solved by subdividing it into simpler problems.

Dynamic programming is based on the recursive division of a problem into simpler subproblems. This procedure is based on the *Bellman's principle* that says that "the subpolicy of an optimal policy is itself optimal" [68]. This stagewise optimization method is the result of a sequence of partial decisions. The procedure avoids a total enumeration of the search space by pruning partial decision sequences that cannot lead to the optimal solution.

The *branch and bound* algorithm and *A** are based on an implicit enumeration of all solutions of the considered optimization problem. The search space is explored by dynamically building a tree whose root node represents the problem being solved and its whole associated search space. The leaf nodes are the potential solutions and the internal nodes are subproblems of the total solution space. The pruning of the search tree is based on a bounding function that prunes subtrees that do not contain any optimal solution. A more detailed description of dynamic programming and branch and bound algorithms may be found in Section 5.2.1.

Constraint programming is a language built around concepts of tree search and logical implications. Optimization problems in constraint programming are modeled by means of a set of variables linked by a set of constraints. The variables take their values on a finite domain of integers. The constraints may have mathematical or symbolic forms. A more detailed description of constraint programming techniques may be found in Section 5.3.1.

Exact methods can be applied to small instances of difficult problems. Table 1.4 shows for some popular NP-hard optimization problems the order of magnitude of the maximal size of instances that state-of-the-art exact methods can solve to optimality. Some of the exact algorithms used are implemented on large networks of workstations

TABLE 1.4 Order of Magnitude of the Maximal Size of Instances that State-of-the-Art Exact Methods can Solve to Optimality

Optimization Problems	Quadratic Assignment	Flow-Shop Scheduling (FSP)	Graph Coloring	Capacitated Vehicle Routing
Size of the instances	30 objects	100 jobs 20 machines	100 nodes	60 clients

For some practical problems, this maximum size may be negligible. For the TSP problem, an instance of size 13,509 has been solved to optimality [32].

TABLE 1.5 The Impact of the Structure on the Size of Instances (i.e., Number of Nodes for SOP and GC, Number of Objects for QAP) that State-of-the-Art Exact Methods can Solve to Optimality (SOP: Sequential Ordering Problem; QAP: Quadratic Assignment Problem; GC: Graph Coloring)

Optimization Problem	SOP	QAP	GC
Size of some unsolved instances	53	30	125
Size of some solved instances	70	36	561

(grid computing platforms) composed of more than 2000 processors with more than 2 months of computing time [546]!

The size of the instance is not the unique indicator that describes the difficulty of a problem, but also its structure. For a given problem, some small instances cannot be solved by an exact algorithm while some large instances may be solved exactly by the same algorithm. Table 1.5 shows for some popular optimization problems (e.g., SOP[17]: sequential ordering problem; QAP[18]: quadratic assignment problem; GC[19]: graph coloring) small instances that are not solved exactly and large instances solved exactly by state-of-the-art exact optimization methods.

Example 1.15 Phase transition. In many NP-hard optimization problems, a phase transition occurs in terms of the easiness/hardness of the problem; that is, the difficulty to solve the problem increases until a given size n, and beyond this value the problem is easier to solve [126]. Then, the hardest problems tend to be in the phase transition boundary. Let us consider the number partitioning problem, a widely cited NP-hard problem. Given a bag S of N positive integers $\{a_1, a_2, \ldots, a_n\}$, find a partition of the numbers into two equally disjoint bags S_1 and S_2 of cardinality $n/2$ that minimizes the absolute value of the difference of their sums:

$$f = \left| \sum_{i \in S_1} a_i - \sum_{i \in S_2} a_i \right|$$

For the number partitioning problem, the phase transition has been identified around the problem size of $n = 35$ [310,474].

Phase transition phenomena have also been identified in various problems such as graph coloring [126], SAT (propositional satisfiability) [558], CSP (constraint satisfaction problems) [706], traveling salesman problems [312], independent set problems [311], and Hamiltonian circuits [126].

In solving SAT problems, instances before the phase transition are easy to solve and those after the phase transition are mostly unsatisfiable [151]. The phase transition is formulated by the ratio between the number of clauses l and the number

[17]See `http://www.iwr.uni-heidelberg.de/groups/comopt/software/TSPLIB95/`.
[18]See `http://www.seas.upenn.edu/qaplib/inst.html`.
[19]See `http://mat.gsia.cmu.edu/COLOR/instances.html`.

of variables n. In a problem with k variables per clause, the phase transition can be estimated as [311]

$$\frac{l}{n} \approx \frac{\ln(2)}{\ln(1 - \frac{1}{2^k})}$$

For instance, for 3-SAT problems, the phase transition has been found experimentally around 4.3 [151].

1.3.2 Approximate Algorithms

In the class of approximate methods, two subclasses of algorithms may be distinguished: approximation algorithms and heuristic algorithms. Unlike heuristics, which usually find reasonably "good" solutions in a reasonable time, approximation algorithms provide provable solution quality and provable run-time bounds.

Heuristics find "good" solutions on large-size problem instances. They allow to obtain acceptable performance at acceptable costs in a wide range of problems. In general, heuristics do not have an approximation guarantee on the obtained solutions. They may be classified into two families: *specific heuristics* and *metaheuristics*. Specific heuristics are tailored and designed to solve a specific problem and/or instance. Metaheuristics are general-purpose algorithms that can be applied to solve almost any optimization problem. They may be viewed as upper level general methodologies that can be used as a guiding strategy in designing underlying heuristics to solve specific optimization problems.

1.3.2.1 Approximation Algorithms

In approximation algorithms, there is a guarantee on the bound of the obtained solution from the global optimum [380]. An ϵ-approximation algorithm generates an approximate solution a not less than a factor ϵ times the optimum solution s [793].

Definition 1.7 ϵ-Approximation algorithms. *An algorithm has an approximation factor ϵ if its time complexity is polynomial and for any input instance it produces a solution a such that*[20]

$$a \leq \epsilon \cdot s \quad \text{if } \epsilon > 1$$
$$\epsilon \cdot s \leq a \quad \text{if } \epsilon < 1$$

where s is the global optimal solution, and the factor ϵ defines the relative performance guarantee. The ϵ factor can be a constant or a function of the size of the input instance.

[20] In a minimization context.

An ϵ-approximation algorithm generates an *absolute performance guarantee*[21] ϵ, if the following property is proven:

$$(s - \epsilon) \leq a \leq (s + \epsilon)$$

Example 1.16 ϵ-Approximation for the bin packing problem. The bin packing problem is an NP-hard combinatorial optimization problem. Given a set of objects of different size and a finite number of bins of a given capacity. The problem consists in packing the set of objects so as to minimize the number of used bins. Approximation algorithms are generally greedy heuristics using the principle "hardest first, easiest last." The first fit greedy heuristic places each item into the first bin in which it will fit. The complexity of the first fit algorithm is $\Theta(n \cdot \log(n))$. An example of a good approximation algorithm for the bin packing problem is obtained by the first fit descending heuristic (FFD), which first sorts the objects into decreasing order by size:

$$\frac{11}{9}\text{opt} + 1$$

where opt is the number of bins given by the optimal solution. Without the sorting procedure, a worst bound is obtained within less computational time:

$$\frac{17}{10}\text{opt} + 2$$

NP-hard problems differ in their approximability. A well-known family of approximation problems is the PTAS class, where the problem can be approximated within any factor greater than 1.

Definition 1.8 PTAS (polynomial-time approximation scheme). *A problem is in the PTAS class if it has polynomial-time* $(1 + \epsilon)$*-approximation algorithm for any fixed* $\epsilon > 0$.

Definition 1.9 FPTAS (fully polynomial-time approximation scheme). *A problem is in the FPTAS class if it has polynomial-time* $(1 + \epsilon)$*-approximation algorithm in terms of both the input size and* $1/\epsilon$ *for any fixed* $\epsilon > 0$.

Some NP-hard problems are impossible to approximate within any constant factor (or even polynomial, unless P = NP)[22].

Example 1.17 PTAS for the 0–1 knapsack problem. Some problems such as Euclidean TSP, knapsack, and some scheduling problems are in the PTAS class. The

[21] Also referred to as bounded error.

[22] At `http://www.nada.kth.se/~viggo/wwwcompendium/`, there is a continuously updated catalog of approximability results for NP optimization problems.

0–1 knapsack problem has an FPTAS with a time complexity of $O(n^3/\epsilon)$. Problems such as the Max-SAT and vertex cover are much harder and are not members of the PTAS class.

The goal in designing an approximation algorithm for a problem is to find tight worst-case bounds. The study of approximation algorithms gives more knowledge on the difficulty of the problem and can help designing efficient heuristics. However, approximation algorithms are *specific* to the target optimization problem (problem dependent). This characteristic limits their applicability. Moreover, in practice, attainable approximations are too far from the global optimal solution, making those algorithms not very useful for many real-life applications.

1.3.3 Metaheuristics

Unlike exact methods, metaheuristics allow to tackle large-size problem instances by delivering satisfactory solutions in a reasonable time. There is no guarantee to find global optimal solutions or even bounded solutions. Metaheuristics have received more and more popularity in the past 20 years. Their use in many applications shows their efficiency and effectiveness to solve large and complex problems. Application of metaheuristics falls into a large number of areas; some them are

- Engineering design, topology optimization and structural optimization in electronics and VLSI, aerodynamics, fluid dynamics, telecommunications, automotive, and robotics.
- Machine learning and data mining in bioinformatics and computational biology, and finance.
- System modeling, simulation and identification in chemistry, physics, and biology; control, signal, and image processing.
- Planning in routing problems, robot planning, scheduling and production problems, logistics and transportation, supply chain management, environment, and so on.

Optimization is everywhere; optimization problems are often complex; then metaheuristics are everywhere. Even in the research community, the number of sessions, workshops, and conferences dealing with metaheuristics is growing significantly!

Figure 1.8 shows the genealogy of the numerous metaheuristics. The heuristic concept in solving optimization problems was introduced by Polya in 1945 [619]. The simplex algorithm, created by G. Dantzig in 1947, can be seen as a local search algorithm for linear programming problems. J. Edmonds was first to present the greedy heuristic in the combinatorial optimization literature in 1971 [237]. The original references of the following metaheuristics are based on their application to optimization and/or machine learning problems: ACO (ant colonies optimization) [215], AIS (artificial immune systems) [70,253], BC (bee colony) [689,835], CA (cultural algorithms) [652], CEA (coevolutionary algorithms) [375,397], CMA-ES (covariance matrix

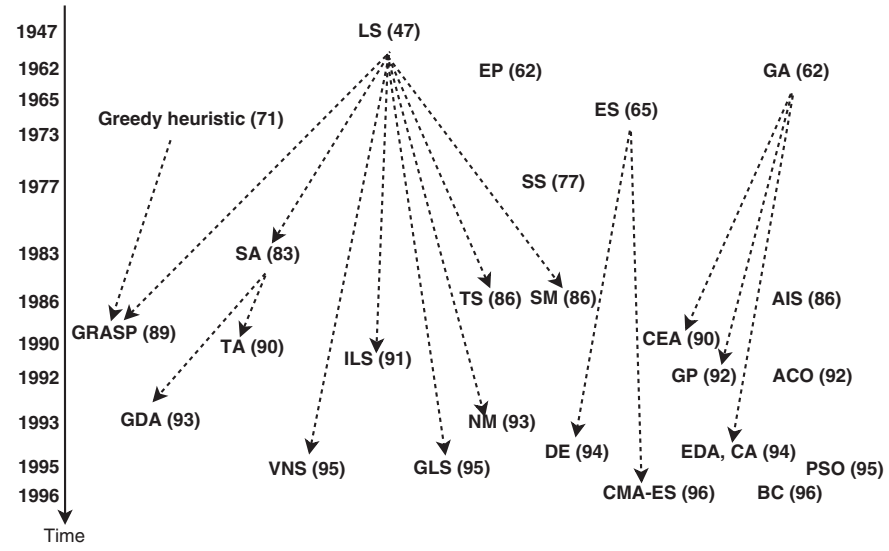

FIGURE 1.8 Genealogy of metaheuristics. The application to optimization and/or machine learning is taken into account as the original date.

adaptation evolution strategy) [363], DE (differential evolution) [626,724], EDA (estimation of distribution algorithms) [47], EP (evolutionary programming) [272], ES (evolution strategies) [642,687], GA (genetic algorithms) [383,384], GDA (great deluge) [229], GLS (guided local search) [805,807], GP (genetic programming) [480], GRASP (greedy adaptive search procedure) [255], ILS (iterated local search) [531], NM (noisy method) [124], PSO (particle swarm optimization) [457], SA (simulated annealing) [114,464], SM (smoothing method) [326], SS (scatter search) [320], TA (threshold accepting) [228], TS (tabu search) [322,364], and VNS (variable neighborhood search) [561].

In designing a metaheuristic, two contradictory criteria must be taken into account: exploration of the search space (diversification) and exploitation of the best solutions found (intensification) (Fig. 1.9). Promising regions are determined by the obtained "good" solutions. In intensification, the promising regions are explored more thoroughly in the hope to find better solutions. In diversification, nonexplored regions

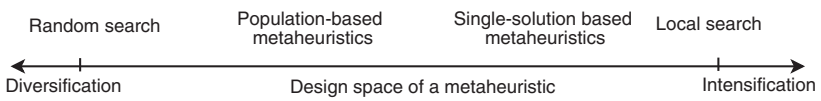

FIGURE 1.9 Two conflicting criteria in designing a metaheuristic: exploration (diversification) versus exploitation (intensification). In general, basic single-solution based metaheuristics are more exploitation oriented whereas basic population-based metaheuristics are more exploration oriented.

must be visited to be sure that all regions of the search space are evenly explored and that the search is not confined to only a reduced number of regions. In this design space, the extreme search algorithms in terms of the exploration (resp. exploitation) are random search (resp. iterative improvement local search). In random search, at each iteration, one generates a random solution in the search space. No search memory is used. In the basic steepest local search algorithm, at each iteration one selects the best neighboring solution that improves the current solution.

Many classification criteria may be used for metaheuristics:

- **Nature inspired versus nonnature inspired:** Many metaheuristics are inspired by natural processes: evolutionary algorithms and artificial immune systems from biology; ants, bees colonies, and particle swarm optimization from swarm intelligence into different species (social sciences); and simulated annealing from physics.
- **Memory usage versus memoryless methods:** Some metaheuristic algorithms are memoryless; that is, no information extracted dynamically is used during the search. Some representatives of this class are local search, GRASP, and simulated annealing. While other metaheuristics use a memory that contains some information extracted online during the search. For instance, short-term and long-term memories in tabu search.
- **Deterministic versus stochastic:** A deterministic metaheuristic solves an optimization problem by making deterministic decisions (e.g., local search, tabu search). In stochastic metaheuristics, some random rules are applied during the search (e.g., simulated annealing, evolutionary algorithms). In deterministic algorithms, using the same initial solution will lead to the same final solution, whereas in stochastic metaheuristics, different final solutions may be obtained from the same initial solution. This characteristic must be taken into account in the performance evaluation of metaheuristic algorithms.
- **Population-based search versus single-solution based search:** Single-solution based algorithms (e.g., local search, simulated annealing) manipulate and transform a single solution during the search while in population-based algorithms (e.g., particle swarm, evolutionary algorithms) a whole population of solutions is evolved. These two families have complementary characteristics: single-solution based metaheuristics are exploitation oriented; they have the power to intensify the search in local regions. Population-based metaheuristics are exploration oriented; they allow a better diversification in the whole search space. In the next chapters of this book, we have mainly used this classification. In fact, the algorithms belonging to each family of metaheuristics share many search mechanisms.
- **Iterative versus greedy:** In iterative algorithms, we start with a complete solution (or population of solutions) and transform it at each iteration using some search operators. Greedy algorithms start from an empty solution, and at each step a decision variable of the problem is assigned until a complete solution is obtained. Most of the metaheuristics are iterative algorithms.

1.3.4 Greedy Algorithms

In greedy or constructive algorithms[23], we start from scratch (empty solution) and construct a solution by assigning values to one decision variable at a time, until a complete solution is generated.

In an optimization problem, where a solution can be defined by the presence/absence of a finite set of elements $E = \{e_1, e_2, \ldots, e_n\}$, the objective function may be defined as $f : 2^E \rightarrow \mathbb{R}$, and the search space is defined as $F \subset 2^E$. A partial solution s may be seen as a subset $\{e_1, e_2, \ldots, e_k\}$ of elements e_i from the set of all elements E. The set defining the initial solution is empty. At each step, a local heuristic is used to select the new element to be included in the set. Once an element e_i is selected to be part of the solution, it is never replaced by another element. There is no backtracking of the already taken decisions. Typically, greedy heuristics are deterministic algorithms. Algorithm 1.2 shows the template of a greedy algorithm.

Algorithm 1.2 Template of a greedy algorithm.

$s = \{\}$; /* Initial solution (null) */
Repeat
  e_i = Local-Heuristic($E \backslash \{e / e \in s\}$) ;
  /* next element selected from the set E minus already selected elements */
  If $s \cup e_i \in F$ **Then** /* test the feasibility of the solution */
    $s = s \cup e_i$;
Until Complete solution found

Greedy algorithms are popular techniques as they are simple to design. Moreover, greedy algorithms have in general a reduced complexity compared to iterative algorithms. However, in most of optimization problems, the local view of greedy heuristics decreases their performance compared to iterative algorithms.

The main design questions of a greedy method are the following:

- **The definition of the set of elements:** For a given problem, one has to identify a solution as a set of elements. So, the manipulated partial solutions may be viewed as subsets of elements.
- **The element selection heuristic:** At each step, a heuristic is used to select the next element to be part of the solution. In general, this heuristic chooses the best element from the current list in terms of its contribution in minimizing locally the objective function. So, the heuristic will calculate the *profit* for each element. Local optimality does not implicate a global optimality. The heuristic may be static or dynamic. In static heuristics, the profits associated with the elements do not change, whereas in dynamic heuristics, the profits are updated at each step.

[23] Also referred to as successive augmentation algorithms.

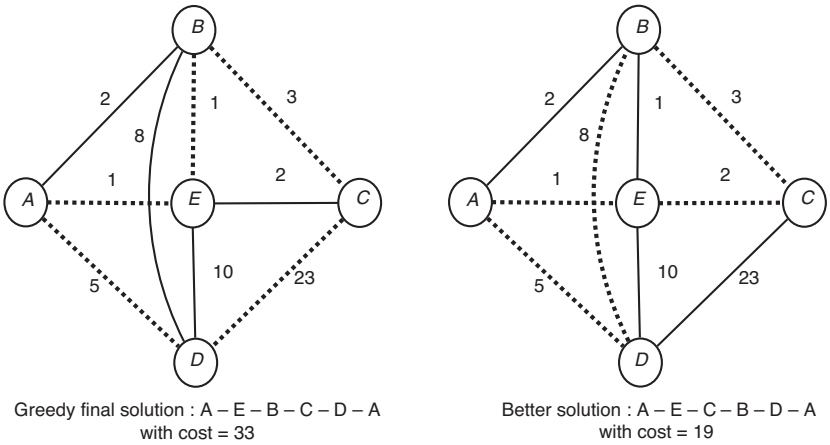

FIGURE 1.10 Illustrating a greedy algorithm for the TSP using a static heuristic. An element is associated with an edge of the graph, and the local heuristic consists in choosing the nearest neighbor. The obtained solution is $(A - E - B - C - D - A)$ with a total cost of 33, whereas a better solution with a cost of 19 is given (right).

Example 1.18 Static greedy algorithm for the TSP. In the TSP problem, the set E is defined by the set of edges. The set F of feasible solutions is defined by the subsets of 2^E that forms Hamiltonian cycles. Hence, a solution can be considered as a set of edges. A heuristic that can be used to select the next edge may be based on the distance. One possible local heuristic is to select the nearest neighbor. Figure 1.10 (left) illustrates the application of the nearest-neighbor greedy heuristic on the graph beginning from the node A. The local heuristic used is static; that is, the distances of the edges are not updated during the constructive process.

Greedy heuristics can be designed in a natural manner for many problems. Below are given some examples of well-known problems.

Example 1.19 Greedy algorithm for the knapsack problem. In the knapsack problem, the set E is defined by the set of objects to be packed. The set F represents all subsets of E that are feasible solutions. A local heuristic that can be used to solve the problem consists in choosing the object minimizing the ratio w_i/u_i where w_i (resp. u_i) represents the weight (resp. utility) of the object i. Figure 1.11 illustrates this greedy heuristic for a given instance of the knapsack problem.

Example 1.20 Greedy algorithm for the minimum spanning tree problem. There is a well-known optimal greedy algorithm for the spanning tree problem, the Kruskal algorithm. The minimum spanning tree problem belongs to class P, in terms of complexity. Given a connected graph $G = (V, E)$. With each edge $e \in E$ is associated a cost c_e. The problem is to find a spanning tree $T = (V, T)$ in graph G that minimizes the total cost $f(T) = \sum_{e \in T} c_e$. For this problem, set E is defined by the edges and set F is defined by all subsets of E that are trees. The local heuristic used consists in choosing first the least costly edges. In case of equality, an edge is randomly

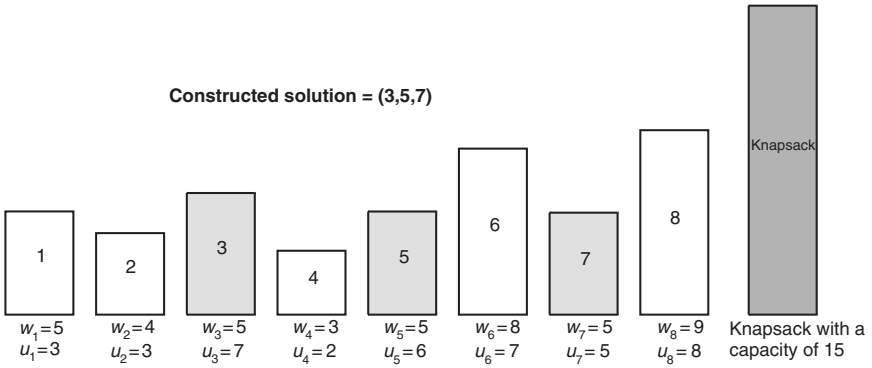

FIGURE 1.11 Illustrating a greedy algorithm for the knapsack problem. An element is associated with an object, and the local heuristic consists in choosing an element minimizing the ratio w_i/u_i. The final solution may not be optimal.

picked. Figure 1.12 illustrates this greedy heuristic for a given instance of the spanning tree problem. This algorithm always generates optimal solutions. Its time complexity is $O(m \cdot \log(m))$ where m represents the number of edges of the graph.

Greedy heuristics are in general myopic in their construction of a solution. Some greedy heuristics (e.g., pilot method) include look-ahead features where the future consequences of the selected element are estimated [38,803].

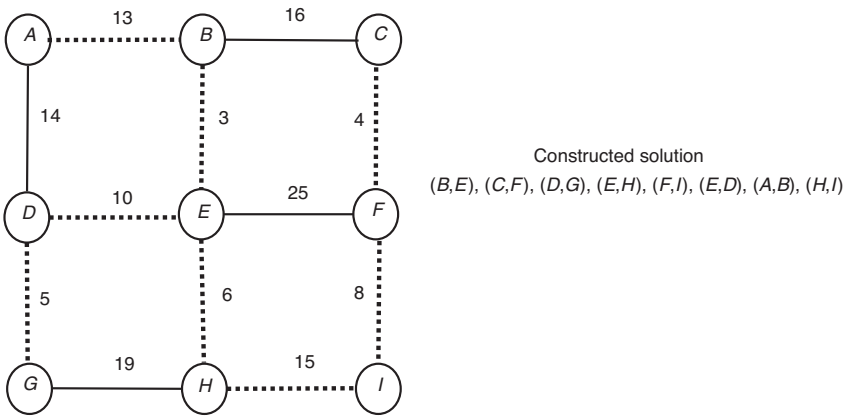

FIGURE 1.12 Illustrating a greedy algorithm for the spanning tree problem. The edge (A, D) has not been selected even if it is less costly than the edge (H, I) because it generates a nonfeasible solution (a cycle).

1.3.5 When Using Metaheuristics?

This section addresses the legitimate in using metaheuristics to solve an optimization problem. The complexity of a problem gives an indication on the hardness of the problem. It is also important to know the *size* of input instances the algorithm is supposed to solve. Even if a problem is NP-hard, small instances may be solved by an exact approach. Moreover, the *structure* of the instances plays an important role. Some medium- or even large-size instances with a specific structure may be solved in optimality by exact algorithms. Finally, the required search time to solve a given problem is an important issue in the selection of an optimization algorithm.

It is unwise to use metaheuristics to solve problems where efficient exact algorithms are available. An example of this class of problems is the P class of optimization problems. In the case where those exact algorithms give "acceptable" search time to solve the target instances, metaheuristics are useless. For instance, one should not use a metaheuristic to find a minimum spanning tree or a shortest path in a graph. Known polynomial-time exact algorithms exist for those problems.

Hence for easy optimization problems, metaheuristics are seldom used. Unfortunately, one can see many engineers and even researchers solving polynomial optimization problems with metaheuristics! So the first guideline in solving a problem is to analyze first its complexity. If the problem can be reduced to a classical or an already solved problem in the literature, then get a look at the state-of-the art best known optimization algorithms solving the problem. Otherwise, if there are related problems, the same methodology must be applied.

Example 1.21 Metaheuristics and LP continuous models. Polynomial-time problems such as linear programming models are very easy to solve with actual commercial (e.g., CPLEX, Lindo, XPRESS-MP, OSL) or free solvers (e.g., LP-solver) that are based on the simplex or interior methods. Some large-scale linear continuous problems having hundreds of thousands variables can be solved by those solvers using advanced algorithms such as the efficient manipulation of sparse matrices. However, for some very large polynomial problems or some specific problem structures, we may need the use of heuristics even if the complexity of this class of problems is polynomial. In an LP model, the number of vertices (extreme points) of the polytope representing the feasible region may be very large. Let us consider the $n \times n$ assignment problem, which includes $2n$ linear constraints and n^2 nonnegativity constraints. The polytope is composed of $n!$ vertices!

Even for polynomial problems, it is possible that the power of the polynomial function representing the complexity of the algorithm is so large that real-life instances cannot be solved in a reasonable time (e.g., a complexity of $O(n^{5000})$). In addition to the complexity of the problem, the required search time to solve the problem is another important parameter to take into account. Indeed, even if the problem is polynomial, the need of using metaheuristic may be justified for real-time search constraints.

Example 1.22 Real-time metaheuristics for polynomial dynamic problems. As an example of justifying the use of metaheuristics for polynomial problems, let us consider the shortest path in a graph of a real-life application that consists in finding a path between any two locations using GPS (Global Positioning System) technology. This graph has a huge number of nodes, and the search time is constrained as the customer has to obtain an answer in real time. In practice, even if this problem is polynomial, softwares in GPS systems are actually using heuristics to solve this problem. For those large instances, the use of polynomial algorithms such as the Dijkstra algorithm will be time consuming.

Many combinatorial optimization problems belong to the NP-hard class of problems. This high-dimensional and complex optimization class of problems arises in many areas of industrial concern: telecommunication, computational biology, transportation and logistics, planning and manufacturing, engineering design, and so on. Moreover, most of the classical optimization problems are NP-hard in their general formulation: traveling salesman, set covering, vehicle routing, graph partitioning, graph coloring, and so on [299].

For an NP-hard problem where state-of-the-art exact algorithms cannot solve the handled instances (size, structure) within the required search time, the use of metaheuristics is justified. For this class of problems, exact algorithms require (in the worst case) exponential time. The notion of "required time" depends on the target optimization problem. For some problems, an "acceptable" time may be equivalent to some seconds whereas for other problems it is equal to some months (production versus design problems). The fact that a problem is not in the P class does not imply that all large instances of the problem are hard or even that most of them are. The NP-completeness of a problem does not imply anything about the complexity of a particular class of instances that has to be solved.

Metaheuristics and IP/MIP problems: Despite the advances in reformulating IP and MIP models, and the development of new efficient procedures such as cutting planes and column generation, IP and MIP problems remain difficult to solve for moderate and large instances in a reasonable time. Let us notice that moderate and even large instances of some *structured* IP problems may be solved optimally.

Metaheuristics and CP: As for MIP models, constraint programming techniques enable to solve small instances of CP models in an optimal manner within a reasonable period of time. For very "tight" constrained problems, those strategies may solve moderate instances.

For nonlinear continuous (NLP) optimization, metaheuristics should be applied, if derivative-based methods, for example quasi-Newton method or conjugate gradient, fail due to a rugged search landscape (e.g., discontinuous, nonlinear, ill-conditioned, noisy, multimodal, nonsmooth, and nonseparable). The function f must be at least of moderate dimensionality (considerably greater than three variables). Those properties characterize many real-world problems. For easy problems, such as purely convex-quadratic functions, quasi-Newton method is typically faster by a factor of about 10 (in terms of computation time to attain a target value for the objective function) over one of the most efficient metaheuristics.

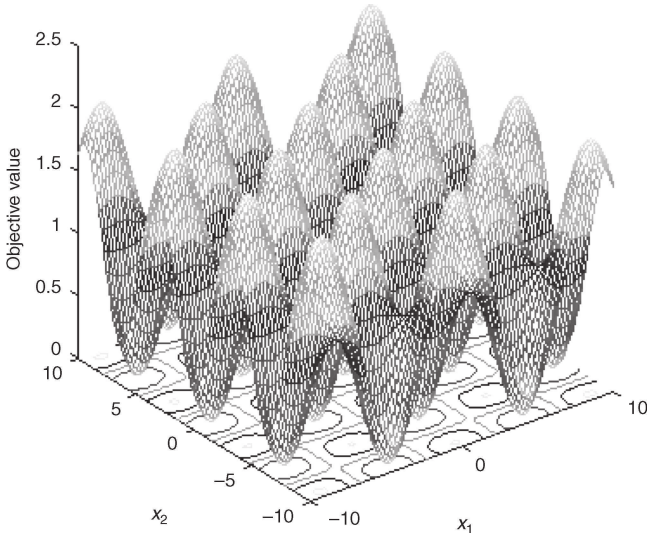

FIGURE 1.13 The Griewangk multimodal continuous function.

Example 1.23 Griewangk multimodal continuous function. This example shows a multimodal function to minimize the *Griewangk* function [774]:

$$f(\vec{x}) = 1 + \sum_{i=1}^{n} \frac{x_i^2}{4000} - \prod_{i=1}^{n} \cos\left(\frac{x_i}{\sqrt{i}}\right) \tag{1.1}$$

where $\vec{x} = \{x_1, x_2, \ldots, x_N\}$, with $x_i \in (-600, 600)$. Figure 1.13 illustrates the landscape associated with the function. The optimal solution for this function is the null vector $x^* = (0, \ldots, 0)$ with $f(x^*) = 0$.

Unlike mathematical programming, the main advantage of using metaheuristics is a restrictive assumption in formulating the model. Some optimization problems cannot be formulated with an unambiguous analytical mathematical notation. Indeed, the objective function may be a *black box* [448]. In a black box optimization, no analytical formulation of the objective exists (Fig. 1.14). Typical examples of optimization problems involving a black box scenario are shape optimization, model calibration (physical or biological), and parameter calibration.

Example 1.24 Optimization by simulation. Many problems in engineering such as in logistics, production, telecommunications, finance, or computational biology (e.g., structure prediction of proteins, molecular docking) are based on simulation to evaluate the quality of solutions. For instance, in risk analysis, Monte Carlo simulations are used

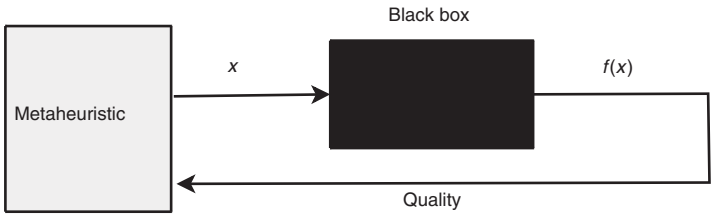

FIGURE 1.14 Black box scenario for the objective function.

to estimate the objective function of a portfolio investment that is represented by the average rate of return and its variance.

A function $f : X \longrightarrow R$ is called a black box function iff

- the domain X is known,
- it is possible to know f for each point of X according to a simulation, and
- no other information is available for the function f.

Very expensive experiments in terms of time and cost are associated with those problems. In general, a simulation must hold to evaluate the solution.

Another example of nonanalytical models of optimization is interactive optimization[24] that involves a human interaction to evaluate a solution. Usually, human evaluation is necessary when the form of the objective function is not known. Interactive optimization can concurrently accept evaluations from many users. Many real-life examples fit into the class of interactive optimization problems where the result should fit particular user preferences:

- Visual appeal or attractiveness [104,183].
- Taste of coffee [372] or color set of the user interface.
- Evolving images [705], 3D animated forms, music composition, various artistic designs, and forms to fit user aesthetic preferences [476,789].

Metaheuristics will gain more and more popularity in the future as optimization problems are increasing in size and in complexity. Indeed, complex problem models are needed to develop more accurate models for real-life problems in engineering and science (e.g., engineering design, computational biology, finance engineering, logistics, and transportation).

Let us summarize the main characteristics of optimization problems justifying the use of metaheuristics:

[24] Aesthetic selection in evolutionary algorithms.

- An easy problem (P class) with very large instances. In this case, exact polynomial-time algorithms are known but are too expensive due to the size of instances.
- An easy problem (P class) with hard real-time constraints (online algorithms). In real-time optimization problems, metaheuristics are widely used. Indeed, in this class of problems, we have to find a "good solution" online. Even if efficient exact algorithms are available to solve the problem, metaheuristics are used to reduce the search time. Dynamic optimization problems represent another example of such problems.
- A difficult problem (NP-hard class) with moderate size and/or difficult structures of the input instances.
- Optimization problems with time-consuming objective function(s) and/or constraints. Some real-life optimization problems are characterized by a huge computational cost of the objective function(s).
- Nonanalytic models of optimization problems that cannot be solved in an exhaustive manner. Many practical problems are defined by a black box scenario of the objective function.
- Moreover, those conditions may be amplified by nondeterministic models of optimization: problems with uncertainty and robust optimization. For some noisy problems, uncertainty and robustness cannot be modeled analytically. Some complex simulations (e.g., Monte Carlo) must be carried out that justify the use of metaheuristics. The ambiguity of the model does not encourage attempting to solve it with exact algorithms. As the data are fuzzy, this class of problems does not necessarily need the optimal solution to be found.

Example 1.25 Design versus control problems. The relative importance of the two main performance measures, quality of solutions and search time, depends on the characteristics of the target optimization problem. Two extreme problems may be considered here:

- **Design problems:** Design problems are generally solved once. They need a very good quality of solutions whereas the time available to solve the problem is important (e.g., several hours, days, months). In this class of problems, one can find the *strategic* problems (long-term problems), such as telecommunication network design and processor design. These problems involve an important financial investment; any imperfection will have a long-time impact on the solution. Hence, the critical aspect is the quality of solutions rather than the search time (Fig. 1.15). If possible, exact optimization algorithms must be used.
- **Control problems:** Control problems represent the other extreme where the problem must be solved frequently in real time. This class of *operational* problems involves short-term decisions (e.g., fractions of a second), such as routing messages in a computer network and traffic management in a city. For operational decision problems, very fast heuristics are needed; the quality of the solutions is less critical.

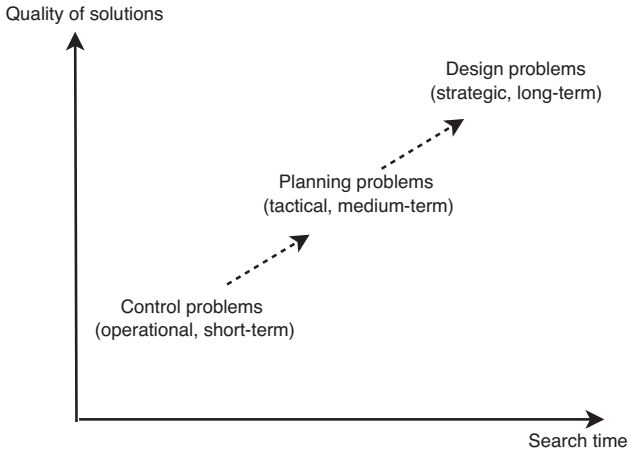

FIGURE 1.15 Different classes of problems in terms of the trade-off between quality of solutions and search time: design (strategic, long-term), planning (tactical, medium-term), control (operational, short-term).

Between these extremes, one can find an intermediate class of problems represented by planning problems and tactical problems (medium-term problems). In this class of problems, a trade-off between the quality of solution and the search time must be optimized. In general, exact optimization algorithms cannot be used to solve such problems.

The *development cost* of solving an optimization problem is also an important issue. Indeed, metaheuristics are easy to design and implement. Open-source and free software frameworks such as ParadisEO allow the efficient design and implementation of metaheuristics for monoobjective and multiobjective optimization problems, hybrid metaheuristics, and parallel metaheuristics. Reusing existing designs and codes will contribute to reducing the development cost.

1.4 MAIN COMMON CONCEPTS FOR METAHEURISTICS

There are two common design questions related to all iterative metaheuristics: the representation of solutions handled by algorithms and the definition of the objective function that will guide the search.

1.4.1 Representation

Designing any iterative metaheuristic needs an encoding (representation) of a solution[25]. It is a fundamental design question in the development of metaheuristics. The encoding plays a major role in the efficiency and effectiveness of any metaheuristic

[25]In the evolutionary computation community, the genotype defines the representation of a solution. A solution is defined as the phenotype.

and constitutes an essential step in designing a metaheuristic. The encoding must be suitable and relevant to the tackled optimization problem. Moreover, the efficiency of a representation is also related to the search operators applied on this representation (neighborhood, recombination, etc.). In fact, when defining a representation, one has to bear in mind how the solution will be evaluated and how the search operators will operate.

Many alternative representations may exist for a given problem. A representation must have the following characteristics:

- **Completeness:** One of the main characteristics of a representation is its completeness; that is, all solutions associated with the problem must be represented.
- **Connexity:** The connexity characteristic is very important in designing any search algorithm. A search path must exist between any two solutions of the search space. Any solution of the search space, especially the global optimum solution, can be attained.
- **Efficiency:** The representation must be easy to manipulate by the search operators. The time and space complexities of the operators dealing with the representation must be reduced.

Many straightforward encodings may be applied for some traditional families of optimization problems (Fig. 1.16). There are some classical representations that

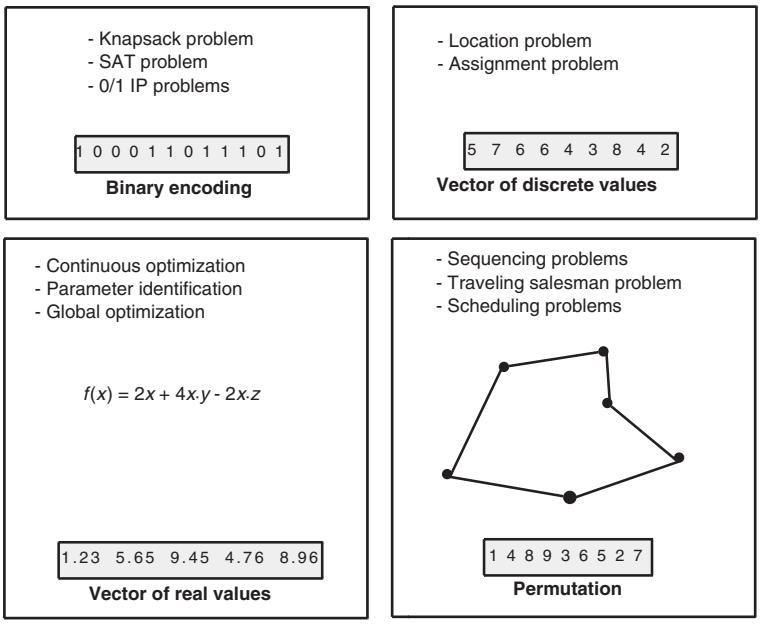

FIGURE 1.16 Some classical encodings: vector of binary values, vector of discrete values, vector of real values, and permutation.

are commonly used to solve a large variety of optimization problems. Those representations may be combined or underlying new representations. According to their structure, there are two main classes of representations: linear and nonlinear.

1.4.1.1 Linear Representations

Linear representations may be viewed as strings of symbols of a given alphabet.

In many classical optimization problems, where the decision variables denote the presence or absence of an element or a yes/no decision, a *binary encoding* may be used. For instance, satisfiability problems and $\{0, 1\}$-linear programs are representative of such problems. The binary encoding consists in associating a binary value for each decision variable. A solution will be encoded by a vector of binary variables.

Example 1.26 Binary encoding for knapsack problems. For a 0/1-knapsack problem of n objects, a vector s of binary variables of size n may be used to represent a solution:

$$\forall i,\ s_i = \begin{cases} 1 & \text{if object } i \text{ is in the knapsack} \\ 0 & \text{otherwise} \end{cases}$$

The binary encoding uses a binary alphabet consisting in two different symbols. It may be generalized to any *discrete values* based encoding using an n-ary alphabet. In this case, each variable takes its value over an n-ary alphabet. The encoding will be a vector of discrete values. This encoding may be used for problems where the variables can take a finite number of values, such as combinatorial optimization problems.

Example 1.27 Discrete encoding for generalized assignment problems. Many real-life optimization problems such as resource allocation may be reduced to assignment problems. Suppose a set of k tasks is to be assigned to m agents to maximize the total profit. A task can be assigned to any agent. A classical encoding for this class of problems may be based on a discrete vector s of size k, where $s[i]$ represents the agent assigned to the task i.

$$s[i] = j \quad \text{if the agent } j \text{ is assigned to task } i$$

Many sequencing, planning, and routing problems are considered as *permutation* problems. A solution for permutation problems, such as the traveling salesman problem and the permutation flow-shop scheduling problem, may be represented by a permutation $\pi = (\pi_1, \pi_2, \ldots, \pi_n)$. Any element of the problem (cities for the TSP, jobs for the FSP) must appear only once in the representation.

Example 1.28 Permutation encoding for the traveling salesman problem. For a TSP problem with n cities, a tour may be represented by a permutation of size n. Each permutation decodes a unique solution. The solution space is represented by the set of all permutations. Its size is $|S| = (n - 1)!$ if the first city of the tour is fixed.

Example 1.29 Reducing the representation space. In this example, we will see how a given problem can be made simpler by choosing a suitable representation. The N-Queens puzzle is the problem of putting N chess queens on an $N \times N$ chessboard such that none of them is able to capture any other using the standard chess queens moves. Any queen is assumed to be able to attack any other. The 8-Queens problem was originally defined by the chess player Max Bezzel in 1848.

A solution for this problem represents an assignment of the eight queens on the chessboard. First, let us encode a solution by a vector of eight Cartesian positions $x = (p_1, p_2, \ldots, p_8)$ where $p_i = (x_i, y_i)$ represents the Cartesian position of the queen i. The number of possibilities (size of the search space) is 64^8 that is over 4 billion solutions. If we prohibit more than one queen per row, so that each queen is assigned to a separate row, the search space will have 8^8 solutions that is over 16 million possibilities. Finally, if we forbid two queens to be both in the same column or row, the encoding will be reduced to a permutation of the n queens. This encoding will reduce the space to $n!$ solutions, which is only 40,320 possibilities for the 8-Queens problem. Figure 1.17 shows a solution for a 8-Queens problem. This example shows how the representation plays a major role in defining the space a given metaheuristic will have to explore.

For continuous optimization problems, the natural encoding is based on *real values*. For instance, this encoding is commonly used for nonlinear continuous optimization problems, where the most usual encoding is based on vectors of real values.

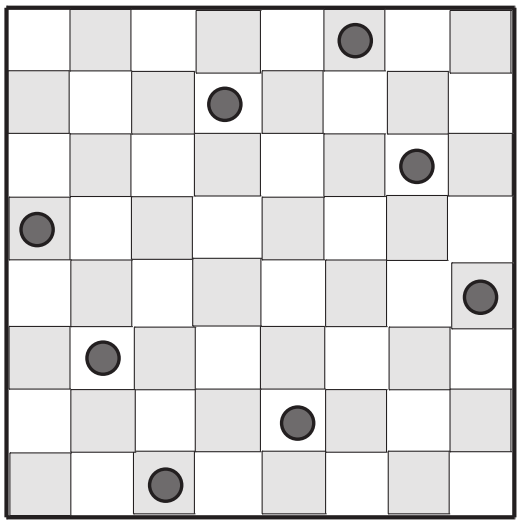

FIGURE 1.17 A solution for the 8-Queens problem represented by the permutation (6,4,7,1,8,2,5,3).

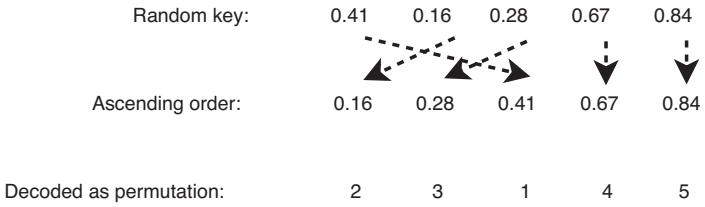

FIGURE 1.18 Random-key encoding and decoding.

Example 1.30 Mixed encodings in parameter optimization. Many optimization problems in engineering sciences consist in finding the best parameters in designing a given component. This class of problems is known as parameter optimization problems. Some parameters may be associated with real values while others are associated with discrete ones. Hence, a solution may be represented by a vector x of mixed values, where $x[i]$ represents the real or discrete value of parameter i. The size of the vector is equal to the number of parameters of the system.

Other "nontraditional" linear representations may be used. Some of them have been defined in the evolutionary computation community:

- **Random-key encoding:** The random-key representation uses real-valued encoding to represent permutations. Random-key encoding is useful for permutation-based representations, where the application of classical variation operators (e.g., crossover) presents feasibility problems [62]. In the random-key encoding, to each object is assigned a random number generated uniformly from [0,1[. The decoding is applied as follows: the objects are visited in an ascending order and each element is decoded by its rank in the sequence (Fig. 1.18).
- **Messy representations:** In linear representations of fixed length, the semantics of the values[26] is tied to its position in the string. In messy representations, the value associated with a variable is independent of its position [329]. Then, each element of the representation is a couple composed of the variable and its value. This encoding may have a *variable length*. It has been introduced to improve the efficiency of genetic operators by minimizing their disruption.
- **Noncoding regions:** Some representation may introduce noncoding regions (introns) in the representation [501,828]. This biological inspired representation has the form

$$x_1|\text{intron}|x_2|\cdots|\text{intron}|x_n$$

where x_i (resp. intron) represents the coding (resp. noncoding) part of the encoding. Noncoding regions are regions of the representation that provide no

[26] In evolutionary algorithms, the value is defined as an allele.

contribution to the objective (quality) of the solution. As in messy representations, this encoding has an impact on recombination search operators.

- **Diploid representations:** Diploid representations include multiple values for each position of the encoding. This representation requires a decoding procedure to determine which value will be associated with a given position. This encoding was first introduced for a quicker adaptation of solutions in solving dynamic cyclic problems [332].
- **Quantum representations:** In quantum computing systems, the smallest unit of information is the *qubit*. Unlike the classical bit, the qubit can be in the superposition of the two values at the same time. The state of a qubit can be represented as

$$|\Psi\rangle = \alpha|0\rangle + \beta|1\rangle$$

where $|\Psi\rangle$ denotes a function wave in Hilbert space, $|0\rangle$ and $|1\rangle$ represent, respectively, the classical bit values 0 and 1, and α and β are complex numbers that satisfy the probability amplitudes of the corresponding states[27]. If a superposition is measured with respect to the basis $\{|0\rangle, |1\rangle\}$, the probability to measure $|0\rangle$ is α^2 and the probability to measure $|1\rangle$ is β^2 [818].
A quantum encoding of n qubits can represent 2^n states at the same time. This means that one can represent an exponential amount of information. Using this probabilistic binary-based representation, one needs to design a decoder to generate and evaluate solutions [356].

Solutions in some optimization problems are encoded by *mixed representations*. The most popular mixed representation is the continuous/integer one, to solve MIP problems. Here, a solution is represented by a vector of mixed values (reals, binary values, and integers).

In *control problems*, decision variables represent values that are control variables taken in time (or frequency). In general, a discretization of time domain is realized. The representation generally used is $x = (c(t_1), \ldots, c(t_i), \ldots)$, where $c(t_i)$ represents the value of the control variable x at time t. If the sequence of values is monotonic, the following incremental representation may be used:

$$x_1 = c(t_1), x_i = (c(t_i) - c(t_{i-1}))$$

1.4.1.2 Nonlinear Representations

Nonlinear encodings are in general more complex structures. They are mostly based on graph structures. Among the traditional nonlinear representations, trees are the most used.

The tree encoding is used mainly for hierarchical structured optimization problems. In tree encoding, a solution is represented by a tree of some objects. For instance,

[27] $\alpha^2 + \beta^2 = 1$.

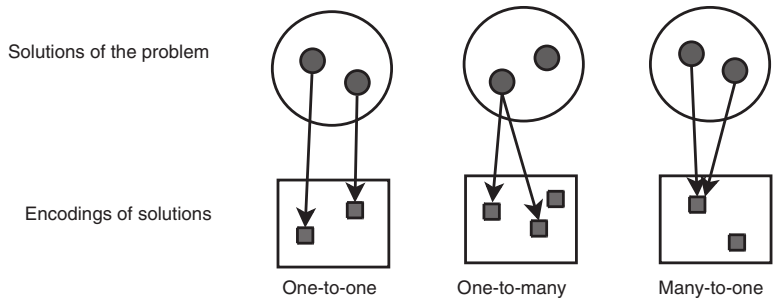

FIGURE 1.19 Mapping between the space of solutions and the space of encodings.

this structure is used in genetic programming. The tree may encode an arithmetic expression, a first-order predicate logic formula, or a program.

Example 1.31 Tree encoding for regression problems. Given some input and output values, regression problems consist in finding the function that will give the best (closest) output to all inputs. Any *S*-expression can be drawn as a tree of functions and terminals. These functions and terminals may be defined in various manners. For instance, the functions may be add, sub, sine, cosine, and so on. The terminals (leaves) represent constants or variables.

Other nonlinear representations can be used such as finite-state machines and graphs.

1.4.1.3 Representation-Solution Mapping The representation-solution mapping function transforms the encoding (genotype) to a problem solution (phenotype). The mapping between the solution space and the encoding space involves three possibilities [251] (Fig. 1.19):

- **One-to-one:** This is the traditional class of representation. Here, a solution is represented by a single encoding and each encoding represents a single solution. There is no redundancy and no reduction of the original search space. For some constrained optimization problems, it is difficult to design such one-to-one mapping.
- **One-to-many:** In the one-to-many mapping, one solution may be represented by several encodings. The redundancy of the encoding will enlarge the size of the search space and may have an impact on the effectiveness of metaheuristics.

Example 1.32 Symmetry in partitioning problems. Partitioning problems, or clustering or grouping problems, represent an important class of problems. Problems such as clustering in data mining, graph partitioning problems (GPP), graph

TABLE 1.6 Partitioning Problems with Their Associated Constraints and Objective Functions

Problem	Constraint	Objective
Graph coloring	Adjacent nodes do not have the same color	Min. number of colors
Bin packing	Sum of elements sizes in any group is less than C	Min. number of groups
Data clustering	Fixed number of clusters	Max. intercluster distance
Graph partitioning	Groups of equal size	Min. number of edges between partitions
Assembly line balancing	Cycle time	Min. number of workstations

All of them are NP-hard problems.

coloring problem (GCP), and bin packing are well-known examples of grouping problems [251]. Grouping problems consist in partitioning a set S of elements into mutually disjoint subsets s_i, where $\cup s_i = S$ and $s_i \cap s_j = \oslash$. The different grouping problems differ in their associated constraints and the objective function to optimize (see Table 1.6).

A straightforward representation associates with each element its group. For instance, the encoding $BAAB$ assigns the first element to group B, the second to group A, the third element to group A, and the last one to group B. The first and the last elements (resp. second and third) are then assigned to the same group. The encoding $ABBA$ represents the same solution. Hence, this representation belongs to the one-to-many class of encodings and is highly redundant. The number of different representations encoding the same solution grows exponentially with the number of partitions.

- **Many-to-one:** In this class, several solutions are represented by the same encoding. In general, those encodings are characterized by a lack of details in the encoding; some information on the solution is not explicitly represented. This will reduce the size of the original search space. In some cases, this will improve the efficiency of metaheuristics. This class of representation is also referred to as *indirect encoding*.

1.4.1.4 Direct Versus Indirect Encodings

When using an indirect representation, the encoding is not a complete solution for the problem. A *decoder* is required to express the solution given by the encoding. According to the information that is present in the indirect encoding, the decoder has more or less work to be able to derive a complete solution. The decoder may be *nondeterministic*. Indirect encodings are popular in optimization problems dealing with many constraints such as scheduling problems. For instance, the constraints associated with the optimization problem are handled by the decoder and will guarantee the validity of the solution that is derived.

Example 1.33 Indirect encodings for the job-shop scheduling problem (JSP). The simple job-shop scheduling problem may be defined as follows. Given a set of j jobs. Each job is composed of M operations to be realized on M machines. Each operation must be realized on a single machine. Each job has an operation that has to be performed on each machine. A schedule indicates at each time slot and on each machine, the operation being processed. The objective function to minimize is the *makespan* (total completion time). Let us denote $E_m(x)$ as the completion time of the last operation performed on machine m according to the schedule x. Then, the makespan can be defined as $C_{\max}(x) = \max_{1 \leq m \leq M} E_m(x)$. The following constraints must be fulfilled: a machine can perform only one operation at a time, the operations should be performed in a predefined order (the operations of a job cannot be executed concurrently on two different machines), and there is only one machine that is able to perform any operation.

A solution should represent a feasible schedule of occupation of the machines that indicates for each time slot of any machine if it is free or which operation of which job is being performed. A direct representation may be defined as the list of machines and the time slots that are used to perform the operations (Fig. 1.20a). For instance, the job i is composed of the operations Op7 and Op3. The operation Op7 is executed on machine m2 from time 1 to 3. The operation Op3 is performed on machine m3 from time 13 to 17, and so on. The assignment of an operation consists in the association of a machine and time slot taking in consideration precedence constraints. The order of execution of operations is defined at the level of operations.

Various indirect representations may be designed for the JSP problem. Such an indirect representation may be simply a permutation of j jobs (Fig. 1.20b). The search space is limited to the set of permutations of j integers, that is, of size $j!$. An encoding mechanism is used to transform the permutation into a complete and feasible schedule. The decoder has a very limited set of information and shall derive much more to obtain valid schedules. Various decoders may be imagined, with a variable degree of stochasticity. A very simple one would consider the permutation as a priority list and would derive a schedule that always gives priority to operations belonging to the highest priority jobs.

A second more rich indirect encoding is an array of $J \times M$ entries. Each job is assigned a class of markers, all associated with one job having the same tag (job number). The markers are then shuffled in the array. Figure 1.20c illustrates such an

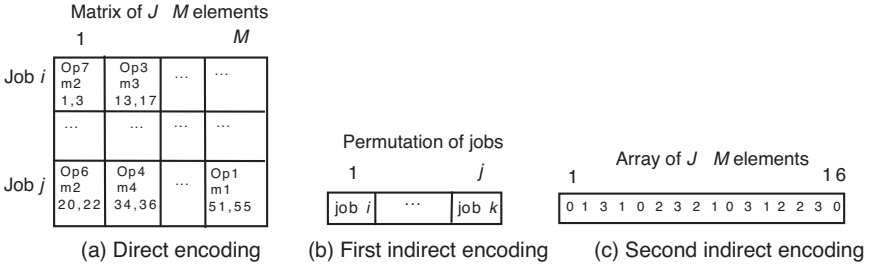

FIGURE 1.20 Direct and indirect encodings for the job-shop scheduling problem.

encoding for a 4×4 JSP. The decoder considers this array from the "left to the right." For each entry, it then schedules as soon as possible the next not-yet scheduled operation of the job associated with the marker that has been found.

Let us notice that the representation has an interaction with search operators and the objective function. Then, finding a suitable representation cannot be completely done without the specification of the search operators and the objective function. For instance, in the relationship between the representation and the search operators, an ideal encoding should have the *proximity* property: similar solutions in terms of their representations (genotypes) must be similar in the phenotype space. The similarity is defined relatively to the performed search operators.

Example 1.34 Encoding of real numbers. Let us consider an encoding for real numbers based on binary vectors. In many evolutionary algorithms such as genetic algorithms, this encoding is chosen to solve continuous optimization problems. Let us consider two consecutive integers, 15 and 16. Their binary representation is, respectively, 01111 and 10000. In the phenotype space, 15 is neighbor to 16, while in the genotype space, 5 bits must be flipped to obtain 16 from 15! Using variation operators based on the flip operator, this disparity between the genotype and the phenotype spaces may generate nonefficient metaheuristics. Gray code encoding solves this problem by mapping two neighbors in the genotype space (one-flip operation) to neighbors in the phenotype space. The main drawback of gray codes is still their ability to deal with dynamic ranges.

1.4.2 Objective Function

The objective function[28] f formulates the goal to achieve. It associates with each solution of the search space a real value that describes the quality or the fitness of the solution, $f : S \rightarrow \mathbb{R}$. Then, it represents an absolute value and allows a complete ordering of all solutions of the search space. As shown in the previous section, from the representation space of the solutions R, some decoding functions d may be applied, $d : R \rightarrow S$, to generate a solution that can be evaluated by the function f.

The objective function is an important element in designing a metaheuristic. It will guide the search toward "good" solutions of the search space. If the objective function is improperly defined, it can lead to nonacceptable solutions whatever metaheuristic is used.

1.4.2.1 Self-Sufficient Objective Functions

For some optimization problems, the definition of the objective function is straightforward. It specifies the originally formulated objective function.

Example 1.35 Straightforward guiding objective function. In many routing problems such as TSP and vehicle routing problems, the formulated objective is to minimize

[28] Also defined as the cost function, evaluation function, and utility function.

a given global distance. For instance, in the TSP, the objective corresponds to the total distance of the Hamiltonian tour:

$$f(s) = \sum_{i=1}^{n-1} d_{\pi(i),\pi(i+1)} + d_{\pi(n),\pi(1)}$$

where π represents a permutation encoding a tour and n the number of cities.

For continuous (linear and nonlinear) optimization problems, the guiding function to optimize by a metaheuristic is simply the target objective function. In those families of optimization problems, the guiding objective function used in the search algorithm is generally equal to the objective function that has been specified in the problem formulation.

1.4.2.2 Guiding Objective Functions For other problems, the definition of the objective function is a difficult task and constitutes a crucial question. The objective function has to be transformed for a better convergence of the metaheuristic. The new objective function will guide the search in a more efficient manner.

Example 1.36 Objective function to satisfiability problems. Let us formulate an objective function to solve satisfiability problems. SAT problems represent fundamental decision problems in artificial intelligence. The k-SAT problem can be defined as follows: given a function F of the propositional calculus in a conjunctive normal form (CNF). The function F is composed of m clauses C_i of k Boolean variables, where each clause C_i is a disjunction. The objective of the problem is to find an assignment of the k Boolean variables such as the value of the function F is true. Hence, all clauses must be satisfied.

$$F = (x_1 \vee \overline{x_4}) \wedge (\overline{x_1} \vee \overline{x_2} \vee x_3) \wedge (x_1 \vee x_3 \vee x_4) \wedge (\overline{x_1} \vee x_2) \wedge (x_1 \vee x_2 \vee x_4) \wedge (x_2 \vee \overline{x_4}) \wedge (\overline{x_2} \vee \overline{x_3})$$

A solution for the problem may be represented by a vector of k binary variables. A straightforward objective function is to use the original F function:

$$f = \begin{cases} 0 & \text{if is } F \text{ false} \\ 1 & \text{otherwise} \end{cases}$$

If one considers the two solutions $s_1 = (1, 0, 1, 1)$ and $s_2 = (1, 1, 1, 1)$, they will have the same objective function, that is, the 0 value, given that the function F is equal to *false*. The drawback of this objective function is that it has a poor differentiation between solutions. A more interesting objective function to solve the problem will be to count the number of satisfied clauses. Hence, the objective will be to maximize the number of satisfied clauses. This function is better in terms of guiding the search toward the optimal solution. In this case, the solution s_1 (resp. s_2) will have a value of 5 (resp. 6). This objective function leads to the `MAX-SAT` model.

1.4.2.3 Representation Decoding The design questions related to the definition of the representation and the objective function may be related. In some problems, the representation (genotype) is decoded to generate the best possible solution (phenotype). In this situation, the mapping between the representation and the objective function is not straightforward; that is, a decoder function must be specified to generate from a given representation the best solution according to the objective function.

Example 1.37 Let us illustrate the relationship between the representation and the objective function within the Steiner tree problem. It is a combinatorial optimization problem with many applications in telecommunication (network design) and biology (phylogenetics). Given a nonoriented weighted graph $G = (V, E)$ where V represents the nodes of the graph and E represents the edges of the graph. The weights associated with the edges are all positive. Let T be a subset of vertices identified as terminals. The goal is to find a minimum-weight connected subgraph that includes all the terminals. The resulting subgraph is obviously a tree. This problem is NP-hard whereas the minimum-weight spanning tree problem is polynomial; that is, the Kruskal or Prim algorithms are well-known efficient algorithms to solve the problem.

A solution of the Steiner tree problem may be characterized by the list of nonterminal nodes X. It is then represented by a vector of binary values. The size of the vector is the number of nonterminal nodes. The Steiner tree associated with a solution is equivalent to the minimum spanning tree of the set $T \cup X$. This is easily obtained by a polynomial algorithm such as the Kruskal algorithm. Figure 1.21 represents an example on an input graph instance, where the terminals are represented by $T = \{A, B, C, D\}$. The optimal solution is $s^* = \{1, 3, 5\}$, which is represented in Fig. 1.22.

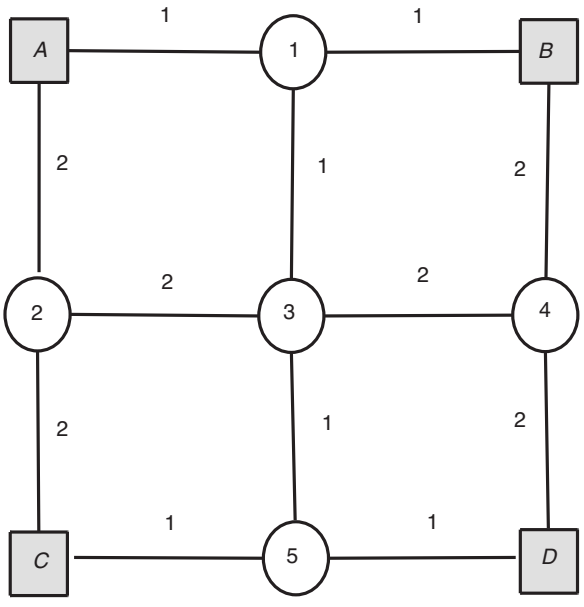

FIGURE 1.21 Instance for the Steiner tree problem: $T = \{A, B, C, D\}$.

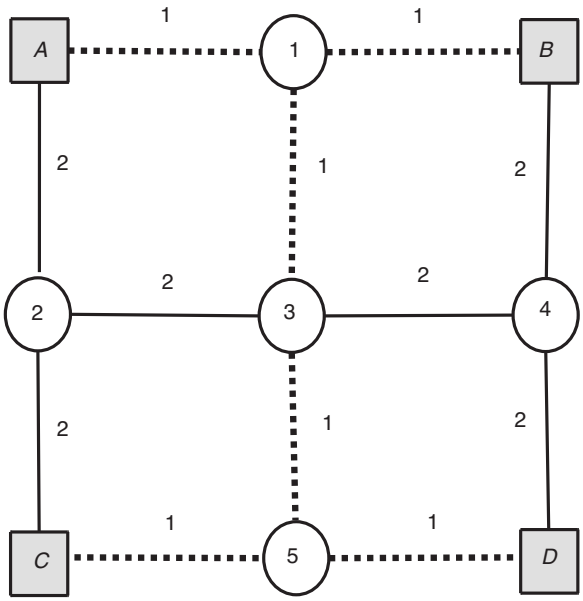

FIGURE 1.22 Optimal solution represented by the set {1, 3, 5}.

1.4.2.4 Interactive Optimization In interactive optimization, the user is involved online in the loop of a metaheuristic. There are two main motivations for designing interactive metaheuristics:

- **The user intervention to guide the search process:** In this case, the user can interact with the search algorithm to converge faster toward promising regions. The objective here is to improve the search process online by introducing dynamically some user knowledge. For example, the user can suggest some promising solutions from a graphical representation of solutions. The user can also suggest the update of the metaheuristic parameters.
  This strategy is widely used in multicriteria decision making in which an interaction is performed between the decision maker and the solver to converge toward the best compromise solution (see Chapter 4).
- **The user intervention to evaluate a solution:** Indeed, in many design problems, the objective function requires subjective evaluation depending on human preferences (e.g., taste of coffee). For some problems, the objective function cannot be formulated analytically (e.g., visual appeal or attractiveness). This strategy is widely used in art design (music, images, forms, etc.) [745]. Some applications may involve many users.

In designing a metaheuristic, the limited number of carried evaluations must be taken into account. Indeed, multiple evaluations of solutions will cause the user

fatigue. Moreover, the evaluation by the user of a solution may be slow and expensive. Hence, the metaheuristic is supposed to converge toward a good solution in a limited number of iterations and using a limited size of population if a population-based metaheuristic is used.

1.4.2.5 Relative and Competitive Objective Functions

In some problems, it is impossible to have an objective function f that associates an absolute value with all solutions. For instance, those problems arise in game theory [175], cooperative or competitive coevolution [620], and learning classifier systems [821]. For instance, in a game the strategy A may be better than B, B better than C, and C better than A.

There are two alternatives to this class of problems: using relative or competitive objective functions. The relative fitness associates a rank with the individual in the population. In competitive fitness, a competition is applied over a subpopulation of solutions. Three different types of competition can be used: bipartite, tournament, and full. Bipartite competition compares two solutions s_1 and s_2 to determine the better one, whereas in case of full competition, all solutions are considered.

Population-based metaheuristics are well suited to this situation. In fact, in population-based metaheuristics, the selection strategies need only the relative or competitive fitness. Hence, the absolute quality of a solution is not necessary to evolve the population.

1.4.2.6 Meta-Modeling

It is well known that most of the time, in metaheuristics, the time-intensive part is the evaluation of the objective function. In many optimization problems, the objective function is quite costly to compute. The alternative to reduce this complexity is to approximate the objective function and then replace the original objective function by its approximation function. This approach is known as *meta-modeling*[29]. Moreover, for some problems, an analytical objective function is not available. In this case, the objective function may be approximated using a sample of solutions generated by physical experiments or simulations.

Example 1.38 Extremely expensive objective functions. A classical example of extremely expensive objective function deals with structural design optimization [52,341]. For instance, in a three-dimensional aerodynamic design optimization, the evaluation of a structure consists in executing a costly CFD (computational fluid dynamics) simulation. A single simulation may take more than 1 day even on a parallel machine. In some problems such as telecommunication network design [752] or molecular docking [766], a time-consuming simulation must be used to evaluate the quality of the generated solutions. It is unimaginable to conduct physical

[29] Also known as surrogates or fitness approximation.

experiments for each potential solution. Moreover, meta-modeling is important in stochastic and robust optimization where additional objective function evaluations are necessary.

Many meta-modeling techniques may be employed for expensive objective functions. They are based on constructing an approximate model from a properly selected sample of solutions:

- **Neural networks:** Neural Network models such as multilayer perceptrons [644] and radial basis function [622] are the commonly used strategies.
- **Response surface methodologies:** This class is based on polynomial approximation of the objective function to create a response surface [571]. The most known methods belonging to this class are the least square method (quadratic polynomials) of Box and Wilson [90] and design of experiments (DOE) of Taguchi [668].
- Other candidate models in approximating objective functions are Kriging models [164], DACE (design and analysis of computer experiments) [679], Gaussian processes [316], and machine learning techniques such as SVM (support vector machines) [791].

The model selection problem is far from being a simple one [101]. There is a trade-off between the complexity of the model and its accuracy. To solve the classical bias and variance dilemma, the use of multiple models is encouraged. Constructing multiple local models instead of a global model can also be beneficial. The reader may refer to Ref. [413] for a more comprehensive survey.

Once the meta-model is constructed, it can be used in conjunction with the original objective function [413]. An alternative use of the original model and the approximated one can also be realized using different management strategies of meta-models[30] (Fig. 1.23) [208]. The trade-off here is the use of an expensive accurate evaluation versus a cheap erroneous evaluation of the objective function.

1.5 CONSTRAINT HANDLING

Dealing with constraints in optimization problems is another important topic for the efficient design of metaheuristics. Indeed, many continuous and discrete optimization problems are constrained, and it is not trivial to deal with those constraints. The constraints may be of any kind: linear or nonlinear and equality or inequality constraints. In this section, constraint handling strategies, which mainly act on the representation of solutions or the objective function, are presented. They can be classified as reject strategies, penalizing strategies, repairing strategies, decoding strategies, and preserving strategies. Other constraint handling approaches using search components

[30] Known as evolution control in evolutionary algorithms.

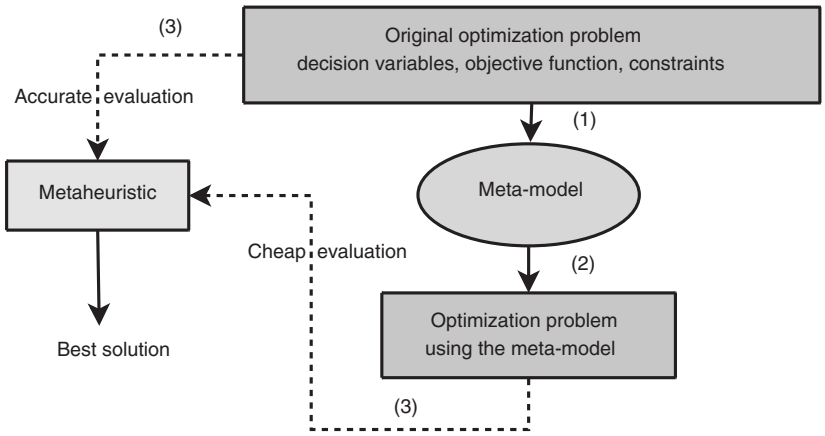

FIGURE 1.23 Optimization using a meta-model. Once the model is constructed (1), the metaheuristic can use either the meta-model (2) or the alternative between the two models (original and approximate) for a better compromise between accuracy and efficiency (3).

not directly related to the representation of solutions or the objective function may also be used, such as multiobjective optimization and coevolutionary models.

1.5.1 Reject Strategies

Reject strategies[31] represent a simple approach, where only feasible solutions are kept during the search and then infeasible solutions are automatically discarded.

This kind of strategies are conceivable if the portion of infeasible solutions of the search space is very small. Moreover, reject strategies do not exploit infeasible solutions. Indeed, it would be interesting to use some information on infeasible solutions to guide the search toward global optimum solutions that are in general on the boundary between feasible and infeasible solutions. In some optimization problems, feasible regions of the search space may be discontinuous. Hence, a path between two feasible solutions exists if it is composed of infeasible solutions.

1.5.2 Penalizing Strategies

In penalizing strategies, infeasible solutions are considered during the search process. The unconstrained objective function is extended by a penalty function that will penalize infeasible solutions. This is the most popular approach. Many alternatives may be used to define the penalties [304].

For instance, the objective function f may be penalized in a linear manner:

$$f'(s) = f(s) + \lambda c(s)$$

[31] Also named "death penalty."

where $c(s)$ represents the cost of the constraint violation and λ the aggregation weights. The search enables sequences of the type (s_t, s_{t+1}, s_{t+2}) where s_t and s_{t+2} represent feasible solutions, s_{t+1} is an infeasible solution, and s_{t+2} is better than s_t.

According to the difference between feasible and infeasible solutions, different penalty functions may be used [656]:

- **Violated constraints:** A straightforward function is to count the number of violated constraints. No information is used on how close the solution is to the feasible region of the search space. Given m constraints, the penalized function $f_p(x)$ of $f(x)$ is defined as follows:

$$f_p(x) = f(x) + \sum_{i=1}^{m} w_i \alpha_i$$

  where $\alpha_i = 1$ if constraint i is violated and $\alpha_i = 0$ otherwise, and w_i is the coefficient associated with each constraint i.
  For a problem with few and tight constraints, this strategy is useless.
- **Amount of infeasibility or repairing cost:** Information on how close a solution is to a feasible region is taken into account. This will give an idea about the cost of repairing the solution.
  For instance, more efficient approaches consist in including a distance to feasibility for each constraint. Considering q inequality constraints and $m - q$ equality constraints, the penalized function $f_p(x)$ will be formulated as follows:

$$f_p(x) = f(x) + \sum_{i=1}^{m} w_i d_i^k$$

  where d_i is a distance metric for the constraint i, $d_i = \alpha_i g_i(x)$ for $i = 1, \ldots, q$ and $d_i = |h_i(x)|$ for $i = q + 1, \ldots, m$. k is a user-defined constant (in general $k = 0, 1$), the constraints $1, \ldots, q$ are inequality constraints and the constraints $q + 1, \ldots, m$ are equality constraints.

When solving a constrained problem using a penalizing strategy, a good compromise for the initialization of the coefficient factors w_i must be found. Indeed, if w_i is too small, final solutions may be infeasible. If the coefficient factor w_i is too high, we may converge toward nonoptimal feasible solutions. The penalizing function used may be

- **Static:** In static strategies, a constant coefficient factor is defined for the whole search. The drawback of the static strategies is the determination of the coefficient factors w_i.

- **Dynamic:** In dynamic strategies, the coefficients factors w_i will change during the search. For instance, the severity of violating constraints may be increased with time. It means that when the search progresses, the penalties will be more strong, whereas in the beginning of the search highly infeasible solutions are admissible [423].

  Hence, a dynamic penalty function will take into account the time (e.g., number of iterations, generations, and number of generated solutions). Using a distance metric, the objective function may be formulated as follows:

$$f_p(x, t) = f(x) + \sum_{i=1}^{m} w_i(t) d_i^k$$

  where $w_i(t)$ is a decreasing monotonic function with t. More advanced functions such as annealing [547] or nonmonotonic functions (see Section 2.4.2) may be used.

  It is not simple to define a good dynamic penalty function. A good compromise for the initialization of the function $w_i(t)$ must be found. Indeed, if $w_i(t)$ is too slow decreasing, longer search is needed to find feasible solutions. Otherwise if $w_i(t)$ is too fast decreasing, we may converge quickly toward a nonoptimal feasible solution.
- **Adaptive:** The previously presented penalty functions (static and dynamic) do not exploit any information of the search process. In adaptive penalty functions, knowledge on the search process is included to improve the efficiency and the effectiveness of the search.

  The magnitude of the coefficient factors is updated according to the memory of the search [350]. The search memory may contain the best found solutions, last generated solutions, and so on. For instance, an adaptive strategy may consist in decreasing the coefficient factors when many feasible solutions are generated during the search, while increasing those factors if many infeasible solutions are generated.

Example 1.39 Adaptive penalization. Let us consider the capacitated vehicle routing problem (CVRP)[32]. An adaptive penalizing strategy may be applied to deal with the demand and duration constraints in a metaheuristic [308]:

$$f'(s) = f(s) + \alpha Q(s) + \beta D(s)$$

$Q(s)$ measures the total excess demand of all routes and $D(s)$ measures the excess duration of all routes. The parameters α and β are self-adjusting. Initially, the two parameters are initialized to 1. They are reduced (resp. increased) if the last μ visited

[32]The CVRP problem is defined in Exercise 1.11.

solutions are all feasible (resp. all infeasible), where μ is a user-defined parameter. The reduction (resp. increase) may consist in dividing (resp. multiplying) the actual value by 2, for example.

1.5.3 Repairing Strategies

Repairing strategies consist in heuristic algorithms transforming an infeasible solution into a feasible one. A repairing procedure is applied to infeasible solutions to generate feasible ones. For instance, those strategies are applied in the case where the search operators used by the optimization algorithms may generate infeasible solutions.

Example 1.40 Repairing strategy for the knapsack problem. In many combinatorial optimization problems, such as the knapsack problem, repairing strategies are used to handle the constraints. The knapsack problems represent an interesting class of problems with different applications. Moreover, many classical combinatorial optimization problems generate underlying knapsack problems. In the 0–1 knapsack problem, one have n different articles with weight w_i and utility u_i. A knapsack can hold a weight of at most w. The objective is to maximize the utility of the articles included in the knapsack satisfying the weight capacity of the knapsack. The decision variable x_j is defined as follows:

$$x_j = \begin{cases} 1 & \text{if the article is included} \\ 0 & \text{otherwise} \end{cases}$$

The problem consists in optimizing the objective function

$$\text{Max } f(x) = \sum_{j=1}^{n} x_j u_j$$

subject to the constraint

$$\sum_{j=1}^{n} w_j x_j \leq w$$

The following repairing procedure may be applied to infeasible solutions (see Algorithm 1.3). It consists in extracting from the knapsack some elements to satisfy the capacity constraint.

Algorithm 1.3 Repairing procedure for the knapsack.

Input: a nonfeasible solution s.
$s^{'} = s$;
While $s^{'}$ nonfeasible (i.e., $\sum_{j=1}^{n} w_j x_j > w$) **Do**
  Remove an item e_i from the knapsack: the element e_i maximizes the ratio $\frac{u_i}{w_i}$;
  $s^{'} = s^{'} \backslash e_i$;
Endo
Output: a feasible solution $s^{'}$.

The repairing heuristics are, of course, specific to the optimization problem at hand. Most of them are greedy heuristics. Then, the success of this strategy will depend on the availability of such efficient heuristics.

1.5.4 Decoding Strategies

A decoding procedure may be viewed as a function $R \rightarrow S$ that associates with each representation $r \in R$ a feasible solution $s \in S$ in the search space. This strategy consists in using indirect encodings (see Section 1.4.1.4). The topology of the search space is then transformed using the decoding function. The decoding function must have the following properties [179]:

- For each solution $r \in R$, corresponds a feasible solution $s \in S$.
- For each feasible solution $s \in S$, there is a representation $r \in R$ that corresponds to it.
- The computational complexity of the decoder must be reduced.
- The feasible solutions in S must have the same number of corresponding solutions in R.
- The representation space must have the locality property in the sense that distance between solutions in R must be positively correlated with the distance between feasible solutions in S.

1.5.5 Preserving Strategies

In preserving strategies for constraint handling, a specific representation and operators will ensure the generation of feasible solutions. They incorporate problem-specific knowledge into the representation and search operators to generate only feasible solutions and then preserve the feasibility of solutions.

This efficient class of strategies is tailored for specific problems. It cannot be generalized to handle constraints of all optimization problems. Moreover, for some problems such as the graph coloring problem, it is even difficult to find feasible initial solution or population of solutions to start the search.

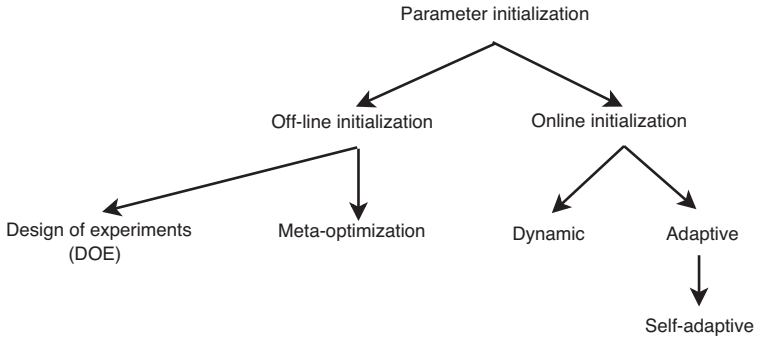

FIGURE 1.24 Parameter initialization strategies.

1.6 PARAMETER TUNING

Many parameters have to be tuned for any metaheuristic. Parameter tuning may allow a larger flexibility and robustness, but requires a careful initialization. Those parameters may have a great influence on the efficiency and effectiveness of the search. It is not obvious to define *a priori* which parameter setting should be used. The optimal values for the parameters depend mainly on the problem and even the instance to deal with and on the search time that the user wants to spend in solving the problem. A universally optimal parameter values set for a given metaheuristic does not exist.

There are two different strategies for parameter tuning: the *off-line*[33] parameter initialization (or meta-optimization) and the *online*[34] parameter tuning strategy (Fig. 1.24). In off-line parameter initialization, the values of different parameters are fixed before the execution of the metaheuristic, whereas in the online approach, the parameters are controlled and updated dynamically or adaptively during the execution of the metaheuristic.

1.6.1 Off-Line Parameter Initialization

As previously mentioned, metaheuristics have a major drawback; they need some parameter tuning that is not easy to perform in a thorough manner. Those parameters are not only numerical values but may also involve the use of search components.

Usually, metaheuristic designers tune one parameter at a time, and its optimal value is determined empirically. In this case, no interaction between parameters is studied. This sequential optimization strategy (i.e., one-by-one parameter) do not guarantee to find the optimal setting even if an exact optimization setting is performed.

[33] Also called endogenous strategy parameters [53].
[34] Also called exogenous strategy parameters.

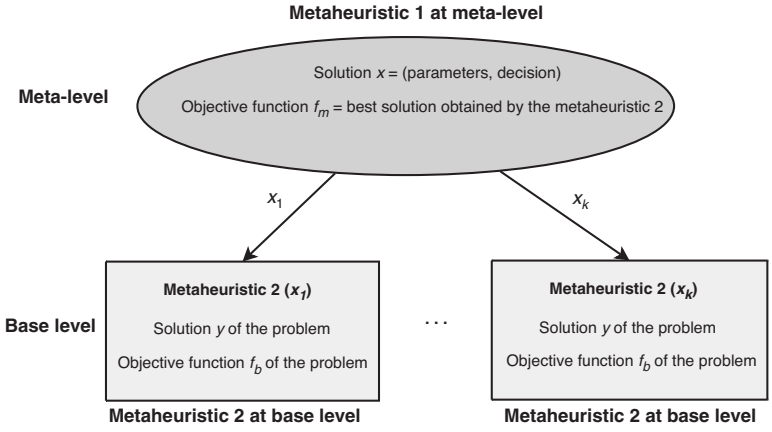

FIGURE 1.25 Meta-optimization using a meta-metaheuristic.

To overcome this problem, *experimental design*[35] is used [88]. Before using an experimental design approach, the following concepts must be defined:

- Factors[36] that represent the parameters to vary in the experiments.
- Levels that represent the different values of the parameters, which may be quantitative (e.g., mutation probability) or qualitative (e.g., neighborhood).

Let us consider n factors in which each factor has k levels, a full factorial design needs n^k experiments. Then, the "best" levels are identified for each factor. Hence, the main drawback of this approach is its high computational cost especially when the number of parameters (factors) and their domain values are large, that is, a very large number of experiments must be realized [683]. However, a small number of experiments may be performed by using *Latin hypercube* designs [536], sequential design, or *fractional design* [562].

Other approaches used in machine learning community such as *racing algorithms* [530] may be considered [76].

In off-line parameter initialization, the search for the best tuning of parameters of a metaheuristic in solving a given problem may be formulated as an optimization problem. Hence, this *meta-optimization* approach may be performed by any (meta)heuristic, leading to a meta-metaheuristic (or meta-algorithm) approach. Meta-optimization may be considered a hybrid scheme in metaheuristic design (see Section 5.1.1.2) (Fig. 1.25).

This approach is composed of two levels: the meta-level and the base level. At the meta-level, a metaheuristic operates on solutions (or populations) representing the parameters of the metaheuristic to optimize. A solution x at the meta-level will represent

[35] Also called *design of experiments* [266].
[36] Also named design variables, predictor variables, and input variables.

all the parameters the user wants to optimize: *parameter values* such as the size of the tabu list for tabu search, the cooling schedule in simulated annealing, the mutation and crossover probabilities for an evolutionary algorithm, and the *search operators* such as the type of selection strategy in evolutionary algorithms, the type of neighborhood in local search, and so on. At the meta-level, the objective function f_m associated with a solution x is generally the best found solution (or any performance indicator) by the metaheuristic using the parameters specified by the solution x. Hence, to each solution x of the meta-level will correspond an independent metaheuristic in the base level. The metaheuristic of the base level operates on solutions (or populations) that encode solutions of the original optimization problem. The objective function f_b used by the metaheuristic of the base level is associated with the target problem. Then, the following formula holds:

$$f_m(x) = f_b(\text{Meta}(x))$$

where Meta(x) represents the best solution returned by the metaheuristic using the parameters x.

1.6.2 Online Parameter Initialization

The drawback of the off-line approaches is their high computational cost, particularly if this approach is used for each input instance of the problem. Indeed, the optimal values of the parameters depend on the problem at hand and even on the various instances to solve. Then, to improve the effectiveness and the robustness of off-line approaches, they must be applied to any instance (or class of instances) of a given problem. Another alternative consists in using a parallel multistart approach that uses different parameter settings (see Chapter 6).

Another important drawback of off-line strategies is that the effectiveness of a parameter setting may change during the search; that is, at different moments of the search different optimal values are associated with a given parameter. Hence, online approaches that change the parameter values during the search must be designed. Online approaches may be classified as follows:

- **Dynamic update:** In a dynamic update, the change of the parameter value is performed without taking into account the search progress. A random or deterministic update of the parameter values is performed.
- **Adaptive update:** The adaptive approach changes the values according to the search progress. This is performed using the memory of the search.
  A subclass, referred to as *self-adaptive*[37] approach, consists in "evolving" the parameters during the search. Hence, the parameters are encoded into the representation and are subject to change as the solutions of the problem.

[37] Largely used in the evolutionary computation community.

In the rest of the book, many illustrative examples dealing with the off-line or online parameters initialization of each metaheuristic or search component are presented.

1.7 PERFORMANCE ANALYSIS OF METAHEURISTICS

Performance analysis of metaheuristics is a necessary task to perform and must be done on a fair basis. A theoretical approach is generally not sufficient to evaluate a metaheuristic [53]. This section addresses some guidelines of evaluating experimentally a metaheuristic and/or comparing metaheuristics in a rigorous way.

To evaluate the performance of a metaheuristic in a rigorous manner, the following three steps must be considered (Fig. 1.26):

- **Experimental design:** In the first step, the goals of the experiments, the selected instances, and factors have to be defined.
- **Measurement:** In the second step, the measures to compute are selected. After executing the different experiments, statistical analysis is applied to the obtained results. The performance analysis must be done with state-of-the-art optimization algorithms dedicated to the problem.
- **Reporting:** Finally, the results are presented in a comprehensive way, and an analysis is carried out following the defined goals. Another main issue here is to ensure the *reproducibility* of the computational experiments.

1.7.1 Experimental Design

In the computational experiment of a metaheuristic, the goals must be clearly defined. All the experiments, reported measures, and statistical analysis will depend on the purpose of designing the metaheuristic. Indeed, a contribution may be obtained for different criteria such as search time, quality of solutions, robustness in terms of the instances, solving large-scale problems, parallel scalability in terms of the number of processors, easiness of implementation, easiness to combine with other algorithms, flexibility to solve other problems or optimization models, innovation using new nature-inspired paradigms, automatic tuning of parameters, providing a tight approximation to the problem, and so on. Moreover, other purposes may be related to outline the contribution of a new search component in a given metaheuristic (representation, objective function, variation operators, diversification, intensification, hybrid models, parallel models, etc.).

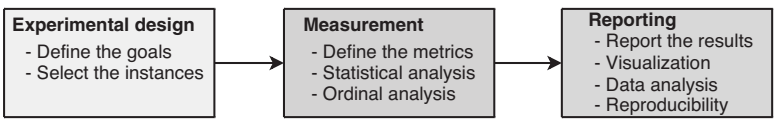

FIGURE 1.26 Different steps in the performance analysis of a metaheuristic: experimental design, measurement, and reporting.

Once the goals and factors are defined, methods from DOE can be suggested to conduct computational tests to ensure a rigorous statistical analysis [562]. It consists in selecting a set of combinations of values of factors to experiment (see Section 1.6.1). Then, the effect of a parameter (factor) p will be the change in the results obtained by the modification of the values of the parameter.

Once the goals are defined, the selection of the input instances to perform the evaluation must be carefully done. The structure associated with the input instances may influence significantly the performance of metaheuristics. Two types of instances exist:

- **Real-life instances:** They represent practical instances of the problem to be solved. If available, they constitute a good benchmark to carry out the performance evaluation of a metaheuristic.
  For some problems, it is difficult to obtain real-life instances for confidentiality reasons. In fact, most of the time, those data are proprietary and not public. For other problems, it is difficult to obtain a large number of real-life instances for financial reasons. For instance, in computational biology and bioinformatics, the generation of some genomic or proteomic data has a large cost. Also, collecting some real-life instances may be time consuming.
- **Constructed instances:** Many public libraries of "standard" instances are available on Internet [339]. They contain well-known instances for global optimization, combinatorial optimization, and mixed integer programs such as OR-Library[38], MIPLIB[39], DIMACS challenges[40], SATLIB for satisfiability problems, and the TSPLIB[41] (resp. QAPLIB) for the traveling salesman problem [646] (resp. the quadratic assignment problem).

  In addition to some real-life instances, those libraries contain in general *synthetic* or randomly generated instances. A disadvantage of *random* instances is that they are often too far from real-life problems to reflect their structure and important characteristics. The advantage of synthetic data is that they preserve the structure of real-life instances. Using a synthetic program, different instances in size and structure may be generated. Evaluating the performances of a given metaheuristic using only random instances may be controversial. For instance, the structure of uniformly generated random instances may be completely different from real-life instances of the problem, and then the effectiveness of the metaheuristic will be completely different in practice (see Section 2.2).

Example 1.41 Random instances may be controversial. Let us consider the symmetric TSP problem with n cities where the distance matrix is generated as follows: each element d_{ij}, $i \neq j$, of the distance matrix is independently generated between [0, 20] using a uniform distribution. Any randomly generated tour represents a good solution. For

[38] http://people.brunel.ac.uk/~mastjjb/jeb/info.html.
[39] http://www.caam.rice.edu/~bixby/miplib/miplib.html.
[40] http://dimacs.rutgers.edu/Challenges/.
[41] http://softlib.rice.edu/softlib/tsplib/.

TABLE 1.7 Some Classical Continuous Functions Used in Performance Evaluation of Metaheuristics

Function	Formulation
Sphere	$f(x) = \sum_{i=1}^{D} x_i^2$
Griewank	$f(x) = \frac{1}{4000} \sum_{i=1}^{D} x_i^2 - \prod_{i=1}^{D} \cos\left(\frac{x_i}{\sqrt{i}}\right) + 1$
Schaffer's f6	$f(x) = 0.5 - \frac{\left(\sin\sqrt{\left(x_1^2+x_2^2\right)}\right)^2 - 0.5}{\left(1+0.001\left(x_1^2+x_2^2\right)\right)^2}$
Rastrigin	$f(x) = \sum_{i=1}^{D} (x_i^2 - 10\cos(2\pi x_i) + 10)$
Rosenbrock	$f(x) = \sum_{i=1}^{D-1} (100(x_{i+1} - x_i^2)^2 + (x_i - 1)^2)$

D represents the number of dimensions associated with the problem.

example, for an instance of 5000 cities, it has been shown that the standard deviation is equal to 408 ($\sigma\sqrt{n}$) and the average cost is 50, 000 ($10 \cdot n$) [637]. According to the central limit theorem, almost any tour will have a good quality (i.e., cost of $\pm 3(408)$ of 50, 000). Hence, evaluating a metaheuristic on such instances is a pitfall to avoid. This is due to the independent random generation of the constants. Some correlation and internal consistency must be introduced for the constants.

Example 1.42 Continuous functions. In continuous optimization, some well-known functions are used to evaluate the performances of metaheuristics[42]: Schaffer, Griewank, Ackley, Rastrigin, Rosenbrock, and so on (see Table 1.7) [685]. These functions have different properties: for instance, the Sphere and Rastrigin are uncorrelated. The most studied dimensions are in the range [10–100]. Selected functions for metaheuristics must contain nonseparable, nonsymmetric, and nonlinear functions. Surprisingly, many used instances are separable, symmetric, or linear. Large dimensions are not always harder to solve. For instance, the Griewank function is easier to solve for large dimensions because the number of local optima decreases with the number of dimensions [815].

The selection of the input instances to evaluate a given metaheuristic may be chosen carefully. The set of instances must be diverse in terms of the size of the instances, their difficulties, and their structure. It must be divided into two subsets: the first subset will be used to tune the parameters of the metaheuristic and the second subset to evaluate the performance of the search algorithms. The calibration of the parameters of the metaheuristics is an important and tricky task. Most metaheuristics need the tuning of

[42]Test problems for global optimization may be found at `http://www2.imm.dtu.dk/~km/GlobOpt/testex/`.

various parameters that influence the quality of the obtained results. The values of the parameters associated with the used metaheuristics must be same for all instances. A single set of the parameter values is determined to solve all instances. No fine-tuning of the values is done for each instance unless the use of an automatic off-line or online initialization strategy (see Section 1.6). Indeed, this will cause an overfitting of the metaheuristic in solving known and specific instances. The parameter values will be excellent to solve the instances that serve to calibrate the parameters and very poor to tackle other instances. The robustness of the metaheuristic will be affected to solve unknown instances. Otherwise, the time to determine the parameter values of the metaheuristic to solve a given instance must be taken into account in the performance evaluation. Different parameter values may be adapted to different structures and sizes of the instances.

1.7.2 Measurement

In the second step, the performance measures and indicators to compute are selected. After executing the different experiments, some statistical analysis will be applied to the obtained results.

In exact optimization methods, the efficiency in terms of search time is the main indicator to evaluate the performances of the algorithms as they guarantee the global optimality of solutions. To evaluate the effectiveness of metaheuristic search methods, other indicators that are related to the quality of solutions have to be considered.

Performance indicators of a given metaheuristic may be classified into three groups [51]: solution quality, computational effort, and robustness. Other qualitative criteria such as the development cost, simplicity, ease of use, flexibility (wide applicability), and maintainability may be used.

1.7.2.1 Quality of Solutions Performance indicators for defining the quality of solutions in terms of precision are generally based on measuring the distance or the percent deviation of the obtained solution to one of the following solutions (Fig. 1.27):

- **Global optimal solution:** The use of global optimal solutions allows a more absolute performance evaluation of the different metaheuristics. The absolute

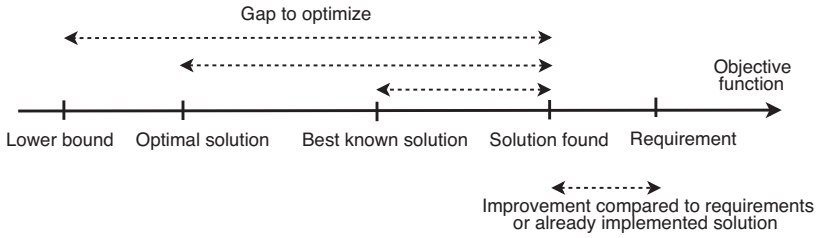

FIGURE 1.27 Performance assessment of the quality of the solutions. We suppose a minimization problem.

difference may be defined as $|f(s) - f(s^*)|$ or $|f(s) - f(s^*)|/f(s^*)$, where s is the obtained solution and s^* is the global optimal solution. Since those measures are not invariant under different scaling of the objective function, the following absolute approximation may be used: $|f(s) - f(s^*)|/|f_{\text{worst}} - f(s^*)|$ [838] or $|f(s) - f(s^*)|/|E_{\text{unif}}(f) - f(s^*)|$ [849], where f_{worst} represents the worst objective cost for the tackled instance[43], and $E_{\text{unif}}(f)$[44] denotes expectation with respect to the uniform distribution of solutions.

The global optimal solution may be found by an exact algorithm or may be available using "constructed" instances where the optimal solution is known *a priori* (by construction) [36]. Built-in optimal solutions have been considered for many academic problems [637]: traveling salesman problem [615], graph partitioning problem [483], Steiner tree problem [461], vertex packing, and maximum clique [677]. Also, for some problems, the optimal quality is known intrinsically. For example, in robot path planning, we have to optimize the distance between the actual position and the final one, and then the optimal solution has a null distance. Unfortunately, for many complex problems, global optimal solutions could not be available. There are also some statistical estimation techniques of optimal values in which a sample of solutions is used to predict the global optimal solution [209].

- **Lower/upper bound solution:** For optimization problems where the global optimal solution is not available, tight lower bounds[45] may be considered as an alternative to global optimal solutions. For some optimization problems, tight lower bounds are known and easy to obtain.

  Example 1.43 Simple lower bound for the TSP. The Held–Karp (HK) 1-tree lower bound for the symmetric TSP problem is quick and easy to compute [371]. Given an instance (V, d) where V is the set of n cities and d the distance matrix. A node $v_0 \in V$ is selected. Let r be the total edge length of a minimum spanning tree over the $n - 1$ cities ($v \in V - \{v_0\}$). The lower bound t is represented by the r value plus the two cheapest edges incident on v_0.

  $$t = r + \min\{d(v_0, x) + d(v_0, y) : x, y \in V - \{v_0\}, x \neq y\}$$

  Indeed, any TSP tour must use two edges e and f incident on the node v_0. Removing these two edges and the node v_0 from the tour yields a spanning tree of $V - \{v_0\}$. Typically, the lower bound t is 10% below the global optimal solution.

Different *relaxation* techniques may be used to find lower bounds such as the classical *continuous relaxation* and the *Lagrangian relaxation*. In continuous relaxation for IP problems, the variables are supposed to be real numbers instead

[43] For some problems, it is difficult to find the worst solution.
[44] This value can be efficiently computed for many problems. The expected error of a random solution is equal to 1.
[45] Lower bounds for minimization problems and upper bounds for maximization problems.

of integers. In Lagrangian relaxation, some constraints multiplied by Lagrange multipliers are incorporated into the objective function (see Section 5.2.1.2).

If the gap between the obtained solution and the lower bound is small, then the distance of the obtained solution to the optimal solution is smaller (see Fig. 1.27). In the case of null distance, the global optimality of the solution is proven. In the case of a large gap (e.g., $> 20\%$), it can be due to the bad quality of the bound or the poor performance of the metaheuristic.

- **Best known solution:** For many classical problems, there exist libraries of standard instances available on the Web. For those instances, the best available solution is known and is updated each time an improvement is found.
- **Requirements or actual implemented solution:** For real-life problems, a decision maker may define a requirement on the quality of the solution to obtain. This solution may be the one that is currently implemented. These solutions may constitute the reference in terms of quality.

1.7.2.2 Computational Effort The efficiency of a metaheuristic may be demonstrated using a theoretical analysis or an empirical one. In theoretical analysis, the worst-case complexity of the algorithm is generally computed (see Section 1.1.2). In general, reporting the asymptotic complexity is not sufficient and cannot tell the full story on computational performances of metaheuristics [419]. The average-case complexity, if it is possible to compute[46], is more practical [93].

In empirical analysis, measures related to the computation time of the metaheuristic used to solve a given instance are reported. The meaning of the computation time must be clearly specified: CPU time or wall clock time, with or without input/output and preprocessing/postprocessing time.

The main drawback of computation time measure is that it depends on the computer characteristics such as the hardware (e.g., processor, memories: RAM and cache, parallel architecture), operating systems, language, and compilers on which the metaheuristic is executed. Some indicators that are independent of the computer system may also be used, such as the number of objective function evaluations. It is an acceptable measure for time-intensive and constant objective functions. Using this metric may be problematic for problems where the evaluation cost is low compared to the rest of the metaheuristics or is not time constant in which it depends on the solution evaluated and time. This appears in some applications with variable length representations (genetic programming, robotics, etc.) and dynamic optimization problems.

Different stopping criteria may be used: time to obtain a given target solution, time to obtain a solution within a given percentage from a given solution (e.g., global optimal, lower bound, best known), number of iterations, and so on.

1.7.2.3 Robustness There is no commonly acceptable definition of robustness. Different alternative definitions exist for robustness. In general, robustness is insensitivity against small deviations in the input instances (data) or the parameters of

[46]It needs a probability distribution of the input instances.

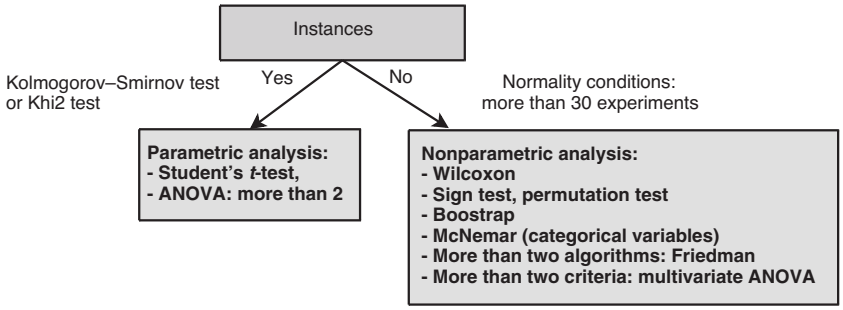

FIGURE 1.28 Statistical analysis of the obtained results.

the metaheuristic. The lower the variability of the obtained solutions the better the robustness [562].

In the metaheuristic community, robustness also measures the performance of the algorithms according to different types of input instances and/or problems. The metaheuristic should be able to perform well on a large variety of instances and/or problems using the same parameters. The parameters of the metaheuristic may be overfitted using the training set of instances and less efficient for other instances.

In stochastic algorithms, the robustness may also be related to the average/deviation behavior of the algorithm over different runs of the algorithm on the same instance.

1.7.2.4 Statistical Analysis Once the experimental results are obtained for different indicators, methods from statistical analysis[47] can be used to conduct the performance assessment of the designed metaheuristics [192]. While using any performance indicator (e.g., the quality of solutions c_i obtained by different metaheuristics M_i or their associated computational efforts t_i), some aggregation numbers that summarize the average and deviation tendencies must be considered. Then, different statistical tests may be carried out to analyze and compare the metaheuristics. The statistical tests are performed to estimate the confidence of the results to be scientifically valid (i.e., determining whether an obtained conclusion is due to a sampling error). The selection of a given statistical hypothesis testing tool is performed according to the characteristics of the data (e.g., variance, sample size) [562] (Fig. 1.28).

Under normality conditions, the most widely used test is the paired *t-test*. Otherwise, a nonparametric analysis may be realized such as the *Wilcoxon test* and the permutation test [337]. For a comparison of more than two algorithms, *ANOVA* models are well-established techniques to check the confidence of the results [146]. Multivariate ANOVA models allow simultaneous analysis of various performance measures (e.g., both the quality of solutions and the computation time). Kolmogorov–Smirnov test can be performed to check whether the obtained results follow a normal

[47] Many commercial (e.g., SAS, XPSS) and free softwares (e.g., R) are available to conduct such an analysis.

(Gaussian) distribution. Moreover, the Levene test can be used to test the homogeneity of the variances for each pair of samples. The Mann–Whitney statistical test can be used to compare two optimization methods. According to a p-value and a metric under consideration, this statistical test reveals if the sample of approximation sets obtained by a search method S_1 is significantly better than the sample of approximation sets obtained by a search method S_2, or if there is no significant difference between both optimization methods.

These different statistical analysis procedures must be adapted for nondeterministic (or stochastic) algorithms [740]. Indeed, most of the metaheuristics belong to this class of algorithms. Many trials (at least 10, more than 100 if possible) must be carried out to derive significant statistical results. From this set of trials, many measures may be computed: mean, median, minimum, maximum, standard deviation, the success rate that the reference solution (e.g., global optimum, best known, given goal) has been attained, and so on. The *success rate* represents the number of successful runs over the number of trials.

$$\text{success rate} = \frac{\text{number of successful runs}}{\text{total number of runs}}$$

The *performance rate* will take into account the computational effort by considering the number of objective function evaluations.

$$\text{performance rate} = \frac{\text{number of successful runs}}{\text{number of function evaluations} \times \text{total number of runs}}$$

When the number of trials n is important, the random variable associated with the average of the results found by a given metaheuristic over those trials tend to follow a Gaussian law of parameters m_0 and $\sigma_0/\sqrt{n}$, where m_0 (resp. σ_0) represents the average (resp. standard deviation) of the random variable associated with one experiment.

Confidence intervals (CI) can be used to indicate the reliability of the experiments. The confidence interval is an interval estimate of the set of experimental values. In practice, most confidence intervals are stated at the 95% level. It represents the probability that the experimental value is located in the interval $m - 1.96\sigma/\sqrt{n}$, $m + 1.96\sigma/\sqrt{n}$. A result with small CI is more reliable than results with a large CI.

1.7.2.5 Ordinal Data Analysis

In comparing n metaheuristics for a given number of m experiments (instances, etc.), a set of ordinal values o_k ($1 \leq k \leq m$) are generated for each method. For a given experiment, each ordinal value o_k denotes the rank of the metaheuristic compared to the other ones ($1 \leq o_k \leq n$). Some ordinal data analysis methods may be applied to be able to compare the different metaheuristics. Those ordinal methods aggregate m linear orders O_k into a single linear order O so that the final order O summarizes the m orders O_k.

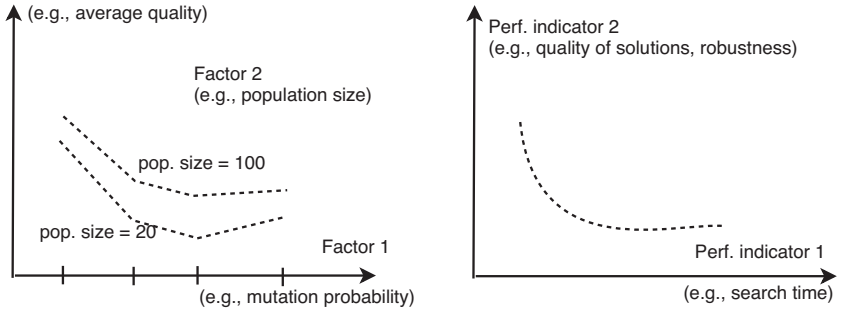

FIGURE 1.29 (a) Interaction plots analyze the effect of two factors (parameters, e.g., mutation probability, population size in evolutionary algorithms) on the obtained results (e.g., solution quality, time). (b) Scatter plots analyze the trade-off between the different performance indicators (e.g., quality of solutions, search time, robustness).

The commonly used ordinal aggregation methods are

- **Borda count voting method:** This method was proposed in 1770 by the French mathematician Jean-Charles de Borda. A method having a rank o_k is given o_k points. Given the m experiments, each method sums its points o_k to compute its total score. Then, the methods are classified according to their scores.
- **Copeland's method:** The Copeland method selects the method with the largest Copeland index. The Copeland index σ is the number of times a method beats other methods minus the number of times that method loses against other methods when the methods are considered in pairwise comparisons. For instance, let m and m' be two metaheuristics and $c_{mm'}$ the number of orders in which the metaheuristic m beats the metaheuristic m'. The Copelan index for the metaheuristic m will be $\sigma_m = \sum_{m'} c_{mm'} - c_{m'm}$. Then, the metaheuristics are ordered according to the decreasing values of the Copeland index σ.

1.7.3 Reporting

The interpretation of the results must be explicit and driven using the defined goals and considered performance measures. In general, it is not sufficient to present the large amount of data results using tables. Some visualization tools to analyze the data are welcome to complement the numerical results [780]. Indeed, graphical tools such as *deviation bars* (confidence intervals, box plots) and interaction plots allow a better understanding of the performance assessment of the obtained results.

Interaction plots represent the interaction between different factors and their effect on the obtained response (performance measure) (Fig. 1.29). *Box plots*[48] illustrate the

[48]Box plots were invented by the statistician John Tukey in 1977 [781].

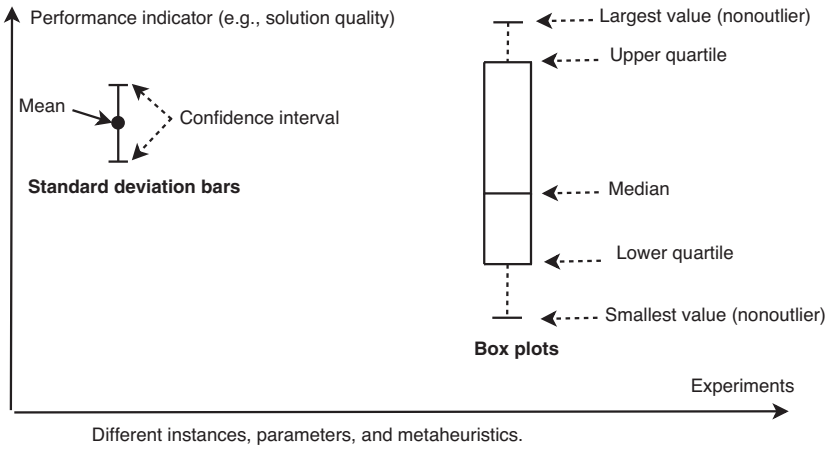

FIGURE 1.30 Some well-known visualization tools to report results: deviation bars, confidence intervals.

distribution of the results through their five-number summaries: the smallest value, lower quartile (Q1), median (Q2), upper quartile (Q3), and largest value (Fig. 1.30) [117]. They are useful in detecting outliers and indicating the dispersion and the skewness of the output data without any assumptions on the statistical distribution of the data.

Moreover, it is important to use *scatter plots* to illustrate the compromise between various performance indicators. For instance, the plots display quality of solutions versus time, or time versus robustness, or robustness versus quality (Fig. 1.29). Other plots measure the impact of a given factor on a performance indicator: time versus instance size and quality versus instance size. Indeed, analyzing the relationship between the quality of solution, the search time, the robustness, and the size/structure of instances must be performed in a comprehensive way. Other visualization tools may be used such as half-normal plots and histograms.

It would also be interesting to report negative results on applying a given metaheuristic or a search component to solve a given instance, problem, or class of problems. Indeed, most of the time only positive results are reported in the literature. From negative results, one may extract useful knowledge.

For a given experimental evaluation of a metaheuristic, *confidence intervals* may be plotted by a segment indicating the confidence interval at 95% level (Fig. 1.30). The middle of the segment shows the average of the experimental values.

More information on the behavior of a stochastic metaheuristic may be obtained by approximating the probability distribution for the time to a target solution value. To plot the empirical distribution for an algorithm and an instance, the i-smallest running time t_i may be associated with the probability $p_i = (i - (1/2))/n$, where n is the number of independent runs ($n \geq 100$), and plots the points $z_i = (t_i, p_i)$ for $i = [1, \ldots, n]$.

A metaheuristic must be well documented to be reproduced. The program must be described in detail to allow its reproduction. If possible, making available the

program, the instances, and the obtained results (complete solutions and the different measures) on the Web will be a plus. The different used parameters of the metaheuristic must be reported. Using different parameters in solving the different instances must also be reported. The use of software frameworks makes better the reproducibility, reusability, and extension of metaheuristics. In fact, if the competing metaheuristics are implemented using the same software framework, the performance metrics such as the search time are less biased to the programming skills and the computing system, and then the comparison is more fair and rigorous.

1.8 SOFTWARE FRAMEWORKS FOR METAHEURISTICS

In this section, the motivations for using a software framework for metaheuristics are outlined. Then, the main characteristics a framework should have are detailed. Finally, the ParadisEO framework that serves to design and implement various metaheuristics (e.g., S-metaheuristics, P-metaheuristics, hybrid, and parallel metaheuristics) in the whole book is presented.

1.8.1 Why a Software Framework for Metaheuristics?

Designing software frameworks for metaheuristics is primordial. In practice, there is a large diversity of optimization problems. Moreover, there is a continual evolution of the models associated with optimization problems. The problem may change or need further refinements. Some objectives and/or constraints may be added, deleted, or modified. In general, the efficient solving of a problem needs to experiment many solving methods, tuning the parameters of each metaheuristic, and so on. The metaheuristic domain in terms of new algorithms is also evolving. More and more increasingly complex metaheuristics are being developed (e.g., hybrid strategies, parallel models, etc.).

There is a clear need to provide a ready-to-use implementation of metaheuristics. It is important for application engineers to choose, implement, and apply the state-of-the art algorithms without in-depth programming knowledge and expertise in optimization. For optimization experts and developers, it is useful to evaluate and compare fairly different algorithms, transform ready-to-use algorithms, design new algorithms, and combine and parallelize algorithms.

Three major approaches are used for the development of metaheuristics:

- **From scratch or no reuse:** Nowadays, unfortunately this is the most popular approach. The basic idea behind the from scratch-oriented approach is the apparent simplicity of metaheuristic code. Programmers are tempted to develop themselves their codes. Therefore, they are faced with several problems: the development requires time and energy, and it is error prone and difficult to maintain and evolve.

  Numerous metaheuristics and their implementation (program codes) have been proposed, and are available on the Web. They can be reused and adapted to a

user problem. However, the user has to deeply examine the code and rewrite its problem-specific sections. This task is often tedious, error prone, takes a long time, and makes harder the produced code maintenance.

- **Only code reuse:** it consists of reusing third-party code available on the Web either as free individual programs or as libraries. Reusability may be defined as the ability of software components to build many different applications [262]. An old third-party code has usually application-dependent sections that must be extracted before the new application-dependent code can be inserted. Changing these sections is often time consuming and error prone.

  A better way to reuse the code of existing metaheuristics is through libraries [804]. The code reuse through libraries is obviously better because these libraries are often well tried, tested, and documented, thus more reliable. They allow a better maintainability and efficiency. Nowadays, it is recognized that the object-oriented paradigm is well-suited to develop reusable libraries. However, libraries allow code reuse but they do not permit the reuse of complete invariant part of algorithms. Libraries do not allow the reuse of design. Therefore, the coding effort using libraries remains important.
- **Both design and code reuse:** The objective of both code and design reuse approaches is to overcome this problem, that is, to redo as little code as possible each time a new optimization problem is dealt with. The basic idea is to capture into special components the recurring (or invariant) part of solution methods to standard problems belonging to a specific domain. These special components are called *design patterns* [293]. A pattern can be viewed as a programming language-independent description of a solution to a general design problem that must be adapted for its eventual use [523]. Useful design patterns related to a specific domain (e.g., metaheuristics) are in general implemented as *frameworks*. A framework approach is devoted to the design and code reuse of a metaheuristic [422]. A framework may be object oriented and defined as a set of classes that embody an abstract design for solutions to a family of related metaheuristics. Frameworks are well known in the software engineering literature. Frameworks can thus be viewed as programming language-dependent concrete realizations of patterns that facilitate direct reuse of design and code. They allow the reuse of the design and implementation of a whole metaheuristic. They are based on a strong conceptual separation of the invariant (generic) part of metaheuristics and their problem-specific part. Therefore, they allow the user to redo very little code, and it improves the quality and the maintainability of the metaheuristics.

  Moreover, unlike libraries, frameworks are characterized by the inverse control mechanism for the interaction with the application code. In a framework, the provided code calls the user-defined one according to the Hollywood property “do not call us, we call you." Therefore, frameworks provide the full control structure of the invariant part of the algorithms, and the user has to supply only the problem-specific details. To meet this property, the design of a framework must be based on a clear conceptual separation between the solution methods and the problems they tackle.

This separation requires a solid understanding of the application domain. The domain analysis results in a model of the domain to be covered by reusable classes with some constant and variable aspects. The constant part is encapsulated in generic/abstract classes or skeletons that are implemented in the framework [22]. The variable part is problem specific, it is fixed in the framework but implemented by the user. This part consists of a set of holes or hot spots [624] that serve to fill the skeletons provided by the framework when building specific applications. It is recommended to use object-oriented composition rather than inheritance to perform this separation [660]. The reason is that classes are easier to reuse than individual methods. Another and completely different way to perform this separation may be used [81]. It provides a ready-to-use module for each part, and the two modules communicate through text files. This allows less flexibility than the object-oriented approach. Moreover, it induces an additional overhead, even if this is small. Nevertheless, this approach is multilanguage allowing more code reuse.

1.8.2 Main Characteristics of Software Frameworks

According to the openness criterion, two types of frameworks can be distinguished: *white* or glass box frameworks and *black box* (opaque) frameworks. In black box frameworks, one can reuse components by plugging them together through static parameterization and composition, and not worrying about how they accomplish their individual tasks [422]. In contrast, white box frameworks require an understanding of how the classes work so that correct subclasses (inheritance based) can be developed. Therefore, they allow more extendability. Frameworks often start as white box frameworks; these are primarily customized and reused through classes specialization. When the variable part has stabilized or been realized, it is often appropriate to evolve to black box frameworks [262].

Nowadays, the white box approach is more suited to metaheuristics. It is composed of adaptable software components intended to be reused to solve specific optimization problems. Unless the automatic or quasi-automatic design of a metaheuristic for a given problem is not solved, the designer must tailor a given metaheuristic to solve the problem. The source code level must be provided to the user to adapt his algorithm. The black box approach can be adapted to some families of optimization problems such as nonlinear continuous optimization problems where the same search components can be used (representation, search operators, etc.). In other families such as combinatorial optimization, the representation and search operators are always tailored to solve a problem using programming languages such as C++ or Java. For instance, the black box approach is used for linear programming optimization solvers (e.g., Cplex, Lindo, XPRESS-MP) that use a modeling language based on mathematical programming, such as the AMPL[49], GAMS,[50] or MPL[51] languages.

[49] `www.ampl.com.`
[50] `www.gams.com.`
[51] `www.maximal-usa.com.`

A framework is normally intended to be exploited by as many users as possible. Therefore, its exploitation could be successful only if some important user criteria are satisfied. The following are the major criteria of them and constitute the main objectives of the used framework in this book:

- **Maximum design and code reuse:** The framework must provide for the user a whole architecture design of his/her metaheuristic approach. Moreover, the programmer may redo as little code as possible. This objective requires a clear and maximal conceptual separation between the metaheuristics and the problems to be solved, and thus a deep domain analysis. The user might therefore develop only the minimal problem-specific code. It will simplify considerably the development of metaheuristics and reduce the development time.
- **Flexibility and adaptability:** It must be possible for the user to easily add new features/metaheuristics or change existing ones without implicating other components. Furthermore, as in practice existing problems evolve and new others arise, these have to be tackled by specializing/adapting the framework components.
- **Utility:** The framework must allow the user to cover a broad range of metaheuristics, problems, parallel distributed models, hybridization mechanisms, multiobjective optimization, and so on. To design optimization methods for hard problems, a lot of metaheuristics exist. Nevertheless, the scientist does not have necessarily the time and the capability to try all of them. Furthermore, to gain effective method, the parameters often need to be tuned. So a platform that can facilitate the design of optimization methods and their test is necessary to produce high-quality results.
- **Transparent and easy access to performance and robustness:** As the optimization applications are often time consuming, the performance issue is crucial. Parallelism and distribution are two important ways to achieve high-performance execution. To facilitate its use, it is implemented so that the user can deploy his/her parallel metaheuristic in a transparent manner. Moreover, the execution of the algorithms must be robust to guarantee the reliability and the quality of the results. The hybridization mechanism allows to obtain robust and better solutions.
- **Portability:** To satisfy a large number of users, the framework must support different material architectures (sequential, parallel, or distributed architecture) and their associated operating systems (Windows, Unix, MacOs).
- **Easy to use and efficiency:** The framework must be easy to use and does not incorporate an additional cost in terms of time or space complexity. The framework must preserve the efficiency of a special-purpose implementation. On the contrary, as the framework is normally developed by "professional" and knowledgeable software engineers and is largely tested by many users, it will be less error prone than ad-hoc special-purpose developed metaheuristics. Moreover, it is well known that the most intensive computational part in a metaheuristic is generally the evaluation of the objective function that is specified by the user to solve his specific problem.

Several frameworks for metaheuristics have been proposed in the literature. Most of them have the following limitations:

- **Metaheuristics:** most of the exiting frameworks focus only on a given metaheuristic or family of metaheuristics such as evolutionary algorithms (e.g., GAlib [809]), local search (e.g., EasyLocal++ [301], Localizer [550]), and scatter search (e.g., OPTQUEST). Only few frameworks are dedicated on the design of both families of metaheuristics. Indeed, a unified view of metaheuristics must be done to provide a generic framework.
- **Optimization problems:** most of the software frameworks are too narrow, that is, they have been designed for a given family of optimization problems: nonlinear continuous optimization (e.g., GenocopIII), combinatorial optimization (e.g., iOpt), monoobjective optimization (e.g., BEAGLE), multiobjective optimization (e.g., PISA [81]), and so on.
- **Parallel and hybrid metaheuristics:** Moreover, most of the existing frameworks either do not provide hybrid and parallel metaheuristics at all (Hotframe [262]) or supply just some parallel models: island model for evolutionary algorithms (e.g., DREAM [35], ECJ [819], JDEAL, distributed BEAGLE [291]), independent multistart and parallel evaluation of the neighborhood (e.g., TS [79]), or hybrid metaheuristics (iOpt [806]).
- **Architectures:** Finally, seldom a framework is found that can target many types of architectures: sequential and different types of parallel and distributed architectures, such as shared-memory (e.g., multicore, SMP), distributed-memory (e.g., clusters, network of workstations), and large-scale distributed architectures (e.g., desktop grids and high-performance grids). Some software frameworks are dedicated to a given type of parallel architectures (e.g., MALLBA [22], MAFRA [481], and TEMPLAR [426,427]).

Table 1.8 illustrates the characteristics of the main software frameworks for metaheuristics[52]. For a more detailed review of some software frameworks and libraries for metaheuristics, the reader may refer to Ref. [804].

1.8.3 ParadisEO Framework

In this book, we will use the ParadisEO[53] framework to illustrate the design and implementation of the different metaheuristics. The ParadisEO platform honors the criteria mentioned before, and it can be used both by no-specialists and by optimization method experts. It allows the design and implementation of

- Single-solution based and population-based metaheuristics in a unifying way (see Chapters 2 and 3).

[52] We do not claim an exhaustive comparison.
[53] ParadisEO is distributed under the CeCill license.

TABLE 1.8 Main Characteristics of Some Software Frameworks for Metaheuristics

Framework or Library	Metaheuristic	Optimization Problems	Parallel Models	Communication Systems
EasyLocal++	S-meta	Mono	-	-
Localizer++	S-meta	Mono	-	-
PISA	EA	Multi	-	-
MAFRA	LS, EA	Mono	-	-
iOpt	S-meta, GA, CP	Mono, COP	-	-
OptQuest	SS	Mono	-	-
GAlib	GA	Mono	Algo-level Ite-level	PVM
GenocopIII	EA	Mono, Cont	-	-
DREAM	EA	Mono	Algo-level	Peer-to-peer sockets
MALLBA	LS EA	Mono	Algo-level Ite-level	MPI Netstream
Hotframe	S-meta, EA	Mono	-	-
TEMPLAR	LS, SA, GA	Mono, COP	Algo-level	MPI, threads
JDEAL	GA, ES	Mono	Ite-level	Sockets
ECJ	EA	Mono	Algo-level	Threads, sockets
Dist. BEAGLE	EA	Mono	Algo-level Ite-level	Sockets
ParadisEO	S-meta P-meta	Mono, Multi COP, Cont	Algo-level Ite-level Sol-level	MPI, threads Condor Globus

[S-meta: S-metaheuristics; P-meta: P-metaheuristics; COP: combinatorial optimization; Cont: continuous optimization; Mono: Monoobjective optimization; Multi: multiobjective optimization, LS: local search; ES: evolution strategy; SS: scatter search; EA: evolutionary algorithms; GA: genetic algorithms; Algo-level: algorithmic level of parallel model; Ite-level: iteration level of parallel models; Sol-level: solution level of parallel models. Unfortunately, only a few of them are maintained and used!.]

- Metaheuristics for monoobjective and multiobjective optimization problems (see Chapter 4).
- Metaheuristics for continuous and discrete optimization problems.
- Hybrid metaheuristics (see Chapter 5).
- Parallel and distributed metaheuristics (see Chapter 6).

ParadisEO is a white box object-oriented framework based on a clear conceptual separation of the metaheuristics from the problems they are intended to solve. This separation and the large variety of implemented optimization features allow a maximum code and design reuse. The separation is expressed at implementation level by splitting the classes into two categories: provided classes and required classes. The provided classes constitute a hierarchy of classes implementing the invariant part of the code. Expert users can extend the framework by inheritance/specialization. The

required classes coding the problem-specific part are abstract classes that have to be specialized and implemented by the user.

The classes of the framework are fine-grained and instantiated as evolving objects embodying one and only one method. This is a particular design choice adopted in ParadisEO. The heavy use of these small-size classes allows more independence and thus a higher flexibility compared to other frameworks. Changing existing components and adding new ones can be easily done without impacting the rest of the application. Flexibility is enabled through the use of the object-oriented technology. Templates are used to model the metaheuristic features: coding structures, transformation operators, stopping criteria, and so on. These templates can be instantiated by the user according to his/her problem-dependent parameters. The object-oriented mechanisms such as inheritance, polymorphism, and so on are powerful ways to design new algorithms or evolve existing ones. Furthermore, ParadisEO integrates several services making it easier to use, including visualization facilities, online definition of parameters, application checkpointing, and so on.

ParadisEO is one of the rare frameworks that provides the most common parallel and distributed models. These models concern the three main parallel models: algorithmic level, iteration level, and solution level. They are portable on different types of architectures: distributed-memory machines and shared-memory multiprocessors as they are implemented using standard libraries such as message passing interface (MPI), multithreading (Pthreads), or grid middlewares (Condor or Globus). The models can be exploited in a transparent way, one has just to instantiate their associated ParadisEO components. The user has the possibility to choose by a simple instantiation for the communication layer. The models have been validated on academic and industrial problems. The experimental results demonstrate their efficiency. The experimentation also demonstrates the high reuse capabilities as the results show that the user redo little code. Furthermore, the framework provides the most common hybridization mechanisms. They can be exploited in a natural way to make cooperating metaheuristics belonging either to the same family or to different families.

ParadisEO is a C++ LGPL open-source framework (STL-Template)[54]. It is portable on Windows, Unix-like systems such as Linux and MacOS. It includes the following set of modules (Fig. 1.31):

- **Evolving objects (EO):** The EO library was developed initially for evolutionary algorithms (genetic algorithms, evolution strategies, evolutionary programming, genetic programming, and estimation distribution algorithms) [453]. It has been extended to population-based metaheuristics such as particle swarm optimization and ant colony[55] optimization.
- **Moving objects (MO):** It includes single-solution based metaheuristics such as local search, simulated annealing, tabu search, and iterated local search.

[54]Downloadable at `http://paradiseo.gforge.inria.fr`.
[55]The model implemented is inspired by the self-organization of *Pachycondyla apicalis* ant species.

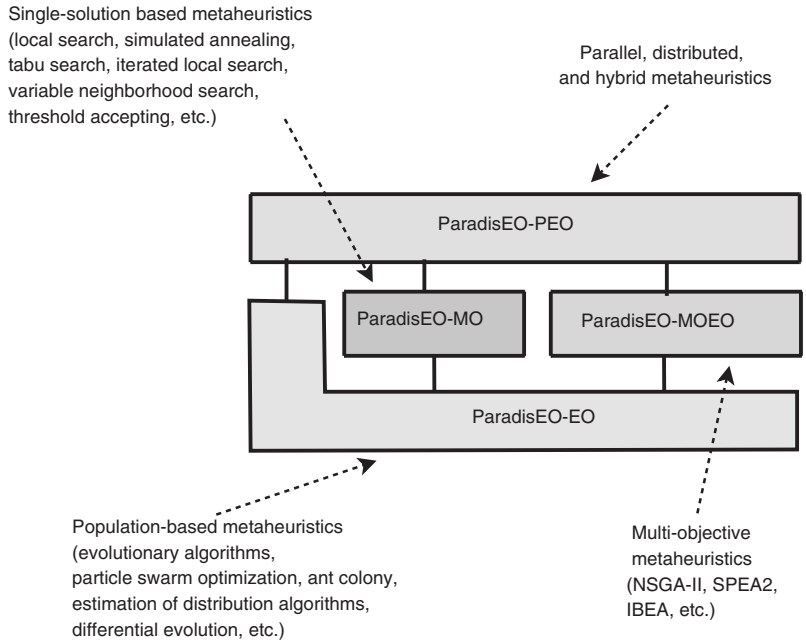

FIGURE 1.31 The different unified modules of the ParadisEO framework.

- **Multiobjective evolving objects (MOEO):** It includes the search mechanisms to solve multiobjective optimization problems such as fitness assignment, diversification, and elitism. From this set of mechanisms, classical algorithms such as NSGA-II, SPEA2, and IBEA have been implemented and are available.
- **Parallel evolving objects (PEO):** It includes the well-known parallel and distributed models for metaheuristics and their hybridization.

1.8.3.1 ParadisEO Architecture The architecture of ParadisEO is multilayered and modular allowing to achieve the objectives quoted above (Fig. 1.32).

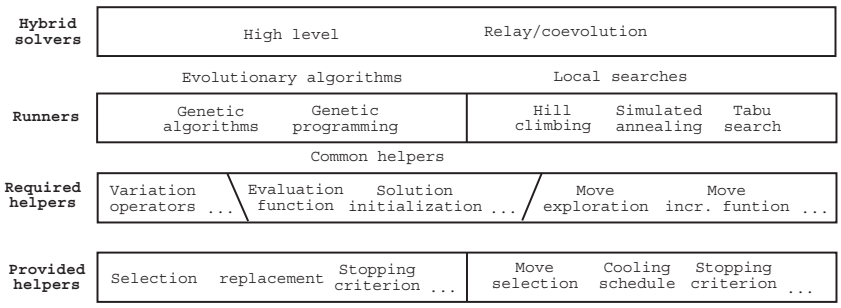

FIGURE 1.32 Architecture of the ParadisEO framework.

This allows particularly a high genericity, flexibility, and adaptability, an easy hybridization, and code and design reuse. The architecture has three layers identifying three major classes: *Solvers*, *Runners*, and *Helpers*.

- **Helpers:** Helpers are low-level classes that perform specific actions related to the search process. They are split into two categories: *population helpers* (PH) and *single-solution helpers* (SH). Population helpers include mainly the transformation, selection, and replacement operations, the evaluation function, and the stopping criterion. Solution helpers can be generic such as the neighborhood explorer class, or specific to the local search metaheuristic such as the tabu list manager class in the tabu search solution method. On the other hand, there are some special helpers dedicated to the management of parallel and distributed models, such as the communicators that embody the communication services. Helpers cooperate between them and interact with the components of the upper layer, that is, the runners. The runners invoke the helpers through function parameters. Indeed, helpers do not have not their own data, but they work on the internal data of the runners.
- **Runners:** The *Runners* layer contains a set of classes that implement the metaheuristics themselves. They perform the run of the metaheuristics from the initial state or population to the final one. One can distinguish the *population runners* (PR) such as genetic algorithms, evolution strategies, particle swarm, and so on and *single-solution runners* (SR) such as tabu search, simulated annealing, and hill climbing. Runners invoke the helpers to perform specific actions on their data. For instance, a PR may ask the fitness function evaluation helper to evaluate its population. An SR asks the movement helper to perform a given movement on the current state. Furthermore, runners can be serial or parallel distributed.
- **Solvers:** Solvers are devoted to control the search. They generate the initial state (solution or population) and define the strategy for combining and sequencing different metaheuristics. Two types of solvers can be distinguished: *single metaheuristic solvers* (SMS) and *multiple-metaheuristic solvers* (MMS). SMS are dedicated to the execution of a single metaheuristic. MMS are more complex as they control and sequence several metaheuristics that can be heterogeneous. They use different hybridization mechanisms. Solvers interact with the user by getting the input data and by delivering the output (best solution, statistics, etc.).

According to the generality of their embedded features, the classes of the architecture are split into two major categories: *provided* classes and *required* classes. Provided classes embody the factored out part of the metaheuristics. They are generic, implemented in the framework, and ensure the control at run time. Required classes are those that must be supplied by the user. They encapsulate the problem-specific aspects of the application. These classes are fixed but not implemented in ParadisEO. The programmer has the burden to develop them using the object-oriented specialization mechanism.

At each layer of the ParadisEO architecture, a set of classes is provided (Fig. 1.32). Some of them are devoted to the development of metaheuristics for monoobjective and multiobjective optimization, and others are devoted to manage transparently parallel and distributed models for metaheuristics and their hybridization.

There are two programming mechanisms to extend built-in classes: function substitution and subclassing. By providing some methods, any class accepts that the user specifies his own function as a parameter that will be used instead of the original function. This will avoid the use of subclassing, which is a more complex task. The user must at least provide the objective function associated with his problem.

1.9 CONCLUSIONS

When identifying a decision-making problem, the first issue deals with modeling the problem. Indeed, a mathematical model is built from the formulated problem. One can be inspired by similar models in the literature. This will reduce the problem to well-studied optimization models. One has also to be aware of the accuracy of the model. Usually, models we are solving are simplifications of the reality. They involve approximations and sometimes they skip processes that are complex to represent in a mathematical model.

Once the problem is modeled, the following roadmap may constitute a guideline in solving the problem (Fig.1.33).

First, whether it is legitimate to use metaheuristics for solving the problem must be addressed. The complexity and difficulty of the problem (e.g., NP-completeness, size, and structure of the input instances) and the requirements of the target optimization problem (e.g., search time, quality of the solutions, and robustness) must be taken into account. This step concerns the study of the intractability of the problem at hand. Moreover, a study of the state-of-the-art optimization algorithms (e.g., exact and heuristic algorithms) to solve the problem must be performed. For instance, the use of exact methods is preferable if the best known exact algorithm can solve in the required time the input instances of the target problem. Metaheuristic algorithms seek good solutions to optimization problems in circumstances where the complexity of the tackled problem or the search time available does not allow the use of exact optimization algorithms.

At the time the need to design a metaheuristic is identified, there are three common design questions related to all iterative metaheuristics:

- **Representation:** A traditional (e.g., linear/nonlinear, direct/indirect) or a specific encoding may be used to represent the solutions of the problem. Encoding plays a major role in the efficiency and effectiveness of any metaheuristic and constitutes an essential step in designing a metaheuristic. The representation must have some desired properties such as the completeness, connexity, and efficiency. The encoding must be suitable and relevant to the tackled optimization problem. Moreover, the efficiency of a representation is also related to the search

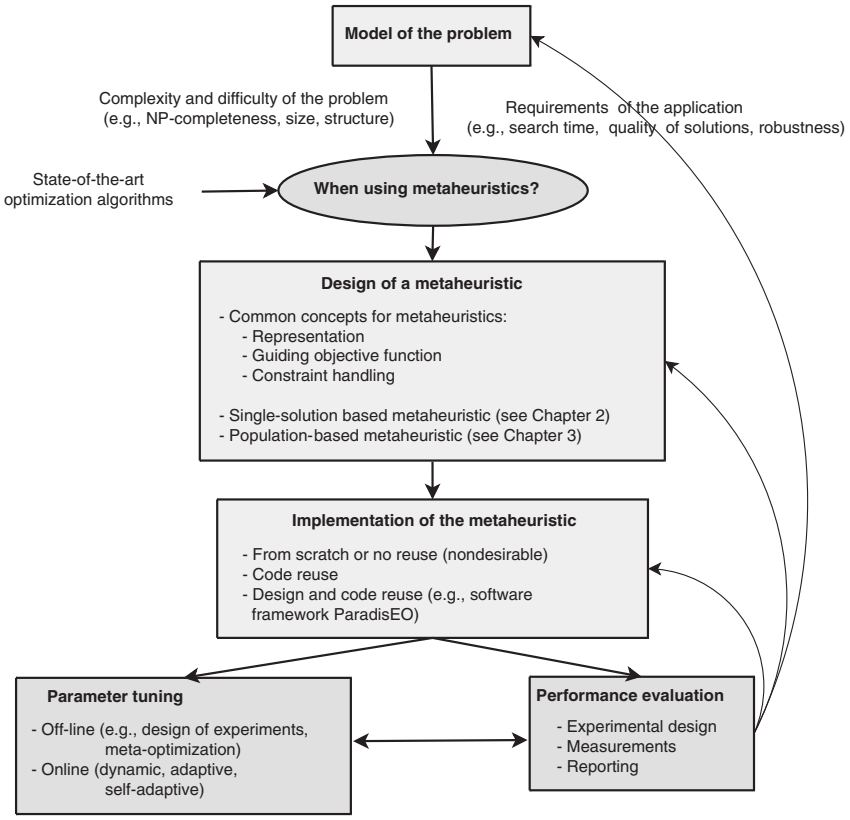

FIGURE 1.33 Guidelines for solving a given optimization problem.

operators applied to this representation (e.g., generation of the neighborhood, recombination of solutions). In fact, when defining a representation, one has to bear in mind how the solution will be evaluated and how the search operators will operate.

- **Objective function:** The objective function is an important element in designing a metaheuristic. It will guide the search toward "good" solutions of the search space. The guiding objective function is related to the goal to achieve. For efficiency and effectiveness reasons, the guiding function may be different from the objective function formulated by the model.
- **Constraint handling:** Dealing with constraints in optimization problems is another important aspect of the efficient design of metaheuristics. Indeed, many continuous and discrete optimization problems are constrained, and it is not trivial to deal with those constraints. Most of the constraint handling strategies act on the representation of solutions or the objective function (e.g., reject, penalizing, repairing, decoding, and preserving strategies).

If the representation, the objective function, and the constraints are improperly handled, solving the problem can lead to nonacceptable solutions whatever metaheuristic is used.

Software frameworks are essential in the implementation of metaheuristics. These frameworks enable the application of different metaheuristics (S-metaheuristics, P-metaheuristics) in a unified way to solve a large variety of optimization problems (monoobjective/multiobjective, continuous/discrete) as well as to support the extension and adaptation of the metaheuristics for continually evolving problems. Hence, the user will focus on high-level design aspects. In general, the efficient solving of a problem needs to experiment many solving methods, tuning the parameters of each metaheuristic, and so on. The metaheuristic domain in terms of new algorithms is also evolving. More and more increasingly complex metaheuristics are being developed. Moreover, it allows the design of complex hybrid and parallel models that can be implemented in a transparent manner on a variety of architectures (sequential, shared-memory, distributed-memory, and large-scale distributed architecture).

Hence, there is a clear need to provide a ready-to-use implementation of metaheuristics. It is important for application engineers to choose, implement, and apply the state-of-the-art algorithms without in-depth programming knowledge and expertise in optimization. For optimization experts and developers, it is useful to evaluate and compare fairly different algorithms, transform ready-to-use algorithms, design new algorithms, and combine and parallelize algorithms. Frameworks may provide default implementation of classes. The user has to replace the defaults that are inappropriate for his application.

Many parameters have to be tuned for any metaheuristic. Parameter tuning may allow a larger flexibility and robustness but requires a careful initialization. Those parameters may have a great influence on the efficiency and effectiveness of the search. It is not obvious to define *a priori* which parameter setting should be used. The optimal values for the parameters depend mainly on the problem and even the instance to deal with and on the search time that the user wants to spend in solving the problem. A universally optimal parameter values set for a given metaheuristic does not exist.

The performance evaluation of the developed metaheuristic is the last step of the roadmap. Worst-case and average-case theoretical analyses of metaheuristics present some insight into solving some traditional optimization models. In most of the cases, an experimental approach must be realized to evaluate a metaheuristic. Performance analysis of metaheuristics is a necessary task to perform and must be done on a fair basis. A theoretical approach is generally not sufficient to evaluate a metaheuristic. To evaluate the performance of a metaheuristic in a rigorous manner, the following three steps must be considered: experimental design (e.g., goals of the experiments, selected instances, and factors), measurement (e.g., quality of solutions, computational effort, and robustness), and reporting (e.g., box plots, interaction plots). The performance analysis must be done with the state-of-the-art optimization algorithms dedicated to the problem according to the defined goals. Another main issue here is to ensure the reproducibility of the computational experiments.

In the next two chapters, we will focus on the main search concepts for designing single-solution based metaheuristics and population-based metaheuristics. Each class

of algorithms shares some common concepts that can be unified in the description and the design of a metaheuristic. This classification provides a clearer presentation of hybrid metaheuristics, parallel metaheuristics, and metaheuristics for multiobjective optimization.

1.10 EXERCISES

Exercise 1.1 Related problems to maximum clique. Given an undirected graph $G = (V, E)$. A clique Q of the graph G is a subset of V where any two vertices in Q are adjacent:

$$\forall i, j \in Q \times Q, (i, j) \in E$$

A maximum clique is a clique with the largest cardinality. The problem of finding the maximum clique is NP-hard. The clique number is the cardinality of the maximum clique. Given the following problems:

- The subset $I \subseteq V$ of maximum cardinality such as the set of edges of the subgraph induced by I is empty.
- Graph coloring.

Find the relationships between the formulated problems and the maximum clique problem. How these problems are identified in the literature?

Exercise 1.2 Easy versus hard optimization problem. Let us consider the set bipartitioning problem. Given a set X of n positive integers $e_1, e_2, \ldots, e_n$ where n is an even value. The problem consists in partitioning the set X into two subsets Y and Z of equal size. How many possible partitions of the set X exist?

Two optimization problems may be defined:

- Maximum set bipartitioning that consists in maximizing the difference between the sums of the two subsets Y and Z.
- Minimum set bipartitioning that consists in minimizing the difference between the sums of the two subsets Y and Z.

To which complexity class the two optimization problems belong? Let us consider the minimum set bipartitioning problem. Given the following greedy heuristic: sort the set X in decreasing order. For each element of $X[i]$ with $i = 1$ to n, assign it to the set with the smallest current sum. What is the time complexity of this heuristic?

Exercise 1.3 PTAS class of approximation. Can the maximum clique problem be approximated by any constant factor?

Exercise 1.4 Size of an instance versus its structure. The size of an instance is not the unique indicator that describes the difficulty of a problem, but also its structure. For a given problem, small instances cannot be solved to optimality while large instances may be solved exactly. Show for some classical optimization problems (e.g., satisfiability, knapsack, bin packing, vehicle routing, and set covering) that some small instances are not solved exactly while some large instances are solved to optimality by the state-of-the-art exact optimization methods.

Exercise 1.5 2-Approximation for the vertex covering problem. The vertex cover problem consists in finding the minimal vertex cover in a given graph. A vertex cover for an undirected graph $G = (V, E)$ is a subset S of its vertices such that each edge has at least one end point in S. For each edge (i, j) in E, one of i or j must be an element of S. Show that it is very easy to find a simple greedy heuristic that guarantees a 2-approximation factor. The complexity of the heuristic must be in the order of $O(m)$ where m is the number of edges.

Exercise 1.6 Specific heuristic. Let us consider the number partitioning problem presented in Example 1.15. Propose a *specific* heuristic to solve this problem. Consider the difference of number pairs in a decreasing order until only one number remains. For instance, if the input instance is (16, 13, 11, 10, 5), the first pair to consider will be (16, 13). Then, their difference is included in the input instance, that is, (3,11,10,5), where 3 represents the partition {16} and {13}.

Exercise 1.7 Representation for constrained spanning tree problems. Given a connected graph $G = (V, E)$, a spanning tree is a minimum size connected and maximum size acyclic subgraph of G spanning all the vertices of V. The large numbers of applications have required the study of variants of the well-known minimum spanning tree problem (MSTP). Given a connected graph $G = (V, E)$, with $n = |V|$, $m = |E|$, a spanning tree is a connected and acyclic subgraph of G spanning all the vertices of V with $n - 1$ edges. Although the MSTP, the more studied problem involving spanning tree, can be solved in polynomial time, the outstanding importance of spanning trees in telecommunication or integrated circuit network design, biology, or computer science has required the development of more complex problems and often NP-hard variants. Indeed, adding some constraints (e.g., node degree, graph diameter) to the MSTP problem makes it NP-hard.

For instance, in the hop-constrained minimum spanning tree problem (HMSTP), the unique path from a specified root node, node 0, to any other node has no more than H hops (edges). Propose an encoding for the HMSTP problem.

Exercise 1.8 Indirect encoding for the bin packing problem. We consider in this exercise the bin packing problem (see Example 1.16). Let us consider an indirect encoding based on permutations. Propose a decoding function of permutations that generates feasible solutions to the bin packing problem. This representation belongs to the one-to-many class of encodings. Analyze the redundancy of this encoding. How the degree of redundancy grows with the number of bins?

Exercise 1.9 Encoding for the equal piles problem. Given a set of n one-dimensional objects of different sizes x_i ($i = 1, \ldots, n$), the objective is to distribute the objects into k piles G_l ($l = 1, \ldots, k$) such that the heights of the piles are as similar as possible:

$$f = \sum_{l=1}^{k} |s_l - S|$$

where $s_l = \sum_{j \in G_l} x_j$ is the sum of sizes for a given subset l and $S = \sum_{i=1}^{n} x_i / k$ is the average size of a subset. The problem is NP-hard even for $k = 2$. Propose a representation of solutions to tackle this problem.

Exercise 1.10 Greedy heuristic for the knapsack problem. In Example 1.40, a greedy heuristic for the 0–1 knapsack problem has been proposed. What will be the characteristic of the algorithm if the order of the elements when sorted by increasing weight is the same compared to their utilities when sorted by decreasing value?

Exercise 1.11 Greedy algorithms for vehicle routing problems. Vehicle routing problems represent very important applications in the area of logistics and transportation [775]. VRP are some of the most studied problems in the combinatorial optimization domain. The problem was introduced more than four decades ago by Dantzig and Ramser. The basic variant of the VRP is the capacitated vehicle routing problem. CVRP can be defined as follows: Let $G = (V, A)$ be a graph where V the set of vertices represents the customers. One vertex represents the depot with a fleet of m identical vehicles of capacity Q. We associate with each customer v_i a demand q_i and with each edge (v_i, v_j) of A a cost c_{ij} (Fig. 1.34). We have to find a set of routes where the objective is to minimize the total cost and satisfy the following constraints:

- For each vehicle, the total demand of the assigned customers does not exceed its capacity Q.

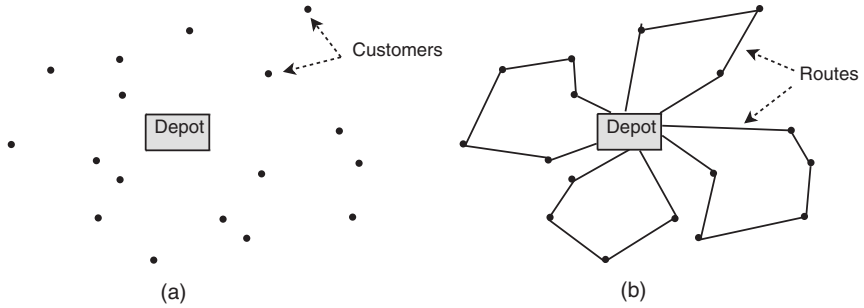

FIGURE 1.34 The capacitated vehicle routing problem. (a) From the depot, we serve a set of customers. (b) A given solution for the problem.

- Each route must begin and end at the depot node.
- Each customer is visited exactly once.

Define one or more greedy algorithms for the CVRP problem. Give some examples of constraints or more general models encountered in practice. For instance, one can propose

- Multiple depot VRP (MDVRP) where the customers get their deliveries from several depots.
- VRP with time windows (VRPTW), in case a time window (start time, end time, and service time) is associated with each customer.
- Periodic VRP (PVRP) in which a customer must be visited a prescribed number of times within the planning period. Each customer specifies a set of possible visit day combinations.
- Split delivery VRP (SDVRP) where several vehicles serve a customer.
- VRP with backhauls (VRPB) in which the vehicle must pick something up from the customer after all deliveries are carried out.
- VRP with pick ups and deliveries (VRPPS) if the vehicle picks something up and delivers it to the customer.

Exercise 1.12 Greedy algorithms for the Steiner tree problem. The goal of this exercise is to design a greedy heuristic for the Steiner tree problem. *Hints*: (a) Construct a graph where the nodes are terminals. The weight associated with an edge connecting two terminals represents the value of the shortest path between those terminals in the original graph. (b) Generate a spanning tree from this graph using the Kruskal algorithm. (c) From the edges obtained from the spanning tree, redesign the original graph using those selected edges and find the Steiner tree.

Exercise 1.13 Greedy algorithms for the bin packing problem. The bin packing problem is a well-known combinatorial problem with many applications such as container or pellet loading, loading trucks with weight capacity, and creating file backup in removable media. Objects of different volumes must be packed into a finite number of bins of capacity C in a way that minimizes the number of bins used. There are many variations of this problem such as 3D or 2D packing, linear packing, pack by volume, and pack by weight.

Let us solve the one-dimensional bin packing problem (Fig. 1.35). Given a finite collection of n weights $w_1, w_2, w_3, \ldots, w_n$ and a collection of identical bins with capacity C (which exceeds the largest of the weights), the problem is to find the minimum number k of bins into which the weights can be placed without exceeding the bin capacity C. An example of a greedy algorithm is the *first fit algorithm* that places each item into the first bin in which it will fit. It requires $\Theta(n \log n)$ time. Propose some improvements of this greedy algorithm. *Hint*: For example, a sorting of the elements may be done before the packing.

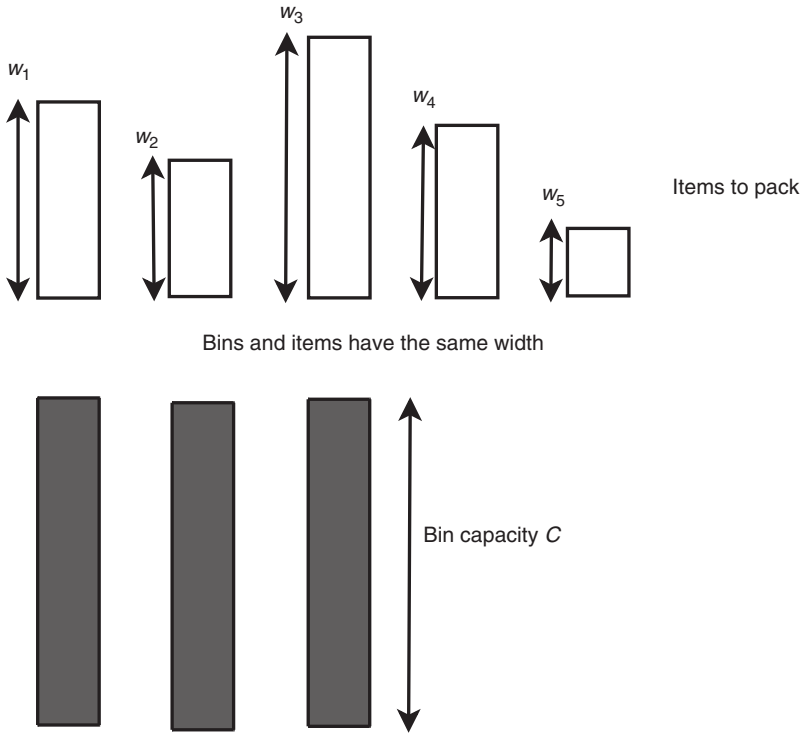

FIGURE 1.35 The one-dimensional bin packing problem.

Exercise 1.14 Random permutation. Design an efficient procedure for the generation of a random permutation.

Exercise 1.15 Generalized TSP problem. The generalized traveling salesman problem (GTSP) is a generalization of the well-known traveling salesman problem. Given an undirected complete graph $G = (V, E)$, where V represents the set of cities. In the GTSP, the set of nodes V is partitioned into m groups $W_1, W_2, \ldots, W_m$ where $0 < m \leq n$ and $W_1 \cup W_2 \cup \cdots \cup W_n = V$. Each city $v_i \in V$ belongs to one and only one group. The groups are disjoint, that is, $\forall i \neq j, W_i \cap W_j = \varnothing$. The objective is to find a minimum tour in terms of distance containing exactly one node from each group W_i. Propose a representation for the problem.

Exercise 1.16 Indirect encoding for the JSP. The job-shop scheduling problem has been defined in Example 1.33. Given the following indirect encoding: an array of j elements, each one being composed of a list of allocations of machines on which the operations are to be executed (Fig 1.36). Propose a decoder that generates a feasible schedule.

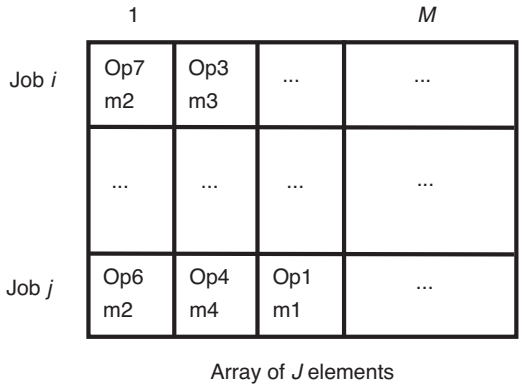

FIGURE 1.36 Indirect encoding for the JSP problem.

Exercise 1.17 Objective function for the vehicle routing problem. For the vehicle routing problem, a solution s may be represented by the assignment of the customers to the vehicles. A neighborhood may be defined as the move of one customer from one vehicle to another. Show that computing the incremental objective function consisting in minimizing the total distance is a difficult procedure.

Exercise 1.18 Objective function for the feature selection problem within classification. The feature selection problem has a large variety of applications in many domains such as data mining. In the feature selection problem, the objective is to find a subset of features such that a classification algorithm using only those selected feature provides the best performances. Any supervised classification algorithm may be used such as the support vector machines, decision trees, or naive Bayes. Given a set of instances I. Each instance is characterized by a large number of d features $F = \{f_1, f_2, \ldots, f_d\}$. Each instance is labeled with the class it belongs to. The problem consists in finding the optimal subset $S \subseteq F$. Propose an objective function for this problem.

Exercise 1.19 Domination analysis of metaheuristics. In combinatorial optimization problems, the domination analysis of a metaheuristic is defined by the number of solutions of the search space S that are dominated by the solution obtained by the metaheuristic. Suppose a metaheuristic H that generates the solution s_H from the search space S. The dominance associated with the metaheuristic H is the cardinality of the set $\{s \in S : f(s) \geq f(s_H)\}$. If the metaheuristic has obtained the global optimal solution s^*, the dominance $\text{dom}(H) = |S|$. Give a critical analysis of this performance indicator.

Exercise 1.20 Performance evaluation in dynamic optimization problems. To evaluate a metaheuristic in a dynamic problem, using classical measures such as the best found solution is not sufficient. Indeed, the concept of solution quality is changing

over time. Propose a performance measure to deal with the quality of solutions in dynamic optimization problems with an *a priori* known time of the environment change t_i, $i \in [1, \ldots, n]$.

Exercise 1.21 Constraint handling. Given an objective function f to minimize and m constraints to satisfy. The new objective function f' that handles the constraints is defined as follows:

$$f'(x) = \begin{cases} f(x) & \text{if } x \text{ is a feasible solution} \\ K - \sum_{i=1}^{s} \frac{K}{m} & \text{otherwise} \end{cases}$$

where x is a solution to the problem, s is the number of satisfied constraints, and K is a very large constant value (e.g., $K = 10^9$). To which class of constraint handling strategies this approach belongs? Perform a critical analysis of this approach.

Exercise 1.22 Evaluation of metaheuristics as multiobjective optimization. Many quantitative and qualitative criteria can be considered to evaluate the performance of metaheuristics: efficiency, effectiveness, robustness, simplicity, flexibility, innovation, and so on. Let us consider only two quantitative criteria: efficiency and effectiveness. Figure 1.37 plots for two metaheuristics, Meta_1 and Meta_2, the evolution in time of the quality of best found solutions. According to each criterion, which

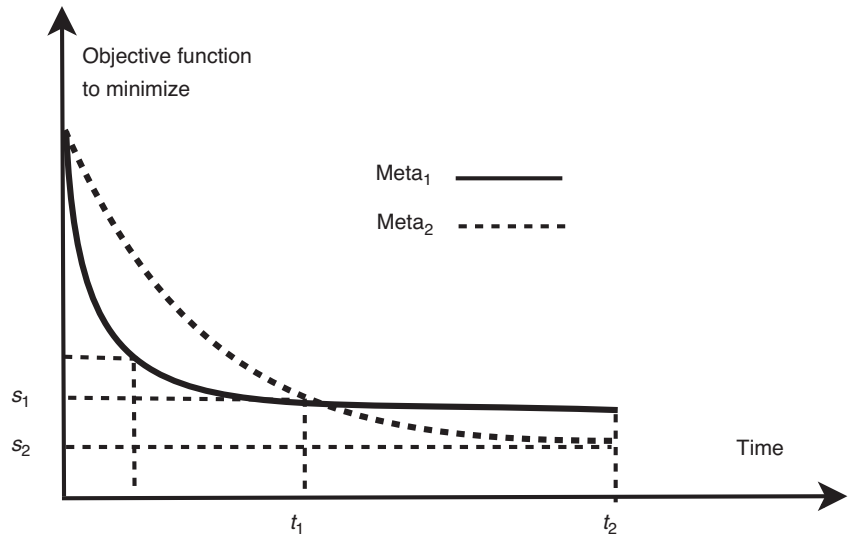

FIGURE 1.37 Efficiency versus effectiveness in the performance evaluation of metaheuristics. In terms of Pareto dominance optimality, no metaheuristic dominates the other one.

metaheuristic may be considered the best one? No metaheuristic dominates the other one for the two criteria. Propose some aggregations of the two criteria that generate a total ordering of metaheuristics. How can we deal with the qualitative criteria?

Exercise 1.23 Theoretical versus experimental evaluation. In comparing the theoretical and the experimental approach of the performance evaluation of a metaheuristic, one can make the following two statements:

- The theoretical approach gives more certain conclusions than the experimental approach.
- The experimental approach gives more certain conclusions than the theoretical approach.

Show that the two statements may be correct.

Exercise 1.24 Black box versus white box for metaheuristic software. Explain why the black box approach for software frameworks is not yet well suited for metaheuristics to solve general optimization problems. For which class of optimization problems, the black box approach may be appropriate for metaheuristics? Compare with the field of continuous linear programming (mathematical programming) and constraint programming.

Exercise 1.25 Software for metaheuristics. Analyze the following software for metaheuristics: ParadisEO, MATLAB optimization module, PISA, Localizer++, HotFrame, GAlib, CMA-ES, ECJ, BEAGLE, GENOCOP III, OpTech, Templar, iOpt, Mozart/Oz, GPC++, and EasyLocal++ in terms of target optimization problems, available metaheuristic algorithms, available hybrid metaheuristics, parallel models of metaheuristics, target architectures, software characteristics (program code, callable black box package, and object-oriented white box library). Which of the previously cited softwares can be considered as a software framework rather than a program or a callable package or library?