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

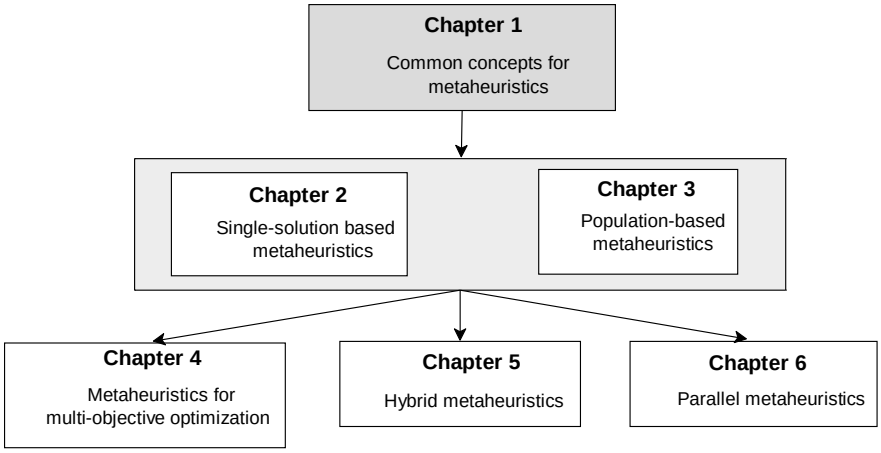


FIGURE P.1 Relationship between the different chapters of the book. The graph presents the dependencies between the chapters.

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

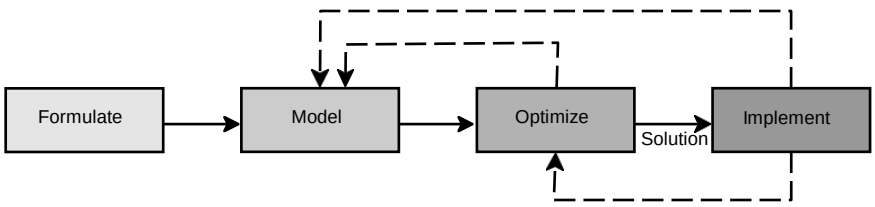


FIGURE 1.1 The classical process in decision making: formulate, model, solve, and implement. In practice, this process may be iterated to improve the optimization model or algorithm until an acceptable solution is found. Like life cycles in software engineering, the life cycle of optimization models 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 $f: S \rightarrow \mathbb{R}$ 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 $-f$.

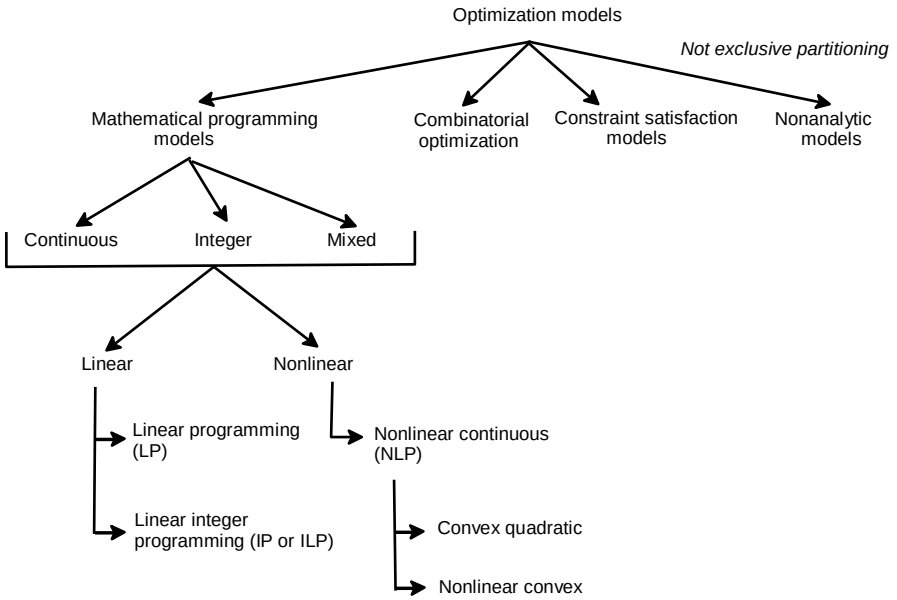


FIGURE 1.2 Classical optimization models. The different classes are possibly overlapping.

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	D 5	D 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 D 21.

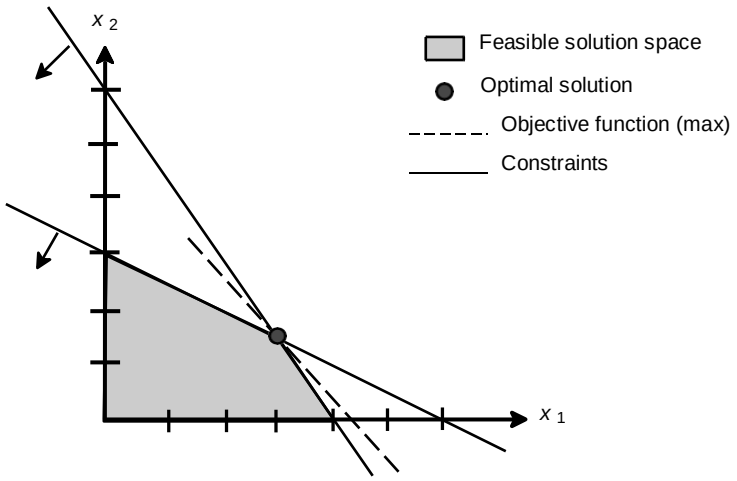


FIGURE 1.3 Graphical illustration of the LP model and its resolution.

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 $f : S \subset \mathbb{R}^n \rightarrow \mathbb{R}$ 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).

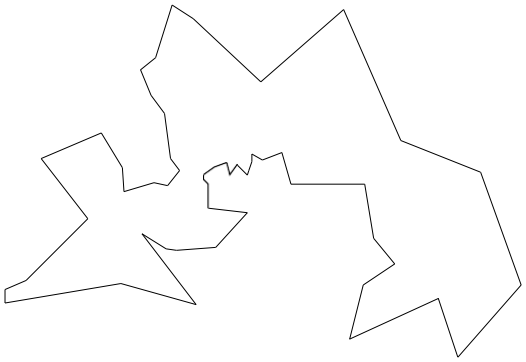


FIGURE 1.4 TSP instance with 52 cities.

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* $\boldsymbol{f}(n) = O(g(n))$ *if there exist positive constants* n_0 *and* c *such that* $\forall\, n > n_0, \boldsymbol{f}(n) \leq c \cdot g(n)$.

In this case, the function $\boldsymbol{f}(n)$ 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:

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

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-K and the Big-Ω notations.

Definition 1.5 Big-K notation. *An algorithm has a complexity $f(n) = K(g(n))$ if there exist positive constants n_0 and c such that $\forall n > n_0$, $f(n) \geq c\, g(n)$. 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-© notation. *An algorithm has a complexity* $\boldsymbol{f}(n) = ©(g(n))$ *if there exist positive constants* n_0, c_1, *and* c_2 *such that* $\forall n > n_0, c_1 \cdot g(n) \leq \boldsymbol{f}(n) \leq c_2 \cdot g(n)$. *The complexity of the algorithm* $\boldsymbol{f}(n)$ *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-*K* and Big-© complexities. The Big-© 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?"

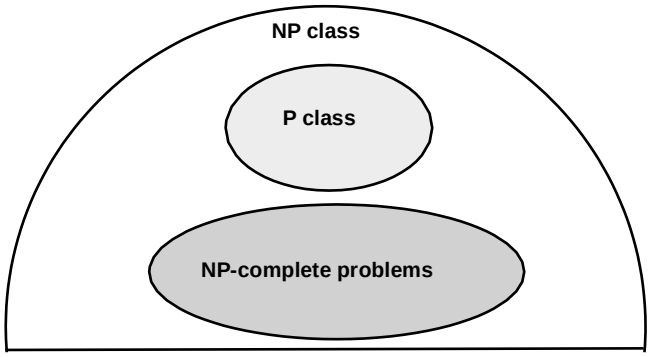


FIGURE 1.6 Complexity classes of decision problems.

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

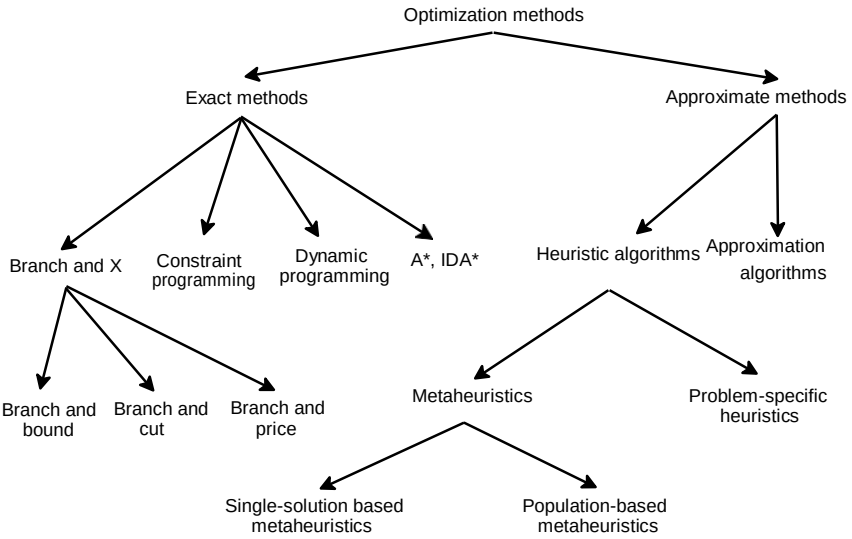


FIGURE 1.7 Classical optimization methods.

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 ϱ-approximation algorithm generates an approximate solution a not less than a factor ϱ times the optimum solution s [793].

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

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

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

[20]In a minimization context.

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

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

Example 1.16 ϱ-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 $\mathbb{O}(n, \log(n))$. 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* $(1+\varrho)$*-approximation algorithm for any fixed* $\varrho > 0$.

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

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 $O(n^3/\varrho)$. 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

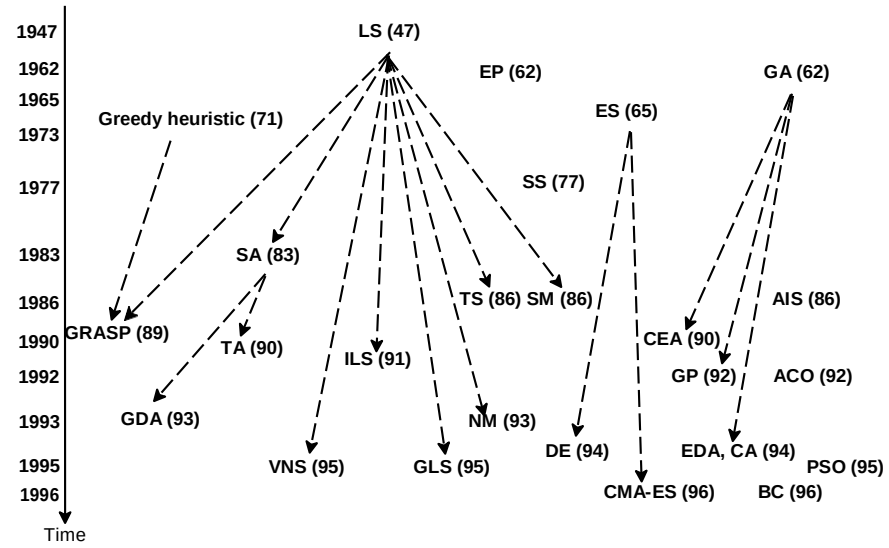


FIGURE 1.8 Genealogy of metaheuristics. The application to optimization and/or machine learning is taken into account as the original date.

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

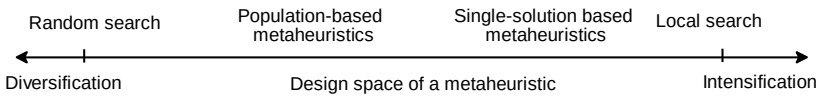


FIGURE 1.9 Two conflicting criteria in designing a metaheuristic: exploration (diversification) versus exploitation (intensification). In general, basic single-solution based metaheuristics are more exploitation oriented whereas basic population-based metaheuristics are more exploration oriented.

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\{e/e ∈ s}) ;
   /* next element selected from the set E minus already selected elements */
   If s ∪ e_i ∈ F Then /* test the feasibility of the solution */
      s = s ∪ 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.

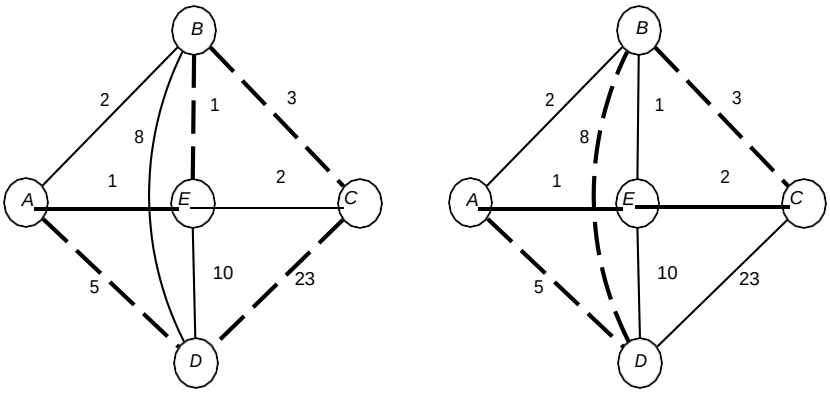


FIGURE 1.10 Illustrating a greedy algorithm for the TSP using a static heuristic. An element is associated with an edge of the graph, and the local heuristic consists in choosing the nearest neighbor. The obtained solution is (A– E – B – C – D –A) with a total cost of 33, whereas a better solution with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

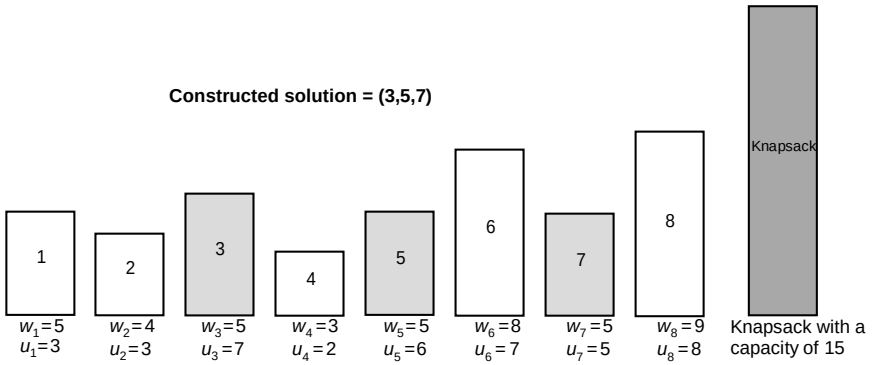


FIGURE 1.11 Illustrating a greedy algorithm for the knapsack problem. An element is associated with an object, and the local heuristic consists in choosing an element minimizing the ratio w_i/u_i. The final solution may not be optimal.

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].

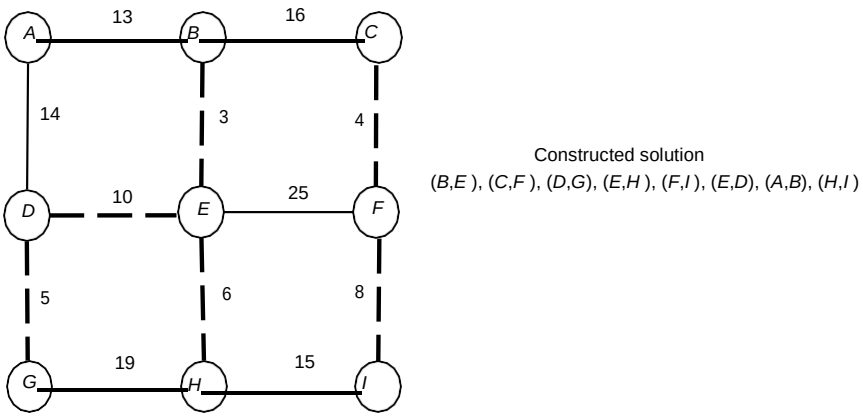


FIGURE 1.12 Illustrating a greedy algorithm for the spanning tree problem. The edge (*A*, *D*) has not been selected even if it is less costly than the edge (*H*, *I*) because it generates a nonfeasible solution (a cycle).

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.

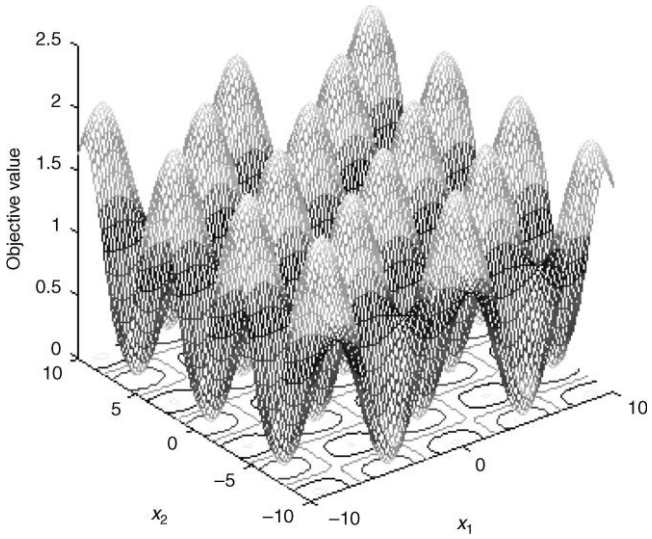


FIGURE 1.13 The Griewangk multimodal continuous function.

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

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

where $x = (x_1, x_2, \ldots, x_N)$, 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

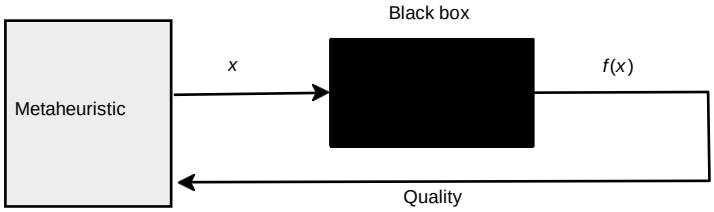


FIGURE 1.14 Black box scenario for the objective function.

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.

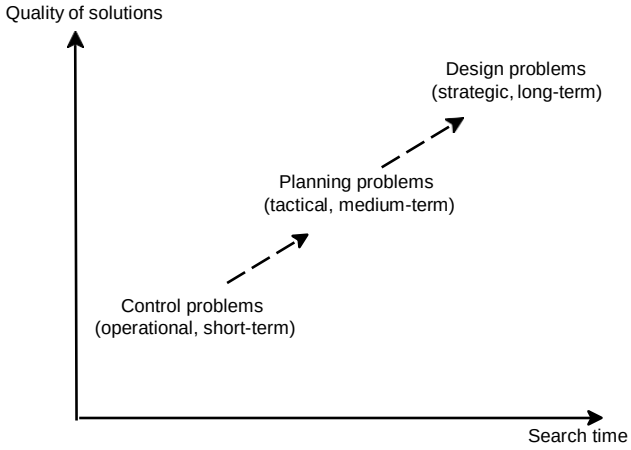


FIGURE 1.15 Different classes of problems in terms of the trade-off between quality of solutions and search time: design (strategic, long-term), planning (tactical, medium-term), control (operational, short-term).

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

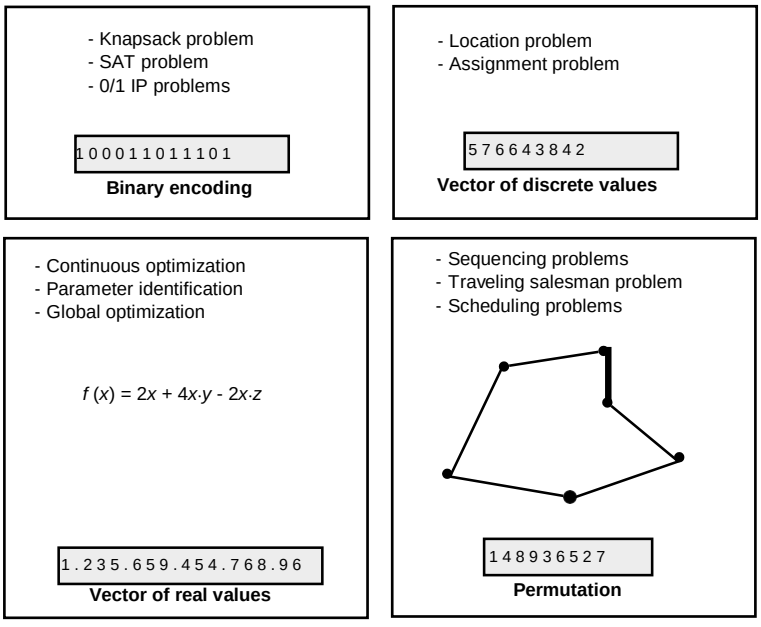


FIGURE 1.16 Some classical encodings: vector of binary values, vector of discrete values, vector of real values, and permutation.

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 assign- ment 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* × *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.

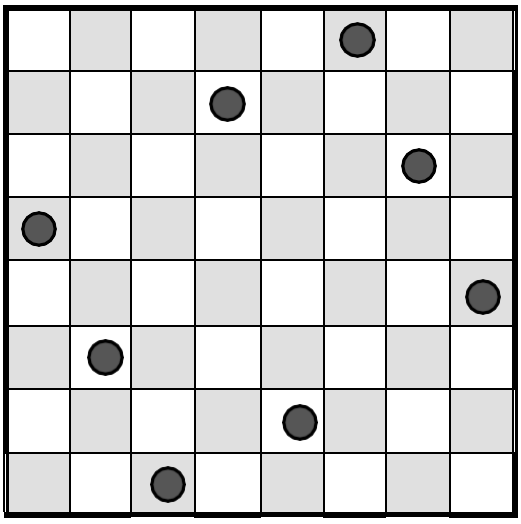


FIGURE 1.17 A solution for the 8-Queens problem represented by the permutation (6,4,7,1,8,2,5,3).

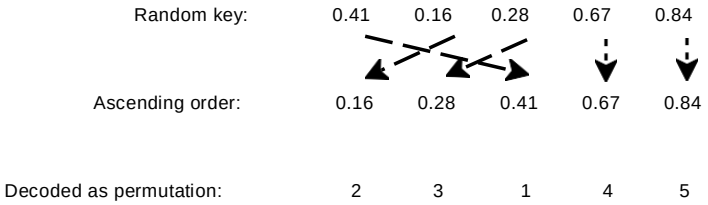


FIGURE 1.18 Random-key encoding and decoding.

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

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

where $|W\rangle$ 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 $1|$ 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] $\alpha^2 + \beta^2 = 1$.

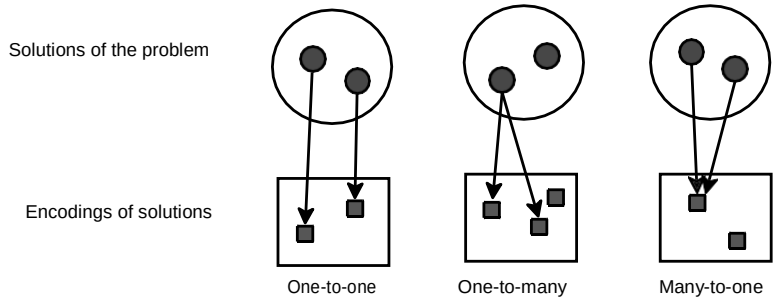


FIGURE 1.19 Mapping between the space of solutions and the space of encodings.

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 $\bigcup s_i = S$ and $s_i \cap s_j \neq \emptyset$. 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 $C_{\max}(x) = \max_{1 \le m \le M} E_m(x)$. 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

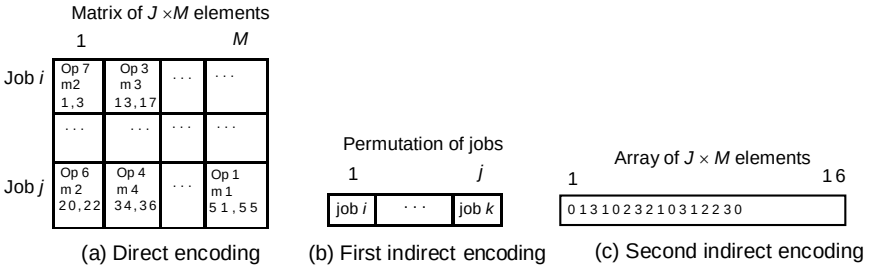


FIGURE 1.20 Direct and indirect encodings for the job-shop scheduling problem.

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, $f : S \to \mathbb{R}$. 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, $d : R \to S$, 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.

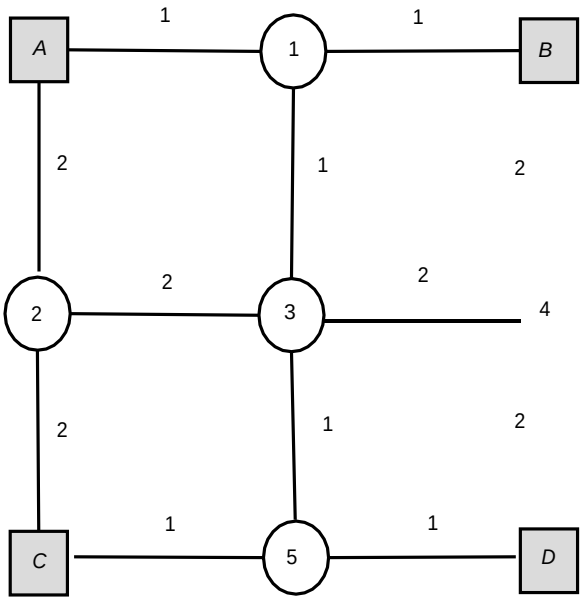


FIGURE 1.21 Instance for the Steiner tree problem: $T = \{A, B, C, D\}$.

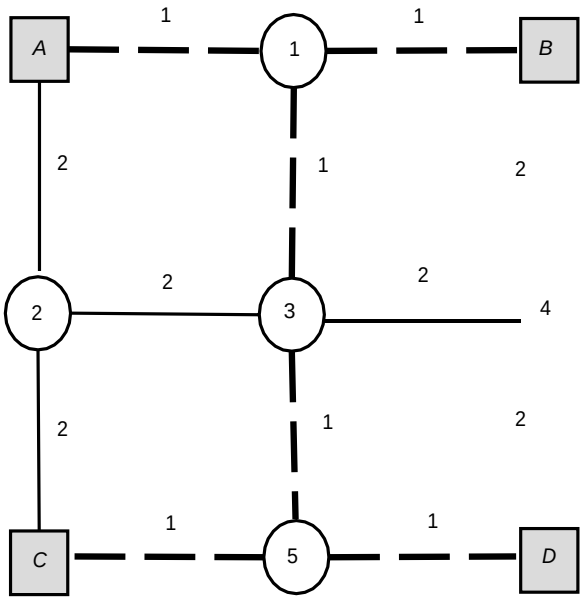


FIGURE 1.22 Optimal solution represented by the set {1, 3, 5}.

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.

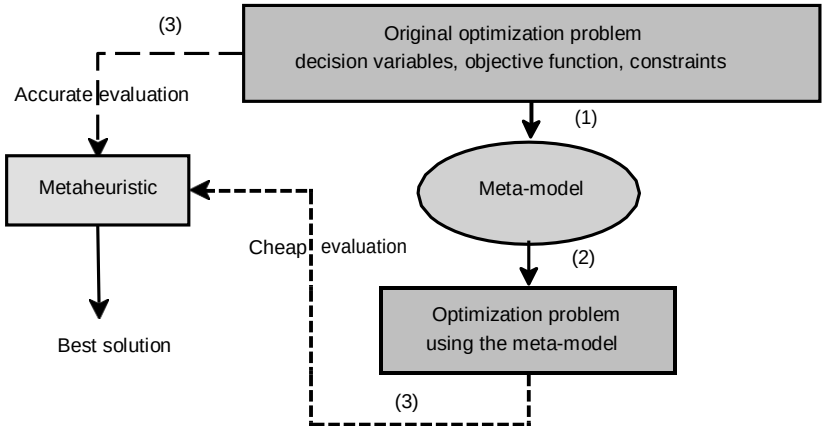


FIGURE 1.23 Optimization using a meta-model. Once the model is constructed (1), the metaheuristic can use either the meta-model (2) or the alternative between the two models (original and approximate) for a better compromise between accuracy and efficiency (3).

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 $\boldsymbol{f}(x)$ is defined as follows:

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

  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:

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

  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 $\frac{u_i}{w_i}$;
  $s' = s' \setminus e_i$;
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.

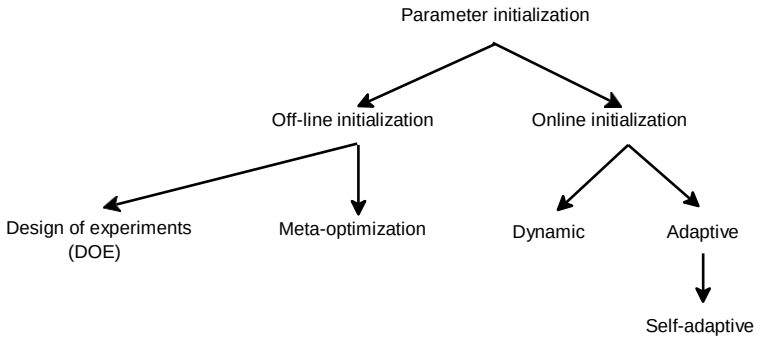


FIGURE 1.24 Parameter initialization strategies.

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.

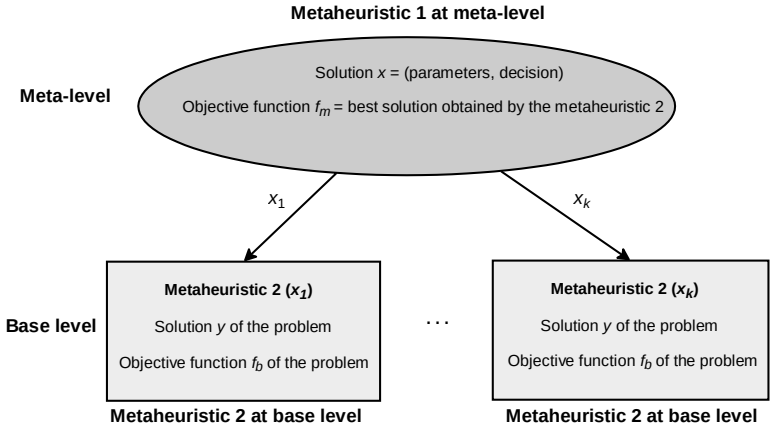


FIGURE 1.25 Meta-optimization using a meta-metaheuristic.

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.).

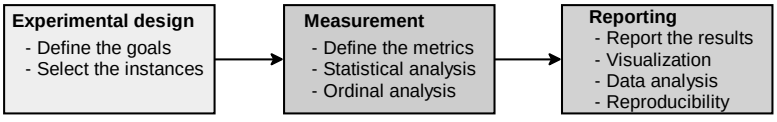


FIGURE 1.26 Different steps in the performance analysis of a metaheuristic: experimental design, measurement, and reporting.

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	$f(x) = \frac{4000}{} \sum_{i=1}^{D} x_i^2 - \prod_{i=1}^{D} \cos\left(\frac{x_i}{\sqrt{i}}\right) + 1$
Schaffer's f6	$f(x) = 0.5 - \frac{\sin\left(x_1^2 + x_2^2\right)^{0.5} - 0.5}{\left(1 + 0.001\left(x_1^2 + x_2^2\right)\right)^2}$
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 algorithm s. 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

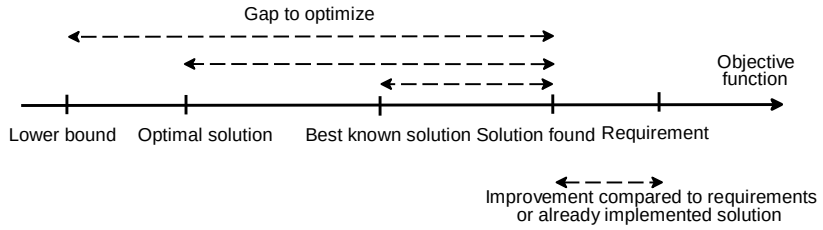


FIGURE 1.27 Performance assessment of the quality of the solutions. We suppose a minimization problem.

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: $|f(s) - f(s^*)| / |f_{worst} - f(s^*)|$ [838] or $|f(s) - f(s^*)| / |E_{unif}(f) - f(s^*)|$ [849], where f_{worst} represents the worst objective cost for the tackled instance[43], and $E_{unif}(f)$[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.

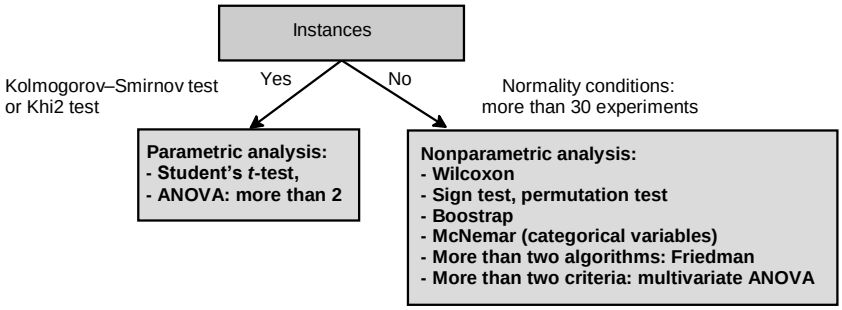


FIGURE 1.28 Statistical analysis of the obtained results.

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.

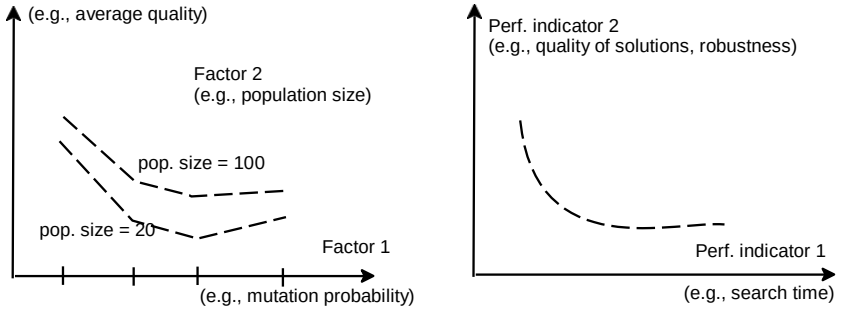


FIGURE 1.29 (a) Interaction plots analyze the effect of two factors (parameters, e.g., mutation probability, population size in evolutionary algorithms) on the obtained results (e.g., solution quality, time). (b) Scatter plots analyze the trade-off between the different performance indicators (e.g.,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

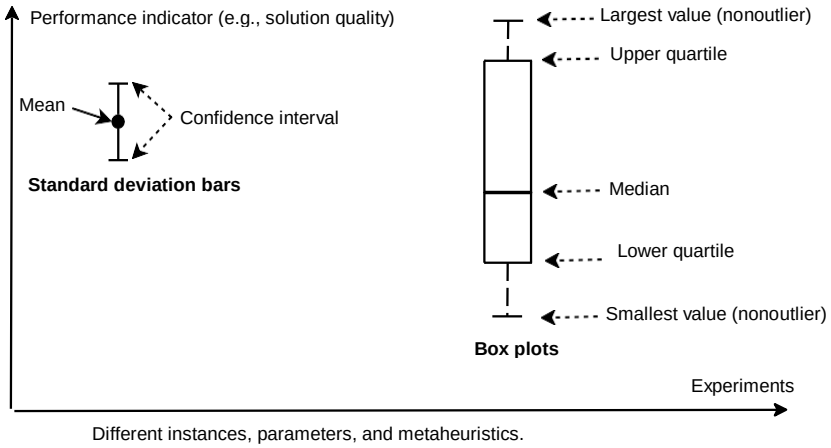


FIGURE 1.30 Some well-known visualization tools to report results: deviation bars, confidence intervals.

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 $i \in [1, \ldots, n]$.

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, multi-objective 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.

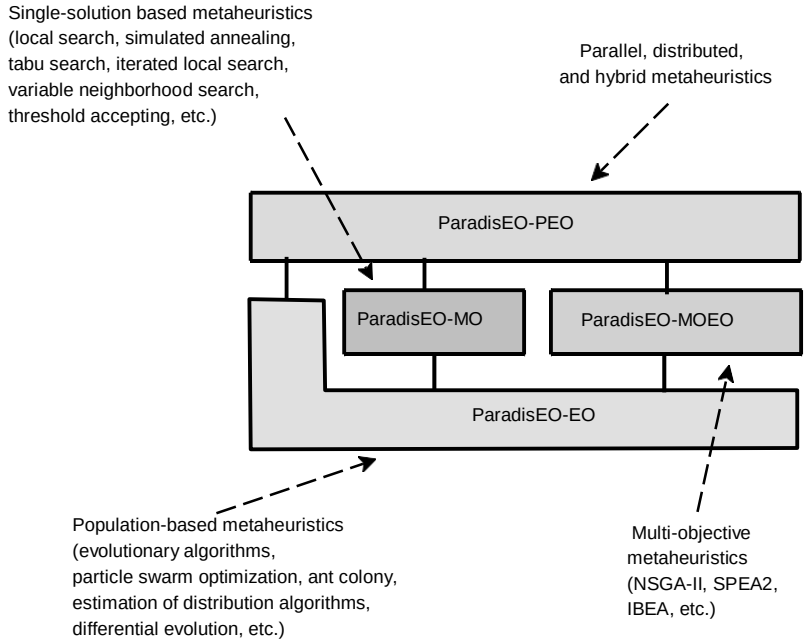


FIGURE 1.31 The different unified modules of the ParadisEO framework.

- **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 multi-layered and modular allowing to achieve the objectives quoted above (Fig. 1.32).

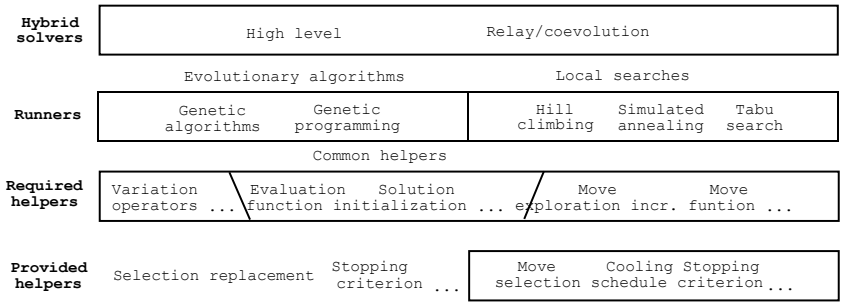


FIGURE 1.32 Architecture of the ParadisEO framework.

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

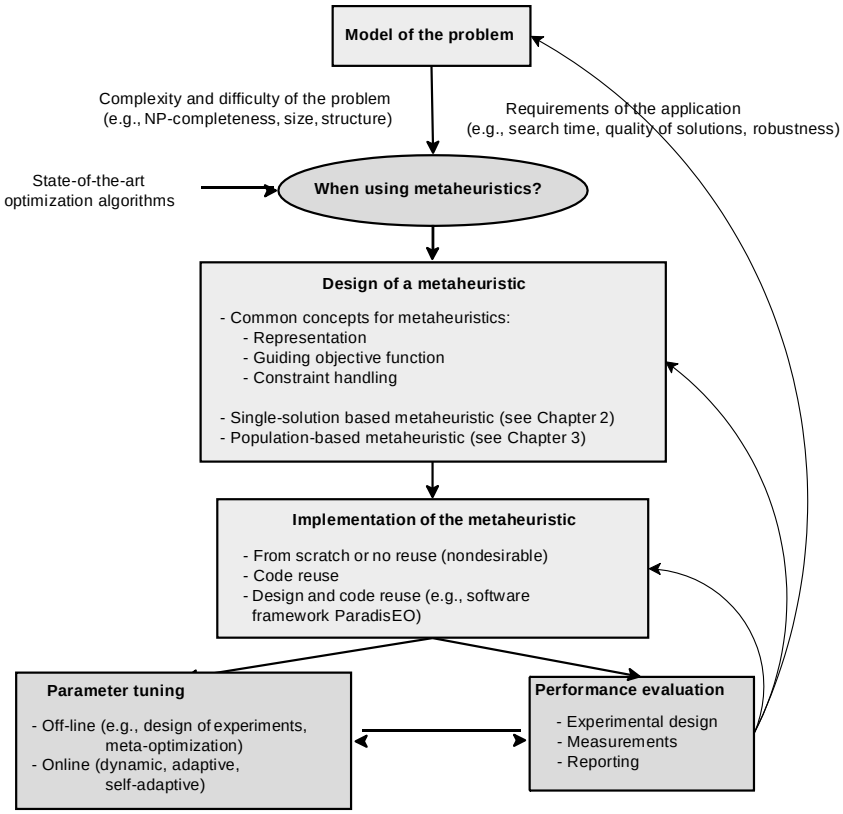


FIGURE 1.33 Guidelines for solving a given optimization problem.

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 $k \geq 2$. 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.

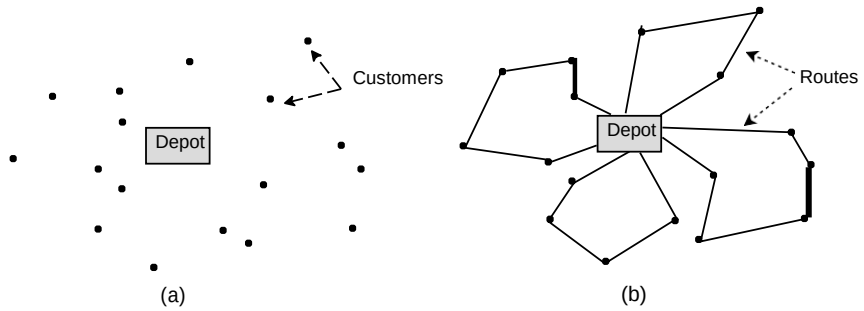


FIGURE 1.34 The capacitated vehicle routing problem. (a) From the depot, we serve a set of customers. (b) A given solution for the problem.

- 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 ©(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.

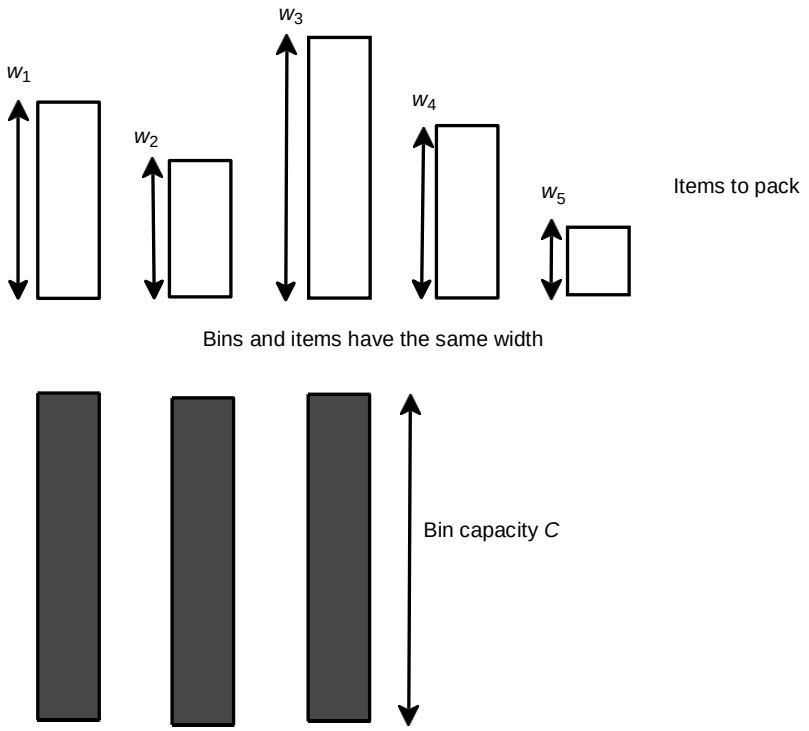


FIGURE 1.35 The one-dimensional bin packing problem.

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 $W_1 \cup W_2 \cup \ldots \cup W_n = V$. Each city $v_i \in V$ belongs to one and only one group. The groups are disjoint, that is, $\forall i, j \neq j$, $W_i \cap W_j = \emptyset$. 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.

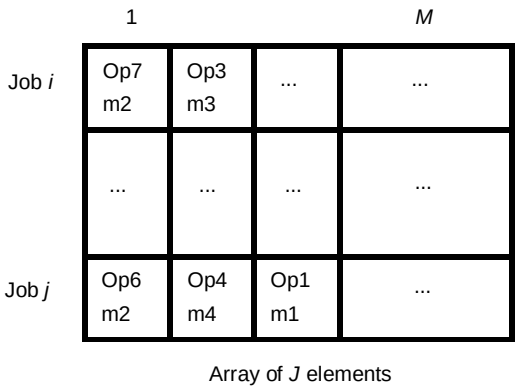


FIGURE 1.36 Indirect encoding for the JSP problem.

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 $\{s \in S : f(s) \preceq f(s_H)\}$. If the metaheuristic has obtained the global optimal solution s^*, the dominance $\mathrm{dom}(H) = |\{\ldots\}|$. 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

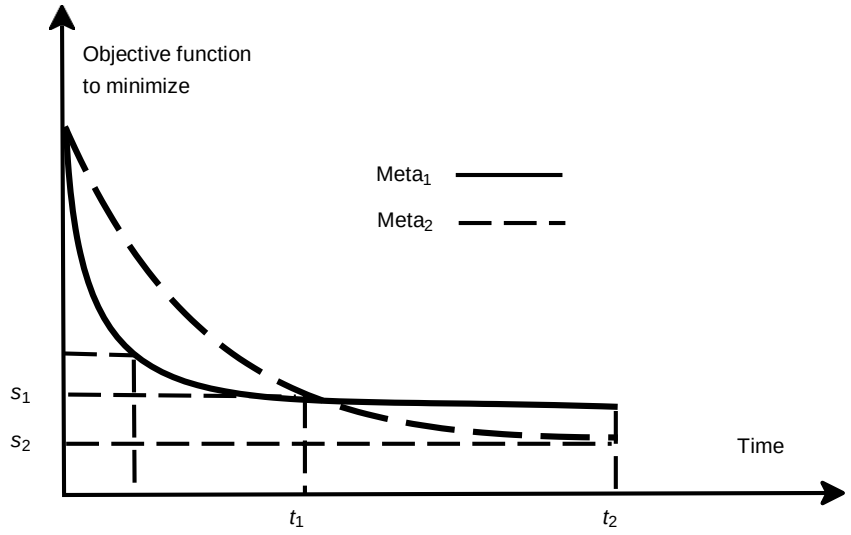


FIGURE 1.37 Efficiency versus effectiveness in the performance evaluation of metaheuristics. In terms of Pareto dominance optimality, no metaheuristic dominates the other one.

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?

CHAPTER 2

Single-Solution Based Metaheuristics

While solving optimization problems, single-solution based metaheuristics (S-metaheuristics) improve a single solution. They could be viewed as "walks" through neighborhoods or search trajectories through the search space of the problem at hand [163]. The walks (or trajectories) are performed by iterative procedures that move from the current solution to another one in the search space. S-metaheuristics show their efficiency in tackling various optimization problems in different domains.

In this chapter, a unified view of the common search concepts for this class of metaheuristics is presented. Then, the main existing algorithms, both from the design and the implementation point of view, are detailed in an incremental way.

This chapter is organized as follows. After the high-level template of S-metaheuristics is presented, Section 2.1 details the common search components for S-metaheuristics: the definition of the neighborhood structure, the incremental evaluation function, and the determination of the initial solution. The concept of very large neighborhoods is also presented. Section 2.2 discusses the landscape analysis of optimization problems. Sections 2.3, 2.4, and 2.5 introduce, in an incremental manner, the well-known S-metaheuristics: local search, simulated annealing, and tabu search. Then, Sections 2.6, 2.7, and 2.8 present, respectively, the iterative local search, the variable neighborhood search, and the guided local search algorithms. In Section 2.9, other S-metaheuristics are dealt with, such as GRASP, the noisy and the smoothing methods. Finally, Section 2.10 presents the ParadisEO–MO (moving objects) module of the ParadisEO framework that is dedicated to the implementation of S-metaheuristics. Some design and implementations of S-metaheuristics such as local search, simulated annealing, tabu search, and iterated local search are illustrated.

2.1 COMMON CONCEPTS FOR SINGLE-SOLUTION BASED METAHEURISTICS

S-metaheuristics iteratively apply the generation and replacement procedures from the current single solution (Fig. 2.1). In the generation phase, a set of candidate solutions are generated from the current solution s. This set $C(s)$ is generally obtained by local

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

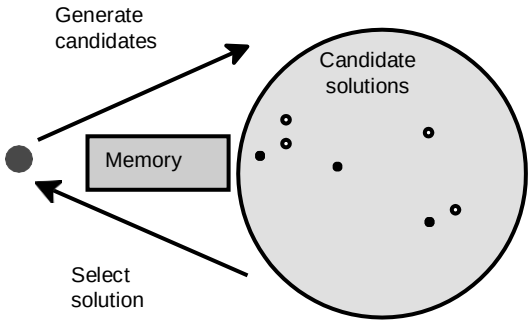


FIGURE 2.1 Main principles of single-based metaheuristics.

transformations of the solution. In the replacement phase,[1] a selection is performed from the candidate solution set $C(s)$ to replace the current solution; that is, a solution $s' \in C(s)$ is selected to be the new solution. This process iterates until a given stopping criteria. The generation and the replacement phases may be *memoryless*. In this case, the two procedures are based only on the current solution. Otherwise, some history of the search stored in a memory can be used in the generation of the candidate list of solutions and the selection of the new solution. Popular examples of such S-metaheuristics are local search, simulated annealing, and tabu search. Algorithm 2.1 illustrates the high-level template of S-metaheuristics.

Algorithm 2.1 High-level template of S-metaheuristics.

Input: Initial solution s_0.
$t = 0$;
Repeat
  /* Generate candidate solutions (partial or complete neighborhood) from s_t */
  Generate($C(s_t)$) ;
  /* Select a solution from $C(s)$ to replace the current solution s_t */
  s_{t+1} = Select($C(s_t)$) ;
  $t = t + 1$;
Until Stopping criteria satisfied
Output: Best solution found.

The common search concepts for *all* S-metaheuristics are the definition of the *neighborhood* structure and the determination of the *initial solution.*

2.1.1 Neighborhood

The definition of the neighborhood is a required common step for the design of any S-metaheuristic. The neighborhood structure plays a crucial role in the performance

[1]Also named transition rule, pivoting rule, and selection strategy.

of an S-metaheuristic. If the neighborhood structure is not adequate to the problem, any S-metaheuristic will fail to solve the problem.

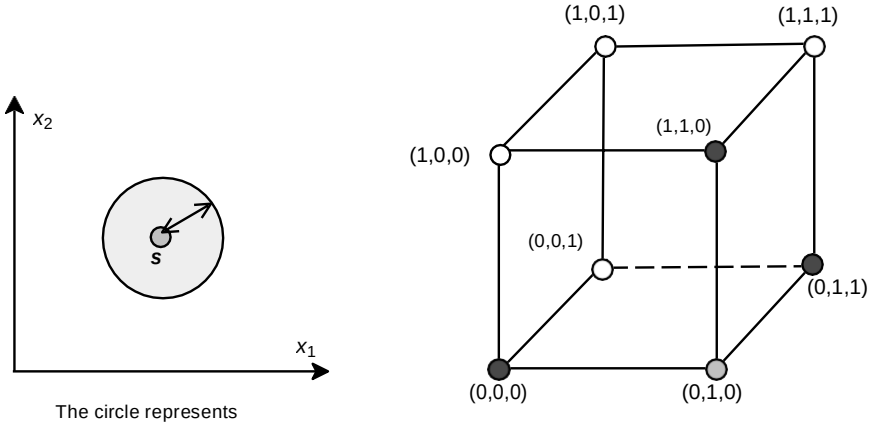
Definition 2.1 Neighborhood. *A neighborhood function N is a mapping* $N : S \rightarrow 2^S$ *that assigns to each solution s of S a set of solutions* $N(s) \subset S$.

A solution s' in the neighborhood of s ($s' \in N(S)$) is called a *neighbor* of s. A neighbor is generated by the application of a *move* operator m that performs a small perturbation to the solution s. The main property that must characterize a neighborhood is *locality*. Locality is the effect on the solution when performing the move (perturbation) in the representation. When small changes are made in the representation, the solution must reveal small changes. In this case, the neighborhood is said to have a strong locality. Hence, a S-metaheuristic will perform a meaningful search in the landscape of the problem. Weak locality is characterized by a large effect on the solution when a small change is made in the representation. In the extreme case of weak locality, the search will converge toward a random search in the search space.

The structure of the neighborhood depends on the target optimization problem. It has been first defined in continuous optimization.

Definition 2.2 *The neighborhood* $N(s)$ *of a solution s in a continuous space is the ball with center s and radius equal to* ϱ *with* $\varrho > 0$.

Hence, one have $N(s) = \{s' \in R^n / \|s' - s\| < \varrho\}$. By using the Euclidean norm, we obtain $\|s' - s\| = \sqrt{(s'_1 - s_1)^2 + (s'_2 - s_2)^2 + \cdots + (s'_n - s_n)^2}$, which is the Euclidean distance between s' and s (Fig. 2.2).



The circle represents the neighborhood of s in a continous problem with two dimensions.

- Nodes of the hypercube represent solutions of the problem.
- The neighbors of a solution (e.g., (0,1,0)) are the adjacent nodes in the graph.

FIGURE 2.2 Neighborhoods for a continuous problem and a discrete binary problem.

If the gradient (derivates of the objective function) can be calculated or approximated, a steepest descent (or ascent) indicating the direction to take can be applied. The main parameter in this first strategy is the length of the move in the selected direction. Otherwise, a random or deterministic generation of a subset of neighbors is carried out. In random generation of a neighbor, the normal variable $(0, \sigma_i)$ is added to the current value, where σ_i represents the main parameter in this second strategy.

The concept of neighborhood can be extended to discrete optimization.

Definition 2.3 *In a discrete optimization problem, the neighborhood $N(s)$ of a solution s is represented by the set $\{s'/d(s', s) \leq \varrho\}$, where d represents a given distance that is related to the move operator.*

The neighborhood definition depends strongly on the representation associated with the problem at hand. Some usual neighborhoods are associated with traditional encodings.

The natural neighborhood for binary representations is based on the Hamming distance. In general, a distance equal to 1 is used. Then, the neighborhood of a solution s consists in flipping one bit of the solution. For a binary vector of size n, the size of the neighborhood will be n. Figure 2.2 shows the neighborhood for a continuous problem and a discrete binary problem. An Euclidean distance less than ϱ and a Hamming distance equal to 1 are used to define the neighborhood.

The Hamming neighborhood for binary encodings may be extended to any discrete vector representation using a given alphabet Σ. Indeed, the substitution can be generalized by replacing the discrete value of a vector element by any other character of the alphabet. If the cardinality of the alphabet Σ is k, the size of the neighborhood will be $(k - 1).\ n$ for a discrete vector of size n.

For permutation-based representations, a usual neighborhood is based on the swap operator that consists in exchanging (or swapping) the location of two elements s_i and s_j of the permutation. For a permutation of size n, the size of this neighborhood is $n(n - 1)/2$. This operator may also be applied to any linear representation. Figure 2.3 shows the neighborhood associated with a combinatorial optimization problem using a permutation encoding. The distance is based on the swap move operator.

Once the concept of neighborhood has been defined, the local optimality property of a solution may be given.

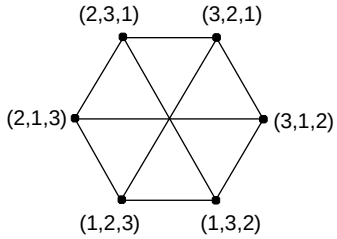


FIGURE 2.3 An example of neighborhood for a permutation problem of size 3. For instance, the neighbors of the solution (2, 3, 1) are (3, 2, 1), (2, 1, 3), and (1, 3, 2).

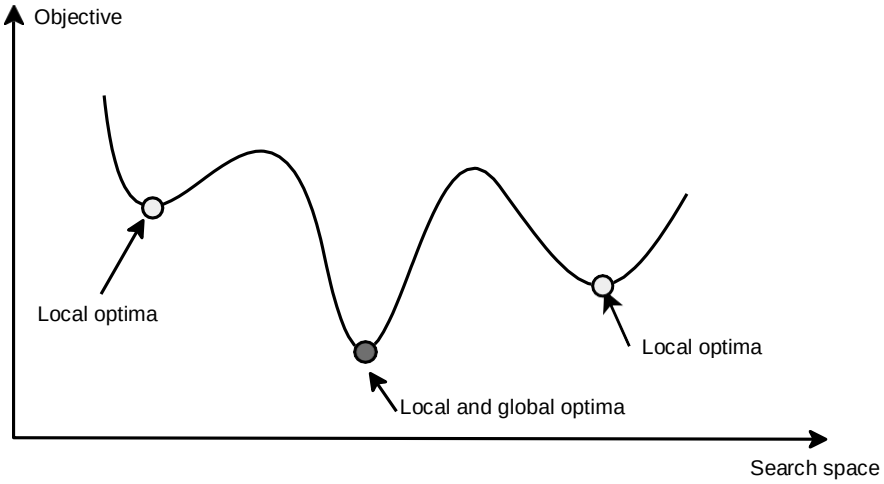


FIGURE 2.4 Local optimum and global optimum in a search space. A problem may have many global optimal solutions.

Definition 2.4 Local optimum. *Relatively to a given neighboring function N, a solution $s \in S$ is a local optimum if it has a better quality than all its neighbors; that is, $f(s) \leq f(s')$[2] for all $s' \in N(s)$ (Fig. 2.4).*

For the same optimization problem, a local optimum for a neighborhood N_1 may not be a local optimum for a different neighborhood N_2.

Example 2.1 ***k*-distance neighborhood versus *k*-exchange neighborhood.** For permutation problems, such as the TSP, the exchange operator (swap operator) may be used (Fig. 2.5). The size of this neighborhood is $n(n-1)/2$ where n represents the number of cities. Another widely used operator is the k-opt operator,[3] where k edges are removed from the solution and replaced with other k edges. Figure 2.5 (resp. Fig. 2.6) illustrates the application of the *2-opt operator* (resp. *3-opt operator*) on a tour. The neighborhood for the 2-opt operator is represented by all the permutations obtained by removing two edges. It replaces two directed edges (π_i, π_{i+1}) and (π_j, π_{j+1}) with (π_i, π_j) and (π_{i+1}, π_{j+1}), where $i \neq j-1, j, j+1 \; \forall i, j$. Formally, the neighbor solution is defined as

$$\pi'(k) = \pi(k) \quad \text{for } k \leq i \text{ or } k > j$$
$$\pi'(k) = \pi(i + j + 1 - k) \quad \text{for } i < k \leq j$$

The size of the neighborhood for the 2-opt operator is $[(n(n-1)/2) - n]$; all pairs of edges are concerned except the adjacent pairs.

[2]For a minimization problem.
[3]Also called k-exchange operator.

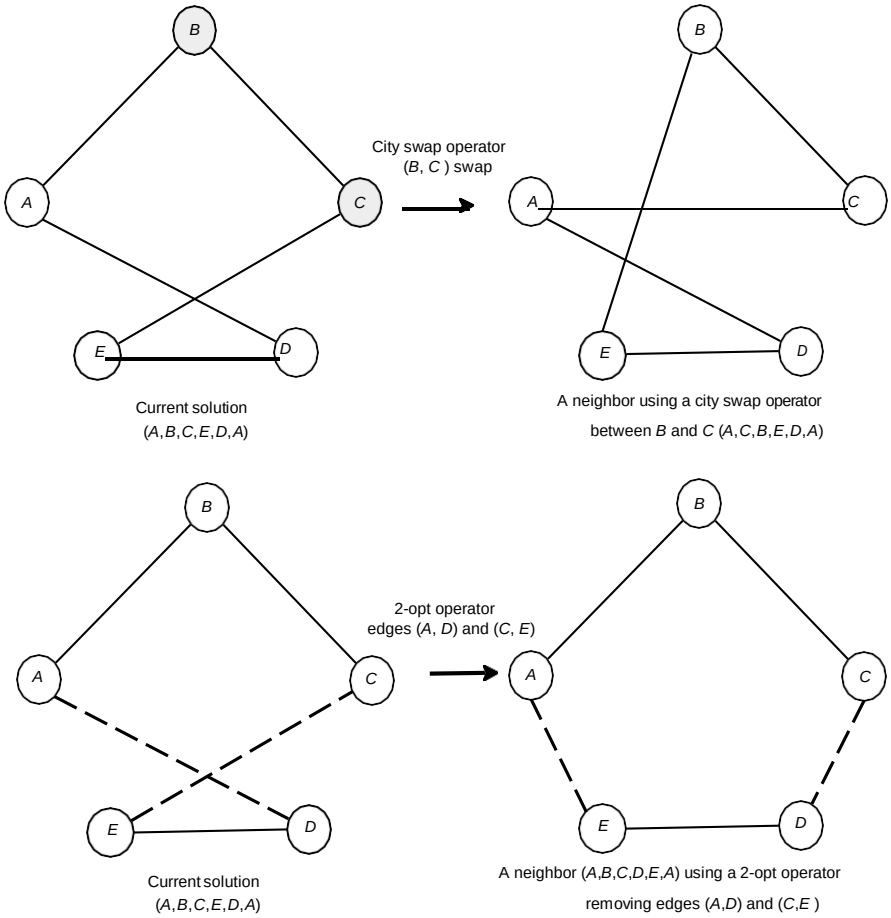


FIGURE 2.5 City swap operator and 2-opt operator for the TSP.

As mentioned, the efficiency of a neighborhood is related not only to the representation but also to the type of problems to solve. For instance, in scheduling problems, permutations represent a priority queue. Then, the relative order in the sequence is very important, whereas in the TSP it is the adjacency of the elements that is important. For scheduling problems, the 2-opt operator will generate a very large variation (weak locality), whereas for routing problems such as the TSP, it is a very efficient operator because the variation is much smaller (strong locality).

Example 2.2 Neighborhoods for permutation scheduling problems. For permutations representing sequencing and scheduling problems, the *k*-opt family of operators is not well suited. The following operators may be used:

- **Position-based neighborhood:** Figure 2.7 illustrates an example of a position-based operator, the *insertion operator*. In the insertion operator, an element at one

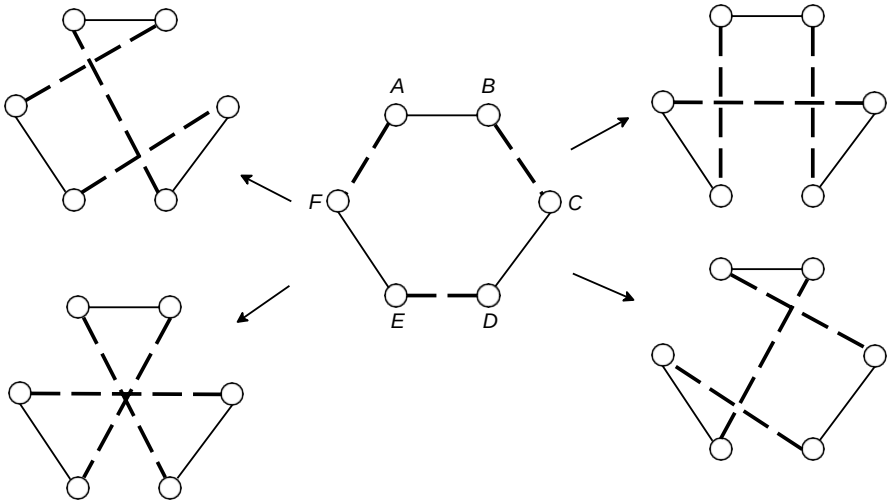


FIGURE 2.6 3-opt operator for the TSP. The neighbors of the solution (*A,B,C,D,E,F*) are (*A,B,F,E,C,D*), (*A,B,D,C,F,E*), (*A,B,E,F,C,D*), and (*A,B,E,F,D,C*).

position is removed and put at another position. The two positions are randomly selected.

- **Order-based neighborhood:** Many order-based operators can be used such as the exchange operator where arbitrarily selected two elements are swapped as shown in Fig. 2.8, and the inversion operator where two elements are randomly selected and the sequence of elements between these two elements are inverted as shown in Fig. 2.9.

Those operators are largely used in scheduling problems and seldom for routing problems such as the TSP for efficiency reasons.

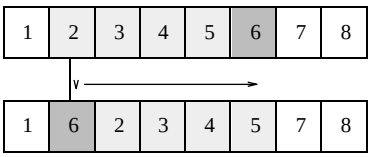


FIGURE 2.7 Insertion operator.

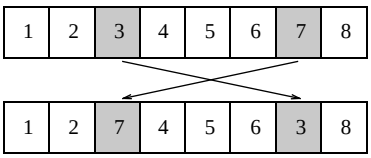


FIGURE 2.8 Exchange operator.

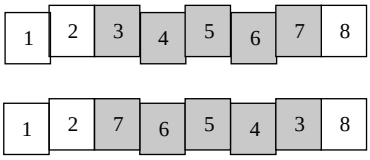


FIGURE 2.9 Inversion operator.

2.1.2 Very Large Neighborhoods

In designing a S-metaheuristic, there is often a compromise between the size (or diameter) and the quality of the neighborhood to use and the computational complexity to explore it. Designing large neighborhoods may improve the quality of the obtained solutions since more neighbors are considered at each iteration (Fig. 2.10). However, this requires an additional computational time to generate and evaluate a large neighborhood.

The size of the neighborhood for a solution s is the number of neighboring solutions of s. Most of the metaheuristics use small neighborhoods that is, in general, a polynomial function of the input instance size (e.g., linear or quadratic function). Some large neighborhoods may be high-order polynomial or exponential function of the size of the input instance. The neighborhood is *exponential* if its size grows exponentially with the size of the problem. Then, the complexity of the search will be much higher. So, the main issue here is to design efficient procedures to explore large neighborhoods. These efficient procedures identify improving neighbors or the best neighbor without the enumeration of the whole neighborhood (Fig. 2.11).

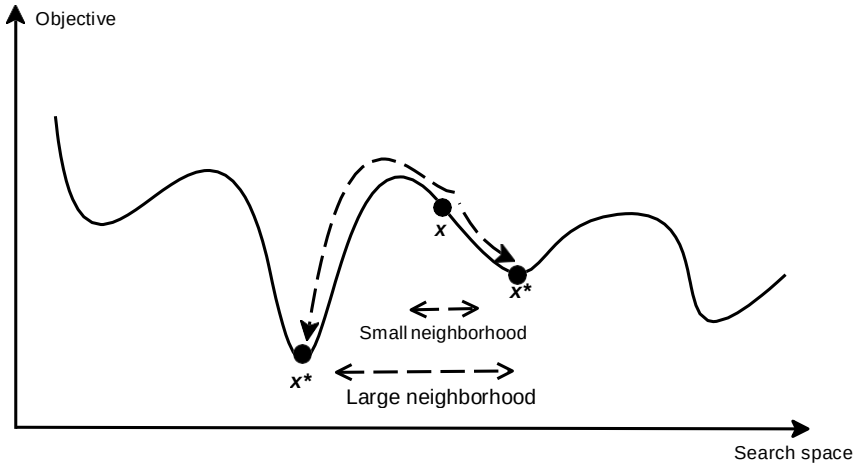


FIGURE 2.10 Impact of the size of the neighborhood in local search. Large neighborhoods improving the quality of the search with an expense of a higher computational time.

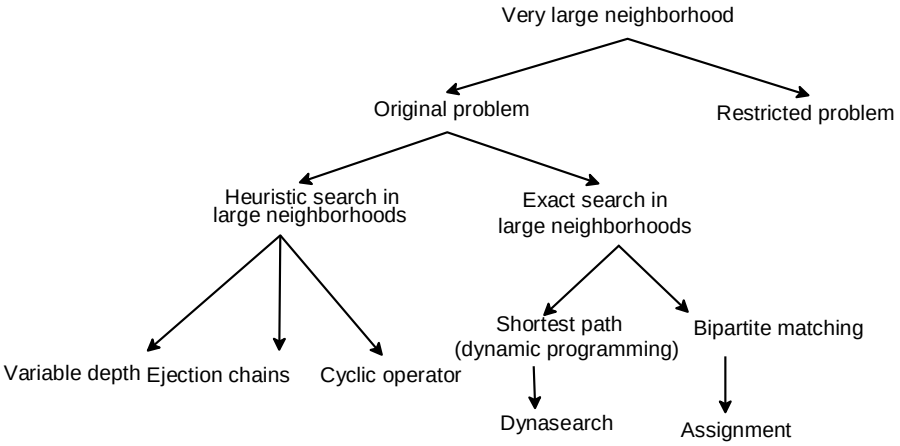


FIGURE 2.11 Very large neighborhood strategies.

A comprehensive survey of very large search neighborhoods may be found in Ref. [14].

2.1.2.1 Heuristic Search in Large Neighborhoods This approach consists in searching heuristically very large neighborhoods. A partial set of the large neighborhood is generated. Hence, finding the best neighbor (i.e., local optimum) is not guaranteed.

Let us consider a given neighborhood N defined by neighbors of distance $k = 1$ from a solution $s(N_1(s) = \{s' \in S/d(s, s') = 1\})$. In a similar way, a larger neighborhood $N_k(s)$ of distance k $(d = 2, 3, \ldots, n)$[4] is defined as the set

$$N_k(s) = N_{k-1}(s) \cup \{s''/\exists s' \in N_k(s) : s'' \in N_1(s')\}$$

Larger is the distance k, larger the neighborhood is. Since $N_n(s)$ is the whole search space, finding the best solution in this neighborhood is NP-hard if the original problem is NP-hard. Variable-depth search methods represent those strategies that explore partially the $N_k(s)$ neighborhoods. In general, a variable distance move (k-distance or k-exchange) with a distance of 2 or 3 may be appropriate.

This idea of variable-depth search methods has been first used in the classical Lin–Kernighan heuristic for the TSP [508] and the ejection chains. An *ejection chain* is a sequence of coordinated moves. Ejection chains were proposed first by Glover to solve the TSP problem [324]. They are based on alternating path methods, that is, alternating sequence of addition and deletion moves.

[4] n is related to the size of the problem (i.e., maximum distance between any two solutions).

Example 2.3 Ejection chains for vehicle routing problems. This example illustrates the application of ejection chains to the capacitated vehicle routing problem [645]. An ejection chain is defined as a sequence of coordinated moves of customers from one route to a successive one. Each ejection chain involves k levels (routes) starting at route 1 and ending at route k (Fig. 2.12). It ejects one node from route 1 to the successive route 2, one node from the route 2 to the route 3, and so on and one node from level $k-1$ to k (Fig. 2.12). The last node is bumped from route k to the route 1. The successive moves represent an ejection chain if and only if no vertex appears more than once in the solution. Figure 2.12 illustrates the ejection chain moves.

The heuristically searched large neighborhoods (e.g., ejection chain strategy, variable depth) have been applied successfully to various combinatorial optimization problems: clustering [220], vehicle routing [645], generalized assignment [832], graph partitioning [230], scheduling [712], and so on.

Many neighborhoods are defined by cycles. Cycle operators may be viewed as the generalization of the well-known 2-opt operator for the TSP problem, where more than two elements of the solution are involved to generate a neighbor.

Example 2.4 Cyclic exchange. A very large neighborhood based on cyclic exchange may be introduced into partitioning problems [15]. Let $E = \{e_1, e_2, \ldots, e_n\}$ be a set of elements to cluster into q subsets. A q-partition may be represented by the clustering $S = \{S_1, S_2, \ldots, S_q\}$. A 2-neighbor of the solution $S = \{S_1, S_2, \ldots, S_q\}$ is obtained by swapping two elements that belong to two different subsets. In its general form, the cyclic operator is based on swapping more than two elements and then involves more than two subsets (Fig. 2.13). A move involving k subsets S_i is represented by a cyclic permutation π of size k ($k \leq q$), where $\pi(i) = j$ represents the fact that the element e_i of the subset S_i is moved to the subset S_j. Hence, a solution $Q = \{Q_1, Q_2, \ldots, Q_q\}$ is a cyclic neighbor of the solution $S = \{S_1, S_2, \ldots, S_q\}$ if there exists a sequence $L = (o_1, \ldots, o_m)$ of size $m \leq k$ of moving single elements that generates Q from S. Let us notice that the first and the last move must concern the same partition (Fig. 2.13). The size of this neighborhood is $O(n^k)$. Using the cyclic operator, finding the best neighbor in the very large neighborhood has been reduced to the subset disjoint minimum cost cycle problem [769].

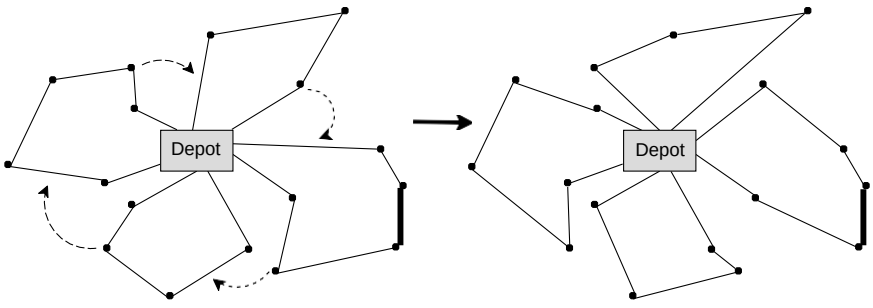


FIGURE 2.12 A four-level ejection chain for vehicle routing problems. Here, the ejection chain is based on a multinode insertion process.

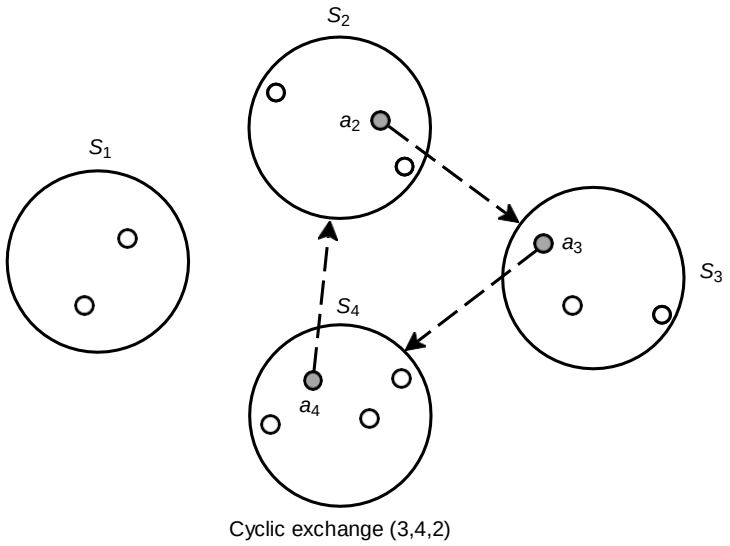


FIGURE 2.13 Very large neighborhood for partitioning problems: the cyclic exchange operator. Node a_2 is moved from subset S_2 to subset S_3, node a_3 is moved from subset S_3 to subset S_4, and node a_4 is moved from subset S_4 to subset S_2.

Indeed, finding an improving neighbor is based on the construction of the *improvement graph* [769]. Let $E = \{e_1, e_2, \ldots, e_n\}$ be the set of elements associated with the partitioning problem and let $x[i]$ be the subset containing the element e_i. The improvement graph $G = (V, E)$ is constructed as follows. The set $V = \{1, 2, \ldots, n\}$ of nodes is defined by the set of elements ($i = 1, \ldots, n$) corresponding to the indices of the elements belonging to E. An edge $(i, j) \in E$ is associated with each move operation that transfers the element i from $x[i]$ to $x[j]$ and removes the element j from $x[j]$. Each edge (i, j) is weighted by a constant c_{ij}, which represents the cost increase of $x[j]$ when the element i is added to the set $x[i]$ and j is removed:

$$c_{ij} = d[\{i\} \cup x[j] - \{j\}] - d[x[j]]$$

Definition 2.5 Subset-disjoint cycle. *A cycle Z in a graph $G = (V, E)$ is subset disjoint if*

$$\forall i, j \in Z, x[i] \neq x[j]$$

that is, the nodes of Z are all in different subsets.

It is straightforward to see that there is a correspondence between cyclic exchange for the partitioning problem and subset-disjoint cycles in the improvement graph, that is, for every negative cost cyclic exchange, there is a negative cost subset-disjoint cycle in the improvement graph. The decision problem of whether there is a subset-disjoint cycle in the improvement graph is an NP-complete problem. Hence, the problem of finding a

negative cost subset-disjoint cycle is NP-hard [769]. So, heuristics are generally used to explore this very large neighborhood [769].

The cyclic operator has been mainly used in grouping problems such as graph partitioning problems, vehicle routing problems [248,307], scheduling problems [283], constrained spanning tree problems [16], or assignment problems [759].

2.1.2.2 Exact Search in Large Neighborhoods This approach consists in searching exactly very large neighborhoods. The main goal is to find an improving neighbor. They are based on efficient procedures that search specific large (even exponential) neighborhoods in a polynomial time.

Searching the best or an improving neighbor for some large neighborhoods may be modeled as an optimization problem. This class of method is mainly based on network flow-based improvement algorithms. Two types of efficient search algorithms solving this optimization problem may be found in the literature [14]:

- **Path finding:** Where shortest path and dynamic programming algorithms are used to search the large neighborhood and identify improving neighbors.
- **Matching:** Where well-known polynomial-time matching algorithms are used to explore large neighborhoods for specific problems.

For instance, swap-based neighborhoods in permutations can be generalized by the use of multiple compounded swaps (e.g., dynasearch).

Definition 2.6 Independent swaps. *Given a permutation* $\pi = \{\pi_1, \pi_2, \ldots, \pi_n\}$, *a swap move* (i,j) *consists in exchanging the two elements* π_i *and* π_j *of the permutation* π. *Two swap moves* (i, j) *and* (k, l) *are independent if* $(max\{i, j\} < min\{k, l\})$ *or* $(min\{i, j\} > max\{k, l\})$.

A swap-based large neighborhood of a permutation can be defined by the union of an arbitrary number of independent swap moves.

Example 2.5 Dynasearch. Dynasearch is another classical large neighborhood approach that has been introduced for optimization problems where solutions are encoded by permutations (e.g., sequencing problems) [150]. The dynasearch two-exchange move is based on improving a Hamiltonian path between two elements $\pi(1)$ and $\pi(n)$ (see Fig. 2.14). The operator deletes the edges $(\pi(i), \pi(i + 1))$, $(\pi(j), \pi(j + 1))$, $(\pi(k), \pi(k + 1))$, and $(\pi(l), \pi(l + 1))$. The following conditions must

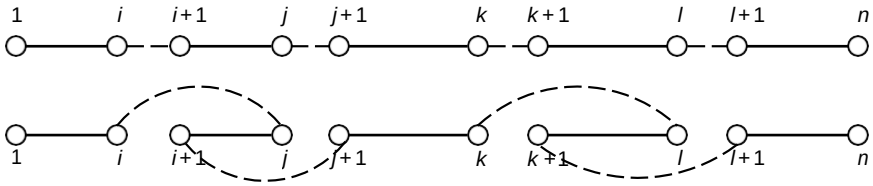


FIGURE 2.14 Dynasearch using two independent two-exchange moves: polynomial exploration of exponentially large neighborhoods.

hold: $1 < i + 1 < \boldsymbol{j} \leq n$, $\pi(\boldsymbol{j} + 1) \neq \pi(i)$, and $1 < k + 1 < l \leq n$, $\pi(l + 1) \neq \pi(k)$. The segments are independent if $j \leq k$ or $l < i$. Indeed, when this condition holds, the segments that are exchanged by the two moves do not share any edge.

The size of this exponential neighborhood is $O(2^{n-1})$. The problem of finding the best neighbor is reduced to the shortest path problem in the improvement graph. Then, it is explored in an efficient polynomial way by a dynamic programming algorithm in $O(n^2)$ for the TSP and in $O(n^3)$ for scheduling problems [94,396]. If the compounded swaps are limited to adjacent pairs, the time complexity of the algorithm is reduced to $O(n^2)$ in solving scheduling problems.

Another exponential neighborhood is defined by the *assignment neighborhood* [345]. It is based on the multiple application of the insertion operator. Let us consider the TSP problem. Giving a tour $S = (1, 2, \ldots, n, 1)$, where $d[i, j]$ represents the distance matrix. The assignment neighborhood consists in constructing the following bipartite improvement graph: select and eject k nodes $V = \{v_1, v_2, \ldots, v_k\}$ from the tour S, where $k = \lfloor n/2 \rfloor$. The set $U = \{u_1, u_2, \ldots, u_{n-k}\}$ represents the nonejected nodes of the tour S; that is, $U \cup V = S$. Then, a subtour $S' = (u_1, u_2, \ldots, u_{n-k}, u_1)$ is created. Finally, a complete bipartite graph $G = (N, V, E)$ is constructed, where $N = \{q_i : i = 1, \ldots, n - k\}$, q_i represents the edge (u_i, u_{i+1}) for $i = 1$ to $n - k - 1$,

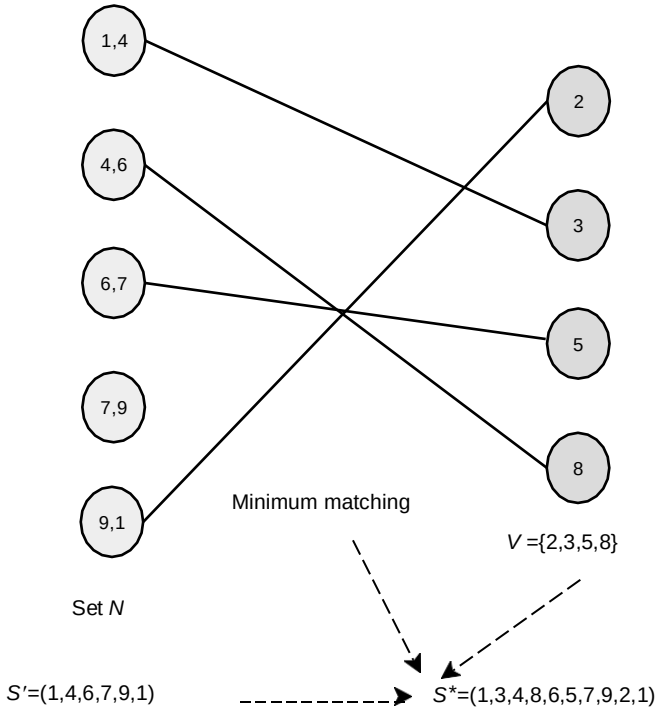


FIGURE 2.15 Assignment neighborhood: constructing the bipartite improvement graph and finding the optimal matching.

and $q_{n-k} = (u_{n-k}, u_1)$ (Fig. 2.15). The set of edges $E = (q_i, v_j)$ is weighted by $c[q_i, v_j] = d[u_i, v_j] + d[v_j, u_{i+1}] - d[u_i, u_{i+1}]$.

A neighbor S^* for the tour S is defined by inserting the nodes of V in the subtour S'. No more than one node is inserted between two adjacent nodes of the subtour S'. The problem of finding the best neighbor is then reduced to the minimum cost matching on a bipartite improvement graph. Figure 2.15 illustrates the construction of the improvement graph on the tour $S = (1, 2, \ldots, 9, 1)$ composed of nine nodes, where $V = \{2, 3, 5, 8\}$ and $U = \{1, 4, 6, 7, 9\}$. Hence, the subtour S' is defined as (1, 4, 6, 7, 9, 1). For simplicity reasons, only the edges of the minimum cost matching are shown for the bipartite graph G. According to the optimal matching, the obtained tour is (1, 3, 4, 8, 6, 5, 7, 9, 2, 1).

The generalization of the assignment operator for arbitrary k and n, and different inserting procedures (e.g., edges instead of nodes) in which the problem is reduced to a minimum weight matching problem is proposed [344].

Matching-based neighborhoods have been applied to many combinatorial optimization problems such as the TSP [347], quadratic assignment problem [203], and vehicle routing problems [225].

2.1.2.3 Polynomial-Specific Neighborhoods Some NP-hard optimization problems may be solved in an efficient manner by restricting the input class instances or by adding/deleting constraints to the target optimization problem.

For instance, many graph problems (e.g., Steiner tree, TSP) are polynomial for *specific* instances of the original NP-hard problem (e.g., series–parallel, outerplanar, Halin). This fact may be exploited in defining large neighborhoods based on those special cases solvable in polynomial time. Indeed, this strategy consists in transforming the input instance into a specific instance that can be solved in a polynomial manner. This class of strategies are not yet well explored in the literature [14].

Example 2.6 Halin graphs—a polynomial solvable instance class. Halin graphs deal with properties of minimal connectivity in graphs. They are generalizations of

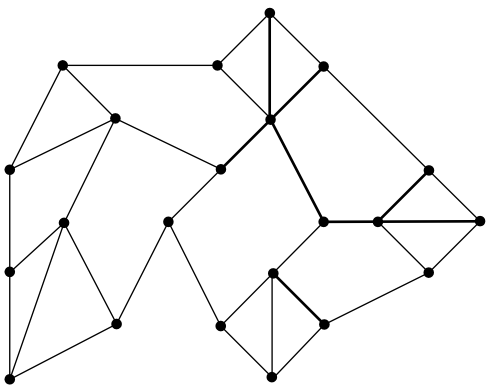


FIGURE 2.16 An example of a Halin graph [734].

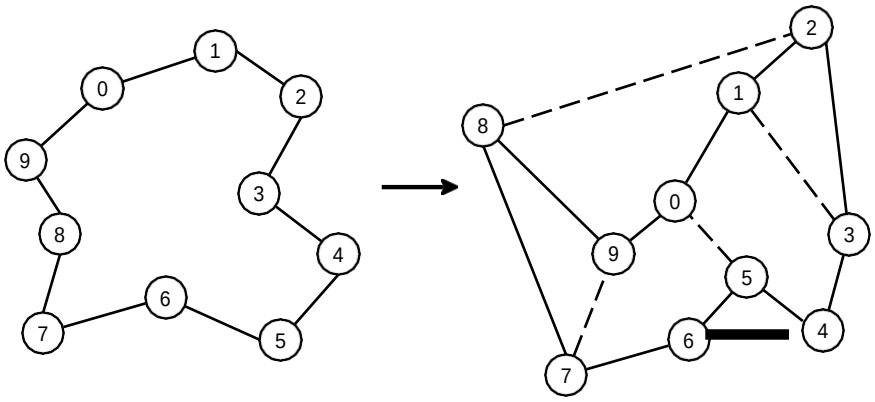


FIGURE 2.17 Extending a given input instance to a Halin graph.

tree and ring graphs. A graph is a Halin graph if it is formed by embedding a tree having no degree-2 vertices in the plane and connecting its leaves by a cycle that crosses none of its edges (Fig. 2.16). For this class of instances, a polynomial-time algorithm is known to solve the traveling salesman problem in $O(n)$ [157] and the Steiner tree problem [822]. Hence, Halin graphs may be used to construct large neighborhoods for the TSP as follows [14]: let π be the current tour. The solution π_H is a Halin extension of π if π_H is a Halin graph and π is a subgraph of π_H (Fig. 2.17). Suppose *HalinConstruct*(π) is an efficient function that returns a Halin extension of the solution π. Then, the neighborhood $N(\pi)$ is constructed as $\{\pi' : \pi'$ is a tour in HalinConstruct(π)$\}$. Hence, finding the best tour in $N(\pi)$ is reduced to find the best tour in the solution *HalinConstruct*(π).

2.1.3 Initial Solution

Two main strategies are used to generate the initial solution: a *random* and a *greedy* approach. There is always a trade-off between the use of random and greedy initial solutions in terms of the quality of solutions and the computational time. The best answer to this trade-off will depend mainly on the efficiency and effectiveness of the random and greedy algorithms at hand, and the S-metaheuristic properties. For instance, the larger is the neighborhood, the less is the sensitivity of the initial solution to the performance of the S-metaheuristics.

Generating a random initial solution is a quick operation, but the metaheuristic may take much larger number of iterations to converge. To speed up the search, a greedy heuristic may be used. Indeed, in most of the cases, greedy algorithms have a reduced polynomial-time complexity. Using greedy heuristics often leads to better quality local optima. Hence, the S-metaheuristic will require, in general, less iterations to converge toward a local optimum. Some approximation greedy algorithms may also be used to obtain a bound guarantee for the final solution. However, it does not mean that using better solutions as initial solutions will always lead to better local optima.

Example 2.7 Greedy initial solutions are not always better. The Clark–Wright greedy heuristic is a well-known efficient heuristic for the TSP and other routing problems, such as the vehicle routing problem. The solutions obtained by this greedy heuristic are much better in quality than random solutions. However, using a local search algorithm based on the 3-opt move operator, experiments show that using the initial solutions generated by this greedy heuristic leads to worse solutions than using random initial solutions in solving the TSP problem [421]. Moreover, the Clark–Wright heuristic may be also time consuming in solving large instances of the TSP problem. For this example, the random strategy dominates the greedy one in both quality and search time!

The random strategy may generate a high deviation in terms of the obtained solutions. To improve the robustness, a hybrid strategy may be used. It consists in combining both approaches: random and greedy. For instance, a pool of initial solutions can be composed of greedy solutions and randomly generated solutions. Moreover, different greedy strategies may be used in the initialization process. In fact, for some optimization problems, many greedy constructive algorithms exist and are easy to design and implement.

Example 2.8 A hybrid approach for the TSP. Given an instance with n cities, the following three algorithms can be used to generate a pool of initial solutions:

- An algorithm that generates a random permutation with uniform probability.
- A nearest-neighbor greedy algorithm that chooses a starting city i randomly. Then, among unsequenced cities, it chooses a city j that minimizes the length between i and j and iterates the same process on the city j until a complete tour is constructed. Using different starting cities allows to generate different final solutions. Hence, n different solutions may be obtained with this greedy procedure.
- Another greedy procedure, the cheapest insertion algorithm, first finds the minimum length path (i, j). Let C be the tour (i, j, i). Then, it chooses an unsequenced city that minimizes the length of the tour C and inserts it into C in the best possible way. This process is iterated until the tour C is composed of all the cities.

In some constrained optimization problems, it may be difficult to generate random solutions that are feasible. In this case, greedy algorithms are an alternative to generate feasible initial solutions.

For some specific real-life operational problems, the initial solution may be initialized (partially or completely) according to expertise or defined as the already implemented solution.

2.1.4 Incremental Evaluation of the Neighborhood

Often, the evaluation of the objective function is the most expensive part of a local search algorithm and more generally for any metaheuristic. A naive exploration of

the neighborhood of a solution s is a *complete* evaluation of the objective function for every candidate neighbor s' of $N(s)$.

A more efficient way to evaluate the set of candidates is the *evaluation* $O(s, m)$ of the objective function when it is possible to compute, where s is the current solution and m is the applied move. This is an important issue in terms of efficiency and must be taken into account in the design of an S-metaheuristic. It consists in evaluating only the transformation $O(s, m)$ applied to a solution s rather than the complete evaluation of the neighbor solution $f(s') = f(s \oplus m)$. The definition of such an incremental evaluation and its complexity depends on the neighborhood used over the target optimization problem. It is a straightforward task for some problems and neighborhoods but may be very difficult for other problems and/or neighborhood structures.

Example 2.9 Incremental evaluation of the objective function. First, let us present an incremental evaluation for the 2-opt operator applied to the TSP. The incremental evaluation can be stated as follows:

$$Of = c(\pi_i, \pi_j) + c(\pi_{i+1}, \pi_{j+1}) - c(\pi_i, \pi_{i+1}) - c(\pi_j, \pi_{j+1})$$

Let us consider the clique partitioning problem defined in Example 2.22. For any class C of the current solution s, $W(i, \emptyset) = 0$ and for $C \neq \emptyset$, $W(i, C) = \sum_{j \in C, j \neq i} w(i, j)$. The incremental evaluation of the objective function f in moving a node i from its class C_i to another one C will be $W(i, C) - W(i, C_i)$. Then, for an improving neighbor, we have $W(i, C) < W(i, C_i)$. The complexity of finding the best class C^* for a node i (i.e., $W(i, C^*)$ is minimum over all classes $C \neq C_i$) is $O(n)$.

The incremental evaluation function may also be approximated instead of having an exact value (e.g., *surrogate functions* [777]). This will reduce the computational complexity of the evaluation, but a less accuracy is achieved. This approach may be beneficial for very expensive incremental functions.

2.2 FITNESS LANDSCAPE ANALYSIS

The main point of interest in the domain of optimization must not be the design of the best algorithm for *all* optimization problems but the search for the most adapted algorithm to a given class of problems and/or instances. The question of whether a given optimization algorithm A is absolutely better than an optimization algorithm B is senseless. Indeed, the NFL theorem proves that, under certain assumptions, no optimization algorithm is superior to any other on *all* possible optimization problems; that is, if algorithm A performs better than B for a given problem, there is always another problem where B performs better than A [823]. No metaheuristic can be uniformly better than any other metaheuristic. The question of superiority of a given algorithm has a sense only in solving a given class of problems and/or instances. Many studies on the analysis of landscapes of different optimization problems have shown that not only different problems correspond to different structures but also

different instances of the same problem correspond to different structures. In fact, no ideal metaheuristic, designed as a black box, may exist.

Example 2.10 Superiority of algorithms and problem classes. A comparison of two optimization methods may be carried out for a given class of problems and/or instances. For instance, there are problems where an evolutionary algorithm does not perform well, namely, on a simple unimodal function and on a random function. In a unimodal continuous function, local search algorithms such as quasi-Newton methods are much faster and efficient. In a random function or needle-in-a-haystack-type landscape (flat everywhere except at a single point), there is no structure that could be learned.

The analysis of landscapes of optimization problems[5] is an important aspect in designing a metaheuristic. The effectiveness of metaheuristics depends on the properties of the landscape (roughness, convexity, etc.) associated with the instance to solve. The study of the landscapes of an optimization problem provides a way of investigating the intrinsic natures and difficulties of its instances to integrate this knowledge about the problem structure in the design of a metaheuristic. The representation, neighborhood, and the objective function define the landscape in a complete manner and then influence on the search efficiency of a metaheuristic. Landscape analysis is performed in the hope to predict the behavior of different search components of a metaheuristic: representations, search operators, and objective function. Thus, analyzing the fitness landscape will help to design better representations, neighborhoods, and objective functions. Moreover, the landscape analysis affords a better understanding of the behavior of metaheuristics or any search component of a metaheuristic. One may explain why a given metaheuristic works or fails on a given problem and/or instance.

The notion of landscape has been first described by Wright in 1932 in biology [827]. In biology, the term fitness is derived from the term "survival of the fittest" in natural selection. The fitness corresponds to the relative ability of an organism to survive, to mate successfully, and to reproduce resulting in a new organism [451]. The fitness landscape has been used to study the dynamics of biological systems (species evolution) and understand evolutionary processes that are driven by specific operators such as mutation and crossover. Afterward, this concept has been used to analyze optimization problems [717,718,428].

Definition 2.7 Search space. *The search space*[6] *is defined by a directed graph* $G = (S, E)$*, where the set of vertices* S *corresponds to the solutions of the problem that are defined by the representation (encoding) used to solve the problem, and the set of edges* E *corresponds to the move operators used to generate new solutions (neighborhood in S-metaheuristics).*

[5]The fitness landscape analysis in not specific to S-metaheuristics. The same concepts may be reused in P-metaheuristics.

[6]Also named the configuration graph or state space.

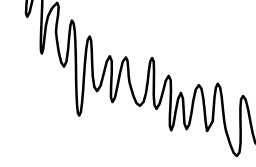
There is an edge between solutions s_i and s_j if the solution s_j can be generated from the solution s_i using a move operator; that is, s_i and s_j are neighbors. Let us notice that using different neighboring relations generates different search spaces and landscapes and then produces, for instance, different numbers of local optima.

Definition 2.8 Fitness landscape. *The fitness landscape l may be defined by the tuple* (G, f)*, where the graph G represents the search space and* f *represents the objective function that guides the search.*

A convenient way to describe a landscape consists in using some geographical terms. Then, considering the search space as the ground floor, we elevate each solution to an altitude equal to its quality. We obtain a landscape made of valleys, plains, peaks, canyon, cliffs, plateaus, basins, and so on (see Table 2.1). A S-metaheuristic may be seen as a trajectory or a walk in the landscape to find the lowest peak. Two successive solutions are always neighbors. The geographical metaphor provides a useful tool to represent the landscapes in one, two, or three dimensions. The problem lies in the difficulty to have a realistic view of the landscape for high-dimensional problems.

Example 2.11 NK-model for landscapes. The NK-model for landscapes, proposed by Kaufmann [452], defines a class of fitness landscapes that can be configured using two parameters N and K. The parameter N represents the dimension of the problem and the parameter K the degree of epistatic interactions between the decision variables of the fitness function. Each solution of the search space is supposed to be

TABLE 2.1 Some Representations of Landscapes in One Dimension Using the Geographical Metaphor

Flat, plain	
Basin, valley	
Rugged plain	
Rugged valley	

a binary vector of size N. Then, the search space may be viewed as an N-dimensional hypercube. The fitness function associated with a solution $x = (x_1, x_2, \ldots, x_N)$ is defined as follows:

$$f(x) = \frac{1}{N} \sum_{i=1}^{N} f_i(x_i, x_{i1}, x_{i2}, \ldots, x_{ik})$$

where the function f_i of the decision variable i depends on its value x_i and the values of k on other decision variables $x_{i1}, x_{i2}, \ldots, x_{ik}$. The decision variables $x_{i1}, x_{i2}, \ldots, x_{ik}$ are selected randomly from $[1, \ldots, N]$. For the 2^{K+1} different inputs, the function f_i assigns a uniformly distributed random number between 0 and 1.

For $K = 0$, the landscape has only one local optimal solution (i.e., global optimal), while for $K = N - 1$, the expected number of local optima is $(2^N/N + 1)$.

Furthermore, to be able to analyze the fitness landscape an important property has to be verified: connexity of the search space; that is, the landscape has to be *connected*. In other words, there exists a path in the search space graph G between any pair of vertices.

Example 2.12 Connexity of the search space. One of the most important properties of the search space G is its connexity. For any two solutions s_i and s_j (nodes of G), there should be a path from s_i to s_j. Hence, for any initial solution s_i there will be a path from s_i to the global optimal solution s^*.

Let us consider the quadratic assignment problem defined in Exercise 2.9. A solution is encoded by a permutation, and the neighborhood is based on the swapping between two elements. The search space (graph G) associated with this representation and neighborhood is connected. In fact, given any couple of permutations, there is always a path in G linking one permutation to another. Moreover, any swapping applied to any permutation of S always leads to an element of S. But, some search spaces do not offer those good properties.

Let us consider the graph coloring problem. For a given graph $H = (W, F)$, a solution is represented by a discrete vector of size $|W|$ that associates with each node of the graph its color. Each solution is a feasible coloring. The neighborhood $N(s)$ is defined as follows: the set of *feasible* neighboring solutions is obtained from the solution s by changing the color of one node. This neighborhood does not satisfy the connexity property for the induced search space. Figure 2.18 shows how the optimal coloring s^* cannot be reached from the initial solution s_1. Suppose that the maximal number of colors to be used is equal to 3. This example also illustrates that visiting infeasible solutions may help enhance the performance of a metaheuristic.

2.2.1 Distances in the Search Space

To analyze the landscape associated with a given problem, one needs to define a distance in the graph G representing the search space. In fact, the distance will define

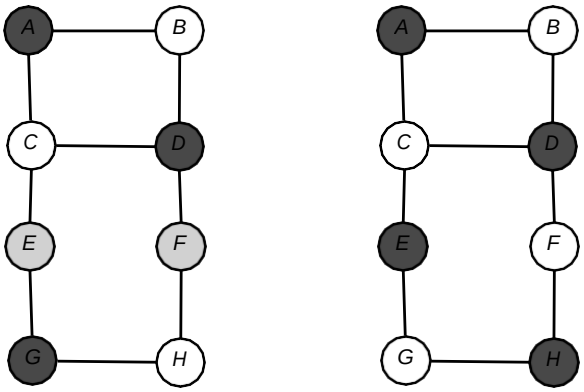


FIGURE 2.18 Connexity of the search space related to the graph coloring problem. The optimal solution cannot be reached from the given initial solution.

the spatial structure of the landscape. The concept of distance may also be useful in the design of some search mechanisms such as diversification and intensification procedures (e.g., path relinking, diversity measure).

Let $d(s_i, s_j)$ be the distance between solutions s_i and s_j in the search space. It is defined as the length of the shortest path in the graph G; that is, the minimum number of application of the move operator to obtain the solution s_j from the solution s_i. In topology, a distance must have the following properties:

Definition 2.9 *Let E be a set, a distance on E is any mapping: $d : E \times E \rightarrow \mathbb{R}^+$, such that*

2.2.1.1 $d(x, y) = 0$ *if and only if* $x = y$ *(separative property);*

2.2.1.2 $\forall x, y \in E \times E : d(x, y) = d(y, x)$ *(symmetrical property);*

2.2.1.3 $\forall x, y, z \in E \times E \times E : d(x, y) + d(y, z) \geq d(x, z)$ *(triangular property).*

Example 2.13 Distances in some usual search spaces. In search spaces using binary representations and the flip move operator, the Hamming distance may be used:

$$d_{\mathrm{H}}(x, y) = \sum_{i=1}^{n} x_i \oplus y_i$$

It represents the number of different bit values between the two solutions. For a problem of size n, the size of the search space is 2^n and its diameter is equal to n.

In search spaces using permutation representations and the exchange move operator, the cardinality of the space is $n!$ and the maximum distance between two permutations is $n - 1$.

As the distance is associated with the landscape of the problem, it must be *coherent*, that is, related to the search operators (neighborhood) used to solve the problem.

Example 2.14 Coherent distance. For the TSP problem, a merely intuitive distance $\delta(t_1, t_2)$ between two solutions t_1 and t_2 is related to the number of edges shared by both tours t_1 and t_2 [465]. Obviously, we have $0 \leq \delta(t_1, t_2) \leq n$, where n is the number of cities. Let us consider a S-metaheuristic that uses the 2-opt neighborhood. The main drawback of using the distance δ is that it is not related to the move operator 2-opt used in the optimization algorithm. For instance, if the 2-opt move is applied twice on a given solution, the generated solution may have 0, 3, or 4 edges not contained in the initial solution. Then, the coherent distance to use is $O_{2-opt}(t_1, t_2)$ that denotes the minimum number of applications of the 2-opt move to obtain t_2 from t_1 [84]. However, there exists no polynomial algorithm to compute such a distance. This is why the distance dealing with the edges is always used in practice.

In some optimization problems and/or neighborhoods, computing a distance between solutions may be a complex task in terms of time complexity. Hence, for some problems computing the distance is an NP-complete problem or high-order polynomial problem.

2.2.2 Landscape Properties

Many indicators may be used to identify the fitness landscape properties: number of local optima, distribution of the local optima, ruggedness, structures of basins of attraction, presence of plateaus (neutral networks), distribution of the fitness (mean, variance), and so on. Most of them are based on statistical studies. Two types of indicators may be distinguished:

2.2.2.1 **Global indicators:** Global approaches provide informations about the struc- ture of the entire landscape [428,525,812]. They determine the structure of the landscape using the whole search space or its approximation. However, in optimization, the goal is to track "good" solutions. These good solutions repre- sent a tiny part of the search space. Hence, the global approach is not sufficient to characterize a fitness landscape to be explored by a metaheuristic. Metaheuristics focus on good solutions that are handled as ordinary points by the global approach.

2.2.2.2 **Local indicators:** The local approach consists in characterizing a local view of the landscape as explored by a metaheuristic. Indeed, metaheuristics have a

local view of the landscape, following some trajectories in the landscape and focusing on good solutions. Then it would be interesting to describe those good solutions and the regions where they are localized. The global approach does not answer those questions in the sense that it gives an average information on a large number of solutions that are not concerned with metaheuristics.

The two approaches are complementary. For instance, the local view of a landscape may be obtained by its exploration using a simple local search algorithm (steepest descent). During the search, the metaheuristic algorithm is used as a probe to extract some indicators that will characterize the landscape.

Example 2.15 Number of local optima. For some landscapes, the number of local optima may be computed theoretically or by enumeration. Let us consider the landscape of the number partitioning problem (see Example 1.15) where the neighborhood consists in moving a number a_i belonging to a partition S_1 to the other partition S_2. The number of local optima in this landscape is [260]

$$\frac{\sqrt{24}}{\pi}\frac{2^n}{n^{\frac{3}{2}}} \approx 2.764\frac{2^n}{n^{\frac{3}{2}}}$$

Notice that the size of the search space is 2^n. Thus, the number of local optima solutions is very large compared to the size of the search space. Indeed, the ratio of the local optima to the size of the search space is in the order of $O(n^{-1.5})$. Another representation or neighborhood must be designed for this problem to reduce the number of local optima.

In general, two different statistical measures are applied: *distribution measures* that study the topology of local optima solutions and the *correlation measures* that analyze the rugosity of the landscape and the correlation between the quality of solutions and their relative distance. In the rest of this section, the most important and commonly used indicators are outlined for each class.

2.2.2.1 Distribution Measures The objective of the distribution measures is the distribution analysis of the local optimal solutions in the landscape projected both in the search space G and in the objective space $\boldsymbol{f}$. The commonly used measures are

- **Distribution in the search space:** For a population P, let us define the average distance $\mathrm{dmm}(P)$ and the normalized average distance $\mathrm{Dmm}(P)$:

$$\mathrm{dmm}(P) = \frac{\overleftarrow{\sum_{s\in P}}\ \overleftarrow{\sum_{t\in P, t\neq s}} \mathrm{dist}(s,t)}{|P|\cdot(|P|-1)} \qquad \mathrm{Dmm}(P) = \frac{\mathrm{dmm}(P)}{\mathrm{diam}(S)}$$

The diameter of a population P of S is the maximal distance between elements of P :

$$\text{diam}(P) = \max_{s,t \in P} \text{dist}(s, t)$$

The normalized average distance characterizes the concentration of the population P in the search space. Notably, a weak distance indicates that the solutions belonging to the population P are clustered in a small region of the search space.

The indicator $O_{\text{Dmm}} = (\text{Dmm}(U) - \text{Dmm}(O))/\text{Dmm}(U)$ represents the variation of the average distance between a uniform population U and a population of local optimal solutions O. The cardinality of the set O is chosen according to the complexity of the local search algorithm (e.g., $10^3 \leq |O| \leq 10^4$).

- **Entropy in the search space:** The entropy concept in information theory is related to the notion of diversity in optimization. It allows to measure the diversity of a given population in the search space. According to a given optimization problem, different mathematical formulations of the entropy can be applied to measure the dispersion of the population elements over the search space. A weak entropy reveals a concentration of solutions, whereas a high entropy shows an important dispersion of the solutions in the search space.

Example 2.16 Entropy for the quadratic assignment problem. For the quadratic assignment problem, the entropy may be based on the mapping of the objects over the different locations [269]:

$$\text{ent}(P) = \frac{-1}{n \log n} \sum_{i=1}^{n} \sum_{j=1}^{n} \frac{n_{ij}}{\text{card } P} \log \frac{n_{ij}}{\text{card } P} \qquad 0 \leq \text{ent}(P) \leq 1$$

where n_{ij} is the number of times an object i is assigned to location j in the population P.

The indicator $O_{\text{ent}} = \text{ent}(U) - \text{ent}(O)/\text{ent}(U)$ represents the entropy variation between the starting population U and the final population of local optima O obtained after performing a local search algorithm on each solution of the population.

To evaluate the scatter of the solutions yielded by a metaheuristic (e.g., local search), a focus must be made on both their entropy and their distribution. The entropy gives an information about the scatter of the solutions but do not give information about the granularity of the concentrations. In fact, scattered concentrations or a single concentration may have the same low entropy. The mean distance gives a complementary information; it measures the concentration of the distribution (in a single region). In the analysis of the scatter of the local optima, the variations of the entropy and the mean distance between the initial population U and the final population O may be observed. Table 2.2 illustrates the fitness landscape structure according to the variation of the distance and the entropy.

TABLE 2.2 Use of the Entropy Variation and the Distance Variation to Analyze the Distribution of the Local Optima

Entropy O_{Ent}	Low variation	High variation	High variation
Distribution O_{Dmm}	Low variation	Low variation	High variation
Landscape			
	Uniform	Multimassif	One-Massif

A mathematical relationship between distribution-based measures and entropy-based measures shows that a low entropy is also a low distribution, but not vice versa [581]

- **Distribution in the objective space:** Many indicators can be used to analyze the distribution of solutions in the objective space. The *amplitude* Amp (P) of an arbitrary population P of solutions is the relative difference between the best quality of the population P and the worst one:

$$\text{Amp}(P) = \frac{|P| \cdot (\max_{s \in P} f(s) - \min_{s \in P} f(s))}{\sum_{s \in P} f(s)}$$

The relative variation of the amplitude O_{Amp} between a starting random population U and the final population O is given by

$$O_{\text{Amp}} = \frac{\text{Amp}(U) - \text{Amp}(O)}{\text{Amp}(U)}$$

The average Gap (O) of the relative gaps between the costs of the population O of local optima and the global optimal solution (or the best known solution) s^*

$$\text{Gap}(O) = \frac{\sum_{s \in O} (f(s) - f(s^*))}{|O| \cdot f(s^*)}$$

A reduced gap indicates that the problem is relatively "easy" to solve.

2.2.2.2 Correlation Measures The objective of correlation measures is to estimate the ruggedness of the landscape along with the correlation between the quality of solutions and their distance to a global optimal solution. The landscape is described as rough (unsmooth) if it contains many local optimal solutions and is characterized by a low correlation between neighboring solutions. Many correlation indicators may be used:

- **Length of the walks:** The average length Lmm (P) of the descent walks relative to the population P can be defined as

$$\text{Lmm}(P) = \frac{\sum_{p \in P} l(p)}{|P|}$$

in which $l(p)$ is the length of the walk starting with the solution $p \in P$. In an intuitive way, the length of the walk gives some information about the ruggedness of the landscape. In a rugged landscape, the optima are numerous and the walks are short, whereas in a smooth landscape the number of optima is smaller and the walks are longer.

- **Autocorrelation function:** The autocorrelation function measures the ruggedness of a landscape, and the autocorrelation function proposed by Weinberger may be used [812]. The *autocorrelation function* $\rho(d)$ measures the correlation of solutions in the search space with distance d. It is approximated by using a large sample of solution pairs (n solutions with distance d):

$$\rho(d) = \frac{\sum_{s,t \in S \times S, \text{dist}(s,t)=d} (f(s) - \bar{f})(f(t) - \bar{f})}{n.\sigma_f^2}$$

This function corresponds to the usual understanding of the landscape ruggedness. The autocorrelation function provides a measure of the influence of the distance d on the variation of the fitnesses of solutions. The measure belongs to the interval $[-1, 1]$. The closer is the absolute value to 1, the larger is the correlation between the solutions.

The autocorrelation function $\rho(1)$ considers only neighboring solutions ($d = 1$). A low value ($\rho(1) \approx 0$) indicates that the variation of fitness between two neighbors is equal on average to the variation between any two solutions, so the landscape is rugged. On the contrary, a high value ($\rho(1) \approx 1$) indicates that two neighbors have a similar fitness value when both solutions are distant, and hence the landscape is smoother.

Another approach to approximate the autocorrelation function consists in analyzing the degree of correlation between solutions of distance s of the landscape by performing a random walk $(f(x_t))$ of size m. The *random walk correlation function* is defined as [812]

$$r(s) \approx \frac{1}{\sigma_f^2 (m - s)} \sum_{t=1}^{m-s} (f(x_t) - \bar{f})(f(x_{t+s}) - \bar{f})$$

The idea is to generate a random walk in the search space via neighboring solutions. At each step, the fitness of the encountered solution is recorded. In this way, a *time series* of fitnesses $f_1, f_2, \ldots, f_n$ is generated.

Based on the autocorrelation function, the *correlation length* is the largest time lag (distance) for which the correlation between two solutions is statistically significant [719]. In other words, the correlation length measures the distance or

time lag m between two configurations, at which the values of one configuration can still provide some information about the expected value of the other. Thus, the larger the correlation length is, the flatter and smoother the landscape is. The search in a smooth landscape is more easy. The smaller is the correlation length, the more rugged is the associated landscape and harder is the search.

The correlation length can be defined as

$$l = \frac{1}{\ln(|r(1)|)} = -\frac{1}{\ln(|\rho(1)|)}$$

This value can be normalized in the interval [0, 1] by using the diameter of the landscape

$$\xi = \frac{l}{\text{diam}(G)}$$

Higher the correlation is, closer the normalized correlation length is to 1. There is no correlation if the normalized correlation ξ is close to 0. In a rough landscape, the correlation length of an iterative random walk is low, and the length of an iterated local search Lmm (U) is low compared to the diameter of the search space. Lmm (U) is the average length of iterated local searches performed from the initial population U. For an arbitrary population P of solutions

$$\text{Lmm}\,(P) = 100 \times \frac{\text{lmm}\,(P)}{n}$$

where lmm (P) is the average of the length of local searches starting from the solutions of the population P.

A random walk in the landscape can be viewed as a time series. Once the time series of the fitnesses is obtained, a model can be built using the *Box–Jenkins approach*, and thus we can make forecast about future values or simulate process as the one that generated the original data [89]. An important assumption must be made here: the landscape has to be statistically isotropic; that is, the time series of the fitnesses forms a stationary random process. This means that the random walk is "representative" of the entire landscape, and thus the correlation structure of the time series can be regarded as the correlation structure of the whole landscape. For instance, one can find an ARMA[7] model that adequately represents the data generating process [387]. A search walk following an AR(1) model is in the form

$$y_t = c + \alpha_1 y_{t-1} + \varrho_t$$

[7]The notation ARMA(p, q) refers to the model with p autoregressive terms and q moving average terms: $y_t = \varrho_t + \sum_{i=1}^{p} \alpha_i y_{t-i} + \sum_{i=1}^{q} \theta_i \varrho_{t-i}$.

This correlation structure given by the time series analysis implies that the fitness at a particular step in a random walk generated on this landscape totally depends on the fitness one step earlier. Knowing the fitness two steps earlier does not give any extra information on the expected value of fitness at the current step. Furthermore, the value of the parameter α_1 is the correlation coefficient between the fitness of two points one step apart in a random walk. Hence, for an AR(1) model, $r(d) = r(1)^d = e^{-d/l}$ where l is the correlation length. Table 2.3 shows some correlation lengths for different optimization problems. Those results have been obtained theoretically and confirmed experimentally [542,718].

The autocorrelation coefficient (or ruggedness) is a normalized measure of the autocorrelation function [29,30]:

$$\xi = \frac{1}{1 - \rho(1)}$$

This measure is based on the autocorrelation function of neighboring solutions $\rho(1)$. It evaluates globally the homogeneousness of the quality of the neighborhood. The larger the correlation, the flatter the landscape.

- **Fitness distance correlation:** Fitness distance correlation analysis measures how much the fitness of a solution correlates with the distance to the global optimum. The relationship between the fitness and the distance to a global optimum will have a strong effect on search difficulty [428,429]. To perform the fitness distance analysis, we need a sample of fitness values $F = \{f_1, f_2, \ldots, f_n\}$ and the corresponding set of distances to the global optimum $D = \{d_1, d_2, \ldots, d_n\}$. The correlation coefficient is given by

$$r = \frac{\text{cov}(F, D)}{\sigma_f \sigma_d}$$

TABLE 2.3 Correlation Lengths for Some Optimization Problems and Landscapes. TSP (Traveling Salesman Problem), QAP (Quadratic Assignment Problem), GBP (Graph Bipartitioning Problem), and NK-Model for Landscapes [542,718,719]

Problem	Move	Diameter	Neighborhood Size	Correlation Length
TSP	2-opt	$n - 1$	$\frac{n(n-1)}{2}$	$\frac{n}{2}$
	Swap	$n - 1$	$\frac{n(n-1)}{2}$	$\frac{n}{4}$
QAP	Swap	$n - 1$	$\frac{n(n-1)}{2}$	$\approx \frac{n}{3}$
GBP	Swap	$\frac{n}{2}$	$\frac{n(n-1)}{2}$	$\approx \frac{n}{8}$
	Flip	n	$n - 1$	$\approx \frac{n}{4}$
NK-model	Flip	n	$n - 1$	$\frac{n}{k+1}$

where

$$\mathrm{cov}(F,D) = \frac{1}{n}\sum_{i=1}^{n}(f_i - \bar{f})(d_i - \bar{d})$$

$\bar{f}$ and σ_f represent the covariance and the standard deviation, respectively. FDC is founded on the following conjectures:

- **Straightforward:** Large positive correlations indicate that the problem is relatively easy to solve since as the fitness decreases, the distance to the global optimum also decreases. The existence of a search path in the landscape with improving fitnesses toward the global optimum seems likely.
- **Misleading:** Large negative correlations indicate that the problem is "misleading" and the move operator will guide the search away from the global optimum. A solution can be very close to the global optimal solution and has a poor quality.
- **Difficult:** Near-zero correlations indicate[8] that there is no correlation between fitness and distance.

A better way to analyze the FDC is to use the *fitness distance plot* that plots the fitness of the solutions against their distance to the global optimal solution (Fig. 2.19). In the case of the presence of many global optima in the problem, the computation of the distance takes the nearest global optimum. One major limitation of the FDC analysis for real-life problems is that its computation requires the knowledge of the global optima. This is not realistic for many optimization problems. In that case, the best known solution may be selected to represent the global optimum. Other critical potential limitations are the

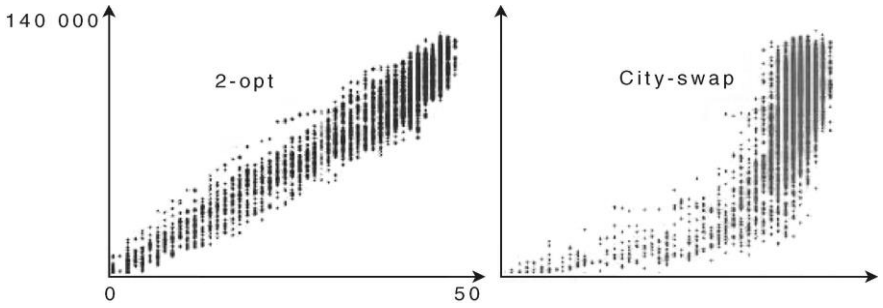


FIGURE 2.19 Using the FDC analysis, the figure shows the fitness distance plot of the instance *att48* of the TSP using the 2-opt and the city-swap neighborhood structures. The left figure for the 2-opt shows a high FDC (0.94) and the right figure shows a less important FDC for the city-swap operator (0.86). For the problem instance *tsp225*, the FDC is 0.99 for the 2-opt and 0.61 for the city swap.

[8]Empirical experiments propose an interval $[-0.15, 0.15]$ [429].

generation of k-distant solutions and the computation of the distance between solutions.

Example 2.17 Landscape and neighborhoods. This example illustrates the importance of avoiding rugged multivalleys in a landscape. Let us consider the symmetric Euclidean TSP (2D or 3D) and analyze the landscape associated with two different neighborhoods: the *city swap* that consists in swapping two cities and the 2-opt that replaces two edges. The landscape associated with the 2-opt may be viewed as a "massif central,"[9] that is, a big deep valley where the local optimal solutions are gathered around the global optimal solution (low average distance to the global optimal solution and low entropy of the population of local optimal solutions) [750]. Indeed, the local optimal solutions are clustered in a reduced region of the search space; the distance between any local optima s_i and the global optimal solution s^* is reduced ($\delta(s_i, s^*) \leq n/3$, where n is the input size). The local optimal solutions share over 60% of their edges. Many studies reach the same conclusion [83,465]. One can reuse this knowledge to design an efficient hybrid metaheuristic [275]. In the 2-opt landscape, a simple local search algorithm allows to obtain relatively good results ($\text{Gap}(O) \leq 10\%$) [750].

This "massif central" structure is diluted when using the city-swap operator, which gives worst local optima compared to the 2-opt operator. The landscape associated with the city-swap operator has a rugged multivalley structure. The local optima are as uniform as random solutions. Moreover, the FDC coefficient and the correlation length are more important for the 2-opt landscape than the city-swap landscape for all instances of the TSPlib [646]. For the 2-opt operator, the FDC correlation is closer to 1; that is, more the solution is closer to the global optimal solution, better is the quality of the solution (Fig. 2.19).

Other measures may be used to characterize the difficulty of problems:

- **Deception:** Deception defines the presence of local optima that attract the metaheuristic away from the global optimum. Some difficult deceptive problems have been constructed for genetic algorithms [327,629].
- **Epistasis:** Epistasis is defined by the interactions of decision variables (solution components). In other words, the epistasis defines the amount of nonlinearity in the problem. A high epistasis is characterized by a high degree of interaction between the different decision variables of the problem. The more rugged is the landscape, the higher is the epistasis [180]. Following the NK-model of landscape, the larger is the value K, the more important is the epistasis of the problem.
- **Multimodality:** It defines the number of local optima in the search space.
- **Neutrality:** It defines the flatness of the landscape, that is, the existence of plateaus: regions with equal fitness.
- **Fractal:** When the variance of the difference in the fitness between two solutions of the search space scales as a *power law* with their distance from each other,

[9] Referred also as "big valley" structure.

the landscape is a fractal; that is,

$$(\| \boldsymbol{f}(s) - \boldsymbol{f}(t) \|^2) \propto d(s, t)^{2h}$$

where s and t represent any solutions of the search space s and t ($s \neq t$) [813].

From a pragmatic point of view, the landscape of an easy instance is a valley (see Table 2.1); on the contrary, the landscape of a difficult instance is a rugged plain. Indeed, the metaheuristic methods easily manage to discover the bottom of a valley, and it is then enough to intensify the search in this area. With regard to the flat and rugged landscapes, there are a lot of places of good quality, but the latter are very dispersed and reaching the best is difficult; one can diversify much, but it is not sufficient for great spaces. For an intermediate landscape, the search algorithms should balance the intensification and the diversification, but this task is uneasy. According to this assumption, a promising approach consists in modifying the structure of the landscape to make it easier, that is, composed of a few deep valleys. The landscape can be modified by changing the neighborhood operator or the objective function.

Example 2.18 Landscape and instances structure. Let us consider the landscape analysis of the quadratic assignment problem where a known standard library exist: the QAPlib [96]. The landscape analysis shows that following the length of walks and the distribution of the local optima (entropy and distance), three different classes of instances may be defined (Fig. 2.20) [42,542]:

- **Nonstructured Landscape:** The instances of this type (type I) are almost all of a random uniform nature. The local optima are very scattered over the search space; the range of their quality is small and their average cost is rather good. So, a local search heuristic will "see" a flat rugged landscape with a good quality of solutions. In such a nonstructured landscape in which all regions have more or less the same quality, a local search will find quickly a good solution.
- **"Massif central":** This class (type II) includes only real or pseudoreal instances. It is characterized by a longer length of walks and a more clustered local optima (small entropy, small diameter), constituting a small number of concentrated "massifs." The landscape is composed of few deep valleys, where the best solutions are localized. In this landscape, an effective local search, such as tabu search, easily finds a promising valley and then exploits it.
- **"Multimassif ":** This class (type III) comprises only large instances. The landscape contains multiple valleys more or less deep, in which the local optima are gathered (small entropy, medium diameter). For more efficiency, a metaheuristic has to both exploit the bottom of the valleys and explore the search space enough to ensure that the best valleys will be exploited. In fact, the exploitation of the bottom of a deep valley (as done by tabu search, for instance) is not needed if this valley is not the best one. However, the main difficulty in designing a metaheuristic is to

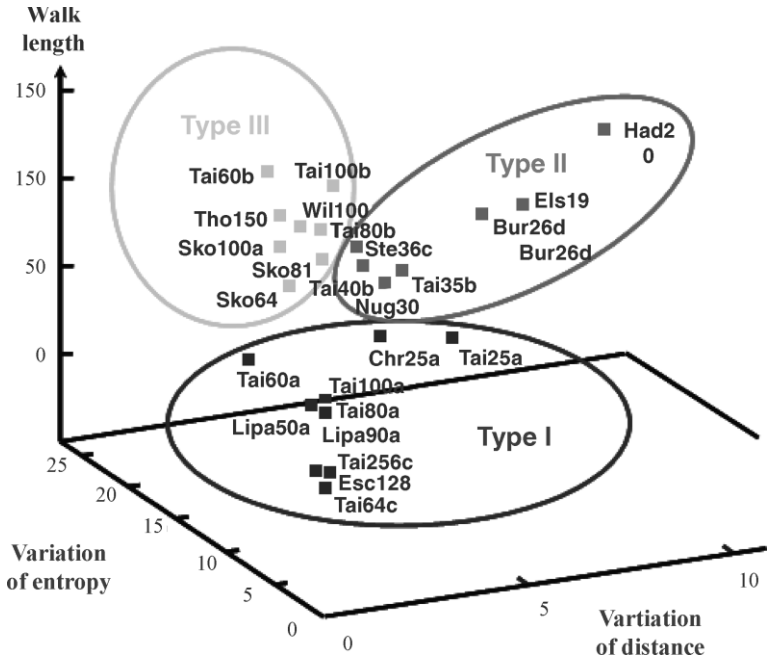


FIGURE 2.20 Correlation between the walk length and the distribution of the local optima (entropy and distance). We observe three different classes of instances.

balance the exploitation of valleys and the exploration of multiple valleys. As a result, the instances of type III are really more difficult to solve by a local search algorithm.

Example 2.19 Landscape and infeasible solutions. Let us consider the graph bi-partitioning problem. Given an undirected graph $G = (V, E)$, the problem is to find a partition of the nodes in two equally sized groups G_1 and G_2, which minimizes the number of edges between nodes of different groups:

$$\text{Min}|(e(G_1, G_2))|$$

where $e(G_1, G_2) = \{(i,j) \in E/i \in G_1, j \in G_2\}$. A straightforward encoding is the binary representation in which each node of the graph is assigned to a group. Let us consider two move operators:

- Exchange operator that consists in swapping the position of a pair of nodes of the graph.
- Flip operator that consists in changing the location of a single node. This move operator may generate infeasible solutions. The search space is much larger than the first one. A penalty function is then integrated in the objective function [29].

The landscape analysis of the GBP shows that the landscape associated with the flip operator presents a higher correlation length and is more smoother than the exchange operator landscape [29]. Hence, the local search heuristic shows a better effectiveness in the flip landscape. This example illustrates that enlarging the search space to infeasible solutions may be more efficient.

2.2.3 Breaking Plateaus in a Flat Landscape

Many optimization problems are characterized by a flat landscape[10] in which there are many plateaus (i.e., many neighbors are equally fitted). The plateaus are tediously crossed by metaheuristics. Indeed, no information will guide the search toward better regions. Hence, it is important to break the plateaus of a landscape to make the optimization problem more easy to solve.

One such strategy consists in changing the objective function. The objective function is generally defined rather straightforward in problem solving: it is simply the value of the utility function to optimize. For some problems, embedding more information in the objective function leads to a significant improvement in the quality of the solutions that are reached.

Definition 2.10 Plateau. *Given a point s in the search space S and a v value taken in the range of values of the criterion f. Given $N(s)$, the set of points s' in the neighborhood of the solution s. Considering X a subset of $N(s)$ defined by $s'' \in X$ iff $f(s'') = v$, X is a plateau iff it contains at least two elements (i.e., $|X| \geq 2$).*

When the landscape is composed of several plateaus, the main objective function of the problem is not sufficient to discriminate neighbors of the current point. On each plateau, a metaheuristic has difficulties to be guided in the neighborhood of the current solution. In this situation, using other information to break those plateaus and guide the search can be useful.

A secondary criterion f' is useful if it can discriminate points that have the same value for the main criterion f and is correlated with the main objective of the problem. The "discrimination power" is thus an important characteristic of a criterion.

Definition 2.11 Discrimination power. *Let $\mathcal{B} \neq (f')$ be the number of different values that are taken by the criterion f' on a plateau of the main criterion f. The discrimination power of the criterion f', $\varphi \neq (f')$, is the average number of $\mathcal{B} \neq (f')$ on a set of solutions.*

A discrimination criterion f' such that $\varphi \neq (f') \approx 1$ is not a good criterion since, on average, there is only one value of f' on each plateau of f. This means that f' cannot be used to guide the metaheuristic on the plateaus of f.

The new objective function can be defined as

$$f''(x) = k_1 \times f(x) + f'(x)$$

[10]Also referred as neutral.

where $\boldsymbol{f}$ is the main objective function of the problem, $\boldsymbol{f}'$ is the discrimination criterion, and k_1 is an upper bound for $\boldsymbol{f}'(x)$ over all solutions of the search space. One needs to define the upper bound for the constant k_1 so that the main objective remains the most important.

Using extra information would help discriminate the solutions with the same value in the objective space but with a different value with regard to the decision space (search space). Generally, this approach does not require any additional work except the implementation of the discrimination criterion. The computational cost of the integrated secondary objectives must be taken into account. Obviously, this depends on the kind of onside properties that are used and the cost to compute them. Another application of this approach in P-metaheuristics would be to maintain the diversity of the population, a key point of a good search in P-metaheuristics.

Many real-life and academic optimization problems are characterized by plateaus in their landscape: scheduling problems using the makespan as the objective [232], satisfiability (SAT) problems using the number of nonsatisfied clauses as the objective [432], graph coloring problem using the number of colors as the objective, and so on.

Example 2.20 Breaking the plateaus of the job-shop scheduling problem. Let us consider here the job-shop scheduling problem presented in Example 1.33. The landscape of the problem using the makespan objective has many plateaus [232]. In addition to the discrimination power of a criterion, the correlation with the main objective must also be taken into account. Two secondary criteria may be integrated into the objective function $\boldsymbol{f}$: H_2, and C_{op}. $H_2(x)$ can be defined as [374]

$$H_2(x) = \sum_{m=1}^{M} E_m^2(x)$$

where $E_m(x)$ is the completion time of the last operation performed on machine m according to the schedule x. $C_{\text{op}}(x)$ represents the number of critical operations of the schedule x. A critical operation is one that, if delayed, would increase the makespan of the schedule if reordering the operations is not allowed. It is commonly held that the less critical operations a schedule contains, the better the schedule is.

The new objective function can be defined as

$$\boldsymbol{f}'(x) = k_1 \times k_2 \times C_{\max}(x) + k_2 \times H_2(x) + C_{\text{op}}(x)$$

where $C_{\max}$ is the makespan, which is the main objective function of the problem

$$C_{\max}(x) = \max_{1 \le m \le M} \{E_m(x)\}$$

k_1 is an upper bound for $H_2(x)$ over all the possible busy schedules,[11] and k_2 is an upper bound for the criterion $C_{\text{op}}(x)$ over all possible schedules. We need to define the upper bounds for the constants k_1 and k_2 so that the makespan remains the main

[11] A schedule x is a busy schedule if for each time slot t, where $0 \le t \le C_{\max}(x)$, there is at least one operation running on a machine.

criterion. k_1 may be equal to $M \times D^2$, where D is the sum of the durations of all the operations of the instance. It is easy to see that the makespan of busy schedules is always smaller than this value. k_2 may be equal to $J \times M$, as there are $J \times M$ operations in the problem, so that there are no more than $J \times M$ critical operations.

Using the *same* metaheuristics, the new objective function $\boldsymbol{f}'$ provides better results in terms of the quality of the local optima that are found [232]. For this problem, this approach does not require a lot of implementation work and the computational cost is low.

The landscape analysis has some limitations. For instance, some measures require data that are difficult to obtain for complex real-life problems: global optimal solutions, number of local optima, distance definition, all solutions of distance d, and so on. However, approximations may be used to deal with those difficulties. As those approximations may lead to incorrect predictions, careful analysis must be carried out from those data [27,575]. Landscape analysis remains an interesting, valuable, and unavoidable tool for the analysis of the different search components of metaheuristics in solving complex optimization problems. The use of a set of different indicators is encouraged, rather than a single magic wand, to analyze the landscape of a problem and its difficulty.

2.3 LOCAL SEARCH

Local search[12] is likely the oldest and simplest metaheuristic method [3,598]. It starts at a given initial solution. At each iteration, the heuristic replaces the current solution by a neighbor that improves the objective function (Fig. 2.21). The search

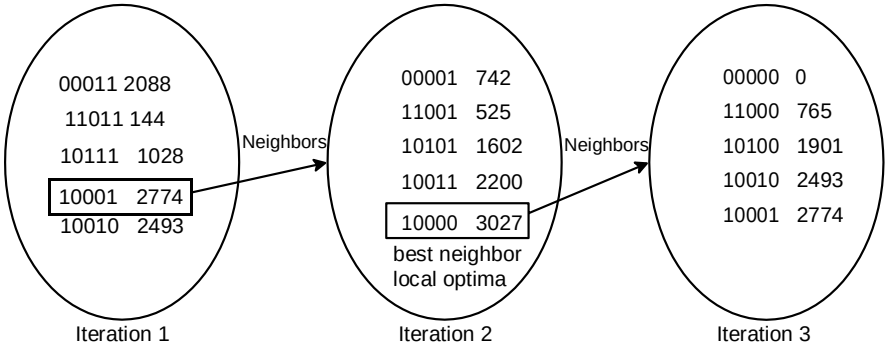


FIGURE 2.21 Local search process using a binary representation of solutions, a flip move operator, and the best neighbor selection strategy. The objective function to maximize is $x^3 - 60x^2 + 900x$. The global optimal solution is $\boldsymbol{f}(01010) = \boldsymbol{f}(10) = 4000$, while the final local optima found is $s = (10000)$, starting from the solution $s_0 = (10001)$.

[12]Also referred as hill climbing, descent, iterative improvement, and so on. In some literature, local search also refers to general S-metaheuristics.

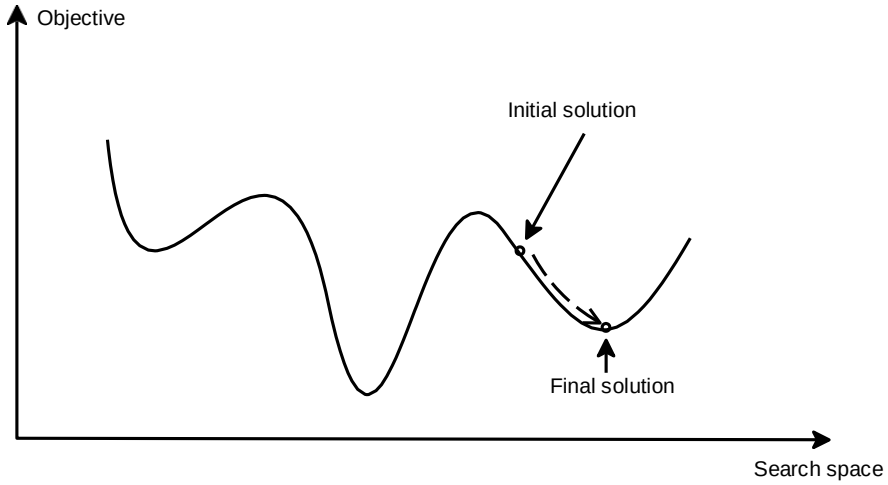


FIGURE 2.22 Local search (steepest descent) behavior in a given landscape.

stops when all candidate neighbors are worse than the current solution, meaning a local optimum is reached. For large neighborhoods, the candidate solutions may be a subset of the neighborhood. The main objective of this restricted neighborhood strategy is to speed up the search. Variants of LS may be distinguished according to the order in which the neighboring solutions are generated (deterministic/stochastic) and the selection strategy (selection of the neighboring solution) (Fig. 2.22).

LS may be seen as a descent walk in the graph representing the search space. This graph may be defined by $G = (S, V)$, where S represents the set of all feasible solutions of the search space and V represents the neighborhood relation. In the graph G, an edge (i, j) will connect to any neighboring solutions s_i and s_j. For a given solution s, the number of associated edges will be $|N(s)|$ (number of neighbors). Algorithm 2.2 illustrates the template of a local search algorithm.

Algorithm 2.2 Template of a local search algorithm.

```
s = s0 ; /* Generate an initial solution s0 */
While not Termination_Criterion Do
   Generate (N(s)) ; /* Generation of candidate neighbors */
   If there is no better neighbor Then Stop ;
   s = s' ; /* Select a better neighbor s' ∈ N(s) */
Endwhile
Output Final solution found (local optima).
```

From an initial solution s_0, the algorithm will generate a sequence $s_1, s_2, \ldots, s_k$ of solutions with the following characteristics (Fig. 1.32):

- The size of the sequence k is unknown a *priori*.
- $s_{i+1} \in N(s_i)$, $\forall i \in [0, k-1]$.

- $f(s_{i+1}) < f(s_i), \forall i \in [0, k-1]$.[13]
- s_k is a local optimum: $f(s_k) \leq f(s), \forall s \in N(s_k)$.

In addition to the definition of the initial solution and the neighborhood, designing a local search algorithm has to address the selection strategy of the neighbor that will determine the next current solution.

2.3.1 Selection of the Neighbor

Many strategies can be applied in the selection of a better neighbor (Fig. 2.23):

2.3.1.1 **Best improvement (steepest descent):** In this strategy, the best neighbor (i.e., neighbor that improves the most the cost function) is selected. The neighborhood is evaluated in a fully deterministic manner. Hence, the exploration of the neigh- borhood is *exhaustive*, and all possible moves are tried for a solution to select the best neighboring solution. This type of exploration may be time-consuming for large neighborhoods.

2.3.1.2 **First improvement:** This strategy consists in choosing the first improving neigh- bor that is better than the current solution. Then, an improving neighbor is im- mediately selected to replace the current solution. This strategy involves a partial evaluation of the neighborhood. In a *cyclic* exploration, the neighborhood is eval- uated in a deterministic way following a given order of generating the neighbors. In the worst case (i.e., when no improvement is found), a complete evaluation of the neighborhood is performed.

2.3.1.3 **Random selection:** In this strategy, a random selection is applied to those neigh- bors improving the current solution.

A compromise in terms of quality of solutions and search time may consist in using the first improvement strategy when the initial solution is randomly generated

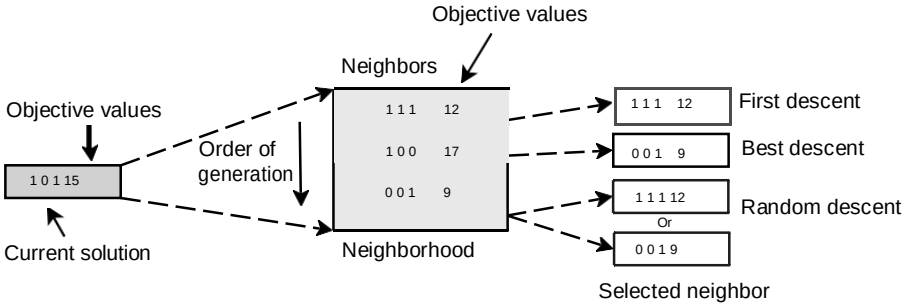


FIGURE 2.23 Selection strategies of improving neighbors.

[13]The problem is supposed to be a minimization problem.

and the best improvement strategy when the initial solution is generated using a greedy procedure. In practice, on many applications, it has been observed that the first improving strategy leads to the same quality of solutions as the best improving strategy while using a smaller computational time. Moreover, the probability of premature convergence to a local optima is less important in the first improvement strategy.

Example 2.21 Exhaustive versus cyclic exploration. Let us consider the neighborhood for the TSP that is based on the 2-opt move operator. A tour is represented by a permutation $(\pi_1, \pi_2, \ldots, \pi_n)$. In the exhaustive exploration, all 2-opt moves are applied. In a cyclic exploration, the order may be given as follows: apply successfully the 2-opt operator using the exchange of vertices (π_1, π_2) with (π_3, π_4), (π_4, π_5), and so on until an improving neighbor is found. In the worst case, the whole neighborhood is explored.

The order of application can be changed during the search. For instance, the initial parameter that determines the order may be chosen randomly. Instead of using always the edge (π_1, π_2), any edge (π_i, π_{i+1}) may represent the initial move.

Example 2.22 Cyclic exploration for a clique partitioning problem (CPP). Given a complete undirected graph $G = (V, E, w)$, where w represents positive or negative weights. To each edge (i, j) is associated a weight $w(i, j) = w(j, i)$. The objective of the CPP problem is to find a partition of the set of vertices V into k cliques $C_1, C_2, .., C_k$, minimizing the sum of the weights of the edges that have both end points in the same clique:

$$f(C_1, C_2, \ldots, C_k) = \frac{1}{2} \sum_{l=1} \sum_{(i,j) \in C_l \times C_l, i \neq j} w(i, j)$$

The number of cliques k is not fixed [125].

A solution s for the CPP problem may be represented by a discrete vector, where each element of the vector specifies the class of the vertex. The size of the vector is equal to the number of vertices $(n = |V|)$. The neighborhood may be defined by the transformation operator that consists in moving a single node to another class C. A class C of a solution s may be empty. The size of this neighborhood varies during the search and is equal to $(k + 1)n - \alpha$, where k is the number of classes and α is the number of single-node classes.

The cyclic exploration of a neighborhood may be based on finding the best class for a given node i. Obviously, one can use the given order of the vertices $(v_1, v_2, \ldots, v_n)$ or a random order generated uniformly. The vertices are explored in the specified order. A move is carried out as soon as an improving solution is found. All the nodes of the graph are tried once before the next time. This complete cyclic exploration is iterated until no improving neighbor is found. At the end of the process, all the nodes are in their best classes (i.e., local optimum related to the given move operator).

2.3.2 Escaping from Local Optima

In general, local search is a very easy method to design and implement and gives fairly good solutions very quickly. This is why it is a widely used optimization method in practice. One of the main disadvantages of LS is that it converges toward local optima. Moreover, the algorithm can be very sensitive to the initial solution; that is, a large variability of the quality of solutions may be obtained for some problems. Moreover, there is no means to estimate the relative error from the global optimum and the number of iterations performed may not be known in advance. Even if the complexity, in practice, is acceptable, the worst case complexity of LS is exponential! Local search works well if there are not too many local optima in the search space or the quality of the different local optima is more or less similar. If the objective function is highly multimodal, which is the case for the majority of optimization problems, local search is usually not an effective method to use.

As the main disadvantage of local search algorithms is the convergence toward local optima, many alternatives algorithms have been proposed to avoid becoming stuck at local optima. These algorithms are become popular since the 1980s. Four different families of approaches can be used to avoid local optima (Fig. 2.24) :

2.3.2.1 **Iterating from different initial solutions:** This strategy is applied in multistart local search, iterated local search, GRASP, and so forth.

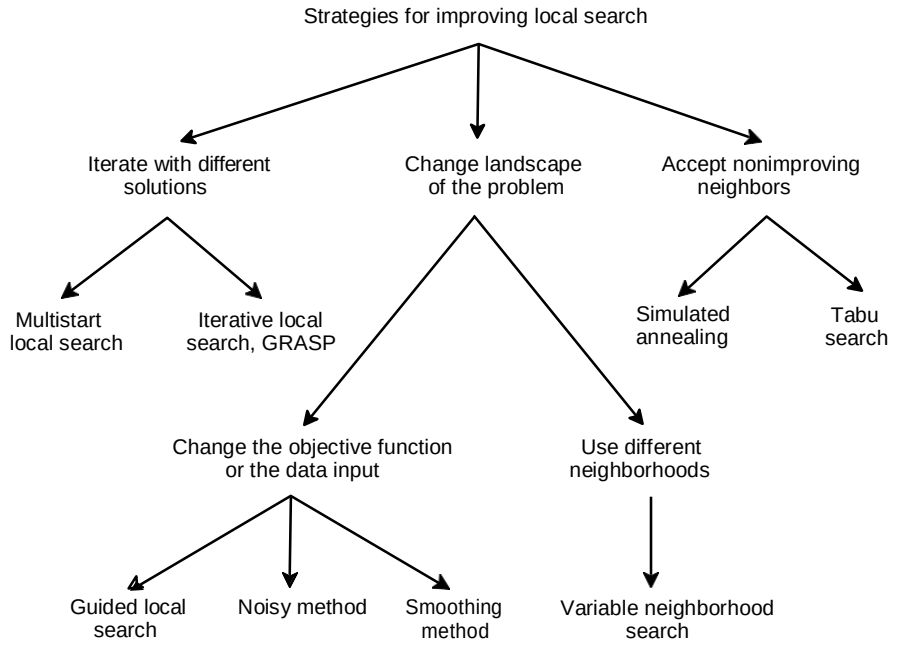


FIGURE 2.24 S-metaheuristic family of algorithms for improving local search and escaping from local optima.

2.3.2.2 **Accepting nonimproving neighbors:** These approaches enable moves that degrade the current solution. It becomes possible to move out the basin of attraction of a given local optimum. Simulated annealing and tabu search are popular representative of this class of algorithms. Simulated annealing was the first algorithm addressing explicitly the question "why should we consider only downhill moves?"

2.3.2.3 **Changing the neighborhood:** This class of approaches consists in changing the neighborhood structure during the search. For instance, this approach is used in variable neighborhood search strategies.

2.3.2.4 **Changing the objective function or the input data of the problem:** In this class, the problem is transformed by perturbing the input data of the problem, the objective function or the constraints, in the hope to solve more efficiently the original problem. This approach has been implemented in the guided local search, the smoothing strategies, and the noising methods. The two last approaches may be viewed as approaches changing the landscape of the problem to solve.

2.4 SIMULATED ANNEALING

Simulated annealing applied to optimization problems emerges from the work of S. Kirkpatrick et al. [464] and V. Cerny [114]. In these pioneering works, SA has been applied to graph partitioning and VLSI design. In the 1980s, SA had a major impact on the field of heuristic search for its simplicity and efficiency in solving combinatorial optimization problems. Then, it has been extended to deal with continuous optimization problems [204,512,596].

SA is based on the principles of statistical mechanics whereby the annealing process requires heating and then slowly cooling a substance to obtain a strong crystalline structure. The strength of the structure depends on the rate of cooling metals. If the initial temperature is not sufficiently high or a fast cooling is applied, imperfections (metastable states) are obtained. In this case, the cooling solid will not attain thermal equilibrium at each temperature. Strong crystals are grown from careful and slow cooling. The SA algorithm simulates the energy changes in a system subjected to a cooling process until it converges to an equilibrium state (steady frozen state). This scheme was developed in 1953 by Metropolis [543].

Table 2.4 illustrates the analogy between the physical system and the optimization problem. The objective function of the problem is analogous to the energy state of the system. A solution of the optimization problem corresponds to a system state. The decision variables associated with a solution of the problem are analogous to the molecular positions. The global optimum corresponds to the ground state of the system. Finding a local minimum implies that a metastable state has been reached.

SA is a stochastic algorithm that enables under some conditions the degradation of a solution. The objective is to escape from local optima and so to delay the convergence. SA is a memoryless algorithm in the sense that the algorithm does not

TABLE 2.4 Analogy Between the Physical System and the Optimization Problem

Physical System	Optimization Problem
System state	Solution
Molecular positions	Decision variables
Energy	Objective function
Ground state	Global optimal solution
Metastable state	Local optimum
Rapid quenching	Local search
Temperature	Control parameter T
Careful annealing	Simulated annealing

use any information gathered during the search. From an initial solution, SA proceeds in several iterations. At each iteration, a random neighbor is generated. Moves that improve the cost function are always accepted. Otherwise, the neighbor is selected with a given probability that depends on the current temperature and the amount of degradation *OE* of the objective function. *OE* represents the difference in the objective value (energy) between the current solution and the generated neighboring solution. As the algorithm progresses, the probability that such moves are accepted decreases (Fig. 2.25). This probability follows, in general, the Boltzmann distribution:

$$P(OE, T) = \mathrm{e}^{-\frac{f(s')-f(s)}{T}}$$

It uses a control parameter, called temperature, to determine the probability of accepting nonimproving solutions. At a particular level of temperature, many trials are explored. Once an equilibrium state is reached, the temperature is gradually decreased

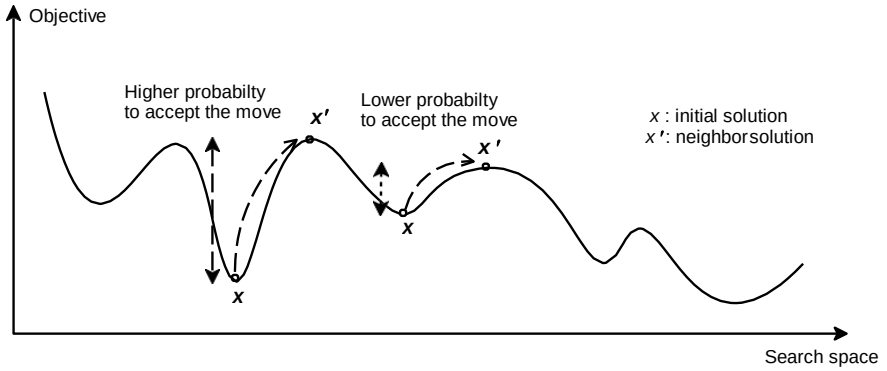


FIGURE 2.25 Simulated annealing escaping from local optima. The higher the temperature, the more significant the probability of accepting a worst move. At a given temperature, the lower the increase of the objective function, the more significant the probability of accepting the move. A better move is always accepted.

according to a cooling schedule such that few nonimproving solutions are accepted at the end of the search. Algorithm 2.3 describes the template of the SA algorithm.

Algorithm 2.3 Template of simulated annealing algorithm.

Input: Cooling schedule.
$s = s_0$; /* Generation of the initial solution */
$T = T_{max}$; /* Starting temperature */
Repeat
  Repeat /* At a fixed temperature */
    Generate a random neighbor s' ;
    $OE = f(s') - f(s)$;
    If $OE \leq 0$ **Then** $s = s'$ /* Accept the neighbor solution */
    Else Accept s' with a probability $e^{\frac{-OE}{T}}$;
  Until Equilibrium condition
  /* e.g. a given number of iterations executed at each temperature T */
  $T = g(T)$; /* Temperature update */
Until Stopping criteria satisfied /* e.g. $T < T_{min}$ */
Output: Best solution found.

Example 2.23 Illustration of the SA algorithm. Let us maximize the continuous function $f(x) = x^3 - 60x^2 + 900x + 100$. A solution x is represented as a string of 5 bits. The neighborhood consists in flipping randomly a bit. The global maximum of this function is 01010 ($x = 10, f(x) = 4100$). The first scenario starts from the solution 10011 ($x = 19, f(x) = 2399$) with an initial temperature T_0 equal to 500 (Table 2.5). The second scenario starts from the same solution 10011 with an initial temperature T_0 equal to 100 (Table 2.6). The initial temperature is not high enough and the algorithm gets stuck by local optima.

In addition to the current solution, the best solution found since the beginning of the search is stored. Few parameters control the progress of the search, which are the temperature and the number of iterations performed at each temperature.

Theoretical analysis of the asymptotic convergence of SA is well developed [478]. The search may be modeled by a Markov chain, where the next state depends only

TABLE 2.5 First Scenario $T = 500$ and Initial Solution (10011)

T	Move	Solution	f	Of	Move?	New Neighbor Solution
500	1	00011	2287	112	Yes	00011
450	3	00111	3803	<0	Yes	00111
405	5	00110	3556	247	Yes	00110
364.5	2	01110	3684	<0	Yes	01110
328	4	01100	3998	<0	Yes	01100
295.2	3	01000	3972	16	Yes	01000
265.7	4	01010	**4100**	<0	Yes	01010
239.1	5	01011	4071	29	Yes	01011
215.2	1	11011	343	3728	No	01011

TABLE 2.6 Second Scenario: T = 100 and Initial Solution (10011). When Temperature is not High Enough, Algorithm Gets Stuck

T	Move	Solution	f	Of	Move?	New Neighbor Solution
100	1	00011	2287	112	No	10011
90	3	10111	1227	1172	No	10011
81	5	10010	2692	< 0	Yes	10010
72.9	2	11010	516	2176	No	10010
65.6	4	10000	**3236**	< 0	Yes	10000
59	3	10100	2100	1136	Yes	10000

on the current state. There is a guarantee of convergence toward the optimal solution:

$$\Pr(s_M \in R) \to 1 \quad \text{as} \quad M \to \infty$$

where R represents the set of global optimal solutions and s_M the solution at iteration M under the following slow cooling schedule:

$$T_k = \frac{\hat{W}}{\log k}$$

where $\hat{W}$ is a constant. In practice, such a cooling schedule is useless because it is an asymptotic convergence; that is, the convergence is obtained after an infinite number of iterations. However, much more work is needed in the analysis of finite time performance [267].

In addition to the common design issues for S-metaheuristics such as the definition of the neighborhood and the generation of the initial solution, the main design issues specific to SA are

- **The acceptance probability function:** It is the main element of SA that enables nonimproving neighbors to be selected.
- **The cooling schedule:** The cooling schedule defines the temperature at each step of the algorithm. It has an essential role in the efficiency and the effectiveness of the algorithm.

The following sections present a practical guideline in the definition of the acceptance probability function and the cooling schedule in SA.

2.4.1 Move Acceptance

The system can escape from local optima due to the probabilistic acceptance of a nonimproving neighbor. The probability of accepting a nonimproving neighbor is proportional to the temperature T and inversely proportional to the change of the objective function OE.

The law of thermodynamics states that at temperature T, the probability of an increase in energy of magnitude, OE, is given by $P(OE, T) = \exp(-OE/kt)$ where k is a constant known as Boltzmann's constant. So, the acceptance probability of a nonimproving move is

$$P(OE,T) = \exp\left(\frac{-\delta E}{kt}\right) > R$$

where OE is the change in the evaluation function, T is the current temperature, and R is a uniform random number between 0 and 1.

At high temperatures, the probability of accepting worse moves is high. If $T = \infty$, all moves are accepted, which corresponds to a random local walk in the landscape. At low temperatures, the probability of accepting worse moves decreases. If $T = 0$, no worse moves are accepted and the search is equivalent to local search (i.e., hill climbing). Moreover, the probability of accepting a large deterioration in solution quality decreases exponentially toward 0 according to the Boltzmann distribution.

2.4.2 Cooling Schedule

The cooling schedule defines for each step of the algorithm i the temperature T_i. It has a great impact on the success of the SA optimization algorithm. Indeed, the performance of SA is very sensitive to the choice of the cooling schedule.

The parameters to consider in defining a cooling schedule are the starting temperature, the equilibrium state, a cooling function, and the final temperature that defines the stopping criteria. A guideline dealing with the initialization of each parameter is given next.

2.4.2.1 Initial Temperature If the starting temperature is very high, the search will be more or less a random local search. Otherwise, if the initial temperature is very low, the search will be more or less a first improving local search algorithm. Hence, we have to balance between these two extreme procedures. The starting temperature must not be too high to conduct a random search for a period of time but high enough to allow moves to almost neighborhood state.

There are three main strategies that can be used to deal with this parameter:

- **Accept all:** The starting temperature is set high enough to accept all neighbors during the initial phase of the algorithm [464]. The main drawback of this strategy is its high computational cost.
- **Acceptance deviation:** The starting temperature is computed by $k\sigma$ using preliminary experimentations, where σ represents the standard deviation of difference between values of objective functions and $k = -3/\ln(p)$ with the acceptance probability of p, which is greater than 3σ [392].

- **Acceptance ratio:** The starting temperature is defined so as to make the acceptance ratio of solutions greater than a predetermined value a_0

$$T_0 = \frac{O^+}{\ln(m_1(a_0 - 1)/m_2 + a_0)}$$

where m_1 and m_2 are the numbers of solutions to be decreased and increased in preliminary experiments, respectively, and O^+ is the average of objective function values increased [2]. For instance, the initial temperature should be initialized in such a way that the acceptance rate is in the interval [40%,50%].

2.4.2.2 Equilibrium State To reach an equilibrium state at each temperature, a number of sufficient transitions (moves) must be applied. Theory suggests that the number of iterations at each temperature might be exponential to the problem size, which is a difficult strategy to apply in practice. The number of iterations must be set according to the size of the problem instance and particularly proportional to the neighborhood size $|N(s)|$. The number of transitions visited may be as follows:

- **Static:** In a static strategy, the number of transitions is determined before the search starts. For instance, a given proportion y of the neighborhood $N(s)$ is explored. Hence, the number of generated neighbors from a solution s is $y \cdot |N(s)|$. The more significant the ratio y, the higher the computational cost and the better the results.
- **Adaptive:** The number of generated neighbors will depend on the characteristics of the search. For instance, it is not necessary to reach the equilibrium state at each temperature. Nonequilibrium simulated annealing algorithms may be used: the cooling schedule may be enforced as soon as an improving neighbor solution is generated. This feature may result in the reduction of the computational time without compromising the quality of the obtained solutions [107].

  Another adaptive approach using both the worst and the best solutions found in the inner loop of the algorithm may be used. Let f_l (resp. f_h) denote the smallest (resp. largest) objective function value in the current inner loop. The next number of transitions L is defined as follows:

$$L = L_B + \lfloor L_B \cdot F_- \rfloor$$

where $\lfloor x \rfloor$ is the largest integer smaller than x, $F_- = 1 - \exp(-(f_h - f_l)/f_h)$, and L_B is the initial value of the number of transitions [25].

2.4.2.3 Cooling In the SA algorithm, the temperature is decreased gradually such that

$$T_i > 0, \forall i$$

and

$$\lim_{i \to \infty} T_i = 0$$

There is always a compromise between the quality of the obtained solutions and the speed of the cooling schedule. If the temperature is decreased slowly, better solutions are obtained but with a more significant computation time. The temperature T can be updated in different ways:

- **Linear:** In the trivial linear schedule, the temperature T is updated as follows: $T = T - \beta$, where β is a specified constant value. Hence, we have

$$T_i = T_0 - i \times \beta$$

  where T_i represents the temperature at iteration i.
- **Geometric:** In the geometric schedule, the temperature is updated using the formula

$$T = \alpha T$$

  where $\alpha \in]0, 1[$. It is the most popular cooling function. Experience has shown that α should be between 0.5 and 0.99.
- **Logarithmic:** The following formula is used:

$$T_i = \frac{T_0}{\log(i)}$$

  This schedule is too slow to be applied in practice but has the property of the convergence proof to a global optimum [303].
- **Very slow decrease:** The main trade-off in a cooling schedule is the use of a large number of iterations at a few temperatures or a small number of iterations at many temperatures. A very slow cooling schedule such as

$$T_{i+1} = \frac{T_i}{1 + \beta T_i}$$

  may be used [521], where $\beta = T_0 - T_F/(L - 1)T_0T_F$ and T_F is the final temperature. Only one iteration is allowed at each temperature in this very slow decreasing function.
- **Nonmonotonic:** Typical cooling schedules use monotone temperatures. Some nonmonotone scheduling schemes where the temperature is increased again may be suggested [390]. This will encourage the diversification in the search space. For some types of search landscapes, the optimal schedule is nonmonotone [352].

- **Adaptive:** Most of the cooling schedules are static in the sense that the cooling schedule is defined completely *a priori*. In this case, the cooling schedule is "blind" to the characteristics of the search landscape. In an adaptive cooling schedule, the decreasing rate is dynamic and depends on some information obtained during the search [402]. A dynamic cooling schedule may be used where a small number of iterations are carried out at high temperatures and a large number of iterations at low temperatures.

2.4.2.4 Stopping Condition Concerning the stopping condition, theory suggests a final temperature equal to 0. In practice, one can stop the search when the probability of accepting a move is negligible. The following stopping criteria may be used:

- Reaching a final temperature T_F is the most popular stopping criteria. This temperature must be low (e.g., $T_{min} = 0.01$).
- Achieving a predetermined number of iterations without improvement of the best found solution [675].
- Achieving a predetermined number of times a percentage of neighbors at each temperature is accepted; that is, a counter increases by 1 each time a temperature is completed with the less percentage of accepted moves than a predetermined limit and is reset to 0 when a new best solution is found. If the counter reaches a predetermined limit R, the SA algorithm is stopped [420].

SA compared to local search is still simple and easy to implement. It gives good results for a wide spectrum of optimization problems: the historical ones such as TSP and VLSI design in different domains of application. A good survey on SA can be found in Refs [1,489,733].

2.4.3 Other Similar Methods

Other similar methods of simulated annealing have been proposed in the literature, such as threshold accepting, great deluge algorithm, record-to-record travel, and demon algorithms (Fig. 2.26). The main objective in the design of those SA-inspired algorithms is to speed up the search of the SA algorithm without sacrificing the quality of solutions.

2.4.3.1 Threshold Accepting Threshold accepting may be viewed as the deterministic variant of simulated annealing [228]. TA escapes from local optima by accepting solutions that are not worse than the current solution by more than a given threshold Q. A deterministic acceptance function is defined as follows:

$$P_i(O(s, s')) = \begin{cases} 1 \text{ if } Q_i \geq O(s, s') \\ 0 \text{ otherwise} \end{cases}$$

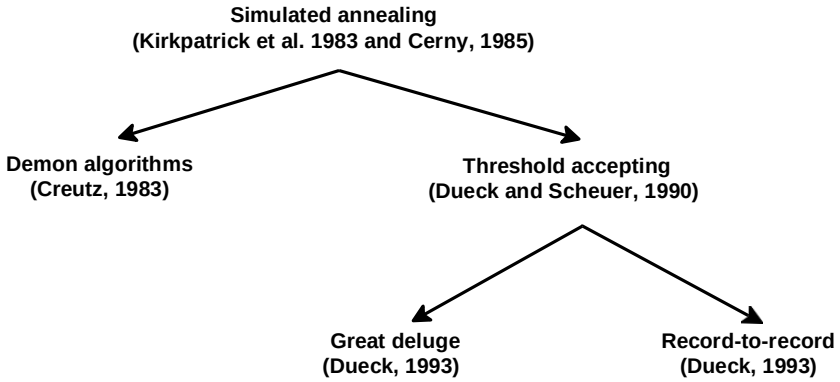


FIGURE 2.26 Genealogy of simulated annealing-based algorithms.

where Q_i is the threshold value at iteration i and $O(s, s')$ is the change in the evaluation function between the current solution s and the neighbor solution s'. The threshold parameter in TA operates somewhat like the temperature in simulated annealing. Algorithm 2.4 describes the template of the TA algorithm. The number of generated neighbors at each iteration is fixed *a priori*. The threshold Q is updated following any annealing schedule.

Algorithm 2.4 Template of threshold accepting algorithm.

Input: Threshold annealing.
$s = s_0$; /* Generation of the initial solution */
$Q = Q_{max}$; /* Starting threshold */
Repeat
  Repeat /* At a fixed threshold */
    Generate a random neighbor $s' \in N(s)$;
    $OE = f(s') - f(s)$;
    If $OE \leq Q$ **Then** $s = s'$ /* Accept the neighbor solution */
  Until Equilibrium condition
  /* e.g. a given number of iterations executed at each threshold Q */
  $Q = g(Q)$; /* Threshold update */
Until Stopping criteria satisfied /* e.g. $Q \leq Q_{min}$ */
Output: Best solution found.

TA is a fast algorithm compared to SA because the generation of random number and exponential functions consume a significant amount of computational time. The literature reports some performance improvements compared to the simulated annealing algorithm in solving combinatorial optimization problems such as the traveling salesman problem [228,565].

In terms of asymptotic convergence, the theoretical properties of TA are similar to those of the SA algorithm [28].

The threshold Q is updated according to an annealing schedule. It must be set as a deterministic nonincreasing step function in the number of iterations i. The threshold decreases at each iteration and then reaches the value of 0 after a given number of iterations.

Example 2.24 Nonmonotone threshold schedule. This simple example will illustrate that the optimal threshold schedule to solve a given problem may be nonmonotone [390]. Let us consider the one-dimensional landscape of a problem shown in Fig. 2.27. The search space is composed of six solutions, and each solution has two neighbors. The number of iterations of the algorithm is fixed to M. The optimal schedule is defined as the one that maximizes the probability of finding the global optimum solution starting from any random initial solution. For many values of M, it has been shown that the optimal threshold schedule is nonmonotone with negative thresholds. For instance, for $M = 9$, the optimal schedule is $(0, -3, 3, -1, -3, 3, 3, -1)$ and the probability to find the global optimum is 0.8372.

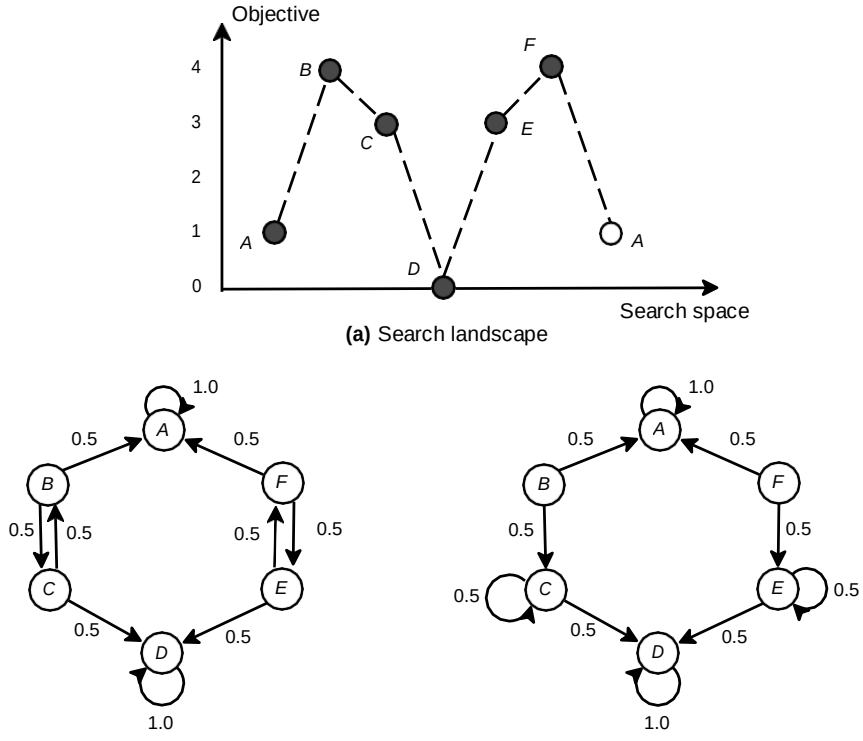


FIGURE 2.27 Nonmonotone optimal threshold schedule [390]. (a) Represents the search landscape of the problem. (b) (Resp. (c)) represents the transition probabilities for $T = 2 (T = 0)$.

Example 2.25 Adaptive nonmonotone threshold—the old bachelor acceptance. In the old bachelor acceptance algorithm, the threshold changes dynamically (up or down) based on the history of the search [390]. After each failure of accepting a new solution, the criterion for accepting is relaxed by slightly increasing the threshold Q_i[14]: $Q_{i+1} = Q_i + \text{incr}(Q_i)$. Hence, after many successive failures, the threshold will be large enough to escape from local optima. On the contrary, after each acceptance of a neighbor solution, the threshold is lowered so that the algorithm becomes more aggressive in moving toward a local optimum $Q_{i+1} = Q_i - \text{decr}(Q_i)$.
The decr and incr functions are based on the following factors:

- The neighborhood size $|N|$. This value has an impact on the "reachability" between solutions (i.e., the diameter and multiplicity of paths in the search space).
- The *age* of the current solution, which is the number of iterations since the last move acceptance. Larger values of the *age* implies greater likelihood that the current solution is a local optimum and that the threshold must increase.
- The amount of time remaining $M - i$, where M represents the total number of iterations and i the current iteration. The threshold update will depend on the proportion of the time used i/M.
- The current threshold value Q_i.

Algorithm 2.5 describes the template of the OBA algorithm.

Algorithm 2.5 Template of the old bachelor accepting algorithm.

$s = s_0$; /* Generation of the initial solution */
$Q_0 = 0$; /* Starting threshold */
$age = 0$;
For $i = 0$ to $M - 1$ **Do**
  Generate a random neighbor $s' \in N(s)$;
  If $f(s') < f(s) + Q_i$ **Then** $s = s'$; $age = 0$;
  Else $age = age + 1$;
  $Q_{i+1} = \left(\frac{age}{a}\right)^b - 1 \times O \times \left(1 - \frac{i}{M}\right)^c$ /* Threshold update */
Endfor
Output: Best solution found.

The threshold is updated as follows:

$$Q_{i+1} = \left(\frac{\text{age}}{a}\right)^b - 1 \times O \times \left(1 - \frac{i}{M}\right)^c$$

Whenever age $= 0$, the algorithm sets the threshold to the most negative value allowable (i.e., $-O$), thus giving the algorithm more "ambition" to improve rapidly. For negative

[14]This is the motivation for the name "old bachelor acceptance."

values of the threshold, the algorithm may prefer a good improving move over a random improving move. The threshold then rises from this negative value until the next move acceptance occurs.

For age > 0, the update rule allows the threshold growth rate to increase with age. The parameters O, a, b, and c afford the ability to fine-tune the growth rate $\text{incr}(Q_i)$ as follows: O represents the granularity of the update, a tunes the threshold growth by a multiplicative factor, b allows a power law growth rate, and c tunes a heuristic "dumping" factor $1 - (i/M)$ used to scale the magnitude of Q_i as $i \longrightarrow M$.

2.4.3.2 Record-to-Record Travel This algorithm is also a deterministic optimization algorithm inspired from simulated annealing [229]. The algorithm accepts a nonimproving neighbor solution with an objective value less than the RECORD minus a deviation D. RECORD represents the best objective value of the visited solutions during the search. The bound decreases with time as the objective value RECORD of the best found solution improves. Algorithm 2.6 describes the template of the RRT algorithm.

Algorithm 2.6 Template of the record-to-record travel algorithm.

Input: Deviation $D > 0$.
$s = s_0$; /* Generation of the initial solution */
$RECORD = f(s)$; /* Starting RECORD */
Repeat
  Generate a random neighbor s' ;
  If $f(s') < RECORD + D$ **Then** $s = s'$; /* Accept the neighbor solution */
  If $RECORD > f(s')$ **Then** $RECORD = f(s')$; /* RECORD update */
Until Stopping criteria satisfied
Output: Best solution found.

The RRT algorithm has the advantage to be dependent on only one parameter, the DEVIATION value. A small value for the deviation will produce poor results within a reduced search time. If the deviation is high, better results are produced after an important computational time.

2.4.3.3 Great Deluge Algorithm The great deluge algorithm was proposed by Dueck in 1993 [229]. The main difference with the SA algorithm is the deterministic acceptance function of neighboring solutions. The inspiration of the GDA algorithm comes from the analogy that the direction a hill climber would take in a great deluge to keep his feet dry. Finding the global optimum of an optimization problem[15] may be seen as finding the highest point in a landscape. As it rains incessantly without end, the level of the water increases. The algorithm never makes a move beyond the

[15]We suppose here a maximization problem.

water level. It will explore the uncovered area of the landscape to reach the global optimum.

Algorithm 2.7 describes the template of the GDA algorithm in a minimization context. A generated neighbor solution is accepted if the absolute value of the objective function is less than the current boundary value, named level. The initial value of the level is equal to the initial objective function. The level parameter in GDA operates somewhat like the temperature in SA. During the search, the value of the level is decreased monotonically. The decrement of the reduction is a parameter of the algorithm.

Algorithm 2.7 Template of the great deluge algorithm.

Input: Level L.
$s = s_0$; /* Generation of the initial solution */
Choose the rain speed UP ; /* $UP > 0$ */
Choose the initial water level $LEVEL$;
Repeat
  Generate a random neighbor s' ;
  If $f(s') < LEVEL$ **Then** $s = s'$ /* Accept the neighbor solution */
  $LEVEL = LEVEL - UP$; /* update the water level */
Until Stopping criteria satisfied
Output: Best solution found.

The great deluge algorithm needs the tuning of only one parameter, the UP value that represents the rain speed. The quality of the obtained results and the search time will depend only on this parameter. If the value of the UP parameter is high, the algorithm will be fast but will produce results of poor quality. Otherwise, if the UP value is small, the algorithm will generate relatively better results within a higher computational time. An example of a rule that can be used to define the value of the UP parameter may be the following [229]: a value smaller than 1% of the average gap between the quality of the current solution and the water level.

2.4.3.4 Demon Algorithms Since 1998 many S-metaheuristics based on the demon algorithm (DA) (see Ref. [165]) have been proposed [824]. The demon algorithm is another simulated annealing-based algorithm that uses computationally simpler acceptance functions.

Algorithm 2.8 describes the main demon algorithm. The acceptance function is based on the energy value of the demon (credit). The demon is initialized with a given value D. A nonimproved solution is accepted if the demon has more energy (credit) than the decrease of the objective value. When a DA algorithm accepts a solution of increased objective value, the change value of the objective is credited to the demon. In the same manner, when a DA algorithm accepts an improving solution, the decrease of the objective value is debited from the demon.

Algorithm 2.8 Template of the demon algorithm.

Input: Demon initial value D
$s = s_0$; /* Generation of the initial solution */
Repeat
  Generate a random neighbor s' ;
  $OE = f(s') - f(s)$;
  If $OE \leq D$ **Then**
    $s = s'$; /* Accept the neighbor solution */
    $D = D - OE$; /* Demon value update */
Until Stopping criteria satisfied
Output: Best solution found.

The acceptance function of demon algorithms is computationally simpler than in SA. It requires a comparison and a subtraction, whereas in SA it requires an exponential function and a generation of a random number. Moreover, the demon values vary dynamically in the sense that the energy (credit) depends on the visited solutions (Markov chain) during the search, whereas in SA and TA the temperature (threshold) is not dynamically reduced. Indeed, the energy absorbed and released by the demon depends mainly on the accepted solutions.

Different variants of the DA algorithm can be found in the literature [610,824]. They differ by the annealing schedule of the acceptance function:

- **Bounded demon algorithm:** This algorithm imposes an upper bound D_0 for the credit of the demon. Hence, once the credit of the demon is greater than the upper bound, no credit is received even if improving solutions are generated.
- **Annealed demon algorithm:** In this algorithm, an annealing schedule similar to the simulated annealing one is used to decrease the credit of the demon. The credit of the demon will play the same role as the temperature in simulated annealing.
- **Randomized bounded demon algorithm:** A randomized search mechanism is introduced in the BDA algorithm. The credit of the demon is replaced with a normal Gaussian random variable, where the mean equals the credit of the demon (D_m), and a specified standard deviation D_{sd}. Hence, the energy associated with the demon will be $D = D_m +$ Gaussian noise.
- **Randomized annealed demon algorithm:** The same randomized search mechanism of the RBDA algorithm is introduced as in the ADA algorithm.

Table 2.7 illustrates the specificities of the different variants of the demon algorithms.

Compared to simulated annealing, the application of demon algorithms to academic and real-life problems show competitive quality of results within a reduced search time [610,824]. Moreover, they are very easy to design and implement and they need tuning of few parameters.

TABLE 2.7 Variants of Demon Algorithms and Their Specific Parts

Algorithm	Specificity
BDA	Initial demon value (upper bound): D_0
	Demon value update: if $D > D_0$, then $D = D_0$
ADA	Demon value update: annealing schedule
RBDA and RADA	Initial demon value: mean D_m
	Acceptance function: $D = D_m +$ Gaussian noise
	Demon value update: $D_m = D_m - OE$

2.5 TABU SEARCH

Tabu search algorithm was proposed by Glover [323]. In 1986, he pointed out the controlled randomization in SA to escape from local optima and proposed a deterministic algorithm [322]. In a parallel work, a similar approach named "steepest ascent/mildest descent" has been proposed by Hansen [364]. In the 1990s, the tabu search algorithm became very popular in solving optimization problems in an approximate manner. Nowadays, it is one of the most widespread S-metaheuristics. The use of memory, which stores information related to the search process, represents the particular feature of tabu search.

TS behaves like a steepest LS algorithm, but it accepts nonimproving solutions to escape from local optima when all neighbors are nonimproving solutions. Usually, the whole neighborhood is explored in a deterministic manner, whereas in SA a random neighbor is selected. As in local search, when a better neighbor is found, it replaces the current solution. When a local optima is reached, the search carries on by selecting a candidate worse than the current solution. The best solution in the neighborhood is selected as the new current solution even if it is not improving the current solution. Tabu search may be viewed as a dynamic transformation of the neighborhood. This policy may generate cycles; that is, previous visited solutions could be selected again.

To avoid cycles, TS discards the neighbors that have been previously visited. It memorizes the recent search trajectory. Tabu search manages a memory of the solutions or moves recently applied, which is called the *tabu list*. This tabu list constitutes the short-term memory. At each iteration of TS, the short-term memory is updated. Storing all visited solutions is time and space consuming. Indeed, we have to check at each iteration if a generated solution does not belong to the list of all visited solutions. The tabu list usually contains a constant number of tabu moves. Usually, the attributes of the moves are stored in the tabu list.

By introducing the concept of solution features or move features in the tabu list, one may lose some information about the search memory. We can reject solutions that have not yet been generated. If a move is "good," but it is tabu, do we still reject it? The tabu list may be too restrictive; a nongenerated solution may be forbidden. Yet for some conditions, called *aspiration criteria*, tabu solutions may be accepted. The admissible neighbor solutions are those that are nontabu or hold the aspiration criteria.

In addition to the common design issues for S-metaheuristics such as the definition of the neighborhood and the generation of the initial solution, the main design issues that are specific to a simple TS are

- **Tabu list:** The goal of using the short-term memory is to prevent the search from revisiting previously visited solutions. As mentioned, storing the list of all visited solutions is not practical for efficiency issues.
- **Aspiration criterion:** A commonly used aspiration criteria consists in selecting a tabu move if it generates a solution that is better than the best found solution. Another aspiration criterion may be a tabu move that yields a better solution among the set of solutions possessing a given attribute.

Some advanced mechanisms are commonly introduced in tabu search to deal with the intensification and the diversification of the search:

- **Intensification (medium-term memory):** The medium-term memory stores the elite (e.g., best) solutions found during the search. The idea is to give priority to attributes of the set of elite solutions, usually in weighted probability manner. The search is biased by these attributes.
- **Diversification (long-term memory):** The long-term memory stores information on the visited solutions along the search. It explores the unvisited areas of the solution space. For instance, it will discourage the attributes of elite solutions in the generated solutions to diversify the search to other areas of the search space.

Algorithm 2.9 describes the template of the TS algorithm.

Algorithm 2.9 Template of tabu search algorithm.

```
s = s0 ; /* Initial solution */
Initialize the tabu list, medium-term and long-term memories ;
Repeat
  Find best admissible neighbor s' ; /* non tabu or aspiration criterion holds */
  s = s' ;
  Update tabu list, aspiration conditions, medium and long term memories ;
  If intensification_criterion holds Then intensification ;
  If diversification_criterion holds Then diversification ;
Until Stopping criteria satisfied
Output: Best solution found.
```

Theoretical studies carried out on tabu search algorithms are weaker than those established for simulated annealing. A simulated annealing execution lies within the convex hull of a set of tabu search executions [28]. Therefore, tabu search may inherit some nice theoretical properties of SA.

TABLE 2.8 The Different Search Memories of Tabu Search

Search Memory	Role	Popular Representation
Tabu list	Prevent cycling	Visited solutions, moves attributes Solution attributes
Medium-term memory	Intensification	Recency memory
Long-term memory	Diversification	Frequency memory

In addition to the search components of local search (hill climbing), such as the representation, neighborhood, initial solution, we have to define the following concepts that compose the search memory of TS: the tabu list (short-term memory), the medium-term memory, and the long-term memory (Table 2.8).

2.5.1 Short-Term Memory

The role of the short-term memory is to store the recent history of the search to prevent cycling. The naive straightforward representation consists in recording all visited solutions during the search. This representation ensures the lack of cycles but is seldom used as it produces a high complexity of data storage and computational time. For instance, checking the presence of all neighbor solutions in the tabu list will be prohibitive. The first improvement to reduce the complexity of the algorithm is to limit the size of the tabu list. If the tabu list contains the last k visited solutions, tabu search prevents a cycle of size at most k. Using hash codes may also reduce the complexity of the algorithms manipulating the list of visited solutions.

In general, attributes of the solutions or moves are used. This representation induces a less important data storage and computational time but skips some information on the history of the search. For instance, the absence of cycles is not ensured.

The most popular way to represent the tabu list is to record the move attributes. The tabu list will be composed of the reverse moves that are forbidden. This scheme is directly related to the neighborhood structure being used to solve the problem. If the move m is applied to the solution s_i to generate the solution s_j ($s_j = s_i \oplus m$), then the move m (or its reverse m^{-1} such that $(s_i \oplus m) \oplus m^{-1} = s_i$) is stored in the list. This move is forbidden for a given number of iterations, named the *tabu tenure* of the move. If the tabu list contains the last k moves, tabu search will not guarantee to prevent a cycle of size at most k.

Example 2.26 Tabu list based on move attributes. Let us consider a permutation optimization problem where the neighborhood is based on exchanging two elements of the permutation. Given a permutation π, a move is represented by two indices (i, j). This move generates the neighbor solution π' such that

$$\pi'(k) = \begin{cases} \pi(k) & \text{for } k \neq i \text{ and } k \neq j \\ \pi(j) & \text{for } k = i \\ \pi(i) & \text{for } k = j \end{cases}$$

The inverse move (j, i) may be stored in the tabu list and is forbidden. A stronger tabu representation may be related to the indices i and j. This will disallow any permutation involving the indices i and j.

In tabu search, many different tabu lists may be used in conjunction. For instance, some ingredients of the visited solutions and/or the moves are stored in multiple tabu lists. A move m is not tabu if the conditions for *all* tabu lists are satisfied. In many optimization problems, it is more and more popular to use simultaneously various move operators, and hence different neighborhoods are defined.

Example 2.27 Multiple tabu lists. Let us consider the maximum independent set problem. Given a graph $G = (V, E)$, the problem consists in finding a subset of vertices X of V, maximizing the cardinality of the set X, and satisfying the constraint that there is no edge $(i, j) \in E$ having its adjacent vertices in X. Tabu search may be applied to determine an independent set of size k [285]. The objective function consists in minimizing the number of edges having their adjacent vertices in X. The neighborhood is defined by the exchange of a vertex $v \in X$ with a vertex $w \in E - X$. The size of the neighborhood is $k(n - k) = O(nk)$, where n represents the number of vertices. Three different tabu lists may be used:

2.5.1.1 The first tabu list T_1 stores the last generated solutions. A hashing technique may be used to speed up the test of tabu conditions [825].

2.5.1.2 the second tabu list T_2 stores the last vertices introduced into X.

2.5.1.3 the third tabu list T_3 stores the last vertices removed from X.

The size of the tabu list is a critical parameter that has a great impact on the performances of the tabu search algorithm. At each iteration, the last move is added in the tabu list, whereas the oldest move is removed from the list. The smaller is the value of the tabu list, the more significant is the probability of cycling. Larger values of the tabu list will provide many restrictions and encourage the diversification of the search as many moves are forbidden. A compromise must be found that depends on the landscape structure of the problem and its associated instances.

The tabu list size may take different forms:

- **Static:** In general, a static value is associated with the tabu list. It may depend on the size of the problem instance and particularly the size of the neighborhood. There is no optimal value of the tabu list for all problems or even all instances of a given problem. Moreover, the optimal value may vary during the search progress. To overcome this problem, a variable size of the tabu list may be used.
- **Dynamic:** The size of the tabu list may change during the search without using any information on the search memory.

Example 2.28 Robust tabu search. In this tabu search algorithm, a uniform random change of the tabu list size in a given interval $[T_{min}, T_{max}]$ is used [738].

- **Adaptive:** In the adaptive scheme, the size of the tabu list is updated according to the search memory. For instance, the size is updated upon the performance of the search in the last iterations [573].

Example 2.29 Reactive tabu search. In the reactive tabu search, the main idea is to increase the size of the tabu list when the short-term memory indicates that the search is revisiting solutions, that is, the presence of a cycle in the search [59]. Hence, the short-term memory stores the visited solutions and not the attributes of the solutions or moves. Hashing techniques are used to reduce the complexity of manipulating such short-term memories.

2.5.2 Medium-Term Memory

The role of the intensification is to exploit the information of the best found solutions (elite solutions) to guide the search in promising regions of the search space. This information is stored in a medium-term memory. The idea consists in extracting the (common) features of the elite solutions and then intensifying the search around solutions sharing those features. A popular approach consists in restarting the search with the best solution obtained and then fixing in this solution the most promising components extracted from the elite solutions.

The main representation used for the medium-term memory is the *recency memory*. First, the components associated with a solution have to be defined; this is a problem-specific task. The recency memory will memorize for each component the number of successive iterations the component is present in the visited solutions. The most commonly used event to start the intensification process is a given period or after a certain number of iterations without improvement.

Example 2.30 Recency memory in solving the GAP. Suppose one has to solve the generalized assignment problem where a solution is encoded by a discrete vector and the neighborhood is defined by the substitute operator. The recency memory may be represented by a symmetric matrix R where an element r_{ij} represents the number of successive iterations the object i has been assigned to the site j. The intensification will consist in starting the search from the best found solution $s*$. The largest values r_{ij} are extracted from the recency matrix R. Then the associated elements i in the solution $s*$ are fixed to the sites j. The search will focus only on the other decision variables of the solution $s*$.

The intensification of the search in a given region of the search space is not always useful. The effectiveness of the intensification depends on the landscape structure of the target optimization problem. For instance, if the landscape is composed of many basins of attraction and a simple TS without intensification component is effective to search in each basin of attraction, intensifying the search in each basin is useless.

2.5.3 Long-Term Memory

As mentioned many times, S-metaheuristics are more powerful search methods in terms of intensification. Long-term memory has been introduced in tabu search to encourage the diversification of the search. The role of the long-term memory is to force the search in nonexplored regions of the search space.

The main representation used for the long-term memory is the *frequency memory*. As in the recency memory, the components associated with a solution have first to be defined. The frequency memory will memorize for each component the number of times the component is present in all visited solutions.

Example 2.31 Frequency memory in solving the GAP. The frequency memory may be represented by a symmetric matrix F where an element f_{ij} represents the number of times the object i has been assigned to the site j from the beginning of the tabu search. The diversification may consist in starting the search from a new solution $s*$. The elements i of the initial solution $s*$ are assigned to the sites j that are associated with smaller values of f_{ij}.

The diversification process can be applied periodically or after a given number of iterations without improvement. Three popular diversification strategies may be applied [306]:

2.5.3.1 **Restart diversification:** This strategy consists in introducing in the current or best solution the least visited components. Then a new search is restarted from this new solution.

2.5.3.2 **Continuous diversification:** This strategy introduces during a search a bias to encourage diversification. For example, the objective function can integrate the frequency occurrence of solution components in the evaluation of current solutions. Frequently applied moves or visited solutions are penalized.

2.5.3.3 **Strategic oscillation:** Introduced by Glover in 1989, strategic oscillation allows to consider (and penalize) intermediate solutions that are infeasible. This strategy will guide the search toward infeasible solutions and then come back to a feasible solution.

Example 2.32 Tabu search for the TSP problem. Designing a TS for the TSP problem needs the definition of the tabu list (short-term memory), the medium- and long-term memories, and the aspiration criterion:

- The short-term memory maintains a list of t edges and prevents them from being selected for consideration of moves for a number of iterations l. After the given number of iterations (tabu tenure), these edges are released. One can use a 2D Boolean matrix to decide if an edge is tabu or not.
- The medium- and long-term memory maintains a list of t edges that have been considered in the last k best (worst) solutions. The process encourages (or discourages) their selection in future solutions using their frequency of

appearance in the set of elite solutions and the quality of solutions they have appeared.

- The usual aspiration criterion that accepts the tabu moves that generate better solutions than the best found one.

As for the intensification, the diversification of the search is not always useful. It depends on the landscape structure of the target optimization problem. For instance, if the landscape is a "massif central" where all good solutions are localized in the same region of the search space within a small distance, diversifying the search to other regions of the search space is useless. The search time assigned to the diversification and the intensification components of TS must be carefully tuned depending on the characteristics of the landscape structure associated with the problem.

TS has been successfully applied to many optimization problems. Compared to local search and simulated annealing, various search components of TS are problem specific and must be defined. The search space for TS design is much larger than for local search and simulated annealing. The degree of freedom in designing the different ingredients of TS is important. The representation associated with the tabu list, the medium-term memory, and the long-term memory must be designed according to the characteristics of the optimization problem at hand. This is not a straightforward task for some optimization problems. Moreover, TS may be very sensitive to some parameters such as the size of the tabu list.

2.6 ITERATED LOCAL SEARCH

The quality of the local optima obtained by a local search method depends on the initial solution. As we can generate local optima with high variability, iterated local search[16] may be used to improve the quality of successive local optima. This kind of strategy has been applied first in Ref. [531] and then generalized in Refs [518,726].

In *multistart local search*, the initial solution is always chosen randomly and then is unrelated to the generated local optima. ILS improves the classical multistart local search by perturbing the local optima and reconsidering them as initial solutions.

Example 2.33 Multistart local search fails for the graph bisection problem. It has been shown that many (i.e., several thousand) random initial solutions are necessary to afford stable solution quality for the graph bisection problem for instances of size $n = 500$ [420]. The number of restarts grows rapidly with the size of instances n (number of nodes) and becomes unreasonable for instances of size $n = 100{,}000$, which arises in real-life problems such as VLSI design problems.

The central limit phenomenon in the landscape of an optimization problem implies that when the size of the problem becomes very large, local optima obtained using

[16]Also known as iterated descent, large-step Markov chains, and chained local optimization.

different random initial solutions are more or less similar in terms of quality [60]. Hence, simple multistart generally fails for very large problem instances [684].

ILS is based on a simple principle that has been used in many specific heuristics such as the iterated Lin–Kernigham heuristic for the traveling salesman problem [418] and the adaptive tabu search for the quadratic assignment problem [751]. First, a local search is applied to an initial solution. Then, at each iteration, a *perturbation* of the obtained local optima is carried out. Finally, a local search is applied to the perturbed solution. The generated solution is accepted as the new current solution under some conditions. This process iterates until a given stopping criterion. Algorithm 2.10 describes the ILS algorithm.

Algorithm 2.10 Template of the iterated local search algorithm.

s_* = local search(s_0) ; /* Apply a given local search algorithm */
Repeat
  s' = Perturb (s_*, search history) ; /* Perturb the obtained local optima */
  s'_* = Local search (s') ; /* Apply local search on the perturbed solution */
  s_* = Accept (s_*, s'_*, search memory) ; /* Accepting criteria */
Until Stopping criteria
Output: Best solution found.

Three basic elements compose an ILS (Fig. 2.28):

- **Local search:** Any S-metaheuristic (deterministic or stochastic) can be used in the ILS framework such as a simple descent algorithm, a tabu search, or simulated annealing. The search procedure is treated as a black box (Fig. 2.29). In the

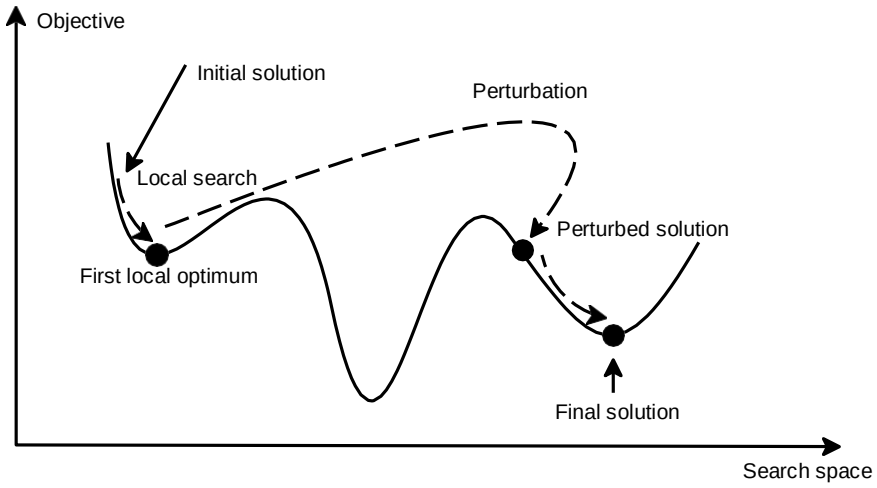


FIGURE 2.28 The principle of the iterated local search algorithm.

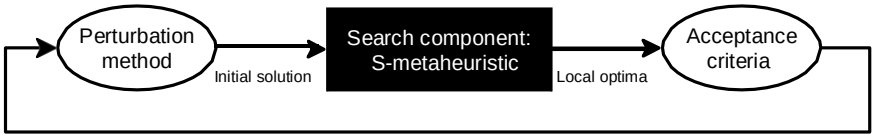


FIGURE 2.29 The search component is seen as a black box for the ILS algorithm.

literature, P-metaheuristics are not considered as candidates in the search procedure as they manipulate populations. However, some population-based metaheuristics integrate the concept of perturbation of the (sub)population to encourage the search diversification.

- **Perturbation Method.** The perturbation operator may be seen as a large random move of the current solution. The perturbation method should keep some part of the solution and perturb strongly another part of the solution to move hopefully to another basin of attraction.
- **Acceptance criteria.** The acceptance criterion defines the conditions the new local optima must satisfy to replace the current solution.

Once the S-metaheuristic involved in the ILS framework is specified, the design of ILS will depend mainly on the used perturbation method and the acceptance criterion. Many different designs may be defined according to various choices for implementing the perturbation method and the acceptance criterion.

2.6.1 Perturbation Method

The first motivation of the ILS algorithm is based on the fact that the perturbation method must be more effective than a random restart approach, where an independent initial solution is regenerated randomly. This will happen for most real-life optimization problems that are represented by structured search landscapes. For those problems, a *biased perturbation* must be carried out. However, for some problems and instances, where the landscape is rugged and flat, random restart may be more efficient. A first exercise in designing an ILS algorithm to solve the problem at hand will be to compare the performance of the random restart and the implemented biased perturbation.

A good balance must be found for the biased perturbation. The length of a perturbation may be related to the neighborhood associated with the encoding or with the number of modified components. A too small perturbation may generate cycles in the search and no gain is obtained. The probability to explore other basins of attraction will be low. Moreover, applying a local search to a small perturbation is faster than for large perturbations. Too large a perturbation will erase the information about the search memory, and then the good properties of the local optima are skipped. The extreme of this strategy is the random restart approach. The more effective is the

S-metaheuristic the larger the values of the perturbation must be. The optimal length depends mainly on the landscape structure of the optimization problem and must not exceed the correlation length (see Section 2.2.2). Even for a specific problem, it will depend on the instance at hand.

The move operator used in the perturbation may be of different nature from the neighborhood relation used in the local search procedure.

Many biased perturbation methods can be designed according to the following criteria:

2.6.1.1 **Fixed or variable perturbations.** The length of the perturbations applied to a local optima may be defined as follows:

- **Static.** The length is fixed *a priori* before the beginning of the ILS search.
- **Dynamic.** The length of the perturbation is defined dynamically without taking into account the search memory.
- **Adaptive.** In this strategy, the length of the perturbation is adapted during the search according to some informations about the search memory. Indeed, the optimal length will depend on the input instance and its structure. More information about the characteristics of the landscape may be extracted during the search.

2.6.1.2 **Random or semideterministic perturbation.** The perturbation carried out on a solution may be a random one (memoryless) in which each move is generated randomly in the neighborhood. This leads to a Markovian chain. In semide- terministic perturbations, the move is biased according to the memory of the search. For instance, the intensification and the diversification tasks of the tabu search algorithm using the medium-term memory and the long-term memory can be applied. The first approach is more popular in the literature, whereas the second approach needs more advanced and complex search mechanisms to be implemented.

2.6.2 Acceptance Criteria

The role of the acceptance criterion combined with the perturbation method is to control the classical trade-off between the intensification and the diversification tasks. The first extreme solution in terms of intensification is to accept only improving solutions in terms of the objective function (strong selection). The extreme solution in terms of diversification is to accept any solution without any regard to its quality (weak selection). Many acceptance criteria that balance the two goals may be applied:

2.6.2.1 **Probabilistic acceptance criteria:** Many different probabilistic acceptance cri- teria can be found in the literature. For instance, the Boltzmann distribution of simulated annealing. In this case, a cooling schedule must be defined.

2.6.2.2 **Deterministic acceptance criteria:** Some deterministic acceptance criteria may be inspired from the threshold accepting algorithms and the related algorithms, such as the great deluge and the record-to-record algorithms.

2.7 VARIABLE NEIGHBORHOOD SEARCH

Variable neighborhood search has been recently proposed by P. Hansen and N. Mladenovic [560]. The basic idea of VNS is to successively explore a set of predefined neighborhoods to provide a better solution. It explores either at random or systematically a set of neighborhoods to get different local optima and to escape from local optima. VNS exploits the fact that using various neighborhoods in local search may generate different local optima and that the global optima is a local optima for a given neighborhood (Fig. 2.30). Indeed, different neighborhoods generate different landscapes [428].

2.7.1 Variable Neighborhood Descent

The VNS algorithm is based on the variable neighborhood descent, which is a deterministic version of VNS. VND uses successive neighborhoods in descent to a local optimum. First, one has to define a set of neighborhood structures N_l ($l = 1, \ldots, l_{\max}$). Let N_1 be the first neighborhood to use and x the initial solution. If an improvement of the solution x in its current neighborhood $N_l(x)$ is not possible, the neighborhood structure is changed from N_l to N_{l+1}. If an improvement of the current solution x is found, the neighborhood structure returns to the first one $N_1(x)$ to restart the search (Fig. 2.31). This strategy will be effective if the different neighborhoods used are complementary in the sense that a local optima for a neighborhood N_i will not be a local optima in the neighborhood N_j. Algorithm 2.11 shows the VND algorithm.

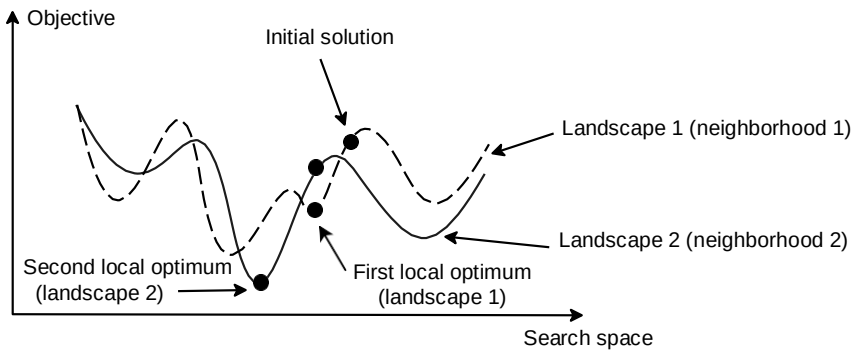


FIGURE 2.30 Variable neighborhood search using two neighborhoods. The first local optimum is obtained according to the neighborhood 1. According to the neighborhood 2, the second local optimum is obtained from the first local optimum.

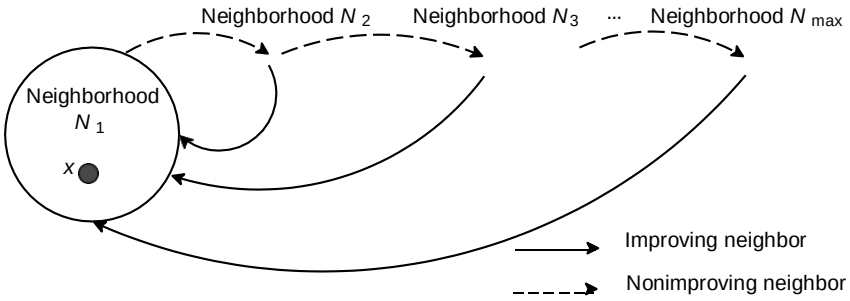


FIGURE 2.31 The principle of the variable neighborhood descent algorithm.

Algorithm 2.11 Template of the variable neighborhood descent algorithm.

Input: a set of neighborhood structures N_l for $l = 1, \ldots, l_{max}$.
$x = x_0$; /* Generate the initial solution */
$l = 1$;
While $l \leq l_{max}$ **Do**
Find the best neighbor x' of x in $N_l(x)$;
If $f(x') < f(x)$ **Then** $x = x'$; $l = 1$;
Otherwise $l = l + 1$;
Output: Best found solution.

The design of the VND algorithm is mainly related to the selection of neighborhoods and the order of their application. The complexity of the neighborhoods in terms of their exploration and evaluation must be taken into account (see Section 2.1.1). The larger are the neighborhoods, the more time consuming is the VND algorithm. Concerning the application order, the most popular strategy is to rank the neighborhoods following the increasing order of their complexity (e.g., the size of the neighborhoods $|N_l(x)|$).

2.7.2 General Variable Neighborhood Search

VNS is a stochastic algorithm in which, first, a set of neighborhood structures N_k ($k = 1, \ldots, n$) are defined. Then, each iteration of the algorithm is composed of three steps: shaking, local search, and move. At each iteration, an initial solution is shaked from the current neighborhood N_k. For instance, a solution x' is generated randomly in the current neighborhood $N_k(x)$. A local search procedure is applied to the solution x' to generate the solution x''. The current solution is replaced by the new local optima x'' if and only if a better solution has been found (i.e., $f(x'') < f(x)$). The same search procedure is thus restarted from the solution x'' in the first neighborhood N_1. If no better solution is found (i.e., $f(x'') \geq f(x)$), the algorithm moves to the next neighborhood N_{k+1}, randomly generates a new solution in this neighborhood, and attempts to improve it. Let us notice that cycling is possible (i.e., $x'' = x$). Algorithm 2.12 presents the template of the basic VNS algorithm.

Algorithm 2.12 Template of the basic variable neighborhood search algorithm.

Input: a set of neighborhood structures N_k for $k = 1, \ldots, k_{max}$ for shaking.
$x = x_0$; /* Generate the initial solution */
Repeat
  $k = 1$;
  Repeat
    Shaking: pick a random solution x' from the k^{th} neighborhood $N_k(x)$ of x ;
    $x'' =$local search(x') ;
    If $f(x'') < f(x)$ **Then**
      $x = x''$;
      Continue to search with N_1 ; $k = 1$;
    Otherwise k=k+1 ;
  Until $k = k_{max}$
Until Stopping criteria
Output: Best found solution.

A more general VNS algorithm where the simple local search procedure is replaced by the VND algorithm can be found in Algorithm 2.13.

Algorithm 2.13 Template of the general variable neighborhood search algorithm.

Input: a set of neighborhood structures N_k for $k = 1, \ldots, k_{max}$ for shaking.
  a set of neighborhood structures N_l for $k = 1, \ldots, l_{max}$ for local search.
$x = x_0$; /* Generate the initial solution */
Repeat
  For k=1 **To** k_{max} **Do**
  Shaking: pick a random solution x' from the k^{th} neighborhood $N_k(x)$ of x ;
  Local search by VND ;
  For l=1 **To** l_{max} **Do**
    Find the best neighbor x'' of x in $N_l(x)$;
    If $f(x'') < f(x)$ **Then** $x = x''$; l=1 ;
    Otherwise l=l+1 ;
  Move or not:
  If local optimum is better than x **Then**
    $x = x''$;
    Continue to search with N_1 ($k = 1$) ;
  Otherwise k=k+1 ;
Until Stopping criteria
Output: Best found solution.

In addition to the design of a simple local search (see Section 2.3) or a VND algorithm, the design of the VNS algorithm is mainly related to the selection of neighborhoods for the shaking phase. Usually, nested neighborhoods are used, where each neighborhood $N_k(x)$ contains the previous one $N_{k-1}(x)$:

$$N_1(x) \subset N_2(x) \subset \cdots \subset N_k(x), \forall x \in S$$

A compromise must be found between intensification of the search and its diversification through the distribution of work between the local search phase and the shaking phase. An increased work in the local search phase will generate better local optima (more intensification), whereas an increased work in the shaking phase will lead to potentially better regions of the search space (more diversification).

Example 2.34 Nested neighborhoods for the shaking phase. For many combinatorial optimization problems, there is a natural definition of nested neighborhoods. For instance, for routing problems such as the traveling salesman problem and the vehicle routing problem, the k-opt operator can be used for different values of k ($k = 2, 3, \ldots$).

As in local search, VNS requires a small number of parameters. For the shaking phase, the single parameter is the number of neighborhood structures k_{max}. If large values of k_{max} are used (i.e., very large neighborhoods are considered), VNS will be similar to a multistart local search. For small values of k_{max}, VNS will degenerate to a simple local search algorithm.

Example 2.35 Many neighborhoods in a real-life application. This example illustrates the use of many neighborhoods ($k = 9$) to solve a scheduling problem [26]. Many oil wells in onshore fields rely on artificial lift methods. Maintenance services such as cleaning, stimulation, and others are essential to these wells. They are performed by workover rigs. Workover rigs are available in a limited number with respect to the number of wells demanding service. The decision which workover rig should be sent to perform some maintenance services is based on factors such as the well production, the current location of the workover rig in relation to the demanding well, and the type of service to be performed. The problem of scheduling workover rigs consists in finding the best schedule S^* for the available m workover rigs, so as to minimize the production loss associated with the wells awaiting service. A schedule is represented by an ordered set of wells serviced by workover rigs. A variable neighborhood search metaheuristic has been proposed for this problem using the following neighborhoods in the shaking procedure:

1. Swapping of routes (SS) where the wells associated with two workover rigs are exchanged.
2. Swapping of wells from the same workover rig (SWSW) where two wells serviced by the same workover rig are exchanged.
3. Swapping of wells from different workover rigs (SWDW) where two wells affected by two different workover rigs are exchanged.
4. Add/drop (AD) where a well affected by a workover rig is reassigned to any position of the schedule of another workover rig.
5. Two applications of the SWSW transformation.
6. Two applications of the SWDW transformation.
7. Three applications of the SWDW transformation.

8. Successive application of two AD transformations.
9. Successive application of three AD transformations.

Many extensions of the GVNS algorithm have been proposed such as VNDS and SVNS [366]. These extensions are not related to the main concept of the VNS algorithm, which is the neighborhood change. In VNDS, the optimization problem is decomposed into subproblems, whereas in SVNS the algorithm addresses how to get out of large valleys (basin of attraction).

2.8 GUIDED LOCAL SEARCH

Guided local search is a deterministic S-metaheuristic that has been mainly applied to combinatorial optimization problems. Its adaptation to continuous optimization problems is straightforward given that GLS sits on top of a local search algorithm [805]. The basic principle of GLS is the dynamic changing of the objective function according to the already generated local optima [808]. The features of the obtained local optima are used to transform the objective function (Fig. 2.32). It allows the modification of the landscape structure to be explored by an S-metaheuristic to escape from the obtained local optima.

In GLS, a set of m features ft_i ($i = 1, \ldots, m$) of a solution are defined. A solution feature defines a given characteristic of a solution regarding the optimization problem to solve. A cost c_i is associated to each feature. When trapped by a local optima, the algorithm will penalize solutions according to some selected features. To each feature i is associated a penalty p_i that represents the importance of the

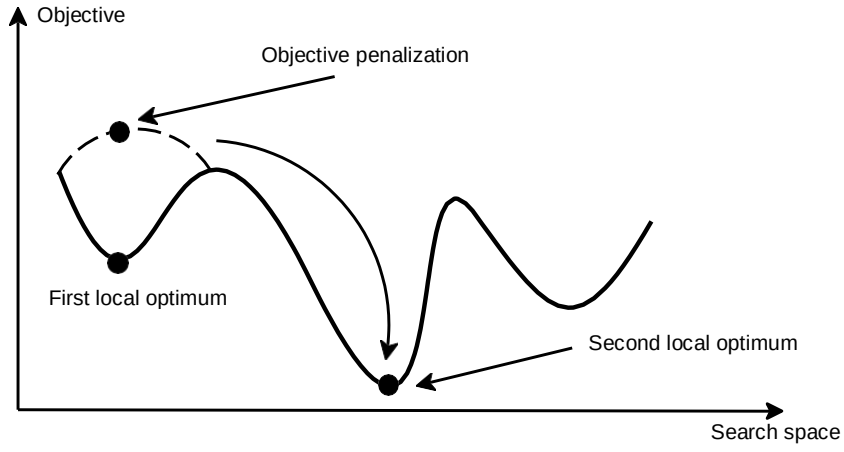


FIGURE 2.32 Guided local search penalizing the solutions found to escape from the local optimum. The objective function is increased according to the features of the obtained solutions.

feature. The objective function f associated with a solution s is then penalized as follows:

$$f'(s) = f(s) + \lambda \sum_{i=1}^{m} p_i I_i(s)$$

where λ represents the weights associated with different penalties and, $I_i(s)$ an indicator function that determines whether the feature ft_i is present in the solution s:

$$I_i(s) = \begin{cases} 1 & \text{if the feature } ft_i \in s \\ 0 & \text{otherwise} \end{cases}$$

All penalties p_i are initialized to 0.

Example 2.36 Identification of suitable features. The application of guided local search to a given optimization problem requires the identification of suitable features of the solutions. The choice of features depends mainly on the model associated with the optimization problem:

- **Routing problems:** In routing problems such as the traveling salesman problem and the vehicle routing problem, a feature may be associated with the presence of an edge (a, b) in the solution. The cost corresponds to the distance (or travel time) associated with the edge.
- **Assignment problems:** In assignment problems, such as the generalized assignment problem and location facility problems, a solution represents an assignment of a given number of objects to a set of locations. A feature may be represented by the pair (i, k), where i represents an object and k represents a specific location.

$$I_{ik}(s) = \begin{cases} 1 & \text{if } s(i) = k \\ 0 & \text{otherwise} \end{cases}$$

  The cost will be handled by the cost c_{ik} in assigning the object i to the location k.
- **Satisfiability problems:** In satisfiability problems, such as the MAX-SAT, the objective is generally related to the number of violated constraints. A feature may be associated with each constraint. The cost of a feature will be related to the constraint violation.

The main question that arises is the way the features are selected and penalized. The goal is to penalize the features that are present in the obtained local optimum or unfavorable features. Given a local optima s^*, a utility u_i is then associated with each

feature i as follows:

$$u_i(s^*) = I_i(s^*)\frac{c_i}{1+p_i}$$

where c_i represents the cost of the feature i. For instance, if a given feature i is not present in the local optimum s^*, the utility of penalizing feature i is 0. Otherwise, the utility will be proportional to the cost c_i and inversely proportional to the penalty p_i. Then, the feature associated with the highest utility will be penalized. Its penalty is incremented by 1, and the scaling of the penalty is normalized by λ.

On one hand, GLS will intensify the search of promising regions defined by lower costs of the features. On the other hand, the GLS algorithm will diversify the search by penalizing the features of the generated local optima to avoid them. The common use of GLS approach is within a S-metaheuristic (local search) framework. However, an extension to P-metaheuristics can be easily envisaged [839]. In addition to penalizing the objective function, one may take into account the penalties to bias the recombination operators of P-metaheuristics (crossover, mutation, etc.). For instance, the encoded features that contribute more to the penalties will be more likely to be changed in the reproduction operators. For efficiency issues, the incremental evaluation of the objective function in local search (see Section 2.1.4) may be extended easily for the augmented objective functions in GLS. Algorithm 2.14 describes the template of the GLS algorithm.

Algorithm 2.14 Template of the guided local search algorithm.

Input: S-metaheuristic *LS*, λ, Features I, Costs c.
$s = s_0$ /* Generation of the initial solution */
$p_i = 0$ /* Penalties initialization */
Repeat
  Apply a S-metaheuristic *LS* ; /* Let s^* the final solution obtained */
  For each feature i of s^* **Do**
    $u_i = \frac{c_i}{1+p_i}$; /* Compute its utility */
  $u_j = max_{i=1,..m}(u_i)$; /* Compute the maximum utilities */
  $p_j = p_j + 1$; /* Change the objective function by penalizing the feature j */
Until Stopping criteria /* e.g. max number of iterations or time limit */
Output: Best solution found.

The input design parameters of the GLS algorithm are represented by the choice of the features (indicator function), their costs, and the λ parameter. The features and the costs are related to the optimization problem at hand. The literature suggests that the performance of the algorithm is not very sensitive to the value of λ [556]. Large values of λ will encourage the diversification, while small values of λ will intensify the search around the local optima. A common strategy to initialize λ consists in dividing the value of the objective function of the local optimum by the average number of features present in the local optimum or near the average change of the objective

function in the neighborhood. The more the S-metaheuristic used is effective, the less is the value of the λ parameter. The penalty needed to escape from local optima must decrease with the increase of the effectiveness of the S-metaheuristic used within GLS.

Example 2.37 Guided local search for the symmetric TSP. A solution of the TSP is encoded by a permutation s. The original objective function consists in minimizing the total distance:

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

where n represents the number of cities. As mentioned above, the features of the TSP may be represented by the edges composing a tour. Then, the indication of the presence of a feature may be represented as follows:

$$I_{(i,j)}(s) = \begin{cases} 1 \text{ if edge } (i, j) \in s \\ 0 \text{ otherwise} \end{cases}$$

The cost associated with an edge (i, j) is the distance d_{ij}. Penalties are represented by a matrix $P_{n,n}$, where p_{ij} represents the penalty of the edge (i, j). All the elements of the matrix P are initialized to 0. The augmented objective function of the problem is given by

$$f'(s) = \sum_{i=1}^{n} d_{i,s(i)} + \lambda p_{i,s(i)}$$

The utility function is defined as follows:

$$u(s^*, (i,j)) = I_{(i,j)}(s)\frac{d_{ij}}{1 + p_{ij}}$$

2.9 OTHER SINGLE-SOLUTION BASED METAHEURISTICS

Some existing S-metaheuristics use other strategies to escape from local optima. Search space smoothing and noisy methods are based on the transformation of the landscape of the problem by changing the input data associated with the problem altering the objective function.

2.9.1 Smoothing Methods

Search space smoothing consists in modifying the landscape of the target optimization problem [326,343]. The smoothing of the landscape associated with the problem

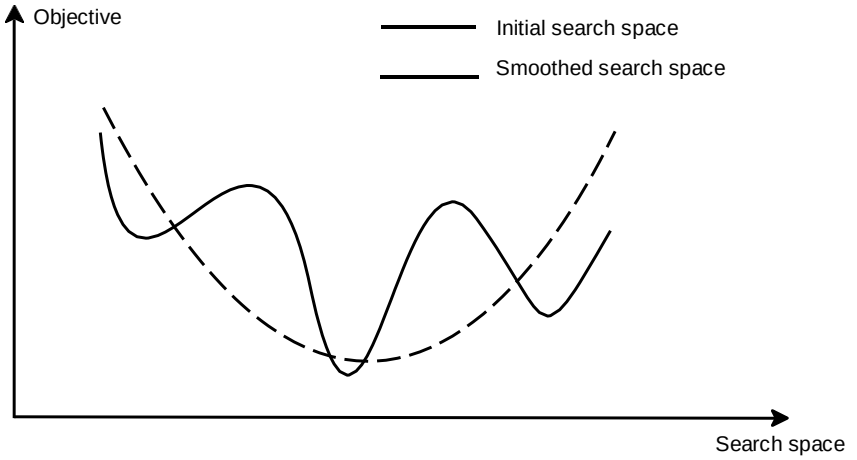


FIGURE 2.33 Search space smoothing that results in an easiest problem to solve. The smoothed landscape has less local optima than the original one.

reduces the number of local optima and the depth of the basins of attraction without changing the location region of the global optimum of the original optimization problem (Fig. 2.33). The search space associated with the landscape remains unchanged, and only the objective function is modified. Once the landscape is smoothed by "hiding" some local optima, any S-metaheuristic (or even a P-metaheuristic) can be used in conjunction with the smoothing technique.

The main idea of the smoothing approach is the following: given a problem instance in a parameter space, the approach will transform the problem into a sequence of successive problem instances with different associated landscapes. Initially, the most simplified smoothed instance of the problem is solved. A local search is then applied. The probability to be trapped by a local optima is minimized. In the ideal case, there is only one local optimum that corresponds to the global optimum (Fig. 2.34). The less the number of local optima, the more efficient a S-metaheuristic. Then, a more complicated problem instance with a rougher landscape is generated. It takes the solution of the previously solved problem as an initial solution and further improves that solution. The solutions of smoothed landscapes are used to guide the search in more rugged landscapes. Any S-metaheuristic can be used in conjunction with the smoothing operation. The last step of the approach consists in solving the original problem.

If the global minimum in the smoothed landscape is used as the initial solution in the original space (see Fig. 2.34), the probability to find the global optimum in the original space may increase considerably.

The main design question concerns the smoothing operation. There are many strategies to smooth a landscape. The smoothing factor α is used to characterize the strength of a smoothing operation. Using different levels of strength will generate various degrees of smoothness. When $\alpha = 1$, there is no smoothing operation, and the landscape is the same as the original one. A smoothing operation is carried out if $\alpha > 1$. The larger the smoothing factor ($\alpha >> 1$), the stronger a smoothing operation and more

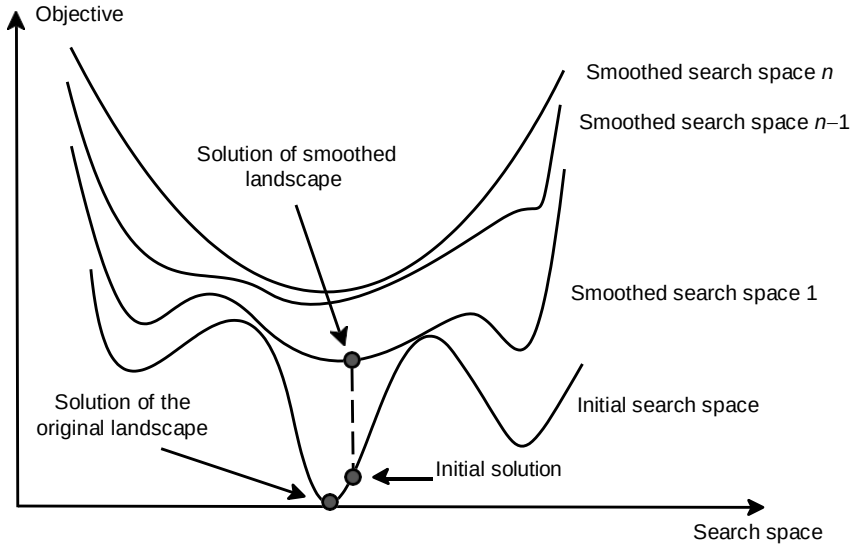


FIGURE 2.34 Successive smoothing of the landscape. The solution found at the step i will guide the search at the iteration $i + 1$ in a more rugged landscape.

flat a landscape. Algorithm 2.15 illustrates the principles of the smoothing algorithm. The original idea of the algorithm relies on the reduced complexity of solving smoothing instances of the original problem and the effectiveness of using intermediate local optima solutions to guide the search toward increasingly complex instances.

Algorithm 2.15 Template of the smoothing algorithm.

Input: S-metaheuristic *LS*, α_0, Instance *I*.
$s = s_0$; /* Generation of the initial solution */
$\alpha = \alpha_0$; /* Initialization of the smoothing factor */
Repeat
  $I = I(\alpha)$; /* Smoothing operation of the instance I */
  $s = LS(s, I)$; /* Search using the instance I and the initial solution s */
  $\alpha = g(\alpha)$; /* Reduce the smoothing factor, e.g. $\alpha = \alpha - 1$ */
Until $\alpha < 1$ /* Original problem */
Output: Best solution found.

Example 2.38 Smoothing operation for the TSP. The smoothing strategy has been applied successfully to many discrete optimization problems [326]. In the case of the TSP, the smoothing operation is based on the fact that a trivial case for the TSP is the one where all the distances between cities are equal: $d_{ij} = \bar{d}, \forall i, j$, where

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

represents the average distance over all the edges. In this case, any tour represents a global optimum solution, and the landscape is flat.

The strength of a smoothing may be represented by the following equation:

$$d_{ij}(\alpha) = \begin{cases} \bar{d} + (d_{ij} - \bar{d})^{\alpha} & \text{if } d_{ij} \geq \bar{d} \\ \bar{d} - (\bar{d} - d_{ij})^{\alpha} & \text{if } d_{ij} < \bar{d} \end{cases}$$

where $\alpha \geq 1$ represents the strength of the smoothing. At each step, the strength α is decreased from a large number (e.g., 10) to 1. Thus, a sequence of more rugged landscapes are generated and searched. The two extreme cases of the sequence are

2.9.1.1 $d_{ij}(\alpha) \longrightarrow \bar{d}$ when $\alpha >> 1$. This flat landscape represents the first instance of the sequence.

2.9.1.2 $d_{ij}(\alpha) = d_{ij}$ when $\alpha = 1$. The last landscape of the sequence represents the original problem.

The main parameters of the smoothing algorithm are the appropriate choice of the initial value of the smoothing factor α and its controlling strategy. The larger the initial value of the smoothing factor α_0, the more time consuming the algorithm.

Some meta-modeling strategies (see Section 1.4.2.5) may also be used to approximate and smooth the objective function such as Gaussian kernels [833] and global polynomial models [505].

2.9.2 Noisy Method

The noisy method is another S-metaheuristic algorithm that is based on the landscape perturbation of the problem to solve [124]. Instead of taking the original data into account directly, the NM considers that they are the outcomes of a series of fluctuating data converging toward the original ones. Some random noise is added to the objective function f. At each iteration of the search, the noise is reduced. For instance, the noise is initially randomly chosen into an interval $[-r, +r]$. The range of the interval r decreases during the search process until a value of 0. Different ways may be used to decrease the noise rate r.

The noising method can be considered as a succession of local searches applied to perturbed data. The data parameters are replaced by "noised" data. For instance, if the input data are completely represented by a graph $G = (V, E)$ and the considered parameters are the weights w_{ij} of the edges (i, j), the noising method will replace those values as follows:

$$w_{ij}^{\text{noised}} = w_{ij} + \rho_{ij}$$

where ρ_{ij} is a noise. Then, a local search procedure is applied to the noised graph. Once a local optimum is obtained, the noise rate r is decreased by decreasing the standard deviation or the absolute value of the mean. The search terminates when the noise is low enough. The extreme stopping creterion is when the mean and the standard

deviation of the noise are equal to 0. Contrary to traditional S-metaheuristics, an improving move may be rejected in the noising algorithm. As in simulated annealing, for instance, a nonimproving move may be accepted.

A more general noising method considers the original data, but the noise is considered in computing the incremental evaluation of the neighbor s' obtained by a transformation carried on the current solution s. Instead of considering the original variation $O(s, s')$, a noised variation $O_{\text{noised}}(s, s')$ is considered by adding a noise to $O(s, s')$: $O_{\text{noised}}(s, s') = O(s, s') + \rho_k$, where ρ_k is a noise changing at each iteration k and depending on the noise rate. This second noising method is a more general one in the sense that the first one may be reduced to this method by summing up the impact of the noises added to the data. The main difference between the two strategies is that randomness does not occur in the same way. In the first noising method, after fixing the noises affecting the original data, the local search process can be deterministic, while in the second noising method, randomness takes place at each iteration of the local search. Algorithm 2.16 shows the general template of the noising algorithm. Noise is added to the data or the variations of the objective function in an absolute or relative way.

Algorithm 2.16 Template of the noising method.

Input: r.
$s = s_0$; /* Generation of the initial solution */
$r = r_{max}$; /* Initialization of the noising factor */
Repeat
  $O_{noised}(s, s') = O(s, s') + r$; /* Noising operation of f */
  $s = LS(s, r)$; /* Search using the noising factor r */
  $r = g(r)$; /* Reduce the noising factor */
Until $r = r_{min}$ /* Original problem */
Output: Best solution found.

The are different ways to add noise to data. The choice of the noising method scheme may have a great impact on the performance of the noising algorithm. A noise is a value taken by a certain random variable following a given probability distribution (e.g., a uniform law or a Gaussian one). The mean and the standard deviation of this probability distribution converge toward 0, and the standard deviation decreases during the search process. The added noise is progressively reduced and becomes nonexistent. The noise rate r introduced characterizes the "strength" of the noise for a chosen probability distribution. Then the decreasing of the noise is obtained by decreasing the noise rate r.

For other S-metaheuristics, some parameters must be tuned for the noising method:

2.9.2.1 **Noise rate:** The extremal values of the noise rate $r_{\max}$ and $r_{\min}$ depend on the data. It is always possible to choose $r_{\min} = 0$.

2.9.2.2 **Decreasing noise rate:** The geometrical decreasing function is usually used ($r = r\alpha$ with $\alpha \in]0, 1[$). More advanced noising methods may be applied [125] (Fig. 2.35).

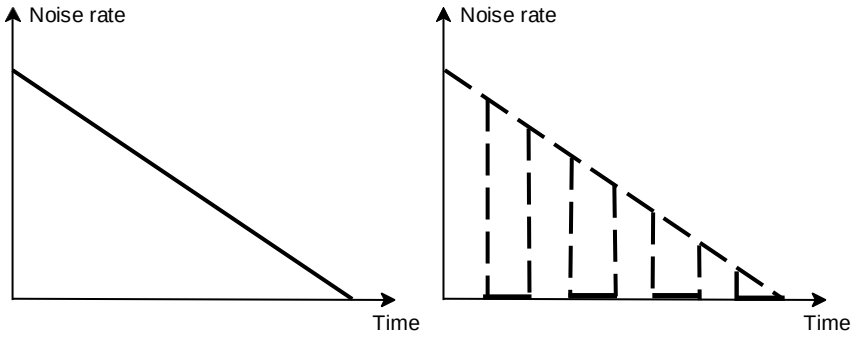


FIGURE 2.35 Different noising methods. (a) The noise rate is decreased linearly down to 0. (b) The noise decreases linearly, but "unperturbed" local searches are applied from time to time.

- **Probability distribution:** In adding the noise, it is possible to apply different probability distributions. In most of the cases, uniform distribution is applied. The noise added could be nonuniform and even close to a Gaussian distribution. The efficiency of a given probability distribution depends on the problem to solve. There is no absolute best probability distribution for all problems.

Example 2.39 Noising algorithm for graph partitioning. Given a complete nonoriented graph $G = (V, E)$ weighted by positive or negative integers, find a partition of V into subsets of which the number is not fixed. The objective is to minimize the sum of the weights of edges with their two extremities in the same subset. A noising method where the noise is randomly added to the weights of the edges, with a uniform distribution into $[-r, +r]$ and a arithmetical decrease of the noise rate r, may be applied [124]. Noised local searches are alternated with unnoised ones. There is a periodical restart of the current solution. The edges have weights in the range $[-w_{\max}, w_{\max}]$. The neighborhood consists in transferring a vertex from the subset to which it currently belongs to another one, which can be empty if we want to create a new subset (if a solution is coded by associating with each vertex the number of the subset to which it belongs, then this neighborhood is a substitution). The examination of the neighborhood for the local search (the noised ones as well as the unnoised ones) is systematic. The best tuning of the parameters depends on the studied graphs: the maximum rate noise $r_{\max}$ is chosen between $0.8w_{\max}$ and $0.95w_{\max}$ and the minimum rate noise $r_{\min}$ is chosen between $0.15w_{\max}$ and $0.5w_{\max}$. For instance, using $0.8w_{\max}$ for $r_{\max}$ and $0.4w_{\max}$ for $r_{\min}$, good results have been obtained. Of course, the total number of iterations depends on the search time that the user wishes to spend to solve the problem. For the frequency of the restart, if α denotes the number of pairs (noised local search, unnoised local search) applied between two restarts and β the number of restarts (so the total number of noised local searches and unnoised ones is equal to $\alpha\beta$), then a good tuning of α and β was to choose α between β and 2β.

Example 2.40 Different noising methods for a clique partitioning problem. Let us consider the clique partitioning problem of Example 2.22. Different noising methods may be applied to the weights of the input graph or on incremental evaluation of the objective function [125]:

- **Uniform noise:** For a node i and for each current class C, $W_{\text{noised}}(i, C)$ is given by the sum of $W(i, C)$ and a noise $\rho \times$ rate where ρ is randomly picked in the interval $[-1, 1]$ with a uniform law.

$$W_{\text{noised}}(i, C) = W(i, C) + \rho \times \text{rate}$$

- **Logarithmic noise:** For a node i and for each current class C, the added noise will be

$$W_{\text{noised}}(i, C) = W(i, C) + \log(\rho) \times \text{rate}$$

  where ρ is picked with a uniform random law in the interval $]0, 1]$.
- **Relative uniform noise:** Here, the noise is added to the weights of the input graph. Then, for a vertex i and for each current class C, we have

$$W_{\text{noised}}(i, C) = \sum_{j \in C, i \neq j} w(i, j) \times (1 + \rho_j \times \text{rate})$$

  where ρ_j is chosen with a uniform random law in the interval $[-1, 1]$.
- **Forgotten vertices noise:** In this unusual noising method, some selected nodes of the graph will not participate in the cyclic exploration of the neighborhood of a solution s. At each neighborhood exploration, a given percentage (rate) of vertices of the input graph G are removed from the exploration. The best perturbed class for a forgotten node is not searched. For a nonforgotten node, the following value is minimized to find the best class for a nonforgotten node:

$$W_{\text{noised}}(i, C) = \sum_{j \in C, i \neq j, j \textbf{ nonforgotten}} w(i, j)$$

- **Forgotten edge noise:** Instead of deleting nodes, this noised method forgets some edges of the graph. The rate ρ denotes the probability that the edge (i, j) is forgotten (random uniform law). For each node i and for each current class C of the current solution,

$$W_{\text{noised}}(i, C) = \sum_{j \in C, i \neq j, (i,j) \textbf{ nonforgotten}} w(i, j)$$

2.9.3 GRASP

The GRASP metaheuristic is an iterative greedy heuristic to solve combinatorial optimization problems. It was introduced in 1989 [255]. Each iteration of the GRASP algorithm contains two steps: construction and local search [256]. In the construction step, a feasible solution is built using a randomized greedy algorithm, while in the next step a local search heuristic is applied from the constructed solution. A similar idea, known as the *semigreedy heuristic*, was presented in 1987, where a multistart greedy approach is proposed but without the use of local search [367]. The greedy algorithm must be randomized to be able to generate various solutions. Otherwise, the local search procedure can be applied only once. This schema is repeated until a given number of iterations and the best found solution are kept as the final result. We notice that the iterations are completely independent, and so there is no search memory. This approach is efficient if the constructive heuristic samples different promising regions of the search space that makes the different local searches generating different local optima of "good" quality. Algorithm 2.17 resumes the template for the GRASP algorithm. The *seed* is used as the initial seed for the pseudorandom number generator.

Algorithm 2.17 Template of the greedy randomized adaptive search procedure.

Input: Number of iterations.
Repeat
  s = Random-Greedy(seed) ; /* apply a randomized greedy heuristic */
  s' = *Local − Search*(s) ; /* apply a local search algorithm to the solution */
Until Stopping criteria /* e.g. a given number of iterations */
Output: Best solution found.

The main design questions for GRASP are the greedy construction and the local search procedures:

2.9.3.1 **Greedy construction:** In the constructive heuristic, as mentioned in Sec- tion 1.3.4, at each iteration the elements that can be included in the partial solution are ordered in the list (decreasing values) using the local heuristic. From this list, a subset is generated that represents the *restricted candidate list*. The RCL list is the key component of the GRASP metaheuristic. It represents the probabilistic aspect of the metaheuristic.

The restriction criteria may depend on

- **Cardinality-based criteria:** In this case, the RCL list is made of the p best elements in terms of the incremental cost, where the parameter p represents the maximum number of elements in the list.
- **Value-based criteria:** It is the most commonly used strategy [649]. It consists in selecting the solutions that are better than a given threshold value. Let c^{max} (resp. c^{min}) be the maximum (resp. the minimum) values for the set of elements

using the local heuristic. The list RCL may be constructed as follows:

$$\{e_i/c_i \leq c^{\min} + \alpha(c^{\max} - c^{\min})\}$$

where α is a user-defined parameter.

At each iteration, a random element is picked from the list RCL. Once an element is incorporated in the partial solution, the RCL list is updated. To update the RCL list, the incremental costs $c'(e)$ of the elements e composing the RCL list must be reevaluated. Algorithm 2.18 shows the template of the randomized part of the GRASP metaheuristic.

Algorithm 2.18 Template of the greedy randomized algorithm.

```
s = {} ; /* Initial solution (null) */
Evaluate the incremental costs of all candidate elements ;
Repeat
    Build the restricted candidate list RCL ;
    /* select a random element from the list RCL */
    e_i = Random-Selection(RCL) ;
    If s ∪ e_i ∈ F Then /* Test the feasibility of the solution */
    s = s ∪ e_i ;
    Reevaluate the incremental costs of candidate elements ;
Until Complete solution found.
```

2.9.3.2 **Local search:** Since the solutions found by the construction procedure are not guaranteed to be local optima, it is beneficial to carry out a local search step in which the constructed solution is improved. Traditionally, a simple local search algorithm is applied. Nevertheless, any S-metaheuristic can be used: tabu search, simulated annealing, noisy method, and so on.

Example 2.41 GRASP for the capacitated minimum spanning tree problem. The capacitated minimum spanning tree problem is a key problem in network design such as in telecommunication and transportation applications. It consists in connecting a set of customer nodes to a root node through a minimum cost connected network subjected to a capacity constraint on all connections. Given a connected undirected graph $G = (V,E)$, where $V = \{v_1, \ldots, v_n\}$ represents the set of vertices and E is the set of edges. Each node (resp. edge) is weighted by nonnegative integers b_i (resp. c_{ij}). The weight c_{ij} represents the cost of the edge, whereas b_i represents the flow of the node. Given an integer Q denoting the capacity that must not be exceeded by any edge and a node $r \in V$ defining the sink node (concentrator or central node) to which all the flows go. The CMST consists of finding a minimum spanning tree T of G that minimizes the total cost of edges and satisfies the following constraint: the sum of the weights of vertices in each connected component of the graph induced in T by $V - \{r\}$ is less than or equal to Q; that is, for each subtree originated from the root r, the total flow does not exceed the link capacity C.

The greedy heuristic is based on the well-known saving heuristic EW of Esau and Williams [246]. This greedy heuristic is popular and widely used into metaheuristics to solve constrained spanning tree problems. Initially, every node is connected directly to the sink node. Hence, there are as many subtrees as the number of nodes. The iteration step consists of selecting the two nodes i and j belonging to two different subtrees and providing the largest saving and satisfying the capacity constraint. The saving is defined as

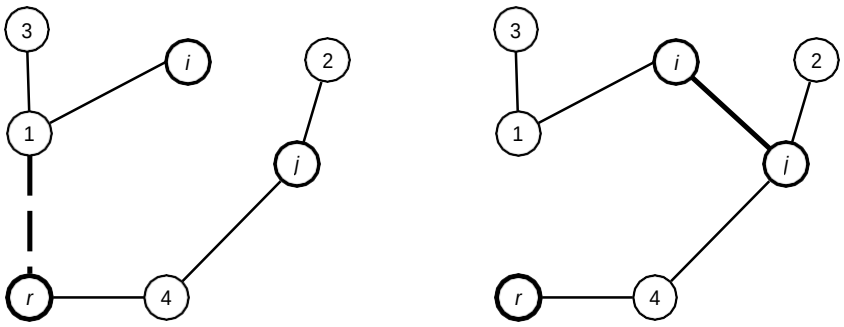
$$s_{ij} = (G_i + G_j) - (G_i + c_{ij}) = G_j - c_{ij}$$

where G_i and G_j are the costs of the links connecting the subtrees associated with vertices i and j to the sink node. Since the nodes of V_j have an edge (r,i) as a new link with connection cost G_i, the savings can be computed as follows:

$$s_{kl} = \begin{cases} s_{kl} - G_j + G_i & \text{for } k \in V \setminus (V_i \cup V_j \cup r) \text{ and } l \in V_j \\ 0 & \text{for } k, l \in V_j \text{ and } k \neq l \\ s_{kl} & \text{otherwise} \end{cases}$$

The algorithm stops when there is no more pair of nodes with positive saving or not exceeding the capacity constraint. The time complexity of the algorithm is $O(n^2 \log(n))$, where n is the number of nodes of the graph G ($n = |V|$). Figure 2.36 illustrates the application of the saving heuristic EW to a given graph.

Let C be the set of all candidate edges, that is, edges satisfying the capacity constraint and belonging to two different subtrees. The saving associated with candidate edges is used to randomize the heuristic. The RCL list associated with the randomized greedy



(a) Before merging nodes i and j **(b)** After merging nodes i and j

FIGURE 2.36 An iteration of the saving heuristic EW. (a) We have two subtrees of vertices $\{1, i, 3\}$ and $\{2, j, 4\}$. We suppose that s_{ij} is the largest saving so that the total flow of vertices $\{1, 2, 3, 4, i, j\}$ does not exceed the capacity Q. (b) Represents the new solution after merging the nodes i and j.

heuristic is defined as follows [190]:

$$\text{RCL} = \{(i, j) \in C / s_{ij} > s^{\min} + \alpha \cdot (s^{\max} - s^{\min})\}$$

where $s^{\min}$ (resp. $s^{\max}$) represents the minimum (resp. maximum) saving of the candidate edges (i, j) ($(i, j) \in C$). Then, any local search algorithm may be applied to the constructed solution.

The main parameter for GRASP is α parameter ($\alpha \in [0, 1]$). It defines the compromise between search exploitation (intensification using more greediness) and search exploration (diversification using more randomness). In fact, small values for α allow larger values for the average quality of the RCL list elements and then encourage exploitation. When $\alpha = 1$, the algorithm is a deterministic greedy algorithm. When $\alpha = 0$, the algorithm is a random greedy algorithm. Smaller values for α generate more elements in the list RCL and then encourage diversification. Many experimental results show a Gaussian normal distribution for the obtained solutions. Figure 2.37 shows the distribution of the constructed solutions as a function of α parameter [649]. The smaller is the α parameter, the larger is the variance between the solutions of the construction phase and worst is the average quality of solutions.

The performance of the GRASP metaheuristic is very sensitive to α parameter. Since this parameter is critical, many strategies may be applied for its initialization:

2.9.3.3 **Static:** In the static strategy, the value of α is initialized to a constant value before the search. The commonly used range is naturally closer to the greedy choice: [0.7, 0.9].

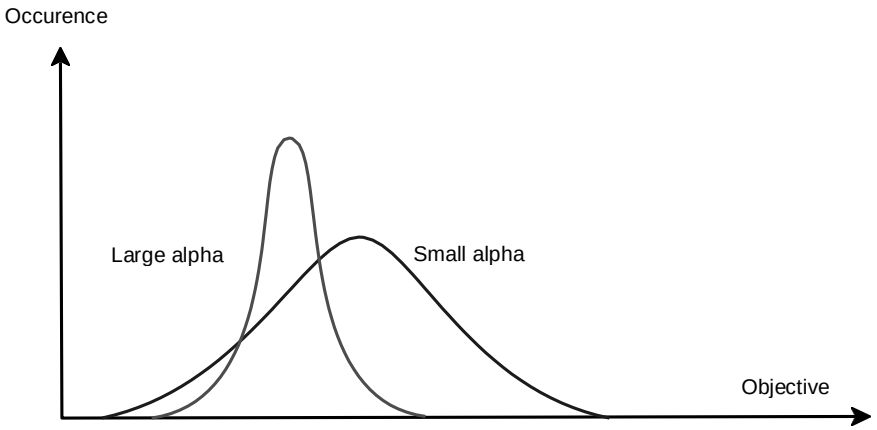


FIGURE 2.37 The distribution of the obtained local optima as a function of α parameter. We suppose a minimization problem in which small values of α tend to a purely random greedy search.

2.9.3.4 **Dynamic:** The dynamic strategy often initializes the α value to a random value at each iteration of the GRASP metaheuristic. For instance, a uniform distribution may be used in the range [0.5, 0.9], or a decreasing nonuniform distribution.

2.9.3.5 **Adaptive:** In this strategy, the value of α is self-tuned. The value is updated automatically during the search function of the memory of the search.

Example 2.42 Self-tuning in reactive GRASP. In reactive GRASP, the value of the parameter α is periodically updated according to the quality of the obtained solutions [623]. At each iteration, the parameter value is selected from a discrete set of possible values $W = \{\alpha_1, \ldots, \alpha_m\}$. The probability associated with each α_i is initialized to a same value $p_i = 1/m, i \in [1, \ldots, m]$. The adaptive initialization in the reactive GRASP consists in updating these probabilities according to the quality of solutions obtained for each α_i. Let z^* be the incumbent solution and A_i the average value of all solutions found using $\alpha = \alpha_i$. Then, the probability p_i for each value of α is updated as follows:

$$p_i = \frac{q_i}{\sum_{j=1}^{m} q_j}, \quad i \in [1, \ldots, m]$$

where $q_j = z^*/A_i$. Hence, larger values of p_i correspond to more suitable values for the parameter α_i.

In addition to the parameters associated with the randomized greedy heuristic, the GRASP metaheuristic will inherit the parameters of the embedded S-metaheuristic. The larger is the variance between the solutions of the construction phase, the larger is the variance of the obtained local optima. The larger is the variance between the initial solutions, the better is the best found solution and the more time consuming is the computation.

2.10 S-METAHEURISTIC IMPLEMENTATION UNDER ParadisEO

ParadisEO–MO (moving objects) is the module of the software framework ParadisEO dedicated to the design of single-solution based metaheuristics (S-metaheuristics). An important aspect in ParadisEO–MO is that the common search concepts of both metaheuristics and S-metaheuristics are factored. All search components are defined as templates (generic classes). ParadisEO uses the genericity concept of objects to make those search mechanisms adaptable. The user implements an S-metaheuristic by deriving the available templates that provide the functionality of the different search components: problem-specific templates (e.g., objective function, representation) and problem-independent templates (e.g., neighbor selection, cooling schedule, and stopping criteria).

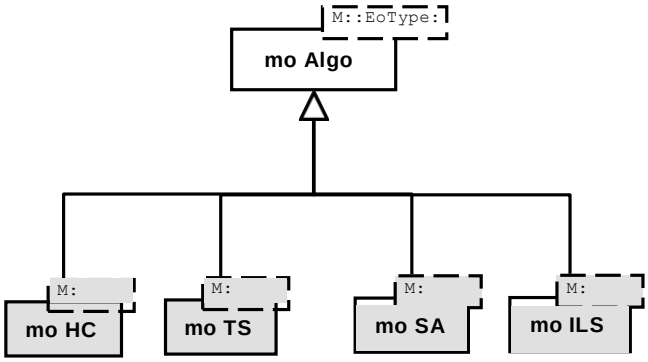


FIGURE 2.38 UML diagram of the `moAlgo` template representing an S-metaheuristic such as hill climbing, tabu search, simulated annealing, iterated local search.

In ParadisEO–MO, the `moAlgo` template represents an S-metaheuristic such as local search (hill climbing), tabu search, simulated annealing, and iterated local search (Fig. 2.38).

Figure 2.43 shows the common concepts and relationships in single-solution based metaheuristics. Hence, most of the search components will be reused by different S-metaheuristics. The aim of this section is to show not only the easy way to design an S-metaheuristic but also the high flexibility to transform an S-metaheuristic to another one reusing most of the design and implementation work. The different implementations are illustrated using the symmetric TSP problem.

2.10.1 Common Templates for Metaheuristics

As seen in Chapter 1, the common search components that have to be designed for any metaheuristic are

2.10.1.1 **Objective function:** The objective function is associated with the template `eoEvalFunc`, which is a problem-specific template. For the TSP problem, it corresponds to the total distance. It is straightforward to implement.

2.10.1.2 **Representation of solutions:** The encoding used for the TSP is based on per- mutations. Hence, an object `Route` is designed to deal with this structure. It corresponds to a permutation vector and its associated fitness. This is also a problem-specific template. However, one can reuse some popular representa- tions. In fact, in ParadisEO, the template associated with the encoding of so- lutions defines, for instance, the binary, discrete vector, real vectors, and trees. Those encodings may be reused to solve an optimization problem.

2.10.1.3 **Stopping criteria:** The stopping criteria for hill climbing are implicit, that is, the algorithm stops when a local optimal solution is found. Then there is nothing to specify.

2.10.2 Common Templates for S-Metaheuristics

As seen in this chapter, the common search components for S-metaheuristics are

2.10.2.1 **Initial solution:** As shown in Section 2.1.3, the initial solution may be gener- ated randomly or by any other heuristic (e.g., available algorithm as a template `moAlgo` or user-specified greedy algorithm).

2.10.2.2 **Neighborhood structure and its exploration:** A neighborhood for the TSP may be defined by the 2-opt operator. A class `TwoOpt` is then derived for the template `moMove`, which represents a move. Hence, a `TwoOpt` object is a `moMove` that can be applied to a object of type `Route` associated with the representation. To specify the exploration order of the neighborhood, the following templates have to defined:

– **First move:** The template `moMoveInit` defines the initial move to apply. The corresponding class in our example is

```
class TwoOptInit : public moMoveInit <TwoOpt>
```

– **Next move:** The template `moNextMove` defines the next move to apply. This template has also to check the end of the exploration.

2.10.2.3 **Incremental evaluation function:** The incremental objective function is asso- ciated with the template `moIncrEval`. The user must derive this template to implement the function. According to a solution and a move, this template is able to compute the fitness of the corresponding neighbor with a higher efficiency.

2.10.3 Local Search Template

Figure 2.39 shows the architecture of the hill-climbing template `moHC`. Once the common templates for all metaheuristics and S-metaheuristics are defined, `only one` search component is left, the neighbor selection strategy. The template associated with the selection strategy of the next neighbor is `moMoveselect`. The usual standard strategies are available in the template (see Section 2.3.1). The user has to choose one of them:[17]

2.10.3.1 **Best improvement** which corresponds to `moBestImprSelect` template.

2.10.3.2 **First improvement** which corresponds to `moFirstImprSelect` template.

2.10.3.3 **Random improvement** which corresponds to `moRandImprSelect` template.

From those defined templates, building a hill-climbing algorithm is completely done. The detailed description of the program associated with a hill-climbing algorithm using the defined search components and their associated templates is as follows:

[17] As the framework is a white box, the user can also design other selection strategies.

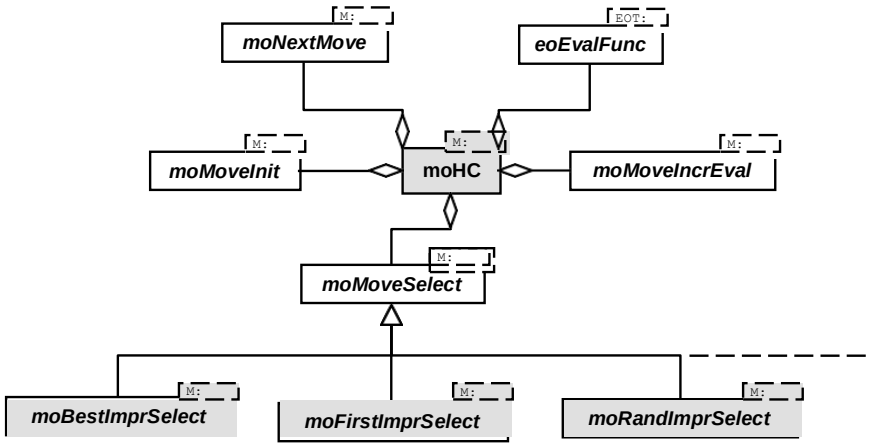


FIGURE 2.39 UML diagram of the hill-climbing template `moHC`. The templates `moEval-Func`, `moNextMove`, `moMoveIncrEval`, and `moMoveInit` are problem specific and need to be completed by the user. The template `moMoveSelect` is problem independent, that is, "plug and play" that can be completely reused.

```
...
//An initial solution.
Route route;
//An eoInit object (see ParadisEO-EO).
RouteInit route_init;
//Initialization.
route_init(route);
//The eoEvalFunc.
RouteEvaluation full_eval;
//Solution evaluation.
full_eval (route);
//The moMoveInit object.
TwoOptInit two_opt_init;
//The moNextMove object.
TwoOptNext two_opt_next;
//The moIncrEval object.
TwoOptIncrEval two_opt_incr_eval;
//The moMoveSelect.
moFirstImprSelect <TwoOpt> two_opt_select;
//or moBestImprSelect <TwoOpt> two_opt select;
//or moRandImprSelect <TwoOpt> two_opt select;
//or MyMoveSelect <TwoOpt> two_opt select;
//The moHC object.
moHC <TwoOpt> hill_climbing (two_opt_init, two_opt_next,
    two_opt_incr_eval, two opt select,full eval);
//The HC is launched on the initial solution.
hill_climbing (route) ;
```

A great advantage of using ParadisEO is that the user has to modify a given template (problem specific such as the objective function or problem independent such as the neighbor selection strategy), only the concerned template is updated; there is no impact on other templates and on the whole algorithm. For instance, if the user wants to define his own neighbor selection strategy, he can design a class for the `TwoOpt` move and use it as follows:

```
class MyMoveSelectionStrategy : public moMoveSelect <TwoOpt>
```

Another advantage of using the software framework is the ability to extend a metaheuristic to another metaheuristic in an easy and efficient manner. In the next section, it is shown how the hill-climbing algorithm has been evolved to tabu search, simulated annealing, and iterated local search. Most of the search components and their associated templates are reused.

2.10.4 Simulated Annealing Template

Figure 2.40 shows the architecture of the simulated annealing template `moSA`. If a comparison between Figs 2.39 and 2.40 is done, one can notice that most of the templates (search components) are the same.

The following search components and their associated templates are reused from the hill-climbing S-metaheuristic (moHC):

2.10.4.1 Objective function that corresponds to the template `eoEvalFunc`.

2.10.4.2 Incremental evaluation function that corresponds to `moMoveIncrEval`.

In addition to the search components associated with local search, the following components have to be defined for simulated annealing algorithm in the template `moSA`:

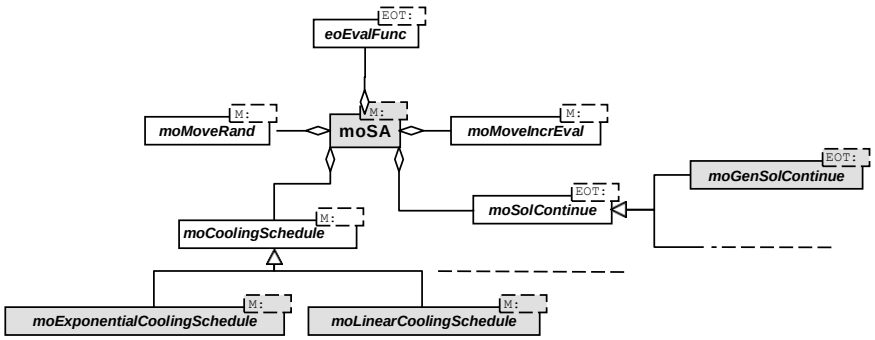


FIGURE 2.40 UML diagram of the simulated annealing template `moSA`. The template `moMoveRand` is specific to the problem; that is, the user has to define it, and the templates `moCoolingSchedule` and `moSolContinue` are problem independent and ready to use.

2.10.4.3 **Cooling schedule:** The cooling schedule of simulated annealing is specified in the template `moCoolingSchedule`. Some cooling functions are available, such as the linear function `moLinearCoolingSchedule` and the geometric one `moExponentialCoolingSchedule`.

2.10.4.4 **Stopping criteria** For any S-metaheuristic (e.g., tabu search), many defined stopping criteria in the template `moSolContinue` may be used.

2.10.4.5 **Random neighbor generation:** The template `moMoveRand` defines how a random neighbor is generated.

In the following simulated annealing program, the lines different from the hill-climbing program are represented in bold. It is easy to extract the characteristics of the implemented SA: a maximum number of iterations as stopping criteria, the Boltzmann distribution acceptance probability, and so on.

```
...
// An initial solution.
Route route;
// An eoInit object (see ParadisEO-EO).
RouteInit route_init;
// Initialization.
route_init(route);
// The eoEvalFunc.
RouteEvaluation full_eval;
// Solution evaluation.
full_eval (route);
// The moRandMove object.
TwoOptRand two_opt_rand;
// The moIncrEval object.
TwoOptIncrEval two_opt_incr_eval;
// The chosen cooling schedule object.
moExponentialCoolingSchedule cool_sched (Tmin, ratio);
// or moLinearCoolingSchedule cool_sched (Tmin, quantity);
// or MyCoolingSchedule cool_sched;
// The moSolContinue.
moGenSolContinue <Route> cont (max step);
// The moSA object.
moSA <TwoOpt> simulated_annealing (two_opt_rand, two_opt_incr_eval,
                                   cont, Tinit, cool_sched, full_eval);
// The simulated_annealing is launched on the initial solution.
simulated_annealing (route) ;
```

2.10.5 Tabu Search Template

Figure 2.41 shows the architecture of the tabu search template `moTS`. If a comparison between Figs 2.39 and 2.41 is done, one can notice that most of the templates (search components) are the same.

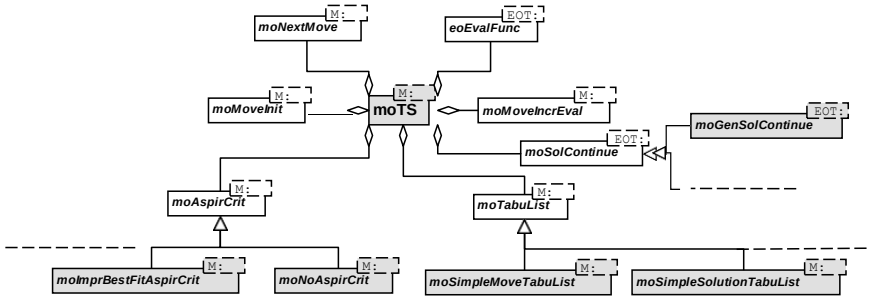


FIGURE 2.41 UML diagram of the tabu search template (moTS object). The templates `moAspirCrit`, `moTabuList`, and `moSolContinue` are problem-independent and ready-to-use templates.

The following search components and their associated templates are reused from the hill-climbing S-metaheuristic (moHC):

2.10.5.1 Objective function that corresponds to the template `eoEvalFunc`.

2.10.5.2 Incremental evaluation function that corresponds to `moMoveIncrEval`.

2.10.5.3 Neighborhood structure and its exploration that corresponds to the templates `moMoveInit`, `moNextMove`.

To design a tabu search algorithm, only the following search components have to be defined in the template `moTS`:

2.10.5.4 **Aspiration criteria** that corresponds to the template `moAspirCrit`. Two as- piration criteria are provided: the `moNoAspirCrit` that always rejects a move and the `moImpBestFitAspirCrit` that accepts a move if the fitness of the neighbor is better than the best found solution.

2.10.5.5 **Tabu list (short-term memory)** that corresponds to the template `moTabu- List`. Two basic tabu lists already exist: the ready-to-use template `moSim- pleMoveTabuList` that contains the moves and the template `moSimpleS- olutionTabuList` that contains the solutions.

2.10.5.6 **Stopping criteria** that corresponds to the template `moSolContinue`. This template allows to select different stopping criteria for tabu search such as

- `moGenSolContinue`, where the S-metaheuristic stops after a given maximum number of iteration.
- `moFitSolContinue`, where the algorithm continues its execution until a target fitness (quality) is reached.
- `moNoFitImprSolContinue`, where the algorithm stops when the best found solution has not be improved since a given number of iterations.
- `moSteadyFitSolContinue`, a combination of the first and the third criteria: the algorithm performs a given number of iterations and then it stops if the best found solution is not improved for a given number of iterations.

In the following tabu search program, the lines different from the hill-climbing program are represented in bold. It is easy to extract the characteristics of this tabu search: a maximum number of iterations as stopping criterion.

```
...
// An initial solution.
Route route;
// An eoInit object (see ParadisEO-EO).
RouteInit route_init;
// Initialization.
route_init(route);
// The eoEvalFunc.
RouteEvaluation full_eval;
// Solution evaluation.
full_eval (route);
// The moMoveInit object.
TwoOptInit two_opt_init;
// The moNextMove object.
TwoOptNext two_opt_next;
// The moIncrEval object.
TwoOptIncrEval two_opt_incr_eval;
// The moTabuList.
moSimpleSolutionTabuList<TwoOpt> tabu_list(10);
// or moSimpleMoveTabuList<TwoOpt> tabu_list(10);
// or MyTabuList<TwoOpt> tabu_list;
// The moAspirCrit.
moNoAspirCrit <TwoOpt> aspir_crit;
// or moImprBestFitAspirCrit<TwoOpt> aspir_crit;
// or MyAspirCrit<TwoOpt> aspir_crit;
//The moSolContinue.
moGenSolContinue <Route> cont (10000);
// or MySolContinue<TwoOpt> cont;
// The moTS object.
moTS <TwoOpt> tabu_search (two_opt_init, two_opt_next, two_opt_incr_eval,
                           tabu_list, aspir_crit, cont, full_eval);
// The TS is launched on the initial solution.
tabu_search (route) ;
```

2.10.6 Iterated Local Search Template

Figure 2.42 shows the architecture of the iterated local search template `moILS`. Once a given S-metaheuristic designed, it is very easy to construct an iterated local search algorithm.

The following search components have to be defined for the ILS algorithm in the template `moILS`:

2.10.6.1 **Local search algorithm:** The local search algorithm to be used must be spec- ified in the `moAlgo` template. Any developed S-metaheuristic may be used,

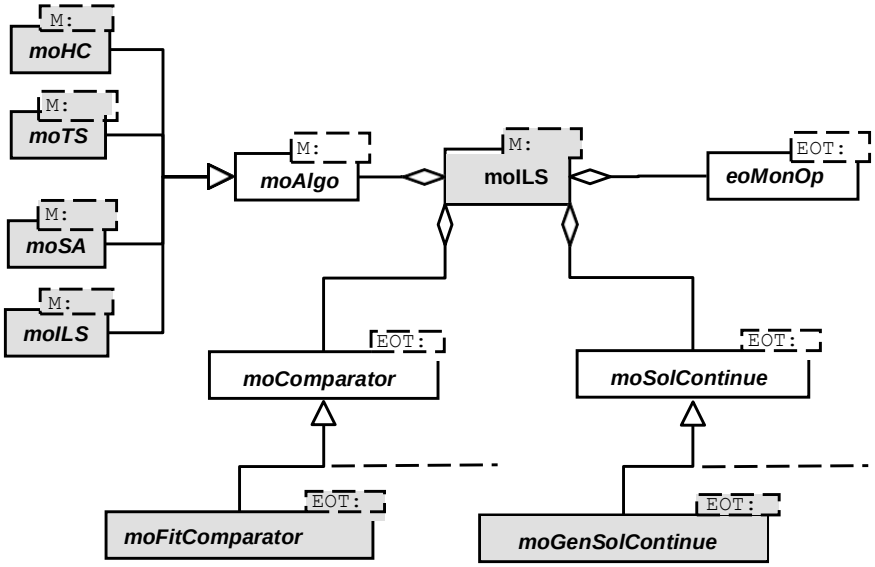


FIGURE 2.42 UML diagram of the iterated local search template `moILS`.

such as hill climbing (`moHC` template), tabu search (`moTS` template), simulated annealing (`moSA` template), any other user-defined S-metaheuristic, or why not another ILS `moILS`.

2.10.6.2 **Perturbation method:** The perturbation method is defined in the template `eoMonOp`.

2.10.6.3 **Acceptance criteria:** The search component dealing with the acceptance of the generated solution is defined in the template `moComparator`. Some usual acceptance functions are available such as `moFitComparator` that selects the new solution if it is better than the best found. Nevertheless, any specific acceptance method can be implemented; that is, a new class must be defined:

```
class MyComparisonStrategy : public moComparator <Route>
```

2.10.6.4 **Stopping criteria:** For any S-metaheuristic (e.g., simulated annealing, tabu search), many defined stopping criteria in the template `moSolContinue` may be used.

Starting from any S-metaheuristic program, the user can easily design an ILS for the symmetric TSP problem. In the following program, the additional lines to a S-metaheuristic program are represented in bold:

```
...
// An initial solution.
Route route;
// An object to initialise this solution.
RouteInit route_init;
```

```
// Initialization.
route_init(route);
// A full evaluation method <=> eoEvalFunc.
RouteEvaluation full_eval;
// Solution evaluation.
full_eval (route);
// The moMoveInit object.
TwoOptInit two_opt_init;
// The moNextMove object.
TwoOptNext two_opt_next;
// The moIncrEval object.
TwoOptIncrEval two_opt_incr_eval;
// moMoveSelect.
moFirstImprSelect <TwoOpt> two_opt_select;
// or moBestImprSelect <TwoOpt> two_opt_select;
// or moRandImprSelect <TwoOpt> two_opt_select;
// or MyMoveSelect <TwoOpt> two_opt_select;
// The moHC object.
moHC <TwoOpt> hill_climbing (two_opt_init, two_opt_next,
                                        two_opt_incr_eval, two_opt_select, full_eval);
   // The moSolContinue object.
moGenSolContinue <Route> cont (1000);
// The moComparator object.
moFitComparator <Route> comparator;
// The eoMonOp, in this example the well known CitySwap.
CitySwap perturbation;
// The moILS object.
moILS<TwoOpt> iterated_local_search(hill_climbing, comparator,
                                        perturbation, full_eval);
//The iterated local search is launched on the initial solution.
iterated_local_search (route) ;
```

2.11 CONCLUSIONS

In addition to the representation, the objective function and constraint handling that are common search concepts to all metaheuristics, the common concepts for single-solution based metaheuristics are as follows (Fig. 2.43):

- **Initial solution:** An initial solution may be specified randomly or by a given heuristic.
- **Neighborhood:** The main concept of S-metaheuristics is the definition of the neighborhood. The neighborhood has an important impact on the performances of this class of metaheuristics. The interdependency between representation and neighborhood must not be neglected. The main design question in S-metaheuristics is the trade-off between the efficiency of the representation/neighborhood and its effectiveness (e.g., small versus large neighborhoods).

- **Incremental evaluation of the neighborhood:** This is an important issue for the efficiency aspect of an S-metaheuristic.
- **Stopping criteria.**

Hence, most of the search components will be reused by different single-solution based metaheuristics (Fig. 2.43). Moreover, an incremental design and implementation of different S-metaheuristics can be carried out. In addition to the common search concepts of S-metaheuristics, the following main search components have to be defined for designing the following S-metaheuristics:

- **Local search:** Neighbor selection strategy.
- **Simulated annealing, demon algorithms, threshold accepting, great deluge and record-to-record travel:** Annealing schedule.
- **Tabu search:** Tabu list, aspiration criteria, medium- and long-term memories.
- **Iterated local search:** Perturbation method, acceptance criteria.
- **Variable neighborhood search:** Neighborhoods for shaking and neighborhoods for local search.

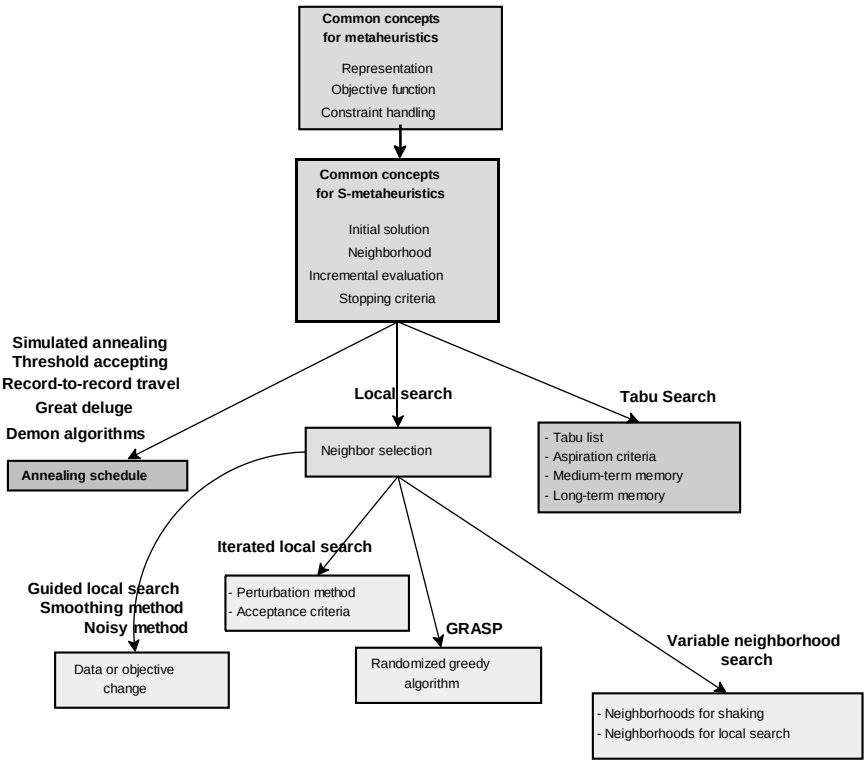


FIGURE 2.43 Common concepts and relationships in S-metaheuristics.

- **Guided local search, smoothing method, noisy method:** Function changing the input data or the objective.
- **GRASP:** Randomized greedy heuristic.

Moreover, there is a high flexibility to transform a S-metaheuristic into another one reusing most of the design and implementation work.

The analysis of landscapes of optimization problems is an important aspect in designing a metaheuristic (Fig. 2.44). It will be one of the most challenging problems in the theory of heuristic search algorithms. Indeed, the properties of the landscape has an important impact on the performance of metaheuristics. They have a major role in describing, explaining, and predicting the behavior of metaheuristics. One of the main lessons to learn is to analyze and exploit the structural properties of the landscape associated with a problem. One can also modify the landscape by changing the representation/neighborhood structure or the guiding function so that it becomes more "easy" to solve (e.g., deep valley landscape).

Once the S-metaheuristic is designed, the ParadisEO–MO software framework allows to implement it easily. The architecture modularity reduces the time and the complexity of designing S-metaheuristics. An expert user can without difficulty extend the already available boxes to suit to his problem and obtain more effective methods. Nevertheless, ParadisEO–MO can be used by newbies with a minimum

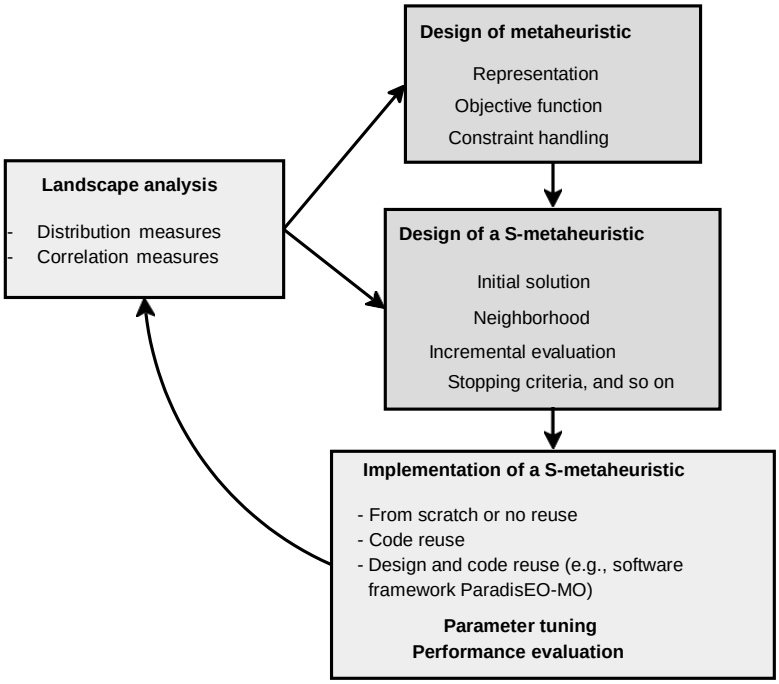


FIGURE 2.44 Development process of a S-metaheuristic.

of code to produce to implement diverse search strategies. A natural perspective is to evolve the open-source software by integrating more search components, heuristics (e.g., variable neighborhood search), and problem-solving environments (e.g., routing, scheduling, and assignment).

Each S-metaheuristic has some parameters to tune. The performance of the different metaheuristics is more or less sensitive to those parameters. However, the optimal tuning of the parameters will never overcome the bad design of search components (e.g., representation, neighborhood, and objective functions). The actual theory in S-metaheuristics, such as simulated annealing, focuses on the asymptotic convergence of the algorithms. It would be interesting to study and analyze more deeply the finite-time behavior of single-solution based metaheuristics. This will help the practical use of theoretical results, for instance, in the parameter tuning of a S-metaheuristics.

2.12 EXERCISES

Exercise 2.1 Initial solution for highly constrained problems. Let us consider the graph coloring problem. We would like to design a S-metaheuristic to solve the problem. Is it easy to generate a feasible random initial solution? Propose a greedy strategy to find a feasible initial solution.

Exercise 2.2 Incremental evaluation of the objective function. In many optimization problems, the encoding is based on permutations. A classical move operator in permutations is the exchange move in which two elements of the permutations are swapped. Find a permutation-based problem where the incremental evaluation associated with the exchange operator is easy (resp. difficult) to design.

Exercise 2.3 Representation and neighborhood for the vertex cover problem. A vertex cover of an undirected graph $G = (V, E)$ is a subset V' of the vertices of the graph containing at least one of the two end points of each edge:

$$V' \subseteq V : \forall \{a, b\} \in E : a \in V' OR b \in V'$$

The vertex cover problem is the optimization problem of finding a vertex cover of minimum size in a graph. In the graph of the Fig. 2.45, *A, C, E, F* is an example of a vertex cover. *B, D, E* is another vertex cover that is smaller. The vertex cover problem is related to the independent set problem: V' is an independent set if and only if its complement, $V \setminus V'$, is a vertex cover set. Hence, a graph with n vertices has an independent set of size k if and only if the graph has an vertex cover of size $n - k$. Propose a representation and a neighboring structure for the vertex cover problem. Can we apply the proposed solution to the independent set problem. Propose an incremental way to evaluate the neighbors.

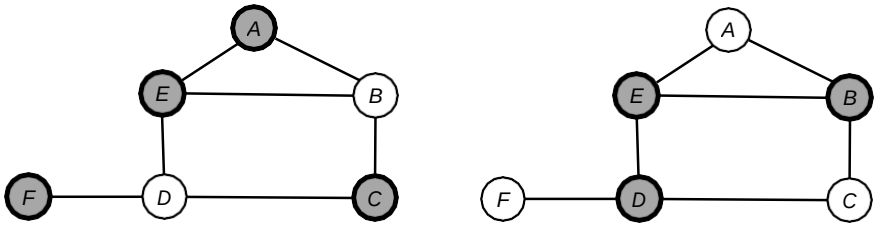


FIGURE 2.45 The vertex cover problem.

Exercise 2.4 Large neighborhood for the capacitated spanning tree problem. Let us consider the capacitated spanning tree problem presented in Example 2.41. The two-exchange neighborhood structure is based on exchanging a single node or a set of nodes between two subtrees. Generalize this neighborhood structure into a very large neighborhood structure based on the cyclic exchange neighborhood.

Exercise 2.5 Large neighborhood for the graph partitioning problem. The graph partitioning problem consists in finding a partitioning of a graph $G = (V, E)$ into two subsets of vertices V_1 and V_2 with $|V_1| = |V_2|$ such that the number of edges connecting vertices from V_1 and V_2 is minimized. Propose a representation and a neighborhood structure for the GPP problem.

The variable-depth search method implements a way to search an exponentially large neighborhood in polynomial time. This is made possible by some intelligent pruning of the search and heuristic rules. Devise a variable-depth search algorithm for the GPP problem. *Hint*: find the best pair $x_1 \in V_1$ and $y_1 \in V_2$ to swap. Swap these vertices and fix their position, that is, they cannot be swapped again in the step. In the kth iteration, $1 \leq k \leq n$, find the best pair x_k, y_k among the remaining $(n-k)^2$ pairs, swap this pair, and fix the position of x_k, y_k. These may not be swapped again. If k reaches n, then the roles of V_1 and V_2 are simply interchanged. It is important then to stop when $k < n$ swaps have been found such that the improvement in the quality of the partition is maximized. How can this be done efficiently?

Exercise 2.6 Neighborhood for different encodings of the JSP. Let us consider the job-shop scheduling problem and the various direct and indirect encodings proposed in Example 1.33. For each encoding, propose a neighborhood and compute its associated size.

Exercise 2.7 Large neighborhood based on cyclical shift. Let us consider the TSP problem. A tour is *pyramidal* if it starts in city 1, then visits cities in increasing order until it reaches city n, and finally returns through the remaining cities in decreasing order. Let $\pi(i)$ represent the city in the ith position of the tour π. A tour π' is a pyramidal neighbor of a tour π if there exists an integer p $(0 \leq p \leq n)$ such that the

following conditions hold:

$$\pi'(1) = \pi(i_1), \pi'(2) = \pi(i_2), \ldots, \pi'(p) = \pi(i_p), \qquad \text{with } i_1 < i_2 < \cdots < i_p$$
$$\pi'(p+1) = \pi(j_1), \pi'(p+2) = \pi(j_2), \ldots, \pi'(n) = \pi(j_{n-p}),$$
$$\text{with } j_1 > j_2 > \cdots > j_{n-p}$$

For instance, if $\pi = (1, 2, 3, 4, 5, 1)$, then a pyramidal neighbor may be $\pi' = (1, 2, 5, 4, 3, 1)$, $(1, 3, 5, 4, 2, 1)$, and so on. What is the size of this pyramidal neighborhood? Design an efficient algorithm to explore it knowing that its complexity may be reduced to $O(n^3)$. The algorithm can be based on the shortest path algorithm in the improvement graph. The drawback of this pyramidal neighborhood is that the edge $(1, 2)$ is always contained in the tours. Find an alternative to solve this issue. Show that if a solution is a local optimum for the pyramidal neighborhood, then it is a local optimum for the 2-opt neighborhood.

Exercise 2.8 Size versus time to explore a large neighborhood. The critical issue in selecting a large neighborhood is the trade-off between the size of the neighborhood and the time to explore it. Let us consider the TSP problem. Compute the size and the time complexities for the following neighborhoods: city swap, 2-opt, k-opt, pyramidal, cyclical shift, compound swaps, and Halin graphs. What is the diameter of the above neighborhoods?

Exercise 2.9 Incremental objective function evaluation for the QAP. The incremental evaluation of the objective function in single-solution based algorithms is a very important issue to reduce the complexity of the evaluation of the neighborhood. In this exercise, we will tackle this question for a well-known assignment problem the quadratic assignment problem. The QAP arises in many applications such as facility location, data analysis, VLSI design, image synthesis, and so on. The problem can be defined as follows. Given a set of n objects $O = \{O_1, O_2, \ldots, O_n\}$, a set of n locations $L = \{L_1, L_2, \ldots, L_n\}$, a flow matrix C, where each element c_{ij} denotes a flow cost between the objects O_i and O_j, a distance matrix D, where each element d_{kl} denotes a distance between location L_k and L_l, find an object–location mapping $m : O \rightarrow L$ that minimizes the objective function f:

$$f = \sum_{i=1}^{n} \sum_{j=1}^{n} c_{ij} \cdot d_{M(i)M(j)}$$

Define an incremental function for the following encoding and move operator. The encoding of a solution is based on a permutation of n integers $s = (l_1, l_2, \ldots, L_n)$ where l_i denotes the location of the object O_i. The move operator is based on the exchange operator where two elements of the permutation are exchanged.

Exercise 2.10 Efficient evaluation of the neighborhood. Let us consider the maximum independent set problem defined in Example 2.27. The neighborhood is defined by the exchange of a vertex $v \in X$ with a vertex $w \in E - X$. The size of the neighborhood is $k(n - k) = O(nk)$ where n represents the number of vertices. Let us denote $\hat{W}_X(v)$ the set of vertices $w \in X$ that are connected to $v \in V$ by an edge. The vertices v in X are sorted according to nonincreasing values of $\hat{W}_X(v)$, and the vertices $w \in V - X$ are sorted according to nondecreasing values of $\hat{W}_X(w)$. Show that the best neighbor may be found in $O(n)$ instead of $O(kn)$.

Exercise 2.11 Difficult incremental objective function evaluation. Identify some optimization problems and local search operators in which it is difficult to compute the incremental objective function.

Exercise 2.12 Distance in permutation space. Find an algorithm with time complexity $O(n)$ that computes the distance between any two permutations. The problem may be reduced to the shortest path between two solutions of the search space.

Exercise 2.13 Distance using 2-opt operator. Let us consider the traveling salesman problem. An S-metaheuristic using the 2-opt neighborhood has been designed to solve the problem. We need a definition of a distance to perform a landscape analysis of the problem. The distance $O_{2-\text{opt}}(t_1, t_2)$ denotes the minimum number of applications of the 2-opt move to obtain t_2 from t_1. Is there any polynomial algorithm to compute such as distance? If not, propose an approximation of such a distance that can be computed efficiently.

Exercise 2.14 Hard unimodal problem for local search. The goal of this exercise is to construct a difficult problem for local search-based algorithms. The well-known types of difficulties for local search are multimodality, full deception, and isolation (needle in a haystack).

Construct an optimization problem in which the landscape is unimodal (any local optimum is a global optimum) and where a simple hill climber finds the global optimal solution in exponential time. Thus, unimodality is not a sufficient indicator to analyze the landscape difficulty of problems for local search-based algorithms.

Exercise 2.15 Greedy algorithms and epistasis. Starting from scratch, a greedy algorithm selects at each iteration a decision variable or component of a solution to be assigned until a complete solution is generated. Show that greedy algorithms are suitable for optimization problems with low epistasis. The response may be based on the following design questions of a greedy algorithm: selection of the order of the components and the value assignment to the components.

Exercise 2.16 Fractal landscape. Show that the NK-model landscape, TSP with the 2-opt operator, and the GBP are fractal landscapes ($h = \frac{1}{2}$).

Exercise 2.17 NFL theorem. Following the NFL theorem, one can state that all optimization algorithms are equivalent. Is it correct?

Exercise 2.18 Fitness distance correlation. The main drawback of the FDC analysis is that it requires the knowledge of the global optimal solution of the target optimization problem. Give some responses to this limitation when

- There are many known global optimal solutions.
- No global optimal solution is known for the problem.

Exercise 2.19 Tabu list representation. Suppose we have to solve the vehicle routing problem that has been defined in Example 1.11. The transformation operator used consists in moving a customer c_k from route R_i to route R_j. The tabu list is represented by the moves attributes. Propose three representations that are increasing in terms of their severity.

Exercise 2.20 Tabu list with random size. The commonly used representation for fixed size tabu lists are circular lists. Indeed, at each iteration, the last move is added to the list, whereas the oldest move is removed from the list. Determine the representation for tabu lists where the size is randomly updated during the search.

Exercise 2.21 Multiple tabu list. In scheduling problems such as the permutation flow-shop scheduling problem, various move operators may be used in parallel during the search. Propose multiple tabu list for the permutation flow-shop scheduling problem.

Exercise 2.22 Strategic oscillation, GRASP, and noisy method. The strategic oscillation approach introduced in tabu search is characterized by alternation between diversification in the search space and intensification of the search into a promising region. We would like to introduce strategic oscillation into the constructive procedure of the GRASP metaheuristic. How can we use the noisy method as a strategic oscillation strategy? Illustrate using a constrained spanning tree problem where the noisy method consists in perturbing the weights w_e of the graph edges e by $w_e \cdot r_i(e)$, where $r_i(e)$ is the perturbation coefficient applied to iteration i. We use the following search memory t, where $t_{i-1}(e)$ represents the number of locally optimal solutions in which the edge e appeared ($0 \leq t_{i-1}(e) \leq i$). The search memory can be viewed as a medium-term memory in tabu search.

Exercise 2.23 Black box for local search. Suppose that we have a black-box solver available to solve a given problem. This black-box solver is based on a local search procedure that has as an input an initial solution and as an output the local optima obtained. Which single-solution based metaheuristic can we use to solve the problem?

Exercise 2.24 Guided local search for the QAP. The quadratic assignment problem has been defined in Exercise 2.9. The goal of this exercise is to design a guided local search algorithm for the QAP. Suppose we have already at hand a local search (or any S-metaheuristic) procedure for the QAP. Then, it is required to define the augmented objective function. Hence, the following concepts have to be formulated: the features of a solution and the cost associated with a feature.

Exercise 2.25 Smoothing algorithms for graph problems. The landscape smoothing strategy has been widely used for the TSP problem. The smoothing operation consists in changing the weights (distances) of the input graph. Propose a smoothing operation for the graph partitioning problems and weighted clique problems.

Exercise 2.26 Smoothing constraints for covering and matching problems. Traditional smoothing techniques focus on the objective function. For some constrained problems, such as covering and matching problems, it would be interesting to smooth the constraints to generate a problem with an idealized landscape structure. Propose such a smoothing operation for constraints.

Exercise 2.27 Smoothing algorithm using P-metaheuristics. Traditionally, the smoothing algorithm has been used in conjunction with local search algorithms (S-metaheuristics). The smoothing algorithm can be extended to be used in conjunction with a population-based metaheuristic. Propose such an extension.

Exercise 2.28 Smoothing factor controlling. There are different ways of controlling the smoothing factor α that represents the strength of the landscape smoothing operation. Which properties must have such a controlling scheme?

Exercise 2.29 Links of the noising method with simulated annealing and threshold accepting. In the way the noising method operates, it may be similar to simulated annealing and threshold accepting algorithms. For which noise distributions may the noising algorithm be reduced to simulated annealing? The same question holds for threshold accepting. One consequence of these generalizations is that they also provide results on the convergence of some noising schemes. These convergence results for simulated annealing [3] and threshold accepting [28] show that there exist noising schemes for which the noising methods converge toward an optimal solution when the number of iterations is infinite. In Ref. [732], the authors present the convergence of the first kind of a noising method by giving sufficient conditions for convergence toward an optimal solution.

Similarly, we may consider that the noising method is a generalization of the threshold accepting algorithm. In this method, the neighborhood is systematically explored and the current solution s is replaced with one of its neighbors s' if s' is better than s or if the increase (for a minimization problem) does not overpass a given threshold; this threshold depends on the iteration and decreases during the process to 0. So, with respect to simulated annealing, the two main differences rely on the

neighborhood exploration and on the fact that the acceptance criterion is no longer the exponential and nondeterministic Metropolis criterion but the following one: at the kth trial, s' is accepted instead of s if $\boldsymbol{f}(s') - \boldsymbol{f}(s) < T_k$, where T_k is the threshold of the kth trial, with $T_k \geq 0$ and $T_{k+1} < T_k$ and $T_K = 0$ if K is the total number of trials. It is quite easy to see that these thresholds can be seen as noises added to the variation of $\boldsymbol{f}$ (i.e., according to the second way of adding a noise described above).

Exercise 2.30 Automatic setting of parameters for the noising method. The objective here is to design an automatic tuning of the parameters of the noising methods. Our goal is to have a noising framework independent of the problem to solve and of the probability distribution of the noises, and with only one parameter: the search time fixed by the user. Propose such an automatic tuning. It would be preferable to spend more time on less noised objective function.

Exercise 2.31 Elite set in an iterative S-metaheuristic. In the design of an iterative S-metaheuristic such as iterated local search or GRASP, our goal is to maintain an elite set of solutions E_s. An iterative S-metaheuristic generates a single solution at each iteration of the algorithm. The elite set has a given maximum size k_{max} in terms of the number of solutions that is updated during the search. In the first k_{max} iterations of the iterative S-metaheuristic, the obtained solution is systematically integrated into the elite set. Which integration policies can we apply to the future iterations taking into account the quality of solutions and the diversity of the elite set E_s.

Exercise 2.32 Toward a dependency of iterations in GRASP. In most of the GRASP metaheuristic implementations, the independence between iterations is assumed. The iterations are memoryless; that is, there is no use of a search memory between successive iterations. For instance, the search does not learn from the history of solutions found. Hence, the same results are obtained independent of the order of the iterations. Which kind of search memory can be introduced into GRASP to

- Improve the quality of solutions.
- Avoid redundant work.
- Speed up the search.

Propose such a strategy that makes the iterations of the GRASP dependent on each other.

Exercise 2.33 From the capacitated vehicle routing problem to the capacitated spanning tree problem. The objective of this exercise is to show how problem-solving results can be reused thanks to a problem reduction. Let us consider the well-known capacitated vehicle routing problem where many metaheuristics have been designed. Our objective is to solve the capacitated minimum spanning tree problem presented in Example 2.41. The concept of clustering of nodes satisfying the capacity constraint is present in both problems. Show that the CMST problem is a relaxation of the CVRP problem. Then, in solving the CVRP, we obtain an upper bound for the

CMST. Once we have a cluster of nodes, finding a minimum spanning tree problem is a polynomial problem where well-known greedy heuristics may be used, such as the Prim or Kruskal ones. Combine these two ideas to design a S-metaheuristic (e.g., GRASP algorithm) for the CMST problem. The same exercise may be applied for a P-metaheuristic (e.g., ant colony optimization) presented in Chapter 3.

Exercise 2.34 GRASP construction method and the noisy method. In the design of a GRASP metaheuristic to solve an optimization problem, there is a need of a randomized greedy heuristic. For some problems, no such randomized greedy heuristic is available. How can we use the noisy method to replace the construction phase of the GRASP metaheuristic. Illustrate the answers using the clique partitioning problem of Example 2.22.

Exercise 2.35 Perturbation in ILS versus shaking in VNS. In the design of an ILS metaheuristic, a perturbation method must be defined. In the VNS metaheuristic, shaking is used to generate an initial solution for local search or the VND method. What are the similarities between these two procedures: perturbation in ILS and shaking in VNS.

Exercise 2.36 Bias functions in the construction procedure of GRASP. In the construction procedure of the GRASP metaheuristic, the next element to introduce in the partial solution from the RCL list is chosen randomly. Indeed, the elements belonging to RCL are assigned an equal probability. Let $r(\sigma)$ denote the rank of element σ and bias($r(\sigma)$) a bias function. The probability $\pi(\sigma)$ of selecting an element from the set RCL is

$$\pi(\sigma) = \frac{\text{bias}(r(\sigma))}{\sum_{\sigma' \in \text{RCL}} \text{bias}(r(\sigma'))}$$

The random bias function commonly used in GRASP is bias(r) $= 1$ for $r \in$ RCL. Propose other probability distributions that may be used to bias the selection of elements from the set RCL.

Exercise 2.37 Robust optimization using S-metaheuristics. Suppose that we have used a S-metaheuristic to solve a given optimization problem. How can we estimate the robustness of the obtained solution in terms of the disturbance of the decision variables.

Exercise 2.38 Local search for the *N*-Queens problem. 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 queen's moves. Any queen is assumed to be able to attack any other. Suppose that the objective function f isto reduce the number of conflicts. The neighborhood is defined as a move of a queen toward its eight neighbor positions on the chess (Fig. 2.46). Design and implement a basic local search algorithm to solve the problem under the ParadisEO framework.

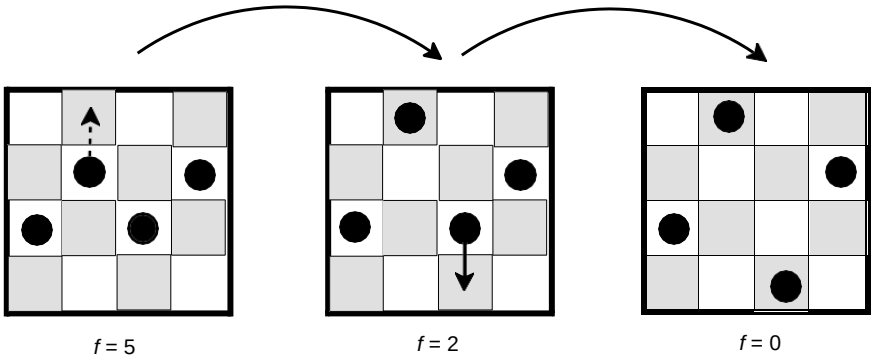


FIGURE 2.46 A local search procedure for the *N*-Queens problem.

Propose an improvement of the local search procedure for a more effective search algorithm.

Exercise 2.39 Flexibility of the local search template. Given a local search template developed under ParadisEO-MO to solve an optimization problem *P* (see Section 2.10.3 or many examples on the Web site of the framework), what the user has to perform if he wants to

- Change the neighbor selection strategy.
- Modify the guiding objective function to solve the same problem *P*.
- Transform the neighborhood but keep the same representation.
- Extend the local search to tabu search, simulated annealing, and iterated local search.
- Solve another optimization problem *Q* that uses the same representation and neighborhood.

Exercise 2.40 Local search under ParadisEO. Evaluate the performance (quality of solutions) of the developed local search algorithm on different benchmarks. Experiment different neighbor selection strategies (first improvement, best improvement) in terms of solution quality and computational time. Test if the algorithm is sensitive to the initial solution by running the algorithm for different random initial solutions. Compare the performances of the local search approach with

- Random walk, which generates randomly successive neighbor solutions. The best found solution is always updated.
- Random sampling, which is a variation of random walk, where a neighbor-generated solution is accepted if it is better than the current solution.

Exercise 2.41 Flexibility of the tabu search template. Given a tabu search developed under ParadisEO–MO to solve an optimization problem *P* (see Section 2.10.5 or the examples on the Web site of the framework), what the user has to perform if he wants to

- Modify the tabu list definition.
- Change the aspiration function.
- Experiment another stopping criteria (e.g., stops when a given fitness (quality) is attained)
- Include medium-term or long-term memories.

Exercise 2.42 Flexibility of the simulated annealing template. Given a simulated annealing template developed under ParadisEO-MO to solve an optimization problem *P* (see Section 2.10.4 or the examples on the Web site of the framework), what the user has to perform if he wants to

- Modify the cooling schedule.
- Include a more efficient incremental objective function. Can is this new function be used by any S-metaheuristic?
- Implement the other SA-inspired algorithms such as threshold accepting, demon algorithms, great deluge, and record-to-record algorithms to solve the problem *P*. Afterward, these algorithms will be available to be used by other users.

Exercise 2.43 Threshold accepting template. The threshold accepting algorithm belongs to the family of simulated annealing algorithms. Show how the simulated annealing template of ParadisEO-MO can be easily modified to implement a threshold accepting template.

Exercise 2.44 Flexibility of the iterated local search template. Given an iterated local search template developed under ParadisEO-MO to solve an optimization problem *P* (see Section 2.10.6 or the examples on the Web site of the framework), what the user has to perform if he wants to

- Replace the S-metaheuristic algorithm (e.g., from local search to tabu search or simulated annealing).
- Experiment another perturbation method and/or acceptance criteria.

CHAPTER 3

Population-Based Metaheuristics

Population-based metaheuristics (P-metaheuristics) share many common concepts. They could be viewed as an iterative improvement in a population of solutions. First, the population is initialized. Then, a new population of solutions is generated. Finally, this new population is integrated into the current one using some selection procedures. The search process is stopped when a given condition is satisfied (stopping criterion). Algorithms such as evolutionary algorithms (EAs), scatter search (SS), estimation of distribution algorithms (EDAs), particle swarm optimization (PSO), bee colony (BC), and artificial immune systems (AISs) belong to this class of metaheuristics.

This chapter is organized as follows. After the high-level template of P-metaheuristics is presented, Section 3.1 details the common search components for P-metaheuristics, namely, the determination of the initial population and the stopping criteria. In Section 3.2, the focus is on evolutionary algorithms such as genetic algorithms, evolution strategies (ESs), evolutionary programming, and genetic programming. The common search components of this family of P-metaheuristics (e.g., selection, variation operators, and replacement) are outlined in Section 3.3. Section 3.4 presents other evolutionary algorithms such as estimation of distribution algorithms, differential evolution, coevolutionary algorithms, and cultural algorithms. Section 3.5 presents the scatter search and path relinking, and Section 3.6 deals with swarm intelligence algorithms (e.g., ant colonies (ACs), particle swarm optimization). Other nature-inspired P-metaheuristics such as bee colonies (e.g., food foraging, queen bee marriage) and artificial immune systems (e.g., clonal selection, immune network) are presented in Section 3.7. Finally, Section 3.8 presents the ParadisEO–EO (evolving objects) module of the ParadisEO software framework that is dedicated to the implementation of P-metaheuristics. Some design and implementation issues of P-metaheuristics such as evolutionary algorithms, particle swarm optimization, and estimation of distribution algorithms are illustrated.

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

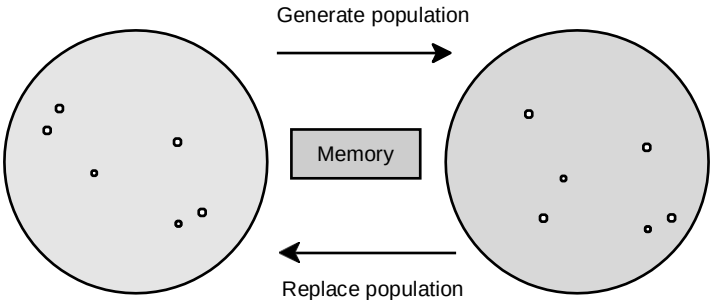


FIGURE 3.1 Main principles of P-metaheuristics.

3.1 COMMON CONCEPTS FOR POPULATION-BASED METAHEURISTICS

Population-based metaheuristics start from an initial population of solutions.[1] Then, they iteratively apply the generation of a new population and the replacement of the current population (Fig. 3.1). In the generation phase, a new population of solutions is created. In the replacement phase, a selection is carried out from the current and the new populations. This process iterates until a given stopping criteria. The generation and the replacement phases may be *memoryless*. In this case, the two procedures are based only on the current population. Otherwise, some history of the search stored in a memory can be used in the generation of the new population and the replacement of the old population. Most of the P-metaheuristics are nature-inspired algorithms. Popular examples of P-metaheuristics are evolutionary algorithms, ant colony optimization, scatter search, particle swarm optimization, bee colony, and artificial immune systems. Algorithm 3.1 illustrates the high-level template of P-metaheuristics.

Algorithm 3.1 High-level template of P-metaheuristics.

```
P = P0; /* Generation of the initial population */
t = 0;
Repeat
   Generate(P't); /* Generation a new population */
   Pt+1 = Select-Population(Pt ∪ P't); /* Select new population */
   t = t + 1;
Until Stopping criteria satisfied
Output: Best solution(s) found.
```

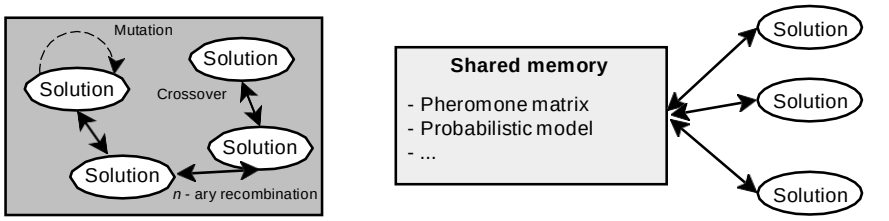
P-metaheuristics differ in the way they perform the generation and the selection procedures and the search memory they are using during the search.

[1]Some P-metaheuristics such as ant colony optimization start from partial or empty solutions.

TABLE 3.1 Search Memories of Some P-Metaheuristics

P-metaheuristic	Search Memory
Evolutionary algorithms	Population of individuals
Scatter search	Population of solutions, reference set
Ant colonies	Pheromone matrix
Estimation of distribution algorithms	Probabilistic learning model
Particle swarm optimization	Population of particles, best global and local solutions
Bee colonies	Population of bees
Artificial immune systems: clonal selection	Population of antibodies

- **Search memory:** The memory of a P-metaheuristic represents the set of information extracted and memorized during the search. The content of this memory varies from a P-metaheuristic to another one (Table 3.1). In most of the P-metaheuristics such as evolutionary algorithms and scatter search, the search memory is limited to the population of solutions. In ant colonies, the pheromone matrix is the main component of the search memory, whereas in estimation distribution algorithms, it is a probabilistic learning model that composes the search memory.
- **Generation:** In this step, a new population of solutions is generated. According to the generation strategy, P-metaheuristics may be classified into two main categories (Fig. 3.2):
  - **Evolution based:** In this category of P-metaheuristics, the solutions composing the population are selected and reproduced using variation operators (e.g., mutation, recombination[2]) acting *directly* on their representations. A new solution is constructed from the different attributes of solutions belongingto the current population. Evolutionary algorithms and scatter search represent



(a) Evolutionary-based P-metaheuristics: evolutionary algorithms, scatter search, ...

(b) Blackboard-based P-metaheuristics: ant colonies, estimation distribution algorithms, ...

FIGURE 3.2 Evolution based versus blackboard based strategies in P-metaheuristics.

[2]Also called crossover and merge.

well-known examples of this class of P-metaheuristics. In EAs, the recombination operator is generally a binary operator (crossover), while in SS, the recombination operator may be an n-ary operator ($n > 2$).

– **Blackboard based**[3]**:** Here, the solutions of the population participate in the construction of a shared memory. This shared memory will be the main input in generating the new population of solutions. The recombination in this class of algorithm between solutions is indirect through this shared memory. Ant colonies and estimation distribution algorithms belong to this class of P-metaheuristics. In the former strategy, the shared memory is represented by the pheromone matrix, while in the latter strategy, it is represented by a probabilistic learning model. For instance, in ant colonies, the generated solutions by past ants will affect the generation of solutions by future ants via the pheromone. Indeed, the previously generated solutions participate in updating the pheromone.

· **Selection:** The last step in P-metaheuristics consists in selecting the new solutions from the union of the current population and the generated population. The traditional strategy consists in selecting the generated population as the new population. Other strategies use some *elitism* in the selection phase where they provide the best solutions from the two sets. In blackboard-based P-metaheuristics, there is no explicit selection. The new population of solutions will update the shared search memory (e.g., pheromone matrix for ant colonies, probabilistic learning model for estimation of distribution algorithms), which will affect the generation of the new population.

As for S-metaheuristics, the search components that allow to define and differentiate P-metaheuristics have been identified. The common search concepts for P-metaheuristics are the determination of the initial population and the definition of the stopping criteria.

3.1.1 Initial Population

Due to the large diversity of initial populations, P-metaheuristics are naturally more exploration search algorithms whereas S-metaheuristics are more exploitation search algorithms. The determination of the initial population is often disregarded in the design of a P-metaheuristic. Nonetheless, this step plays a crucial role in the effectiveness of the algorithm and its efficiency. Hence, one should pay more attention to this step [522].

In the generation of the initial population, the main criterion to deal with is diversification. If the initial population is not well diversified, a premature convergence can occur for any P-metaheuristic. For instance, this may happen if the initial population is generated using a greedy heuristic or a S-metaheuristic (e.g., local search, tabu search) for each solution of the population.

[3]A blackboard system is an artificial intelligence application based on the blackboard architectural model, where a shared knowledge base, the "blackboard," is iteratively updated by a diverse group of agents [245].

TABLE 3.2 Analysis of the Different Initialization Strategies

Strategy	Diversity	Computational Cost	Quality of Initial Solutions
Pseudo-random	++	+++	+
Quasi-random	+++	+++	+
Sequential diversification	++++	++	+
Parallel diversification	++++	+++	+
Heuristic	+	+	+++

The evaluation is better with more plus (+) sign. Sequential and parallel diversification strategies provide in general the best diversity followed by the quasi-random strategy. The heuristic initialization provides in general better solutions in terms of quality but with the expense of a higher computational cost and a reduced diversity. This will depend on the fitness landscape of the tackled optimization problem. For some landscapes (e.g., flat rugged), the diversity may remain important.

In some P-metaheuristics such as scatter search, the diversification criterion is explicitly taken into account in the generation of the initial population. Some diversification criteria are optimized in the generation of the initial population such as maximizing the minimum distance between any two solutions of the initial population:

$$\text{Max}_{i=1,n}(\text{Min}_{j=1,i-1}\{d_{ij}\})$$

where d_{ij} represents the distance in the decision space between two solutions i and j and n is the size of the population.

Strategies dealing with the initialization of the population may be classified into four categories: random generation, sequential diversification, parallel diversification, and heuristic initialization. They may be analyzed according to the following criteria: diversity, computational cost, and quality of the solutions (Table 3.2).

3.1.1.1 Random Generation Usually, the initial population is generated randomly. For instance, in continuous optimization, the initial real value of each variable may be generated randomly in its possible range.

Example 3.1 Uniform random population in global optimization. In continuous optimization, each decision variable x_j is defined to be in a given range $[l_j, u_j]$. Let us generate a randomly distributed initial population P_0 of size n. Each solution x_i of the population is a k-dimensional real vector $x_{ij}, j \in [1, k]$. Each element of the vector x_{ij} is generated randomly in the range $[l_j, u_j]$, representing the lower and the upper bounds for each variable:

$$x_{ij} = l_j + \text{rand}_j[0, 1] \cdot (u_j - l_j), i \in [1, n], j \in [1, k]$$

where rand_j is a uniformly distributed random variable in the range [0, 1]. If the limits (lower and upper bounds) are not well defined, the bounds should be initialized large enough to encompass the search space delimited by optimal solutions.

The random generation may be performed according to *pseudo-random* numbers or a *quasi-random* sequence of numbers. The most popular random generation of the population is the pseudo-random one[4] (Fig. 3.4). Indeed, as it is impossible in practice to generate algorithmically truly independent random numbers (genuine random), pseudo-random numbers are generated using various classical generators (e.g., congruential quadratic, recursive lagged Fibonacci) [313]. Care should be taken to use pseudo-random generators that provide good properties (e.g., sensibility to seeds and coefficients).

In a quasi-random sequence, the goal of the generator is related not only to the independence between the successive numbers but also to their dispersion. Many quasi-random generators that try to cover the search space exist in the literature (e.g., Niederreiter) [313]. In a quasi-random sequence, the diversity of the population is generally better than in a pseudo-random generation (Table 3.2).

In discrete optimization, the same uniform random initialization may be applied to binary vectors, discrete vectors, and permutations.

3.1.1.2 Sequential Diversification The initial population can also be uniformly sampled in the decision space. Some initialization procedures do not require any evaluation of the objective function or constraints associated with the optimization problem. In a sequential diversification, the solutions are generated in sequence in such a way that the diversity is optimized.

Example 3.2 Simple sequential inhibition process. A well-known sequential diversification strategy in statistics is the simple sequential inhibition (SSI) process [210]. Given Q the current subpopulation, which initially is composed of only one randomly picked solution. Any new selected solution must be at a minimum distance O to all other solutions of the current subpopulation Q. Hence, a pseudo-random solution is generated until it verifies this property. This process is repeated until the population is filled with the specified number of solutions. The drawback of this strategy is its relatively high computation cost. The number of iterations performed cannot be known *a priori*. However, it ensures a good distribution of the population, that is, the minimum distance between any two solutions of the initial population is greater than O. Let us notice that the distance threshold $O > 0$ is a parameter of the strategy.

3.1.1.3 Parallel Diversification In a parallel diversification strategy, the solutions of a population are generated in a parallel independent way.

Example 3.3 Uniform sampling using Latin hypercube sampling. Let n be the size of the population and k be the number of variables. The variable range of each variable i is divided into n equal segments of size $(u_i - l_i)/k$, where l_i and u_i represent successively the lower and upper bounds for each variable i. A Latin hypercube sample

[4]A sequence of infinite size is random if the amount of information it contains is also infinite. It is impossible to test the randomness of a finite sequence [471]. In this book, the term random refers to pseudo-random.

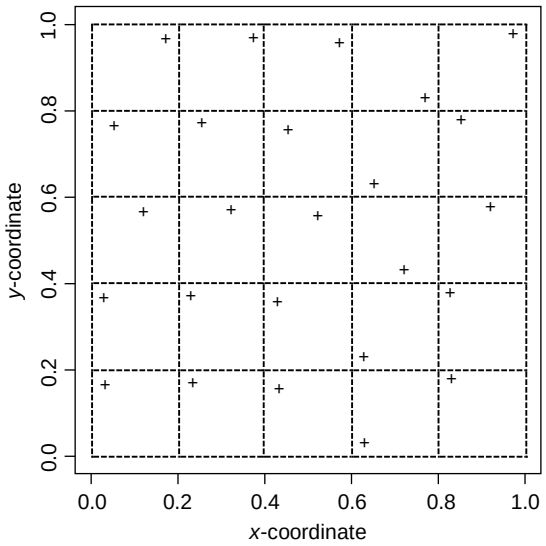


FIGURE 3.3 In the Latin hypercube strategy, the search space is decomposed into 25 blocks and a solution is generated pseudo-randomly in each block.

is generated by sampling a random real number that is generated in each segment [536]. This strategy is similar to nonaligned systematic sampling, a well-known sampling design procedure in statistics [658]. Figures 3.3 and 3.4 illustrate the difference between pseudo-random generation and uniform sampling. It is clearly shown that using Latin hypercube sampling, the coverage is much better with uniform sampling than using pseudo-random generation.

To generalize this procedure to a population of vector solutions, an unbiased Knuth shuffling strategy may be used. A random permutation of n elements (from 1 to n) is generated. The solution with index j is assigned the value located at the position $\pi(j)$th of the permutation. This procedure is iterated for all variables. This strategy gives a good overall random unbiased distribution of the population in the decision space and does not require any evaluation of the objective function or constraints [513].

Example 3.4 Diversified initial population of permutations. Let us consider a permutation problem, the quadratic assignment problem (QAP), which is defined in Exercise 2.9. A solution is represented by a permutation π of m integers $\{1, 2, \ldots, m\}$, where m is the size of the problem (number of locations and objects). A diversification criterion may be selected such as for each object i and for each location j, there is a solution of the initial population in which the object i is mapped on the location j [167]. The maximum distance between any two solutions will be equal to $\sqrt{2n}$. First, a random initial solution X_1 is generated. The second solution X_2 is constructed by initializing $\pi(k_i^1)$ to site i, where k_i^1 represents the object on the location i in X_1. Then, the third solution X_3 is constructed by initializing $\pi(k_i^2)$ to site i, where k_i^2 represents the object on the location i in X_2. The same process is iterated $n - 1$ times to

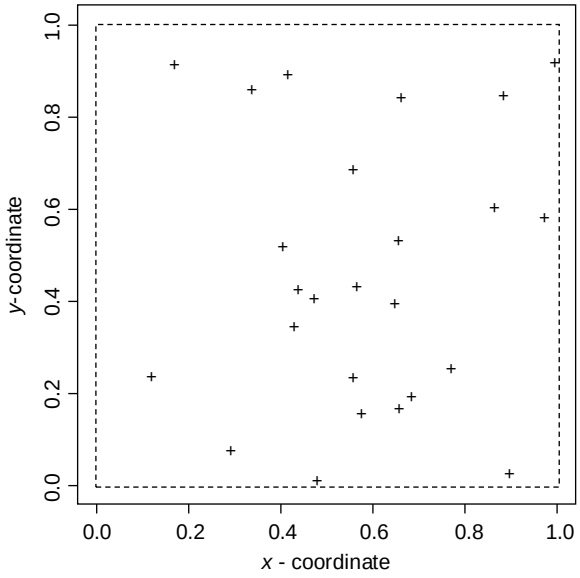


FIGURE 3.4 In the pseudo-random generation, 25 solutions are generated independently in the search space.

generate a population of size n. Figure 3.5 illustrates such a diversified initial population of permutations.

More generally, the generation of the initial population in a way to optimize a given diversification criterion can be defined as an optimization problem. This problem may be as (or more) difficult as the original problem to solve. This is why in most of the cases, heuristic or metaheuristic algorithms (e.g., greedy algorithms) are used to solve the optimization problem dealing with generation of a diversified population. The

X_1	5	3	2	1	6	4
X_2	6	2	3	5	4	1
X_3	4	5	6	2	1	3
X_4	3	1	4	6	5	2
X_5	2	6	1	4	3	5
X_6	1	4	5	3	2	6

FIGURE 3.5 Deterministic diversification of the initial population of permutations in an assignment problem.

problem can be formulated as follows: given a search space S, generate a population P of n solutions that maximizes its diversity Div(P). The objective function Div(P) may be any diversity measure such as the distribution or the entropy of the solutions in the decision space (see Section 2.2.2.1), discrepancy, and dispersion [581].

Example 3.5 Hybrid initialization approach. A practical hybrid approach may first generate a subpopulation Q of random solutions (e.g., pseudo-random generation). The subpopulation Q has a size k ($k < n$). Then, given the already assigned initial subpopulation Q, the problem is to find a new solution s maximizing the minimum distance min(dist(s, Q)) to solutions of the subpopulation Q. The new solution integrates the population Q and the process is iterated until a complete population of size n is initialized (Fig. 3.6).

3.1.1.4 Heuristic Initialization Any heuristic (e.g., local search) can be used to initialize the population. For instance, as shown for S-metaheuristics, a greedy heuristic may be used to determine the initial solution. Depending on the fitness landscape associated with the optimization problem (e.g., "big valley"), this strategy may be more effective and/or efficient than a random initialization. If this strategy has to be used in a P-metaheuristic, it is obvious to "randomize" the greedy procedure to obtain different solutions from the greedy procedure. The main drawback of this approach is that the initial population may lose its diversity, which will generate a premature convergence and stagnation of the population (Table 3.2).

3.1.2 Stopping Criteria

Many stopping criteria based on the evolution of a population may be used. Some of them are similar to those designed for S-metaheuristics.

- **Static procedure:** In a static procedure, the end of the search may be known *a priori*. For instance, one can use a fixed number of iterations (generations), a limit on CPU resources, or a maximum number of objective function evaluations.
- **Adaptive procedure:** In an adaptive procedure, the end of the search cannot be known *a priori*. One can use a fixed number of iterations (generations) without improvement, when an optimum or a satisfactory solution is reached (e.g., a

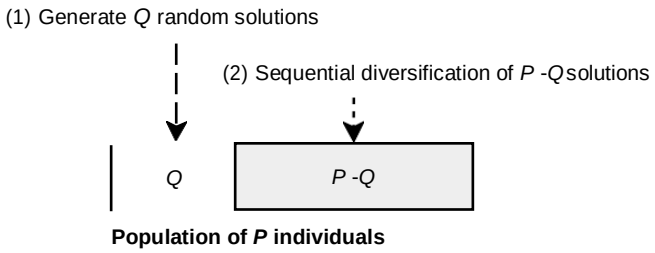


FIGURE 3.6 Hybrid initialization of the population.

given error to the optimum or an approximation to it when a lower bound is known beforehand).

Some stopping criteria are specific to P-metaheuristics. They are generally based on some statistics on the current population or the evolution of the population. Mostly, they are related to the diversity of the population. It consists in stopping the algorithm when the diversity measure falls below a given threshold. This stopping criteria deal with the stagnation of the population. When the population stagnates, keeping the execution of a P-metaheuristic is useless.

3.2 EVOLUTIONARY ALGORITHMS

In the nineteenth century, J. Mendel was the first to state the baselines of heredity from parents to offsprings. Then in 1859, C. Darwin presented the theory of evolution in his famous book *On the Origin of Species* [176]. In the 1980s, these theories of creation of new species and their evolution have inspired computer scientists in designing evolutionary algorithms. Different main schools of evolutionary algorithms have evolved independently during the past 40 years: *genetic algorithms* (GA), mainly developed in Michigan, USA, by J. H. Holland [383,384]; *evolution strategies*, developed in Berlin, Germany, by I. Rechenberg [642,643] and H-P. Schwefel [685,687]; and *evolutionary programming* by L. Fogel in San Diego, USA [272,273]. Later, in the end of 1980s, *genetic programming* has been proposed by J. Koza [480]. Each of these constitutes a different approach; however, they are inspired by the same principles of natural evolution [43]. A good introductory survey can be found in Ref. [45].

Evolutionary algorithms are stochastic P-metaheuristics that have been successfully applied to many real and complex problems (epistatic, multimodal, multiobjective, and highly constrained problems). They are the most studied population-based algorithms. Their success in solving difficult optimization problems in various domains (continuous or combinatorial optimization, system modeling and identification, planning and control, engineering design, data mining and machine learning, artificial life) has promoted the field known as *evolutionary computation* (EC) [45].

EAs are based on the notion of *competition*. They represent a class of iterative optimization algorithms that simulate the evolution of species (Fig. 3.7). They are based

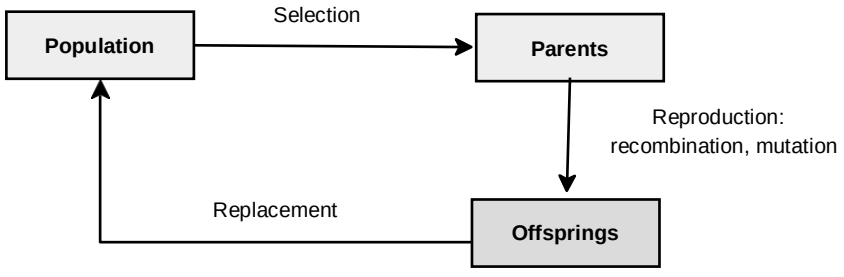


FIGURE 3.7 A generation in evolutionary algorithms.