Course Handouts

Combinatorial optimization and metaheuristic algorithms

Master 1 Artificial Intelligence and Data Science

Dr. Abdelkamel, Ben Ali

benali-abdelkamel@univ-eloued.dz

Computer Science Department

University of El Oued

June 2024

Contents

Abstract

Welcome to the syllabus for the course "Combinatorial Optimization and Metaheuristic Algorithms". This course aims to provide students with a comprehensive understanding of the principles, techniques, and applications of combinatorial optimization and metaheuristic algorithms.

Combinatorial optimization problems (COPs) arise in various real-world scenarios where the goal is to find the best arrangement or combination of elements from a finite set. These problems are often challenging due to their inherent complexity and the exponential growth of possible solutions.

Metaheuristic algorithms offer powerful strategies for solving COPs efficiently. By exploring both single-solution based and population-based metaheuristics, students will learn versatile approaches to address diverse optimization challenges. Furthermore, we will delve into multi-objective optimization, where the goal is to optimize multiple conflicting objectives simultaneously.

Throughout this course, students will not only gain theoretical knowledge but also practical experience through hands-on implementation and analysis of metaheuristics. By the end of the course, students will be equipped with valuable skills to tackle complex COPs and contribute to advancements in various domains.

I hope this syllabus serves as a guide to your learning journey and inspires you to explore the fascinating world of combinatorial optimization and metaheuristics.

Best wishes for a rewarding and enriching learning experience!

Key-words: Combinatorial Optimization - Metaheuristic Algorithms - Multi-objective Problems - Practical Analysis Solution Space Exploration

Abdelkamel Ben Ali

ملخص

مرحبا بكم في المنهج الدراسي لمقرر **تحسين التوافقي والخوارزميات الميتا-استكشافية**.
يهدف هذا المقرر إلى تزويد الطلاب بفهم شامل للمبادئ والتقنيات والتطبيقات المتعلقة بتحسين التوافقي والخوارزميات الميتا-استكشافية.

اتظهر مشكلات تحسين التوافقي ($COPs$) في العديد من السيناريوهات الواقعية حيث يكون الهدف هو العثور على أفضل ترتيب أو تركيب لعناصر من مجموعة محدودة. غالبا ما تكون هذه المشكلات صعبة بسبب تعقيدها الذاتي والنمو الأسي للحلول المحتملة.

توفر الخوارزميات الميتا-استكشافية استراتيجيات قوية لحل مشكلات تحسين التوافقي بكفاءة. من خلال استكشاف الخوارزميات القائمة على الحل الفردي والخوارزميات القائمة على المجموعات، سيتعلم الطلاب أساليب متنوعة للتعامل مع تحديات التحسين المختلفة. بالإضافة إلى ذلك، سنغوص في موضوع التحسين متعدد الأهداف، حيث يكون الهدف تحسين أهداف متعارضة متعددة في الوقت ذاته.

على مدار هذا المقرر، لن يكتسب الطلاب المعرفة النظرية فقط، بل سيكتسبون أيضا خبرة عملية من خلال تنفيذ وتحليل الخوارزميات الميتا-استكشافية عمليا. بنهاية المقرر، سيكون الطلاب مجهزين بمهارات قيمة تمكنهم من مواجهة مشكلات تحسين توافقي معقدة والمساهمة في التطورات في مختلف المجالات.

آمل أن يكون هذا المنهج بمثابة دليل لرحلتكم التعليمية وأن يلهمكم لاستكشاف العالم الرائع للتحسين التوافقي والخوارزميات الميتا-استكشافية.

أطيب التمنيات بتجربة تعليمية مثمرة وممتعة!

الكلمات المفتاحية
تحسين التوافقي - الخوارزميات الميتا-استكشافية - مشكلات متعددة الأهداف - التحليل العملي - البحث في فضاء الحلول

بن علي عبد الكامل
أستاذ محاضر بجامعة الوادي

Chapter 1

Introduction to complexity theory and optimization

1.1 Complexity theory

Complexity theory was introduced to address fundamental questions in theoretical computer science, such as:

- What are the limits of computers?
- What are the fundamental limits that are independent of technology?
- What computational problems are beyond reach?

This theory provides mathematical tools to analyze algorithm performance, and enables us to determine and qualify the intrinsic difficulty of problems. By understanding the complexity of problems, we can assess the computational resources required to solve them and identify those that may be impractical to solve efficiently.

1.1.1 Complexity of algorithms

An algorithm designed to solve a given problem requires two crucial resources when executed on a machine: CPU time and memory space. The study of algorithm complexity corresponds to analyzing the comparative efficiency of algorithms in terms

of their consumption of these computational resources. We refer to *time complexity* (more significant) and *space complexity*.

The evaluation of time complexity can be done experimentally or formally.

- Experimental time complexity entails measuring the CPU time required by an algorithm (implemented on a machine) during its execution. Naturally, the running time is subject to hardware factors and the coding language used.
- The formal approach is independent of the technical characteristics. It entails estimating the cost of an algorithm based on the number of fundamental instructions required to achieve the problem solution. The cost measurement is given in terms of the problem size, denoted by n. In other words, the cost function, denoted by $f(n)$, of an algorithm maps the number of fundamental operations required to solve any problem instance of that size. Time complexity is generally evaluated in the worst-case scenario.

Asymptotic complexity

The goal when determining the time complexity of an algorithm is not to obtain an exact quantity, but an asymptotic bound on the number of fundamental operations. In assessing time complexity, the aim is to provide an estimate of the order of magnitude of the fundamental operations required as the size of the problem increases. Several asymptotic notations are employed to provide bounds on the order of magnitude and compare the growth rates of different cost functions. The asymptotic symbol "O" (read as "big oh") is the most popular.

Definition 1.1 (Big O notation)**.** *An algorithm has time complexity* $f(n) \in O(g(n))$ *if there exist two positive constants* c *and* n_0*, such that* $\forall n > n_0, f(n) \leq c \cdot g(n)$.

We say that the function $f(n)$ is asymptotically bounded (or dominated) by the function $g(n)$. The notation "O" is used to express an upper bound on algorithm complexity.

Polynomial algorithms vs. Exponential algorithms

The distinction between polynomial and exponential algorithms is essential for characterizing the intrinsic difficulty of problems.

Definition 1.2 (Polynomial algorithm)**.** *An algorithm is said to be polynomial time if its complexity is in $O(p(n))$, where $p(n)$ is a polynomial function with respect to the problem size n. A polynomial function of degree $k = 1, 2, \ldots$ can be defined as follows:*

$$p(n) = a_k \cdot n^k + a_{k-1} \cdot n^{k-1} \ldots a_1 \cdot n + a_0$$

where $a_k > 0$. The corresponding algorithm has complexity of $O(n^k)$.

Definition 1.3 (Exponential algorithm)**.** *An algorithm is said to be exponential time if its complexity is in $O(k^n)$, where k is a constant strictly greater than 1.*

Polynomial algorithms are *feasible*, while exponential algorithms are generally *infeasible*.

1.1.2 Complexity of problems

The complexity of a problem is equivalent to the complexity of the best-known algorithm (in terms of execution speed) capable of solving it. Problems are thus distinguished into *tractable* (or relatively easy to solve), for which polynomial algorithms exist, and *intractable* (or difficult to solve), for which no polynomial algorithms exist. [1].

$\mathcal{P}$ vs. $\mathcal{NP}$

The complexity theory concerns decision problems[2]. An important aspect of complexity theory is categorizing problems into complexity classes based on their time

[1] It is essential to distinguish between problems that become intractable for large sizes due to the exponential growth of their time complexity, and problems that are algorithmically unsolvable. The latter, known as *undecidable* problems, could never have any algorithm to solve them, even with unlimited computing resources. The most famous example of undecidable problems is the halting problem (Turing, 1937).

[2] A decision problem is one whose answer is "yes" or "no".

complexity. There are two major classes of decision problems: the $\mathcal{P}$ class (*Polynomial*) and the $\mathcal{NP}$ class (*Nondeterministic Polynomial*).

- The $\mathcal{P}$ class represents all decision problems that can be solved by a deterministic machine[3] in polynomial time. Thus, this class corresponds to the set of problems for which polynomial-time algorithms exist to solve them.
- The $\mathcal{NP}$ class contains all decision problems that can be solved by a nondeterministic machine[4] in polynomial time, i.e., the validity of a proposed solution (certificate) can be tested in polynomial time relative to the problem size.

The question of whether "$\mathcal{P} = \mathcal{NP}$" is central and open in complexity theory. It became prominent in 1971 with Cook's paper (Cook, 1971). Answering this question entails determining if finding a solution is as efficient as verifying its validity. In other words, a positive answer implies that all decision problems in $\mathcal{NP}$ are also in $\mathcal{P}$. Of course, $\mathcal{NP}$ encompasses $\mathcal{P}$, meaning for every decision problem in $\mathcal{P}$, there exists a nondeterministic algorithm solving it. Thus, $\mathcal{P} \subseteq \mathcal{NP}$. However, the conjecture $\mathcal{NP} \subset \mathcal{P}$ remains an open question, persisting for several decades.

$\mathcal{NP}$-completeness

Another fundamental question in complexity theory is whether an $\mathcal{NP}$ problem is harder than all others. Such problems define the $\mathcal{NP}$-complete class.

Definition 1.4 ($\mathcal{NP}$-complete)**.** *A decision problem* Π *is called* $\mathcal{NP}$*-complete if it belongs to the* $\mathcal{NP}$ *class, and all problems in* $\mathcal{NP}$ *can be reduced to* Π *in polynomial time.*

A decision problem Π *reduces* to another decision problem Π' if there exists a polynomial-time algorithm that transforms any input u of Π into an input u' of Π', such that u corresponds to a positive instance (whose answer is "yes") of Π if and only if u' is a positive instance of Π'.

[3] In complexity theory, Turing machines and Random Access Machines serve as benchmarks for measuring time and space complexities.

[4] A nondeterministic machine, unlike a deterministic one, has multiple possible transitions from which several different continuations are possible without any specification of which transition will be taken.

$\mathcal{NP}$-hardness

The $\mathcal{NP}$-complete class refers to the most challenging problems in $\mathcal{NP}$, for which polynomial algorithms are never found to solve them optimally with a deterministic machine. $\mathcal{NP}$-complete problems are equivalent by polynomial reduction, implying the following interesting property: if a polynomial algorithm were found to solve any $\mathcal{NP}$-complete problem, then polynomial algorithms could be derived for all other problems in this class, and thus we could conclude that $\mathcal{P} = \mathcal{NP}$ (Solnon, 2005).

Definition 1.5 ($\mathcal{NP}$-hard)**.** *$\mathcal{NP}$-hard problems are optimization problems whose associated decision problems are $\mathcal{NP}$-complete.*

Indeed, most real optimization problems are $\mathcal{NP}$-hard and do not admit efficient algorithms for solving large instances. Finding exact (optimal) solutions to these problems requires exponential time. Metaheuristic algorithms provide an important alternative for finding approximate (near optimal) solutions for $\mathcal{NP}$-hard problems within practical time constraints.

1.2 Optimization

1.2.1 Definition of optimization

Optimization refers to the systematic process of finding a better solution (or a set of better solutions) within a huge set of feasible alternatives. A solution corresponds to a feasible configuration of the variables of the optimization problem at hand. The set of all possible solutions corresponds to the *search space*. The optimality of solutions is defined in terms of minimizing (or maximizing) a function quantifying the cost (or quality) of a solution. This function is called the *cost function*, *objective function*, or *fitness*. Optimization problems arise in various fields, including engineering, bioinformatics, economics, operations research, and computer science.

In mathematical terms, optimization involves finding the values of n variables $\mathbf{x} = (x_1, x_2, \ldots, x_n)$ that optimize the fitness function $f(\mathbf{x})$ while satisfying certain constraints. These variables could represent decision variables, parameters, or design

variables, depending on the nature of the problem. A minimization optimization problem[5] can be formulated as (Nocedal and Wright, 1999):

$$\begin{aligned} \text{Minimize} \quad & f(\mathbf{x}) && (1.1)\\ \text{subject to} \quad & g_j(\mathbf{x}) \leq 0, \quad \text{for } j = 1, 2, \ldots, p && (1.2)\\ & h_k(\mathbf{x}) \neq 0, \quad \text{for } k = 1, 2, \ldots, q && (1.3)\\ & x_i \in D_i = [x_i^{\min}, x_i^{\max}], i = 1, 2, \ldots, n && (1.4) \end{aligned}$$

where $f(\mathbf{x})$ represents the objective function to be minimized, g_i $(j = 1, 2, \ldots, p)$ correspond to the inequality constraints, and h_j $(k = 1, 2, \ldots, q)$ to the equality constraints. The variables $\mathbf{x} = (x_1, x_2, \ldots, x_n)$ are often subject to certain bounds or domain restrictions.

1.2.2 Importance of optimization

Optimization techniques play a crucial role in improving efficiency, reducing costs, and enhancing decision-making processes across a wide range of industries and applications.

Consider transportation logistics, where optimization methods help determine the most efficient routes for delivery trucks, minimizing fuel consumption and transportation time. In manufacturing, optimization methods are used to optimize production schedules, minimize downtime, and maximize resource utilization. In finance, portfolio optimization helps investors allocate assets to maximize returns while managing risk.

Optimization plays a pivotal role in streamlining processes, optimizing resources, and improving systems. By leveraging optimization techniques, organizations can achieve significant cost savings, enhance productivity, and gain a competitive advantage in the market.

Let's explore the significance of optimization in the field of bio-informatics. In this

[5]Note that searching for solutions that minimize the objective function is equivalent to searching for solutions that maximize the inverse of the objective function: $\min\{f(\mathbf{x})\} = \max\{-f(\mathbf{x})\}$. For this reason, we will henceforth, without loss of generality, refer to minimization problems only.

field, optimization techniques are employed in various applications, such as sequence alignment, protein structure prediction, and drug discovery (Blum and Festa, 2016, Calvet et al., 2023).

- For instance, in sequence alignment, optimization algorithms are used to align DNA, RNA, or protein sequences. This process helps identify similarities and differences between sequences, which is crucial for understanding evolutionary relationships, identifying functional regions, and predicting protein structures.

- Additionally, optimization methods are applied in protein structure prediction, where the goal is to predict the three-dimensional structure of a protein based on its amino acid sequence. This involves optimizing complex energy functions to find the most stable and biologically relevant protein conformation.

- Moreover, in drug discovery, optimization techniques are utilized to design and optimize small molecule compounds with desired pharmacological properties. Optimization algorithms help identify potential drug candidates with optimal binding affinity, specificity, and bio-availability, thereby accelerating the drug development process.

By leveraging optimization methods in bio-informatics, researchers can accelerate scientific discovery, gain insights into biological systems, and develop novel therapies for various diseases.

1.2.3 Types of optimization problems

Optimization problems can be classified into various categories based on the nature of the objective function, constraints, and decision variables involved. Some common types of optimization problems include (Hillier and Lieberman, 2015):

- **Linear Programming (LP):** Involves optimizing a linear objective function subject to linear equality and inequality constraints.

- **Integer Programming (IP):** Similar to linear programming but with the additional constraint that decision variables must take integer values.

- **Nonlinear Programming (NLP):** Deals with optimizing nonlinear objective functions subject to nonlinear constraints.
- **Combinatorial Optimization (CO):** Focuses on optimizing discrete decision variables, often encountered in problems such as scheduling, routing, and network design.

Each type of optimization problem presents its own set of challenges and requires specialized techniques for resolution.

1.2.4 Challenges in optimization

While optimization offers immense potential for improving decision-making and efficiency, it also presents several challenges. Some of the key challenges include (Hillier and Lieberman, 2015):

- **Computational complexity:** Many optimization problems are computationally challenging and may require significant computational resources to solve, especially for large-scale problems.
- **Multi-objective optimization:** In real-world scenarios, optimization often involves multiple conflicting objectives that need to be optimized simultaneously, leading to trade-offs and complexities.
- **Constraint handling:** Optimization problems often come with constraints that must be satisfied, adding another layer of complexity to the problem-solving process.
- **Dynamic environments:** In dynamic environments, where parameters and constraints change over time, traditional optimization techniques may struggle to adapt and find optimal solutions.

Overcoming these challenges requires innovative approaches, advanced algorithms, and interdisciplinary collaboration.

Chapter 2

Combinatorial optimization

The chapter explores optimization problems that involve discrete decision variables and combinatorial structures. Combinatorial optimization problems (COPs) arise across various domains, including logistics, manufacturing, telecommunications, and computer science, where they play a crucial role in decision-making and resource allocation.

COPS involve finding the best arrangement or selection of elements from a finite set to optimize an objective function while satisfying a set of constraints. These problems are characterized by their discrete nature, combinatorial complexity, and the absence of a straightforward mathematical formulation. Instead, they often require the exploration of a vast search space to identify optimal or near-optimal solutions.

Throughout this chapter, we delve into the fundamental concepts and notions underlying COPS. We explore key COPs, such as the Binary Knapsack Problem (BKP), Traveling Salesman Problem (TSP), Job Scheduling Problem (JSP), Quadratic Assignment Problem (QAP), Linear Ordering Problem (LOP), and Graph Coloring Problem (GCP). Each problem presents unique challenges and requires tailored solution approaches to effectively address them.

2.1 Definitions

Optimization problems naturally fall into two main categories: those with real variables and those with discrete variables. In discrete optimization, there exists a

sub-category known as *combinatorial optimization*. These problems are characterized by enumerable spaces of possible solutions. Solutions to COPs can take various forms, such as integers, bit strings, integer vectors, graph structures, or combinations of these representations within complex structures.

In a highly formal manner, a COP with a single-objective can be defined as follows (inspired by (Blum and Roli, 2003)):

Definition 2.1 (Combinatorial optimization problem, COP)**.** *A COP* $(\mathcal{S}, f)$ *is defined by:*

- n *decision variables* $\mathbf{x} = (x_1, x_2, \ldots, x_n) \in \mathbb{Z}^n$*;*
- *the range of each decision variable (explicit bounds):* $x_i \in D_i = [x_i^{\min}, x_i^{\max}]$ *for* $i = 1, 2, \ldots, n$*;*
- *an objective function* f *to minimize, where* $f : D_1 \times D_2 \times \ldots \times D_n \to \mathbb{R}$*;*

The search space (also called parameter space, configuration space, or environment) corresponds to the set of values that can be taken by the decision variables:

$$\Omega = \{s = ((x_1, v_1), (x_2, v_2), \ldots, (x_n, v_n)) \mid v_i \in D_i\}. \tag{2.1}$$

The set of all feasible solutions (also called the feasible space or the decision space or the solution space) is:

$$\mathcal{S} = \{s \in \Omega \mid s\text{satisfies all problem constraints}\}. \tag{2.2}$$

The set $\mathcal{S}$ can describe a combinatorial structure, such as spanning trees of a graph, paths, matching, orderings, bit strings, etc.

Solving the problem $(\mathcal{S}, f)$ consists of finding a solution $s^* \in \mathcal{S}$ such that $f(s^*)$ is minimal.

$$s^* = \arg\min\{f(s) \mid s \in \mathcal{S}\}. \tag{2.3}$$

The objective function f defines a total order relation among the feasible solutions of $\mathcal{S}$. The best possible solution s^* is called the ***global optimum*** or the ***optimal***

solution of the problem [1].

Definition 2.2 (Global optimum)**.** *A solution $s^* \in \mathcal{S}$ is globally optimal if $\forall s \in \mathcal{S}$: $f(s^*) \leq f(s)$. We refer to it as a strict global optimum if $f(s^*) < f(s)$ $\forall s \in \mathcal{S}$.*

The solution that is locally better within a restricted region of the search space is called a ***local optimum*** or a ***suboptimal solution***. Figure 2.1 illustrates the concepts of global optimum and local optimum.

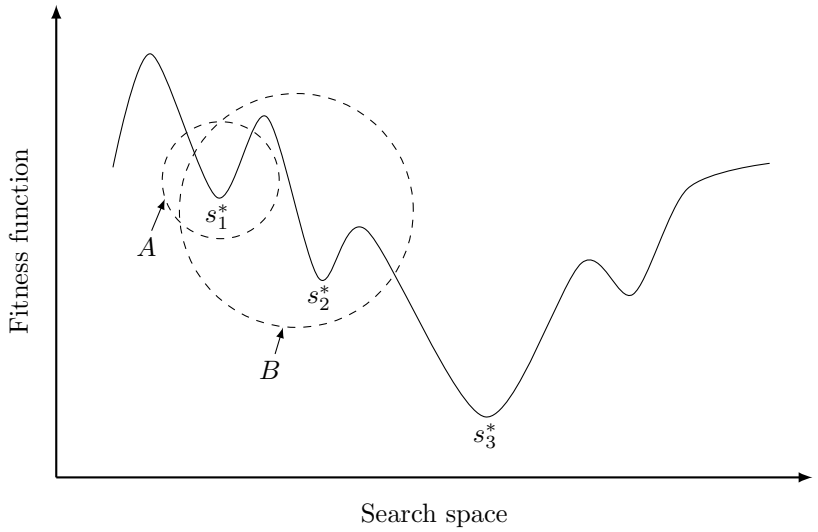

Figure 2.1: Illustration of global optimum and a local optimum. s_1^* is a local optimum within solution region A; s_2^* is a local optimum within solution region B; s_3^* is a global optimum.

The definition of the notion of local optima requires that the solution space $\mathcal{S}$ be topological, i.e., on which we can define neighborhood relations between solutions. The ***neighborhood*** of a solution encompasses neighboring solutions that can be generated by making small changes to this solution. More formally, the notion of neighborhood is defined as follows (inspired by (Blum and Roli, 2003)).

Definition 2.3 (Neighborhood structure)**.** *A neighborhood structure (or relation) is a function $\mathcal{N} : \mathcal{S} \rightarrow 2^{|\mathcal{S}|}$ that assigns to each solution $s \in \mathcal{S}$ a set of neighbors $\mathcal{N}(s) \subseteq \mathcal{S}$. $\mathcal{N}(s)$ is called the neighborhood of s.*

[1] s^* is also referred to as the minimum (maximum) if the problem is stated in terms of minimizing (maximizing) an objective function.

As already mentioned, neighborhood structures are most often implicitly defined by specifying the changes or mutations that need to be applied to a solution to generate all its neighbors. Applying such an operator to produce a neighboring solution $s' \in \mathcal{N}(s)$ from a solution s is commonly referred to as a move.

The notion of local optimum relative to a neighborhood can be formally defined as follows.

Definition 2.4 (Local optimum)**.** *A solution $\hat{s} \in \mathcal{S}$ is locally optimal relative to a neighborhood relation $\mathcal{N}$ if $\forall s \in \mathcal{N}(\hat{s}) \subset \mathcal{S}$: $f(\hat{s}) \leq f(s)$. We refer to it as a strict local optimum if $f(\hat{s}) < f(s)$ $\forall s \in \mathcal{N}(\hat{s})$.*

Note that any global optimum is obviously also a local optimum relative to any neighborhood relation.

2.2 Two main types of COPs

Two main families of COPs are generally distinguished: *permutation problem*s, whose solutions can be encoded with permutations of elements, and *binary problems*, whose solutions can be given by binary structures.

Examples of permutation COPs include the Traveling Salesman Problem (TSP). This problem can be encoded with a permutation of integers in the set $\{1, 2, \ldots, n\}$, where n is the number of cities to visit. Each permuted element is a city, while the order of the elements in the permutation directly indicates the order in which the cities are visited. For a symmetric TSP with a specific departure city, $|\mathcal{S}| = (n-1)!/2$ corresponds to the number of all possible routes.

Examples of binary optimization problems include the Binary Knapsack Problem (BKP). The problem can be encoded as a binary vector of length n, where n is the number of objects. The i-th bit in the vector indicates the state of object o_i; 0 means that this object is selected, 1 means it is discarded. The size of the search space for the BKS is $|\mathcal{S}| = 2^n$.

An instance of a COP is obtained by assigning numerical values to all parameters. In other words, an instance refers to a specific concrete instantiation of the problem

with fully specified data. For example, an instance of the TSP is a list of cities (a list of points on the plane), while an instance of the BKP corresponds to a set of objects (weights and values) along with a knapsack capacity.

2.3 Some COPs

In this section, we will explore some COPs, namely the Binary Knapsack Problem (BKP), Traveling Salesman Problem (TSP), Job Scheduling Problem (JSP), Quadratic Assignment Problem (QAP), Linear Ordering Problem (LOP), and Graph Coloring Problem (GCP). Through this exploration, our aim is to gain insight into the combinatorial structure of each problem and understand its inherent complexity.

2.3.1 Binary Knapsack Problem (BKP)

The BKP presents the task of selecting items to maximize the total value while not exceeding the capacity of a knapsack.

- **Problem:** Given a set of n objects, each with a value v_i and weight w_i, and a knapsack with a capacity W, the objective is to select a subset of objects to maximize the total value while ensuring that the total weight does not exceed W.

- **Search space size:** The search space size of the BKP can be expressed as 2^n. This is because each object can be either included or excluded from the knapsack, resulting in two possibilities for each item.

- **Encoding:** A potential solution of the BKS (a subset of items) can be encoded with a binary string of length n, denoted as $\mathbf{x} = (x_1, x_2, ..., x_n)$, where each bit x_i represents whether object i is included (1) or excluded (0) from the knapsack.

- **Fitness:** Solution fitness is determined by evaluating the total value of the selected items. Given a bit string $\mathbf{x}$, the fitness $f(\mathbf{x})$ is calculated as the sum of the values of the selected objects, subject to the constraint that the total weight

does not exceed W.

$$f(\mathbf{x}) = \sum_{i=1}^{n} v_i x_i \tag{2.4}$$

$$\text{subject to} \sum_{i=1}^{n} w_i x_i \leq W \tag{2.5}$$

2.3.2 Traveling Salesman Problem (TSP)

The TSP can be formally expressed as follows:

- **Problem:** Let $G = (V, E)$ be an undirected complete graph where V is a set of n cities and E is the set of edges between cities. Each edge e_{ij} is associated with a non-negative distance d_{ij} representing the (euclidean) distance between city i and city j. The goal is to find a Hamiltonian cycle (that visits each vertex exactly once) in G such that the total distance of the cycle is minimized.

- **Search space size:** The total number of possible paths is determined by the factorial of the number of cities, denoted as $n!$, which grows exponentially with the number of cities. For 58 cities, the number of candidate paths exceeds 5×10^{78}, which is approximately the number of atoms in the known universe.

- **Encoding:** The potential solutions (Hamiltonian cycles) are naturally encoded as permutations of the cities, denoted as $\pi = (\pi_1, \pi_2, ..., \pi_n)$, where π_i gives the i-th city visited in the cycle.

- **Fitness:** Solution quality is determined by evaluating the total distance of the Hamiltonian cycle. Given a permutation π, the fitness $f(\pi)$ is calculated as the sum of distances along the edges of the cycle, i.e.,

$$f(\pi) = \sum_{i=1}^{n} d_{\pi_i \pi_{i+1}} + d_{\pi_n \pi_1}. \tag{2.6}$$

  The goal is to find the permutation π that minimizes $f(\pi)$.

2.3.3 Job Scheduling Problem (JSP)

The JSP poses the challenge of scheduling a set of jobs on machines to optimize a certain objective, such as minimizing the total completion time or maximizing machine utilization.

- **Problem:** Given a set of n jobs, each with a processing time p_{ij} indicating the time required to process job i on machine j, the objective is to schedule these jobs on a set of m machines in a way that optimizes a specific criterion.
- **Search space size:** The search space size of JSP can vary depending on the problem instance and the scheduling rule used. However, it can be expressed as $n!$, representing all possible permutations of the n jobs.
- **Encoding:** The natural encoding of the JSP is the permutation, representing the order in which the jobs are processed on each machine. Each job is assigned a specific position in the permutation, representing its scheduled position in the sequence of jobs processed on each machine.
- **Fitness:** JSP solution fitness is determined by evaluating a specific criterion, such as the total completion time, makespan (total time to complete all jobs), or machine utilization. Given a permutation $\pi = (\pi_1, \pi_2, ..., \pi_n)$ representing a potential schedule, the fitness value $f(\pi)$ is calculated based on the chosen objective function. For example, for minimizing the makespan, the completion time C of the last job π_n on the last machine $j = m$:

$$f(\pi) = C_{\pi_n,m}. \tag{2.7}$$

The completion time $c_{\pi_i,j}$ for job i on machine j is given by:

$$C_{\pi_i,j} = \begin{cases} p_{\pi_i,j}, & i = j = 1 \\ p_{\pi_i,j} + C_{\pi_{i-1},j}, & i > 1, j = 1 \\ p_{\pi_i,j} + C_{\pi_i,j-1}, & i = 1, j > 1 \\ p_{\pi_i,j} + \max(C_{\pi_{i-1},j}, C_{\pi_i,j-1}), & i > 1, j > 1 \end{cases} \tag{2.8}$$

2.3.4 Quadratic Assignment Problem (QAP)

The QAP is the task of assigning a set of facilities to a set of locations in such a way as to minimize the total cost, which is the sum of the costs associated with each pair of facilities multiplied by the distance between their assigned locations.

- **Problem:** Given n facilities and n locations, along with matrices representing the flow between facilities and the distances between locations, the objective is to find an assignment of facilities to locations that minimizes the total cost.
- **Search space size:** The search space size can be expressed as $n!$, representing all possible permutations of the n facilities.
- **Encoding:** A potential solution of the QAP (facility assignment) can be represented by a permutation of the facilities, where each facility is assigned to a specific location. The i-th permuted element in the set $\{1, 2, \ldots, n\}$ gives the location to which the i-th facility is assigned.
- **Fitness:** QAP solution fitness is determined by evaluating the total cost of the assignment. Given a permutation π representing a potential solution, the fitness $f(\pi)$ is calculated as the sum of the costs associated with each pair of facilities multiplied by the distance between their assigned locations:

$$f(\pi) = \sum_{i=1}^{n} \sum_{j=1}^{n} h_{ij} \cdot d_{\pi(i)\pi(j)}, \tag{2.9}$$

  where h_{ij} represents the flow between facilities i and j, and $d_{\pi(i)\pi(j)}$ is the distance between the locations to which facilities i and j are assigned.

2.3.5 Linear Ordering Problem (LOP)

The LOP is to find an optimal permutation of a set of items that minimizes or maximizes a certain objective, such as the total distance or cost associated with the order.

- **Problem:** Given a set of n items and a square matrix $H = [h_{ij}]$ representing the pairwise distances or costs between items, the objective is to find a linear order of the items that optimizes the specified objective.

- **Search space size:** The search space size is $n!$, representing all possible orderings of the n items.

- **Encoding:** The LOP is naturally encoded as a permutation of the items.

- **Fitness:** Fitness is determined by evaluating the objective function associated with the specified optimization criterion. Given a permutation π representing a potential solution, the solution fitness $f(\pi)$ is calculated based on the objective function:

$$f(\pi) = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} h_{\pi_i \pi_j}, \tag{2.10}$$

2.3.6 Graph Coloring Problem (GCP)

The GCP in the task of coloring the vertices of a graph in such a way that no two adjacent vertices share the same color.

- **Problem:** Given an undirected graph $G = (V, E)$ with a set of vertices V and a set of edges E, the objective is to assign a color to each vertex in V such that no two adjacent vertices share the same color.

- **Search space size:** The number of all possible color configurations is exponential in the number of vertices. It can be expressed as k^n, where k is the number of available colors and $n = |V|$ is the number of vertices.

- **Encoding:** An GCP potential solution (vertex coloring) can be encoded as an integer vector of size n, where each element represents the color assigned to the corresponding vertex. Each color is identified by an integer value in the set $\{1, 2, \ldots, k\}$.

- **Fitness:** The fitness value is determined by evaluating whether the coloring satisfies the constraint that no two adjacent vertices share the same color. Given

a vertex coloring $\mathbf{c}$, the fitness $f(\mathbf{c})$ is calculated as follows:

$$f(\mathbf{c}) = \begin{cases} 1, & \text{if the coloring is valid (no adjacent vertices share the same color)} \\ 0, & \text{otherwise} \end{cases} \tag{2.11}$$

This fitness function assigns a value of 1 if the vertex coloring is valid and 0 otherwise, indicating whether the solution meets the constraints of the GCP.

2.4 Solving COPs with metaheuristics

COPs are often very challenging to solve exactly in terms of computation time. For many practical problems, the main difficulty arises from the combinatorial explosion of possible solutions to explore, i.e., the search space size is exponential with the problem size. Theoretical studies, as discussed in Section 1.1.2, classify these problems as $\mathcal{NP}$-hard, meaning no exact algorithms are known to solve them in polynomial time. The famous $\mathcal{P} \neq \mathcal{NP}$ conjecture suggests that such algorithms are unlikely to exist.

The difficulty in solving a COP may also be related to a complex structure of the search space and costly evaluation of the objective function. These difficulties rule out exhaustive enumeration as a method for finding the solution. Hence, a wide variety of mathematical approaches and algorithms have been proposed to effectively solve $\mathcal{NP}$-hard COPs.

Optimization methods can be broadly categorized into two main families: exact methods and approximate methods. Exact methods guarantee finding optimal solutions for small-sized instances in limited time but quickly become impractical for large instances. They rely on dynamic programming or use implicit enumeration techniques such as Branch & Bound methods. They require exponential time in the worst case. Approximate methods provide a very interesting alternative when facing complex problems of large size. Unlike exact methods, approximate methods provide "good-quality" (approximate) solutions in a *reasonable* computation time.

Metaheuristics[2] are general approximate methods that can be adapted and applied

[2]The term "metaheuristic" is derived from the composition of two Greek words. The word "heuris-

to different optimization problems. There are two types of metaheuristics: single-solution based metaheuristics, also known as local search methods, and population-based metaheuristics. In the next two chapters, we provide a simplified description of the most important metaheuristics from each class. For a detailed presentation of these metaheuristics (and others), see, for example, (Blum and Roli, 2003, Boussaïd et al., 2013, Gendreau and Potvin, 2010, Talbi, 2009).

Two fundamental concepts in the development of metaheuristic method are the notions of *diversification* (or *exploration*) and *intensification* (or *exploitation*). Diversification entails effectively sampling different zones of the solution space, while intensification focuses the search within a specific zone to find a local optimum. Therefore, metaheuristic algorithms need to maintain a suitable balance between intensification and diversification in their stochastic search strategies.

tic" is derived from the ancient Greek word "heuriskein," meaning "to discover," while the suffix "meta," also from an ancient Greek word, means "beyond" or "at a higher level."

Chapter 3

Single-solution based metaheuristics

Single-solution-based metaheuristics (SS-metaheuristics) focus on iteratively improving a single solution to an optimization problem. Unlike population-based metaheuristics (P-metaheuristics), which maintain multiple solutions, SS-metaheuristics efficiently explore complex combinatorial search spaces with limited computational resources.

In this chapter, we first discuss heuristic construction and local search methods. Then, we explore various SS-metaheuristic algorithms, including Simulated Annealing (SA), Iterated Local Search (ILS), Variable Neighborhood Search (VNS), Tabu Search (TS), and Greedy Randomized Adaptive Search Procedure (GRASP). These SS-metaheuristics offer powerful tools for solving different COPs.

3.1 Heuristic construction

Heuristic construction, also known as greedy construction, is a simple approach to generate initial solutions for COPs. Greedy or constructive methods start with an empty solution and build a solution by assigning values to a decision variable at each construction step until a complete solution is generated. Greedy heuristics are typically deterministic algorithms. They are straightforward to design and generally have reduced complexity compared to iterative local search or SS-metaheuristc algorithms.

Two main design considerations for greedy construction are:

- **Definition of the set of elements:** It is necessary to identify a solution as a set of elements and partial solutions as subsets of elements. For a COP where a solution can be defined by the presence or absence of a finite set of elements $E = \{e_1, e_2, \ldots, e_n\}$, the objective function can be defined as $f : 2^E \to \mathbb{R}$, and the search space is defined as $\mathcal{S} \subset 2^E$. A partial solution s can be considered as a subset $\{e_1^s, e_2^s, \ldots, e_k^s\}$ from the set E. The set defining the initial solution is empty.

- **Element selection heuristic:** At each step, a specific heuristic (tailored to the problem at hand) is used to select the next element to be included in the solution under construction. Typically, this heuristic chooses the best element from the current candidate list in terms of its contribution to locally minimizing the fitness function. Thus, the selection heuristic calculates the profit or gain for each element. Once an element $e_i \in E$ is selected and added to the solution, it is never replaced by another element (There is no backtracking on decisions already made). The selection can be *static* or *dynamic*. In a static selection, the profits/gains associated with the elements do not change during the construction process, while in a dynamic selection, they are updated at each construction step.

Here is a simple pseudocode algorithm for greedy construction:

3.1.1 Greedy algorithm for TSP

A greedy heuristic for TSP may work as follows:

- The set E of elements is the set of edges.

- The set $\mathcal{S}$ of feasible solutions represents all subsets of 2^E that form Hamiltonian cycles.

- A solution is considered as a set of edges.

- A heuristic that can be used to select the next edge may be based on distance. A possible local heuristic is to select the nearest neighbor.

Algorithm 3.1: Greedy solution construction

```
1 s ← {} {Initialize an empty solution}
2 repeat
      {Select the next element based on a greedy rule, from the set
       E minus the elements already selected}
3     e ← SpecificHeuristic(E \ {e/e ∈ s})
      {Check the feasibility of the solution}
4     if (s ∪ e_i ∈ S) then
          {Add the selected element to the solution}
5         s ← s ∪ e;
6     end
7 until a complte solution is formed;
8 return s;
```

- This greedy heuristic is static; i.e., edge distances are not updated during the construction process.

Algorithm 3.2 shows the pseudocode for the nearest neighbor heuristic algorithm for TSP.

Algorithm 3.2: Nearest neighbor algorithm for TSP

```
   Input: Graph G(E, V)
   Output: Tour T visiting all cities
   {Start from an arbitrary city}
1  start_city ← irand(1..n);
2  T ← T ∪ {start_city};
3  Mark all cities as unvisited;
4  repeat
       {Select the nearest unvisited city as the next destination}
5      city ← SelectNearestCity(T);
       {Add the selected city to the tour}
6      T ← T ∪ {city};
7  until a complete tour constructed;
8  Return to the starting city start_city to complete the tour;
9  return T;
```

This greedy algorithm starts from an arbitrary city and iteratively selects the nearest unvisited city as the next destination until all cities have been visited. It then returns to the starting city to complete the tour. The time complexity of this algorithm

depends on the method used to find the nearest unvisited city but is typically $O(n^2)$.

An illustrative example Suppose we have 5 cities labeled A, B, C, D, and E, and the -symmetric- distances between them are as follows:

	A	B	C	D	E
A	0	10	15	20	25
B	10	0	12	18	22
C	15	12	0	8	16
D	20	18	8	0	14
E	25	22	16	14	0

Now, find the shortest route that visits each city exactly once and returns to the starting city.

1. Start at city A.
2. From city A, find the nearest city that has not been visited yet. In this case, it is city B ($distance = 10$).
3. Move to city B.
4. From city B, find the nearest unvisited city. It is city C ($distance = 12$).
5. Move to city C.
6. From city C, the nearest unvisited city is city D ($distance = 8$).
7. Move to city D.
8. From city D, the nearest unvisited city is city E ($distance = 14$).
9. Move to city E.
10. Finally, return to the starting city A to complete the tour.

The resulting tour is: $A \to B \to C \to D \to E \to A$. It has a total distance of $10 + 12 + 8 + 14 + 25 = 69$.

3.1.2 Greedy algorithm for BKP

A greedy construction algorithm for the BKP may work as follows:

- The set E is the set of objects to be packed.
- The set $\mathcal{S}$ represents all subsets of E that are feasible solutions (that do not exceed the maximum capacity of the knapsack).
- A usable local heuristic is to choose the object maximizing the ratio $r_i = v_i/w_i$ where v_i (resp. w_i) represents the value (resp. weight) of object i.

Algorithm 3.3: Greedy algorithm for BKP

Input: Items with weights w_i and values v_i, knapsack capacity W

Output: Subset of items to include in the knapsack

```
1  Sort items in descending order of value-to-weight ratio r_i = v_i/w_i;
2  Initialize an empty knapsack S;
3  Initialize current weight weight of the knapsack to 0;
4  for each object i in sorted order do
5      if weight + w_i ≤ W then
6          Add item i to S;
7          weight ← weight + w_i;
8      end
9  end
10 return S;
```

An illustrative example Let's illustrate the greedy construction for a BKP problem with a capacity of 20 and eight objects:

1. Object 1: $w_1 = 2$, $v_1 = 10$
2. Object 2: $w_2 = 5$, $v_2 = 20$
3. Object 3: $w_3 = 3$, $v_3 = 15$
4. Object 4: $w_4 = 7$, $v_4 = 25$
5. Object 5: $w_5 = 1$, $v_5 = 8$

6. Object 6: $w_6 = 4$, $v_6 = 12$
7. Object 7: $w_7 = 6$, $v_7 = 18$
8. Object 8: $w_8 = 9$, $v_8 = 30$

Calculate the value-to-weight ratio for each Object:

- Object 1: $r_1 = 10/2 = 5.00$
- Object 2: $r_2 = 20/5 = 4.00$
- Object 3: $r_3 = 15/3 = 5.00$
- Object 4: $r_4 = 25/7 = 3.57$
- Object 5: $r_5 = 8/1 = 8.00$
- Object 6: $r_6 = 12/4 = 3.00$
- Object 7: $r_7 = 18/6 = 3.00$
- Object 8: $r_8 = 30/9 = 3.33$

Sorting the items based on their value-to-weight ratio in descending order:

1. Object 5 ($v_5 = 8, w_5 = 1, r_5 = 8.00$)
2. Object 1 ($v_1 = 10, w_1 = 2, r_1 = 5.00$)
3. Object 3 ($v_3 = 15, w_3 = 3, r_3 = 5.00$)
4. Object 2 ($v_2 = 20, w_2 = 5, r_2 = 4.00$)
5. Object 4 ($v_4 = 25, w_4 = 7, r_4 = 3.57$)
6. Object 8 ($v_8 = 30, w_8 = 9, r_8 = 3.33$)
7. Object 6 ($v_6 = 12, w_6 = 4, r_6 = 3.00$)
8. Object 7 ($v_7 = 18, w_7 = 6, r_7 = 3.00$)

Now, add objects to the knapsack one by one until we reach the maximum capacity of 20:

- Add Object 5: Total Value = 8, Remaining Capacity = 19
- Add Object 1: Total Value = 18, Remaining Capacity = 17
- Add Object 3: Total Value = 33, Remaining Capacity = 14

- Add Object 2: Total Value = 53, Remaining Capacity = 9
- Add Object 4: Total Value = 78, Remaining Capacity = 2

At this point, we only have 2 unit of capacity left, but no other Object can fit. Therefore, the knapsack is now full. The total value of the objects in the knapsack is 78.

3.2 Local search

3.2.1 General principle

The general idea of local search is to initiate a search process with a single initial solution and then improve its fitness through successive iterations by moving within a predefined neighborhood.

Given a neighborhood relation $\mathcal{N}$, a simple local search method is defined as follows (see Algorithm 3.4). Initially, an initial solution is generated either randomly or using a greedy heuristic; function `GenerateInitialSolution()`. Then, at each iteration, a solution $s' \in \mathcal{N}(s)$ better than s is selected; function `SelectImprovedNeighbor(`$\mathcal{N}(s)$`)`. This new improved solution replaces the current solution, and a new local search iteration begins. The search continues until no further improvement is possible. This process corresponds to the simplest local search method, known as the *hill climbing* in the context of maximization problems.

Algorithm 3.4: Local search procedure

```
1 s ← GenerateInitialSolution()
2 while A local optimum relative to N is not reached do
3 |   s ← SelectImprovedNeighbor(N(s))
4 end
5 return s
```

3.2.2 Improvement strategies

There are mainly two strategies to implement function `SelectImprovedNeighbor(`$\mathcal{N}(s)$`)`:

- The first strategy, called "*best-improvement*", is to choose the best neighbor:

$$s' = \arg\min\{f(s'') \mid s'' \in \mathcal{N}(s)\}.$$

  This requires examining and evaluating the fitness values of all possible neighbors.

- The second strategy, called "*first-improvement*", is to examine the neighbors one by one and return the first neighbor that improves solution fitness. This search policy is used when the neighborhood size is very large and/or evaluating all neighbors is very time-consuming.

3.2.3 Time complexity of local search

The time complexity of a local search procedure is determined by several factors.

- Firstly, it depends on the size of the neighborhood within which the search moves. A larger neighborhood typically implies more potential solutions to explore, which can increase the time complexity.

- Secondly, the time complexity is influenced by the efficiency of the move operation applied to generate neighboring solutions. The efficiency of this operation can vary based on the problem domain and the specific neighborhood structure used.

- Lastly, the time complexity is also affected by the time required to estimate the fitness change between the current solution and its neighboring solutions. This step involves evaluating the fitness function for each solution in the neighborhood, which can contribute significantly to the overall time complexity of the local search algorithm. Therefore, optimizing the fitness change estimation process can help improve the efficiency of the local search procedure.

3.2.4 Performance of local search

Generally, the performance of a local optimization method heavily depends on the neighborhood structure $\mathcal{N}$ used. In fact, a local search algorithm partitions the solution space into several *basins of attraction* of local optima relative to the neighborhood used. A basin of attraction $B(\widehat{s})$ of a local optimum $\widehat{s} \in \mathcal{S}$ is a subset of the solution space, i.e., $B(\widehat{s}) \subset \mathcal{S}$, such that $\forall s \in B(\widehat{s}), f(s) \geq f(\widehat{s})$. When launching a local search method from a solution $s \in B(\widehat{s})$, this method will return $\widehat{s}$ at the end of the search (if the method is not equipped with a mechanism to escape local optima, as in SS-metaheuristic algorithms).

In the following sections, we present the most important SS-metaheuristics: Simulated Annealing (SA), Iterated Local Search (ILS), Variable Neighborhood Search (VNS), Tabu Search (TS), and GRASP method. Each of these SS-metaheuristics has its own mechanism for escaping local optima (i.e., a mechanism to diversify the exploration of the search space and increase the chance of encountering a global optimum).

3.2.5 2-opt local search for TSP

The 2-opt based local search is a simple and effective heuristic for the TSP. It operates by iteratively removing two edges from the current tour and reconnecting them in a different way to improve the tour length. Here is the pseudocode for the 2-opt based algorithm with the first-improvement strategy :

The algorithm starts with an initial tour T, which can be generated randomly or using constrictive heuristics (see 3.1.1). It then iterates through all possible pairs of edges in the tour and evaluates if swapping them would improve the tour length. If an improvement is found ($change < 0$), the edges are swapped, and the process continues until no further improvement is possible.

The `TwoOptSwap` function takes as input the current tour T and the indices of the four nodes (i, j, k, l) defining the edges to be swapped. It returns the new tour obtained by swapping these edges. Figure 3.1 illustrates the 2-opt move.

The `fitnessChange` function calculates the difference in tour lengths between the current tour and the new tour. This fitness change is estimated by comparing their

Algorithm 3.5: 2-opt based local search for TSP

Input: Graph $G(V, E)$ representing the TSP instance
Output: Improved tour

```
1  T ← Initial tour;
2  repeat
3  |  improvement ← false;
4  |  for each pair of edges (i, j) and (k, l) where i, j, k, l are distinct do
5  |  |  T' ← TwoOptSwap(T, i, j, k, l);
6  |  |  change ← fitnessChange(T, new_tour);
7  |  |  if change > 0 then
8  |  |  |  T ← T';
9  |  |  |  improvement ← true;
10 |  |  |  break;
11 |  |  end
12 |  end
13 until No improvement possible (improvement = false);
```

total tour lengths. It is the difference between the sum of distances of new edges and the sum of distances of old edges.

$$change = (d'_1 + d'_2) - (d_1 + d_2); \tag{3.1}$$

where d_1 and d_2 (resp. d'_1 and d'_2) are the old edge distances (resp. the new edge distances). If this tour length difference is negative, it indicates an improvement in the tour quality.

This 2-opt local optimization algorithm has a time complexity of $O(n^2)$, where n is the number of cities in the TSP instance. Although it may not always find the optimal solution, it often produces good-quality solutions in practice.

3.2.6 2-exchange local search for knapsack problem

Here is the pseudocode for the best-improvement 2-exchange local search algorithm for BKP:

In this algorithm, we start by randomly initializing a feasible solution S. We then iterate through the following steps until no improvement is possible:

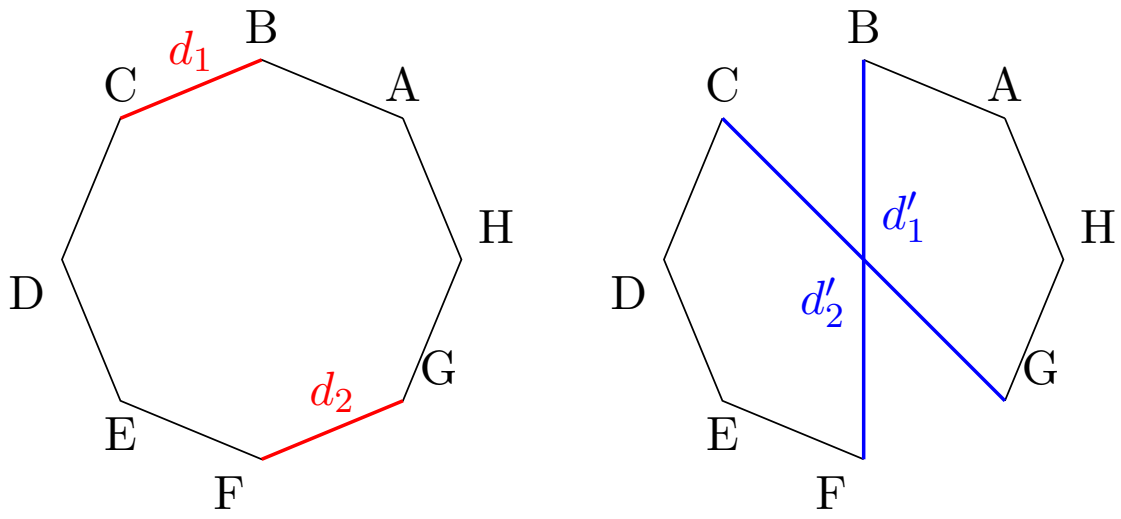

Figure 3.1: 2-opt move. The two red edges (left) are replaced by the two blue ones (right),

Algorithm 3.6: 2-exchange based local search for BKP

Input: Knapsack capacity W, object weights w_i, object values v_i
Output: Optimal selection of objects

```
Randomly initialize a feasible solution S;
S_best ← S;
repeat
    N(S) ← Generate all feasible neighboring solutions of S using 2-exchange
    neighborhood;
    S' ← Solution in N(S) with the highest total value;
    if S' is better than S_best then
        S_best ← S';
    end
    if S' is better than S then
        S ← S';
    end
until no improvement possible;
return S_best
```

1. Generate all neighboring solutions $N(S)$ of the current solution S using a 2-exchange neighborhood structure. The 2-exchange neighborhood considers all possible pairs of items in the knapsack and explores solutions obtained by exchanging two items between the knapsack and outside the knapsack.

2. Select the solution S' from $N(S)$ with the highest value.

3. If S' is better than the best solution found so far (S_{best}), update S_{best}.

4. If S' is better than the current solution S, update S to S'.

The algorithm terminates when no better solution is found. Finally, it returns the best solution S_{best} found during the search process.

3.3 Simulated Annealing (SA)

SA (Nikolaev and Jacobson, 2010) is a classic SS-metaheuristic method. It originates from the framework of statistical mechanics (the Metropolis acceptance criterion (Metropolis et al., 1953)). It was introduced as an optimization method for solving COPs for the first time in (Černỳ, 1985, Kirkpatrick et al., 1983). The basic principle of SA is inspired by the physical annealing process used in metallurgy to improve the quality of a solid. During the transition of a metal from the liquid to the solid state, the metal loses energy and eventually takes on a crystalline structure. The perfection or quality of this structure depends on the cooling rate. A gradual decrease in temperature allows obtaining an absolute minimum of energy and thus a well-structured crystal. In optimization, the objective function is assimilated to the material's energy.

SA is the first SS-metaheuristic to apply an explicit strategy to avoid local optima. It allows the traversal of a neighborhood of lower quality when a local optimum is reached. The idea is to make a move (i.e., move within the local neighborhood of the current solution) according to a probability distribution that depends on the quality of the different neighbors: the best neighbors have higher probabilities, while the worst ones have lower probabilities. The SA algorithm uses a parameter called the *temperature* (by analogy with the natural inspiration), which plays an important role. At high temperature, all neighboring solutions have roughly the same probability of being accepted. At low temperature, moves that degrade the solution quality have low probabilities of being applied. While for zero temperature, no degradation of solution quality is accepted. Thus, the temperature is high at the beginning of the search and

then gradually decreases to approach 0 at the end. In this way, the probability of degrading solution quality decreases over time.

The pseudocode of SA is given in Algorithm 3.7. The algorithm starts by generating an initial solution and initializing the temperature (lines 1 to 3). At each iteration k of the while-loop (lines 4 to 12), a solution s' is generated in the neighborhood $\mathcal{N}(s)$ of the current solution s (line 5) either randomly or using a predefined rule. If s' is of better quality than s, then s' becomes the new current solution (line 7). Otherwise, if s' is worse than s, s' is accepted (line 9) with a probability depending on the current temperature T_k, and the difference between the fitness values of s and s' ($f(s') - f(s)$, the degree of solution degradation). This probability is usually calculated according to a Boltzmann distribution $\exp(-\frac{f(s')-f(s)}{T_k})$. The temperature decreases after each SA iteration (line 11). It is controlled during the search by a decreasing function that defines a cooling schedule.

3.3.1 Basic SA for TSP

This outline covers the steps of a basic SA algorithm applied to the TSP. It use reversal moves for generating neighboring solutions.

1. Initialization

- Generate an initial tour s randomly or using a heuristic algorithm as the nearest neighbor algorithm (see Section 3.1.1).
- Start with an initial temperature T_0.
- Set the termination condition.

2. Neighbor generation (Reversal move)

- Randomly select two distinct indices i and j (where $i < j$) representing the positions of two cities in the current tour s.
- Reverse the order of cities between indices i and j (inclusive) in s to generate a neighboring tour s'.

Algorithm 3.7: Simulated annealing algorithm

```
  {Initialize current solution}
1 s ← GenerateInitialSolution()
2 k ← 0
  {Initialize the temperature according to cooling schedule}
3 T_k ← InitialTemperature()
4 repeat
      {Generate a neighboring solution}
5     s' ← GenerateNeighborSolution(N(s))
6     ΔE ← f(s') − f(s)
7     if (ΔE < 0) then
          {Accept s' as the new solution}
8         s ← s'
9     else
          {Generate a random number in the range [0, 1]}
10        ρ ← rand01()
11        if (ρ < e^(−ΔE/T_k)) then
              {Accept a lower-quality solution}
12            s ← s'
13        end
14    end
      {Decrease the temperature according to cooling schedule}
15    T_{k+1} ← Cooling(T_k, k)
16    k ← k + 1
17 until stopping criterion met
18 return s
```

3. Evaluation

- Calculate the fitness value (total distance) of the new tour s'.

4. Acceptance criteria

- If the neighboring tour s' is better (has smaller total distance), accept it.
- If s' is worse (has larger total distance), accept it with a probability determined by the Metropolis criterion and the current temperature: $e^{-\frac{\Delta f}{T_i}}$, where Δf is the change in total distance between s and s', and T_i is the current temperature at iteration i.

5. Cooling schedule

- At each iteration i, reduce the temperature according to a predefined cooling schedule. For example, update the temperature using logarithmic decrement:

$$T_i = \frac{T_0}{\log(i+1)}.$$

  The temperature decreases logarithmically with each SA iteration.

6. Termination

- Repeat steps 2-5 until a termination condition is met (e.g., reaching a maximum number of iterations or a sufficiently low temperature).

7. Output

- Output the best tour found (smallest total distance) during the SA process.

3.3.2 Basic SA for BKP

This outline covers the steps of a basic SA algorithm for the BKP.

1. Initialization

- Generate an initial solution S by randomly selecting items to include in the knapsack or using a greedy construction (see Section 3.1.2).
- Start with an initial temperature T_0.
- Set the termination condition.

2. Neighbor generation (Swap move)

- Randomly select two distinct objects in the current knapsack S.
- Swap the inclusion status of these two objects (i.e., if one is included, exclude it, and vice versa) to generate a neighboring solution S'.

3. Evaluation

- Calculate the objective value (total value of objects included in the knapsack) of the new solution S'.
- If the total weight of objects in S' exceeds the knapsack capacity W, set the objective value to a very low value to penalize this solution.

4. Acceptance criteria

- If the neighboring solution has a higher objective value, accept it.
- If the neighboring solution has a lower objective value, accept it with a probability determined by the Metropolis criterion and the current temperature: $e^{-\frac{\Delta f}{T_i}}$, where Δf is the change in objective value and T_i is the current temperature at iteration i.

5. Cooling schedule

- At each iteration i, reduce the temperature according to a predefined cooling schedule (e.g., geometric cooling). For example, update the temperature using the formula:

$$T_i = \alpha \cdot T_{i-1},$$

  where α is the cooling rate.

6. Termination

- Repeat steps 2-5 until a termination condition is met (e.g., reaching a maximum number of iterations or a sufficiently low temperature).

7. Output

- Output the best solution found (highest objective value) during the SA process.

3.4 Iterated Local Search (ILS)

The principle of ILS (Lourenço et al., 2010) is simple: instead of repeatedly applying a local search procedure to solutions generated independently of each other, ILS generates starting solutions for a local search procedure by randomly perturbing previously found suboptimal solutions. This means that ILS performs better than a multi-start local search using the same local search procedure.

The perturbation step plays an important role because it allows exploration of different regions within the solution space. It provides a new initial solution for local search to escape local optima basins. The perturbation mechanism must meet the following requirements. The perturbed solution must be located in a different attraction basin than the current local optimum. In other words, the perturbation mechanism must ensure that the local search procedure used does not return to the current local optimum when starting from the perturbed solution. However, at the same time, the perturbed solution must be closer to the previous suboptimal solution than to a randomly generated solution. This is to avoid a multi-start local search from randomly generated initial solutions.

The pseudocode of ILS is described by Algorithm 3.8. The search process starts by constructing an initial solution s; function `GenerateInitialSolution()`. This solution is then improved via a local search procedure; function `LocalSearch(`s`)`. After this initialization, the ILS search is done by iterating over three steps. The first step consists of applying a perturbation procedure to the current local optimum s to obtain a perturbed solution s'; function `Perturbation(`$s, history$`)`. The parameter *history* refers to the possible influence of search history on the process. As mentioned earlier, the perturbation mechanism must prevent cycles in the search, i.e., return to already visited local optima. Additionally, the *strength* of perturbation (i.e., the amount of change) must be well chosen. This is because, on the one hand, a small perturbation may not allow the ILS algorithm to escape the current attraction basin, and on the other hand, a large perturbation may make the search similar to a multi-start local search from random solutions. The second step consists of performing the local search procedure on the perturbed solution to obtain a new local optimum s'. The final step

requires selecting between solutions s and s'; function `Acceptance`$(s, s', history)$. Typically, the ILS algorithm chooses the better solution between s and s'. However, other acceptance criteria that exploit the search history can be used.

Algorithm 3.8: Iterated Local Search Algorithm

```
  {create initial solution}
1 s ← GenerateInitialSolution()
  {apply local search}
2 s ← LocalSearch(s)
3 repeat
      {perturb the current solution}
4     s' ← Perturbation(s, history)
      {apply local search to perturbed solution}
5     s' ← LocalSearch(s')
      {choose between new and current solution}
6     s ← AcceptanceCriteria(s, s', history)
7 until Termination condition satisfied
8 return s
```

3.4.1 Basic ILS for TSP

This outline covers the steps of a basic ILS algorithm applied to the TSP.

1. Initialization

- Generate an initial solution s randomly or using a heuristic algorithm (see Section 3.1.1).
- Set parameters: maximum number of iterations, perturbation strength.

2. Local search

- Apply a local search algorithm (e.g., 2-opt) to improve the current solution s (see Section 3.2.5).
- Repeat the local search process until no further improvement can be made.

3. Main loop

Repeat the following steps until the stopping criterion is met:

- Apply a perturbation operator to the current solution s to introduce diversity. Perturbation can involve random changes, such as random swaps, reversals, or insertions.
- Apply local search (of step 2) to the perturbed solution s'.
- Choose between the new solution s' and the current solution s. If s' is better than s, replace s with s'.

4. Stopping criterion

- Repeat the main loop until a termination condition is met (e.g., reaching a maximum number of iterations).

5. Output

- Output the best solution found (smallest total distance) during the ILS process.

3.4.2 Basic ILS for BKP

This outline covers the steps of a basic ILS algorithm for the BKP.

1. Initialization

- Generate an initial solution s randomly or using a heuristic algorithm (see Section 3.1.2).
- Set parameters: maximum number of iterations, perturbation strength.

2. Local search

- Apply a local search algorithm (e.g., 1-flip neighborhood) to improve the current solution s.

- In 1-Flip neighborhood, for each object, toggle its inclusion status (i.e., if it is in the knapsack, remove it; if it is not in the knapsack, add it) and evaluate the solution fitness. Keep the change if it increases the total value.
- Repeat the local search process until no further improvement can be made.

3. Main loop

Repeat the following steps until the stopping criterion is met:

- Apply a perturbation operator to the current solution s. The perturbation can randomly swap the inclusion status of two objects in the knapsack. This introduces a small change to the solution.
- Apply local search (of step 2) to the perturbed solution s'.
- Choose between s' and s. If s' is better than s, replace s with s'.

4. Stopping criterion

- Repeat the main loop until a termination condition is met (e.g., reaching a maximum number of iterations).

5. Output

- Output the best solution found (highest total value) during the ILS process.

3.5 Variable Neighbourhood Search (VNS)

VNS (Hansen et al., 2010) is an SS-metaheuristic proposed in 1995 (Mladenovic, 1995, Mladenović and Hansen, 1997). It is based on the idea of systematically changing the neighborhood structure. The change in neighborhood occurs in two phases: in a local search step to find local optima and in a perturbation step to escape the valleys containing them.

The pseudo-code of the basic VNS algorithm is described in Algorithm 3.9. The algorithm operates on a predefined set of neighborhood structures, often arranged

in increasing order of size: $|\mathcal{N}_1| < |\mathcal{N}_2| < \ldots < |\mathcal{N}_{k_{\max}}|$. k denotes the index of the current neighborhood $\mathcal{N}_k$. The algorithm starts with an initial solution s at line 1. The neighborhood is initialized to $\mathcal{N}_1$ at line 3. Each iteration of the inner loop (lines 4 to 13) involves applying a perturbation to s in the $k-$th neighborhood (line 5), followed by a local search procedure (best-improvement) applied to the perturbed solution s' (line 6). The local search can use any neighborhood structure and is not limited to the set of neighborhoods $\mathcal{N}_k$ $(k = 1, 2, \ldots, k_{\max})$. If s'' is of better quality than s, s'' replaces s (line 8) and the local neighborhood is reset to $\mathcal{N}_1$ (line 9). Otherwise, the algorithm considers the next local neighborhood $\mathcal{N}_{k+1}$ (line 11).

Algorithm 3.9: Variable Neighborhood Search Algorithm

```
   Let N_k (k = 1, 2, ..., k_max) be a predefined set of local neighborhood structures
 1 s ← GenerateInitialSolution()
 2 repeat
 3 |   k ← 1
 4 |   while (k < k_max) do
 5 |   |   s' ← RandomSelection(s, N_k)
 6 |   |   s'' ← LocalSearch(s')
 7 |   |   if (f(s'') < f(s)) then
 8 |   |   |   s ← s''
 9 |   |   |   k ← 1
10 |   |   else
11 |   |   |   k ← k + 1
12 |   |   end
13 |   end
14 until Stopping criterion met
15 return s
```

The process of changing the neighborhood structure in case of no improvement corresponds to a diversification mechanism in the search. The effectiveness of this systematic neighborhood change strategy can be explained by the fact that a "bad" location on the search landscape given by one neighborhood might be a "good" location given by another neighborhood. In other words, different neighborhoods may not necessarily yield the same search landscapes, and consequently, a local search strategy behaves differently on each of them. Thus, the choice of neighborhood structures is a critical point of the VNS algorithm. Neighborhood structures must exploit the various

properties and characteristics of the search space, i.e., they must provide different abstractions of the search space.

3.5.1 Neighborhood structures for TSP

Here are some neighborhood structures that can be applied to the TSP:

- **Neighborhood 1 (2-opt):** Generate neighbors by applying the 2-opt mutation, which swaps two edges in the tour to create a new tour.
- **Neighborhood 2 (Insertion):** Generate neighbors by inserting a city from the tour into a different position, creating a new tour.
- **Neighborhood 3 (Swap):** Generate neighbors by swapping the positions of two cities in the tour.

3.5.2 Neighborhood structures for BKP

Here are some neighborhood structures for the BKP:

- **Neighborhood 1 (Flip):** Generate neighbors by flipping the value of one object in the solution, either from 0 to 1 or from 1 to 0.
- **Neighborhood 2 (Swap):** Generate neighbors by swapping the inclusion status of two objects in the solution.
- **Neighborhood 3 (Random perturbation):** Generate neighbors by randomly perturbing the current solution, for example, by randomly changing the inclusion status of a subset of objects.

3.6 Tabu Search (TS)

TS is one of the most widely used SS-metaheuristic algorithms for solving COPS. The first article introducing the basic idea of TS was published in 1986 (Glover, 1986). The TS algorithm explicitly exploits the search history, both to avoid local optima

and to implement a search space exploration strategy. Like the SA algorithm, TS accepts lower-quality solutions when a local optimum is obtained.

The simple TS algorithm applies a best-improvement local search to improve the current solution and uses short-term memory to escape local optima and prevent cycles in the search. This memory is implemented by a list called the *tabu list* that keeps track of the most recently visited solutions and prohibits returning to them. The neighbors of the current solution are therefore limited to solutions that do not belong to the tabu list, i.e., solutions that have not yet been visited. In the following, we use the term *allowed set* to refer to the set of unvisited solutions.

More formally, the canonical operation of the TS method is described by Algorithm 3.10. The algorithm starts by generating an initial solution s, initializing the best obtained solution s_{opt} with s, and inserting s into the tabu list $TabuList$ (lines 1-3). At each algorithm iteration, the best neighboring solution in the allowed set ($\mathcal{N}(s) \setminus TabuList$) is chosen, even if it is worse, as the new current solution s (see function `BestNeighbor(`$\mathcal{N}(s) \setminus TabuList$`)`). Then, this solution is added to the tabu list (see function `Update(`$TabuList, s$`)`). If the tabu list is full, it replaces one of its elements (usually in FIFO order, i.e., the tabu list is implemented using a queue).

Algorithm 3.10: Tabu Search Algorithm

```
1  s ← GenererSolutionInitiale()
2  s_opt ← s
3  ListeTabou ← {s}
4  repeat
5  |  s ← BestNeighboring(N(s) \ ListeTabou)
6  |  Update(ListeTabou, s)
7  |  if (f(s) < f(s_opt)) then
8  |  |  s_opt ← s
9  |  end
10 until Termination criterion satisfied
11 return s_opt
```

As mentioned earlier, the use of a tabu list prohibits moves to recently visited solutions; thus, it prevents cycles and forces the algorithm to accept non-improving moves. The size of the tabu list, called *tabu tenure*, can strongly influence the search: on one hand, a too small size will force the algorithm to explore larger regions of

the search space; on the other hand, a too large size may lead the algorithm to exploit a local region, as it prohibits moves to a very large number of solutions. The tabu tenure can evolve over time dynamically (Battiti and Tecchiolli, 1994). This strategy increases the size of the tabu list when cycles are detected (to bring more diversification) and decreases it in case of no improvement (to favor intensification).

For a more practical and efficient implementation, the attributes of visited solutions are kept -in the tabu list- instead of the complete solutions. Attributes are generally solution components, moves, differences between solutions, etc. Since more than one attribute can be considered, a tabu list is introduced for each of them. The allowed set is therefore generated according to the tabu conditions defined by the set of attributes and the corresponding tabu lists. This strategy is more efficient but results in information loss because prohibiting an attribute likely implies the exclusion of multiple solutions. Thus, unvisited solutions of good quality may be excluded from the allowed set. To overcome this issue, so-called *aspiration criteria* are defined to allow the acceptance of a solution even if it is prohibited by the tabu conditions. The most commonly used aspiration criterion accepts solutions that are better than the current optimal solution s_{opt}.

3.6.1 TS mechanism for TSP

We highlights the key aspects of a TS mechanism for the TSP using 2-opt neighborhood operator as follows:

- **Tabu list maintenance:** The **tabu list** stores recently applied 2-opt moves. This list prevents revisiting previously explored solutions, promoting diversification in the search process.

- **Tabu tenure definition:** The **tabu tenure** is the number of iterations a move remains in the tabu list. It controls how long a move is considered taboo, influencing the exploration-exploitation trade-off.

- **Move validity assurance:** Before accepting a candidate 2-opt move, the algorithm checks if it is in the tabu list. Tabu moves are excluded unless an aspiration

criterion is met, allowing superior solutions to be accepted, overriding the tabu status.

3.6.2 TS Mechanism for BKP

We outlines the key elements of a TS mechanism for the BKP as follows:

- **Tabu List maintenance:** In the BKP context, a move involves selecting an item to add or remove from the knapsack. The choice of move depends on the specific strategy employed, such as selecting the item with the highest marginal benefit or removing the item with the lowest marginal benefit. The **tabu list** stores recently applied moves.
- **Tabu tenure definition:** idem as for TSP.
- **Move validity assurance:** idem as for TSP.

3.7 Greedy Randomized Adaptive Search Procedure (GRASP)

The GRASP SS-metaheuristic (Resende and Ribeiro, 2010) is a conceptually simple yet often effective optimization technique. It was proposed by Feo and Resende in (Feo and Resende, 1989, 1995). This iterative or multiple-start optimization algorithm combines construction heuristics with a local search step. Each iteration involves constructing a solution followed by applying a local optimization procedure to this solution. The best obtained solution is returned at the end of the search process. A general outline of the GRASP method is presented in Algorithm 3.11.

The main building component of the GRASP method is the probabilistic solution construction procedure. It is an iterative construction, starting from an empty solution and adding solution components until a complete solution is obtained. This memoryless construction [1] relies on a randomized greedy strategy. At each construction step,

[1] In contrast, Ant Colony Optimization (ACO) algorithms, see Section 4.3, also construct solutions incrementally, but they memorize and use search history.

Algorithm 3.11: GRASP Algorithm

```
repeat
    s ← RandomizedGreedyConstruction()
    s ← LocalSearch(s)
    if (f(s) < f(s_opt)) then
        s_opt ← s
    end
until Stopping condition met
return s_opt
```

solution components to be added to the currently constructing solution (a partial solution) are chosen from a list called the Restricted Candidate List (RCL). This list contains, at each construction step, a subset of solution components that can be used to extend the partial solution. Elements of the RCL are selected based on a heuristic criterion that locally estimates the quality of solution components. After determining the RCL list, a solution component is chosen uniformly at random. An important parameter of the GRASP algorithm is the size of the RCL list, denoted by α. If $\alpha = 1$, the solution construction is deterministic and the constructed solutions are greedy. In the other extreme case, i.e., if the RCL always contains all possible solution components that can be added to the partial solutions, a random solution is generated without any influence from the heuristic rule. Therefore, α is a critical parameter that requires careful tuning.

The second phase of the GRASP method consists of performing a local search procedure to improve the constructed solutions. This can be a basic local search like hill climbing, or more advanced like SA or TS methods.

According to (Blum and Roli, 2003), the GRASP algorithm can be more effective if two conditions are satisfied: (1) the solution construction strategy samples promising regions of the search space, and (2) the constructed solutions serve as good starting points for the local search used, i.e., the constructed solutions are located in basins of attraction of high-quality local optima.

3.7.1 A greedy randomized construction heuristic for TSP

The pseudocode for a greedy randomized construction heuristic for the TSP is given in 12. This tour construction procedure combines the nearest neighbor heuristic and randomization for selecting the next city to add to the partial tour:

Algorithm 3.12: Greedy Randomized Construction for TSP

Input: Set of n cities C, distance matrix $D = [d_{ij}]_{n \times n}$
Output: Tour T

```
Initialize an empty tour T;
Randomly select a starting city to begin the tour;
Add the starting city to tour T;
while T does not contain all cities do
    Compute candidate list C of unvisited cities based on their proximity to the
    last city in T;
    Assign probabilities to cities in C using a combination of nearest neighbor
    heuristic and randomization;
    Randomly select the next city to visit from C based on the assigned
    probabilities;
    Add the selected city to tour T;
end
Return to the starting city to complete the tour;
return tour T;
```

3.7.2 A greedy randomized construction heuristic for BKP

The pseudocode for a greedy randomized construction procedure for the BKP is shown in Algorithm 13. This subset construction procedure selects objects based on their profit-to-weight ratio, favoring objects with higher profitability relative to their weight.

Algorithm 3.13: Greedy Randomized Construction for BKP

Input: Set of objects O with values p_i and weights w_i, knapsack capacity W
Output: Selected objects S

```
1  Compute the profit-to-weight ratio r_i = v_i / w_i for all objects i = 1, 2, ..., n;
2  Initialize an empty list L of unselected objects;
3  Initialize an empty knapsack S;
4  while Knapsack S is not full and L is not empty do
5  |  Normalize the profit-to-weight ratios in L so that they sum to 1;
6  |  Randomly select an object i from L with probability proportional to r_i;
7  |  if Adding object i to S does not exceed capacity W then
8  |  |  Add object i to knapsack S;
9  |  end
10 |  Remove object i from list L;
11 end
12 return Selected objects in knapsack S;
```

Chapter 4

Population-based metaheuristics

This chapter serves as a comprehensive exploration of three prominent population-based metaheuristics: Genetic Algorithms (GAs), Particle Swarm Optimization (PSO), and Ant Colony Optimization (ACO). These algorithms, inspired by natural processes, revolutionize the field of optimization by leveraging the power of populations to efficiently search for high-quality solutions in complex problem spaces.

GAs draw inspiration from the principles of evolution, mimicking the process of natural selection to iteratively evolve a population of candidate solutions toward optimal or near-optimal solutions. PSO, on the other hand, takes cues from the collective behavior of bird flocks and fish schools, where individuals adjust their positions in the search space based on personal experience and social interactions to find the best solution collectively. ACO, inspired by the foraging behavior of ants, uses pheromone trails to guide the search process, allowing the algorithm to discover promising regions in the search space.

4.1 Genetic Algorithms (GAs)

4.1.1 Biological metaphor and principles

The Genetic Algorithms (GAs) are among the most prominent examples of evolutionary algorithms (EAs), with roots tracing back to the pioneering work of John Holland and his colleagues at the University of Michigan (Goldberg, 1989, Holland, 1975). These algorithms were introduced as methods inspired by the analogy between

solving optimization problems and the evolutionary processes observed in biological populations. In biological populations, individuals are not ”programmed” to solve specific problems; rather, they continuously adapt to their environment. The theory of neo-Darwinism illustrates that this adaptive trait in living beings arises from a combination of diversity generation and the mechanism of natural selection.

GAs are built upon this biological model. For instance, consider the example of rabbits (adapted from (**?**)): at any given moment, there exists a population of rabbits. Among them, some are faster and more intelligent than others. These swifter and smarter rabbits face lower risks of predation by foxes, allowing more of them to survive and engage in reproduction. Naturally, even slower and less intelligent rabbits may survive due to chance. As the surviving rabbit population breeds, a diverse mix of genetic material is produced. Some slower rabbits mate with faster ones, while some fast rabbits mate with others of their kind. Additionally, mutations occasionally introduce variations into the genetic makeup. The resulting offspring tend to inherit the traits of their faster and smarter parents, making them faster and smarter than those in the original population. This continuous process of adaptation ensures the survival of the fittest within the population.

4.1.2 Terminology

GAs adopt terminology from natural genetics, but with some distinctions tailored to the optimization context. Here is an overview:

- **Individuals or Chromosomes:** These are the basic units in a GA population, representing potential solutions to the optimization problem.
- **Genes:** Within a chromosome, genes represent specific features or traits. They are arranged linearly, with each gene controlling the inheritance of one or more features.
- **Locus:** This refers to the specific position on a chromosome where a gene for a particular feature is located.
- **Alleles:** Genes can exist in different states or values, known as alleles.

In this context, each genotype (a single chromosome) corresponds to a potential solution to the optimization problem. The phenotype of a chromosome, i.e., its meaning or interpretation, is defined by the user.

The evolutionary process carried out on a population of chromosomes involves searching through a space of potential solutions. This search aims to balance two objectives: exploiting the best solutions found so far and exploring different regions of the search space (Blum and Roli, 2003). While hill climbing focuses on exploiting the best solution for possible improvement, it may overlook exploration of the search space. Conversely, a random search explores the search space but may miss exploiting promising regions.

GAs excel at balancing these two objectivesâĂŤexploration and exploitationâĂŤmaking them effective general-purpose optimization methods.

4.1.3 Structure of Genetic Algorithms

The structure of a standard GA resembles that of any Evolutionary Algorithm (EA). Here is an overview of its components:

During generation (or iteration) t, a GA maintains a *population* P of m potential solutions (individuals or chromosomes), denoted as $P(t) = \{X_1^t, X_2^t, \ldots, X_m^t\}$. Each solution X_i is evaluated to determine its "fitness". Subsequently, a new population (generation $t + 1$) is formed by selecting the fittest individuals. The members of this new population undergo alterations through *crossovers* and *mutations* to produce novel solutions.

- **Crossover:** This operation combines features of two parent chromosomes to form two offspring with exchanged segments of the parents. For example, if individuals are represented by five-dimensional vectors, then the crossover of two parent chromosomes $(a_1, b_1, c_1, d_1, e_1)$ and $(a_2, b_2, c_2, d_2, e_2)$ after the second gene would yield the children $(a_1, b_1, c_2, d_2, e_2)$ and $(a_2, b_2, c_1, d_1, e_1)$. Crossover facilitates the exchange of information among different potential solutions, aiding in the exploitation of the search space.

- **Mutation:** This operation randomly alters one or more genes of a selected (offspring) chromosome with a probability equal to the mutation rate. This introduces additional variability into the population, aiding in the exploration of the search space.

A GA requires several components, procedures, or operators, and parameters to define it for a specific problem. The critical elements, italicized in Algorithm **??**, include:

- a genetic representation (encoding) of potential solutions to the problem,
- an evaluation function (objective function or fitness) quantifying solution "adaptation",
- a strategy for generating an initial population of potential solutions,
- a mechanism for parent selection (selection operator),
- genetic operators for creating new solutions (crossover and mutation operators),
- a mechanism for survivor selection (replacement operator),
- values for various algorithm and control parameters (population size, crossover rate, mutation rate, stopping criterion, etc.).

4.1.4 General functioning of GAs

Once an initial population of individuals encoding potential solutions to the optimization problem is created, a GA evolves these individuals through a process of *evolution* and *selection* based on *evaluation*, until it generates a population of individuals representing good solutions.

The pseudocode of a standard GA is illustrated in Algorithm 4.1. The initial population marks the starting point of the search process (line 1). It is typically generated randomly, although problem-specific knowledge can be utilized to construct greedy solutions. Following the creation of the initial population, the fitness values of all individuals are evaluated (line 2).

The reproduction cycle (lines 3 to 9) is central to the generation of a new population. This phase encompasses the selection of parent individuals (line 4), their combination through crossover (line 5), the mutation of resulting offspring (line 6), and the subsequent evaluation of these offspring (line 7). Selection and evolution operators, such as *crossover* and *mutation*, are standard components of GAs.

The new population (*offspring*) generated during the reproduction cycle is then used in conjunction with the current population (P) to determine the population of the next generation (line 8). Finally, the algorithm returns the best solution found at the end of the search.

The flowchart of a standard GA is depicted in Figure 4.1.

Algorithm 4.1: Pseudo-code of standard GA

```
1  P ← GenererPopulationInitiale(N)
2  Evaluation(P)
3  repeat
4  |   parents ← SelectionParents(P)
5  |   offsprings ← Crossover(parents)
6  |   offspring ← Mutation(offsprings)
7  |   Evaluation(offsprings)
8  |   Replacement(P, offsprings)
9  until Stopping condition met
10 return Best solution in P
```

4.1.5 Representation (Encoding of Individuals)

The initial step in designing any metaheuristic algorithm is to establish a bridge between the context of the original optimization problem and the problem-solving space where the search occurs. This often involves simplifying or abstracting certain real-world aspects to create a well-defined and tangible problem context in which potential solutions can exist and be evaluated. This first design step is to decide how potential solutions should be specified and stored so they can be manipulated by various operators and procedures of the metaheuristic algorithm.

In the context of GAs, the objects forming potential solutions in the original problem context are termed *phenotypes*, while their encoding, i.e., the individuals

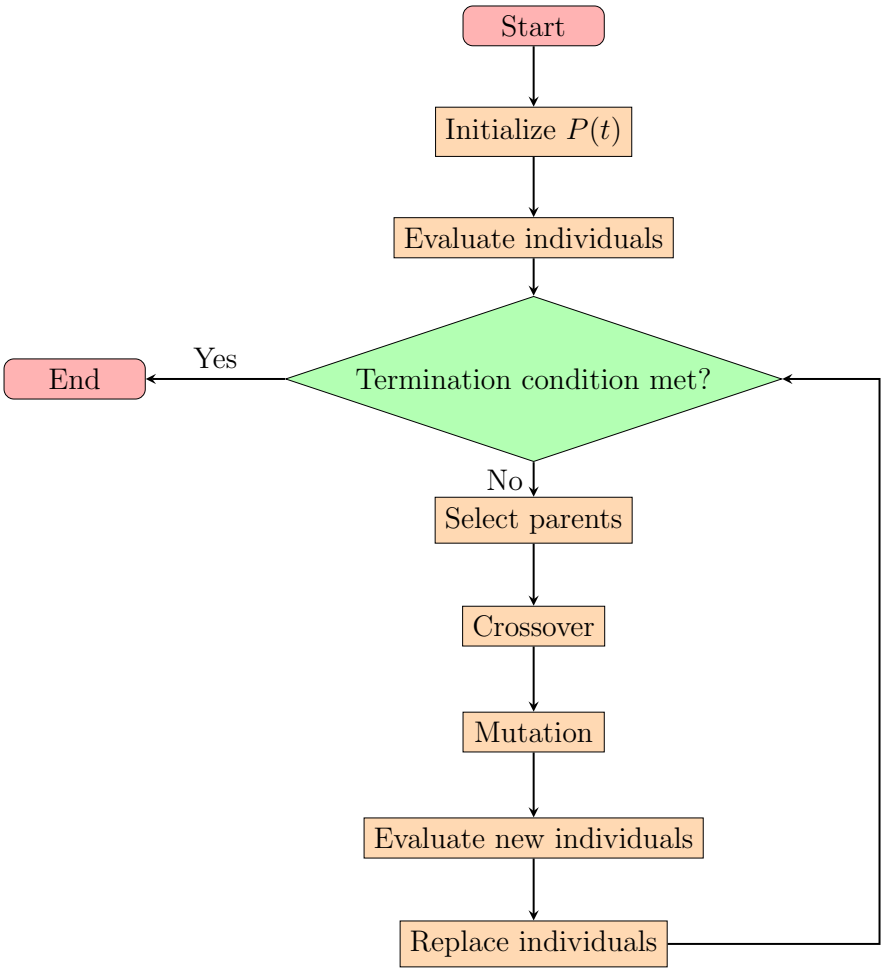

Figure 4.1: Flowchart of standard GA.

within GA, are referred to as *genotypes*. This initial design step is commonly known as ***representation or encoding***, as it specifies a mapping from phenotypes to a set of genotypes. For instance, in an optimization problem where possible solutions are integers, the set of attainable integers would constitute the set of phenotypes. It is crucial to recognize that the phenotypic space may significantly differ from the genotypic space, and all evolutionary exploration occurs within the genotypic space. A solution (a good phenotype) is obtained by decoding the best genotype found at the end of the search process.

The chosen representation plays a pivotal role in the efficiency and efficacy of a metaheuristic algorithm, constituting an indispensable stage in algorithm design. It must be tailored and relevant to the optimization problem being addressed. Addition-

ally, the effectiveness of a solution representation is closely tied to the search operators applied to it (mutation, crossover, neighborhood, etc.). Hence, when defining a representation, one must consider how the solution will be evaluated and how the search operators will function.

Several alternative encodings may exist for a given optimization problem. A sound encoding should possess the following essential characteristics (Talbi, 2009):

- **Completeness:** All solutions pertinent to the problem must be encoded.
- **Connectivity:** A search path must exist between any two possible solutions in the search space. Every solution in the search space, particularly the global optimal one, should be reachable by the search operators.
- **Efficiency:** The representation ought to be readily evaluated and manipulable by the search operators. The time complexities of the fitness evaluation and search operators should be reduced.

Some conventional representations (Figure 4.2) are widely employed across a broad spectrum of optimization problems, either separately or in combination.

4.1.6 Objective function or Fitness

The evaluation function (also referred to as the cost function, objective function, or fitness) defines the target to be achieved. As mentioned earlier, it acts as the environment, assessing potential solutions based on their suitability. It assigns a real value to each solution in the search space ($\mathcal{S}$), indicating its quality or fitness: $f : \mathcal{S} \rightarrow \mathbb{R}$. This value establishes an absolute measure and enables the definition of a complete ordering among all solutions in the search space. The objective function plays a crucial role in the design of a GA or any metaheuristic algorithm, guiding the search towards "good" solutions in the search space.

4.1.7 Population

The population serves as the repository for (the representation of) potential solutions. It comprises a collection of genotypes that collectively undergo evolution. The

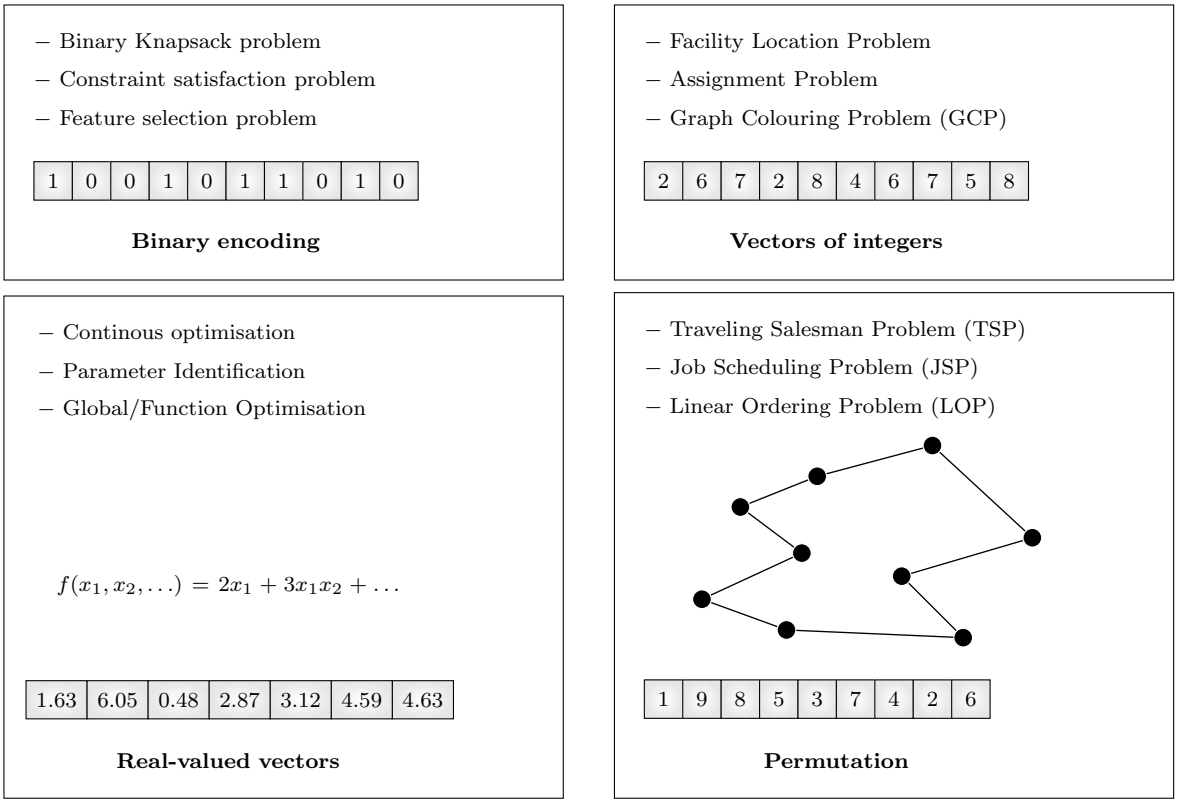

Figure 4.2: Some examples of classical representations: binary encoding, vector of discrete values, vector of real numbers, and permutation encoding.

population evolves and adapt over time. Defining a population, given a representation, can be as straightforward as specifying the number of individuals it contains, known as the ***population size***.

In nearly all GA applications, the population size remains fixed throughout the evolutionary process. Selection operators (parent selection and survivor selection) operate at the population level, considering the entire current population when making choices. For instance, the best individual from a population may be chosen to seed the next generation, or the worst individual may be replaced by a new one. This population-level activity stands in contrast to variation operators, which act on one or more parent individuals.

The ***diversity*** of a population measures the variety of different solutions present. There is not a single measure of diversity; it can refer to the number of unique fitness values, phenotypes, or genotypes. It is important to note that the presence of only one fitness value in the population does not necessarily mean only one phenotype is present, as multiple phenotypes can share the same fitness. Similarly, the presence of only one phenotype does not imply only one genotype. However, if only one genotype is present, it suggests that there is only one phenotype and one fitness value.

4.1.7.1 Initial population

The initial population serves as the starting point for GAs and other P-metaheuristics. The P-metaheuristic algorithms leverage diverse initial populations to explore the search space effectively. Meanwhile, individual-based metaheuristics like the SA optimizer or TS method are better suited for exploiting the search space. Constructing the initial population is critical for both the effectiveness and efficiency of the algorithm. Therefore, it is crucial to pay close attention to this step.

When generating the initial population, the key consideration is diversification. If the initial population lacks diversity, it may lead to premature convergence, hindering the algorithm performance. For instance, using a greedy heuristic or a SS-metaheuristic method for generating the entire population can limit diversity. To ensure diversity, a portion of the initial population should be randomly generated.

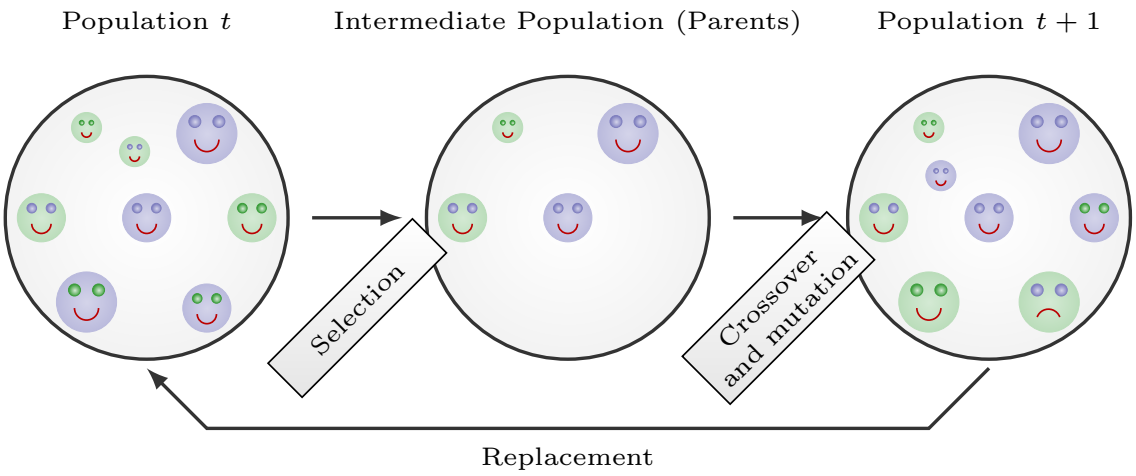

Figure 4.3: Main principles of GA.

4.1.8 Parent Selection

Parent selection plays a crucial role in GAs by determining which individuals will contribute to the reproduction process. Its primary goal is to identify high-quality individuals for reproduction, thereby enhancing the overall quality of the population. Typically, parent selection in GAs is probabilistic, favoring individuals with higher fitness values while still providing a chance for lower-quality individuals to contribute.

There are two common methods for assigning selection scores/probabilities to individuals:

- **Proportional assignment:** Each individual absolute fitness value is considered.
- **Rank-based assignment:** Individuals are ranked based on their fitness values relative to others in the population. This rank ordering enables a fair comparison across individuals.

By employing suitable parent selection strategies, GAs can strike a balance between exploration and exploitation, leading to more effective optimization processes. Some well-known parent selection strategies include: roulette wheel selection, tournament selection, and rank-based selection.

4.1.8.1 Roulette wheel selection

This is the most common selection strategy, where each individual is assigned a selection probability proportional to its relative fitness. Imagine a roulette wheel where each individual occupies a sector whose angle is proportional to its fitness value (see Figure 4.4). Selection of μ individuals is performed through μ independent spins of the roulette wheel, with each spin selecting a single individual. The fittest individuals occupy larger sectors and thus have a higher chance of being chosen.

Concretely, the roulette wheel selection proceeds as follows. Let f_i be the fitness of individual $i = 1, \ldots, N$ in the population. The selection probability of individual i (p_i) is calculated as follows:

$$p_i = \frac{f_i}{\sum_{j=1}^{N} f_j}. \tag{4.1}$$

An individual i is selected if a uniformly chosen random number between 0 and 1 falls within the interval $\left[\sum_{j=1}^{i-1} p_j, \sum_{j=1}^{i} p_j\right]$.

In roulette wheel selection, when all individuals have very close fitness values, the fittest individuals will not be favored. On the other hand, when fitness has a high amplitude (i.e., a large distance between its extreme values), this selection strategy may overlook low-quality individuals.

4.1.8.2 Tournament selection

Random subsets of k individuals, called tournaments, are randomly selected, and the fittest individual from each tournament is chosen for reproduction (see Figure 4.5). This method is simple and effective, especially when the tournament size is appropriately chosen. To select μ individuals, the tournament selection is repeated μ times.

4.1.8.3 Ranking selection

Individuals are ranked based on their fitness values, and selection probabilities are assigned according to their ranks. Higher ranks (starting from 1) are assigned to individuals of better quality while rank N (N being the population size) is assigned to the worst individual (see Figure 4.6). This strategy favors individuals with higher

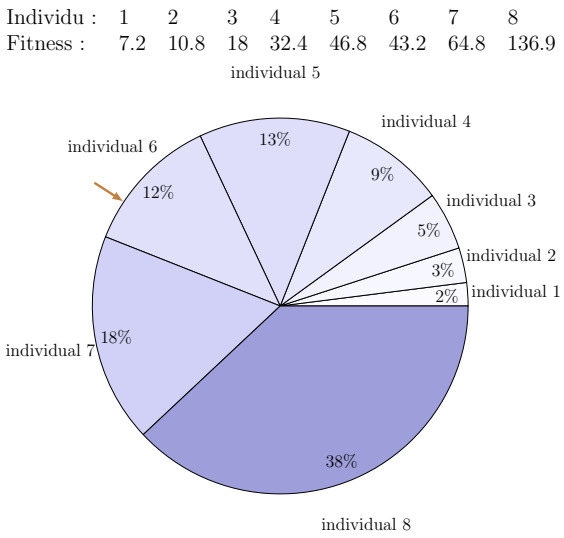

Figure 4.4: Roulette wheel selection.

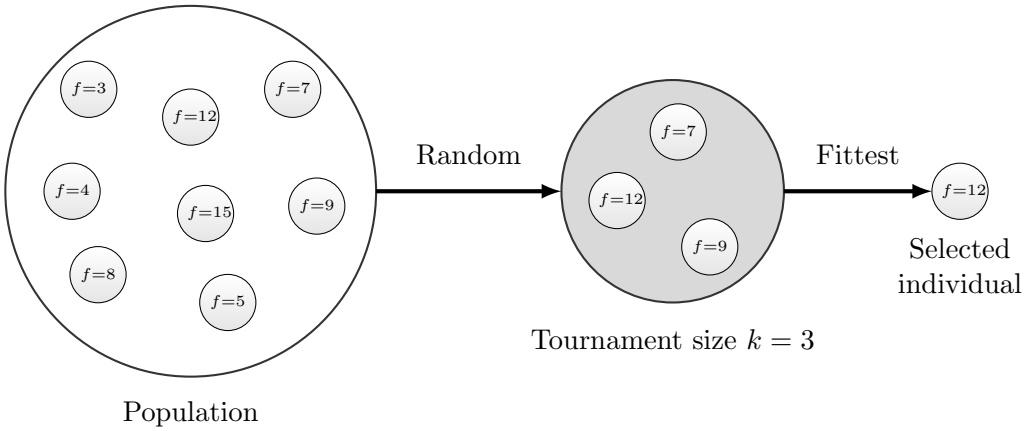

Figure 4.5: Tournament selection strategy. For instance, a tournament of size $k = 3$ is conducted. Three solutions are randomly selected from the population. The best solution among them is then chosen.

ranks (i.e., good fitness). It ensures that even individuals with lower fitness values have a chance to contribute to the next generation. The probability of selecting individuals can be calculated based on their rank in the population, as follows:

$$p(i) = \frac{2(N - r(i))}{N(N-1)}, \tag{4.2}$$

where $r(i)$ is the rank of individual i in the linear ranking.

4.1.9 Mutation

In GAs, the phase of reproduction involves creating new individuals from selected parents. This process is facilitated by **variation operators**, which generate new candidate solutions in the search space. The two primary variation operators in GAs are mutation (a unary operator) and crossover (a binary or n-ary operator).

Mutation is a fundamental operator applied to a single individual to introduce a small, random change, thereby producing a mutant or offspring. Its purpose is to inject novel genetic information into the population. The mutation operation is controlled by a probability parameter P_m, which determines the likelihood of altering each element (gene) in the individual. Typically, small values are recommended for P_m (e.g., $P_m \in [0.001, 0.01]$), ensuring subtle changes in the population.

Some key considerations should be taken into account when designing or utilizing a mutation operator are (Talbi, 2009):

- **Ergodicity:** The mutation operator should allow exploration of the entire solution space, ensuring no regions are left unexplored.
- **Validity:** It is crucial that the mutation operator yields valid solutions. However, this may not always be feasible, especially for constrained optimization problems.
- **Locality:** Mutation should induce small changes in the solution. Controlling the size of these changes ensures that the effect on the phenotype (observable solution characteristics) remains proportional to the changes made in the genotype

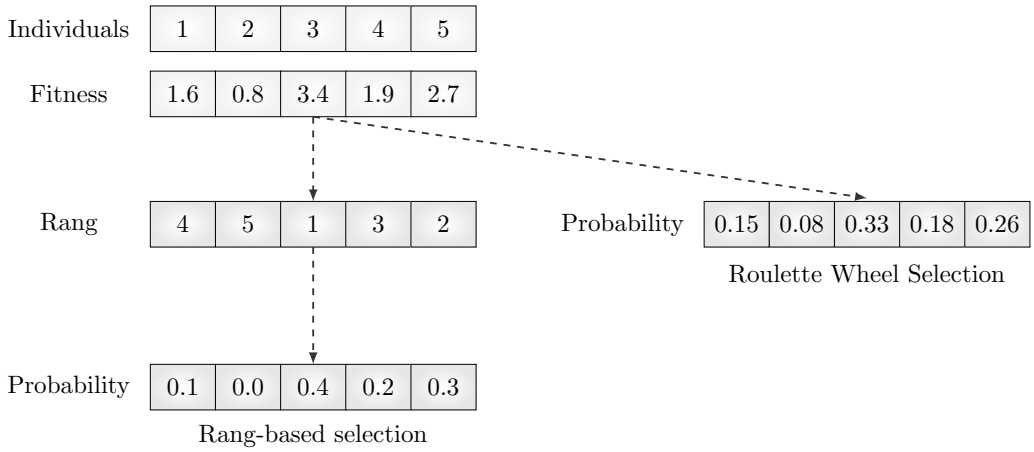

Figure 4.6: Ranking strategy using linear ranking.

(the genetic representation). Strong locality ensures meaningful exploration of the problem landscape, while weak locality may lead to random exploration.

The specific implementation of the mutation operator depends on the type of encoding used. Here are some common mutation strategies for different types of discrete encoding:

- **Mutation for binary encoding:** The *Bit-flip* operator is frequently employed. It flips each gene with a low probability P_m, resulting in a stochastic alteration of the individual genotype (see Figure 4.7). The number of bits flipped is proportional to P_m.
- **Mutation for discrete encoding**: Discrete encoding involves changing the value associated with an element to another value from a predefined set (see Figure 4.8). This can introduce diversity into the population.

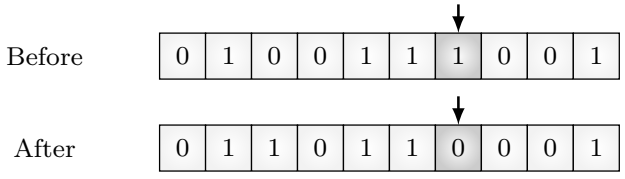

Figure 4.7: *Bit-flip* mutation for binary encoding.

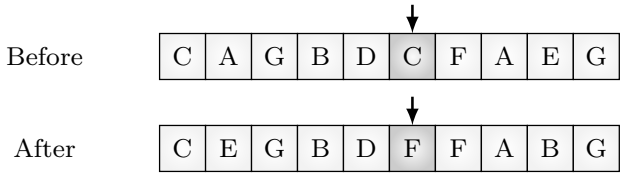

Figure 4.8: Mutation for discrete encoding.

- **Mutation for permutation encoding**: In permutation representations, mutation operations alter the order of permuted elements. This can be achieved through operations like exchange, reversal, insertion, or scramble, each altering the permutation vector in a specific way (see Figure 4.9).

These mutation strategies ensure that the population continues to explore the solution space effectively, balancing exploration and exploitation to find optimal or near-optimal solutions.

4.1.10 Crossover

In contrast to unary operators (like mutation), the crossover operator is binary and sometimes n-ary. Within GAs, it holds a pivotal role as a search operator. The purpose of crossover operators is to inherit or merge certain characteristics from two parents to generate two offspring. Similar to the mutation operator, crossover operators are stochastic and their design primarily depends on the representation used. Consequently, different crossover operators must be defined for different solution representations. When designing or utilizing a crossover operator, several critical points should be considered (Talbi, 2009):

- **Heritability:** The crossover operator must inherit genetic material from both parents. It is considered a pure crossover operator (with strong heritability) if both parents are identical, it produces two identical offspring. Hence, mutation and crossover operators complement each other. In this context, mutation introduces diversification into individuals by introducing missing values into the current population.

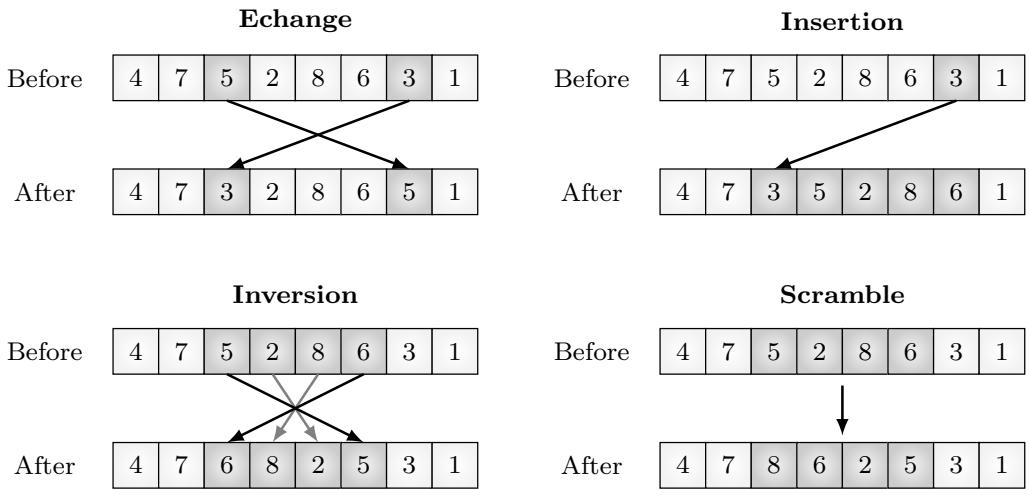

Figure 4.9: Permutation encoding — Exchange, reversal, inversion, and scramble mutations.

- **Validity:** The crossover operator must construct valid solutions. However, this may not always be feasible, especially for constrained optimization problems.

The crossover rate $P_c \in [0, 1]$ represents the proportion of parents on which a crossover operator will act. The optimal parameter value for P_c is related to other parameters such as population size N, mutation probability P_m, and selection strategy. The most commonly used rates fall within the interval $[0.45, 0.95]$.

Below, we illustrate some commonly used crossover types for linear representations (binary, permutation, and real-valued vectors) (**?**).

4.1.10.1 Crossovers for binary encoding

Three standard crossover operators are commonly used for binary representations. They all start with two parents and create two offspring.

Single-point crossover ($1X$): Single-point crossover is simpler for binary encoding. It operates by selecting a crossover point r randomly in the interval $[1, l-1]$ (with l being the length of the linear encoding), then dividing the two parents at this point and creating the two offspring by exchanging the segments (see Figure 4.10).

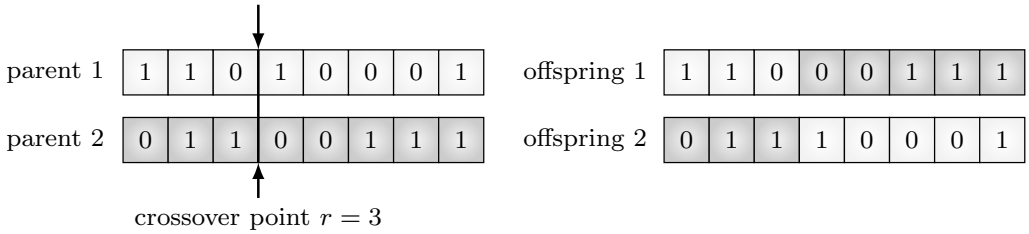

Figure 4.10: Binary representation — Crossover $1X$.

Multi-point crossover (nX): Single-point crossover can easily be generalized to multi-point crossover, where the chromosome is divided into more than two contiguous gene segments, and the offspring are created by taking alternate segments from the parents. In practice, this means choosing n random crossover points within $[1, l-1]$, as illustrated in Figure 4.11 for $n = 2$.

Uniform crossover: Crossover operators $1X$ and nX work by dividing parents into a certain number of contiguous gene sections and reassembling them to produce offspring. In contrast, uniform crossover operates by treating each gene independently and randomly choosing which parent it should be inherited from. This is implemented by generating a vector of l random variables from a uniform distribution over $[0, 1]$. For each gene position, if the value is less than a parameter p (usually 0.5), the gene value is inherited from the first parent; otherwise, from the second. The second offspring is created using the inverse mapping. This is illustrated in Figure 4.12.

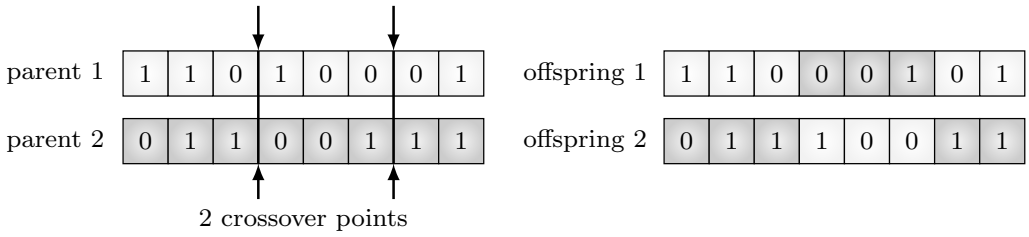

Figure 4.11: Binary representation — Crossover $2X$.

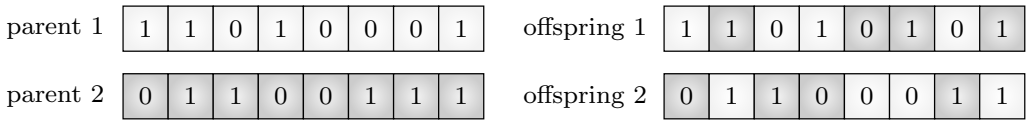

Figure 4.12: Binary representation — Uniform crossover. Random numbers in $[0.3, 0.6, 0.1, 0.4, 0.8, 0.7, 0.3, 0.5]$ and $p = 0.5$ are used to decide inheritance for this example.

4.1.10.2 Crossovers for permutation encoding

Permutation encoding presents challenges when designing crossover operators. Simply swapping sub-permutations between parents can lead to invalid solutions, with elements duplicated or omitted. To overcome these issues, alternative approaches are explored, such as constructing a complete order of all elements or defining movements that connect pairs of elements. Various specialized crossover operators have been developed for permutation problems, aiming to retain as much shared information between parents as possible.

We describe hereafter some widely recognized and commonly employed crossover operators for permutation problems.

Order Crossover (OX): The Order Crossover (OX) operator begins by randomly selecting two crossover points. Segments between these points are then copied from the first parent (P1) into the offspring at the same absolute positions. From the second parent (P2), beginning from the second crossover point, elements not already selected from P1 are chosen to fill in the offspring (see Figure 4.13). The second offspring is constructed similarly, but with the roles of the parents reversed.

The OX operator is a pure crossover operator. If filling or picking starts from the first crossover point, the operator will not be pure. From P1, the relative order and absolute positions of elements are preserved. From P2, only the relative order of elements is preserved.

Partially Mapped Crossover (PMX): The PMX was initially proposed by Goldberg and Lingle as a crossover operator for TSP , and it has since become one of the most widely used operators for permutation problems. Such combinatorial optimiza-

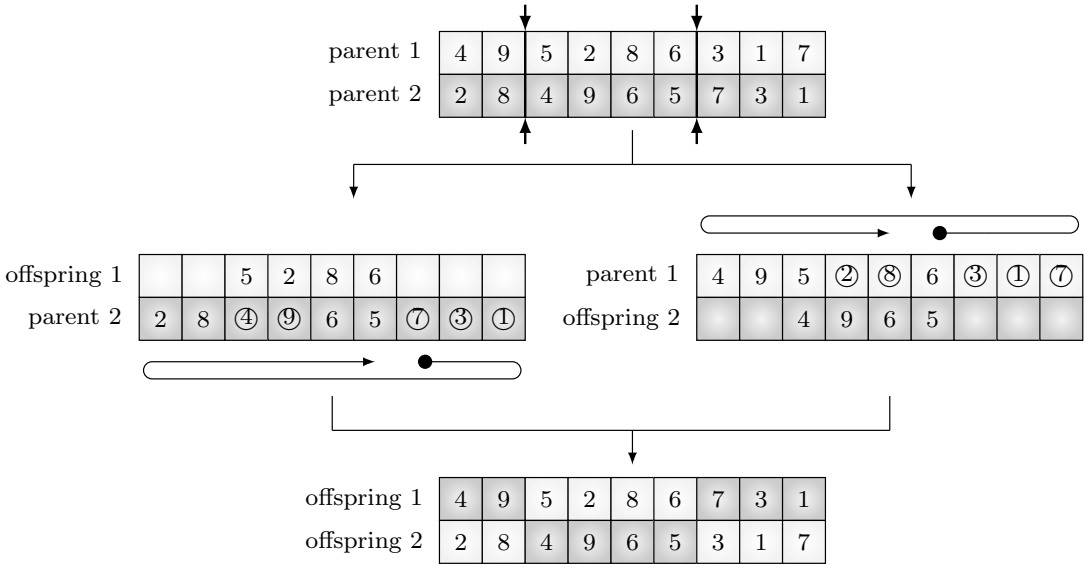

Figure 4.13: Permutation encoding — Crossover OX.

tion problems require maintaining adjacency relationships between elements, which strongly influence solution fitness.

Over the years, several variations of PMX have emerged in the literature. Here, we follow the definition provided by Whitley , which works as follows (see Figures 4.14, 4.15, and 4.16):

1. Randomly select two crossover points and copy the elements between them from the first parent (P1) into the first child.

2. Starting from the first crossover point, identify the elements between the two crossover points in the second parent (P2) that have not been copied.

3. For each of these elements (say i), find in the child the element (say j) that has been copied into its position from P1.

4. Place i in the position occupied by j in P2, knowing that j will not be placed there (since j is already present in the child).

5. If the position occupied by j in P2 has already been filled in the child by another element k, place i in the position occupied by k in P2.

6. After handling the elements within the crossover section, the remaining positions in this child can be filled from P2.

The second child is created similarly by reversing the roles of the parents.

Edge Recombination Crossover (ERX): The ERX crossover is based on the concept that the child should primarily inherit adjacency relationships from one parent to construct a solution. This operator has undergone several revisions, with the edge-3 crossover, as described by Whitley (), being the most commonly used version, aiming to preserve common adjacency relationships (edges).

To implement ERX, we first construct an adjacency table, which for each element stores the other elements it is adjacent to in both parents. A symbol '+' in an element adjacency list indicates that the edge exists in both parents. The process proceeds as follows:

1. Construct the edge table containing all adjacency lists.
2. Define a variable *current_element* representing the last element added to the child permutation (initially empty).
3. Randomly select an initial element and set it as the first element of the permutation.
4. Remove all occurrences of *current_element* from all adjacency lists.
5. Examine the elements in the adjacency list of *current_element*:
   - If there is a common adjacency relationship, select it as the next element;

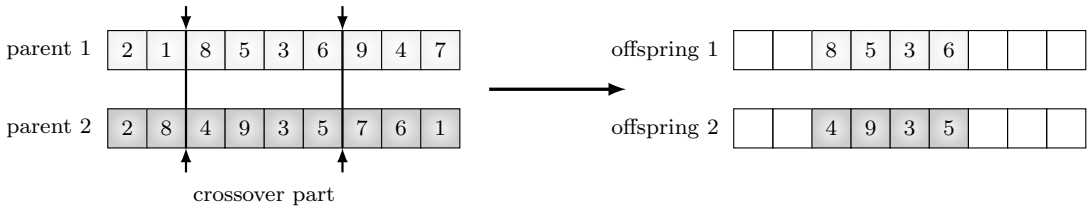

Figure 4.14: Permutation encoding — Crossover PMX, Step 1 : Copy the randomly selected portion from the first parent into the the first child.

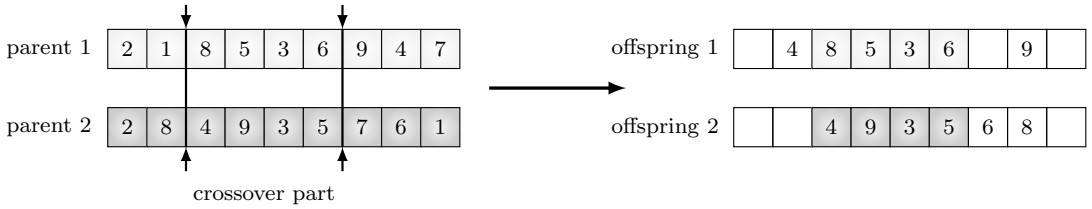

Figure 4.15: Permutation encoding - PMX Crossover, Step 2: For constructing the first child (C1) (respectively the second child [C2]), consider placing elements that appear in the portion between the two crossover points of P2 (resp. P1) but not in P1 (resp. P2). For C1, the position of 4 in P2 is occupied by 8 in C1, so we can place 4 in the position freed by 8 in P2. The position of 9 in P2 is occupied by 5 in C1, so we first look at the position occupied by 5 in P2, which is position 6. This position is already occupied by 6, so we place 9 in the position freed by 6 in P2. For C2, the position of 8 in P1 is occupied by 4 in C2, so we can place 8 in the position freed by 4 in P1. The position of 6 in P1 is occupied by 5 in C2, so we first look at the position occupied by 5 in P1, which is position 4. This position is already occupied by 9, so we place 6 in the position freed by 9 in P1. Finally, note that the values 6 and 5 are located between the two crossover points of the two parents.

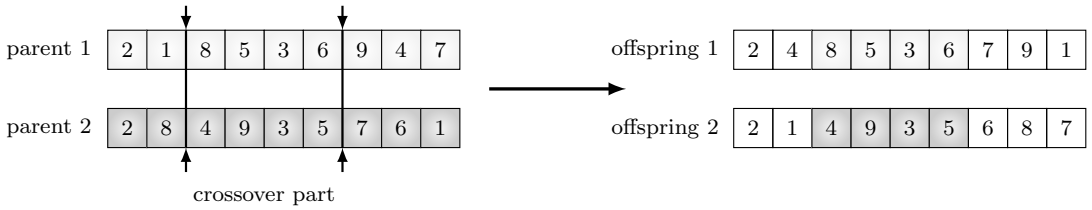

Figure 4.16: Permutation encoding — Crossover PMX, Step 3 : Copy the remaining elements from the second parent into the same positions in the child.

- Otherwise, choose the element with the fewest adjacency relationships in its own list;
- If multiple elements have the same minimum number of relationships, select one randomly.

6. If the adjacency list of *current_element* is empty, explore the other end of the child for extension; otherwise, randomly choose a new, unselected element.

It is clear that it is only in this latter case that new *edges* will be introduced.

The Edge-3 operator is illustrated by the following example where the parents are the same permutations used in the PMX example [2 1 8 5 3 6 9 4 7] and [2 8 4 9 3 5 7 6 1], resulting in the edge table shown in Table 4.I and the construction illustrated in Table 4.II. It is worth noting that only one child is created by this operator.

4.1.11 Replacement Strategies

The **replacement** phase concerns the selection of survivors from both the parent and offspring populations. As the population size is often constant, it allows for the selection of individuals according to a given strategy. This decision is often based on their fitness values, favoring those with better quality. Unlike parent selection, which is generally stochastic, survivor selection is often deterministic. Two common replacement strategies are:

Table 4.I: Edge-3 crossover — Example of the edge table.

Elements	Edges
1	2+, 6, 8
2	7, 1+, 8
3	5+ 9, 6
4	9+, 8, 7
5	8, 3+, 7
6	3, 7, 9, 1
7	4, 5, 2, 6
8	1, 2, 5, 4
9	6, 4+, 3

Table 4.II: Edge-3 crossover — Example of the permutation construction.

Candidates	Elements selected	Reason	Sub-permutation
All	4	Random	[4]
9+, 8, 7	9	Common edge	[4 9]
6, 3	3	Shorter list	[4 9 3]
5+ 6	5	Common edge	[4 9 3 5]
8, 7	7	Random (both candidates have two elements in the list)	[4 9 3 5 7]
2, 6	6	Shorter list	[4 9 3 5 7 6]
1	1	Only one element in the list	[4 9 3 5 7 6 1]
2+, 8	2	Common edge	[4 9 3 5 7 6 1 2]
8	8	Last element	[4 9 3 5 7 6 1 2 8]

- **Generational replacement**: Offspring systematically replace their parents to create the next population. This strategy is applied in a canonical GA as proposed by J. Holland.

- **Quasi-stationary or Elitist replacement**: In each GA generation, only part of the population is renewed. A particular case is when only one child is generated in each generation. In this case, two parents are chosen. Then, these two parents undergo crossover and mutation operators to generate two children. Subsequently, one of the two children replaces an individual in the population; typically, it replaces the least fit individual in the population (if it is of better quality). This elitist replacement strategy poses the risk of rapid convergence as a drawback.

Most recent GAs apply an elitist strategy (quasi-stationary GAs).

4.1.12 Parameterization and Termination criterion

Common parameters of GAs are (Talbi, 2009):

- **Population size** N: The larger the population size, the better the convergence towards "good" solutions. Sampling errors are more significant in small populations. However, the time complexity of GAs increases linearly with the

population size. A compromise must be found between the quality of solutions obtained and the overall algorithm execution time. In practice, the population size is generally between 20 and 100. Some theoretical results indicate that the population size should grow exponentially with the size of the individuals, leading to very large populations to be practical.

- **Mutation probability** p_m: This control parameter represents the percentage chance of mutation of a child. A high mutation probability will disrupt a given individual and the search will more likely be random. Generally, small values are recommended for the mutation probability ($p_m \in [0.001, 0.01]$). Usually, the mutation probability is initialized to $1/n$ where n is the size of the individuals. Thus, on average, only one decision variable (gene) is mutated.

- **Crossover probability** p_c: This control parameter represents the percentage chance of crossover between two parents. The crossover probability is generally set between moderate and high values (e.g., $[0.3, 0.9]$).

4.1.12.1 Termination criterion

GA is an iterative algorithm, so a suitable **termination criterion** must be defined. If the problem has a known optimal fitness level, probably a known optimum of the objective function, the termination condition would be the discovery of a solution with this fitness value. If it is known that the real-world problem model contains necessary simplifications, or may contain noise, one can accept a solution that reaches the optimal value with a given accuracy. However, GAs are stochastic and do not guarantee reaching such an optimum, so this condition might never be met, and the algorithm might never stop. Therefore, this condition must be extended with a condition that certainly stops the algorithm. The following options are commonly used for this purpose:

1. A maximum CPU time.

2. A maximum number of generations or of fitness evaluations.

3. No improvement in fitness over a given period of time (i.e., for a certain number of generations or fitness evaluations).

4. Lack of diversity in the population.

Technically, the termination condition in such cases is a disjunction: optimal value reached or specific condition satisfied. If the problem has no known optimum, a condition from the above list or a similar condition guaranteeing the algorithm termination is simply used.

4.2 Particle Swarm Optimization (PSO)

4.2.1 Origins and principles

Particle Swarm Optimization (PSO) is a swarm intelligence technique originally introduced by Kennedy and Eberhart in 1995 (Eberhart and Kennedy, 1995, Kennedy and Eberhart, 1995). This P-metaheuristic method draws inspiration from the flocking and swarming phenomena observed in animals such as migrating birds and schooling fish. These small animals exhibit intelligent behaviors to overcome their individually limited capabilities and leverage collective capacity to face predators or search for food.

The PSO algorithm is a general-purpose and relatively simple optimization strategy to implement. It was originally proposed for solving continuous variable optimization problems. Since its inception, it has been successfully applied to solve a wide variety of optimization problems.

In PSO, the population is called a "swarm", and the individuals, each representing a potential solution to the problem at hand, are called "particles". The process of searching for a high-quality solution involves a group of particles as vectors moving through the search space. Each particle is characterized by a vector representing its position (i.e., the effective representation of the solution) and a vector representing its change in position called "velocity". These two vectors determine the particle trajectory in the search space.

The PSO optimizer relies on the following two rules:

- Each particle remembers the best position it has encountered during its movements in the search space and tends to return to it.
- Each particle within the swarm is informed of the best position discovered by its neighborhood (adjacent particles) and tends to move toward it.

4.2.2 Basic model and formulas

In PSO, a swarm topology $\mathcal{N}$ is defined, where particles within a common neighborhood communicate with each other. This neighborhood is determined based on particle identifiers rather than topological information like Euclidean distances between points in the search space.

Let's m the swarm size and n the dimension of the search space (i.e., the number of decision variables). Each particle $i = 1, 2, \ldots, m$ has two main state vectors:

- The current position: $X_i = (X_{i,1}, X_{i,2}, \ldots, X_{i,n})$;
- The current velocity: $V_i = (V_{i,1}, V_{i,2}, \ldots, V_{i,n})$.

Additionally, each particle has an auxiliary vector to remember the best position it has encountered, denoted as $P_i = (P_{i,1}, P_{i,2}, \ldots, P_{i,n})$, with its corresponding fitness value ($f(P_i)$) denoted as $pbest_i$. Moreover, the particle is informed of the best-known position within its neighborhood, denoted as $P_g = (P_{g,1}, P_{g,2}, \ldots, P_{g,n})$, with its fitness value ($f(P_g)$) denoted as $gbest$. The adaptation of particles (i.e., the quality of their search positions) is guided by the objective function of the problem being solved.

The first PSO step is to initialize the swarm, where the position and velocity of each particle are randomly generated and/or heuristically set within permissible ranges. Subsequently, for each particle $i = 1, 2, \ldots, m$, the initial position is assigned to P_i. Then, the best position P_i among all particles is assigned to P_g. After this initialization phase, the algorithm iterates through an evolution procedure.

At each PSO iteration $(t + 1)$, the state (search position and velocity) of each particle in the swarm is modified based on its current state ($X_i(t)$ and $V_i(t)$), its current best position ($P_i(t)$), and the current global best position ($P_g(t)$). This modification follows two fundamental formulas for each dimension of the search space

$j = 1, 2, \ldots, n$:

$$V_{i,j}(t+1) = V_{i,j}(t) + r_1 c_1 (P_{i,j}(t) - X_{i,j}(t)) + r_2 c_2 (P_{g,j}(t) - X_{i,j}(t)). \tag{4.3}$$

$$X_{i,j}(t+1) = X_{i,j}(t) + V_{i,j}(t+1). \tag{4.4}$$

In these equations, c_1 and c_2 are positive parameters determining the influence of P_i and P_g in the velocity update, while r_1 and r_2 are uniformly randomly chosen numbers in the range $[0, 1]$. Additionally, a parameter $V_{max} > 0$ is used to limit each coordinate of V_i to the interval $[-V_{max}, V_{max}]$, ensuring that particles remain within the search space.

Figure 4.17 visualizes the modification of a particle search position during the optimization process.

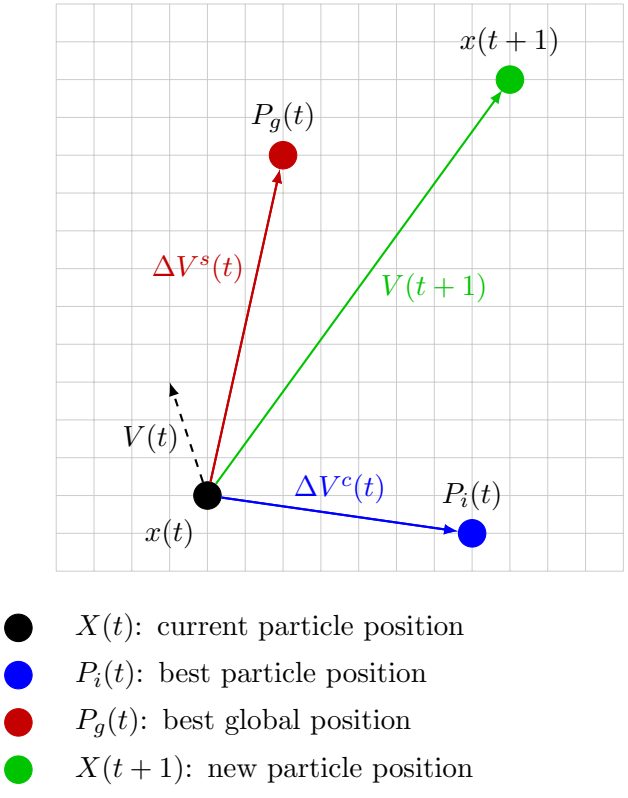

Figure 4.17: Particle search position update.

4.2.3 Particle movement and evaluation

After the particles move to their new positions, their updated fitness values are evaluated. Subsequently, the personal best position P_i for each particle and the global best position P_g are updated based on the fitness values of the new positions. For each particle i within the swarm, if the new position is better than its current position, P_i is updated to the new position, and $pbest_i$ takes the corresponding fitness value. Similarly, if the best P_i among all particles is of higher quality than the current P_g, P_g is replaced by this P_i, and *gbest* takes the corresponding fitness value. This movement and evaluation process repeats until a certain stopping criterion is met.

Figure 4.18 illustrates the overall flowchart of the PSO algorithm.

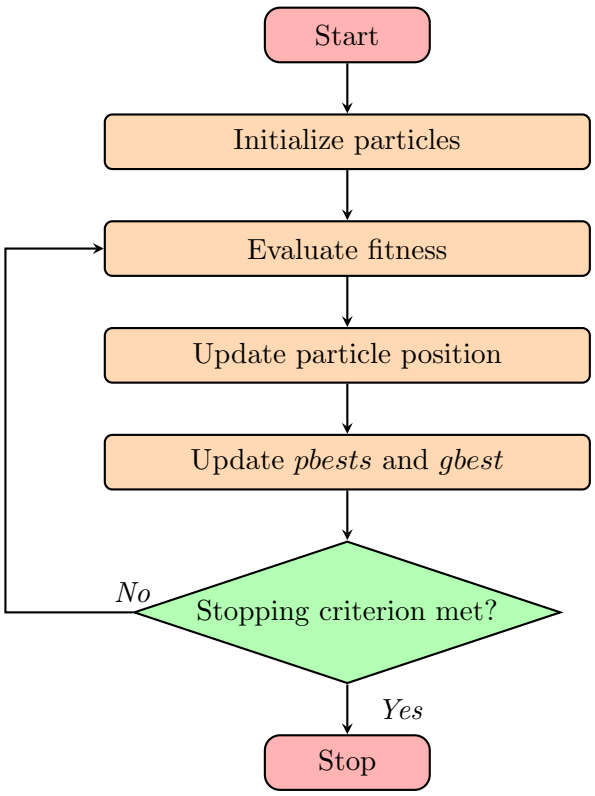

Figure 4.18: Overall flowchart of the PSO Algorithm.

The velocity vector directs the search process towards promising areas in the search space, reflecting the ”sociability” of the particles. The velocity update formula comprises three components:

- The first, known as *inertia*, corresponds to the current velocity value.
- The second, termed *cognitive influence*, represents the contribution of experiences gained by the particle individually.
- The third, referred to as *social influence*, accounts for the contribution of experiences gained collectively by the swarm.

These three components are illustrated in Figure 4.17 by the vectors $V(t)$, $\Delta V^c(t)$, and $\Delta V^s(t)$, respectively.

4.2.4 Variants of the PSO algorithm

The PSO algorithm is favored for its simplicity in implementation, quickly gaining popularity and widespread use across various practical problems. One of its key advantages is its minimal need for parameter tuning. However, it often faces the challenge of prematurely converging to suboptimal solutions. To address this issue and enhance its performance, researchers have proposed several modifications to the basic PSO algorithm (Poli et al., 2007). Let's explore three key alterations below:

4.2.4.1 Inertia weight PSO

Shi and Eberhart proposed the *inertia weight* model (Shi and Eberhart, 1998). This modification involves introducing an inertia weight (w) to adjust the velocity of each particle at each dimension and time step (t). The formula for updating the velocity is given by:

$$V_{i,j}(t+1) = w \times V_{i,j}(t) + r_1 c_1 (P_{i,j}(t) - X_{i,j}(t)) + r_2 c_2 (P_{g,j}(t) - X_{i,j}(t)) \qquad (4.5)$$

The inertia weight w serves to balance between exploration and exploitation. Initially set to a high value (close to 1), it encourages global exploration of the search space. Gradually reducing w within the range of $[0.2, 0.5]$ enables finer solutions to be obtained.

4.2.4.2 Constricted PSO

Clerc introduced the *constriction factor* model (Clerc, 1999, Clerc and Kennedy, 2002) to strike a balance between diversification and intensification. This extension incorporates a constriction factor (χ) to adjust the velocity before updating the particle position. The formula for the velocity update is given by:

$$V_{i,j}(t+1) = \chi \left(V_{i,j}(t) + r_1 c_1 (P_{i,j}(t) - X_{i,j}(t)) + r_2 c_2 (P_{g,j}(t) - X_{i,j}(t)) \right) \tag{4.6}$$

Smaller values of χ accelerate convergence but offer limited exploration, while higher values slow convergence but provide more exploration.

4.2.4.3 Comprehensive learning PSO (CLPSO)

The CLPSO variant was designed in Liang et al. (2006) to preserve the diversity of solutions and enhance the performance of the standard PSO. In CLPSO, all personal best positions *pbests* within the swarm can potentially be used as exemplars to modify the velocity vector of a particle, unlike the original PSO that uses the personal and global best positions. Additionally, each dimension of a particle in CLPSO can learn from different *pbests* for different dimensions for a few generations. Unlike the original PSO that learns from two exemplars simultaneously, CLPSO allows learning from multiple exemplars at different times. CLPSO aims to avoid falling into deep local optima, performs well on multimodal problems, and searches more promising regions to find the global optimum. The CLPSO variant has been shown to be effective in addressing exploration, exploitation, and convergence issues in optimization problems.

In CLPSO, each particle i adjusts its velocity vector on each dimension either according to its $pbest_i$ or to $pbest_k$ of another particle $k \neg i$. The selection of one or the other *pbest* is nondeterministic and performed according to the learning probability (Pc_i) of the particle. The Pc_i is set between the minimum (Pc_{min}) and maximum (Pc_{max}) learning probabilities as follows:

$$Pc_i = Pc_{min} + (Pc_{max} - Pc_{min}) \frac{\exp\left[\frac{10(i-1)}{m-1}\right] - 1}{\exp(10) - 1}, \tag{4.7}$$

Good results has been obtained by setting $Pc_{min} = 0.05$ and $Pc_{max} = 0.5$ (Liang et al., 2006). Small Pc_i values encourage strong exploitation ability (the particle will be attracted by its own success), while large values suggest strong exploration ability (the particle will follow the best search experience of other particles).

In CLPSO, each particle i uses a vector of exemplars $E_i = (E_{i,1}, E_{i,2}, \ldots, E_{i,n})$. Each element $E_{i,j} \in \{1, 2, \ldots, m\}$ gives which *pbest* (index) in the swarm particle i should follow for dimension j. Exemplars E_i are assigned as follows: for each dimension j, a random number ρ is first drawn from $[0, 1]$. If ρ is greater than Pc_i then, $E_{i,j}$ is set to i; otherwise, it is set to the index k ($k \neq i$) of the winner *pbest* of a tournament selection between two randomly picked particles. To ensure good convergence properties, each particle reassigns all its exemplars if its *pbest* is not improved for a maximum number of successive optimization steps termed the refreshing gap g. In Liang et al. (2006) g was set empirically to 7. In CLPSO, the velocity update takes the following form:

$$V_{i,j}(t+1) = V_{i,j}(t) + r_j \cdot c \cdot \left(pbest_{E_{i,j}}(t) - X_{i,j}(t)\right). \tag{4.8}$$

4.2.5 Binary PSO algorithm (BPSO)

Various PSO variants have been proposed for discrete variable optimization. The first binary version was introduced in (Kennedy and Eberhart, 1997). In this version, denoted BPSO (for Binary PSO), for each particle $i = 1, 2, \ldots, m$ in the swarm and each dimension $j = 1, 2, \ldots, n$ of the search space, the position value $X_{i,j}$ takes either 0 or 1, and the velocity value $V_{i,j}$ is interpreted as the probability of element $X_{i,j}$ having the value 0 or 1. Several functions can fulfill this condition, transforming the velocity value $V_{i,j}$ into a probability in the interval $[0, 1]$. The Sigmoid function is commonly used for this transformation and is defined as follows:

$$sig(V_{i,j}) = \frac{1}{1 + \exp\left(-V_{i,j}\right)} \tag{4.9}$$

The BPSO algorithm employs this function to modify the search position of a

particle $i = 1, 2, \ldots, m$ as follows:

$$X_{i,j} = \begin{cases} 1 & \text{if } \rho < sig(V_{i,j}) \\ 0 & \text{otherwise} \end{cases} \qquad j = 1, 2, \ldots, n \tag{4.10}$$

where ρ is a uniformly chosen random number in the range $[0, 1]$.

The velocity update formula (from Eq. 4.3) remains unchanged, except that $X_{i,j}$ is now a binary value, and thus $(P_{i,j} - X_{i,j})$ and $(P_{g,j} - X_{i,j})$ will take values of -1, 0, or $+1$. The original PSO algorithm for continuous variable optimization uses the parameter $V_{max} > 0$ to limit $V_{i,j}$ to the interval $[-V_{max}, V_{max}]$. This parameter is also utilized in the BPSO variant; however, as we can observe, it merely limits the extreme probability that $X_{i,j}$ takes the value 0 or 1. For example, with $V_{max} = 6.0$ (see Figure 4.19), the probabilities will range from 0.0025 to 0.9975. As a result, new binary strings will still be tested even after each bit has taken its best value. For large values of V_{max}, e.g., 10.0, it is unlikely that new binary strings will emerge. Thus, the role of V_{max} in BPSO is to set a limit for additional exploration after the algorithm converges. In other words, this parameter controls the final mutation rate of the binary vector. While a large value of V_{max} for the original PSO in continuous search spaces widens the region explored by a particle, the opposite occurs in the BPSO variant, a small value for V_{max} allows for a higher mutation rate.

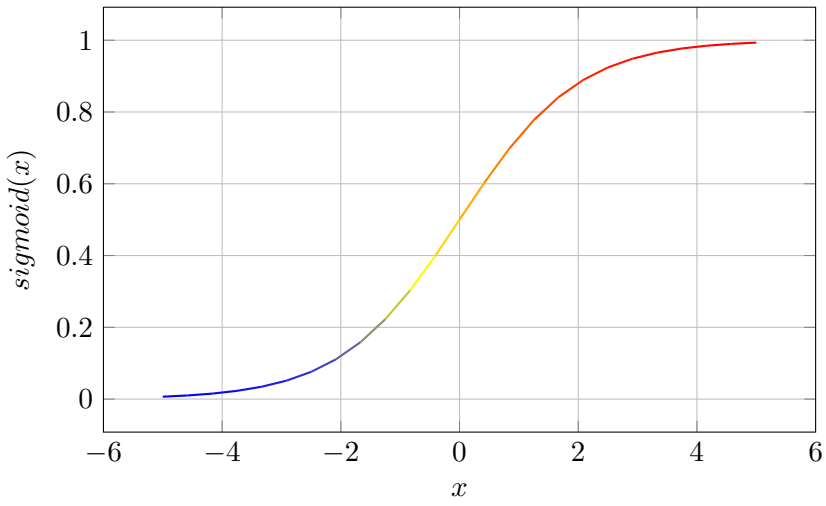

Figure 4.19: Sigmoid function.

4.2.6 PSO for permutation problems

In the basic PSO algorithm, particles are represented as points in the search space. All dimensions of the search space are independent, and therefore, position and velocity updates are performed independently for each dimension. However, this formulation is not applicable to permutation problems because the dimensions are not independent. It is possible to produce invalid solutions, with duplicated and/or omitted elements in the permutation. To address these conflicts, particle update strategies have been proposed in the literature for permutation encoding.

In the strategy proposed in (Hu et al., 2003), similar to the basic PSO algorithm, velocity represents the likelihood of the particle moving to a new position in the search space; it is highly probable that if velocity is large, the particle will shift to a new permutation. The velocity change formula remains unchanged. However, velocity is bounded to absolute values as it represents only the difference between particles. The update of a particle is then performed as follows: the velocity vector is first normalized by dividing it by the largest element. The velocity elements are then scaled to be between 0.0 and 1.0 and, consequently, interpreted as probabilities. Then, for each dimension of the search space, it is randomly determined if there will be a position swap with a probability given by the velocity. If a swap is required, the position will take the value of the same position of the best particle in the swarm, P_g, by permuting the values. This process is illustrated in Figure 4.20 (adapted from (Hu et al., 2003)).

As we can observe, with this modification strategy, the particle always follows the best position in the swarm, P_g; it could remain at its current position forever if it is identical to P_g. To overcome this issue, the authors proposed the random exchange (mutation) of a pair of positions in the permutation if the particle is identical to P_g.

Tasgetiren et al. in (Tasgetiren et al., 2004a, b) proposed a heuristic rule called Smallest Position Value (SPV) to transform the real representation of the position vector into a permutation. This rule is based on the random keys representation (Bean, 1994). To simplify the description of this transformation, we illustrate in Figure 4.21, for a particle i, the real position representation X_i and the corresponding permutation π_i (adapted from (Tasgetiren et al., 2004b)). In this example, according

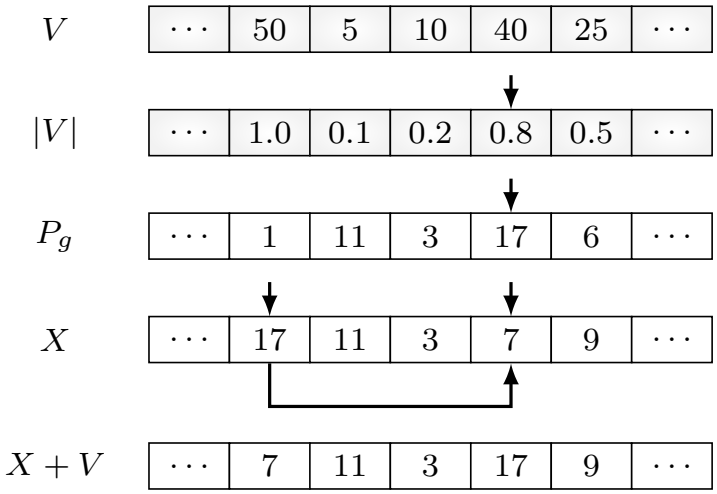

Figure 4.20: Position update for permutation encoding.

to the SPV rule, the smallest position value is $X_{i,5} = -1.20$, so dimension $j = 5$ is assigned to the first element $\pi_{i,1}$ of the permutation; the second smallest position value is $X_{i,2} = -0.99$, so dimension $j = 2$ is assigned to the second element $\pi_{i,2}$ of the permutation, and so on. In other words, dimensions are sorted according to the SPV rule, i.e., based on position values to construct the permutation.

Dimension, j	1	**2**	3	4	**5**	6
Valeur de position, $X_{i,j}$	1.8	**−0.99**	3.01	−0.72	**−1.20**	2.15
Element de permutation, $\pi_{i,j}$	**5**	**2**	4	1	6	3

Figure 4.21: Transformation of the real representation into a permutation according to the SPV rule.

4.3 Ant Colony Optimization (ACO)

4.3.1 Principles of ACO

ACO is a powerful optimization approach inspired by the foraging behavior of real ants. Just as ants communicate through pheromone trails to find the shortest path to food sources, ACO algorithms simulate this behavior to efficiently solve NP-hard optimization problems.

Imagine a colony of ants searching for food. Initially, they explore their surroundings randomly. When an ant discovers a food source, it evaluates its quality and quantity before returning to the nest, leaving behind a trail of pheromones. Other ants follow these trails, preferring paths with stronger pheromone concentrations. Over time, the collective efforts of the ants converge on the shortest path between the food source and the nest.

Similarly, in ACO algorithms, a population of artificial "ants" constructs solutions to optimization problems. These solutions are built step by step, guided by both problem-specific information (heuristics) and artificial pheromone trails. As ants deposit pheromones on favorable paths, ACO algorithms update pheromone levels to reinforce good solutions and explore new ones.

The key components of ACO algorithms are the pheromone model and the solution construction procedure. The pheromone model maintains information about the quality of solutions, while the construction procedure probabilistically generates new solutions based on this information. Through iterations of solution construction and pheromone updates, ACO algorithms gradually converge on high-quality solutions to complex optimization problems.

The beauty of ACO lies in its simplicity and adaptability. By mimicking the natural behavior of ants, ACO algorithms effectively tackle a wide range of combinatorial optimization problems, from the famous TSP to routing in telecommunications networks.

Now, let's delve deeper into the main steps of ACO algorithms as outlined in Algorithm 4.2, exploring how they translate the principles of ant behavior into efficient problem-solving strategies.

Algorithm 4.2: Ant Colony Optimization

```
1 InitializePheromone()
2 repeat
3 |   SolutionConstruction()
4 |   LocalSearch() {Optional}
5 |   UpdatePheromone()
6 until Termination condition satisfied
```

4.3.2 Solution construction procedure

During each algorithm iteration, each artificial ant construct a solution. The process begins with an empty solution $S^p = \langle\rangle$. At each step, ant choose the next solution component c' to add to the current partial solution $S^p \subseteq \mathcal{C}$ from the set of candidate components $\mathcal{N}(S^p) \subseteq \mathcal{C}$. This choice is based on a probability $Pr(c'|S^p)$, influenced by both the amount of pheromone $\tau_{c'}$ and the heuristic value $\eta_{c'}$ associated with the elements in $\mathcal{N}(S^p)$.

Many ACO algorithms use a probability rule initially proposed for Ant System (AS) (Dorigo et al., 1996) to select a component from $\mathcal{N}(S^p)$. This rule is given by:

$$Pr(c'|S^p) = \frac{{\tau_{c'}}^{\alpha} \cdot \eta_{c'}^{\beta}}{\sum_{c'' \in \mathcal{N}(S^p)} {\tau_{c''}}^{\alpha} \cdot \eta_{c''}^{\beta}} \qquad \forall c' \in \mathcal{N}(S^p). \tag{4.11}$$

Here, α and β are parameters determining how much influence pheromone values and heuristic values have on the choice probabilities. When α is close to zero, the process resembles a multiple-start greedy algorithm; when β is close to zero, solution construction relies solely on pheromone trails, with heuristic information being ignored.

4.3.3 Updating pheromone trails

Pheromone values are updated during the execution of an ACO algorithm to guide the search towards good solutions. This process involves two complementary steps: pheromone *evaporation* and pheromone *deposit*. In most ACO algorithms, pheromone evaporation reduces all pheromone values by a certain factor, defined by the following equation (López-Ibánez et al., 2015):

$$\tau_c = (1 - \rho) \cdot \tau_c \qquad \forall c \in \mathcal{C}. \tag{4.12}$$

Here, $\rho \in [0, 1]$ is the evaporation rate.

Pheromone deposit involves increasing the pheromone values of solution components that appear in a set of good solutions constructed in the current or previous iterations. The general form of this operation is given by the following equation

(López-Ibánez et al., 2015):

$$\tau_c = \tau_c + \sum_{s_k \in S^{upd} | c \in s_k} w_k \cdot F(s_k), \tag{4.13}$$

where S^{upd} is the set of good solutions chosen for pheromone deposit, w_k represents the weight assigned to solution $s_k \in S^{upd}$, and $F(s_k)$ is a function proportional to the quantity of s_k. If $f(s) < f(s')$ for a minimization problem, this function ensures that $F(s) \geq F(s')$. The quantity $w_k \cdot F(s_k)$ corresponds to the amount of pheromone deposited by solution s_k.

Several enhancements have been made to the initial ACO algorithm (Ant System (AS)), leading to different variants such as Elitist Ant System (EAS) (Dorigo et al., 1996), Ant Colony System (ACS) (Dorigo and Gambardella, 1997), Max-Min Ant System (MMAS) (Stützle and Hoos, 2000), Rank-based Ant System (RAS) (Bullnheimer et al., 1999), and Best-Worst Ant System (BWAS) (Cordon et al., 2000). For more information, interested readers are referred to (Dorigo and Stützle, 2010, López-Ibánez et al., 2015).

Chapter 5

Multi-Objective optimization with GAs

Multi-objective optimization (MOO) is a critical field within optimization that involves optimizing multiple conflicting objectives simultaneously. Unlike traditional single-objective optimization, where a single optimal solution is sought, MOO aims to find a set of solutions that represent a trade-off between competing objectives, known as the Pareto frontier or Pareto optimal solutions.

Multi-objective optimization is of paramount importance in various real-world applications where decision-making involves considering multiple conflicting criteria. These applications span diverse domains such as engineering design, finance, logistics, and resource allocation. By optimizing multiple objectives concurrently, MOO enables decision-makers to explore and understand the trade-offs inherent in complex decision problems, leading to better-informed decisions and improved outcomes.

In this chapter, we delve into the methodology of solving multi-objective optimization problems using GAs, a popular metaheuristic approach known for its effectiveness in exploring complex search spaces and handling multiple objectives simultaneously. Multi-Objective Genetic Algorithms (MOGAs) extend traditional GAs to handle multiple objectives by maintaining a diverse population of solutions that approximate the Pareto frontier. Notable MOGA algorithms such as NSGA-II (Non-dominated Sorting Genetic Algorithm II) and SPEA2 (Strength Pareto Evolutionary Algorithm 2) have emerged as solid references in the field, offering efficient and effective approaches for

solving multi-objective optimization problems.

Through a detailed exploration of MOO principles, MOGAs, and exemplary algorithms like NSGA-II and SPEA2, this chapter aims to equip readers with a comprehensive understanding of multi-objective optimization with GAs.

5.1 Multi-objective optimization problems

Multi-objective optimization, involves minimizing (or maximizing) multiple objective functions simultaneously. Unlike single-objective optimization, the optimal solution for a multi-objective problem is not a single solution but an assortment of solutions, called the set of Pareto optimal solutions or the Pareto front. The optimality of solutions is not defined in terms of simultaneous minimization or maximization of the objective functions but in terms of compromise with objectives of the optimization problem. Any compromise solution is optimal in the sense that no improvement can be made in one objective without degrading at least one other objective. This aspect arises from conflicts between the various objective functions.

Definition 5.1. ***Multi-objective optimization problem (MOP)****: Let* $\mathbf{x} = (x_1, x_2, \ldots, x_n)$ *denote the decision vector, and* $f_j(x)$ *represent the* j*-th objective function to be minimized or maximized, where* $j = 1, 2, \ldots, m$ *and* m *is the number of objective functions. The goal is to find a solution* $\mathbf{x}^*$ *that optimizes all objective functions simultaneously, subject to any constraints imposed by the MOP:*

$$\textit{Minimize (or maximize):} \quad f_j(x), \quad \textit{for } j = 1, 2, \ldots, m \tag{5.1}$$

$$\textit{Subject to:} \quad g_i(x) \leq 0, \quad \textit{for } i = 1, 2, \ldots, p \tag{5.2}$$

$$h_i(x) = 0, \quad \textit{for } i = 1, 2, \ldots, q \tag{5.3}$$

where $\mathbf{g}(\mathbf{x}) = (g_1(\mathbf{x}), g_2(\mathbf{x}), \ldots, g_p(\mathbf{x}))$ *represents the inequality constraints,* $\mathbf{h}(\mathbf{x}) = (h_1(\mathbf{x}), h_2(\mathbf{x}), \ldots, h_q(\mathbf{x}))$ *represents the equality constraints, and* $\mathbf{x}$ *belongs to the feasible solution region* $\mathcal{S}$*.*

Afterwards, we only discuss minimization objective functions, considering maxi-

mization objective functions as entirely reciprocal ($\min\{f_j(\mathbf{x})\} = -\max\{f_j(\mathbf{x})\}$).

Table 5.I gathers the terms that we will use later on.

Table 5.I: Terminology and definitions

Term	**Synonyms**	**Definition**
Decision vector	Decision parameters Decision variables	Set of variables of the MOP $\mathbf{x} = (x_1, x_2, \ldots, x_n)$
Objective function	Optimization criterion	Function to be minimized $f_j, j = 1, 2, \ldots, m$
Objective vector	Criterion vector	Set of objectives of the MOP $\mathbf{f}(\mathbf{x}) = (f_1(\mathbf{x}), f_2(\mathbf{x}), \ldots, f_m(\mathbf{x}))$
Search space	Parameter space	Set of values that can be taken by the variables, denoted Ω
Feasible space	Solution space Decision space	Set of feasible solutions satisfying all problem constraints, denoted $\mathcal{S}$
Objective space	Criterion space	Set of all feasible solution fitness vectors, denoted $\Lambda = \mathbf{f}(\mathcal{S})$
Solution	Configuration Point	A point $\mathbf{x}$ in the feasible space or its image $\mathbf{y} = \mathbf{f}(\mathbf{x})$ in the objective space

Figure 5.1 illustrates the different spaces for a bi-criteria problem with two decision variables.

5.2 Importance and real-world examples

Multi-objective optimization is of paramount importance in various fields due to its ability to find trade-off solutions that balance conflicting objectives. This is particularly valuable in decision-making processes where multiple criteria need to be considered simultaneously. Some common applications of multi-objective optimization include:

- **Engineering design**: Designers often need to optimize multiple objectives, such as cost, performance, reliability, and sustainability. For example, in designing a new car, engineers may want to minimize both fuel consumption and emissions while maximizing safety and performance.

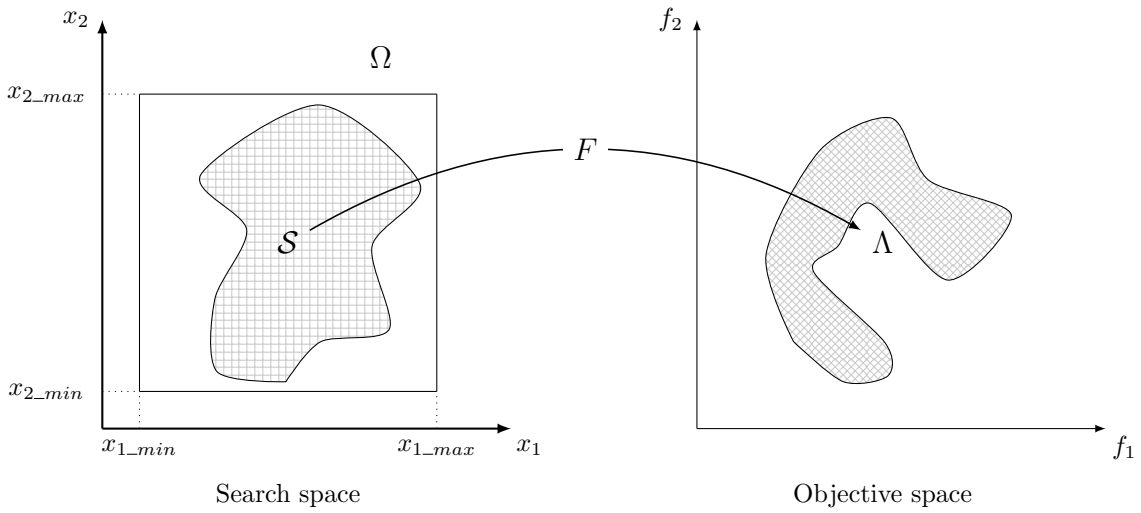

Figure 5.1: Search space, feasible region, and objective space.

- **Finance and investment**: Investors may have multiple objectives, such as maximizing returns while minimizing risk or achieving portfolio diversification. Multi-objective optimization techniques can help investors make informed decisions that balance these conflicting objectives.

- **Healthcare**: Treatment planning often involves optimizing multiple conflicting objectives, such as maximizing patient survival while minimizing treatment side effects. Multi-objective optimization methods can assist healthcare professionals in developing personalized treatment plans for patients.

- **Supply chain management**: Companies aim to optimize various objectives, including cost, lead time, inventory levels, and customer satisfaction. Multi-objective optimization approaches can optimize the supply chain network to achieve a balance between these objectives.

These are just a few examples of the wide-ranging applications of multi-objective optimization in various domains.

5.3 Pareto dominance

In this section, we will introduce the basic concepts of multi-objective optimization, including Pareto dominance, Pareto optimality, the Pareto front, dominance relations, and trade-off solutions.

For single-objective problems, the sought-after solution is clearly defined: the one that minimizes the objective function. In the case of MOPs, identifying the best compromises among all the considered objective functions requires defining a partial order relation among the feasible solutions in the search space, called the dominance relation.

The most well-known and commonly used is the Pareto dominance relation, formulated by the economist Vilfredo Pareto at the end of the 19th century. A solution is said to dominate another solution if it is at least as good as the other solution in all objectives and strictly better in at least one objective.

In order to formally define this notion, the usual comparison relations $=, <, \leq$ are extended to vectors. Let $\mathbf{x}^{(1)} = (x_1^{(1)}, x_2^{(1)}, \ldots, x_n^{(1)})$ and $\mathbf{x}^{(2)} = (x_1^{(2)}, x_2^{(2)}, \ldots, x_n^{(2)})$ be two solutions in the search space, and let $\mathbf{f}(\mathbf{x}^{(1)}) = (f_1(x_1^{(1)}), f_2(x_2^{(1)}), \ldots, f_m(x_n^{(1)})$ and $\mathbf{f}(\mathbf{x}^{(2)}) = (f_1(x_1^{(2)}), f_2(x_2^{(2)}), \ldots, f_m(x_n^{(2)})$ denote their respective objective vectors.

Definition 5.2. ***Pareto Dominance****: Solution* $\mathbf{x}^{(1)}$ *dominates solution* $\mathbf{x}^{(2)}$*, denoted* $\mathbf{x}^{(1)} \prec \mathbf{x}^{(2)}$*, if and only if:*

1. $f_j(\mathbf{x}^{(1)}) \leq f_j(\mathbf{x}^{(2)})$ *for all objectives* $j = 1, 2, \ldots, m$*, and*

2. *There exists at least one objective* k *such that* $f_k(\mathbf{x}^{(1)}) < f_k(\mathbf{x}^{(2)})$.

Solutions that do not dominate each other are said to be *equivalent* or *incomparable*. Figure 5.2 illustrates, using a two-dimensional example, the Pareto dominance relation. For a point $\mathbf{y} = (f_1(\mathbf{x}), f_2(\mathbf{x}))$ in the objective space, we distinguish three zones:

- The preference zone contains the points (solutions in the objective space) dominated by point $\mathbf{y}$.

- The dominance zone contains the points dominating point $\mathbf{y}$.

- The equivalence zone is the area containing points that are incomparable to point $\mathbf{y}$.

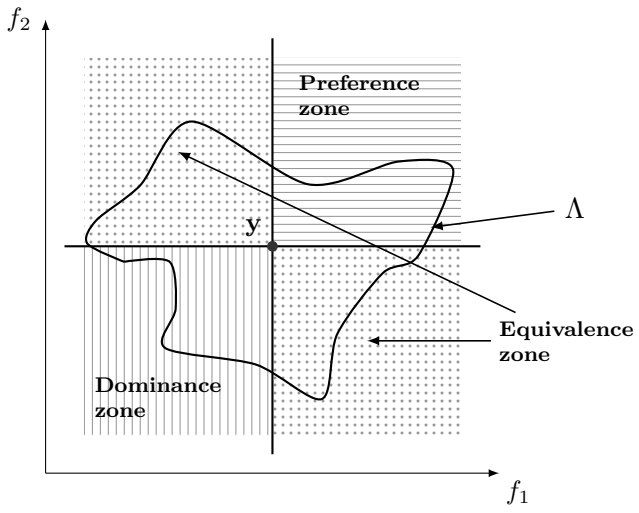

Figure 5.2: Pareto dominance.

In addition to Pareto dominance, we can define the notions of local and global dominance. A solution is locally non-dominated if there does not exist any other solution in its neighborhood that dominates it. On the other hand, a solution is globally non-dominated if it is not dominated by any other solution in the entire search space.

Definition 5.3. ***Local Pareto dominance****: A solution is locally non-dominated if there does not exist any other solution in its neighborhood that dominates it. Mathematically, a solution* $\mathbf{x}$ *is locally non-dominated if:*

$$\nexists \mathbf{x}' \in N(\mathbf{x}) \text{ such that } \mathbf{f}(\mathbf{x}') \prec \mathbf{f}(\mathbf{x}), \tag{5.4}$$

where $N(\mathbf{x})$ *represents the neighborhood of solution* $\mathbf{x}$*.*

Definition 5.4. ***Global Pareto dominance****: A solution is globally non-dominated if it is not dominated by any other solution in the entire search space. Mathematically, a solution* $\mathbf{x}$ *is globally non-dominated if:*

$$\nexists \mathbf{x}' \in \mathcal{S} \text{ such that } \mathbf{f}(\mathbf{x}') \prec \mathbf{f}(\mathbf{x}), \tag{5.5}$$

where $\mathcal{S}$ denotes the entire feasible search space.

Figure 5.3 illustrates the concept of local and global Pareto dominance in two dimensions.

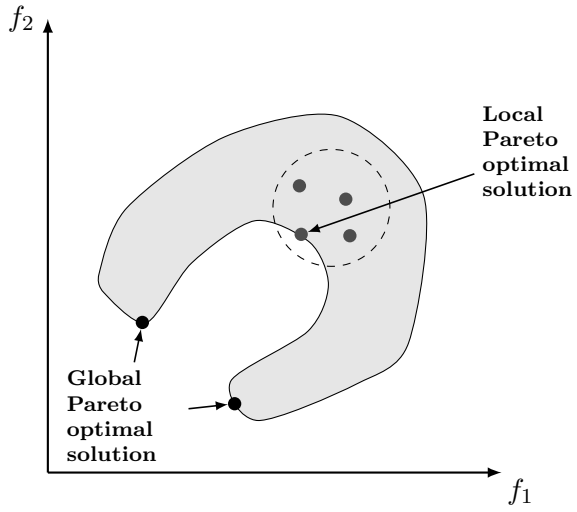

Figure 5.3: Local and global Pareto dominance.

A solution that is not dominated by any other solution in the feasible region is known as a ***Pareto optimal solution***. The set of all Pareto optimal solutions forms the ***Pareto front***, representing the trade-off between conflicting objectives.

Definition 5.5. ***Pareto optimal set****: The Pareto optimal set, denoted by $\mathcal{P}^*$, consists of all solutions that are not dominated by any other solution in the feasible region. Mathematically, the Pareto optimal set is defined as:*

$$\mathcal{P}^* = \{\mathbf{x} \in \mathcal{S} \mid \nexists \mathbf{x}' \in \mathcal{S} \text{ such that } \mathbf{f}(\mathbf{x}') \prec \mathbf{f}(\mathbf{x})\}. \tag{5.6}$$

Definition 5.6. ***Pareto front****: The Pareto front, denoted by PF, represents the set of objective vectors corresponding to all Pareto optimal solutions. Mathematically, the Pareto front is defined as:*

$$\mathcal{FP} = \{\mathbf{f}(\mathbf{x}) \mid \mathbf{x} \in \mathcal{P}^*\}. \tag{5.7}$$

The global Pareto front gathers all the best possible compromises for the MOP. This set is also called the ***optimal Pareto front****, or more generally, the* ***compromise***

surface *for higher-dimensional problems. Solving a POM is to find solutions from the global Pareto front.*

5.4 Solution Approaches for MOPs

The goal of solving MOPs is to assist decision-makers in selecting a suitable solution from the Pareto optimal set. A crucial aspect of multi-objective optimization revolves around how the resolution method interacts with the decision-maker.

In multi-objective optimization, Pareto optimal solutions cannot be globally ranked. Therefore, it falls upon the decision-maker to specify their preferences to establish a total order among the Pareto solutions, enabling them to select their preferred solution. This leads to the classification of resolution approaches into three main categories based on when the decision-maker's preferences are incorporated into the process (Miettinen, 1999):

- **A Priori Methods** (*Preferences* $\rightarrow$ *Search*): In these approaches, the decision-maker is required to define the compromise they aim to achieve before initiating optimization. This often involves transforming the multi-objective problem into a single-objective one by combining the different objective functions into a single scalar function using objective aggregation methods. The reduced problem becomes:

$$\min \mathbf{v}(\mathbf{f}(\mathbf{s})), \quad s \in \mathcal{S} \tag{5.8}$$

  where $\mathbf{v}$ is the utility function expressing the decision-maker's preferences. For instance, if the utility function is linear:

$$\mathbf{v}(\mathbf{f}(\mathbf{s})) = \sum_{i=1}^{m} \lambda_i f_i(s), \tag{5.9}$$

  the decision-maker must evaluate the weights $\lambda_i \geq 0$ for each objective function f_i *a priori*. However, defining the utility function before optimization can be challenging, requiring the decision-maker to have a deep understanding of the problem.

- **Progressive or Interactive Methods** (*Search* $\leftrightarrow$ *Preferences*): Here, the decision-maker gradually refines their choice of compromises during optimization. Based on insights gained from problem resolution, the decision-maker can clearly define their preferences, which are then considered by the resolution method in subsequent search steps. This iterative process continues until the decision-maker is satisfied with the results.

- **A Posteriori Methods** (*Search* $\rightarrow$ *Preferences*): These methods do not require the decision-maker to specify preferences or interact during optimization. Instead, they provide a set of compromise solutions at the end of the search, allowing the decision-maker to choose *a posteriori*. This approach is practical for problems with a small number of objectives and a reduced Pareto optimal set. However, selecting a solution from this set based on the decision-maker's preferences can be complex.

Each solution approach has its advantages and disadvantages, which depend on the problem properties and the decision-maker's skills.

Advantages and disadvantages of different methods

A Priori methods: These methods are effective when the objective space exhibits convexity properties.

Progressive or interactive methods: While these methods allow the decision-maker to refine their preferences during optimization, they can monopolize the decision-maker's attention throughout the process. This can be disadvantageous if evaluating objective functions is time-consuming or if frequent decision-maker interventions are required.

A Posteriori methods: The main advantage is the reproduction of a set of optimal solutions, but selecting a solution from this set based on the decision-maker's preferences can be challenging. These methods often use metaheuristic algorithms, such as GAs, which are well-suited for complex optimization problems.

In the remainder of this chapter, we focus on multi-objective Genetic Algorithms (MOGAs) using a Pareto-based approach.

5.5 Main components of MOGAs

The aim of solving a MOP using a GA is to converge towards a diverse set of solutions along the Pareto front. In addition to the common elements found in single-objective GAs, MOGAs tailored for Pareto optimization typically incorporate three crucial search components (Talbi, 2009):

- **Fitness assignment (or Ranking):** This component plays a pivotal role in steering the search towards Pareto optimal solutions by evaluating the compromise between all objectives. It assigns a scalar value to each new solution (represented as an objective vector) to determine its fitness.
- **Maintaining diversity (or Nesting):** The primary objective here is to generate a diverse set of Pareto optimal solutions across both parameter and objective spaces. This diversity helps in exploring different regions of the Pareto front.
- **Elitism:** The concept of elitism involves preserving the best (Pareto optimal) solutions throughout the evolutionary process. These elite solutions are retained to expedite convergence and enhance the overall performance of the algorithm.
- **Hybridization:** Moreover, when MOGAs are combined with other optimization techniques to bolster their effectiveness, they are referred to as hybrid MOGAs.

Figure 5.4 provides a visual representation of a MOGA incorporating these essential mechanisms. The distinctions between various MOGAs primarily stem from the implementation of these mechanisms and the strategy employed for parent selection.

5.5.1 Ranking procedures

Ranking procedures play a crucial role in multi-objective optimization by allowing the comparison and ranking of candidate solutions based on their performance with respect to multiple objectives. Unlike simplifying approaches that transform the problem into a single objective, ranking methods based on Pareto optimality directly

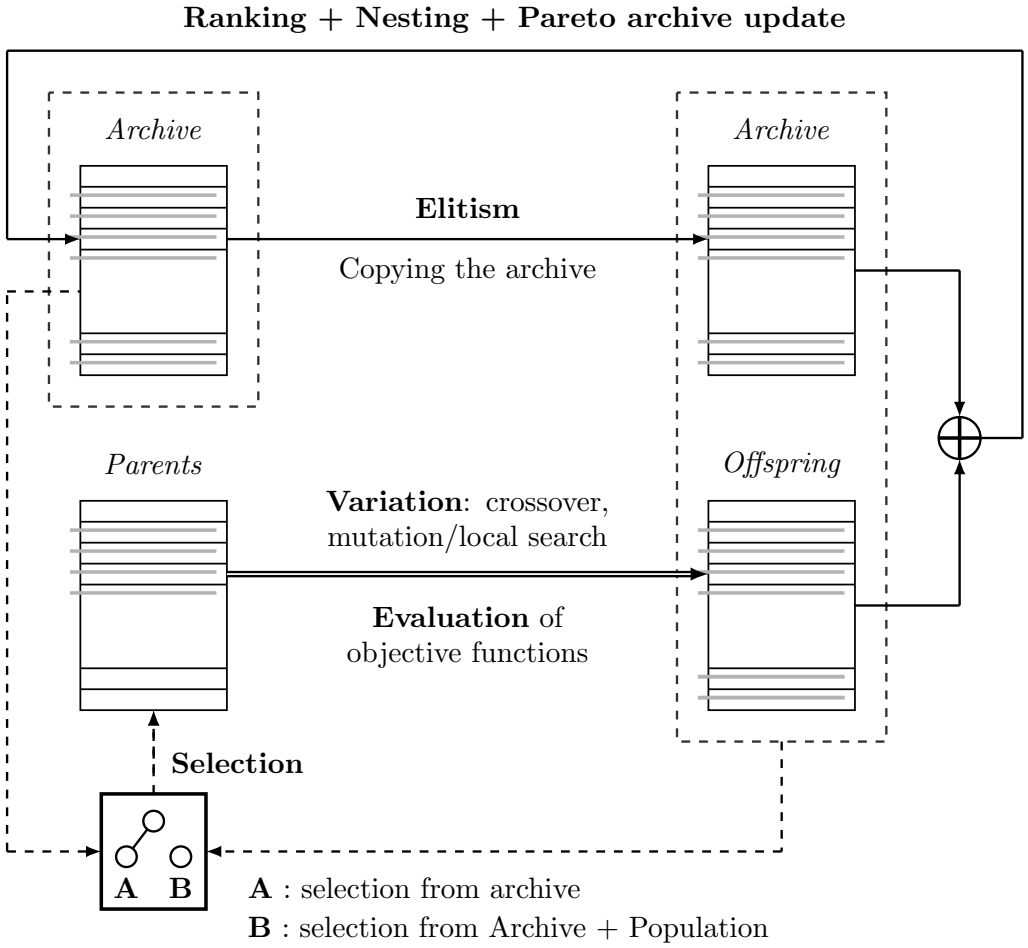

Figure 5.4: Structure of an elitist MOGA with nesting (inspired by (Regnier, 2003)).

evaluate each solution by considering its domination relative to others, providing a comprehensive view of solution quality.

5.5.1.1 The Significance of the Pareto Approach

The Pareto approach, initially proposed by Goldberg (Goldberg, 1989), offers a significant advantage by enabling the generation of optimal solutions in concave regions of the Pareto front. Unlike non-Pareto methods, it considers all objectives without simplifying them into a single criterion, allowing for better exploration of trade-offs between different dimensions of the problem.

5.5.1.2 Ranking Techniques

Several ranking techniques have been developed in the literature, each with its own specifics. Among these, the most widely used method is the Non-dominated Sorting Genetic Algorithm (NSGA), which ranks the population into multiple Pareto fronts based on their dominance.

5.5.1.3 Advantages of Pareto Approaches

One of the major advantages of Pareto approaches is their ability to evaluate the quality of a solution considering the entire population. Unlike methods based on utility functions, which assign absolute values to solutions, Pareto approaches evaluate the quality of a solution relative to its global context, providing a more nuanced perspective of solution performance.

In summary, ranking procedures based on the Pareto approach are essential in the multi-objective optimization process by enabling the comparison and selection of the best solutions based on their suitability with respect to all objectives. They offer a comprehensive view of solution quality and facilitate the search for optimal solutions in a multi-dimensional space.

5.5.1.4 NSGA Ranking Procedure

The NSGA ranking procedure was originally proposed by Goldberg in (Goldberg, 1989) and implemented by Srinivas and Deb in (Srinivas and Deb, 1994). It is employed to establish an ordering among solutions within a population.

Initially, all non-dominated individuals in the population are assigned rank 1, forming the first front $\mathcal{F}_1$. Then, individuals dominated solely by the individuals in $\mathcal{F}_1$ receive rank 2, forming the second front $\mathcal{F}_2$. In general, an individual receives rank k if and only if it is dominated solely by individuals belonging to the union $\mathcal{F}_1 \cup \mathcal{F}_2 \cdots \cup \mathcal{F}_{k-1}$. This process is repeated until all individuals in the population have been assigned a rank.

Figure 5.5 visually illustrates the concept of dominance rank in the case of bi-objective problems. Figure 5.6 demonstrates the execution of this ranking procedure

on a set of solutions.

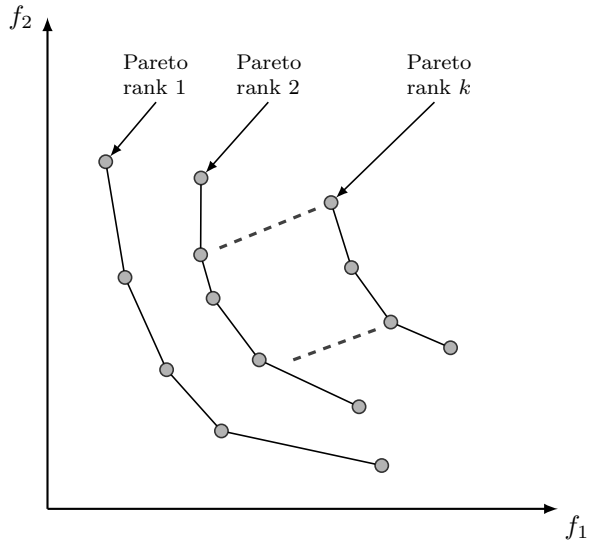

Figure 5.5: Goldberg ranking (NSGA), for a bi-objective MOP.

Algorithm 5.1 provides the pseudo-code of the Pareto ranking algorithm. In this algorithm, the variable P represents the set of points (p_i) for which we want to identify the different ranks. This intuitive algorithm has an average complexity of $O(mN^3)$, where m is the number of objective functions and N is the population size. The authors of NSGA-II (Deb et al., 2002) improved this algorithm and reduced its complexity to $O(mN^2)$.

5.5.2 Niche formation

It is important to recall that the goal in Pareto-based solving of a MOP is to obtain diversified solutions on the Pareto optimal subspace. Ranking methods, like the NSGA method presented in the previous section, ensure convergence towards Pareto optimal solutions. However, diversification of solutions is not taken into account, which makes it impossible to discover the entire Pareto front. To achieve a diversified population, in both parameter and/or objective spaces, ranking methods must be combined with niche formation and maintenance techniques.

In general, diversification techniques penalize solutions with high solution densities around them. They can be classified into three categories (Talbi, 2009, Zitzler et al.,

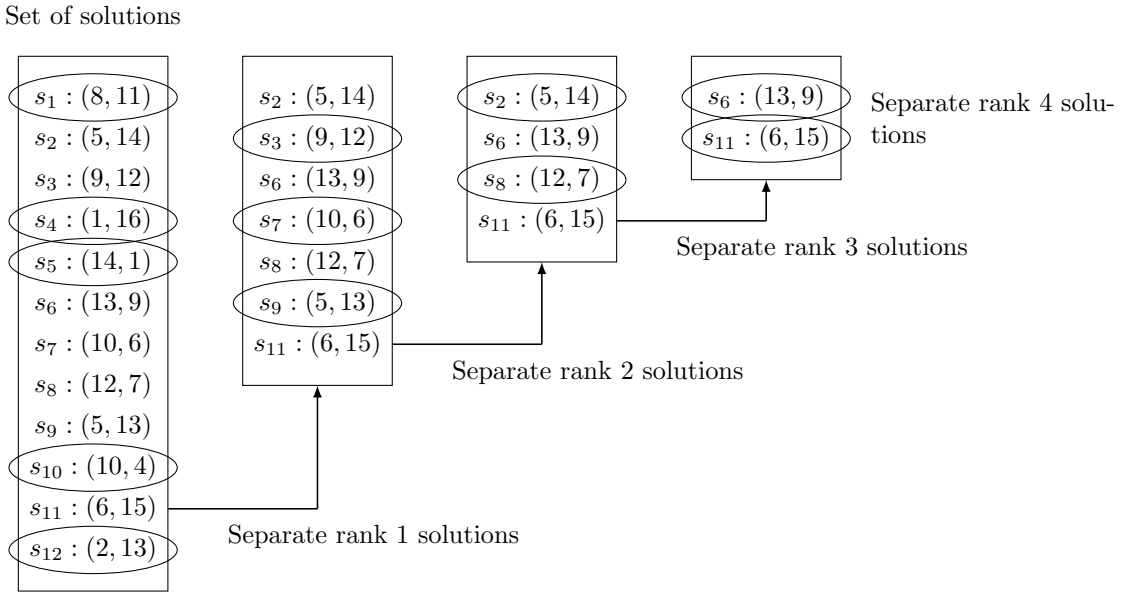

Figure 5.6: Illustration of the NSGA ranking procedure for a bi-objective MOP (inspired by (Talbi, 2009)).

Algorithm 5.1: Pareto Ranking (NSGA)

```
1  CurrentRank ← 1
2  while (|P| ≠ 0) do
3  |  foreach p_i ∈ P do
4  |  |  if (p_i is non-dominated) then
5  |  |  |  Rank(p_i) ← CurrentRank
6  |  |  end
7  |  end
8  |  foreach p_i ∈ P do
9  |  |  if (Rank(p_i) = CurrentRank) then
10 |  |  |  Remove p_i temporarily from P
11 |  |  end
12 |  end
13 |  CurrentRank ← CurrentRank + 1
14 end
```

2004):

- **Neighborhood Methods**: As we will see later in this chapter, algorithms like SPEA2 and NSGA-II are characterized by density estimates based on neighborhood principles. SPEA2 considers the distance between a solution s_i and

its k-th nearest neighbor to estimate density. NSGA-II is characterized by a density estimation related to the environment of each solution, called crowding distance.

- **Performance Sharing Methods**: These methods define the neighborhood of a solution according to a kernel function K, called a sharing function, which takes the distance between two solutions as input. For each solution s_i, the distances $d(s_i, s_j)$ between s_i and all other solutions s_j in the population *pop* are calculated. Then, the estimated density of solution s_i, called the niche count, is given by the sum of the values of the sharing function K applied to all distances: $m(s_i) = \sum_{s_j \in pop} K(d(s_i, s_j))$. The values of function m should be high for solutions that are close to each other and low for isolated solutions. Thus, the shared fitness f' of a solution s_i is equal to the original fitness value $f(s_i)$ divided by the solution density $m(s_i)$: $f'(s_i) = \frac{f(s_i)}{m(s_i)}$. In this way, the performance of a solution will be degraded depending on the density of nearby solutions. One of the most commonly used sharing functions is the one proposed by Goldberg and Richardson in (Goldberg et al., 1987): $sh(d(s_i, s_j)) = \begin{cases} 1 - \frac{d(s_i,s_j)}{\sigma} & \text{if } d(s_i, s_j) < \sigma \\ 0 & \text{otherwise} \end{cases}$. This function returns a value in the interval $]0, 1]$ if the two solutions are close, and 0 if the distance between them exceeds a certain threshold σ. Such a sharing method was used in the NSGA algorithm (Srinivas and Deb, 1994), the precursor of NSGA-II.

- **Histograms**: This approach involves partitioning the search space into several hypergrids defining neighborhoods. The population density around a solution is estimated by the number of solutions occupying the same hypercube.

It is noteworthy that one of the most important issues in diversity maintenance approaches concerns distance measurement. Several measures can be used, such as Hamming distance or Euclidean distances. Moreover, distance can be calculated in both parameter and/or objective spaces. Generally, niche formation and maintenance are performed in the objective space due to their simplicity.

5.5.3 Elitism

In single-objective cases, elitism involves keeping the best individual for future generations to prevent its possible loss during the application of genetic variation operators. Implementing elitism in the context of multi-objective problem solving (MOPs) is more challenging than in the single-objective case. This difficulty arises from the fact that the best solution is not a single solution, but a set of solutions. The idea is therefore to retain a limited number of these optimal solutions in a secondary population, called an *archive*, of constant size associated with the current population (Zitzler et al., 2000).

As described in (Talbi, 2009), if elitism is used only to prevent the possible loss of obtained Pareto optimal solutions, it is called a passive elitism strategy. In this case, the archive is considered as a separate secondary population that has no impact on the search process. Passive elitism will only guarantee non-degrading performance of the algorithm in terms of approximating the Pareto front. On the other hand, if elitism is used in the search process, meaning elites are used to generate new solutions, it is called an active elitism strategy. Active elitism enables rapid and robust convergence to a good approximation of the Pareto front (Zitzler et al., 2000). Most recent MOGAs apply active elitism strategies.

As shown in Figure 5.4, parent individuals can be selected from the archive (as is the case for NSGA-II and SPEA2) or from the population formed by archive members and children from the previous generation. It should be noted that the sizes of elite populations, parent individuals, and children can be identical or distinct.

At each generation, the archive is updated based on newly explored solutions. Generally, archive members are chosen using a niche method that preserves the most diverse solutions across the entire Pareto front. Therefore, the archive retains the most representative Pareto optimal solutions obtained throughout the search.

5.5.4 Hybridization

Similar to the single-objective case, another important mechanism to enhance the performance of a multi-objective metaheuristic is to hybridize it with other optimiza-

tion methods. Hybrid multi-objective metaheuristics have been recognized for over a decade as competitive approaches for handling large MOPs (Ehrgott and Gandibleux, 2008, Talbi, 2015). Talbi in (Talbi, 2015) classified hybrid metaheuristics in the domain of MOP solving into three main categories:

- **Hybridization of metaheuristics with (meta)heuristics**: This is the most common form of hybridization. It involves combining concepts and search components from different (meta)heuristics to produce an effective algorithm.
- **Hybridization of metaheuristics with exact methods**: In this case, pure metaheuristics are combined with mathematical programming methods, such as branch and bound and branch and cut methods, to explore very large neighborhoods or to complete partial solutions.
- **Hybridization of metaheuristics with data mining and learning techniques**: This category includes approaches that choose and adapt the control parameters of an algorithm without user intervention, approaches that apply search operators (e.g., recombination operators of population metaheuristics, neighborhood structures of single-solution metaheuristics) adaptively, and approaches that transform the initial MOP into a reduced MOP by decomposing it into several subproblems or by adding new constraints.

For MOGAs, hybridization often involves substituting the mutation operator with a local search procedure (i.e., memetic algorithms). This procedure is applied to generate one or more neighbors of the individual that will undergo mutation. Among several multi-objective memetic algorithms proposed in the literature, we can mention the algorithm presented in (Barichard and Hao, 2003), which combines a MOGA with tabu search (applied to the multi-objective knapsack problem), and the algorithm proposed in (Meunier et al., 2000), which applies a local search procedure to improve the population of a MOGA (applied to the network design problem).

Types of MOEAs

There are several types of MOEAs, each with its unique characteristics and strategies. Some popular MOEAs include:

- NSGA-II (Nondominated Sorting Genetic Algorithm II)
- SPEA2 (Strength Pareto Evolutionary Algorithm 2)
- MOEA/D (Multi-Objective Evolutionary Algorithm Based on Decomposition)
- PESA-II (Pareto Envelope-based Selection Algorithm II)
- MOGA (Multi-Objective Genetic Algorithm)

Each MOEA employs different mechanisms and strategies to explore and exploit the solution space effectively. In the subsequent sections, we will delve deeper into some of these prominent MOEAs, discussing their principles, algorithms, and applications in solving real-world multi-objective optimization problems.

5.6 NSGA-II (Nondominated Sorting Genetic Algorithm II)

NSGA-II (**?**) is an elitist version of the NSGA (Non-dominated Sorting Genetic Algorithm) proposed by Srinivas and Deb (**?**), which directly implements Goldberg's ranking technique (Goldberg, 1989). NSGA-II employs a selection operator based on crowding distance in the objective space to maintain diversity and promote niching.

5.6.1 General operating principle

As depicted in Figure **??**, NSGA-II manages two populations of equal size N. The first population P_t (of elite solutions) contains the best individuals encountered up to generation t, and the second population Q_t (of offspring) contains individuals generated from the elite population. The initial step of a generation t involves merging

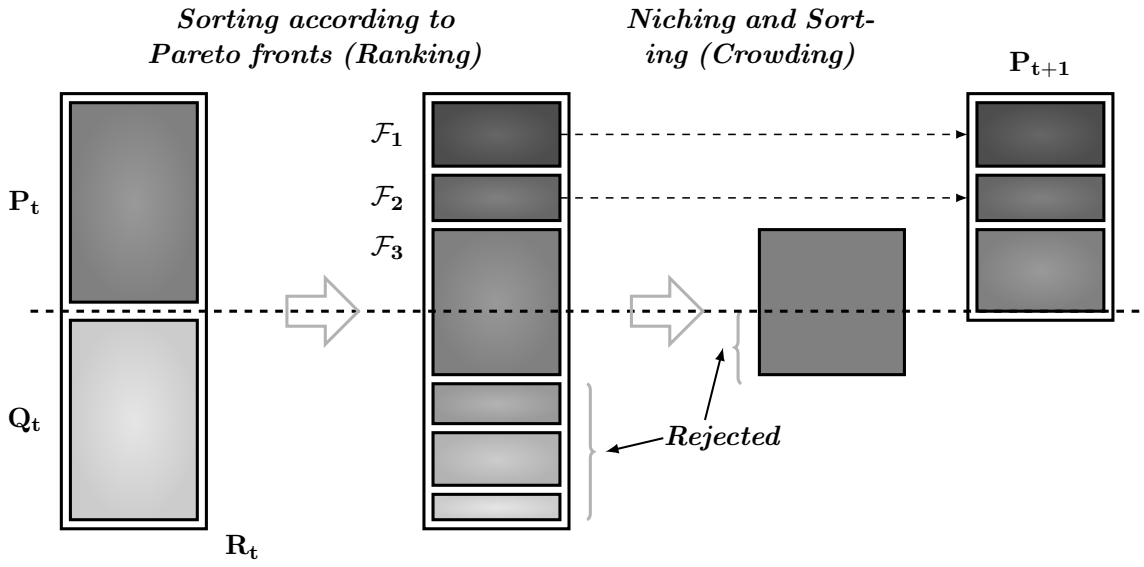

Figure 5.7: Overall operation of the NSGA-II algorithm (according to (Deb et al., 2002)).

the two populations $R_t = P_t \cup Q_t$ and then ranking individuals based on their Pareto front (i.e., applying a ranking procedure, see Section 5.5.1).

The second phase involves constructing the diversified archive P_{t+1}, which contains the N best individuals from the population R_t. To achieve this, individuals from successive fronts of R_t are inserted into P_{t+1} as long as the number of inserted elements remains below the limited size N. When it is not possible to insert all individuals from the same front into the archive, selection is based on truncation using the crowding distance (described below).

The final phase involves creating a new offspring population Q_{t+1} by applying selection, crossover, and mutation operators on the individuals from P_{t+1}. Algorithm 5.2 summarizes these steps.

5.6.2 Niche formation: Calculation of the Crowding distance

The NSGA-II algorithm is enhanced by a method for distributing the adaptation among individuals in the same front, called the crowding distance. Algorithm 5.3 presents the pseudo-code of the algorithm for assigning this distance. In this algorithm, $\mathcal{I}(i).k$ denotes the value of the k^{th} objective function of individual i belonging

Algorithm 5.2: Pseudo-code of NSGA-II algorithm

```
Initialize the archive P_0 of size N
Create population Q_0 of size N from P_0 by applying selection and variation operators
Initialize time counter t ← 0
while (The termination condition is not satisfied) do
    Create R_t = P_t ∪ Q_t
    Calculate fronts F = {F_1, F_2, ...} of R_t using a ranking algorithm (see Algorithm 5.1)
    Empty P_{t+1} and initialize i = 0
    while |P_{t+1}| + |F_i| ≤ N do
        P_{t+1} = P_{t+1} ∪ F_i
        i = i + 1
    end
    Calculate crowding distance for each individual in F_i (see Algorithm 5.3)
    Sort individuals of front F_i according to crowding distance in descending order
    Choose the first N − |P_{t+1}| elements of F_i: P_{t+1} = P_{t+1} ∪ F_i[1 : N − |P_{t+1}|]
    Select parents from P_{t+1} and create a new population Q_{t+1} by applying crossover and mutation operators
    Increment the time counter t ← t + 1
end
```

to front $\mathcal{I}$. The complexity of this procedure depends on the sorting algorithm. In the worst case (all individuals in the population form a single front), the sorting operates in $O(mN \log N)$ steps.

The crowding distance associated with point i is equal to the sum over all objective functions of the gap between the two points that bound this point. This measure first favors the extreme points, then the more isolated ones. Figure 5.8 illustrates, for a bi-objective MOP, the calculation associated with point i.

The crowding distance is not only involved in the selection of archive members, but also serves in selecting parent individuals. It is used as a criterion when the competing individuals are part of the same front. Based on the fitness assignment principle of SPEA2 (see Section 5.7), we can formulate the fitness of the NSGA-II algorithm as follows:

$$\text{Fitness}_{\text{NSGA-II}}(i) = \mathcal{I}(i) + \frac{1.0}{\mathcal{I}_{\text{distance}}(i) + 2.0} \tag{5.10}$$

Algorithm 5.3: Pseudo-code of the crowding distance assignment algorithm

```
1  foreach individual i belonging to front I do
2  |  I_distance(i) = 0
3  |  l = |I|
4  |  foreach objective function k = 1, ..., m do
5  |  |  Sort the l individuals of front I according to the value of objective
   |  |  function k in ascending order
6  |  |  Set I_distance(1) = I_distance(l) = ∞ (so that the extreme points of the front
   |  |  are always selected)
7  |  |  for i ← 2 to (l − 1) do
8  |  |  |  I_distance(i) = I_distance(i) + (I(i + 1).k − I(i − 1).k)
9  |  |  end
10 |  end
11 end
```

where $\mathcal{I}(i)$ corresponds to the front of individual i and $\mathcal{I}_{\text{distance}}(i)$ is its crowding distance. In the denominator, the value 2.0 is added to ensure that its values are greater than 0 and that the values of the second term of the sum are less than 1. Thus, the fitness values of individuals from the k-th front are in the interval $]k, k + 0.5]$.

5.7 SPEA2 (Strength Pareto Evolutionary Algorithm 2)

Zitzler et al. also made modifications to their SPEA algorithm (Zitzler and Thiele, 1999) to improve its performance (Zitzler et al., 2001).

5.7.1 General operating principle

The first step is to create an initial population $\mathbf{P}_0$ of size N and initialize the external archive $\overline{\mathbf{P}}_0$ of size $\overline{N}$ to the empty set. The SPEA2 algorithm will regularly update this archive based on new non-dominated individuals.

At each generation, the archive is diversified and filled by keeping the $\overline{N}$ non-dominated individuals from both the archive and the previous population. When it is not possible to insert all non-dominated individuals, selection is based on truncation

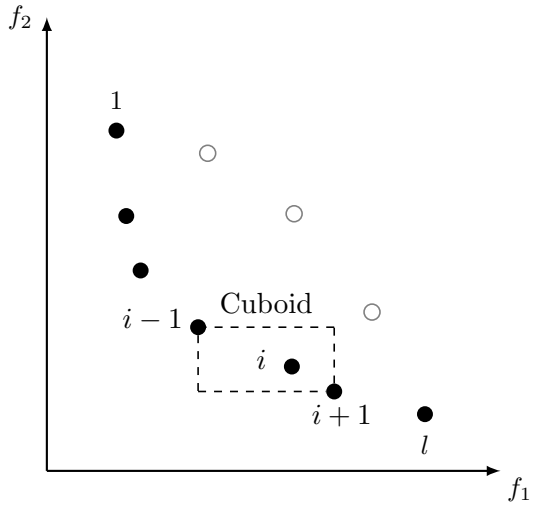

Figure 5.8: Illustration of crowding distance calculation (adapted from (Deb et al., 2002)). The points marked by black circles correspond to solutions from the same front.

using a dominance relation that takes into account the distances (in objective space) between individuals. However, if the number of non-dominated individuals is less than $\overline{N}$, the archive is supplemented with the least dominated individuals from both the archive and the population, relative to the fitness described below.

The last phase of a generation involves creating the population $\mathbf{P}_{t+1}$. This simply requires applying the crossover and mutation operators to individuals selected from $\overline{\mathbf{P}}_{t+1}$ and inserting the offspring into $\mathbf{P}_{t+1}$.

Figure 5.9 and Algorithm 5.4 illustrate the overall operating scheme of SPEA2.

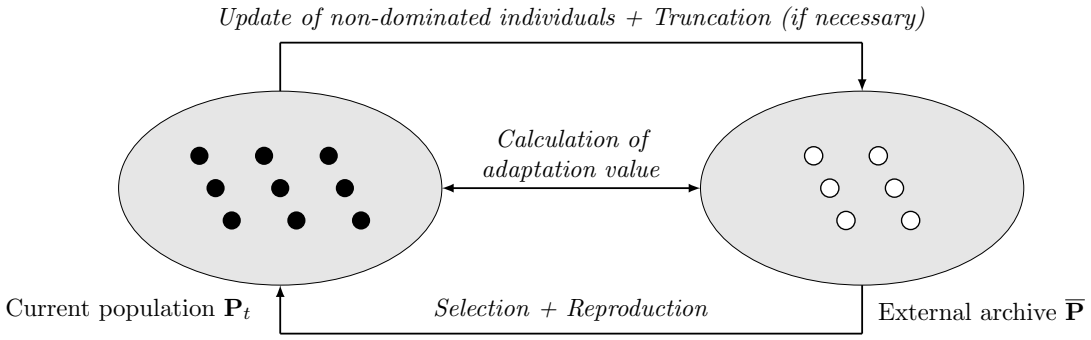

Figure 5.9: Illustration of the overall operation of the SPEA2 algorithm.

Algorithm 5.4: Pseudo-code of SPEA2 algorithm

– Generate the initial population $\mathbf{P}_0$ of size N
– Create the empty external archive $\overline{\mathbf{P}}_0$ of size $\overline{N}$
while *(The termination condition is not satisfied)* **do**
  – Calculate the fitness value for all individuals in $\mathbf{P}_t \cup \overline{\mathbf{P}}_t$ (see paragraph 5.7.2)
  – Insert all non-dominated individuals from $\mathbf{P}_t \cup \overline{\mathbf{P}}_t$ into $\overline{\mathbf{P}}_{t+1}$
    • If the size of $\overline{\mathbf{P}}_{t+1}$ exceeds $\overline{N}$, reduce $\overline{\mathbf{P}}_{t+1}$ using the truncation operator (see paragraph 5.7.3)
    • Otherwise, if the size of $\overline{\mathbf{P}}_{t+1}$ is less than $\overline{N}$, fill the archive $\overline{\mathbf{P}}_{t+1}$ by inserting the $\overline{N} - \left|\overline{\mathbf{P}}_{t+1}\right|$ best dominated individuals from $\mathbf{P}_t \cup \overline{\mathbf{P}}_t$
  – Select parent individuals from $\overline{\mathbf{P}}_{t+1}$ and create the population $\mathbf{P}_{t+1}$ by applying crossover and mutation operators
  – Increment the time counter $t \leftarrow t + 1$
end

5.7.2 Assignment of fitness value

The calculation of the fitness value in SPEA2 is carried out in three steps. Initially, each individual i from the set $\mathbf{P} \cup \overline{\mathbf{P}}$ receives a strength $S(i)$ equal to the total number of individuals it dominates:

$$S(i) = \left|\left\{j | j \in \mathbf{P}_t \cup \overline{\mathbf{P}}_t \wedge i \prec j\right\}\right| \tag{5.11}$$

where $|.|$ represents the cardinality of a set and the symbol $\prec$ denotes the Pareto dominance relation. Next, each individual is assigned a raw fitness score $R(i)$ equal to the sum of strengths of individuals it dominates among the archive and the population:

$$R(i) = \sum_{j \in \mathbf{P}_t \cup \overline{\mathbf{P}}_t \wedge j \prec i} S(j) \tag{5.12}$$

Figure 5.10 illustrates, with a 2-dimensional example, the process of assigning scores.

Finally, to distinguish individuals with the same score, additional information $D(i)$ estimating the density of solutions around an individual is added to the score to form

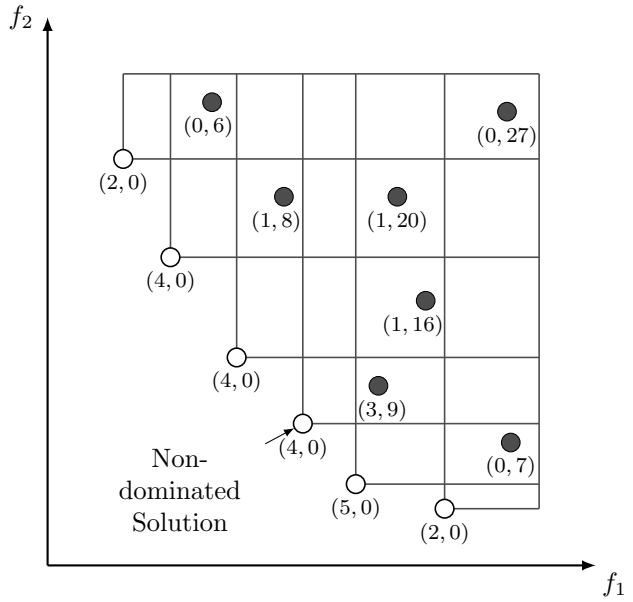

Figure 5.10: Score assignment procedure (*strength*, *raw fitness*) of SPEA2.

the fitness:

$$Fitness_{SPEA2}(i) = R(i) + D(i) \tag{5.13}$$

where the function $D(i)$ is defined as follows:

$$D(i) = \frac{1}{\sigma_i^k + 2.0} \tag{5.14}$$

where σ_i^k corresponds to the distance (in the objective space) between individual i and its k^{th} nearest neighbor. More precisely, for each individual i, the distances to all individuals j in the set $\mathbf{P} \cup \overline{\mathbf{P}}$ are calculated and stored in a list. After sorting this list in ascending order, the k^{th} element yields the desired distance. Typically, $k = \sqrt{N + \overline{N}}$.

5.7.3 Archive truncation procedure

When the current non-dominated set exceeds the limited size of the archive ($\overline{N}$), a truncation procedure is invoked to remove $\left|\overline{\mathbf{P}}t + 1\right| - \overline{N}$ individuals. At each iteration, the individual i for which $i \leq_d j$ for all individuals j in $\overline{P}t + 1$ is removed, with:

$$i \leq_d j \qquad \forall\, 0 < k < \left|\overline{\mathbf{P}}_{t+1}\right| : \sigma_i^k = \sigma_j^k \;\; \vee \tag{5.15}$$

$$\exists\, 0 < k < \left|\overline{\mathbf{P}}_{t+1}\right| : \left[\left(\forall\, 0 < l < k : \sigma_i^l = \sigma_j^l\right) \;\wedge\; \sigma_i^k < \sigma_j^k\right] \tag{5.16}$$

where σ_i^k corresponds to the distance between individual i and its k^{th} nearest neighbor in $\overline{\mathbf{P}}_{t+1}$. In other words, the individual with the minimum distance to another individual is chosen at each step; if there are multiple individuals with the same minimum distance, the choice is made based on the second distance and so on. Figure 5.11 illustrates the archive truncation technique used in the SPEA2 algorithm. Finally, it is noteworthy that this truncation procedure has an average complexity of $O(M^2 \log M)$, where $M = N + \overline{N}$ (Zitzler et al., 2001).

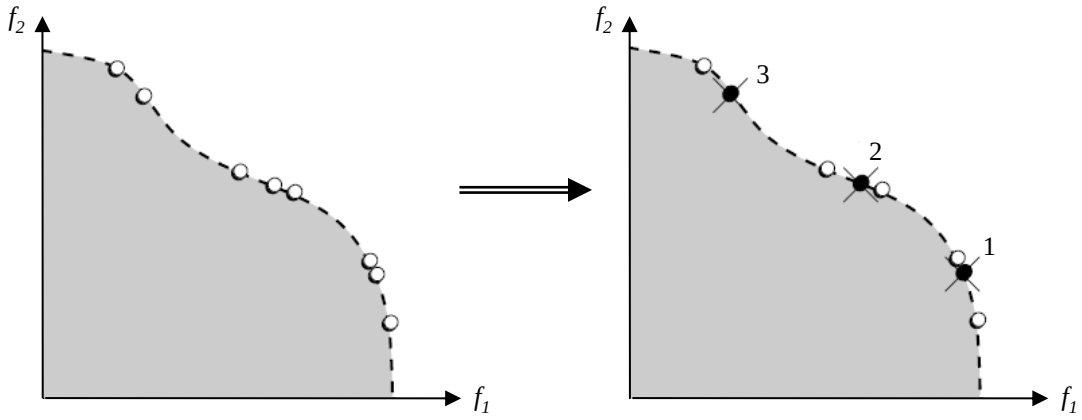

Figure 5.11: Illustration of the archive truncation procedure used in the SPEA2 algorithm (adapted from (Zitzler et al., 2001)). The left part of the figure shows the Pareto front. The right part illustrates the order in which the truncation operator eliminates solutions. Note that the front is defined here in terms of maximization and that the size of the archive is $\overline{N} = 5$.

Bibliography

Vincent Barichard and Jin-Kao Hao. Genetic tabu search for the multi-objective knapsack problem. *Tsinghua Science and Technology*, 8(1):8–13, 2003.

Roberto Battiti and Giampietro Tecchiolli. The reactive tabu search. *ORSA journal on computing*, 6(2):126–140, 1994.

James C Bean. Genetic algorithms and random keys for sequencing and optimization. *ORSA journal on computing*, 6(2):154–160, 1994.

Christian Blum and Paola Festa. *Metaheuristics for String Problems in Bio-informatics*. John Wiley & Sons, 2016.

Christian Blum and Andrea Roli. Metaheuristics in combinatorial optimization: Overview and conceptual comparison. *ACM Computing Surveys (CSUR)*, 35(3): 268–308, 2003.

Ilhem Boussaïd, Julien Lepagnot, and Patrick Siarry. A survey on optimization metaheuristics. *Information Sciences*, 237:82–117, 2013.

Bernd Bullnheimer, Richard F Hartl, and Christine Strauss. A new rank based version of the Ant System. A computational study. *Cent. Eur. J. Oper. Res. Econ.*, 7(1): 25–38, 1999.

Laura Calvet, Sergio Benito, Angel A Juan, and Ferran Prados. On the role of metaheuristic optimization in bioinformatics. *International Transactions in Operational Research*, 30(6):2909–2944, 2023.

Vladimír Černỳ. Thermodynamical approach to the traveling salesman problem: An efficient simulation algorithm. *Journal of optimization theory and applications*, 45 (1):41–51, 1985.

Maurice Clerc. The swarm and the queen: towards a deterministic and adaptive particle swarm optimization. In *Evolutionary Computation, 1999. CEC 99. Proceedings of the 1999 Congress on*, volume 3, pages 1951–1957. IEEE, 1999.

Maurice Clerc and James Kennedy. The particle swarm-explosion, stability, and convergence in a multidimensional complex space. *IEEE transactions on Evolutionary Computation*, 6(1):58–73, 2002.

Stephen A Cook. The complexity of theorem-proving procedures. In *Proceedings of the third annual ACM symposium on Theory of computing*, pages 151–158. ACM, 1971.

Oscar Cordon, Iï£¡aki Fernï£¡ndez de Viana, Francisco Herrera, and Llanos Moreno. A New ACO Model Integrating Evolutionary Computation Concepts: The Best-Worst Ant System. In *Dorigo, M., Middendorf, M., Sï£¡tzle, T. (eds.) Abstract proceedings of ANTS 2000 - From Ant Colonies to Artificial Ants: Second International Workshop on Ant Algorithms*, pages 22–29. IRIDIA„ Universitï£¡ Libre de Bruxelles, Brussels, Belgium, 2000.

Kalyanmoy Deb, Amrit Pratap, Sameer Agarwal, and TAMT Meyarivan. A fast and elitist multiobjective genetic algorithm: Nsga-ii. *IEEE transactions on evolutionary computation*, 6(2):182–197, 2002.

Marco Dorigo and Luca Maria Gambardella. Ant colony system: a cooperative learning approach to the traveling salesman problem. *IEEE Transactions on evolutionary computation*, 1(1):53–66, 1997.

Marco Dorigo and Thomas Stützle. Ant colony optimization: overview and recent advances. In *Handbook of metaheuristics*, pages 227–263. Springer, 2010.

Marco Dorigo, Vittorio Maniezzo, and Alberto Colorni. Ant system: optimization by a colony of cooperating agents. *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, 26(1):29–41, 1996.

Russell Eberhart and James Kennedy. A new optimizer using particle swarm theory. In *Micro Machine and Human Science, 1995. MHS'95., Proceedings of the Sixth International Symposium on*, pages 39–43. IEEE, 1995.

Matthias Ehrgott and Xavier Gandibleux. Hybrid metaheuristics for multi-objective combinatorial optimization. In *Hybrid metaheuristics*, pages 221–259. Springer, 2008.

Thomas A Feo and Mauricio GC Resende. A probabilistic heuristic for a computationally difficult set covering problem. *Operations research letters*, 8(2):67–71, 1989.

Thomas A Feo and Mauricio GC Resende. Greedy randomized adaptive search procedures. *Journal of global optimization*, 6(2):109–133, 1995.

Michel Gendreau and Jean-Yves Potvin. *Handbook of metaheuristics*, volume 2. Springer, 2010.

Fred Glover. Future paths for integer programming and links to artificial intelligence. *Computers & operations research*, 13(5):533–549, 1986.

David E Goldberg. Genetic algorithms in search, optimization and machine learning 'addison-wesley, 1989. *Reading, MA*, 1989.

David E Goldberg, Jon Richardson, et al. Genetic algorithms with sharing for multimodal function optimization. In *Genetic algorithms and their applications: Proceedings of the Second International Conference on Genetic Algorithms*, pages 41–49. Hillsdale, NJ: Lawrence Erlbaum, 1987.

Pierre Hansen, Nenad Mladenović, Jack Brimberg, and José A. Moreno Pérez. Variable Neighborhood Search. In *Handbook of Metaheuristics*, pages 61–86. Springer US, 2010.

Frederick S Hillier and Gerald J Lieberman. *Introduction to operations research*. McGraw-Hill, 2015.

John H Holland. Adaptation in natural and artificial systems. An introductory analysis with application to biology, control, and artificial intelligence. *Ann Arbor, MI: University of Michigan Press*, 1975.

Xiaohui Hu, Russell C Eberhart, and Yuhui Shi. Swarm intelligence for permutation optimization: a case study of n-queens problem. In *Swarm intelligence symposium, 2003. SIS'03. Proceedings of the 2003 IEEE*, pages 243–246. IEEE, 2003.

J. Kennedy and R.C. Eberhart. Particle Swarm Optimization. In *Proc. IEEE Int. Conf. on N.N*, pages 1942–1948, 1995.

J. Kennedy and R.C. Eberhart. A discrete binary version of the particle swarm algorithm. In *Int. IEEE Conf. on Systems, Man, and Cyber*, volume 5, pages 4104–4108, 1997.

Scott Kirkpatrick, C Daniel Gelatt, Mario P Vecchi, et al. Optimization by simulated annealing. *science*, 220(4598):671–680, 1983.

Jing J Liang, A Kai Qin, Ponnuthurai N Suganthan, and S Baskar. Comprehensive learning particle swarm optimizer for global optimization of multimodal functions. *IEEE transactions on evolutionary computation*, 10(3):281–295, 2006.

M López-Ibánez, T Stützle, and M Dorigo. Ant colony optimization: A component-wise overview. *Techreport, IRIDIA, Universite Libre de Bruxelles*, 2015.

Helena R Lourenço, Olivier C Martin, and Thomas Stützle. Iterated local search: Framework and applications. In *Handbook of metaheuristics*, pages 363–397. Springer, 2010.

Nicholas Metropolis, Arianna W Rosenbluth, Marshall N Rosenbluth, Augusta H Teller, and Edward Teller. Equation of state calculations by fast computing machines. *The journal of chemical physics*, 21(6):1087–1092, 1953.

Herve Meunier, E-G Talbi, and Philippe Reininger. A multiobjective genetic algorithm for radio network optimization. In *Evolutionary Computation, 2000. Proceedings of the 2000 Congress on*, volume 1, pages 317–324. IEEE, 2000.

Kaisa Miettinen. *Nonlinear Multiobjective Optimization*. Kluwer, 1999.

Nenad Mladenovic. A variable neighborhood algorithm-a new metaheuristic for combinatorial optimization. In *papers presented at Optimization Days*, volume 12, 1995.

Nenad Mladenović and Pierre Hansen. Variable neighborhood search. *Computers & operations research*, 24(11):1097–1100, 1997.

Alexander G Nikolaev and Sheldon H Jacobson. Simulated annealing. In *Handbook of metaheuristics*, pages 1–39. Springer, 2010.

Jorge Nocedal and Stephen J Wright. *Numerical optimization*. Springer, 1999.

Riccardo Poli, James Kennedy, and Tim Blackwell. Particle swarm optimization. *Swarm intelligence*, 1(1):33–57, 2007.

Jérémi Regnier. *Conception de systèmes hétérogènes en Génie Electrique par optimisation évolutionnaire multicritère*. PhD thesis, Institut National Polytechnique de Toulouse, 2003.

Mauricio GC Resende and Celso C Ribeiro. Greedy randomized adaptive search procedures: Advances, hybridizations, and applications. In *Handbook of metaheuristics*, pages 283–319. Springer, 2010.

Yuhui Shi and Russell Eberhart. A modified particle swarm optimizer. In *Evolutionary Computation Proceedings, 1998. IEEE World Congress on Computational Intelligence., The 1998 IEEE International Conference on*, pages 69–73. IEEE, 1998.

Christine Solnon. *Contributions à la résolution pratique de problèmes combinatoires –des fourmis et des graphes–*. PhD thesis, Université Lyon 1, 2005.

Nidamarthi Srinivas and Kalyanmoy Deb. Muiltiobjective optimization using non-dominated sorting in genetic algorithms. *Evolutionary computation*, 2(3):221–248, 1994.

Thomas Stützle and Holger H Hoos. MAX–MIN ant system. *Future generation computer systems*, 16(8):889–914, 2000.

El-Ghazali Talbi. *Metaheuristics: from design to implementation*, volume 74. John Wiley & Sons, 2009.

El-Ghazali Talbi. Hybrid metaheuristics for multi-objective optimization. *Journal of Algorithms & Computational Technology*, 9(1):41–63, 2015.

M Fatih Tasgetiren, Mehmet Sevkli, Yun-Chia Liang, and Gunes Gencyilmaz. Particle swarm optimization algorithm for permutation flowshop sequencing problem. In *International Workshop on Ant Colony Optimization and Swarm Intelligence*, pages 382–389. Springer, 2004a.

Mehmet Fatih Tasgetiren, Mehmet Sevkli, Yun-Chia Liang, and Gunes Gencyilmaz. Particle swarm optimization algorithm for single machine total weighted tardiness problem. In *Evolutionary Computation, 2004. CEC2004. Congress on*, volume 2, pages 1412–1419. IEEE, 2004b.

Alan Mathison Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London mathematical society*, 2(1):230–265, 1937.

Eckart Zitzler and Lothar Thiele. Multiobjective evolutionary algorithms: A comparative case study and the strength Pareto approach. *IEEE transactions on Evolutionary Computation*, 3(4):257–271, 1999.

Eckart Zitzler, Kalyanmoy Deb, and Lothar Thiele. Comparison of multiobjective evolutionary algorithms: Empirical results. *Evolutionary computation*, 8(2):173–195, 2000.

Eckart Zitzler, Marco Laumanns, Lothar Thiele, et al. Spea2: Improving the strength pareto evolutionary algorithm. Technical Report 103, Computer Engineering and

Networks Laboratory (TIK), Swiss Federal Institute of Technology (ETH) Zurich, Gloriastrasse 35, CH-8092 Zurich, Switzerland, 2001.

Eckart Zitzler, Marco Laumanns, and Stefan Bleuler. A tutorial on evolutionary multiobjective optimization. *Metaheuristics for multiobjective optimisation*, pages 3–37, 2004.