CHAPTER

Foundations of combinatorial optimization, heuristics, and metaheuristics

22

Bochra Rabbouch[a], Hana Rabbouch[b], Foued Saâdaoui[c], and Rafaa Mraihi[d]

[a]*Higher Institute of Applied Sciences and Technology of Sousse, University of Sousse, Sousse, Tunisia,* [b]*Higher Institute of Management of Tunis, University of Tunis, Tunis, Tunisia,* [c]*Department of Statistics, Faculty of Sciences, King Abdulaziz University, Jeddah, Saudi Arabia,* [d]*Ecole Supérieure de Commerce de Tunis, Campus Universitaire de Manouba, Manouba, Tunisia*

1. Introduction

Combinatorial optimization (CO) problems are an important class of problems where the number of possible solutions grows combinatorially with the problem size. These kinds of problems have attracted the attention of researchers in computer sciences, operational research, and artificial intelligence. The fundamental objective of CO is to find the optimal or near-optimal solution for a complex problem using different optimization techniques to minimize costs and maximize profits and performances.

This chapter provides background on different materials, notations, and algorithms in CO. The chapter is organized as follows: Sections 3 and 4 discuss the analysis and complexity of algorithm theories. Section 5 discusses modeling CO problems to simplify realistic complications and reduce problem difficulties. Modeling consists of illustrating the graph theory concepts, the mathematical programming, and the constraint programming paradigms. Finally, Section 6 presents solution methods, including exact methods, heuristics, and metaheuristics.

2. Combinatorial optimization problems

A CO problem is an optimization problem where the number of possible solutions is finite and grows combinatorially with the problem size. It aims to look for the perfect solution from a very huge solution space and allows an excellent usage of limited resources in order to attain a fundamental objective within a running time bounded by a polynomial in the input size. The quality of optimization relies on how quickly it is possible to find the optimal solution. The CO problem has emerged in industrial, production, logistic environments, and computer systems.

A CO problem can be modeled by a set of variables to find a satisfying solution respecting a set of constraints while optimizing an objective function. The problem P can be written as:

Comprehensive Metaheuristics. https://doi.org/10.1016/B978-0-323-91781-0.00022-3

$$P = \langle X, D, C, f \rangle,$$

where

- $X = \{x_1, \ldots, x_n\}$ is the set of variables;
- $D = \{D(x_1), \ldots, D(x_n)\}$ is the set of domains of variables, $D(x_n)$ is the domain of x_n;
- $C = C_1, \ldots, C_n$ is the set of constraints over variables; and
- f is the objective function to be optimized.

It is possible to write the objective function to optimize as:

$$\text{optimize } \{f(F) : F \in \mathcal{F}\},$$

where *optimize* is replaced by either *minimize* or *maximize* and including the following settings:

- A finite set, $E = \{e_1, \ldots, e_n\}$.
- A weight function, $w : E \to \mathbb{Z}$, $w(e_i)$ is the weight of e_i.
- A finite family, $\mathcal{F} = \{F_1, \ldots, F_m\}$, $F_i \subseteq E$ are the feasible solutions.
- A cost function, $f : \mathcal{F} \to \mathbb{Z}, f(F) = \sum_{e \in F} w(e)$ (additive cost function).

For the most part, similar problems are able to be defined as integer programs with binary variables to verify if every member of the collection is a part of the subset or not. Furthermore, because optimization problems are of minimization or maximization type, so for maximization problems for example, note that if the function to maximize is well defined, then minimizing the negation of this will maximize the original function.

CO is the most popular type of optimization and it is often linked with graph theory and routing and it includes the vehicle routing problem [1, 2], the traveling salesman problem [3], the knapsack problem [4], the bin packing and cutting stock problem [5], and the bus scheduling problem [6]. Some well-known combinatorial games and puzzles include Chess [7], Sudoku [8], and Go [9].

3. Analysis of algorithms

An algorithm consists of an ordering set of constructions for solving a problem. In computer science, the analysis of algorithms is the determination of the whole quantity of resources (such as time and storage) necessary to execute them. This helps scientists to compare algorithms in terms of their efficiency, speed, and resource consumption by specifying an estimate number of operations without regard to the specific implementation or input used. Then, it presents a regular measure of algorithm complexity regardless of the platform or the problem cases solved by the algorithm.

Generally, when analyzing algorithms, the fact or most used to measure performance is the time spent by an algorithm to solve the problem. Time is expressed in terms of number of elementary operations such as comparisons or branching instructions.

A complete analysis of the running time of an algorithm requires the following steps:

- implement the whole algorithm completely;
- determine the time needed for each basic operation;
- identify unknown quantities that can be used to describe the frequency of execution of the basic operations;

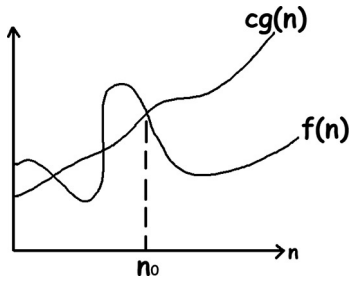

FIG. 1

$g(n)$ is an asymptotic upper bound for $f(n)$.

- establish a realistic model for the input to the program;
- analyze the unknown quantities, supposing the modeled input; and
- calculate the total running time by multiplying the time by the frequency for each operation, then adding all the products.

In a related context, the asymptotic efficiency of an algorithm reveals that the running time of an algorithm increases when the size of an input tends to infinity.

To specify an asymptotic upper bound [10] in algorithm analysis, we generally use the O notation and we say that such algorithm is of an Order n ($O(n)$) where n is the size of the problem (the total number of operations executed by the algorithm is at most a constant time n). The O notation is used to describe the complexity of an algorithm in a worst-case scenario. So, let f and g be functions from $\mathbb{N}$ to $\mathbb{N}$, $f(n)$ is of $O(g(n))$ if there exist positive constants c and n_0 and for all $n \geq n_0$, $0 \leq f(n) \leq cg(n)$ and that means informally that f grows as g or slower (Fig. 1).

If the opposite happens, that is, if $g(n) = O(f(n))$, we specify an asymptotic lower bound [10]; we generally use the Ω notation. More specifically, $f(n)$ is of $\Omega(g(n))$ if there exist positive constants c and n_0 such that $0 \leq cg(n) \leq f(n)$, for all $n \geq n_0$ (Fig. 2).

Finally, to specify an asymptotically tight bound [10], we use the θ notation and we say that $f(n)$ is of $\theta g(n)$ if there exist positive constants c_1, c_2, and n_0 and for all $n \geq n_0$, $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$. The latter means that f and g have precisely the same rate of growth (Fig. 3).

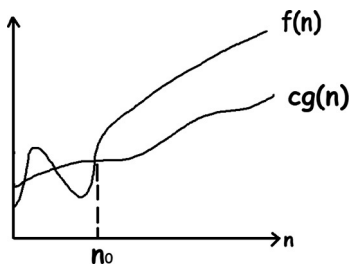

FIG. 2

$g(n)$ is an asymptotic lower bound for $f(n)$.

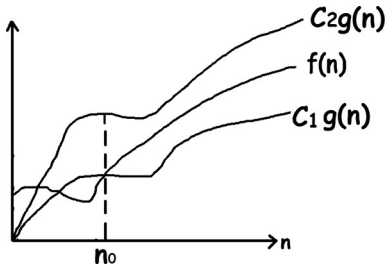

FIG. 3

$g(n)$ is an asymptotically tight bound for $f(n)$.

4. Complexity of algorithms

Computational complexity theory is a field in theoretical computer science and applied mathematics, used to analyze the efficiency of algorithms and help solve a problem with handling with resources required. Most complexity problems ask a question and expect an answer. The specific kind of question asked characterizes the problem, which requires either a "yes" or a "no" answer. Such problems are called decision problems. In complexity theory, we usually find it convenient to consider only decision problems, instead of problems requiring all sorts of different answers. For example, any maximization or minimization problem where the aim is to find a solution with the maximum possible profit or the minimum possible cost can be automatically transformed to a decision problem formulated as: "is there a real number X that represents the solution to the problem?" Then, the complexity theory can be applied to optimization problems.

An important issue that comes up when considering CO problems is the classification of problems. In this context, those remarkable classes of problems are generally described as presented in the sections that follow.

- **Class *P*: Polynomial time problems:** This class of problem consists of all decision problems for which exists a polynomial time deterministic algorithm to solve them efficiently. This class of decision problems includes tractable problems that can be solved sufficiently by a deterministic Turing machine in a polynomial time.
- **Class *NP*: Nondeterministic polynomial time problems:** This class of problems consists of all decision problems that can be solved by polynomial time nondeterministic algorithms. We can obtain the solution algorithm by guessing a solution and check whether the guessed solution is correct or not. Furthermore, Class NP contains the problems in which "verifying" the solution of the problem is quick, but finding the solution for the problem is difficult. Problems of this class can be solved by a nondeterministic Turing machine in a polynomial time.
- **Class *NP Hard*: Nondeterministic polynomial time hard problems:** The *NP*-hard problems are at least as hard as the hardest problems in *NP* and each problem in *NP* can be solved by reducing it to different polynomial problems.
- **Class *NP Complete*: Nondeterministic polynomial time complete problems:** Those problems are the most difficult problems in their class. A problem is classified as *NP* complete if it satisfies two

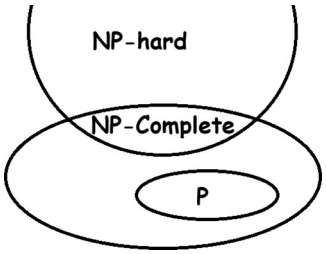

FIG. 4

Complexity classes.

conditions: it should be in the set of *NP* class and at the same time it should be *NP hard*. This class contains the hardest problems in computer science.

Fig. 4 illustrates how the different complexity classes are related to each other.

5. Modeling a CO problem

Solving an optimization problem
$\text{Problem} \Rightarrow \text{Model} \Rightarrow \text{solution(s)}$

Modeling an optimization problem helps to simplify the reality complications and reduces the difficulties of the problem. The design of a "good" model projects expert knowledge of the domain, variables, and constraints, as well as computer-based resolution methods. In addition, models can capture and organize the understanding of the system, permit early exploration of alternatives, increase the decomposition and modularization of the system, facilitate the system's evolution and maintenance, and finally, facilitate the reuse of parts of the system in new projects.

5.1 Graph theory concepts

A graph is a data structure that can be used to model relations, processes, and paths in practical systems and realistic problems. Graph theory was developed in 1735 when Leonhard Euler solved negatively (proved that the problem has no solution) the historically notable problem in mathematics called the Seven Bridges of Königsberg.

A graph can be defined as an ordered pair $G = (V, A)$ that links a set of V vertices by a set of A arcs or edges (each edge is generally associated with two vertices) where V and A are usually finite. If there is more than one edge between two vertices, the graph is a multigraph. If edges have no orientation, the graph is undirected and there is no difference between an arc (x, y) and an arc (y, x), but when edges have orientations, the graph is directed. In addition, a mixed graph is a graph in which some edges have orientations and others have no orientations, and a weighted graph is a graph in which there is a number or a weight associated to each edge (cost, distance, etc.).

A walk is defined as a sequence of adjacent nodes (neighbors) and is denoted by $w = [v_1, v_2, \ldots, v_n]$. The walk is closed (circuit) if $v_1 = v_k (\forall k > 1)$. If there are no repeated nodes in a walk, it is called

path. A walk $w = [v_1, v_2, \ldots, v_k]$ with no repeated nodes except $v_1 = v_k$ is a cycle. A Hamiltonian cycle is a cycle that includes every vertex exactly once. An Eulerian walk is a walk in which each edge appears exactly once, and an Eulerian circuit is a closed walk in which each edge appears exactly once.

A graph is called complete if there is an edge between every pair of vertices and is called connected if there is a path between every pair of vertices. A tree is a connected graph with no cycles.

Many practical problems can be modeled by graphs especially in telecommunications, mathematics, computer sciences, data transmission, neural networks, transportation, logistics, and even in chemistry, biology, and linguistics.

5.2 Mathematical optimization model

A mathematical optimization model is an abstraction that allows to simplify and summarize a system. Once a mathematical model is made, a list of inputs, outputs, decision variables, and constraints is prepared and hence the problem is decorticated and the system behavior can be predicted.

To model an optimization problem, mathematical programming formulations represent the main goal as objective function, which is the output (a single score) to maximize or minimize. In addition, mathematical programs represent problem choices as decision variables that influence the objective function with consideration to the constraints on variable values expressing the limits on how big or small possible decision variables can get. Decision variables may be discrete or continuous:

- A variable is discrete if its set of values is fixed above or is countable (only have integer values, for example).
- A variable is continuous if it may take any variable in a specified interval.

Different families of optimization models are used in practice to formulate and solve decision-making problems. In general, the most successful models are based on mathematical programming and constraint programming. To classify mathematical programming formulations, five categories can be founded:

- Linear programming (LP): A class of optimization problems where objective function and constraints are expressed by linear functions.
- Integer programming (IP): Linear problems that restrict the variables to be integers.
- Mixed-integer linear programming (MILP): Linear problems in which decision variables are both discrete and continuous.
- Nonlinear programming (NLP): Objective function and constraints are expressed by nonlinear functions.
- Mixed-integer nonlinear programming: Nonlinear problems in which some decision variables are integers.

Mathematical programs and their terminology are due to George B. Dantzig, the inventor of the simplex algorithm for solving the LP problems in 1947 [11]. Since then, LP has been widely used and has attained a strong position in practical optimization. An interesting survey paper providing references to papers and reports whose purpose is to give overviews of linear, integer, and mixed-integer optimization, including solution techniques, solvers, languages, applications, and textbooks, is presented by Newman and Weiss [12].

5.2.1 Linear programming

An LP model is an optimization method to find the optimal solution to some special case mathematical optimization model where the objective and the constraints are required to be linear. The LP also contains the decision variables, which are the unknown quantities or decisions that are to be optimized.

An LP model can be considered as:

$$\text{Minimize } F_{LP} = c_1x_1 + c_2x_2 + \cdots + c_jx_j + \cdots + c_nx_n. \tag{1}$$

Subject to:

$$a_{11}x_1 + a_{12}x_2 + \cdots + a_{1j}x_j + \cdots + a_{1n}x_n (\leq, =, \text{or} \geq) b_1, \tag{2}$$

$$a_{i1}x_1 + a_{i2}x_2 + \cdots + a_{ij}x_j + \cdots + a_{in}x_n (\leq, =, \text{or} \geq) b_i, \tag{3}$$

$$a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mj}x_j + \cdots + a_{mn}x_n (\leq, =, \text{or} \geq) b_m, \tag{4}$$

$$x_1, \ldots, x_j, \ldots, x_n \geq 0, \tag{5}$$

where

- $x_1, \ldots, x_j, \ldots, x_n$ are the decision variables that are constrained to be nonnegative.
- $c_1, \ldots, c_j, \ldots, c_n$ are the coefficients in the objective function for the optimization problem.
- $b_1, \ldots, b_i, \ldots, b_m$ are the right-hand side values in the constraints sets.
- $a_{11}, \ldots, a_{ij}, \ldots, a_{mn}$ are the variables coefficients in the constraints set.

Eq. (1) presents the objective function that assesses the quality of the solution and which aims to minimize (or maximize) the value of F_{LP}.

The inequalities (2, 3, 4, 5) are the constraints that a valid solution (called a feasible solution) to the problem must satisfy. Each constraint requires that a linear function of the decision variables is either equal to, not less than, or not more than a scalar value. Besides, a common condition simply states that each decision variable must be nonnegative.

A solution that satisfies all constraints is called a feasible solution. The feasible solution with the best objective value (either the lowest in the case of minimization problem or the highest in the case of maximization problem) is called the optimal solution.

The development of LP is considered one of the most important scientific advances of the mid-20th century. A very efficient solution procedure for solving linear programs is the simplex algorithm. The simplex method proceeds by enumerating feasible solutions and improving the value of the objective function at each step. The termination step is reached after a finite number of different transitions. A strength of this method is its robustness: it allows solving any linear program even for problems with one or more optimal solutions (can provide all optimal solutions, if they exist), finding redundant constraints in the formulation, discovering instances when the objective value is unbounded over the feasible region, and generating feasible solutions to start the procedure or to prove that the problem has no feasible solution. For a deep explanation of LP and the simplex method, the reader is referred to the textbook [13].

5.2.2 Integer linear programming

Integer linear programs are extension of linear programs where decision variables are restricted to be integer values. They generally concern model problems counting some values (e.g., commodities) and enforce representing them by integer decision variables. A special case of integer linear programs is

when modeling problems with yes/no decisions then the decision variables are initiated as binary values and are restricted to 0/1 values only.

5.2.3 Mixed-integer linear programming

Mixed-integer optimization problems appear naturally in many contexts of real-world problems and overcome several new technical challenges that do not appear in the nonmixed counterparts. In addition, MILPs can optimize decisions that take place in complex systems by including integer, binary, and real variables to form logical statements.

Mixed-integer programs are linear programs composed by linear inequalities (constraints) and linear objective function to optimize with the added restriction that some, but not necessarily all, of the variables must be integer-valued. The term "integer" can be replaced by the term "binary" if the integer decision variables are restricted to take on either 0 or 1 values. A standard mixed-integer linear problem is of the form:

$$z^* = Min\ c^T x,\quad x \in X. \tag{6}$$

Subject to:

$$A\,x(\leq, =, \text{or} \geq) b, \tag{7}$$

$$l \leq x \leq u, \tag{8}$$

where Eq. (7) refers to the linear constraints and Eq. (8) refers to the bound constraints. In addition, in an MILP, there are three main categories of decision variables:

- positive real variables ($x_j \geq 0, \forall x_j \in \mathbb{R}_+$);
- positive integer variables ($x_j \geq 0, \forall x_j \in \mathbb{Z}_+$); and
- binary variables ($x_j \in \{0, 1\}$).

In MILPs, the best objective feasible solution is called the optimal solution. The notation $x^* \in X$ designs the optimal solution to find and $\overline{x} \in X$ designs any feasible solution to the problem. In general, solving an MILP aims to generate not only a decreasing sequence of upper bounds $u_1 > \cdots > u_k \geq z^*$ but also an increasing sequence of lower bounds $l_1 < \cdots < l_k \leq z^*$. For minimization problems, any feasible solution $\overline{x} \in X$ yields an upper bound $\overline{u} = c\overline{x}$ on the optimal value, namely $\overline{u} \geq z^*$. In some cases, since it is possible to achieve function values without converging to the optimal values, MILPs then have no optimal solution but only feasible ones; those problems are called unbounded. In some other cases, no solution exists to an MILP and thus the problem is called infeasible.

To obtain lower bounds for minimization problems, we must have recourse to a relaxation of the problem.

Given (P) a minimization problem and a problem (RP) where

$$(P) \quad z^* = \min\{c(x) : x \in X \subseteq \mathbb{R}^n\}$$

and

$$(RP) \quad \widetilde{z} = \min\{\widetilde{c}\,(x) : x \in \widetilde{X} \subseteq \mathbb{R}^n\},$$

(RP) is a relaxation of (P) if $X \subseteq \widetilde{X}$ and $\widetilde{c}\,(x) \leq c(x)\ \forall x \in X$.

Proposition 1. If RP is a relaxation of P, $\widetilde{z} \leq z^*$.

Proof. Let x^* be an optimal solution of P, then $x^* \in X \subseteq \widetilde{X}$ and $\widetilde{c}(x^*) \leq c(x^*) = z^*$. Since $x^* \in \widetilde{X}$, we have $\widetilde{z} \leq \widetilde{c}(x^*)$.

Proposition 2. Let x^*_{RP} *be an optimal solution of* RP. *If* x^*_{RP} *is feasible for* $P(x^*_{RP} \in X)$ *and* $\widetilde{c}(x^*_{RP}) = c(x^*_{RP})$, *then* x^*_{RP} *is also optimal for* P.

The different types of relaxations that can be applied to an MILP problem are:

- Linear relaxation: This consists of removing the integrality constraint of a given decision variable. The resulting relaxation is a linear program. This relaxation technique transforms an NP-hard optimization problem (integer programming) into a related problem that is solvable in polynomial time (LP) and the solution to the relaxed linear program can be used to gain information about the solution to the original integer program.
- Relaxation by elimination: This relaxation technique is the most used, especially to easily compute a lower bound for a minimization MILP problem. This straightforward method consists of deleting one or more constraints.
- Lagrangian relaxation: This is a relaxation method that approximates a difficult problem of optimization by a simpler problem. The idea consists of omitting the "difficult" constraints and penalizing the violation of those constraints by adding a Lagrange multiplier for each one of them, which imposes a cost on violations for all feasible solutions. The resulted relaxed problem can often be solved more easily than the original problem.

Modeling optimization problems as MILPs can be done by different ways respecting some arrangements and by bringing the problem characteristics on constraints, decision variables, and objective functions in these different modeling approaches. Some models may be narrower than others (in terms of the number of constraints and variables required) but may be more difficult to solve than larger models. Improving the efficiency of MILP models requires a deep understanding on how solvers work. Then, the model can be solved quickly and practically.

5.2.4 LP solvers

Linear programs were efficiently solved using the simplex method, developed by George Dantzig in 1947, or interior point methods (also called barrier methods) introduced by Khachiyan in 1979 [12]. The simplex algorithm has an exponential worst-case complexity. Interior point methods were first only a proof that linear programs can be solved in polynomial time. Simply because the theoretical complexity of the simplex method is worse than that of interior point algorithms does not mean that the simplex method always performs worse in practice. The implementations of the two types of algorithms are important for run-time performance, as is the type of linear program being solved. Improved versions of both methods present the base for most powerful computer programs for solving LP problems.

During the 1950s, even small LP problems were not tractable. Several decades later, large-scale MILPs involving huge numbers of variables, inequalities, and realistically sized integer programming instances become tractable. In addition to algorithmic features, especially heuristic algorithms and cutting plane methods, progress is due to both hardware capabilities and software effectiveness (LP solvers). Nowadays, experts can easily solve an optimization model through a direct use of LP solvers.

The most competitive programming solvers [12] are CPLEX [14], GuRoBi [15], and Xpress [16]. However, CPLEX (ILOG, Inc.) is the most powerful, well-known commercial LP/CP solver and is considered an efficient popular solver due to its computational performance features. The actual

computational performance is the result of a combination of different types of improvements such as primal simplex, dual simplex, and barrier methods; branch-and-bound, branch-and-cut, and cutting-planes techniques; heuristics and Relaxation-Induced Neighborhood Search (RINS); parallelization; and so on. An interesting explanation that helps understanding the key principles in CPLEX including methods and algorithms is presented in [17].

Solving an optimization model through an LP solver requires generating vectors containing objective function coefficients, constants, and matrices of data in accepting forms; then, the specialized solution technique can be implemented. Programming languages such as C++ have been used to convert parameters and to write scripts and code them in the required form by the optimization solvers. However, modeling languages present a real advance in the ease of use of mathematical programming by making it simple to express very large and complex models. The modeling languages allow using special constructs similar to the way in which they appear when written mathematically, such as the expression of objective functions and inequalities. Thus, the development of models became more rapid and more flexible in debugging and replacing or adding constraints.

Furthermore, different algebraic modeling languages used for programming problems appeared, such as AMPL [18], GAMS [19], Mosel [16], and OPL [14]. AMPL and GAMS modeling languages can be used on a variety of solvers such as CPLEX, GuRoBi, and Xpress, and other solvers including nonlinear ones. However, Mosel is compatible with a limited number of solvers but has the advantage that it is a compiled language, which is faster to read into the solver for large models than AMPL and GAMS are.

The OPL is an algebraic programming language that makes coding easier and shorter than it is with a general-purpose programming language. It is part of the CPLEX software package and therefore tailored for the IBM ILOG CPLEX and IBM ILOG CPLEX CP Optimizers (can also resolve the constraint programming approaches). In recent years, integer programming and constraint programming approaches have been complementary strengths to solve optimization problems. The OPL programming language supports this complementarity by backing the traditional algebraic mathematical notations and integrating the modern language of constraint programming, involving the ability to choose search procedures to benefit from both technologies. It includes [20]:

- OPLScript: a script language where models are objects to make easy the solving of a sequence of models, multiple instances of the same model, or decomposition schemes such as column generation and Benders decomposition algorithms.
- The OPL component library: to integrate an OPL model directly in an application by generating C++ code or by using a COM component.
- A development environment.

5.3 Constraint programming

CO problems can be expressed on a declarative or on a procedural formulation. The declarative formulation directly expresses constraints and the objective function then attempts to find the solution without the distraction of algorithmic features. The procedural formulation defines the manner to solve the problem by providing an algorithm. The artificial intelligence community invented ways to twist declarative and procedural formulations to work together. This idea brought about the introduction of

logic programming, constraint logic programming, constraint handling rules, and constraint programming.

Constraint programming is a new paradigm evolved within operations research, computer sciences, algorithms, and artificial intelligence. As its name reveals, constraint programming is a two-level architecture including a constraint and a programming component. It is considered an emerging declarative programming language for describing feasibility problems. It consists of parameters, sets, decision variables (like mathematical programming), and an objective function and deals with discrete decision variables (binary or integer).

A constraint is a logical relation among several variables, each taking a value in a given domain. Its purpose is to restrict the possible values that variables can take. Thus, constraint programming is a logic-based method to model and solve large CO problems based on constraints. The idea of constraint programming is to solve problems by stating constraints that must be satisfied by the solution. Considering its roots in computer science, constraint programming's basic concept is to state the problem constraints as invoking procedures; then, constraints are viewed as relations or procedures that operate on the solution space. Constraints are stored in the *constraint store* and each constraint contributes some mechanisms (enumeration, labeling, domain filtering) to reduce the search space by discarding infeasible solutions using constraint satisfiability arguments. A constraint solver is used to solve the problem dealing with a *constraint satisfaction problem* (CSP), which is a manner to describe the problem declaratively by using constraints over variables. It can be seen as a set of objects that must satisfy the given constraints and hold among the given decision variables. It can be seen as a set of objects that must satisfy the given constraints and hold among the given decision variables. Therefore, it introduces the domain of possible values of finite decision variables and presents relations between those variables, then it tries to solve them by constraint satisfaction methods.

A CSP is defined as a triple (X, D, C) where

- $X = \{x_1, x_2, \ldots, x_n\}$ is the set of variables.
- $D = \{d_1 \times d_2 \times \cdots \times d_n\}$ is the domain (sets of possible values) of the variables d_i is a finite set of potential values for x_i.
- $C = \{c_1, c_2, \ldots, c_m\}$ is a set of constraints. Each constraint is a relation on the subset of domains $C_i \subseteq d_{i1} \times \cdots \times d_{il}$ which defines the possible values of the variables $x_{i1}, \ldots, x_{il}$ acceptable by the constraint C_i.

The solution of the CSP is a tuple $\langle v_1, \ldots, v_n \rangle$ assigned to the variables $x_1, \ldots, x_n$ where each variable x_i has a possible value from its domain set d_i and all constraints C are satisfied.

5.3.1 Constraint propagation

Solving a constraint programming is ensured by combining a systematic search process with inference techniques. The systematic search process consists of enumerating all possible variable-value combinations and construing a search tree where the tree's root represents the problem to be solved. At each node in the search tree, domain filtering and constraint propagation techniques are used to perform inference. Each constraint is related by a domain filtering algorithm to eliminate inconsistent values (cannot be part of any solution) from the variable domains. The domain filtering algorithm take turns until no one is able to prune a domain any more or one domain becomes empty (i.e., the problem has no solution). Practically, the most effective filtering algorithms are those associated with global constraints. The constraint propagation process is iteratively applying the domain filtering algorithm. It

is handled then by communicating the updated domains (domains where the unpromising values are discarded) to all of the constraints that are stated over this variable. Note that the aim of constraint propagation is to bring down the original problem and to know whether the problem is solved, may be feasible, or may be infeasible.

5.3.2 Global constraints

An important aspect of constraint programming is global constraints. A global constraint is a way to strengthen the constraint propagation by performing the pruning process. It consists of choosing one constraint (so-called global constraint) rather than several simple constraints. This concept may remove inconsistent solutions efficiently, eases the task of modeling the problem as CSP, and allows saving complexity and computational time.

5.3.3 Systematic search

The fundamental algorithmic step in solving a CSP is search. The search technique can be complete or not. A complete systematic search algorithm guarantees finding the solutions if they exist, then deducing the optimal solutions or proving that the problem is insoluble. The main disadvantage of these algorithms is that they take a very long computational time. An example of systematic complete algorithms is the backtracking search algorithm. It is a technique that performs a depth-first traversal of a search tree. The algorithm incrementally maintains a partial assignment that specifies consistent values with the constraints whose variables are all assigned. The beginning is with an empty assignment, and at each step, the process chooses a variable and a value in its domain. Whenever it detects a partial assignment that violates any of the constraints and cannot be extended to a solution, backtracking is performed and it returns to the most recently instantiated decision that still had alternatives available.

An example of systematic incomplete search algorithms is the local search for solving CSP. Whereas the nodes in the search tree in backtracking search are presented as partial sets of assignments to the variables, they represent complete assignments in local search. Each node is associated with a cost value resulted from a cost function and a neighborhood function is applied to detect adjacent nodes. The search progresses by iteratively selecting neighbors and applying the algorithm to look for neighbors of lowest cost because a standard cost function is the measure of the number of constraints that are not satisfied.

For further information about constraint programming, the reader is referred to [20–25].

6. Solution methods

Solving NP-hard problems has been a challenging topic for many researchers almost since the beginning of computer history (long before the concept of NP-hardness was discovered). Those problems can be solved by exact algorithms, but the best approach is to search for good approximation algorithms, heuristics, and metaheuristics. In this section, we review some of the methods used to solve the vehicle routing problem mentioned in the next chapter. However, because exact algorithms do not present practical solutions to our problem in this thesis, discussion of the techniques will be brief compared to techniques of approximation algorithms.

6.1 Exact algorithms

An exact algorithm is an algorithm that solves a problem to optimality. In most cases, this kind of algorithm generates optimal solutions (if they exist) by reducing the solution space and the number of different alternatives needed to be analyzed or by evaluating implicitly every possible solution to obtain the best one. Those methods perform well for small problems but are not very suitable to solve NP-hard problems, especially when considering their limitations for large-scale problems. In addition, they are time-consuming algorithms even for small instances where they require an exponential time.

In what follows, we review two exact approaches to solve CO problems: branching methods (Branch & Bound, Branch & Cut, Branch & Price, Branch & Infer) and dynamic programming, both of which divide a problem in different subproblems. Branching algorithms split problems into independent subproblems, solve the subproblems, and output as the optimal solution to the original problem the best feasible solution found along the search. Alternatively, a dynamic programming algorithm is appropriate for a smaller set of optimization problems. It breaks down the initial problem into not independent subproblems in a recursive manner and then recursively finds the optimal solution to the subproblems.

6.1.1 Branching algorithms

The basic branching algorithm determines a repeated dived-and-conquer strategy to find the optimal solution to a given hard problem P_0. A branching step is presented by creating a list of subproblems $P_1,\ldots,P_n$ of P_0 and each P_i is solved if it is possible. The obtained solution becomes a candidate solution of P_0 and the best candidate solution is called the *incumbent solution*. Else, if a P_i is too hard, it is branched again (treated as P_0) and so on, recursively, until all the subproblems are treated, and then the incumbent solution is the optimal solution required *Opt* (Algorithm 1).

In literature, there are many popular algorithms that exploit the branching strategy. For example, the *Branch & Bound algorithm*, which combines the branching strategy and the bounding procedure, the

Algorithm 1 An elementary branching algorithm for a CO problem.

```
Begin
   1:Opt←∞and P←{P0}
   2: WHILE P is nonempty DO
   3: Define subproblems Pi=P1,...,Pn of P0, add them to P and remove P0
   4: Choose a subproblem Pi, remove it from P and solve it
   5: IF infeasible THEN
   6:   define subproblem Pi1=P11,...,Pn1 of Pi, add them to P and remove Pi
   7: ELSE
   8:   set x is the candidate solution of Pi
   9:   set Opt←optimize{x,Opt}
  10: END
  11:RETURN Opt the optimal solution of the problem P
End
```

Branch & Cut algorithm using the branching strategy with the cutting steps, and the *Branch & Infer algorithm*, which designs the branching method with inference.

- **Branch & Bound**: The Branch & Bound algorithm is the most popular branching algorithm that can greatly reduce the solution space. It is a complete tree-based search used in both constraint programming and integer programming, but in rather different forms. In CP, the branching is combined with inference aimed at reducing the amount of choices needed to explore. In IP, the branching is linked with a relaxation (to provide the bounds), which eliminates the exploration of nonpromising nodes for which the relaxation is infeasible or worse than the best solution found so far.

  This algorithm is based on:
  - **Branching**: This step consists of dividing recursively the feasible set of a problem P_0 into smaller subproblems, organized in a tree structure, where each node refers to a subproblem P_i that only looks for the subset ("leaves" of the tree) at that node.
  - **Bounding**: By solving the equivalent relaxed problem (some constraints are ignored or replaced with less strict constraints), a lower bound and/or an upper bound are calculated for the cost of the solutions within each subset.
  - **Pruning**: Pruning by optimality where a solution cannot be improved by decomposing more of the tree, pruning by bound where the current solution in that node is worse than the best known bound, or pruning by infeasibility when the solution is infeasible. Then, there are no reasons to apply the branching for this node and the subset is discarded.

  Otherwise, the subset will be further branched and bounded and the algorithm ends when all solutions are pruned or fixed.

  The search strategy refers to how to explore the tree and how to select the next node to process: either in depth or in progress. A depth-first search strategy is most economical in memory: the last constructed node is split and the exploration is done deep down leaves of the tree until a node is pruned and discarded. In this case, the strategy returns to the last unexplored node and goes back in depth, and so on. Only one branch at a time is explored and is stored in a stack.

  In a progressive strategy (frontier search), the separate node is the one of weakest evaluation. The advantage is often to more quickly improve the best solution and thus accelerate the search since it will be more easily possible to prune nodes. This strategy consumes more memory (it can require a lot of memory to store the candidate list) and generally is much faster than the depth-first strategy. Besides, in order to accelerate the method as much as possible, it is elementary to have the best possible bounds.

  The Branch & Bound method can reduce the search step, but it is not always easily applicable because sometimes it is very hard to calculate effective lower and upper bounds. In addition, it may not be sufficient for large-scale problems where finding an optimal solution requires a long computation time.
- **Branch & Cut**: The Branch & Cut algorithm is an extension of the Branch & Bound algorithm, with the addition of cutting-plane methods to calculate the lower bound at each node of the tree to define a small feasible set of solutions. This method is exploited especially in large integer programming problems. A CO problem is formulated as an integer programming problem and is solved with exponentially many constraints where those constraints are generated on demand. The solution is determined by adding a number of valid linear inequalities (cuts) to the formulation to improve the

relaxation bound and then solving the sequence of LP relaxations resulting from the cutting-plane methods.

Cuts can be used all over the tree and are called globally valid or used only in the resultant node, in which case they are called locally valid.

The algorithm is based on:

– adding the valid inequalities to the initial formulation: this step consists of creating a formulation with better LP relaxation;
– solving current LP relaxation by cutting off the solution with valid inequalities;
– cutting and branching: only with the initial LP relaxation (root node); and
– branching and cutting: at all nodes in the tree.

The Branch & Cut approach is the heart of all modern mixed-integer programming solvers. It succeeds in reducing the number of nodes to explore and define feasible regions, which helps in easily finding the optimal solutions.

- **Branch & Price**: The Branch & Price algorithm is an extension of the Branch & Bound algorithm. It is a useful method for solving large integer linear programming (ILP) and MILP where the task is to generate strong lower and upper bounds. It is useful also for solving a list of LP relaxations of the integer LP. This method combines the Branch & Bound algorithm and column generation method. The column generation search method solves an LP with exponentially many variables where variables represent complex objects. Then, the Branch & Price method solves a mixed LP with exponentially many variables branching over column generation where at each node of the search tree, columns (or variables) can be added to the LP relaxation to improve it. At the approach's beginning, some of those variables (columns) are excluded from the LP relaxation of the large MIP in order to reduce memory and are then added back to the LP relaxation as needed where most of them may be assigned to zero in the optimal solution. Then, the major part of the columns are irrelevant for solving the problem.

  The algorithm involves:

  – developing an appropriate formulation to form the master problem, which contains many variables;
  – generating a restricted master problem to be solved, which is a modified version of the master problem and contains only some columns;
  – solving a subproblem called a pricing problem to obtain the column with a negative reduced cost; and
  – adding the column with the negative reduced cost to the restricted master problem and thus optimizing relaxation.

  If the solution to the relaxation is not integral, the branching occurs and resolves a pricing problem.

  If cutting planes are used to resolve LP relaxations within a Branch & Price algorithm, the method is known as Branch & Price & Cut.
- **Branch & Infer**: The Branch & Infer framework is a promising approach that combines the branching methods with inference and unifies the classical Branch & Cut approach from integer LP with the usual operational semantics of finite domain constraint programming [23]. This method is used for integer and finite domain constraint programming. For an ILP, and considering the constraint language of ILP with the primitive and nonprimitive constraints, we can obtain the relaxed version of a combinatorial problem by the primitive constraints. Then, inferring a new

primitive constraint corresponds to the generation of a cutting plane that cuts off some part of this relaxation. But the basic inference principle in finite domain constraint programming is domain reduction (reduction of nonprimitive constraints to primitive constraints) and for each nonprimitive constraint, a propagation algorithm plans to remove inconsistent values from the domain of the variables occurring in the constraint. Note that arithmetic constraints are primitive in ILP, whereas in CP they are nonprimitive. That is why, each arithmetic constraint is handled individually in CP.

In addition, because the primitive constraints are not expressive enough or because the complete reduction is computationally not feasible, the branching methods are applied to split the problem into subproblems and process each subproblem independently.

The Branch & Infer method is very useful not only for resolving the CO problem but also for solving CSPs. The rule system of the Branch & Infer consists of the rules *bi_infer*, *bi_branch*, and *bi_clash* together with the basic three alternatives:

- *bi_sol* (satisfiability)
- *bi_climb* (branch and bound): using lower bounding constraints to find an optimal solution
- *bi_bound, bi_opt* (branch and relax): In contrast to branch and bound, which uses only lower bounds, branch and relax works with two bounds: global lower bound *glb* and local upper bound *lub* (computed for each subproblem). The local upper bounds may introduce a new rule to prune the search tree. If a local upper bound is smaller than the best known global lower bound, then the corresponding subproblem cannot lead to a better solution and therefore can be discarded

6.1.2 Dynamic programming

A dynamic programming algorithm (whose study began in 1957 [26]) is a class of backtracking algorithm where subproblems are repeatedly solved. It is based on simplifying complicated problems by splitting them, in a recursive manner, into a finite number of related but simpler problems, resolving those subproblems, and then deducing the global solution for the original problem.

The first step is to generalize the original problem by creating an array of simpler subproblems and giving a recurrence relating some subproblems to other subproblems in the array. The second step is to find the optimal value searched for each subproblem and compute the best values (solutions) for the more complicated subproblems by exploiting values already computed for the easier problems, which is guaranteed through the above recurrence feature. The last step to use the array of values calculated to choose the best solution for the original problem.

6.2 Heuristics

To solve CO problems, exact methods are not very suitable considering their limitations for large-scale problems. Hence, in the last decades, there has been extensive research and efforts allotted to the development of approximate algorithms (heuristics and metaheuristics) that are often applied to produce good-quality solutions for optimization problems in a reasonable computation time (polynomial time).

The term heuristic is inspired from the Greek word "Heuriskein," which means to find or to discover. Heuristics are approaches applied after discovering, learning, and looking for the good-quality solutions to solve problems. Heuristics can be considered as search procedures that iteratively generate and evaluate possible solutions. However, there is no guarantee that the solution obtained will be a satisfactory solution, or that the same solution quality will be retrieved every time the algorithm is

run. But generally, the key to a heuristic algorithm's success relies on its ability to adapt to the specifications of the problem, to deal with the behavior of its basic structure, and especially to stay away from local optima.

6.2.1 The constructive heuristics

A constructive heuristic is a heuristic method that starts with an empty set of solutions and repeatedly constructs a feasible solution, according to some constructive rules, until a general complete solution is built. It varies from local search heuristics, which began from a complete solution and tried to improve it using local moves. Constructive heuristics are efficient techniques and have great interest as they generally succeed in finding satisfactory solutions in a very short time. Such solutions have the ability to be applied to identify the initial solution for metaheuristics.

The construction of the solution is made through some construction rules. Such rules are generally related to various aspects of the problem and the objective looked for. In this section, we describe constructive heuristics applied for routing problems [27, 28] where the objective is principally to minimize the travel distance while servicing all customers [29]. In this context, constructive heuristics are called *Route construction heuristics* and aim to select nodes or arcs sequentially based on cost minimization criteria until creating a feasible solution [30].

Based on cost minimization criterion, we can distinguish:

- **Nearest neighbor heuristic**: The nearest neighbor heuristic was originally initialized by Tyagi [31] in 1968. This heuristic builds routes by adding un-routed customers that respect the near neighbor criterion. At each iteration, the heuristic looks for the closest customer to the depot and then to the last customer added to the route. A new route is started unless the search fails adding customers in the appropriate positions or no more customers are waiting.
- **Savings heuristic**: The savings heuristic was originally proposed by Clarke and Wright [1] in 1964. It builds one route at a time by adding un-routed customers to the current route, with satisfying the savings criterion. The saving criterion is a measure of cost formula by combining given weights α_1, α_2, and α_3.
- **Insertion heuristic**: The insertion heuristic was originally conceived by Mole and Jameson [32] in 1976. It constructs routes by inserting un-routed customers in appropriate positions. The first step consists of initializing each route using an initialization criterion. After initializing the current route, and at each iteration, the heuristic uses two other formulations to insert a new un-routed customer into the selected route between two adjacent routed customers.
- **Sweep heuristic**: The sweep heuristic was originally implemented by Gillet and Miller [33] in 1974. It can be considered a two-phase heuristic as it includes clustering and routing steps. It begins by computing the polar angle of un-routed customers and ranking them in an ascending order. The first customer in the list is called "the reference line." Hence, un-routed customers are added to the current route with consideration to their polar angle and with respect to the depot. Physically, this process is similar to a counterclockwise sweep movement considering the depot as central point, and starting from and ending in the reference line.

Constructive heuristics are able to build feasible solutions quickly, which is an important feature because many real-world optimization problems need fast response time. Whereas, the solutions generated are for low-quality until those methods cannot look more than one iteration step ahead.

6.2.2 The improvement heuristics

In contrast to constructive heuristics, which start from an empty solution and build it iteratively, improvement heuristics begin with a complete solution and aim to improve it using simple local modifications called moves. Improvement heuristics are considered local search heuristics that only accept possibilities that enhance the solution's quality and improve its objective function.

From the beginning, the heuristic deals with a complete solution and attempts to yield a good solution by iteratively applying modification metrics. Once the alterations improve it, the current solution is replaced by the new improved solution. However, if it causes a worsening objective function, the process will be repeated and the heuristic performs acceptable operations until no more improvements are found. This improvement metric can be considered a solution intensification procedure or a guided local search.

Improvement operators proposed for this purpose are diversified. In the context of route improvement heuristics, operators can manage to move a customer from one route to another, it is the case of *inter route operators* modify multiple routes at the same time. In addition, operators can manage to exchange the position of customers in the same route, it works on a single route and is called *intraroute operators*.

Heuristic techniques are related to some vocabulary in the context of exploring promising solutions. The *search space* describes the space covering all the feasible solutions for the problem. The *local search* designs the process that improve the solution by moving from a candidate solution to another, which is generated through the *neighborhood* aspect. We say that the new improved solution is the *neighbor* of the previous one. In the search space, the neighbor of a candidate solution generated by the local search process to improve the objective function can be a local optimum or can be the best searched solution and is called the global optimum. Fig. 5 illustrates a search space including local and global optimums.

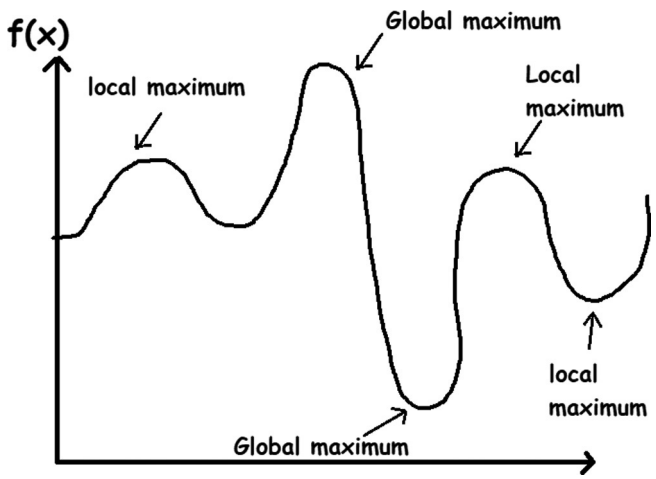

FIG. 5

Local and global optimums in a search space.

Formally, an optimization problem (minimization problem of the form $\min_{x \in S} f(x)$) may be expressed by the couple (S, f) where S represents the set of finite feasible solutions and $f: S \rightarrow \mathbb{R}$ is the objective function to optimize. The objective function assigns to each solution $x \in S$ of the search space a real number aiming to evaluate x and determining its cost. A solution $x^* \in S$ is a global optimum (global minimum) if it has a better objective function than all solutions of the search space, that is, $\forall x \in S, f(x^*) \leq f(x)$. Hence, the main goal in solving an optimization problem is to find a global optimal solution x^*.

A neighbor solution x' is an altered solution by a set of elementary moves to improve the objective function. The set of all solutions that can be reached from the current solution with respect to the move set is called the neighborhood of the current solution and is notated as $N(x) \subset S$. Relative to the neighborhood function, a solution $x \in N(x)$ is a local optimum if it has a better quality than all its neighbors, that is, $\forall x' \in N(x), f(x) \leq f(x')$.

The straightness of a local search heuristic can appear if it has the ability to escape from the local optima solutions and finds then the global optimum. In addition, improvement heuristics are powerful concepts helping conceiving and guiding the search under metaheuristics, especially when used to formulate metaheuristic initial populations.

6.3 Metaheuristics

Unlike exact algorithms, metaheuristics deliver satisfactory solutions in a reasonable computation time even when tackling large-size problems. This characteristic shows their effectiveness and efficiency to solve large problems in many areas. The term *metaheuristic* was invented by Glover in 1986 [34]. Many classification criteria may be used for metaheuristics [35]:

- **Nature inspired versus nonnature inspired**: A large number of metaheuristics are inspired from natural processes. Evolutionary algorithms are inspired by biology, ant and bee colony algorithms are inspired by social species and social sciences, and simulated annealing is inspired by physical processes.
- **Memory usage versus memoryless methods**: A metaheuristic is called memoryless if there is no information extracted dynamically to use during the search, like as in Greedy Randomized Adaptive Search Procedure (GRASP) and simulated annealing metaheuristics. However, some other metaheuristics can use some information extracted during the search to explore more promising regions and then intensify the search, like the case of short-term and long-term memories in tabu search.
- **Deterministic versus stochastic**: A deterministic metaheuristic solves an optimization problem by making deterministic decisions (e.g., tabu search), whereas stochastic metaheuristics apply some random rules during the search (e.g., simulated annealing, evolutionary algorithms). In deterministic algorithms, using the same initial solution will lead to the same final solution, whereas in stochastic metaheuristics, different final solutions may be obtained from the same initial solution.
- **Population-based search versus single-solution-based search**: Single-solution-based algorithms (e.g., tabu search, simulated annealing) manipulate and transform a single solution during the search, whereas population-based algorithms (e.g., ant colony optimization, evolutionary algorithms) develop a whole population of solutions. These two families have complementary characteristics: single-solution-based metaheuristics are exploitation oriented; they have the power

to intensify the search in local regions. Population-based metaheuristics are exploration oriented; they allow a better diversification in the whole search space.

- **Iterative versus greedy**: Most of the metaheuristics are iterative algorithms that start with a population of solutions (a complete solution) and alter it at each iteration using some search operators. Alternatively, greedy algorithms start from an empty solution, and at each step a decision variable of the problem is assigned until a complete solution is constructed (e.g., GRASP metaheuristic).

In the following, we present some important metaheuristic techniques.

6.3.1 Simulated annealing

This is one of the most popular local search-based metaheuristics. It is inspired from the physical annealing process for crystalline solids, where a solid placed in a heat bath and heated up by elevating temperature. When it melts, it is slowly cooled by gradually lowering the temperature, with the objective of relaxing toward a low-energy state and obtaining a strong crystalline structure. The fundamental aspect to a successful annealing system is related to the rate of cooling metals, which must begin with a sufficiently high temperature to not obtain flaws (metastable states) and lowering it gradually until it converges to an equilibrium state [36].

The simulated annealing search algorithm was introduced in 1982 by Kirkpatrick et al. [37] who applied the Metropolis algorithm [38] from statistical mechanics. This algorithm simulates a thermodynamical system by creating a sequence of states at a given temperature. Then, in each iteration, an atom is randomly displacing, where E is the change in energy resulting from the displacement. If the displacement causes a decrease in the system energy, that is, the energy difference ΔE between the two configurations is negative: ($\Delta E < 0$), then the new configuration is accepted and is used as a starting point for the next step. Either, if $\Delta E > 0$, the new configuration is accepted with the probability:

$$P(\Delta E) = \exp^{(-\Delta E/bT)},$$

where T is the current temperature and b is a constant called a Boltzmann constant. Depending on the Boltzmann distribution, the displacement is accepted or the old state is preserved. By repeating this process for a long time, an equilibrium state called thermal equilibrium would be reached.

Table 1 illustrates the analogy between the physical system and the optimization problem.

The simulated annealing method is a memoryless stochastic algorithm. There is no exploitation of information gathered during the search process and it includes randomization aspects to the selection process. The algorithm method starts from a feasible solution x (considering a problem of cost

Table 1 Simulated annealing analogy.

Physical system	Optimization problem
System state	Feasible solution
Energy	Objective function
Ground state	Global optimum solution
Metastable state	Local optimum
Careful annealing	Simulated annealing

Algorithm 2 The simulated annealing algorithm.

```
Begin
    1:Generate an initial solution x
    2:T←T_max/*initialize temperature*/
    3:Repeat
    4: For(i=0;i<numIterations;i++) Do
    5:  Generate a new solution x′ within the neighborhood of x(x′ ε N(x))
    6:  ΔE ← f(x′)-f(x)
    7:  If ΔE<0 Then
    8:    x ← x′
    9:  Else
   10:    p= Random (0,1)/*generate a random number in the interval (0,1)*/
   11:    If p < exp^(-ΔE/bT) Then
   12:      x ← x′
   13: T ← α * T/*reduce current temperature*/
   14:Until a stopping criterion has reached /*T<T_min*/
   15:Return the best solution x
End
```

minimization $f(x)$) and by analogy to the annealing process, a random neighbor x' is generated where only improvement moves are accepted and that if it satisfies the equation $\Delta E = f(x') - f(x) < 0$. Furthermore, the algorithm begins with a high temperature T to allow a better exploration of the search space and this temperature decreases gradually during the search process. The temperature function is updated using a constant variable α on $T(t + 1) = \alpha T(t)$, where t is the current iteration and the typical values of α vary between 0.8 and 0.99. These values can provide a very small diminution of the temperature. For each temperature, a number of moves according to the Metropolis algorithm is executed to simulate the thermal equilibrium. Finally, the algorithm will be stopped when a stopping criterion has reached (Algorithm 2).

To summarize, simulated annealing is a robust and generic probabilistic technique for locating good approximations to the global optimum in numerous search spaces to resolve CO problems [39, 40] and especially the vehicle routing problem [41–49].

6.3.2 The tabu search

The tabu search is a famous search technique coined by Glover [34] in 1986. The metaheuristic name was inspired from the word "taboo" which means forbidden. Moreover, in the tabu search approach, some possible solutions, with consideration to a short-term memory, are tabu and are then merged to a tabu list that stores nonpromising recently applied moves or solutions. Based on this process, tabu search allows a smart exploring of the search space, which helps to avoid the trap of local optima.

The tabu list is a central element of the tabu search, which aims to record the recent search moves to discard neighbors recently visited and then intensify the search in unexplored regions to encourage looking for optimal solutions. The tabu list avoids cycles and prevents visiting already visited solutions,

as it memorizes previous search trajectory through a so-called short-term memory. The short-term memory is updated at each iteration and allows the storage of the attributes of moves in the tabu list. The size of the tabu list is generally fixed and contains a constant number of tabu moves. When it is filled, some old tabu moves are eliminated to allow recording new moves and the duration that a move is maintained as tabu is called the tabu tenure.

However, when necessary, the tabu search process selects the best possible neighboring solution even if it is already in the tabu list if it would lead to a better solution than the current one. Such improvements are applied using algorithmic tools and are called aspiration criterion, which consists of accepting a tabu move if the current solution improves best till now.

The main steps of a basic tabu search algorithm for a cost minimization problem can be described in Algorithm 3.

The tabu search algorithm deals also with advanced lists called medium-term memory and long-term memory that improves the intensification and the diversification facts in the search process. The intensification is maintained through the medium-term memory where recording attributes related to the best solutions found ever (the elite solutions) to exploit them by extracting the common features to guide the search in promising areas of the search space and then to intensify the search around good solutions. The long-term memory has been proposed in tabu search to stimulate the diversification of

Algorithm 3 The tabu search algorithm.

```
Begin
   1: Initialize an empty tabu list T
   2: Generate an initial solution x
   3: Let x*←x/ * x* is the best so far solution */
   4: Repeat
   5:  Generate a subset S of solutions in N(x)/ * N(x) is the current neighborhood
       of x */
   6:  Select the best neighborhood move x′ ϵ S, where f (x′)<f (x)
   7:  If f(x′)<f(x*) Then
   8:    x*←x′/* aspiration condition: if current solution improves best now, accept
         it even if it is in the tabu list */
   9:    x←x′/* update current solution */
  10:     T←T+x′/* update tabu list */
  11: Else
  12:  If (x′ ϵ N(x)/T) Then
  13:    x←x′/* update current solution if the new solution is not tabu */
  14:    If (f(x′)<f(x*)) Then
  15:      x*←x′/* update the best till now solution if the new solution is better in
           quality */
  16:    T←T+x′/* update tabu list */
  17: Until a stopping condition has reached
  18: Return the best solution x*
End
```

Algorithm 4 The tabu search pattern.

```
Begin
   1:Initialize an empty list; empty medium term and long term memories
   2:Generate an initial solution x
   3:Let x*←x/* x* is the best so far solution */
   4:Repeat
   5: Generate a subset S of solutions in N(x)/* N(x) is the current neighborhood of
      x */
   6: Select the best neighborhood move x′ ϵ S./* nontabu or aspiration criterion
      holds */
   7:   x*←x′
   9:   Update current solution, tabulist, aspiration conditions, medium - and long
        - term memories
  10:   If intensification criterion is maintained, Then intensification
  11:   If diversification criterion is maintained, Then diversification
  12:Until a stopping condition has reached
  13:Return solution x*
End
```

the search. The information on explored regions along the search are stored in the long-term memory to investigate the unexplored regions not examined yet and then to enhance exploring the unvisited areas of the solution space.

The tabu search advanced mechanisms presented as a general tabu search pattern designed in Algorithm 4.

The tabu search is a powerful algorithmic approach. It has been widely applied with success to tackle numerous CO problems and especially the vehicle routing problem [47, 50–54] considering its flexibility and its competence to deal with complicated constraints that are typical for real-life problems.

6.3.3 Greedy randomized adaptive search procedure (GRASP)

The GRASP metaheuristic is a multistart, iterative, greedy, memoryless metaheuristic that was introduced by Feo and Resende [55] in 1989 to solve CO problems. The basic algorithm contains two main steps: construction and local search, which are called in each iteration. The construction phase consists of building a solution through a randomized greedy algorithm. Once a feasible solution is reached, its neighborhood is investigated during the local search step, which is applied to improve the constructed solution. This procedure is repeated until a fixed number of iterations is attained and the best overall solution is kept as the result (Algorithm 5).

A greedy algorithm aims to construct solutions progressively by including new elements into a partial solution until a complete feasible solution is obtained. An ordered list containing decreasing values of all candidate elements that can be included into a complete solution (that do not destroy its feasibility) is introduced and a greedy evaluation function is applied to evaluate the list and to select the next

Algorithm 5 The greedy randomized adaptive search procedure pattern.

```
Begin
    1:Initialize the number of iteration
    2:Repeat
    3: x=The_gready_randomized_algorithm;
    4: x′=The_local_search_procedure (x)
    5:Until A given number of iterations
    6:Return Best solution found
End
```

elements. This function aims to compute the function cost after including the element to be incorporated into the partial solution under construction and evaluate it. If this element brings the smallest incremental increase to the function cost, it is selected and then removed from the candidate set of solutions (Algorithm 6).

Solutions generated through greedy algorithms are not usually optimal solutions. Hence the idea to apply random steps to greedy algorithms. Randomization can diversify the search space, escape from the local traps, and generate various solutions. Greedy randomized algorithms have the same principle as the greedy procedure illustrated earlier but make use of the randomization process. The procedure starts as in simple greedy algorithm by a candidate set of elements that may incorporated in a complete solution. The evaluation of those elements is presented through the greedy evaluation function that leads to a second list including only the best elements with the smallest incremental costs, called the restricted candidate list (RCL). The element to be incorporated in the current solution is selected randomly from this list and then the set of candidate elements is updated and the incremental costs are reevaluated (Algorithm 7).

Algorithm 6 The greedy algorithm.

```
Begin
    1:x={}/* initial solution is null */
    2:Initialize the candidate set of solutions C ϵ S
    3:Evaluate the incremental cost: c(e) ∀e ϵ C
    4:Repeat
    5: Select an element e′ ϵ C with the smallest incremental cost c(e′)
    6: Incorporate e′ into the current solution x←x ∪{e′}
    7: Update candidate set C
    8: Reevaluate the incremental cost c(e)∀e ϵ C
    9:Until C=∅
   10:Return solution x
End
```

Algorithm 7 The greedy randomized algorithm.

```
Begin
    1:x={}/* initial solution is null */
    2:Initialize the candidate set of solutions C ϵ S
    3:Evaluate the incremental cost: c(e) ∀e ϵ C
    4:Repeat
    5: Construct a list including the best elements with the smallest incremental
       costs
    6: Select randomly an element e′ from the restricted candidate list
    7: Incorporate e′ into the current solution x←x∪{e′}
    8: Update candidate set C
    9: Reevaluate the incremental cost c(e)∀e ϵ C
   10:Until C=∅
   11:Return solution x
End
```

Even by using the greedy randomized algorithms, solutions generated are not guaranteed to be optimal. The neighborhood of the best solution constructed is investigated and a local search technique is applied to replace the current solution by the best solution in its neighborhood. The local search procedure is stopped when no more improvements are found in the neighborhood of the current solution (Algorithm 8).

The GRASP metaheuristic is a successful method to solve CO problems, especially vehicle routing problems [56] since it benefits from good initial solutions that usually lead to promising final solutions due to randomization. In addition to the local search procedures, it ensures the intensification and diversification of the search space, allowing it to find the global optimum. Moreover, it is easy to implement in context of algorithm development and coding, which is an especially interesting characteristic of the GRASP metaheuristic.

Algorithm 8 The local search procedure (x).

```
Begin
    1:x′←x /* The initial solution is the solution generated by the greedy
      randomized algorithm */
    2:Repeat
    3: Find x″ ϵ N(x′) with f(x″)< f(x′)
    4: x′←x″
    5:Until No more improvement found
    6:Return solution x′
End
```

6.3.4 Ant colony optimization (ACO)

The ACO algorithm is a probabilistic population-based metaheuristic for solving CO problems that was developed by Dorigo et al. [57] in 1991. The basic idea in ACO algorithms is to mimic the collective behaviors of real ants when they are looking for paths from the nest to food locations that are usually the shortest paths. The system can be considered a multiagent system where a highly structured swarm of ants cooperates to find food sources employing coordinated interactions and indirectly communicating through chemical substances known as pheromones, which is left during their trip on the ground to mark the trails to and from the food source.

The mechanism of the foraging behavior of ants is described as follows: new ants move randomly until detecting a pheromone trail and, with a high probability, they will follow this route and the pheromone is enhanced by the increasing number of ants. Generally, an ant colony is able to find the shortest path between two points due to the greater pheromone concentration. Since every ant goes back to the nest after having visited the food source, it conceives a pheromone trail in both directions. Ants taking a short path return quickly, which leads to a more rapid increase of the pheromone concentration than would occur on a longer path. Thus, concurrency ants tend to follow this path due to the fast accumulation of the pheromone, which is known as positive feedback or autocatalysis. Hence, most of the ants are directed to use the shortest path based on the experience of previous ants.

The pheromone is a volatile substance having a decreasing effect over time (evaporation process). It will vanish into the air if it is not refreshed (reinforcement process). The quantity of pheromone left depends on the remaining quantity of food, and when that amount has finished, ants will stop putting pheromones onto the trail.

Based on the ants' behavior and the concepts illustrated, the ACO algorithm relies on artificial ants to solve hard CO problems. Artificial ants will randomly construct the solution by adding some solution components iteratively to the partial solution with regard to the concentration of pheromone until a complete solution is initialized. The randomization fact is used to allow the construction of a variety of different solutions. The pheromone trails save problem-specific information about promising constructed solutions in a common memory to guide generating other solutions. It is considered the memory of the whole ant search system.

The pheromone trails are altered dynamically during the search and the pheromone is updated. Hence, given two nodes i and j, we indicate the edge in between as $(i;j)$ and the associated pheromone value as τ_{ij}. When passing an edge $(i;j)$, an ant modifies the pheromone value τ_{ij} by simply increasing it using a constant value and then applying the reinforcement phase:

$$\tau_{ij} \leftarrow \tau_{ij} + \Delta\tau.$$

The evaporation phase when pheromone values decreases over the time is similar to the evaporation of real pheromone to avoid being trapped in local optima, for example. The decrease of the pheromone concentration usually occurs in an exponential way, using an evaporation parameter $\rho \in]0, 1]$:

$$\tau_{ij} \leftarrow (1-\rho)\tau_{ij}.$$

Due to the evaporation phase, solutions of bad quality are eliminated and new areas of the search space can be discovered that enhance the diversification process to not be trapped in local optimum solutions.

The steps of ACO are described in Algorithm 9.

Algorithm 9 The ant colony optimization algorithm.

```
Begin
   1:Determine number of ants n_a
   2:Initialize pheromone trails
   3:Repeat
   4: For (k=1;k≤n_a;K++) Do /* For each ant Do */
   5:  Construct solution x_k
   6:  Update pheromone trails
   7: End
   8: Apply pheromone evaporation
   9: Apply pheromone reinforcement
  10:Until a stopping condition has reached
  11:Return solution x*
End
```

ACO algorithms have become a very popular tool to solve CO problems and they have achieved widespread success in solving different optimization problems such as real-world vehicle routing problems [6, 58–60].

6.3.5 Genetic algorithms

The baselines of heredity from parents to offsprings and the theory of evolution have inspired computer scientists since the 1950s to simulate them in designing evolutionary algorithms [61]. The most known evolutionary algorithm is the genetic algorithm developed in 1975 by John Holland [62]. Genetic algorithms are stochastic population-based metaheuristics that have been successfully applied to solve widespread complex CO problems. The main idea of genetic algorithms is to iteratively mimic the evolution of species and the survival of the fittest theories. Table 2 illustrates the analogy between the evolutionary system and the optimization problem.

Basic genetic algorithms start from a randomly generated population (Algorithm 10). Every individual of the population is composed of a set of chromosomes and is evaluated by a fitness value to determine its ability to survive. The fittest two individuals are selected to reproduce (the crossover step) and generate a super-fit offspring. The offspring is altered with a small probability (the mutation step)

Table 2 Genetic algorithm analogy.

Metaphor	Optimization problem
Individual	Feasible solution
Population	The solution set
Chromosomes	The encoded solution
Genes	Elements of the encoded solutions
Fitness	Objective function
Offspring	Generated solution

Algorithm 10 The basic genetic algorithm.

```
Begin
    1:Generate the initial population
    2:Representation
    3:Repeat
    4: Fitness evaluation
    5: Selection
    6: Crossover
    7: Mutation
    8: Replacement
   10:Until a stopping condition has reached
   11:Best individual or best population found
End
```

and the new generated offspring take place in the new population (the replacement step) until a termination criterion is reached and the best individual or population is found.

In the first step, an initial population of solutions is generated, which will be improved in the next steps of the genetic algorithm. Basically, the initial population is produced randomly, but it can be generated using constructive heuristics or some single solution-based heuristics such as tabu search, GRASP, and so on. Next, the representation step consists of initializing and encoding the initial population and split the individuals composing it on a set of chromosomes including the genes. Traditionally, the binary representation was the most used method to encode the initial population, but nowadays various representations are used and are generally problem-specific representations such as real-valued vectors, permutations, and real discrete vectors.

The fitness function refers to the objective function that associates a cost to each feasible solution to describe its quality. The fitness evaluation in a genetic algorithm consists of judging the ability of individuals to survive through a fitness value to compare them at each iteration. Then, based on the fitness evaluation, the fittest individuals are selected. The selection step is one of the fundamental components of a genetic algorithm that may lead to encouraging solutions. Individuals are chosen for reproducing according to their fitness by mean of selection strategies [35], such as:

- Roulette wheel selection: This is the most applied selection strategy. It consists of accrediting to each individual a selection probability that is proportional to its fitness value. The probability P_i of an individual i having a fitness f_i is:

$$P_i = f_i / \left(\sum_{j=1}^{n} f_j \right).$$

- Tournament selection: In tournament selection, the size k of the tournament group is fixed from the beginning and the strategy involves randomly selecting k individuals from the population. Then a tournament is applied to them and the best one is chosen.

- Rank-based selection: A high rank is associated to individuals with good fitness and the rank is scaled linearly using the following formula:

$$P(i) = 2 - s/\mu + 2 \cdot r(i)(s-1)/\mu(\mu-1)$$

  where s is the selection pressure ($1.0 < s < 2.0$), μ is the size of population, and $r(i)$ is the rank associated with the individual i.

The selection strategy leads to choosing two parents to reproduce. A crossover method is applied by combining the genes of two individuals, which produces a new offspring. The traditional genetic algorithm used strings of bits to present chromosomes. Three classical crossover operators have been widely implemented:

- One-point crossover: A point from each chromosome is selected randomly and the two chromosomes are cut at the corresponding points and the blocks are exchanged. The one-point crossover is the basic crossover operator.
- Two-point crossover: In each chromosome, two points are chosen randomly and the blocks between the two points are exchanged by the two chromosomes.
- Uniform crossover: Two individuals are recombined independently of their size. Each gene in the offspring is selected randomly from one of the parents. Each parent has equal probability to generate the offspring.

In addition to these three crossover operators, various crossover operators have been produced for real-valued representations in addition to others that are problem-specific crossover operators that enhance the chance to develop the fittest offsprings [35, 63].

Unlike the crossover that is considered a binary operator, the mutation is a unary operator acting on a single individual. It aims to alter with small probability the generated offspring. The most popular mutation operator is the inversion mutation where a random gene changes its position and there exist other mutation techniques to bring small alterations to the individual and thus guarantee the diversification in the search space. The whole process will be repeated until a stopping criterion is reached, which can be a fixed number of iterations, a lapse of time that passed, or when there are no more improvements in the generated results. The genetic algorithm usually finds promising results until it deals with both intensification and diversification on the search space. For these reasons, it has been widely applied to tackle different optimization problems, especially vehicle routing problems [63–71].

7. Conclusion

In this chapter, we illustrated a technical background for different materials and notations related to the CO problem. First, we introduced CO problems and discussed the analysis and complexity of algorithms. Then, as solving a problem requires modeling it before hand, we described concepts of modeling CO problems, including graph theory, mathematical models and programming, and constraint programming techniques. Finally, we highlighted some solution methods for solving those problems, including exact algorithms, heuristics, and metaheuristics.

References

[1] G. Clarke, J.W. Wright, Scheduling of vehicles from central depot to number of delivery points, Oper. Res. 12 (4) (1964) 568–581.
[2] G. Dantzig, J. Ramser, The truck dispatching problem, Manag. Sci. 6 (1) (1959) 80–91.
[3] M. Flood, The traveling-salesman problem, Oper. Res. 4 (1) (1956) 61–75.
[4] H. Kellerer, U. Pferschy, D. Pisinger, Knapsack Problems, Springer, Berlin, Germany, 2004, https://doi.org/10.1007/978-3-540-24777-7.
[5] J.M.V. De Carvalho, LP models for bin packing and cutting stock problems, Eur. J. Oper. Res. 141 (2002) 253–273.
[6] J. Euchi, R. Mraihi, The urban bus routing problem in the Tunisian case by the hybrid artificial ant colony algorithm, Swarm Evol. Comput. 2 (2012) 15–24.
[7] J. Watkins, Across the Board: The Mathematics of Chessboard Problems, Princeton University Press, 2004.
[8] R. Lewis, Metaheuristics can solve Sudoku puzzles, J. Heuristics 13 (2007) 387–401.
[9] B. Bouzy, T. Cazenave, Computer go: an AI oriented survey, Artif. Intell. 132 (2001) 39–103.
[10] T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein, Introduction to Algorithms, second ed., MIT Press, 2001.
[11] M.S. Bazaraa, J.J. Jarvis, H.D. Sherali, Linear Programming and Network Flows, second ed., John Wiley & Sons, Inc., New York, NY, 1990.
[12] A.M. Newman, M. Weiss, A survey of linear and mixed-integer optimization tutorials, INFORMS Trans. Educ. 14 (1) (2013) 26–38.
[13] G.B. Dantzig, Linear Programming and Extensions, Princeton University Press, 1998.
[14] IBM, ILOG CPLEX IBM, International Business Machines Corporation, Incline Village, NV, 2009.
[15] GuRoBi, GuRoBi Optimizer, GuRoBi Optimization Inc., Houston, 2009.
[16] FICO, Xpress-MP Optimization Suite, 2008. http://www.fico.com/en/Products/OMTools/Pages/FICO-Xpress-Optimization-Suite.aspx,Minneapolis.
[17] R. Lima, IBM ILOG CPLEX What is Inside of the Box?, EWO Seminar Carnegie Mellon University, 2010.
[18] R. Fourer, D. Gay, B. Kernighan, AMPLA Modelling Language for Mathematical Programming, Thomson Brooks/Cole, Pacific Grove, CA, 2003.
[19] GAMS, GAMS Distribution 23.9.1, GAMS, Washington, DC, 2012.
[20] P. Van Hentenryck, Constraint and integer programming in OPL, INFORMS J. Comput. 14 (4) (2002) 345–372.
[21] P. Laborie, J. Rogerie, P. Shaw, P. Vilam, IBM ILOG CP optimizer for scheduling, Constraints 23 (2) (2018) 210–250, https://doi.org/10.1007/s10601-018-9281-x.
[22] K.R. Apt, Principles of Constraint Programming, Cambridge University Press, 2003.
[23] A. Bockmayr, T. Kasper, Branch and Infer: a unifying framework for integer and finite domain constraint programming, INFORMS J. Comput. 10 (3) (1998) 287–300.
[24] B. De Backer, V. Furnon, P. Shaw, P. Kilby, P. Prosser, Solving vehicle routing problems using constraint programming and metaheuristic, J. Heuristics 6 (4) (2000) 501–523.
[25] P. Shaw, Using constraint programming and local search methods to solve vehicle routing problems, in: Proceedings of the CP-98, 1998, pp. 417–431.
[26] R. Bellman, Dynamic Programming, Princeton University Press, 1957.
[27] O. Bräysy, M. Gendreau, Vehicle routing problem with time windows, part I: route construction and local search algorithms, Transp. Sci. 39 (1) (2005) 104–118.
[28] O. Bräysy, M. Gendreau, Vehicle routing problem with time windows, part II: metaheuristics, Transp. Sci. 39 (1) (2005) 119–139.
[29] P. Toth, D. Vigo, An overview of vehicle routing problems, in: The Vehicle Routing Problem, Society for Industrial and Applied Mathematics, Philadelphia, PA, 2001, pp. 1–26 (Chapter 1).

[30] M.M. Solomon, Algorithms for the vehicle routing and scheduling problems with time window constraints, Oper. Res. 35 (2) (1987) 166–324.
[31] M. Tyagi, A practical method for the truck dispatching problem, J. Oper. Res. Soc. Jpn 10 (1968) 76–92.
[32] R. Mole, S. Jameson, A sequential route-building algorithm employing a generalised savings criterion, Oper. Res. Q. 27 (1976) 503–511.
[33] B. Gillet, L. Miller, A heuristic algorithm for the vehicle-dispatch problem, Oper. Res. 22 (1974) 340–349.
[34] F. Glover, Future paths for integer programming and links to artificial intelligence, Comput. Oper. Res. 13 (5) (1986) 533–549.
[35] E.G. Talbi, Metaheuristics: From Design to Implementation, John Wiley & Sons, 2009.
[36] B. Rabbouch, F. Saâdaoui, R. Mraihi, Efficient implementation of the genetic algorithm to solve rich vehicle routing problems, Oper. Res. 21 (2021) 1763–1791.
[37] S. Kirkpatrick, C.D. Gelatt, M.P. Vecchi, Optimization by simulated annealing, Science 220 (4598) (1983) 671–680.
[38] N. Metropolis, A. Rosenbluth, M. Rosenbluth, A. Teller, E. Teller, Equation of state calculations by fast computing machines, J. Chem. Phys. 21 (6) (1953) 1087–1092.
[39] P.J.M. Van Laarhoven, E.H.L. Aarts, Simulated Annealing: Theory and Applications, D. Reidel/Kluwer Academic Publishers, Dordrecht, Boston, Norwell, Massachusetts, 1987.
[40] R.W. Eglese, Simulated annealing: a tool for operational research, Eur. J. Oper. Res. 46 (3) (1990) 271–281.
[41] S. Afifi, D. Dang, A. Moukrim, A simulated annealing algorithm for the vehicle routing problem with time windows and synchronization constraints, in: 7th International Conference, Learning and Intelligent Optimization (LION7), Catania, Italy, 2013, pp. 259–265.
[42] S. Birim, Vehicle routing problem with cross docking: a simulated annealing approach, Procedia Soc. Behav. Sci. 235 (2016) 149–158.
[43] A.V. Breedam, Improvement heuristics for the vehicle routing problem based on simulated annealing, Eur. J. Oper. Res. 86 (3) (1995) 480–490.
[44] W.-C. Chiang, R.A. Russell, Simulated annealing metaheuristics for the vehicle routing problem with time windows, Ann. Oper. Res. 63 (1) (1996) 3–27.
[45] S.C.H. Leung, J. Zheng, D. Zhang, X. Zhou, Simulated annealing for the vehicle routing problem with two-dimensional loading constraints, Flex. Serv. Manuf. J. 22 (1) (2010) 61–82.
[46] S.-W. Lin, V.F. Yu, S.-Y. Chou, Solving the truck and trailer routing problem based on a simulated annealing heuristic, Comput. Oper. Res. 36 (2009) 1683–1692.
[47] I.H. Osman, Metastrategy simulated annealing and tabu search algorithms for the vehicle routing problem, Ann. Oper. Res. 41 (1993) 421–451.
[48] R. Tavakkoli-Moghaddam, N. Safaei, Y. Gholipour, A hybrid simulated annealing for capacitated vehicle routing problems with the independent route length, Appl. Math. Comput. 176 (2) (2006) 445–454.
[49] S. Yu, C. Ding, K. Zhu, A hybrid GA-TS algorithm for open vehicle routing optimization of coal mines material, Expert Syst. Appl. 38 (8) (2011) 10568–10573.
[50] J.F. Cordeau, G. Laporte, A. Mercier, A unified tabu search heuristic for vehicle routing problems with time windows, J. Oper. Res. Soc. 52 (8) (2001) 928–936.
[51] J.F. Cordeau, G. Laporte, A. Mercier, Improved tabu search algorithm for the handling of route duration constraints in vehicle routing problem with time windows, J. Oper. Res. 55 (5) (2004) 542–546.
[52] S. Faiz, S. Krichen, W. Inoubli, A DSS based on GIS and tabu search for solving the CVRP: the Tunisian case, Egypt. J. Remote Sens. Space Sci. 17 (1) (2014) 105–110.
[53] S. Krichen, S. Faiz, T. Tlili, K. Tej, Tabu-based GIS for solving the vehicle routing problem, Expert Syst. Appl. 41 (14) (2014) 6483–6493.
[54] J.A. Sicilia, C. Quemada, B. Royo, D. Escuin, An optimization algorithm for solving the rich vehicle routing problem based on variable neighborhood search and tabu search metaheuristics, J. Comput. Appl. Math. 291 (2016) 468–477.

[55] T.A. Feo, M.G.C. Resende, A probabilistic heuristic for a computationally difficult set covering problem, Oper. Res. Lett. 8 (1989) 67–71.
[56] H. Yahyaoui, S. Krichen, A. Dekdouk, A decision model based on a GRASP genetic algorithm for solving the vehicle routing problem, Int. J. Appl. Metaheuristic Comput. 9 (2) (2018) 72–90.
[57] M. Dorigo, V. Maniezzo, A. Colorni, The ant system: an autocatalytic optimization process, Dept. of Electronics, Politecnico di Milano, Italy, 1991. Tech. Rep.
[58] Y. Li, H. Soleimani, M. Zohal, An improved ant colony optimization algorithm for the multi-depot green vehicle routing problem with multiple objectives, J. Clean. Prod. 227 (2019) 1161–1172.
[59] X. Wang, T.M. Choi, H. Liu, X. Yue, A novel hybrid ant colony optimization algorithm for emergency transportation problems during post-disaster scenarios, IEEE Trans. Syst. Man Cybern. Syst. 48 (4) (2018) 545–556.
[60] T. Yalian, An improved ant colony optimization for multi-depot vehicle routing problem, Int. J. Eng. Technol. 8 (5) (2016) 385–388.
[61] A. Fraser, Simulation of genetic systems by automatic digital computers I. Introduction, Aust. J. Biol. Sci. 10 (1957) 484–491.
[62] J.H. Holland, Adaptation in Natural and Artificial Systems, MIT Press, Cambridge, MA, 1975.
[63] J.Y. Potvin, S. Bengio, The vehicle routing problem with time windows—part II: genetic search, INFORMS J. Comput. 8 (1996) 165–172.
[64] B.M. Baker, M.A. Ayechew, A genetic algorithm for the vehicle routing problem, Comput. Oper. Res. 30 (5) (2003) 787–800.
[65] T.D. Berov, A vehicle routing planning system for goods distribution in urban areas using Google maps and genetic algorithm, Int. J. Traffic Transp. Eng. 6 (2) (2016) 159–167.
[66] P.L.N.U. Cooray, T.D. Rupasinghe, Machine learning-based parameter tuned genetic algorithm for energy minimizing vehicle routing problem, J. Ind. Eng. 2017 (2017). 3019523.
[67] S. Karakatic, V. Podgorelec, A survey of genetic algorithms for solving multi depot vehicle routing problem, Appl. Soft Comput. 27 (2015) 519–532.
[68] M.A. Mohammed, M.K.A. Ghani, R.I. Hamed, S.A. Mostafa, M.S. Ahmad, D.A. Ibrahim, Solving vehicle routing problem by using improved genetic algorithm for optimal solution, J. Comput. Sci. 21 (2017) 255–262.
[69] P.R.O. da Costa, S. Mauceri, P. Carroll, F. Pallonetto, A genetic algorithm for a green vehicle routing problem, Electron Notes Discrete Math. 64 (2018) 65–74.
[70] A. Ramalingam, K. Vivekanandan, Genetic algorithm based solution model for multi-depot vehicle routing problem with time windows, Int. J. Adv. Res. Comput. Commun. Eng. 3 (11) (2014) 8433–8439.
[71] T. Vidal, T.G. Crainic, M. Gendreau, N. Lahrichi, W. Rei, A hybrid genetic algorithm for multidepot and periodic vehicle routing problems, Oper. Res. 60 (3) (2012) 611–624.