Hybrid optimization approaches for vehicle routing problems with profits

Trong-Hieu Tran

RÉPUBLIQUE FRANÇAISE
Liberté Égalité Fraternité

Université de Toulouse

Doctorat de l'Université de Toulouse

préparé à l'Université Toulouse III - Paul Sabatier

Méthodes d'optimisation hybrides pour des problèmes de routages avec profits

---

Thèse présentée et soutenue, le 13 décembre 2023 par

Trong Hieu TRAN

École doctorale
EDMITT - Ecole Doctorale Mathématiques, Informatique et Télécommunications de Toulouse

Spécialité
Informatique et Télécommunications

Unité de recherche
IRIT : Institut de Recherche en Informatique de Toulouse

Thèse dirigée par
Hélène FARGIER et Cédric PRALET

Composition du jury
M. Christian ARTIGUES, Président, LAAS-CNRS Toulouse
M. Gilles AUDEMARD, Rapporteur, Université d'Artois
M. Aziz MOUKRIM, Rapporteur, Université de Technologie de Compiègne
Mme Christine SOLNON, Examinatrice, INSA de Lyon
Mme Hélène FARGIER, Directrice de thèse, IRIT-CNRS, Université Toulouse III - Paul Sabatier
M. Cédric PRALET, Co-directeur de thèse, ONERA

THÈSE

En vue de l'obtention du

DOCTORAT DE L'UNIVERSITÉ DE TOULOUSE

Délivré par : *l'Université Toulouse 3 Paul Sabatier (UT3 Paul Sabatier)*

Présentée et soutenue le *13/12/2023* **par :**

Trong Hieu TRAN

Hybrid optimization approaches for vehicle routing problems with profits

---

JURY

Gilles AUDEMARD	Professeur d'Université	Rapporteur
Christian ARTIGUES	Directeur de Recherche	Examinateur
Hélène FARGIER	Directrice de Recherche	Directrice de thèse
Aziz MOUKRIM	Professeur d'Université	Rapporteur
Cédric PRALET	Directeur de Recherche	Directeur de thèse
Christine SOLNON	Professeur d'Université	Examinateur

École doctorale et spécialité :
MITT : Domaine STIC : Intelligence Artificielle

Unité de Recherche :
Institut de Recherche en Informatique de Toulouse (IRIT-CNRS, UMR 5505)

Directeur(s) de Thèse :
Hélène FARGIER et *Cédric PRALET*

Rapporteurs :
Gilles AUDEMARD et *Aziz MOUKRIM*

Résumé

L'optimisation combinatoire est une branche de l'optimisation mathématique qui se concentre sur la recherche de solutions optimales parmi un ensemble fini de combinaisons possibles, tout en respectant un ensemble de contraintes et en maximisant ou minimisant une fonction objective. Pour résoudre ces problèmes, les méthodes incomplètes sont souvent utilisées en pratique, car ces dernières peuvent produire rapidement des solutions de haute qualité, ce qui est un point critique dans de nombreuses applications.

Dans cette thèse, nous nous intéressons au développement d'approches hybrides qui permettent d'améliorer la recherche incomplète en exploitant les méthodes complètes. Pour traiter en cas pratique, nous considérons ici le problème de tournées de véhicules avec profits, dont l'objectif est de sélectionner un sous-ensemble de clients à visiter par des véhicules de manière à maximiser la somme des profits associés aux clients visités.

Plus précisément, nous visons tout d'abord à améliorer les algorithmes de recherche incomplets en exploitant les connaissances acquises dans le passé. L'idée centrale est de : (i) apprendre des conflits (combinaisons de décisions qui conduisent à une violation de certaines contraintes ou à une sous-optimalité des solutions) et les utiliser pour éviter de réexaminer les mêmes solutions et guider la recherche, et (ii) exploiter les bonnes caractéristiques de solutions élites afin de produire de nouvelles solutions ayant une meilleure qualité. En outre, nous étudions le développement d'un solveur générique pour des problèmes de routage complexes pouvant impliquer des clients optionnels, des véhicules multiples, des fenêtres temporelles multiples, des contraintes supplémentaires, et/ou des temps de transition dépendant du temps. Le solveur générique proposé exploite des sous-problèmes pour lesquels des méthodes de raisonnement dédiées sont disponibles.

L'efficacité des approches proposées est évaluée par diverses expérimentations sur des instances classiques et sur des données réelles liées à un problème d'ordonnancement pour des satellites d'observation de la Terre, qui inclut éventuellement des profits incertains.

Mots-clés : *Recherche opérationnelle, optimisation combinatoire, méthodes hybrides complètes/incomplètes, métaheuristiques, apprentissage de clauses, programmation dynamique, routage avec profits, satellites d'observation.*

Abstract

Combinatorial optimization is an essential branch of computer science and mathematical optimization that deals with problems involving a discrete and finite set of decision variables. In such problems, the main objective is to find an assignment that satisfies a set of specific constraints and optimizes a given objective function. One of the main challenges is that these problems can be hard to solve in practice. In many cases, incomplete methods are preferred to complete methods since the latter may have difficulties in solving large-scale problems within a limited amount of time. On the other hand, incomplete methods can quickly produce high-quality solutions, which is a critical point in numerous applications.

In this thesis, we investigate hybrid approaches that enhance incomplete search by exploiting complete search techniques. For this, we deal with a concrete case study, which is the vehicle routing problem with profits. In particular, we aim to boost incomplete search algorithms by extracting some knowledge during the search process and reasoning with the knowledge acquired in the past. The core idea is two-fold: (i) to learn conflicting solutions (that violate some constraints or that are suboptimal) and exploit them to avoid reconsidering the same solutions and guide search, and (ii) to exploit good features of elite solutions in order to hopefully generate new solutions having a higher quality. Furthermore, we investigate the development of a generic framework by decomposing and exchanging information between sub-modules to efficiently solve complex routing problems possibly involving optional customers, multiple vehicles, multiple time windows, multiple side constraints, and/or time-dependent transition times.

The effectiveness of the approaches proposed is shown by various experiments on both standard benchmarks (e.g., the Orienteering Problem and its variants) and real-life datasets from the aerospace domain (e.g., the Earth Observation Satellite scheduling problem), and possibly involving uncertain profits.

Keywords: *Operations research, combinatorial optimization, hybrid complete/incomplete approaches, metaheuristics, clause learning, dynamic programming, routing with profits, orienteering problems, Earth Observation Satellites.*

Acknowledgments

First and foremost, I would like to express my deepest gratitude to my supervisors, Cédric Pralet and Hélène Fargier, for their invaluable guidance and continuous encouragement throughout my doctoral journey. Their expertise and mentorship have been instrumental in shaping this thesis. Their patience and advice were remarkable, and working with them was an incredibly honorable experience for me.

I am profoundly grateful to the reviewers of my manuscript, Prof. Gilles Audemard and Prof. Aziz Moukrim, for their thorough evaluation and valuable suggestions, which have significantly enhanced the quality of this thesis. I also extend my sincere appreciation to the members of my doctoral committee, Christian Artigues and Prof. Christine Solnon, for their time, insightful feedback, and constructive criticism, all of which greatly contributed to refining this manuscript.

I would like to acknowledge the institutes where I conducted my research - IRIT, ONERA, and ANITI - for providing the necessary resources and facilities essential for the successful completion of this study.

To my family - Ba, Má, and Huy - your unconditional love, encouragement, and sacrifices have been my source of strength and motivation. Your constant support is invaluable, and I would like to express my deepest gratitude for all you do for me. This thesis is a testament to your tireless support and belief in me.

To my beloved Hà, who has stood by me through the highs and lows of this journey, your love and support have been my guiding light throughout this journey. I am also grateful for the companionship and joy brought by you, Húc, and Hem, who provided comfort during challenging times. I am thrilled to share this milestone with you and look forward to many more in the future.

Special thanks to my teammates at ONERA and ANITI - Louis, Samuel, Romain B., D. Anh, Anouck, Guillaume, Florent, Romain G., Jérôme - for their camaraderie, collaboration, and shared experiences, which have enriched my research and personal growth. Thanks should also go to my dear friends, Dr. Yung and Le Minh, with whom I have shared a longstanding friendship long before the inception of this thesis.

I would be remiss in not mentioning my esteemed teachers at INSA Toulouse, Marie-Jo Huguet, and Mohamed Siala, whose belief in my abilities and encouragement ignited the spark for pursuing this thesis. Your guidance continues to resonate in my work.

Lastly, I wish to express my gratitude to all those who have supported me in ways seen and unseen throughout this journey. Your contributions, whether large or small, have not gone unnoticed and are deeply appreciated.

Toulouse, 12 March 2024
Trong-Hieu TRAN

Contents

List of Figures

List of Tables

List of Algorithms

Part I

Background and context

CHAPTER 1

Introduction

Combinatorial optimization is an essential branch of computer science and mathematical optimization that deals with problems involving a discrete and finite set of decision variables. In such problems, the main objective is to find an assignment that satisfies a set of specific constraints and optimizes a given objective function.

Combinatorial optimization emerges in many research fields, ranging from operations research to artificial intelligence. It can handle numerous real-world applications impacting our daily lives, from industrial engineering to management science, from the biological industry to the medical field, from Earth observation problems to the space sector in general. For example, in logistics and transportation, optimizing delivery routes can lead to significant cost savings and higher customer satisfaction. In scheduling, efficiently allocating resources can maximize productivity and minimize downtime. For biological insights, combinatorial optimization enhances genome sequencing for DNA analysis and can be used to design proteins. In the aerospace industry, scheduling satellite activities facilitates data collection for space missions. Overall, the applications are diverse, highlighting the practical relevance of solving combinatorial optimization problems.

Most of the combinatorial optimization problems are NP-hard, hence finding an optimal solution can be very time-consuming or even computationally intractable. In many practical cases, these problems are indeed hard to solve due to the large size of the instances and/or the presence of complex constraints such as temporal constraints and/or capacity constraints. Also, some variants may involve the uncertainty or time-dependency aspect that models

real-life issues. For example in routing problems, the transition times between customers may depend on some specific conditions (e.g., congestion or rush hours) that vary throughout the day.

Solving techniques Solving a combinatorial optimization problem means exploring the solution space to identify an optimal solution. This process is often referred to as a *search* technique, as it involves traversing different regions of the solution space to search for the best solution or solutions that approximate the optimal objective value. Over the decades, numerous researchers have studied both *complete* and *incomplete* methods for solving combinatorial optimization problems.

- Complete methods, also known as exhaustive search algorithms, aim to find an optimal solution despite the hardness of the problem. This can be done by systematically searching the entire solution space and selecting a solution that satisfies the constraints while optimizing the objective function. Complete methods guarantee the finding of an optimal solution, but the search can be computationally expensive and such methods may not be feasible for large problem instances.
- Incomplete methods, on the other hand, do not guarantee the finding of an optimal solution, but rather aim to find a good solution within an acceptable amount of time. These methods are typically faster than complete methods because they do not explore the entire search space, but instead, use heuristics or other techniques to quickly guide the search toward promising solutions.

In many practical cases, incomplete methods are often preferred to complete methods. This is mainly because complete methods have difficulties in solving large-scale problems within a limited amount of time. On the other hand, incomplete methods can quickly produce high-quality solutions even within a limited computational time, which is a critical point in many applications. Despite the widespread use of incomplete search algorithms, there remain many limitations: such algorithms may get stuck in local optima and the solution quality heavily depends on the choice of the heuristics and search strategies. To address these challenges, there is a need for techniques that can help prevent the search algorithm from getting stuck in local optima and guide it toward promising regions of the search space. This is why in the literature, numerous hybrid approaches are proposed to increase the efficiency

of incomplete search methods by exploiting various algorithmic ideas from complete techniques.

Case study: vehicle routing problems with profits Among various combinatorial optimization problems, in this thesis, we mainly focus on a well-known variant of the Vehicle Routing Problem (VRP), where a set of vehicles must visit a set of customers. In this variant, a profit is associated with each customer and the primary objective is to find an optimal solution, which means to find, given the vehicle available, sequences of visits traversing a subset of customers and maximizing the profits acquired in the end.

Despite its broad applications to different sectors like logistics or tourism, this variant has not received much attention for practical applications in the aerospace sector. One such application, which is also the main motivation behind this research, is the problem of scheduling (Agile) Earth Observation Satellite missions. The latter can be modeled as a complex variant of the routing problem involving optional customer selections, time window constraints, time-dependent transition times, and even time-dependent profits. Concerning solving techniques for large-size vehicle routing problem with profits, most of the existing methods in the literature use incomplete search. Yet, there exist relatively few hybrid complete/incomplete approaches for efficiently solving different problem variants.

Research statement and contributions Given the challenges mentioned earlier, the topic of this thesis is the following:

In this thesis, we investigate the enhancement of incomplete search techniques by exploiting knowledge acquired during the search, together with techniques used in complete search.

More specifically, the subsequent chapters delve into the details of the hybrid approaches that we propose for a practical case study, namely vehicle routing problems with profits. The remainder of this dissertation is organized as follows.

- In Chapter 2, we first provide the formal background and notations about the combinatorial optimization problems and incomplete techniques. Then, we specifically focus on hybrid approaches between complete and incomplete techniques available in the literature with the objective of enhancing these incomplete techniques.

- In Chapter 3, we present an overview of the case study considered in this thesis, that is the routing problems with profits. In particular, we are interested in the Orienteering Problem (OP) and its variations, a specific problem class within this family that has received a lot of attention in the literature and for which there exist many variants. We also briefly describe its practical applications to the aerospace domain, for instance, for the (Agile) Earth Observation Satellite scheduling problem.

- In Chapter 4, we present a hybrid approach combining incomplete search with conflict learning and provide an empirical study of how to efficiently manage these conflicts with an application to the Orienteering Problems with Time Windows (OPTWs). The key idea is to learn conflicting solutions (that violate some constraints or that are suboptimal) and exploit them to avoid reconsidering the same solutions and to guide search.

- In Chapter 5, we present a novel method to exploit good features of elite solutions using dynamic programming techniques. The objective is to enhance the performance of a black-box incomplete search algorithm by recombining high-quality solutions in an effective way. We conduct various experiments on a classical OPTW benchmark and a novel realistic benchmark generated for the satellite scheduling problem, considering different factors such as time-dependent transition times or reward uncertainty.

- In Chapter 6, we present a generic framework that can efficiently solve complex routing problems by decomposing them into subproblems, for which efficient reasoning mechanisms can be used. The objective is to avoid developing a new specific algorithm for each variant of the routing problem with profits. The framework proposed is able to efficiently solve variants that possibly involve optional customers, multiple vehicles, multiple time windows, multiple side constraints, and/or time-dependent transition times.

- In Chapter 7, we summarize our main contributions and provide directions for future research.

CHAPTER 2

Background and notations

In this chapter, we present a brief overview of well-known hybrid approaches for solving hard combinatorial optimization problems. Our main focus is the hybridization between incomplete and complete techniques, whose objective is to enhance traditional incomplete search algorithms. Before discussing these hybridizations, we first introduce several basic notations related to combinatorial optimization problems and then describe basic incomplete solving techniques such as construction heuristics, local search, and metaheuristics.

2.1 Combinatorial optimization problems

2.1.1 Problem definition

Definition 2.1. *A Combinatorial Optimization Problem (COP) $\mathcal{P}$ is defined by:*

- *a set of variables $X = \{x_1, \ldots, x_n\}$;*
- *a set of finite and discrete variable domains $D = \{D_1, \ldots, D_n\}$;*
- *(optional) a set of constraints $C = \{C_1, \ldots, C_k\}$, where a constraint C_i is a relation defined over k variables $x_{i_1}, \ldots, x_{i_k}$:*

$$C_i : D_{i_1} \times \ldots \times D_{i_k} \rightarrow \{0, 1\}$$

- *an objective function f to be optimized (minimized or maximized), where*

$$f : D_1 \times \ldots \times D_n \rightarrow \mathbb{R}$$

In this definition, the adjective 'combinatorial' refers to the presence of finite and discrete variable domains. In the case where the domains D_i are continuous, $\mathcal{P}$ is called a *continuous* optimization problem. In practice, real-life problems can be more complex, and the goal is often to optimize several objective functions at the same time. Such problems are called the *multi-objective* optimization problems.

Definition 2.2. *A **solution** to a combinatorial optimization problem $\mathcal{P} = (X, D, C, f)$ is an assignment of values to the variables $s = \{(x_i, v_i) \mid x_i \in X, v_i \in D_i\}$. The set of all possible assignments, denoted by $\mathcal{S}_\mathcal{P}$ or shortly $\mathcal{S}$, is also called a search (or solution) space. A solution s is called **feasible** or **valid** if it satisfies all constraints $C_1, \ldots, C_k$. Besides, a solution s is called a **partial** solution if one or more variables are not assigned yet in s, otherwise s is called a **complete** solution.*

Solving a COP corresponds to finding an optimal solution $s^* \in \mathcal{S}$, i.e. a feasible solution providing an optimal objective value. The objective function value of a solution is also called *solution quality*. As a result, solutions that provide an objective function value close to the optimal value are considered to be of *high-quality* or *good-quality*. The problem can be stated as a minimization or a maximization problem depending on whether the given objective function f is to be minimized or maximized. Without loss of generality, we deal with a **minimization** problem where we try to minimize the objective function value. Then, the globally optimal solutions can be formally defined as follows.

Definition 2.3. *A solution s^* is called **globally optimal** if and only if*

$$s^* \in \mathcal{S} \wedge f(s^*) \leq f(s) \ \forall s \in \mathcal{S}$$

It is also useful to clarify the distinction between *problems* and *problem instances*. In general, a problem (also called a *problem class* or an *abstract problem*) can be seen as a set of problem *instances* that contain specific input data (e.g., variable domains, constraints, and objective functions). For example, given the problem of 'finding a shortest path that visits a set of points in the Euclidean plan' that is also known as the Traveling Salesman

Problem (TSP) (Bellmore & Nemhauser (1968)), an *instance* of this problem corresponds to a specific set of input data, a *solution* to this problem instance is a path connecting all given points regardless of length, and among these candidate paths, a *globally optimal solution* is a shortest one (i.e., with minimal length).

2.1.2 Solving techniques

Solving a COP corresponds to searching for (near-)optimal solutions in the solution space defined by a specific instance. In a sense, all computational approaches for solving hard COPs can also be characterized as *search algorithms*. Basically, the simplest idea behind the search approach is to iteratively generate and evaluate solutions with respect to the constraints and the objective function. This raises the need for efficient search procedures, and over decades, many search algorithms were developed to tackle these problems. These algorithms can be classified as either *complete* or *incomplete*.

Complete or systematic search algorithms aim to exhaustively explore the entire search space, in the worst case, to determine an optimal solution. They guarantee that, for every finite size problem instance, either an optimal solution is found in bounded time, or, if no solution exists, this fact is proved with certainty (Papadimitriou & Steiglitz (1998); Wolsey & Nemhauser (1999)). This property is called the *completeness* of the algorithm. Examples of complete methods include brute-force search, backtracking search (Golomb & Baumert (1965)), branch-and-bound (Lawler & Wood (1966)), dynamic programming (Howard (1960); Bellman (1966)), and so forth. However, the search space often contains a huge number of solutions and typically grows at least exponentially with the size of the instance. In such cases, even the most powerful computer cannot enumerate all solutions within a reasonable amount of time. In order to push back the combinatorial explosion, the complete search algorithms often employ pruning techniques and/or ordering heuristics (the order in which variables or values are considered) to reduce the search space. Yet, for NP-hard combinatorial optimization problems, no polynomial time algorithm exists, assuming that $\mathcal{P} \neq \mathcal{NP}$ (Garey & Johnson (1979)). Hence, these complete methods may require an exponential computation time in the worst-case scenario.

Incomplete algorithms, on the other hand, focus on finding good-quality solutions by exploring only some parts of the search space. Although these methods do not guarantee to provide an optimal solution, they can find

high-quality solutions in a significantly reduced amount of time. Moreover, they usually can be stopped at any point during their execution, which is called the *anytime* property of search algorithms. Such incomplete methods include *heuristic search*, where a solution is progressively built based on efficient heuristics; *local search*, where various neighborhoods help improve the current solution; and *metaheuristics* like genetic algorithms (Sampson (1976)), tabu search (Glover (1989)), or iterated local search (Lourenço et al. (2001)), to name just a few. In general, the fundamental differences between incomplete methods rely on how they generate and evaluate solutions, which has a significant impact on practical performance. Hence, selecting appropriate techniques and optimizing their parameters is crucial for achieving satisfactory results.

2.2 Incomplete search algorithms

In the following, we summarize fundamental principles about incomplete search techniques for solving COPs before detailing advanced approaches hybridizing incomplete and complete techniques.

2.2.1 Constructive search

Constructive search methods are also called heuristic search methods or construction heuristics. The key principle is to generate *complete* solutions by iteratively extending *partial* solutions. At each iteration, a solution component is considered to be added to the partial solution based on predefined rules or heuristics. The process is repeated until a complete solution is found or a stopping condition is reached. A well-known constructive search method is based on greedy heuristics, where the choice that appears to be optimal is iteratively selected at each stage. For example, a famous greedy strategy for the Traveling Salesman Problem (TSP) is the Nearest Neighbor heuristic: at each construction step, visit the nearest unvisited node (Rosenkrantz et al. (1977)). This heuristic can yield a good-quality solution within a reasonable number of steps.

Constructive methods are often very fast due to their single-pass nature, meaning that there is no backtracking or revisiting of previous decisions. Nonetheless, during the construction phase, dead-end situations may occur when the current partial solution cannot be extended further due to the

violation of one or more constraints. Also, constructive methods often lead to suboptimal solutions. Precisely, the quality of the solutions produced by construction heuristics may heavily depend on the order in which solution components are added, and these heuristics are usually imperfect. That is why constructive search algorithms are often used to generate initial solutions for other advanced search techniques, such as local search and metaheuristics.

Interestingly, there is a remarkable connection between constructive search methods that build solutions incrementally and complete tree-based search methods that explore the search tree systematically. Indeed, many complete search algorithms use a constructive search method as a basis and add some forms of backtracking (Golomb & Baumert (1965)). The latter means undoing the last decisions made, going back to the previous state, and trying a different option when reaching a dead-end situation. For constraint satisfaction problems, this happens when the current solution violates some constraints. By using backtracking, the algorithm can explore all the partial solutions until it either finds a complete one or proves that none exists. For optimization problems, the algorithm continues exploring the whole search space to find the best solution.

2.2.2 Local search and perturbative search

In the literature, the terms "local search" and "perturbative search" refer to methods that move from one solution to another one in the search space by perturbing one or more solution components. Precisely, local search starts with an initial solution and iteratively moves to neighboring solutions with the hope of improving the objective function value. The search process is repeated until a stopping condition is reached. The concept of neighborhood, which is a key feature of local search algorithms, can be formally defined as follows.

Definition 2.4. *A* ***neighborhood function*** *is a mapping* $\mathcal{N} : \mathcal{S} \rightarrow 2^{\mathcal{S}}$ *that defines, for each solution* $s \in \mathcal{S}$*, a set of neighboring solutions* $\mathcal{N}(s) \subseteq \mathcal{S}$*.* $\mathcal{N}(s)$ *is also called the* ***neighborhood*** *of* s*. We say that* s' *is a* ***neighbor*** *or* ***neighboring solution*** *of* s *if* $s' \in \mathcal{N}(s)$*.*

Intuitively, the neighborhood function describes how to make small changes to a solution in order to get other solutions that are, in some sense, near to it. Then, the local search process is guided by an evaluation function. The latter is used for assessing or ranking neighbors of the current solution.

Definition 2.5. *An **evaluation function** is a mapping $g : \mathcal{S} \to \mathbb{R}$ that assigns a value to each solution $s \in \mathcal{S}$.*

In pure optimization problems, the value returned by the evaluation function directly corresponds to the quantity to be optimized. For problems containing both constraints and an objective function, an evaluation function g may combine both feasibility and optimality measures. Sometimes, different evaluation functions can also provide more effective guidance towards high-quality or optimal solutions e.g., Guided Local Search (Voudouris (1997)). In general, the evaluation function is problem-specific and its choice is, to some degree, dependent on the search space and underlying neighborhood structures. For the sake of simplicity, we assume in the following discussions that the objective function f of the problem serves as an evaluation function g, and the latter must be minimized.

The introduction of the neighborhood structure and the evaluation function allows us to define the concept of locally optimal solutions.

Definition 2.6. *A solution $s \in \mathcal{S}$ is **locally minimal** (or a **local minimum**) with respect to an evaluation function $g : \mathcal{S} \to \mathbb{R}$ and a neighborhood structure $\mathcal{N} : \mathcal{S} \to 2^{\mathcal{S}}$ if and only if*

$$\forall s' \in \mathcal{N}(s) : g(s) \leq g(s')$$

***Local maxima** can be defined analogously.*

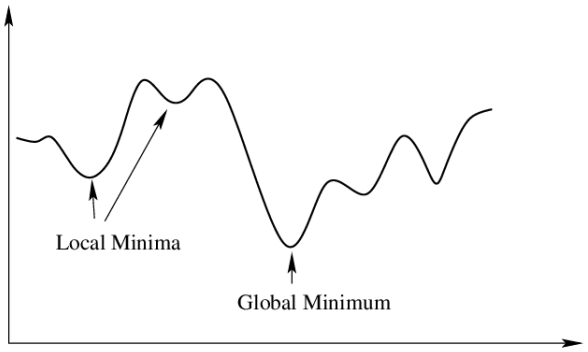


Figure 2.1 – Local vs. global minimum

Said differently, a solution is locally optimal if it does not have an improving neighbor. However, it is important to note that, under this definition, the

local optimality depends on the evaluation function g and the neighborhood structure $\mathcal{N}$. This means that a solution s that is a local optimum with respect to $(g_1, \mathcal{N}_1)$ is not necessarily a local optimum with respect to another pair $(g_2, \mathcal{N}_2)$.

A problem instance together with the evaluation function and the neighborhood structure define the topology of a so-called *search landscape*. The latter can be visualized as a labeled graph in which the nodes are solutions labeled by their objective function value, and the arcs represent the neighborhood relations between these solutions. A single *move* in the search landscape corresponds to the application of a neighborhood operator on a solution s to produce a neighbor $s' \in \mathcal{N}(s)$. Then, a *search trajectory* corresponds to a finite sequence of solutions $(s_0, \ldots, s_k)$ where (s_i, s_{i+1}) is a move i.e. $s_{i+1} \in \mathcal{N}(s_i)$. Intuitively, any search trajectory can be seen as a walk in the neighborhood graph or search landscape.

Frequently, the term *fitness landscape* is used to refer to the same concept, which was initially introduced in the context of evolutionary theory (Wright (1932)), and then adopted in the study of the factors underlying the behavior of evolutionary algorithms (Jones et al. (1995)) and metaheuristics in general (Fonlupt et al. (1999)).

Iterative improvement The basic local search is usually called an iterative improvement local search. This strategy is also known as iterative descent or hill-climbing. Its high-level description is presented in Algorithm 2.1. Basically, the algorithm starts from an initial solution $s \in \mathcal{S}$, which can be randomly chosen in the search space or generated by a constructive search method (Line 1). Then, it tries to improve the current solution by iteratively performing moves from one solution to a better neighboring solution w.r.t. the evaluation function g (Line 4). Such a move is also called an *improving move*. The search process stops once it finds a local minimum i.e. once no improving move is found (Line 6). The best solution found so far (called the *incumbent* solution) is memorized and returned upon termination of the algorithm. It is important to note that, in Algorithm 2.1, we omit the stopping conditions, e.g., conditions related to the maximum CPU time available.

Local search may seem like a straightforward approach to design and implement, yet it often requires considerable effort and a deep understanding of the structure of the problem in order to obtain an efficient solver. In essence, the effectiveness of a local search algorithm strongly relies on the

Algorithm 2.1: BasicLocalSearch

```
1 s ← generateInitialSolution()
2 s* ← s
3 while true do
4 |   s ← selectNeighbor(N(s))
5 |   if isFeasible(s) ∧ g(s) < g(s*) then s* ← s
6 |   else return s*
```

evaluation of the neighborhood and the *selection* of the next neighbor (Line 4).

How to evaluate a neighborhood? The choice of an appropriate neighborhood relation and the evaluation function is one of the most crucial factors affecting the performance of local search methods. In general, this choice needs to be made in a problem-specific way. Besides, evaluating the neighborhood of a solution may be computationally expensive, especially if the neighborhood size is large or the evaluation function is complex. Therefore, the value of the evaluation function should be maintained using incremental updates after each search step instead of recomputing it from scratch in order to improve the computational efficiency.

Constraint-based Local Search (CBLS) (Hentenryck & Michel (2005)) is a typical example that addresses these genericity and incrementality issues in designing effective local search. Its architecture consists of a modeling part and a search part. In terms of modeling, CBLS uses specific concepts, called *invariants* and *differentiable objects*, to model variables, constraints, and objective functions. The core point of this framework is the incremental update or maintenance of these objects during the search process, especially by using a dependency graph between objects. At the search level, CBLS does not prescribe any specific heuristics for each problem. Rather, it supports various abstractions to simplify the implementation of different algorithmic variants as well as for different problems.

In another direction, the evaluation of the neighborhood often requires determining whether a neighbor solution satisfies all logical constraints. Infeasible neighbors can be either discarded or penalized by the evaluation function. The simplest option is to maintain feasibility at all times and explore only feasible solutions in the neighborhood. However, the regions

containing the set of feasible solutions may be too narrow or disconnected. In some problems, it is even hard to find the first feasible solution that satisfies all the constraints. In such cases, it can be preferable to relax some of the constraints to explore a larger search space. To satisfy both the feasibility and optimality requirements, a possible approach is to add penalties for constraint violations into the objective functions. Technically, it is not easy to design an appropriate penalty scheme, which depends on the problem as well as the desired trade-off between feasibility and optimality. If the penalty is too large, it may discourage the exploration of the infeasible regions from the very beginning of the process. On the other hand, a low penalty will be negligible with respect to the objective function, hence, the search algorithm will spend a lot of time exploring the infeasible regions.

How to select the next neighbor? There are different strategies to choose the next neighbor using only local information on the quality of the neighbors. Two common deterministic strategies are:

- *best improvement*: this strategy consists in evaluating all neighboring solutions and selecting a neighbor that provides the best improvement in the evaluation function value;
- *first improvement*: the algorithm examines neighboring solutions one by one and simply selects the first one that improves the evaluation function value.

Example 2.1. *A well-known example of a local search algorithm is the min-conflict heuristic (Minton et al. (1992)) used for solving Constraint Satisfaction Problems (CSPs). Here, constraints may be violated during the search process, and that violation is measured and treated as an evaluation function. During the local search procedure, given a complete assignment, the values of some variables are changed to move from the current solution to a neighboring solution. The key idea is to always make a decision that minimizes the total number of violations at each step (i.e. best-improvement strategy). Min-conflict and related techniques are often used to solve decision problems, but optimization problems can also be solved by considering a series of decision problems and iteratively tightening an upper bound on the objective function.*

One limitation of the best improvement approach is the necessity of scanning the whole neighborhood to find the next move, which can be very time-consuming. On the other hand, the first improvement approach is faster

but may miss the best neighbor. In both cases, the algorithm eventually gets stuck at local optima. To address this issue, the simplest idea is to introduce randomness in the search process. This allows the algorithm to explore a wider range of solutions and avoid premature convergence, thus potentially escaping local minima. Examples of incorporating randomness in neighborhood selection include:

- Random Walk (Selman et al. (1993)), where the next neighbor is chosen randomly with a small probability (controlled by a parameter). This random selection allows the algorithm to occasionally explore solutions that would not be reached based on the current search trajectory. By doing so, the algorithm may have a chance to discover unexpected promising regions of the search space;

- Roulette-Wheel Selection (Holland (1975)), that uses a probability distribution to choose a neighbor based on its quality. This approach gives higher-quality solutions a greater chance of being selected, while still allowing for some exploration of other solutions.

2.2.3 Metaheuristics

Local search can provide fairly good-quality solutions very quickly. However, it can frequently visit the same locations within the search space and get stuck at local optima. This is why other techniques were introduced for avoiding or escaping from local minima, and directing the search towards globally optimal or near-optimal solutions. Such techniques are also called *metaheuristics*, which is a term commonly used in the Artificial Intelligence (AI) and Operations Research (OR) communities (Blum & Roli (2003); Glover & Kochenberger (2006)). Typical representatives include tabu search, simulated annealing, evolutionary computation, ant colony optimization, iterated local search, and so on. In principle, metaheuristics can be characterized as high-level strategies or generic search frameworks aiming at robustly and effectively exploring a search space.

Once again, the quality of the solutions obtained can be very sensitive to the initial solution, neighborhood function, underlying evaluation function as well as search strategies. Based on this remark, several approaches can be listed (Humeau et al. (2013)).

- *Iterating from different initial solutions:* The idea is simply to reinitialize the search process by restarting from a different point in the search space whenever a local minimum is encountered. This simple strategy is employed in multi-start local search and iterated local search (Lourenço et al. (2001)), as well as in the greedy randomized adaptive search procedure (Feo & Resende (1995)).

- *Accepting non-improving moves:* One can relax the improvement criterion and allow moves that degrade the current solution. This strategy is applied in simulated annealing (Kirkpatrick et al. (1983)) and tabu search (Glover (1989)).

- *Changing the neighborhood landscape:* One can change the neighborhood size or structure during the search process as in variable neighborhood search (Hansen & Mladenović (1999)), or change the objective function or the constraints as in guided local search (Voudouris (1997)).

- *Using learning techniques:* One can tune the heuristics based on specific knowledge acquired during search, as in ant colony optimization (Dorigo et al. (2006)).

Globally, *intensification* and *diversification* are two key concepts in metaheuristic approaches. Intensification refers to the process of exploring the neighborhood of a current solution, while diversification refers to the process of escaping from local optima and exploring new regions of the search space. Finding a balance between intensification and diversification is crucial for achieving a good performance. On one side, the search process should be able to quickly identify regions in the search space containing high-quality solutions. On the other side, it must not waste too much time in regions of the search space that are either already explored or do not provide high-quality solutions.

This section is not intended to provide an exhaustive description of metaheuristics but rather to review some of the most popular techniques available in the literature. Several metaheuristics described here will be recalled later in the next chapters, such as iterated local search, greedy randomized adaptive search procedure, tabu search, and path relinking.

Simulated Annealing (SA) is a metaheuristic based on an analogy with the annealing process used in metallurgy, where a metal is heated to a high

temperature and then slowly cooled down to reach its low-energy state. SA was originally presented in statistical mechanics (Metropolis et al. (1953)) and then applied to search algorithms for combinatorial optimization problems (Kirkpatrick et al. (1983)).

The basic pseudocode of SA is provided in Algorithm 2.2. In principle, SA works similarly to a classic iterative improvement algorithm, except that it selects a random move at each step and occasionally allows degrading moves by using a parameter T called the *temperature*. This parameter is used to control the probability of accepting worse solutions as the search progresses (Line 7).

Algorithm 2.2: SIMULATED ANNEALING

```
1  s ← generateInitialSolution()
2  s* ← s
3  T ← setInitTemperature()
4  while termination conditions not met do
5  |   s' ← selectRandomNeighbor(N(s))
6  |   if g(s') ≤ g(s) then s ← s'
7  |   else s ← s' with probability p(s' | T, s) = e^(−(g(s')−g(s))/T)
8  |   T ← updateTemperature(T)
9  |   if g(s) < g(s*) then s* ← s
10 return s*
```

Two critical points in simulated annealing consist in setting the initial temperature (Line 3) and the *cooling schedule* that specifies how to adjust the temperature over time (Line 8). Basically, the algorithm starts with a high temperature and then slowly decreases the temperature over time. A high temperature allows the algorithm to make many random moves initially, leading to the exploration a broader region of the search space. As the temperature decreases, the algorithm becomes more selective and mainly focuses on improving the objective function. A cooling schedule that is too fast may trap the algorithm in a local minimum, while a slow cooling schedule may waste computational resources.

Tabu Search (TS) is another popular metaheuristic that makes use of a memory for both escaping from local minima and guiding the search pro-

cess (Glover (1989)). As illustrated in Algorithm 2.3, the simplest version of TS uses a short-term memory, called a *tabu list*, to prevent the search from returning to recently visited solutions. This memory can be implemented by explicitly storing either previously visited solutions, or only solution features, or recent local moves. At each iteration of TS, this memory is updated. A parameter called *tabu tenure* determines the duration for which these restrictions apply (Line 7). Basically, using a tabu mechanism has the same effect as dynamically restricting the neighborhood of the current solution.

However, storing only local moves or features of solutions may exclude high-quality solutions from the set of neighbors. To address this problem, TS additionally employs a so-called *aspiration criterion*, which can override the tabu status of moves that lead to an improvement in the best solution found so far (Line 5).

Algorithm 2.3: TABU SEARCH

```
1  s ← generateInitialSolution()
2  s* ← s
3  τ ← initializeTabuList()
4  while termination conditions not met do
5  |  N_a(s) ← {s' ∈ N(s) | s' does not violate a tabu condition, or it
   |    satisfies the aspiration criterion, i.e. g(s') < g(s*)}
6  |  s' ← selectNeighbor(N_a(s))
7  |  τ ← updateTabuList(τ, s, s', tabuTenure)
8  |  s ← s'
9  |  if g(s) < g(s*) then s* ← s
10 return s*
```

The performance of TS strongly depends on the choice of the tabu tenure parameter. If the tabu tenure is too small, the search may follow cycles in the search space, while a large tabu tenure may be too restrictive and may miss high-quality solutions. The tabu tenure can also be dynamically adapted during the search, leading to more robust algorithms (Glover (1990)). Overall, determining a good choice for the tabu tenure typically requires empirical experimentation and fine-tuning for different problems. Besides, the efficiency of TS can be further increased by using techniques exploiting intermediate and long-term memory to achieve a higher diversification of the search process (Glover & Laguna (1998)).

Iterated Local Search (ILS) essentially works like a multi-start local search but focuses the search on the solutions that are returned by an embedded local search, instead of using repeated random trials (Stützle (1999); Lourenço et al. (2001)). For instance, the starting solution for the next iteration can be created by perturbing the local optimum returned at the previous iteration. The overall sketch of ILS is presented in Algorithm 2.4.

Algorithm 2.4: ITERATED LOCAL SEARCH

```
1 s ← generateInitialSolution()
2 s ← localSearch(s)
3 s* ← s
4 while termination conditions not met do
5 |   s' ← perturbation(s)
6 |   s' ← localSearch(s')
7 |   if g(s') < g(s*) then s* ← s'
8 |   s ← applyAcceptanceCriterion(s, s')
9 return s*
```

Three core components of an ILS algorithm include a perturbation phase (Line 5), a local search phase (Line 6), and an acceptance criterion (Line 8). The strength of a perturbation refers to the number of solution components that are modified. This modification is usually done in a non-deterministic way in order to diversify search. Importantly, a weak perturbation may lead the search back to the previous search region, while a strong perturbation makes the search resemble a random multi-start strategy. Besides, there is no restriction on the embedded local search procedure within the ILS framework. Obviously, more effective local search methods potentially lead to a better performance. Further details and references to applications of ILS can be found in the survey of Lourenço et al. (2003).

Greedy Randomized Adaptive Search Procedure (GRASP) is a multi-start method that combines constructive heuristics and local search (Feo & Resende (1995); Pitsoulis & Resende (2002)). Its pseudocode is given in Algorithm 2.5. At each iteration, a solution (not necessarily feasible) is constructed in a greedy randomized way (Line 3). A repairing procedure can then be invoked to restore solution feasibility when needed (Line 4). Subsequently, this feasible solution is improved by using an embedded local

search algorithm until a local optimum is found (Line 5). The best overall solution is registered (Line 6) and returned once a stopping condition is met (Line 7).

Algorithm 2.5: GRASP

```
1 s* ← nil
2 while termination conditions not met do
3 |   s ← greedyRandomizedContruction()
4 |   if ¬isFeasible(s) then s ← Repair(s)
5 |   s ← localSearch(s)
6 |   if g(s) < g(s*) then s* ← s
7 return s*
```

In contrast to standard construction heuristics, the construction phase in GRASP (Line 3) does not necessarily add the best solution component (i.e. with the maximal heuristic value) at each construction step. Instead, it randomly selects a move from a collection of highly ranked solution components, called *restricted candidate list*. Intuitively, GRASP can be seen as a repetitive sampling technique, where each iteration outputs a sample solution from an unknown distribution (Resende & Ribeiro (2010)).

It is important to note that the basic version of GRASP does not make use of the search history for the construction phase. Due to its simplicity, GRASP is often able to produce good solutions in a very short amount of time. It can be even faster with parallel implementations, where only a single global variable is required to store the best solution found over all processors (Ribeiro et al. (2002a)). A detailed discussion as well as many applications of GRASP can be found in the surveys of Ribeiro et al. (2002b); Resende & Ribeiro (2010).

Variable Neighborhood Search (VNS) mainly relies on the adaptation of the local search procedure to escape from local optima. In principle, VNS successively explores a set of neighborhoods instead of a unique neighborhood (Hansen & Mladenović (1999, 2001)). The general idea of VNS is based on the fact that a local optimum is defined relatively to a neighborhood relation. Indeed, a local optimum with respect to neighborhood $\mathcal{N}_1$ is not necessarily a local optimum with respect to another neighborhood $\mathcal{N}_2$. Thus,

VNS allows the neighborhood structure to be dynamically changed during search to escape from local optima.

An outline of basic VNS is shown in Algorithm 2.6. Let $\mathcal{N}_1, \ldots, \mathcal{N}_{k_{max}}$ be a set of predefined neighborhood structures, which are usually ordered in an increasing neighborhood size. At each iteration, VNS selects a kth-neighbor $s' \in \mathcal{N}_k(s)$ of the current solution s (Line 5). If an improvement is obtained, the algorithm comes back to neighborhood $\mathcal{N}_1$ (Line 7). Otherwise, it switches to the next neighborhood (Line 9) and repeats the above steps until some stopping condition is satisfied. In the literature, other extensions of this basic VNS alternate neighborhoods in different ways. For example, Skewed VNS (Brimberg et al. (2015)) is conceptually related to ILS. A description of other VNS variants and their applications are provided by Hansen et al. (2019).

Algorithm 2.6: VARIABLE NEIGHBORHOOD SEARCH

```
1  s ← generateInitialSolution()
2  s* ← s
3  k ← 1
4  while termination conditions not met & k ≤ k_max do
5  |   s' ← selectNeighbor(N_k(s))
6  |   if g(s') < g(s) then
7  |   |   s ← s', k ← 1
8  |   |   if g(s') < g(s*) then s* ← s'
9  |   else k ← k + 1
10 return s*
```

Guided Local Search (GLS) is based on the recognition that a local optimum with respect to the evaluation function g may not be locally optimal with respect to another evaluation function g'. As a result, the fundamental idea underlying GLS (Voudouris (1997); Voudouris & Tsang (1999)) involves dynamically modifying the evaluation function by penalizing solution features that are frequently present in visited solutions. These penalties are applied to increase the cost of solutions that contain these features and gradually reduce the attractiveness of the current local minimum over time. These adjustments guide the search towards unexplored regions of the search

space. In some cases, GLS may accept moves in infeasible regions and take into account the constraint violations in the evaluation function.

A generic template of GLS is depicted in Algorithm 2.7, whose embedded local search is generic. In essence, GLS works similarly to the iterative improvement procedure. Yet, a crucial step of GLS is the computation of the new evaluation function based on the current locally optimal solution (Line 7). The underlying penalizing scheme is not easy to appropriately define and is of course problem-dependent. A precise survey, guidelines, and representative applications of GLS are provided by Voudouris et al. (2010).

Algorithm 2.7: GUIDED LOCAL SEARCH

```
1 s ← generateInitialSolution()
2 s* ← s
3 g' ← g
4 while termination conditions not met do
5 |   s ← localSearch(s, g')
6 |   if isFeasible(s) ∧ g(s) < g(s*) then s* ← s
7 |   g' ← updateEvaluation(s, g')
8 return s*
```

Ant Colony Optimization (ACO) is a metaheuristic approach inspired by the behavior of ants for finding the shortest paths between their nest and a food source (Blum (2005); Dorigo et al. (2006)). The basic idea is to use a population of artificial ants that communicate with each other by depositing and following pheromone trails on a graph that represents the problem to be solved.

An outline of ACO is given in Algorithm 2.8. At each iteration, a population of virtual ants explores the search space by constructing solutions incrementally (Line 3). Precisely, for each ant, the next solution component is probabilistically chosen based on the concentration of pheromone trails and heuristic information. Pheromone trails act as the collective memory of the ants and guide their decisions.

Then, a local search procedure may be applied to improve the overall best solutions obtained so far (Line 4). Finally, the pheromone trails are dynamically updated to balance exploration and exploitation (Line 5). Basically, as ants discover better solutions, they deposit more pheromones along

the edges they traverse, thus reinforcing the paths leading to high-quality solutions. Also, the pheromone values decay over time to avoid stagnation and promote diversity. Other versions of ACO mostly differ in the way they update the pheromone values (Dorigo & Gambardella (1997); Stützle & Hoos (2000)).

Algorithm 2.8: Ant Colony Optimization

```
1 initializePheromones()
2 while termination conditions not met do
3 |   constructAntSolutions()
4 |   applyLocalSearch()
5 |   updatePheromones()
```

Evolutionary Computation (EC) is a metaphor for algorithms based on Darwinian principles of natural selection. These algorithms, known as evolutionary algorithms (EAs) (Bäck et al. (1997)), are inspired by the evolution process that happens in nature. Algorithm 2.9 shows a high-level sketch of an EA. In principle, EAs work with a group of individuals and use specific operators to create the next generation. These operators include recombination (or crossover) to combine individuals (Line 4), and modification (or mutation) to introduce some randomness (Line 5). Then, the selection of individuals for the next generation is performed based on their fitness (Line 7). Concretely, individuals having a higher fitness have a higher chance of being selected for the next generation.

Algorithm 2.9: Evolutionary Algorithm

```
1 P ← generateInitialPopulation()
2 evaluate(P)
3 while termination conditions not met do
4 |   P′ ← recombine(P)
5 |   P″ ← mutate(P′)
6 |   evaluate(P″)
7 |   P ← select(P″, P)
```

Various EAs were proposed over the years with three different strands:

evolutionary programming (EP) (Fogel (1962, 1998, 1999)), evolutionary strategies (ES) (Beyer & Schwefel (2002)), and genetic algorithm (GA) (Sampson (1976); Srinivas & Patnaik (1994)). Recently, other members of the EA family such as genetic programming (GP) (Koza et al. (1994)) and scatter search (SS) (Glover (1997b)) were developed and applied to COPs and optimization problems in general.

Path Relinking (PR) is also a member of the EA family in the sense that it tries to merge features of different solutions. PR was originally proposed as an intensification strategy that explores trajectories connecting elite solutions obtained by other metaheuristics (Glover (1997a); Glover & Laguna (1998); Glover et al. (2000)). Intuitively, PR favors selecting attributes of high-quality solutions in order to hopefully find better solutions.

A high-level pseudocode of PR is shown in Algorithm 2.10. Basically, PR works with a pair of solutions: s_1, called the initial solution, and s_2 the guiding solution. The procedure starts by computing Δ, the set of moves needed to reach s_2 from s_1, i.e. the difference between these two solutions (Line 2). Starting from the initial solution (Line 4), at each iteration, it examines all possible moves $m \in \Delta$ and applies the best one m^* (Line 6). In this context, m^* is the move that results in the lowest cost solution, i.e. that minimizes $g(s \oplus m)$ where $s \oplus m$ represents the solution obtained by applying move m to solution s. Then, the set of available moves is updated (Line 7), and the best move is applied (Line 8). If an improvement is found, the best solution is updated (Line 9). The above steps are repeated until there is no more candidate move (Line 5). Finally, a local search procedure can be applied to s^*, with the hope of improving the solution quality obtained so far (Line 10).

From this basic scheme, several alternatives were considered, including forward PR (Laguna & Marti (1999)), backward PR (Resende & Ribeiro (2003); Aiex et al. (2005)), mixed PR (Glover (1997a); Ribeiro & Rosseti (2007)), truncated PR (Andrade & Resende (2007); Resende et al. (2010)), greedy randomized adaptive PR (Binato et al. (2001)), or evolutionary PR (Resende & Werneck (2004)). These variants differ in how to incorporate attributes from the guiding solutions and balance the trade-offs between computation time and solution quality. In the literature, PR was shown to be effective when being hybridized with other metaheuristics, both as a diversification and as an intensification strategy. For example, a survey of

Algorithm 2.10: PATH RELINKING

```
1  s_1 ← a starting solution, s_2 ← a guiding solution
2  Δ ← computeDifference(s_1, s_2)
3  s* ← argmin{g(s) | s ∈ {s_1, s_2}}
4  s ← s_1
5  while Δ ≠ ∅ do
6  |   m* ← argmin{g(s ⊕ m) | m ∈ Δ}
7  |   Δ ← Δ \ {m*}
8  |   s ← s ⊕ m*
9  |   if g(s) < g(s*) then s* ← s
10 s* ← localSearch(s*)
11 return s*
```

combining GRASP with PR is provided by Resendel & Ribeiro (2005). Besides, PR can be used as an enhanced crossover operator when hybridizing with genetic algorithms. This idea was shown to be efficient on various problems such as the flow shop scheduling problem (Reeves & Yamada (1998)) or different variants of the routing problem with profits (Souffriau et al. (2010); Karbowska-Chilinska & Zabielski (2014); Campos et al. (2014)).

Different classifications of metaheuristics There exist various ways to classify and describe metaheuristics from different points of view.

For instance, metaheuristics can be classified based on the number of solutions used in the search process: a single solution or a population of solutions. Algorithms that work on a single solution are referred to as *trajectory-based* metaheuristics. They include metaheuristics that are based on local search, such as tabu search, iterated local search, and variable neighborhood search. *Population-based* metaheuristics, on the contrary, use a set of solutions or a probability distribution over the search space to guide the search process. Representative examples of this class are evolutionary computation and ant colony optimization.

Another classification is based on whether the metaheuristics use memory or not, so-called *memory-based* vs. *memory-less* methods. Memory-less algorithms, for example, perform a Markov process (Howard (1960)), as they only rely on the current state to decide the next action. Iterated local search or genetic algorithms are typical examples of this class. On the other hand,

memory-based metaheuristics can learn from their experience and adapt their search strategy accordingly. They use some form of memory to store and retrieve information about the search history and/or the search state. Examples of memory-based metaheuristics include tabu search and ant colony optimization.

Last, metaheuristics can be categorized into *nature-inspired* metaheuristics or *non-nature-inspired* metaheuristics. Nature-inspired algorithms, such as evolutionary computation and ant colony optimization, draw inspiration from natural processes. Meanwhile, non-nature-inspired ones such as tabu search and iterated local search do not have any connection to natural phenomena.

2.3 Hybrid approaches combining complete and incomplete techniques

Over the decades, so-called hybrid optimization techniques have become an active research area for addressing NP-hard optimization problems. In fact, when dealing with complex real-world scenarios, many incomplete algorithms do not rely on one stand-alone classical (meta)heuristic but rather exploit various algorithmic ideas from complete techniques such as tree search, dynamic programming (DP), mathematical programming (MP), constraint programming (CP), and SAT solving. Such hybridizations aim at combining the strengths of both complete and incomplete techniques for solving currently intractable problems. The key ideas are: (i) incomplete algorithms can exploit complete search techniques to increase efficiency; (ii) complete algorithms can follow some incomplete search mechanisms to improve scalability.

In the literature, there is a wide range of hybrid approaches combining incomplete methods with complete search techniques. Due to an excessive number of approaches, providing an exhaustive description is obviously not feasible here. In the following, we specifically focus on hybrid techniques designed to enhance the performance of classical local search methods and metaheuristics in solving hard combinatorial optimization problems. Among existing classifications (Talbi (2002); Blum & Roli (2003); Puchinger & Raidl (2005); Raidl (2006); Prestwich (2008)), we distinguish these techniques according to their control strategy, which can be either *collaborative* or *integrative*. The former means that different algorithms exchange information but

are not part of each other, whereas the latter corresponds to systems that exploit another technique as a subcomponent.

2.3.1 Collaborative combinations

In this case, complete and incomplete algorithms may be executed sequentially, in parallel, or intertwined. This section mainly focuses on sequential combinations where complete techniques are used either to initialize the solutions of incomplete search algorithms or to post-optimize of the solutions obtained by incomplete search algorithms.

Using complete techniques as pre-processing As illustrated in Figure 2.2, complete techniques can help define or reduce the search space for the incomplete search procedure by generating promising initial solutions or populations.

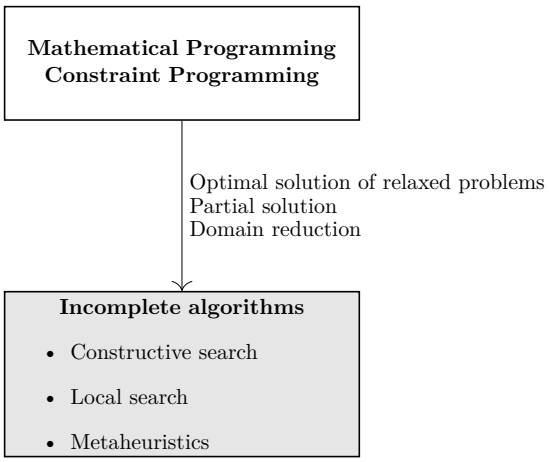


Figure 2.2 – Using complete techniques as a preprocessing step

A common way to find a promising starting point for a metaheuristic is to exploit the optimal solution of a relaxed version of the original problem. Frequently, a Linear Programming (LP) relaxation of the problem considered is exploited for this purpose. This can be done by relaxing the integrality constraint of integer variables. In other words, an LP relaxation allows the decision variables of the problem to take any real values while optimizing the objective function. The resulting relaxed problem is often easier than the

original problem and can be solved to optimality by efficient methods such as the simplex algorithm (Dantzig (1951)). However, an optimal solution to the LP relaxation is usually infeasible for the original problem.

To deal with this issue, a possible approach is the "dive-and-fix" strategy: (i) fix the integral part of the optimal relaxed solution, i.e. the decision variables having integer values in that solution and (ii) optimize the fractional or non-integral part. This approach is also referred to as *core methods* since the original problem is reduced and only its "hardcore" is further optimized. Vasquez et al. (2001) present such an example for solving the 0-1 multi-dimensional knapsack problem. First, they solve a series of LP relaxations to optimality, and then they use multiple tabu search (TS) to search for the values of the non-integral part. This approach is further improved by Vasquez & Vimont (2005) and extended for the multiknapsack problem by Puchinger et al. (2006).

Another possibility is to restore the solution feasibility by using a simple rounding procedure or more sophisticated repair strategies. This idea is applied in a genetic algorithm (GA) for both the multi-constrained knapsack problem (MKP) (Raidl (1998)) and the generalized assignment problem (Feltl & Raidl (2004)). In these works, the authors first solve the LP-relaxation of the problem and then use a randomized rounding procedure to create a population of integral solutions for GA. Additionally, they apply randomized repair and improvement operators to yield an even more promising initial population for the GA. Also for the MKP, Plateau et al. (2002) present a similar search scheme, but the relaxation of the original problem is solved by using an interior point method with early termination. Feasible integer solutions are generated by rounding and applying several specific heuristics. These solutions are then used as an initial population for a path-relinking/scatter search.

In a loose combination, CP can also serve as a preprocessing step for an incomplete solver (Wallace (1996)). In CP, the so-called *constraint propagation* mechanism can be used to efficiently reduce the domains of the variables by filtering values leading to dead-end situations. On this point, Lever (1996) employs CP to precompute all possible moves for a resource reallocation problem. In another direction, tree-based search methods can provide (partial) solutions that can be further refined or expanded by incomplete methods (Prestwich (2000)). As an example, Dell'Amico & Lodi (2005) describe some adaptations to perform a tabu search over partial solutions obtained at inner nodes within a CP tree search.

Using complete techniques as post-optimization In the opposite way, complete techniques can be used as a post-optimization module for improving the quality of solutions obtained by incomplete algorithms. A common approach is to merge elite components from a set of high-quality solutions (Figure 2.3). This is based on the fact that, in many optimization problems, high-quality solutions often share numerous features with the globally optimal solutions. Remarkably, solution merging can be done with not just two but possibly multiple input solutions.

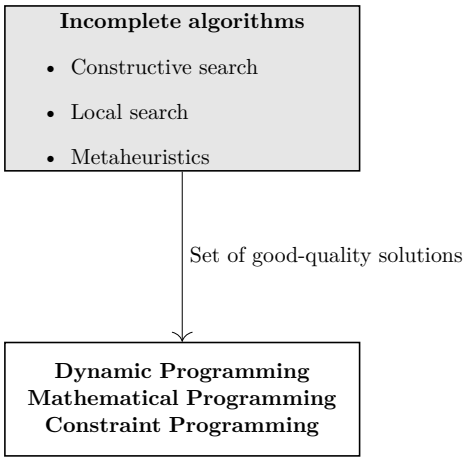


Figure 2.3 – Using complete techniques as post-optimization

For instance, Clements et al. (1997) present a hybrid column generation approach to solve a production line scheduling problem. First, they propose a specific heuristic, called Squeaky Wheel Optimization (SWO), to generate a feasible solution to the problem. This procedure is run several times to generate a set of diverse solutions. These solutions are then used as columns of a set partitioning formulation for the problem, which is solved to optimality. Their reports show that a classical tabu search is outperformed by the hybrid method proposed.

A similar approach for solving the TSP is studied by Applegate et al. (1998). The authors first derive a set of different tours by repeatedly running an ILS algorithm. Then, a strongly restricted graph is built based on the set of edges traversed in these solutions. The TSP on this restricted graph is then solved to optimality. Their empirical results indicate that the optimal TSP solution found on this restricted graph is typically superior to the best solution found by ILS alone.

In the same spirit, Klau et al. (2004) employ Integer Linear Programming (ILP) as a post-optimization procedure for the Prize-collecting Steiner tree problem. Here, they first run a memetic algorithm (MA), a population-based metaheuristic that combines different search strategies, to generate a set of solutions. Then, the ILP part consists in using an exact branch-and-cut algorithm to produce a final solution given the solutions obtained by MA.

In some sense, Hu & Lim (2014) describe a similar search scheme for the Team Orienteering Problem with Time Windows (TOPTW), but perform it in an iterative manner. Precisely, their algorithm, called I3CH, first runs both local search and simulated annealing to build a pool of good-quality solutions. Then, it applies a recombination procedure to produce a better solution by solving a set packing problem based on Mathematical Programming (MP). Besides using MP as a post-optimization module, this approach also exploits the solution recombined to determine a new starting point for the incomplete search process. An enhanced version of I3CH is proposed in Su & Nan (2023) for a variant of TOPTW, but using the same MP approach for the route recombination procedure.

2.3.2 Integrative combinations

In integrative combinations using both incomplete and complete techniques, one technique is embedded in another algorithm. In other words, there is a master algorithm and at least one integrated slave module.

Numerous hybrid systems integrate incomplete search principles (slave module) within complete algorithms (master module) to increase scalability. For instance, incorporating randomness or heuristics during a backtracking search can make it more flexible and guided, yet may sacrifice the completeness of the search (Harvey & Ginsberg (1995); Lynce & Marques-Silva (2007)). Within tree-based search procedures, (meta)heuristics can be used to guide the branching strategy (Crawford (1993); Benoist & Bourreau (2003)). Also, incomplete search algorithms can be beneficial for improving exact mathematical programming techniques, for instance, to compute better bounds for branch-and-bound (Woodruff (1999); Lever (2005)), to provide columns in column generation (Ribeiro Filho & Lorena (2000); Puchinger & Raidl (2004)), or to generate cuts in Benders decomposition (Poojari & Beasley (2009); Raidl et al. (2014)), to name just a few.

Here, our attention is shifted towards hybrid approaches where incomplete search algorithms serve as master components and complete techniques

are used to improve the search performance. Prominent examples include, but are not limited to, enhancements of the solution construction or local improvement procedures, special techniques for exploring large neighborhoods, or intelligent recombination of good solutions. Each of these points is detailed below.

Guiding the constructive search by look-ahead strategies In constructive search, one can exploit look-ahead enhancements commonly used in the complete search to quickly avoid inconsistent or suboptimal search spaces. Precisely, for the selection of the next solution component to be added at every step, MP or CP can be used to make better decisions by evaluating their long-term impacts rather than only using their direct contribution to the value of the evaluation function (Figure 2.4).

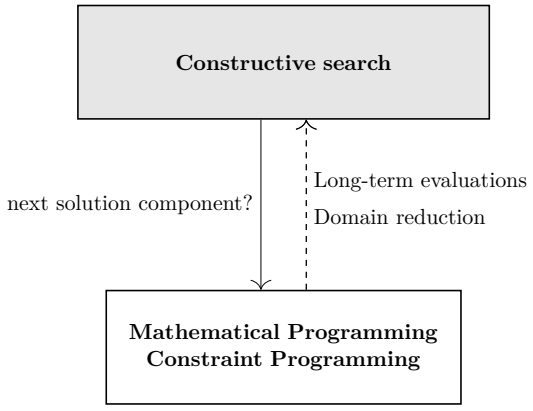


Figure 2.4 – Using complete techniques to guide constructive search

A common approach consists in solving problem relaxations, where constraints of the original problem are loosened or omitted. Information obtained from the relaxed problem can be exploited to determine the next promising moves or eliminate moves that possibly lead to suboptimal solutions. Following such an idea, Maniezzo (1999) presents the Approximate Nondeterministic Tree Search (ANTS) algorithm for the quadratic assignment problem (QAP). In this work, the basic search scheme is based on ACO but uses lower bounds as heuristic information to influence the probabilistic decisions of the ants during the solution construction phase. These bounds are computed by solving a relaxed linear assignment problem to estimate the completion cost of a partial solution. However, as a lower bound

must be computed at each construction step, the resolution of the relaxed problem must have a very low computational complexity. Maniezzo & Carbonaro (2000) and Maniezzo et al. (2001) also show the good performance of ANTS in other applications. Besides linear relaxations, bounds or dual values obtained through Lagrangian relaxations are also exploited in several works (Beasley (1990); Ribeiro et al. (2002c)).

In another direction, CP/SAT techniques can be employed to quickly detect inconsistent decisions during the construction process based on the set of constraints to be satisfied at the end. For CSPs, Prestwich (2002) presents an incomplete algorithm combined with pruning techniques used in systematic search. This algorithm called Incomplete Dynamic Backtracking (IDB), constructs a solution by: (i) randomly picking unassigned variables and assigning values to them; (ii) randomly unassigning k assigned variables when a dead-end is reached, i.e. when reaching a configuration where the current partial assignment cannot be further extended. This basic schema is then enhanced by using forward checking to quickly prune decisions leading to dead-ends. The approach also uses dynamic variable orderings based on domain sizes to guide the selection of an unassigned variable at each step. Another well-known example is the tight coupling between CP and ACO proposed by Meyer (2008) and Solnon (2010). Here, CP is used to avoid the construction of infeasible solutions during the construction phase of ACO, while pheromone learning in ACO is used to guide the variable/value selection in CP.

Enhancing constructive search by learning techniques Learning is a popular technique used in complete tree search algorithms to increase the efficiency in SAT/CSP solving. A prominent example is the *nogood* learning mechanism that was first introduced by De Kleer (1986) and Ginsberg & McAllester (1994). Technically, a nogood is a constraint that can be extracted during search. It defines a combination of variable assignments that cannot simultaneously be chosen due to the constraints of the problem. Intuitively, these nogoods record information about the search history and can be used to avoid repeating the same decisions in the future. Based on this concept, numerous works study the integration of learning modules within constructive search algorithms to analyze the failures encountered and direct the future search effort toward better solutions, as illustrated in Figure 2.5.

For example, Yokoo (1994) proposes the Weak Commitment Search algo-

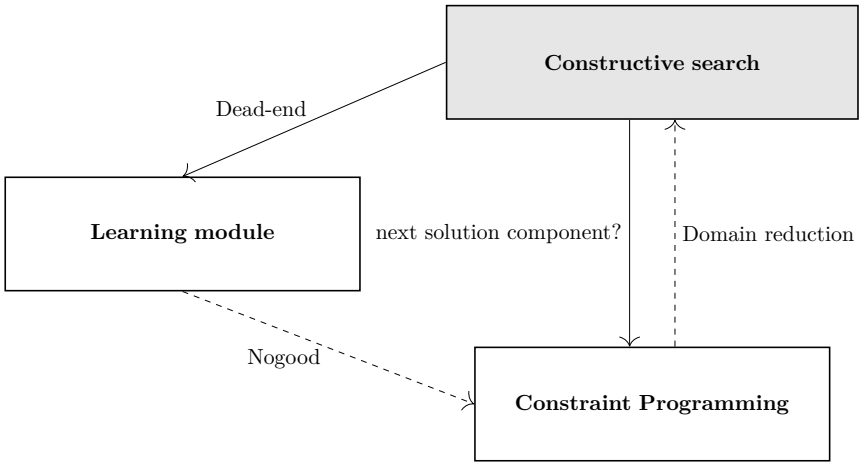


Figure 2.5 – Using learning techniques to enhance constructive search

rithm for solving CSPs. The algorithm greedily constructs consistent partial assignments, and when a dead-end is reached, it randomly restarts from scratch rather than performing chronological backtracking. Nogoods are also extracted and recorded to avoid the re-exploration of the same space. Similarly, Richards & Richards (2000) present a Learn-SAT algorithm that incrementally constructs a solution to a CSP. The construction phase is also guided by the nogood constraints recorded so far. Notably, the authors propose another procedure, called "learning-by-merging", that uses the SAT resolution mechanism on the clauses responsible for emptying a variable domain. The objective is to generalize nogoods learned at each dead-end in order to prune larger parts of the search space. Next, in the work of Jussien & Lhomme (2002), the authors design a decision-repair algorithm for CSPs and apply it to open-shop problems. They use filtering algorithms when completing a partial assignment. Once an inconsistency is found, they try to extract conflicts and use learned clauses to choose the variable to be unassigned. This *unassignment* can be seen as a local search move to another neighboring assignment. Also, Fang & Ruml (2004) present the Complete Local Search that uses constraint learning for SAT solving. In principle, they generate new clauses when reaching a local minimum. In particular, the clauses generated are used to dynamically change the objective function which guides the search.

Guiding local improvements Inference and learning techniques are effective for guiding not only constructive search (Figure 2.4) but also in the context of local search. As an example, De Backer et al. (1997) present a hybrid LS/CP approach for tackling the vehicle routing problem with side constraints. For the CP model, they use decision variables $next_i$ associated with each node i to indicate which node is visited after i. They apply local search to assign values to these variables and exploit a CP solver at each step to check the consistency of the current assignment as well as to filter the domains of the remaining $next_i$ variables (but not to *search* for solutions). Notably, this approach is quite generic and can be applied to different problem classes. Another example is the UnitWalk algorithm (Hirsch & Kojevnikov (2005)) that combines local search and unit propagation for SAT solving. In principle, the algorithm starts with a random initial assignment and iteratively modifies it by flipping the truth value of Boolean variables. The modification step using these 'flip' moves is enhanced with unit propagation for early detection of inconsistency. Such a unit propagation is a key technique used in complete SAT solving to eliminate branches in the search tree that do not contain any model of the given formula. This is done by propagating the logical consequences of atomic variable assignments to the clauses of the model until no further unit clause is found or an inconsistency is detected. In the latter case, the algorithm restarts using a slightly different variable ordering and possibly a modified value ordering. Empirical results show that UnitWalk achieves a better performance while performing fewer 'flip' moves compared to other contemporary incomplete SAT solvers.

Also, MP techniques can be exploited to guide the local improvement procedure toward better search regions. As an example, Puchinger & Raidl (2008) propose a variant of the VNS metaheuristic, called Relaxation Guided VNS, for the multiple knapsack problem. Their main search strategy uses different types of neighborhood structures. For each neighborhood $\mathcal{N}_i$ where $i \in \{1, \ldots, k\}$, they first solve a relaxed problem $R(\mathcal{N}_i)$ to obtain an objective value z_i. These values are then used to determine the most promising neighborhood to explore next, for example, a neighborhood whose index $i^* \in argmin\{z_i \mid i \in \{1, \ldots, k\}\}$ yields the best improvement.

Exactly exploring large neighborhoods Large neighborhoods containing more solutions offer better chances to reach high-quality local optima, but they require a considerable amount of time to scan and evaluate the whole

set of neighbors. Hence, the bottleneck of local search in large neighborhoods is the task of searching for a better (or even the best) neighboring solution. In this line, a possible approach is to model the problem of searching for the best neighbor as an optimization problem and solve it to optimality, instead of enumerating or randomly sampling the neighborhood. As illustrated in Figure 2.6, methods for solving these neighborhood search problems can range from dynamic programming to MP or CP solvers.

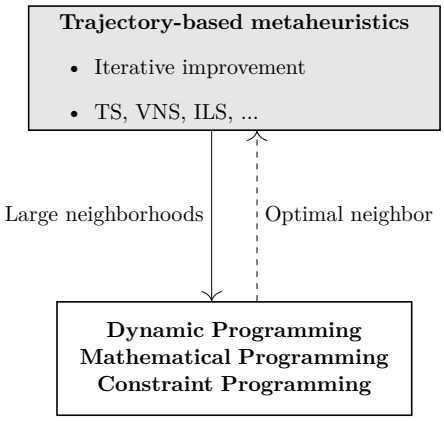


Figure 2.6 – Using complete techniques to explore large neighborhoods

Exactly exploring large neighborhoods based on CP For problems involving complex side constraints, the feasible solution space may be small but the neighborhood exploration may require a significant computational effort since checking solution feasibility is expensive. To overcome this difficulty, a well-known approach is to integrate CP techniques within incomplete algorithms, as CP is efficient in constraint handling. This approach is particularly effective when the neighborhood is tightly constrained. In other words, CP techniques are of interest when constraint propagation significantly reduces the search space, thus generating much fewer feasible neighbors than traditional incomplete algorithms. In the following, we summarize two main possibilities to explore neighborhoods using CP, depending on how the neighborhood search problem is formulated.

The first approach is introduced by Pesant & Gendreau (1996, 1999) for the TSP with time windows (TSPTW) and a scheduling problem. Subsequent works are also described by Backer et al. (2000) and Shaw et al. (2002)

for other problem benchmarks. In a generic manner, a high-level sketch of this approach is described in Algorithm 2.11. The latter is similar to the best-improvement strategy in local search except that the best neighbor is determined using complete techniques. The key point here is to define a CP problem $\mathcal{P}(s)$ modeling the full exploration of neighborhood $\mathcal{N}(s)$ given the current complete solution s (Line 4). Concretely, every feasible solution to this CP problem must correspond to a valid move in the neighborhood graph. Then, this CP problem can be solved to optimality (Line 5), and the optimal solution obtained is selected as the next neighbor in case of improvement (Line 6).

Algorithm 2.11: BEST NEIGHBORHOOD SEARCH

```
1 s ← generateInitialSolution()
2 s* ← s
3 while termination conditions not met do
4 |   P(s) ← an optimization problem defined on the neighborhood
  |     N(s)
5 |   s* ← solve(P(s))
6 |   if g(s) < g(s*) then s* ← s
7 return s*
```

The second possibility to explore neighborhoods using CP was first presented by Shaw (1998). The corresponding approach is commonly known as *Large Neighborhood Search* (LNS), whose neighborhood is implicitly defined by a *destroy* and *repair* method. Its generic template is given in Algorithm 2.12. In the destroy phase, the idea is to destruct a part of the current solution. This is done by keeping a fragment s_p of the current solution s (i.e., keeping the value assignment for a subset of variables) and erasing the values of the other variables in $r = s \setminus s_p$, called the *free* segment (Lines 4-5). Then, in the repair phase, the algorithm attempts to improve the destroyed solution while respecting the fixed values of variables in s_p. The solution is repaired by using either a construction heuristic or a general-purpose solver like a MIP or CP solver. For the latter case, the neighborhood search problem can be defined by searching r^* the optimal assignment for the free segment given that s_p (the fixed part) cannot be altered (Lines 6-7). An optimal repair method will be slower than the heuristic one, but may potentially produce high-quality solutions in fewer iterations. One of the first examples

of such a search scheme is the shuffle heuristic for the job-shop scheduling problem proposed by Applegate & Cook (1991). It consists in defining a partial solution by leaving the sequence fixed for a few machines and solving to optimality the scheduling subproblem for the remaining machines. In the literature, this technique is applied to various optimization problems, such as scheduling problems (Caseau & Laburthe (1996); Nuijten & Le Pape (1998)), quadratic assignment problems (Mautor & Michelon (1997)), and vehicle routing problems (Shaw (1998)).

Algorithm 2.12: LARGE NEIGHBORHOOD SEARCH

```
1  s ← generateInitialSolution()
2  s* ← s
3  while termination conditions not met do
4      r ← selectFreeSegment()
5      s_p ← s \ r   i.e. remove the values of the variables in r
6      P(r) ← a search problem defined over the variables r
7      r* ← solve(P(r))
8      s ← s_p ∪ r*   i.e. join two partial solutions to form a complete one
9      if g(s) < g(s*) then s* ← s
10 return s*
```

Exactly exploring large neighborhoods based on MP Similarly to CP, MP techniques are also used to effectively explore large neighborhoods as in both Algorithms 2.11 and 2.12. In fact, the neighborhood search problem can be modeled using mixed-integer programming (MIP), and a MIP solver can be applied to find an optimal neighboring solution. For example, Duarte et al. (2007) adapt the idea of Algorithm 2.11 by integrating MIP within an ILS metaheuristic for the referee assignment problem. Here, a MIP solver is used to replace a local search procedure based on *swap* and *replace* moves. Similarly, Prandtstetter & Raidl (2008) use Integer Linear Programming (ILP) techniques within a general VNS framework to examine large neighborhoods for the car sequencing problem. Also, Hu et al. (2008) describe a hybrid VNS to solve the generalized minimum spanning tree problem, where a MIP solver is used to optimize local parts within candidate solutions, using a procedure that is somewhat close to Algorithm 2.12.

Exactly exploring large neighborhoods based on DP Other works showcase the effectiveness of dynamic programming for exploring neighborhoods having an exponential size. One example is the *dynasearch* algorithm first proposed by Potts & van de Velde (1995). Consider, for instance, the application of dynasearch to the single-machine scheduling problem, whose objective is to find a processing order of n jobs on one machine so as to minimize a given cost function (e.g., total tardiness, makespan, etc.). In this case, a simple neighborhood is defined by the *swap* move, also called *two-exchange* move. If a solution is represented as a permutation of n jobs $\pi = (\pi_1, \ldots, \pi_n)$, then a neighboring solution can be obtained by swapping two objects (π_i, π_j) from π. Rather than applying a single swap move at each iteration, a dynasearch swap move corresponds to a series of pairwise independent swap moves. Thanks to the independence of each swap move, it is possible to define a recursive enumeration algorithm based on DP such that the resulting exploration is polynomial in time and space. Dynasearch is commonly applied to problems where solutions are represented as permutations, including the TSP and the linear ordering problem (Congram (2000)), the single machine total weighted tardiness problem (Congram (2000); Congram et al. (2002); Grosso et al. (2004)), and a time-dependent variant of it (Angel & Bampis (2005)). A general observation is that dynasearch is generally faster than a standard best-improvement descent algorithm and may return slightly better solutions.

Another popular example is the Very Large-Scale Neighborhood search (VLSN) introduced by Ahuja et al. (2000a, 2001). In these works, the authors consider cyclic and path exchange neighborhoods for tackling problems that can be formulated as kinds of set partitioning problems. Several examples of such problems include graph coloring, vehicle routing, scheduling, and so on. Basically, the primary objective is to seek a partition of items into disjoint subsets while minimizing a specific cost function. Then, a cyclic exchange move between k subsets $\{T_1, \ldots, T_k\}$ is formally represented as a cyclic permutation π of length k, where $\pi_i = j$ means that an element $j \in T_i$ moves to subset T_{i+1} (or T_1 if $i = k$). The authors then reformulate the search problem within this neighborhood as a problem of finding the shortest path in a directed graph. The latter is solved by using a dynamic programming algorithm, and the optimal solution obtained exactly corresponds to the best neighbor in the space of cyclic exchange moves. A detailed survey of VLSN is presented by Ahuja et al. (2002).

Furthermore, Gillard & Schaus (2022) present a novel method that ex-

ploits Decision Diagrams (DD) within LNS to solve combinatorial optimization problems having a dynamic programming formulation. The authors design a generic neighborhood exploration procedure by reusing the idea of restricted Decision Diagrams introduced by Bergman et al. (2016b). Intuitively, to avoid memory explosion, only a subset of feasible but promising solutions is kept and represented in such a restricted DD. Then, the optimal path (i.e. with the shortest length) of the resulting DD can be computed to get the new current solution in LNS. The effectiveness of this approach is shown on two sequencing problems: TSPTW and a production planning problem.

Merging solutions Efficiently merging solutions, which may be seen as exploring a large neighborhood defined by two or more solutions, can also be achieved using complete techniques. Several approaches following this idea have already been presented in Section 2.3.1, but in a sequential architecture. Here, we particularly consider approaches where complete techniques are exploited iteratively within a metaheuristic. For example, in population-based metaheuristics, complete techniques can be used to find the optimal offspring (Figure 2.7).

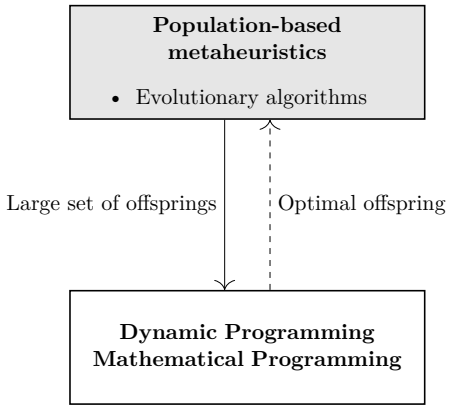


Figure 2.7 – Using complete techniques as search components in population-based metaheuristics (e.g., crossover, mutation)

As an illustration, Aggarwal et al. (1997) suggest such an optimized crossover operator to replace the classical crossover operator in EAs for the independent set problem. Ahuja et al. (2000b) present a matching-based optimized crossover for a genetic algorithm used for solving the quadratic

assignment problem. Similarly, Cotta & Troya (2003) apply B&B as an operator embedded in an EA for identifying the optimal offspring. The optimal merging of solutions can also be computed using dynamic programming, for example, for solving the k-cardinality tree problem as proposed by Blum (2006). From a theoretical point of view, Eremeev (2008) studies the computational complexity of producing the best possible offspring from two parental solutions having binary representations.

2.4 Conclusion

This chapter provided basic foundations for the subsequent chapters with a particular focus on several well-known hybrid complete/incomplete approaches available in the literature for tackling combinatorial optimization problems. Globally, designing and implementing hybrid techniques can be rather complicated. It requires deep knowledge about a broad spectrum of algorithmic techniques, programming, and data structures. For further details, we refer the reader to a precise overview of hybridization between incomplete algorithms and constraint programming (CP) provided by Focacci et al. (2003), or the survey on promising combinations of metaheuristics and integer linear programming (ILP) techniques provided by Puchinger & Raidl (2005).

Overall, incomplete techniques are often preferred in practice when dealing with large-size problems thanks to their effectiveness. Numerous advanced hybrid approaches are also proposed to further improve the performance, yet selecting appropriate techniques for each specific problem and optimizing their parameters are crucial for achieving satisfactory results. These challenges motivated our study on the enhancement of existing incomplete approaches for combinatorial optimization problems and particularly for routing problems with profits, which are our target applications. These latter problems will be presented in the next chapter.

CHAPTER 3

Orienteering Problem and its variations: A case study

Among numerous combinatorial optimization problems, we are specifically interested in routing problems with profits, especially those related to the problem of Earth Observation Satellite scheduling. Essentially, this problem can be viewed as a variant of a routing problem where a profit is associated with each customer. The primary objective here is to find an optimal schedule, which means to find an order to visit a subset of customers and maximize the profits acquired in the end.

For the sake of clarity, we first provide an overview of the routing problem with profits. Then, we dive into the orienteering problem, a specific problem class within this family that has received a lot of attention in the literature. Additionally, we discuss typical variations of the orienteering problem, highlighting their applicability in modeling diverse real-life scenarios in the context of satellite scheduling. For each problem variation, we also summarize various approaches proposed in the literature, in particular incomplete algorithms that can handle large-scale problem instances.

3.1 Routing problems with profits

The Traveling Salesman Problem (TSP) is a classic optimization problem whose objective is to find an optimal route to serve a set of customers. The Vehicle Routing Problem (VRP) is a generalization of the TSP using a fleet

of vehicles instead of just one salesperson. In practice, modeling and solving real-world VRPs is particularly challenging, as their applications in real contexts often involve complex constraints and objectives to reflect customer demands and preferences. Various extensions or variants of the VRP were proposed to make it more realistic and applicable. Among them, many practical problems can be modeled appropriately as *routing problems with profits*, whose objective is to maximize the total profit collected by visiting the customers rather than minimizing the total cost (distance or travel time). This variant can be applied to various real-life scenarios where each potential destination has a different value or reward associated with it.

Basically, routing problems with profits can be classified based on how the profit and the travel cost (distance or time) are modeled. As presented in Feillet et al. (2005), in the case of a single vehicle (namely the TSP with profits), three generic problems can be distinguished.

- The Profitable Tour Problem (PTP) (Dell'Amico et al. (1995)) combines both profits and travel costs in the objective function. Each component can be weighted differently to reflect its relative importance.
- The Prize-Collecting TSP (PCTSP) (Balas (1989)) aims to minimize the total travel cost while ensuring that at least a certain amount of profit is collected. In this case, the profit aspect is seen as a constraint.
- The Orienteering Problem (OP) (Golden et al. (1987)) focuses on maximizing the total collected profit while not exceeding a given travel cost. In this type of problem, the travel cost is seen as a constraint.

As shown in Figure 3.1, the OP and its variants appear to be the most studied problem among other routing problems with profits in the literature. At first glance, the OP is relatively similar to the regular routing problems (e.g., TSP or VRP) where a route visiting customers needs to be optimized. Yet, two crucial differences should be considered since they make solving an OP significantly different and more time-consuming.

- In an OP, *the selection of customers is optional*. In fact, each customer has a certain profit and requires a certain amount of time to be visited. Due to the limited time budget, only a subset of customers can be selected while the goal is to maximize the total profit. This selection is related to the well-known knapsack problem (KP) (Salkin & De Kluyver (1975)) but is more complex. This is because the time required to visit

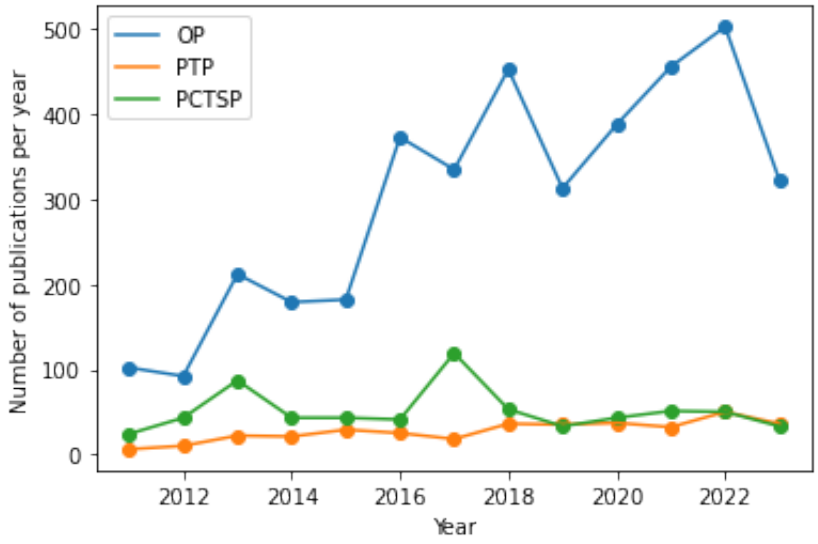


Figure 3.1 – Number of publications per year with keywords: "Profitable tour problem", "Prize-collecting TSP", "Orienteering Problem"

a customer depends on which other customers are selected and in which order.

- In an OP, *the objective is to maximize the total profit collected*. This implies that finding the shortest route in the OP is not an objective in itself. However, reducing travel times may help to visit an extra customer or replace a visited customer with another one having a higher profit. Overall, different solutions having the same profit but different travel costs are considered equally good.

From Figure 3.2, we observe that the orienteering problem has numerous applications across various domains. Representative examples include, but are not limited to: home fuel delivery (Golden et al. (1987)), telecommunication networks building (Thomadsen & Stidsen (2003)), tourist trip planning (Souffriau et al. (2008)), military application (Wang et al. (2008)), or satellite scheduling problems (Peng et al. (2019)). Several variants may require additional sets of specific constraints and/or different objective functions in order to model real-life scenarios, for example: budget limitations, uncertain profits, combinations of visits increasing or decreasing the sum of profits, traffic congestion, time-dependent preferences, and so on.

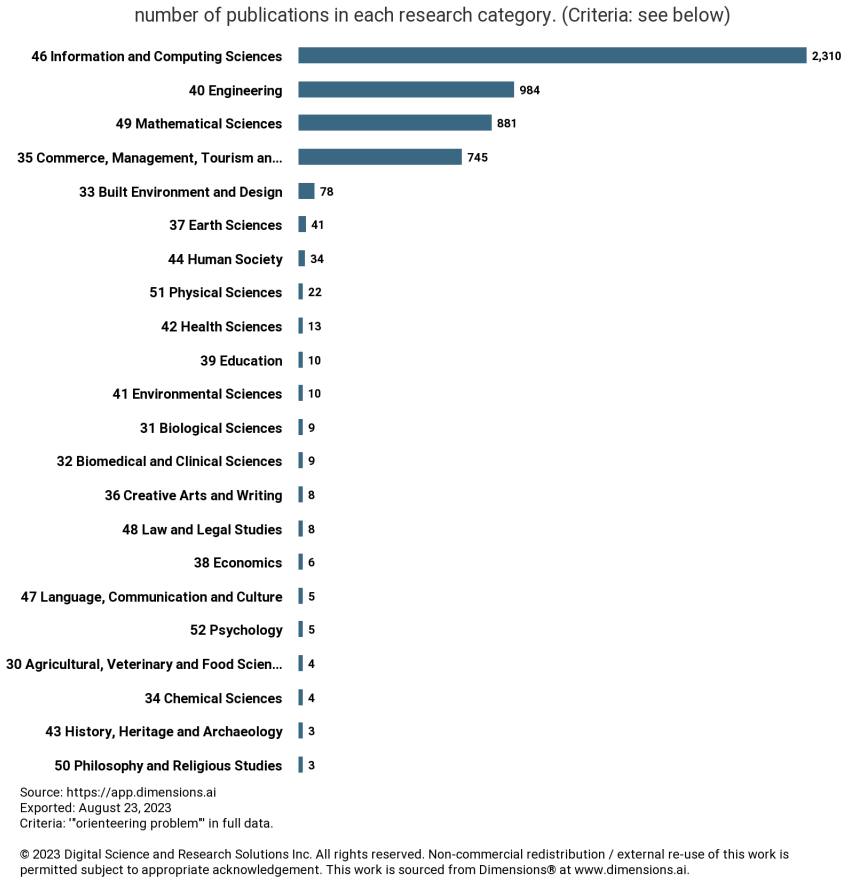


Figure 3.2 – The number of publications in each research category with the keyword “orienteering problem”

Technically, the orienteering problem is difficult to solve because it combines two NP-hard problems: the traveling salesman problem and the knapsack problem, which are both already difficult problems on their own. Moreover, the time-dependent aspects or side constraints such as time window constraints can make the problem even harder to solve. In such cases, the search space becomes very complex, and finding an optimal solution requires exploring many possible combinations of both selections and sequencing decisions. In the following, we focus on providing formal definitions as well as well-known solution approaches for the OP and several typical variations.

3.2 OP variants and solving approaches

3.2.1 Classical OP

Problem definition In the classical Orienteering Problem (OP), the objective is to find a route that maximizes the total collected profit while respecting the travel cost constraint.

The OP can be formally defined as follows. Consider $I = \{0, \ldots, N+1\}$ a set of nodes (customers) that can be visited, where 0 and $N+1$ are considered as the source and sink depots (or the start and end nodes). Each node $i \in I$ is associated with a non-negative reward (or profit) R_i (each depot has zero profit $R_0 = R_{N+1} = 0$). A non-negative travel cost (time or distance) between nodes i and j is denoted as $tt(i, j)$ for each pair of distinct nodes $i \in I, j \in I$. In general, travel costs are assumed to satisfy the triangular inequality: $\forall i, j, k \in I, tt(i, j) + tt(j, k) \geq tt(i, k)$. Intuitively, for any three distinct nodes $i, j, k \in I$, the direct route from i to j is never longer than any detour through an intermediate node k. Besides, a visit duration d_i can be considered for each node $i \in I$, however, we assume that these visit durations are already included in the travel costs. This assumption holds for all variants of OP considered in the following.

In an OP, each node can be visited at most once. A *solution* of the OP is a sequence $\sigma = [i_0, \ldots, i_{q+1}]$ that starts and ends at the two depots ($i_0 = 0$, $i_{q+1} = N+1$), and visits a subset of distinct nodes $S = \{i_1, \ldots, i_q\} \subseteq \{1, \ldots, N\}$. A solution is *feasible* if and only if the total travel cost does not exceed T_{max}, a predefined limited budget (e.g., total travel duration). Then, the goal is to find an *optimal* solution σ^*, which is feasible and maximizes the total reward collected (i.e. $\sum_{i \in \sigma} R_i$).

Solution approaches Since OP is proven as NP-hard in the work of Golden et al. (1987), no polynomial time algorithm has been designed to solve this problem to optimality, or is expected to be designed assuming that $\mathcal{P} \neq \mathcal{NP}$ (Garey & Johnson (1979)). Several exact algorithms were proposed by Laporte & Martello (1990); Gendreau et al. (1998a); Feillet et al. (2005) to solve the OP, yet they are very time-consuming when dealing with large-size instances in practical applications. In these cases, incomplete approaches (heuristics or metaheuristics) are usually preferred.

More specifically, various heuristics can be used to tackle the OP, such as

the stochastic and the deterministic heuristic (Tsiligirides (1984)), the center-of-gravity heuristic (Golden et al. (1987)), the four-phase heuristic (Ramesh & Brown (1991)), or the five-step heuristic (Chao et al. (1996a)). Gendreau et al. (1998b) give a few reasons why it is so difficult to design good heuristics for the OP. One of them is because the score of a node and the time to reach this node are independent. This makes it difficult to determine which node will be part of the optimal solution.

Furthermore, there exist different (hybrid) metaheuristics proposed to efficiently solve the OP. For example, Schilde et al. (2009) develop a solution approach for a bi-objective OP using a Pareto ACO and a multi-objective VNS, both hybridized with path relinking. Şevkli & Sevilgen (2010); Şevkli & Sevilgen (2010) introduce different variants of the Particle Swarm Optimization (PSO) algorithm. Other advanced metaheuristics include the Multi-Level VNS (Liang et al. (2013)), the GRASP-PR (Campos et al. (2014)) or the Memetic-GRASP (Marinakis et al. (2015)).

3.2.2 Team variant

Problem definition The extension of the OP to multiple vehicles is known as the Team Orienteering Problem (TOP) (Chao et al. (1996b)) or Multiple Tour Maximum Collection Problem (MTMCP) (Butt & Cavalier (1994)). The goal is to determine a set of M routes $\{\sigma_1, \ldots, \sigma_M\}$, given that:

(i) each node can be visited at most once,

(ii) every route σ_m must not exceed a limited travel cost (and must respect custom constraints if there exist some),

(iii) the objective is to maximize the total collected profit across the set of routes, i.e. maximize $\sum_{m=1}^{M} \sum_{i \in \sigma_m} R_i$.

It should be noted that the M routes can have different $T_{max,m}$ limited travel costs. For the sake of simplicity, we assume that T_{max} is the same for all routes.

Solution approaches By considering multiple vehicles, the number of decision variables and constraints increases significantly. For solving a TOP, only a few exact algorithms were developed, using column generation (Butt

& Ryan (1999)), branch-and-price (Boussier et al. (2007)), branch-and-cut-and-price (Poggi et al. (2010)), or branch-and-cut (Dang et al. (2013a)).

Due to the complexity of the problem, the research on TOPs mainly focuses on heuristic algorithms. Compared to OP, computational times for the TOP heuristics are significantly higher. This is because TOP heuristics should consider the transfers of nodes from one route to another, the grouping of nodes together, and the exploitation of the time budget in each route.

A first heuristic for the TOP is introduced by Chao et al. (1996b). This heuristic is relatively similar to a five-step heuristic defined for the OP by Chao et al. (1996a). Tang & Miller-Hooks (2005) develop a tabu search heuristic embedded in an Adaptive Memory Procedure that alternates between small and large neighborhoods. Archetti et al. (2007) present a slow and a fast VNS. They also introduce two variants of tabu search: one that explores only feasible search regions and another that explores unfeasible search regions. ACO is also used to solve TOP instances in the work of Ke et al. (2008) by applying different methods to construct feasible solutions. Next, a TOP can be solved using GLS (Vansteenwegen et al. (2009b)) and Skewed VNS (Vansteenwegen et al. (2009a)). The latter can obtain good TOP solutions in only a few seconds of computational time. Souffriau et al. (2010) propose a hybrid GRASP-PR algorithm, where PR is used to build a new hopefully improved solution. Another hybrid algorithm, called Memetic Algorithm (MA), is proposed by Bouly et al. (2010). This algorithm combines GA and some LS techniques. Other advanced metaheuristics for the TOP include the Discrete PSO (Muthuswamy & Lam (2011)), PSO-inspired algorithms (Dang et al. (2011, 2013b)), the Multi-Restart Simulated Annealing (Lin & Kernighan (1973)), the Genetic Algorithm (Ferreira et al. (2014)), the Pareto Mimic Algorithm (Ke et al. (2016)), and so forth.

3.2.3 Time windows

Problem definition The time window constraints arise in the context where the service at a particular node has to start within a predefined time window. This leads to two well-studied variants of the OP, namely the OP with Time Windows (OPTW) (Kantor & Rosenwein (1992)) and the TOP with Time Windows (TOPTW) (Labadie et al. (2012)).

It is also worth noting that some nodes may have *multiple time windows* (Tricoire et al. (2010); Souffriau et al. (2013); Lin & Vincent (2015)). A straightforward way to model this problem is that each node can be du-

plicated based on the number of time windows. Then a constraint must be added to make sure that no duplicate nodes are simultaneously visited.

Formally, each node $i \in I$ has a time window $[O_i, C_i]$, and a solution σ is *feasible* if and only if every node is visited within its time window. It is important to note that each time window constraint always restricts the start of the service, not the entire service. In other words, an early arrival to a particular node i leads to waiting times until the opening date O_i. Meanwhile, a late arrival (i.e., service starting after the closing date C_i) causes an infeasibility issue. However, the service is allowed to start before the closing date and finish after the closing date. Therefore, the service time should be modeled appropriately and a straightforward way to do that is to include the service times in the travel times.

Solution approaches Only a few exact algorithms are available to solve OPTW. For example, Righini et al. (2006) design a bi-directional dynamic programming algorithm to solve OPTW instances to optimality. Righini & Salani (2009) also present an enhanced version, called Decremental State Space Relaxation (DSSR), which helps reduce the number of states to be explored. Another approach is proposed by Duque et al. (2015) using a so-called pulse framework. Precisely, this algorithm is inspired by an exact method for shortest path problems with side constraints and incorporates problem-specific knowledge from the OPTW. This pulse framework empirically outperforms DSSR on highly constrained OPTW instances. Nevertheless, these approaches become quite ineffective when dealing with less constrained instances from Montemanni et al. (2011), even in the case of a single vehicle (OPTW). For the team variant (TOPTW), several works apply mathematical programming (Tae & Kim (2015); El-Hajj et al. (2015); Hapsari et al. (2018)) or constraint programming (Gedik et al. (2017)) to solve the problem to optimality, but until now, there are not many exact methods proposed for this problem variant.

Regarding incomplete methods, Kantor & Rosenwein (1992) describe a straightforward insertion heuristic to solve OPTW, where the node with the highest ratio "score over insertion time" is inserted into the current route at each step, while respecting the time window constraints. Mansini et al. (2006) develop a simple constructive heuristic and a VNS-based approach for the OPTW. Moreover, numerous (meta)heuristic techniques are developed for solving both OPTW and TOPTW. Vansteenwegen et al. (2009c) intro-

duce an ILS framework to effectively solve TOPTW within a short amount of time. Another hybridization between GRASP and evolutionary local search (ELS) is proposed by Labadie et al. (2011). Here, five simple constructive heuristics are used in GRASP for generating distinct initial solutions that are further improved by the ELS algorithm. Gambardella et al. (2012) propose an enhanced Ant Colony System (ACS) to overcome the drawbacks of the ACS they designed earlier (Montemanni et al. (2011)). Also, Lin & Vincent (2012) study the SA algorithms for TOPTW. Gavalas et al. (2013) focus on the tourist trip design problem (TTDP) as an application of TOPTW. They solve it with two clustered-based algorithms that extend ILS with cluster-based heuristics for the construction phase, where nodes are grouped into disjoint clusters based on geographical criteria. Hu & Lim (2014) propose the I3CH framework that is based on LS and SA to explore the solution space. Then, a Route Recombination procedure (using Mathematical Programming) is invoked to produce the best combination of paths explored so far. An enhanced version of I3CH is proposed by Su & Nan (2023) for solving a variant involving multiple deliverymen. Cura (2014) study an Artificial Bee Colony approach with SA as an acceptance criterion. Gunawan et al. (2015a,c) propose an ILS algorithm for solving (T)OPTWs and use multiple operators in the local search phase. A well-tuned algorithm called SAILS is presented by Gunawan et al. (2015b). In this work, using SA helps escape from a local optimum by accepting a worse solution with a small probability.

Regarding the *multiple time windows* variants, Tricoire et al. (2010) study a variant of OP, namely Multi-period OP with multiple TW (MuPOPTW) for a real industrial case. The multi-period aspect means that each customer node may have different time windows for each given day. To solve this problem, they propose a constructive heuristic and a VNS metaheuristic. In general, VNS provides good solutions although it requires more computation time. Souffriau et al. (2013) present a hybrid of GRASP and ILS, namely GRILS. This approach is used to tackle the Multi-Constraint Team Orienteering Problem with Multiple Time Windows (MC-TOP-MTW), a complex variant of the TOPTW. In this problem, a certain number of additional knapsack constraints are included in the problem. As there are no benchmark instances for the MC-TOP-MTW, the performance of the algorithm proposed is evaluated by using three related problems, the TOPTW, the Selective Vehicle Routing Problem with Time Windows (SVRPTW) (Boussier et al. (2007)) and the Multi-Constraint Team Orienteering Problem with Time Windows (MC-TOPTW) (Garcia et al. (2013)). Last, Lin & Vincent

(2015) propose two versions of SA with a restart strategy to solve the MC-TOP-MTW variant.

3.2.4 Time-dependent travel times

Problem definition In the OP, the travel time between two given nodes is assumed to be a static value. However, in many practical situations, the travel time actually depends on the network properties, such as rush hours, congestion levels, construction zones on certain links, and so on, which may affect the travel time between two nodes. The OP variant where the travel time between two nodes depends on the departure time at the first node is called the Time Dependent OP (TDOP) (Fomin & Lingas (2002); Verbeeck et al. (2014)). Another variant called Time Dependent OP with Time Windows (TDOPTW) (Verbeeck et al. (2013)) considers the time window constraints for the nodes. Furthermore, considering multiple vehicles or paths can lead to another variant, called the Time Dependent Team OP with Time Windows (TDTOPTW) (Garcia et al. (2013)).

Formally, to move from node $i \in \{0, 1, \ldots, N\}$ to node $j \in \{1, \ldots, N+1\}$, a minimum transition time $tt(i, j, \tau)$ is required, where τ is the departure time from node i. This formulation allows us to consider the cases where the transition time between two locations depends on the conditions (e.g., congestion or rush hours) that vary throughout the day. Transition function tt usually satisfies the FIFO property according to which the earlier a transition starts, the earlier it ends, *i.e.* $\tau + tt(i, j, \tau) \leq \tau' + tt(i, j, \tau')$ for $\tau \leq \tau'$. It may also satisfy the triangular inequality, *i.e.* $tt(i, j, \tau) + tt(j, k, \tau') \geq tt(i, k, \tau)$, where $\tau' = \tau + tt(i, j, \tau)$.

Solution approaches Verbeeck et al. (2014) propose an algorithm to solve TDOP based on the combination of an Ant Colony System (ACS) with a time-dependent local search procedure. Gunawan et al. (2014) study the real-life TDOP application where the goal is to provide automatic tour guidance to a large leisure facility. They use four different metaheuristics: a restart greedy heuristic, a restart Variable Neighborhood Descent, a basic version of ILS, and a modified ILS with adaptive perturbation size and probabilistically intensified restart.

Besides, Garcia et al. (2010) study an application of the TDOPTW, called the Personalized Electronic Tourist guide (PET). Here, the public transportation infrastructure that influences the travel time between two nodes

is incorporated. A hybrid algorithm combining two heuristics is used to solve the problem. The first heuristic focuses on the calculation of the average travel time between all pairs of nodes, while the second one implements ILS (Vansteenwegen et al. (2009c)). Similarly, the tourist trip design problem (TTDP) in a large urban area is formulated as a TDOPTW and then solved with GA in the work of Abbaspour & Samadzadegan (2011). Also, a fast ACO algorithm is proposed by Verbeeck et al. (2013) to solve the TDOPTW.

Furthermore, Garcia et al. (2013) adapt ILS to deal with a TD-TOPTW concerning a real case study for the city of San Sebastian. Also, Gavalas et al. (2014) extend a previous work (Gavalas et al. (2013)) to solve the TD-TOPTW using cluster-based algorithms.

3.2.5 Other variants

Over past years, numerous variants of OP were presented and discussed in the literature, including but not limited to:

- Generalized OP (Wang et al. (1996); Geem et al. (2005); Wang et al. (2008)) where the objective function is a more complicated (non-linear) function of the nodes visited.
- OP with compulsory vertices (Gendreau et al. (1998a)), where a subset of nodes has to be visited.
- OP with stochastic profits (Ilhan et al. (2008)), whose goal is to maximize, within a time limit, the probability of collecting more than a specified target level.
- Bi-objective OP (Schilde et al. (2009)), where one objective is to achieve more rewards and the other objective is to gain more free time.
- Capacitated TOP (Archetti et al. (2009)), where a non-negative demand is associated with each node and the total demand covered by each route cannot exceed a given capacity. In many real-life applications, the capacity of each vehicle is indeed an issue to take into account.
- Clustered OP (Angelelli et al. (2014)), where nodes are clustered in groups. A generalization is the Set OP (Archetti et al. (2018)), where a profit is associated with each cluster.

- OP with variable profits (Erdoğan & Laporte (2013)) or time-dependent profits (Yu et al. (2019); Peng et al. (2019); Khodadadian et al. (2022)), where profits are not simply static values.

3.3 Application to the aerospace domain

Over the last decades, variants of the OP have been widely used to model many practical problems in logistics or tourism. More recently, OP variants have also been used to address applications in the aerospace domain, in particular for planning activities of Earth Observation Satellites (EOSs).

Basically, an EOS (Bensana et al. (1999a)) is designed to collect images through its sensors and is widely used in many important fields, such as natural disaster monitoring or cartography. Precisely, the mission of observation satellites is to capture images over specific targets on the Earth's surface at the request of different users including governments, research institutes, and companies. In this context, each request generates a profit associated with each observation. This profit can be related to the quality and resolution of the images captured. Moreover, for non-geostationary satellites, the observation targets are not permanently visible due to the rotation of the satellite around the Earth. More specifically, for a candidate ground target i, the time frame during which the satellite overflies target i over a given orbit around the Earth corresponds to a time window for target i. Also, to observe a target, the instrument of the EOS must be pointed to that target, and between two consecutive observations, the EOS must perform a maneuver to change its orientation. Such a maneuver requires some transition time between the two observations. By analogy with the OPTW, a single satellite can be seen as a vehicle that tries to visit some customers (i.e., ground targets) within their predefined time windows. Due to temporal constraints and/or an excessive number of requests, it is often not possible to observe all the targets, and each target captured yields a distinct profit. In the end, the primary goal is to maximize the cumulative profit obtained at the end of the orbit while satisfying the operational constraints.

One step further, the Agile EOS (AEOS) (Lemaıtre et al. (2002)) is a new generation of EOS that can maneuver around its gravity center along the three axes (roll, pitch, and yaw), while moving on its orbit. Compared to a non-agile EOS maneuverable only on the roll axis, an AEOS offers more flexibility to perform the observation tasks. Indeed, as illustrated in Figure

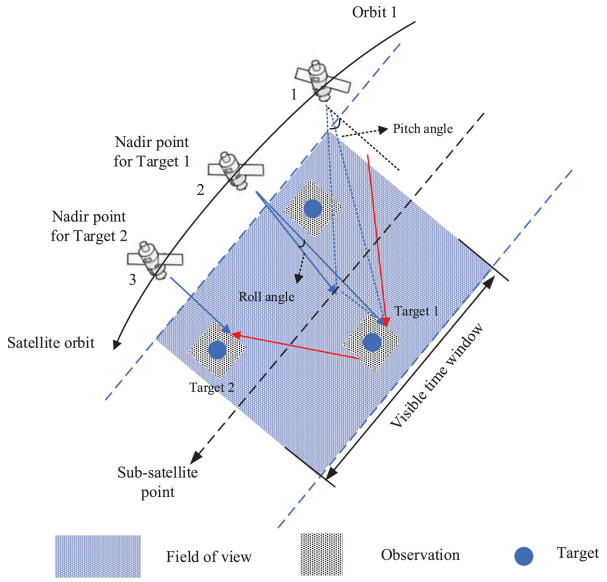


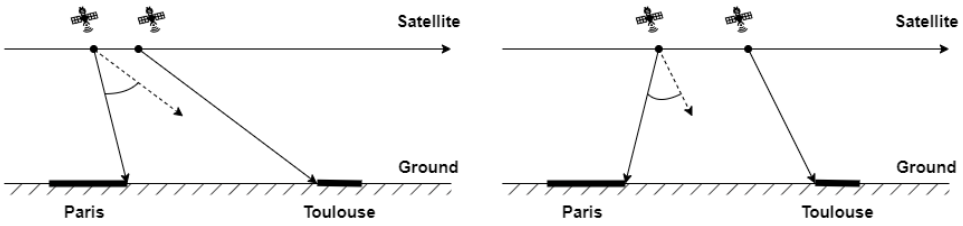
Figure 3.3 – An agile satellite captures images over a target at different start times on its orbit (source: Peng et al. (2019))

3.3, the mobility along the roll angle allows the satellite to point on the left or right, and the maneuverability along the pitch angle allows the satellite to observe targets by pointing forward or backward. As illustrated in Figure 3.4, the agility of AEOS also leads to crucial points in terms of routing problems.

- First, the minimum transition time required to adjust the viewing angle between consecutive observations is not constant. Rather, it strongly depends on the end time (or the start time) of the previous observation, thus being time-dependent.

- Second, when capturing images of targets, the viewing angle of the instrument also affects the quality of the images captured. For instance, the images captured over Toulouse in Figure 3.4a may have a worse quality than the one captured in Figure 3.4b. In other words, the profit gained also depends on the time at which the acquisition is performed.

As a result, the problem obtained for this scenario can be modeled as an OP with time-dependent transition times, time-dependent profits, and time windows (TDOP-TDPTW) (Peng et al. (2019, 2020)).



(a) (b)

Figure 3.4 – An illustration of two consecutive acquisitions over Paris and Toulouse at different observation start times; the transition time on the left is shorter than the transition time on the right

Besides, practical satellite scheduling problems involve other complex operational constraints concerning, for example, the image download process or the onboard memory limitations (Rojanasoonthon et al. (2003); Liu et al. (2017)). These constraints may also impact the performance of the remote sensing systems but they are out of the scope of this thesis.

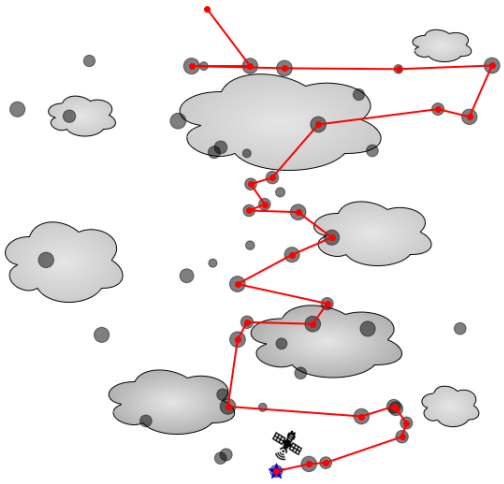


Figure 3.5 – An existing plan of activities (red line) needs to be revised during a cloudy day

In practice, the problem becomes more challenging in an uncertain context, where input data may change at the last minute. For instance, the

reward associated with each acquisition can be impacted by changes in the weather forecast (Liao & Yang (2007); Wang et al. (2015); Povéda et al. (2019)). As illustrated in Figure 3.5, during cloudy days, the images captured over a region have poor quality, which induces a reduction in their rewards. In such scenarios, the existing plan may no longer be good enough, and it can be useful to optimize again the sequence of observations to be performed. Moreover, the time available for re-planning is often limited, which makes the problem more complicated.

3.4 Conclusion

This chapter provides an overview of the orienteering problem and its typical variants. Benchmark instances of these variants are also available online. [1, 2] For further details, we refer to the surveys of Vansteenwegen et al. (2011); Gunawan et al. (2016); Vansteenwegen & Gunawan (2019). Overall, we observe that many researchers have extended their works beyond the classical OP by considering more complex features, such as correlated nodes, non-linear objective functions, multiple time windows, and so on. Some variants have not received much attention but are relevant for practical applications in the aerospace domain, such as time-dependent variants like TDOP or TD-TOPTW which are essential for capturing real-world scenarios. For these variants, being able to model congestion issues is a crucial requirement for the solving techniques before they can be implemented in practice.

Concerning solving techniques in the literature, there exists a large number of incomplete methods but relatively few hybrid complete/incomplete approaches for efficiently solving different large-scale OP variants. Meanwhile, the idea of hybridizing incomplete algorithms with complete techniques has become more and more popular over the past years. Hence, our studies presented in the following chapters of this thesis mainly focus on exploring such hybrid approaches with the aim of enhancing the (meta)heuristic search algorithms. The effectiveness of these approaches is demonstrated by dealing with several variants of orienteering problems and then focusing on a practical use case related to Earth Observation Satellites.

[1] `https://www.mech.kuleuven.be/en/cib/op`

[2] `https://www.hds.utc.fr/~moukrim/dokuwiki/en/top`

Part II

Contributions

CHAPTER 4

Integrating clause learning to incomplete search

In this chapter, we describe our first contribution, which aims to enhance metaheuristics for the OP and its variants by incorporating an external and long-term memory module. Precisely, we consider several types of conflicts that can be extracted during search and study how to effectively manage these conflicts. These conflicts are then reused to prune neighborhoods or diversify the search by answering queries formulated by the incomplete search process. We show the effectiveness of this framework with an application to the Orienteering Problems with Time Windows (OPTW).

The rest of this chapter is organized as follows. First, we describe a general framework combining clause learning and incomplete search in Section 4.1 and present an instantiation of this framework for solving OPTWs in Section 4.2. Following this, we detail a conflict generation procedure in Section 4.3 and propose three data structures usable for managing these conflicts as clauses in Section 4.4. In Section 4.5, we present the experimental results obtained on OPTW benchmarks to demonstrate the benefit of the architecture proposed, while the integration effort required is rather small. Finally, we compare our approach with relevant works in Section 4.6.

4.1 Incomplete search using a clause base

Local search-based algorithms are shown to be effective in solving NP-hard combinatorial optimization problems. However, a common issue is the repeated exploration of the same regions in the search space, which wastes computation time and resources, especially for large neighborhoods. Therefore, techniques for pruning neighborhoods, that are inconsistent or suboptimal, are essential. Moreover, in OPTWs, the search space is exponentially large, so diversification strategies are necessary to explore unvisited yet promising search regions.

To overcome these issues, a well-known technique is to use memory to improve the search efficiency. As presented in Chapter 2, this idea was originally proposed in tabu search (TS) (Glover (1989)), where a short-term memory can be used to avoid cycles and medium-term or long-term memories can be used to diversify the search. Besides, among the techniques that record some knowledge during search, conflict-directed approaches are commonly used for guiding the tree-based search. Typical examples are conflict-directed backjumping (Chen & Van Beek (2001)) and conflict-directed search for scheduling (Vilím et al. (2015)). Frequently, conflicts are represented as clauses and exploited during complete search procedures, such as in conflict-driven clause learning (CDCL) (Marques-Silva et al. (2009)) or lazy clause generation (LCG) (Schutt et al. (2013)).

A general framework This contribution aims to combine incomplete search with clause learning techniques used in SAT. For this purpose, we propose the hybrid architecture described in Figure 4.1, which is mainly inspired by the efficient complete search methods that use clause generation (e.g., CDCL or LCG). The architecture proposed consists of three main components: an *incomplete search process* (ISP), a *clause generator* (CG), and a *clause base* (CB).

The global search scheme works as follows. Each time ISP, the main search engine, converges to a locally optimal solution, the CG module analyzes this solution and produces clauses holding on Boolean decision variables of the problem. The clauses generated represent either the reasons why the current solution cannot be improved or conditions forbidding the local optimum or regions around to be reached again in the future. The clauses generated are then sent to CB. The latter is responsible for storing

the clauses and answering various queries that are relevant for the main ISP to prune or guide the neighborhood exploration. In this architecture, the clauses are generated in a lazy way, only for the parts of the search space that ISP decides to explore. By doing so, the architecture involves a tight interaction between the ISP and CB modules as well as a less frequent clause generation phase.

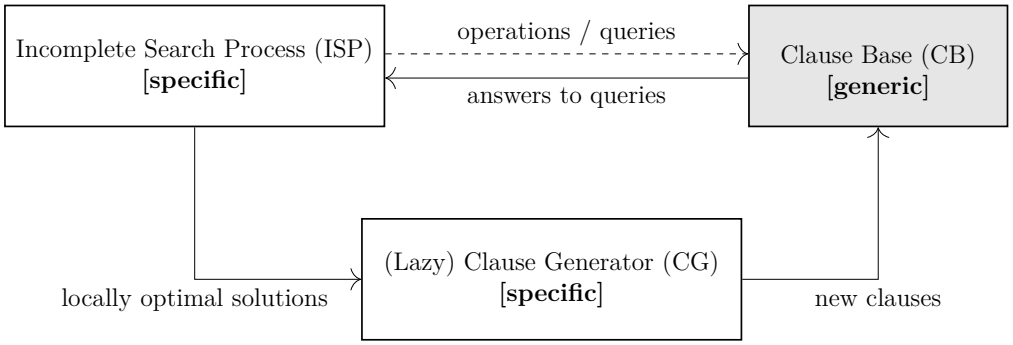


Figure 4.1 – Incomplete search combined with a clause base

With regards to tabu search, the architecture obtained memorizes clauses instead of just storing recent local moves or recent solutions in a tabu list. One impact is that the clause manager must be able to quickly reason about the clauses collected, instead of just reading explicitly forbidden configurations. Concerning CDCL or LCG, one key difference is that ISP is free to assign or unassign variables *in any order*, while the standard *implication graph* data structure used by CDCL or LCG relies on the assumption that the ordering of decision variables in the successive layers of the implication graph is consistent with the "chronological" order used for assigning and unassigning these variables during search. All these points raise several basic research questions:

- Which generic clause base data structure should be used to be able to follow the decisions made in *any* order by an incomplete search process and to quickly reason about the set of clauses memorized?

- What is the effort required to integrate an existing specific ISP within such a generic architecture, and which key functions should the clause base offer?

- What is the content of the clause generation module that analyzes the locally optimal solutions?

- From a practical point of view, is clause generation beneficial for an incomplete search that might have to explore thousands of successive neighborhoods per second?

4.2 An application to the OPTW

To evaluate the above framework, in the following, we consider the Orienteering Problem with Time Windows (OPTW) presented in Chapter 3. Globally, this problem can be decomposed into a selection subproblem and a sequencing subproblem. Regarding the selection aspect, we introduce one Boolean decision variable $x_i \in \{0, 1\}$ associated with each customer node i, where $x_i = 1$ means that node i is visited. One key idea is that pruning at the selection level can help significantly reduce the search space.

Technically, our primary goal is to boost an existing ISP for OPTWs thanks to clause learning. Therefore, we directly reuse the state-of-the-art LNS algorithm for OPTWs proposed by Schmid & Ehmke (2017) and integrate this algorithm within the hybrid architecture proposed. The enhanced version, called *LNS-CB* for *"LNS with a Clause Base"*, is depicted in Algorithm 4.1, where the few changes made on the baseline LNS version are highlighted in gray. Starting from an initial solution (Line 1), LNS-CB iteratively destroys and repairs the current solution following the standard concept of LNS (Lines 6-7). As in the baseline LNS algorithm, it also uses an elite pool to record the best solutions obtained so far (Line 4). This pool is reset whenever a better solution is found and extended when a new equivalent solution is obtained (Line 11). When no improvement is found after K iterations, a restart is performed by picking a random solution in the elite pool (Lines 12-13).

The destroy phase at Line 6 exactly follows the idea proposed by Schmid & Ehmke (2017). In this phase, several visited nodes are removed from σ, the current solution. Rather than removing individual nodes one by one, the approach removes sequences of customers using well-tuned heuristics. The key differences compared to the classical LNS algorithm are (a) the use of CB as an argument for the repair function, the objective being to improve the repair phase (Line 7), and (b) the generation of clauses each time a full

Algorithm 4.1: LNS-CB

```
1  σ ← construct()
2  clauseGeneration(σ, CB)
3  σ* ← σ;
4  elitePool ← {σ}
5  while time limit is not reached do
6      σ ← destroy(σ)
7      σ ← repair(σ, CB)
8      clauseGeneration(σ, CB)
9      if σ better than σ* then
10         σ* ← σ;
11         update elitePool
12     else if no improvement after K iterations then
13         σ ← a random solution in elitePool
14 return σ*
```

solution is produced, to enhance the knowledge stored in CB (Lines 2, 8).

The new repair phase is detailed in Algorithm 4.2. It takes as an input the current solution σ and CB. Given σ, we denote U as the set of unvisited nodes, and F as the set of feasible insertion moves (n, p) defined by a node $n \in U$ and a position p in σ. All insertion alternatives for each unvisited node are explored by `evalNeighborhood(σ, U, CB)` (Lines 2, 7). In this procedure, CB is used to prune neighbors that are invalid according to the clauses registered. At each step of the repair procedure, node insertions are iterated by selecting a move that has the best evaluation according to the well-tuned heuristics of the original LNS method (Line 4), and they are performed until there is no more feasible move (Line 3).

The neighborhood evaluation function corresponds to Algorithm 4.3. It first determines the unvisited nodes that *must* be visited according to CB (Line 1), and if there is no such mandatory node, it determines the unvisited nodes that *can* be visited according to CB (Line 3). Then, for each node selected, the algorithm determines its best insertion position according to tuned insertion heuristics defined in the original LNS (Schmid & Ehmke (2017)). In the end, the algorithm returns all pairs made by a node and its best insertion position. Basically, when the selection problem is tightly constrained, CB is especially efficient in pruning infeasible neighborhoods.

Algorithm 4.2: `repair`(σ,CB)

```
1 U ← nodes that are not in σ
2 F ← evalNeighborhood(σ, U, CB)
3 while F ≠ ∅ do
4 |   (n*, p*) ← select(F)
5 |   Insert node n* at position p* in σ
6 |   U ← U \ {n*}
7 |   F ← evalNeighborhood(σ, U, CB)
8 return σ
```

Algorithm 4.3: `evalNeighborhood`(σ,U,CB)

```
  /* Evaluation at the selection level                                  */
1 U' ← {n ∈ U | CB allows decision [x_n = 1] and forbids decision [x_n = 0] }
2 if U' = ∅ then
3 |   U' ← {n ∈ U | CB allows decision [x_n = 1]}
  /* Evaluation at the sequencing level                                 */
4 F ← ∅
5 for each n ∈ U' do
6 |   p ← best feasible insertion position for n in σ
7 |   if p ≠ nil then
8 |   |   F ← F ∪ {(n, p)}
9 return F
```

4.3 Lazy clause generation procedure

As mentioned earlier, the clause generation module strongly depends on the problem considered. This section details the clause generation procedure developed for the OPTWs. In this case, we consider only clauses holding over the selection decisions, and not clauses related to the detailed sequencing decisions defining the order of the visits. This is based on the fact that managing clauses over customer selections is easier and faster. Indeed, given N customers, we only need to manage constraints over N Boolean variables x_i expressing the selection of node i, instead of N^2 Boolean variables $next_{ij}$ commonly used for expressing sequencing decisions between any pair of nodes (i, j). For large-size instances, the number of sequencing variables can be

extremely high, however, the search space pruned by each clause over $next_{ij}$ variables can be relatively small, which is why we focus on the selection variables.

Several kinds of clauses can be generated during the search, and the generation of these clauses exploits *problem-dependent* techniques, as for the cuts generated in Logic-Based Benders decomposition (Hooker & Ottosson (2003)). For OPTWs, as illustrated in Algorithm 4.4, we consider two possible types of conflicts and generate them in a lazy way, i.e. only when a local minimum σ^* is reached. The generation of these clauses and parameters used in this procedure are detailed hereafter.

Algorithm 4.4: `clauseGeneration`(σ^*,CB)

```
   /* Local-optima-based clause generation (Section 4.3.1/optional) */
1  U ← sort{i ∉ σ*} based on their rewards R_i
2  Y ← best approxSize elements in U
3  Generate a temporary clause ⋁_{j∈Y} x_j
   /* Time-window-based clause generation (Section 4.3.2)           */
4  for i ∉ σ* do
5      V ← select(σ*, i, maxConfSize)
6      C ← extractMinTWconflicts(V ∪ {i})
7      for each TW-conflict S_c ∈ C do
8          Generate clause ⋁_{j∈S_c} ¬x_j
```

4.3.1 Clauses related to local optima

Inspired by tabu search, the core idea here is to avoid revisiting again and again local optima explored so far. To do so, whenever reaching a locally optimal solution σ^*, it is possible to generate clause $\bigvee_{j \notin \sigma^*} x_j$ to force that at least one node unvisited in σ^* must be selected in the future. Such a clause is called a local optimum conflict or *Lopt-conflict*. Adding this clause is safe in the sense that it will never forbid a solution that is strictly better than the best solution found so far.

Obviously, a smaller clause has a higher pruning power while requiring less management effort. Thus, rather than generating a *full* Lopt-conflict, we can derive an *approximate* Lopt-conflict $\bigvee_{j \in Y} x_j$ where Y contains at most *approxSize* nodes that are not involved in σ^*. To hopefully improve the

solution quality, we can favor the selection of nodes having a high profit in the future. This can be done by keeping in Y the best *approxSize* unvisited nodes in terms of profit (Algorithm 4.4, Lines 1-2). However, a large set of approximate clauses can increase the risk of pruning optimal solutions. To overcome this issue, these approximate clauses are used by CB only during a certain number of steps called *tabuSize*, similarly to a short-term tabu list (Algorithm 4.4, Line 3). So the main objective is to hopefully diversify search, which is also the reason why the generation of approximate Lopt-conflicts is optional here. This can be easily managed by adjusting the two global parameters *approxSize* and *tabuSize*.

Last, we could also avoid suboptimal solutions by generating a Pseudo-Boolean constraint $\sum_{i\in\{1,\dots,N\}} R_i x_i \geq LB + 1$ taking into account the reward R_i associated with each customer i and the best total profit known so far LB. We omit this point here, as we only focus on clause generation in this chapter, but we will explain it in more detail in Chapter 6.

4.3.2 Clauses generated based on temporal constraints

The other idea is to try to find valid explanations for infeasible local moves if possible. This can be done by analyzing the infeasibility or finding conflicts based on temporal constraints (e.g., travel times, time windows, and global limited time budget). Such conflicts can be formally defined as follows.

Definition 4.1. *Consider $\{1,\dots,N\}$ the set of nodes excluding the depots, where each node is associated with a predefined time window. Then, a time-window-based conflict (called **TW-conflict**) is a subset of nodes $S_c \subseteq \{1,\dots,N\}$ such that there is no sequence of visits that traverses all the nodes in S_c and respects their time window constraints.*

Intuitively, a TW-conflict S_c expresses that at least one customer $i \in S_c$ must not be selected due to temporal constraints. Hence, if all customers in $S_c \setminus \{i\}$ are visited, such an TW-conflict can be used to quickly detect that customer i cannot be selected during the construction process, without explicitly examining all possible insertions of i at every position in the current sequence of visits σ. Nevertheless, enumerating all such TW-conflicts is impossible. Thus, in Algorithm 4.4, we describe a procedure for generating clauses based on TW-conflicts in a *lazy* way. Technically, whenever a locally optimal sequence σ^* is found over nodes in S^*, we seek TW-conflicts preventing the other nodes from being added to σ^*. In other words, we try to find

valid explanations for every unvisited node i, if possible (Algorithm 4.4, Lines 4-8). To bound the complexity of the explanation procedure, we consider a maximum length of a TW-conflict referred to as *maxConfSize*. With a predefined *maxConfSize*, the algorithm first heuristically selects a set V containing *maxConfSize*-1 nodes in σ^* that *might* prevent a node $i \notin \sigma^*$ from being visited (Algorithm 4.4, Line 5). There are various problem-dependent heuristics for this selection, for example, we use here the *NearestTimeWindow* heuristic: to define V, we choose nodes $j \in S^*$ such that the distance between the midpoint of the time window of j and the midpoint of the time window of i is as small as possible. The idea here is to try and fill V with nodes that compete with i for being visited. Then, in function `extractMinTWconflicts`, a dynamic programming (DP) procedure determines whether $V \cup \{i\}$ is truly a TW-conflict. If so, it also extracts $\mathcal{C}$, a set of TW-conflicts of minimal cardinality and stores the corresponding clauses in CB (Algorithm 4.4, Lines 6-8). Indeed, the smaller the clauses the better, since smaller clauses prune larger parts of the search space.

For an easier understanding, let us describe a simple example before going into details about the procedure for extracting minimal TW-conflicts.

Example 4.1. *Consider the toy instance containing $N=7$ customers provided in Figure 4.2. For each customer, its time window and profit are explicitly given in the figure. We also assume that all travel times are equal to* 3 *time units and that all customer service times are equal to* 0.

Assume that solution $\sigma^* = [1, 2, 4, 6]$ is a current local optimum constructed based on the GreatestProfit heuristic, which considers inserting the customer having a highest profit at each construction step.

Then, we try to generate clauses by analyzing solution σ^*. In other words, we try to identify the reason why customers 3, 5, *and* 7 *cannot be added to σ^*. Let us assume that maxConfSize $= 3$.*

- *For customer* 3, *it is possible to see that visiting customers* 1, 2, *and* 3 *together is not possible. As a result, $S_{c_1} = \{1, 2, 3\}$ is a minimal TW-conflict and clause $c_1 : \neg x_1 \vee \neg x_2 \vee \neg x_3$ is generated.*

- *For customer* 5, *the selection is forbidden due to customer* 2 *since the transition requires 3 time units. Therefore, $S_{c_2} = \{2, 5\}$ is a TW-conflict and clause $c_2 : \neg x_2 \vee \neg x_5$ is generated.*

- *For customer* 7, *there may not be a valid explanation due to the visit order in σ^*. In other words, with another visit order of the customers*

in σ^*, there actually exists a solution that can visit customer 7, for example the sequence $\sigma' = [7, 1, 2, 6, 4]$.

- *For customer 8, no clause is added since there is no valid explanation of restricted size. However, there exists a valid TW-conflict $\{2, 4, 6, 8\}$ if one allows a larger value for maxConfSize.*

Last, to avoid reproducing σ^*, a clause $c_3 : x_3 \vee x_5 \vee x_7$ can be added to CB, as proposed in Section 4.3.1.

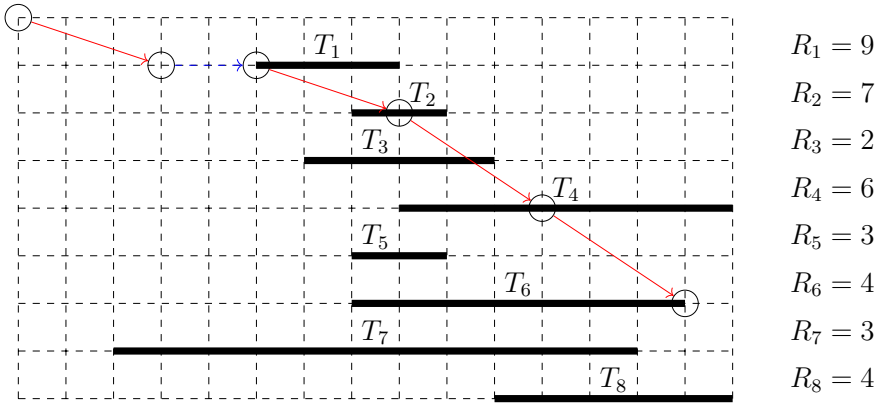


Figure 4.2 – OPTW example involving 8 customers, and representation of a valid sequence of visits

Minimal TW-conflict extraction Checking whether S_c is a TW-conflict is strongly NP-complete when the size of $S_c \subseteq \{1, \ldots, N\}$ is not bounded, since it is equivalent to determining whether there exists a feasible solution to a Traveling Salesman Problem with Time Windows (TSPTW) over the customers in S_c (Savelsbergh (1985)). However, if $|S_c|$ is bounded, the problem of checking and extracting the *minimal* TW-conflict is polynomial and solvable within a reasonable amount of time.

This can be done by using a dynamic programming procedure. The key principle is to compute, for each set S_c, quantities of the form $arr(S_c, i)$ representing the earliest arrival time at node $i \in S_c$ for a path starting at node 0, visiting all nodes in $S_c \setminus \{i\}$, and ending at node i. Similar to existing methods for TSP (Bellman (1962); Held & Karp (1962)), starting

from $arr(\{0\}, 0) = 0$, these quantities can be computed by increasing the size of S_c following a recursive formula.

$$arr(S_c, i) = \min(arr(S_c, j, i) \mid j \in S_c \setminus \{i\}) \tag{4.1}$$
$$arr(S_c, j, i) = \max(arr(S_c \setminus \{i\},\ j),\ O_j) + tt(j, i) \tag{4.2}$$

Recall that we denote $tt(i, j)$ as the time-independent transition time between two distinct customers i and j and $[O_i, C_i]$ as the time window available for starting the visit of node i. So, Equation 4.1 ensures that $arr(S_c, i)$ is the earliest arrival time at node i, while Equation 4.2 computes the arrival time at node i knowing that j is visited *just before* i. It is also noteworthy that the extension of the recursive formula above to the variant involving time-dependent transition times is quite straightforward.

Proposition 4.1. *Assume the availability of values returned by $arr(S_c, i)$ for any set S_c and node $i \in S_c$. Then, $S_c \subseteq V$ is a TW-conflict if and only if, for every $i \in S_c$, at least one of following conditions is satisfied:*
(a) $arr(S_c, i) > C_i$
(b) $max\ (arr(S_c, i), O_i) + tt(i, N + 1) > T_{max}$

Proof. Intuitively, the first condition corresponds to late arrivals at the last node i, while the second one corresponds to the violation of the global time limit when returning to the depot node $(N + 1)$. □

Next, the pseudocode of function `extractMinTWconflicts` is provided in Algorithm 4.5, based on Proposition 4.1. This function takes as an input a set of nodes $V \subseteq [1..N]$ and tries to determine all minimal TW-conflicts $S_c \subseteq V$ (minimal in terms of cardinality).

Proposition 4.2. *The* `extractMinTWconflicts(V)` *algorithm always returns TW-conflicts of minimal cardinality among TW-conflicts included in V, if such conflicts exist.*

Proof. The algorithm seeks TW-conflicts of size k in an increasing order (Line 3). Once a valid TW-conflict of size k is found, it continues seeking other TW-conflicts having a same size k until all subsets of size k are checked (Lines 10-11). Then, if there exists at least one conflicts having size k, the algorithm immediately stops and does not check any subsets of size larger than k (Lines 12-13). This implies that the TW-conflicts returned, if any, are of minimal cardinality in V. □

Algorithm 4.5: `extractMinTWconflicts(V)`

```
1  C ← ∅
2  arr({0}, 0) = 0
3  for k ∈ {2, ..., |V|} do
4      for S_c ⊆ V with |S_c| = k do
5          nFails ← 0
6          for node i ∈ S_c do
7              Compute arr(S_c, i) by following Equations 4.1 and 4.2
8              if arr(S_c, i) > C_i or
                 max(arr(S_c, i), O_i) + tt(i, N + 1) > T_max then
9                  nFails ← nFails + 1
10         if nFails = k then
11             C ← C ∪ {S_c}
12     if |C| > 0 then
13         return C
14 return ∅
```

Proposition 4.3. *The* `extractMinTWconflicts(V)` *algorithm, which is an extension of Held-Karp algorithm, has worst-case time complexity* $\mathcal{O}(2^n n^2)$ *and space complexity* $\mathcal{O}(n2^n)$*, where* n *is the number of customers in* V*.*

Proof. The algorithm considers 2^n subset $S_c \subseteq V$, where n is the number of customers in V (Lines 3-4). Then, it computes the value of $arr(S_c, i)$ for each node $i \in S_c$ following Equations 4.1 and 4.2. This computation has a worst-case time complexity $\mathcal{O}(n^2)$ and a space complexity $\mathcal{O}(n)$ (since $arr(S_c, i, j)$ are intermediate variables and do not consume memory for storage). Thus, `extractMinTWconflicts(V)` has worst-case time complexity $\mathcal{O}(2^n n^2)$ and space complexity $\mathcal{O}(n2^n)$. □

In practice, this procedure is pretty fast if there exist small TW-conflicts in V, since it does not necessarily enumerate all possible sets of $S_c \subseteq V$. In contrast, if no conflict exists, this procedure is time-consuming when $|V|$ is large. This is why, in this work, we only consider sets V of restricted size when invoking `extractMinTWconflicts`.

4.4 Data structures for the clause base (CB)

The CB part is responsible for storing the clauses generated during search. Due the separation of CB and the core search engine, CB can be externally saved and reused when ISP invokes it. As shown in Figure 4.1, ISP needs to frequently query CB, meaning that there is a need for *continuous* and *incremental* interactions between these two components. In other words, ISP needs to update the current state to CB whenever a decision is made. Meanwhile, CB also needs to quickly indicate to ISP whether a decision is worth considering given the current state. This raises many challenging questions about the choice of a specific data structure for CB. In principle, an ideal clause manager must be able to:

- quickly integrate all the clauses generated step-by-step and compactly represent them (possibly with some trashing when the size of CB becomes too large);

- frequently update the partial assignment of the decision variables over which the clauses hold, to keep up-to-date knowledge of the content of the current solution considered by the main ISP. For LNS-CB, this occurs whenever a node is selected or removed, and for a general ISP these assign/unassign decisions can be sent to CB in *any* order;

- quickly answer queries formulated by ISP, such as "evaluate whether decision $[x_i = 1]$ is feasible". For OPTW, if CB proves that this decision is infeasible given the current assignment and the clauses generated, then testing the insertion of node i in the current solution σ is unnecessary (neighborhood pruning). Another example is: "evaluate whether decision $[x_i = 1]$ is mandatory". If so, node i *must* be inserted into σ (mandatory visit).

The previous operations and queries required over CB are actually not specific to OPTW. In the following, we study three *generic* versions for CB:

- CB-UnitPropagation, where CB stores a list of clauses and performs *incremental* and *decremental* unit propagation to *evaluate* the consistency of the clause store for a given partial assignment of the x_i variables;

- CB-IncrementalSAT, where CB stores a list of clauses and employs powerful modern SAT solvers supporting incremental or assumption-based solving (Eén & Sörensson (2003); Nadel & Ryvchin (2012); Audemard et al. (2013));
- CB-OBDD, where the clauses are stored in an Ordered Binary Decision Diagram (OBDD), a data structure defined in the field of knowledge compilation that has good compactness and efficiency properties (Bryant (1986); Darwiche & Marquis (2002)).

4.4.1 CB based on unit propagation

For this version of CB, unit propagation is used to prune infeasible values for the decision variables. In SAT, unit propagation can be achieved based on the *two-watched literals* technique (Moskewicz et al. (2001)), which consists in maintaining, for each clause, two distinct literals that *can* take value *true*. In case there is no valid watched literal for a clause c, an inconsistency is detected. If only a single valid watched literal l is found, then clause c becomes *unit* and l must necessarily be *true* to satisfy the clause. In this case, literal $\neg l$ takes value *false* and unit propagation is applied to other clauses to further detect other propagated decisions.

Example 4.2. *Consider a clause* $c : x_1 \vee x_2 \vee x_3 \vee x_4 \vee x_5$ *as illustrated in Figure 4.3. Initially, all literals are unassigned yet (i.e. called free variables) and two watched literals of* c *can be randomly chosen e.g.,* $\{x_2, x_4\}$.

- *Let us assume that two decisions* $[x_1 = false]$ *and* $[x_5 = false]$ *are made. In this case, we do not need any update since the two watched literals remain valid. This also means that* c *is still consistent.*
- *If decision* $[x_4 = false]$ *is made, then literal* x_4 *is not valid anymore and cannot be watched by* c*. This leads to the use of another valid literal, e.g.,* x_2*, as a new watched literal for* c.
- *If decision* $[x_2 = false]$ *is made, however, no other valid watched literal can replace* x_2*, thus* c *is a unit clause. We can infer that* x_3 *necessarily takes values true.*

Consider another clause $c' : \neg x_3 \vee x_6 \vee x_7$*. In this case,* c' *can be replaced by* $x_6 \vee x_7$*, or equivalently* $\neg x_3$ *cannot be a valid watched literal for* c'.

In SAT, one advantage of the watched literal technique is that no literal reference needs to be updated when chronological backtracking occurs. But during incomplete search, variables can be assigned or unassigned in any order and some adaptations are required to maintain the watched literals. Precisely, to handle random variable unassignments and perform *decremental* unit propagation, we maintain a list of *complementary* watched literals *for each unit clause* c (see Figure 4.3). Clause c is revised whenever one complementary watched literal l' becomes free due to unassignment decisions, and in this case, l' can directly become a watched literal for clause c.

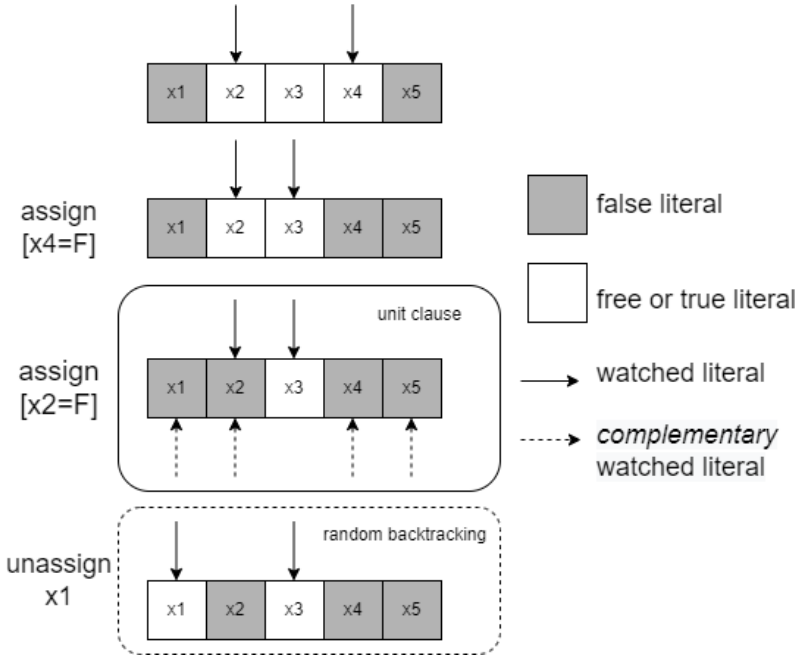


Figure 4.3 – Incremental and decremental unit propagation

Example 4.3. *Following Example 4.2, when c is a unit clause, we maintain a list of complementary watched literals for c, whose values are false, e.g.,* $\{x_1, x_2, x_4, x_5\}$. *For example, when* x_1 *is unassigned,* x_1 *can directly replace* x_2 *as a new valid watched literal for c. In this case, c is not a unit clause anymore, and* x_3 *is unassigned if there is no other unit clause that propagates* $[x_3 = true]$. *This may also trigger the revision of other clauses involving* $\neg x_3$ *as a complementary watched literal.*

To answer the queries formulated by the ISP, we additionally record a *justification* for each literal l referred to as $justif(l)$. Basically, we have the three following cases:

- $justif(l) = \top$ means that literal l takes value true because of a decision received from the ISP,
- $justif(l) = c$ means that literal l is propagated by unit clause c,
- $justif(l) = nil$ means that there is no clue about the truth value of l.

Then, a decision like $[x = 1]$ is allowed if and only if literal $\neg x$ is not propagated or decided yet, i.e. $justif(\neg x) = nil$. The justification of each literal is updated during incremental and decremental unit propagation. Obviously, as unit propagation is incomplete, CB-UnitPropagation may not detect some infeasible or mandatory node selections.

Example 4.4. *Consider four clauses*

$$c_1 : \neg x_1 \vee \neg x_2$$
$$c_2 : \neg x_4 \vee \neg x_5$$
$$c_3 : x_2 \vee x_3 \vee x_4$$
$$c_4 : x_2 \vee x_3 \vee x_5$$

If decision $[x_1 = 1]$ *is made, clause* c_1 *becomes unit and we have* $justif(x_1) = \top$ *and* $justif(\neg x_2) = c_1$*. The other justifications take value nil. This implies that decision* $[x_3 = 0]$ *is still evaluated as possible, even if it would lead to a dead-end.*

4.4.2 CB based on an incremental SAT solver

The idea of using incremental SAT solving was first proposed by Audemard et al. (2013) to improve the efficiency of the search for Minimal Unsatisfiable Sets. In this case, the goal is to reuse as much information as possible between the successive resolutions of similar SAT problems. This is done by working with *assumptions*. Basically, an assumption $\mathcal{A}$ is a set of literals $\{l_1, \ldots, l_k\}$ where each literal is considered as an additional (unit) clause by the solver, but this unit clause is *not permanently* added to the original CNF formula $\mathcal{F}$ defining the problem to be solved. In the context of OPTWs, the assumptions

are exactly the current node selection decisions. Then, a call $\texttt{solve}(\mathcal{F}, \mathcal{A})$ to an incremental SAT solver tries to find a model of $\mathcal{F}$ that satisfies all the assumptions in $\mathcal{A}$. By doing this, the incremental SAT solver can reuse some previous knowledge and learn new clauses that will potentially be reused for future calls $\texttt{solve}(\mathcal{F}', \mathcal{A}')$ using an updated CNF formula $\mathcal{F}'$ or an updated set of assumptions $\mathcal{A}'$.

At the level of CB, to determine whether literal $l : [x_i = a]$ can still be assigned value true, it suffices to call $\texttt{solve}(\mathcal{F}, \mathcal{A} \cup \{l\})$ where $\mathcal{A}$ is the set of assumptions representing the selection decisions made so far by the ISP module. Then, decision $[x_i = a]$ is forbidden by CB if and only if this call returns UNSAT. Contrarily to CB-UnitPropagation, the CB-IncrementalSat method is complete (as it performs a full look ahead).

Example 4.5. *Following Example 4.4, decision* $[x_3 = 0]$ *would directly be detected as invalid after assigning* $[x_1 = 1]$*. This is because* $\texttt{solve}(\mathcal{F}, \{x_1, \neg x_3\})$ *would return UNSAT.*

Last, we employ an additional optimization to reduce the number of calls to the `solve` function: when searching for the possible values of variable x_i given a set of assumptions $\mathcal{A}$, if $\texttt{solve}(\mathcal{F}, \mathcal{A} \cup \{x_i\})$ or $\texttt{solve}(\mathcal{F}, \mathcal{A} \cup \{\neg x_i\})$ returns a solution where another variable x_j takes value 1, then $[x_j = 1]$ is obviously allowed and calling $\texttt{solve}(\mathcal{F}, \mathcal{A} \cup \{x_j\})$ is unnecessary.

4.4.3 CB based on Ordered Binary Decision Diagrams

The last CB data structure studied here makes use of Ordered Binary Decision Diagrams (OBDDs) (Bryant (1986)). Storing conflict clauses in an OBDD during a systematic search process has been explored in the past, e.g., for a search process based on DPLL (Ignatiev & Semenov (2011)). We extend such an approach to deal with an incomplete search engine that again can assign/unassign the decision variables of the problem in any order.

As illustrated in Figure 4.4, an OBDD defined over a set of Boolean variables X is a directed acyclic graph composed of one root node, two leaf nodes labeled by $\top$ and $\bot$, and non-leaf nodes labeled by a decision variable $x_i \in X$. Each node associated with variable x_i has two outgoing arcs corresponding to assignments $[x_i = 0]$ and $[x_i = 1]$ respectively (dotted and plain arcs in the figure). The paths from the root node to leaf node $\top$ correspond to the assignments that satisfy the logical formula represented by the OBDD, while the paths leading to leaf node $\bot$ correspond to the inconsistent assignments.

Additionally, OBDDs are *ordered*, meaning that the variables always appear in the same order in any path from the root to the leaves.

CNF:
$(\neg x_1 \vee \neg x_2)$
$\wedge(\neg x_4 \vee \neg x_5)$
$\wedge(x_2 \vee x_3 \vee x_4)$
$\wedge(x_2 \vee x_3 \vee x_5)$

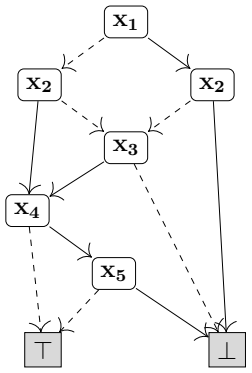


Figure 4.4 – A conjunction of clauses and an equivalent OBDD

In practice, to get a compact representation, OBDDs are also *reduced*, meaning that redundant nodes (that have the same children) are recursively merged. Such a data structure offers several advantages, including (1) the capacity to be exponentially more compact than for instance an explicit representation of all models of the logical formula that the OBDD represents, and (2) the capacity to perform several operations and to answer several queries in polynomial time. For instance, given two OBDDs $\mathcal{O}_F$ and $\mathcal{O}_G$ representing logical formulas f and g and that use the same variable ordering, operation "$\mathcal{O}_F \wedge \mathcal{O}_G$" computes an OBDD representing $f \wedge g$ in polynomial time in the number of nodes in $\mathcal{O}_F$ and $\mathcal{O}_G$.

In CB-OBDD, one OBDD referred to as $\mathcal{O}_{CB}$ stores the clauses learned during search. Initially, $\mathcal{O}_{CB}$ only contains leaf node $\top$ since all models are accepted. Each generated clause c_k can be transformed into an OBDD $\mathcal{O}_{c_k}$, and a set of new clauses $\{c_1, \ldots, c_n\}$ is added to $\mathcal{O}_{CB}$ by $\mathcal{O}_{CB} \leftarrow [\mathcal{O}_{c_1} \wedge \ldots \wedge \mathcal{O}_{c_n}] \wedge \mathcal{O}_{CB}$ (conjunction of the elementary OBDDs associated with the new clauses followed by a batch addition into $\mathcal{O}_{CB}$). During search, CB-OBDD records the current list of assignments A_{CB} made by the incomplete search process (the assumptions). To determine whether a decision $[x = 1]$ is allowed, it suffices to *condition* $\mathcal{O}_{CB}$ by A_{CB}, and then to check that assignment $[x = 0]$ is not *essential* (not mandatory) for the resulting OBDD. The *conditioning primitive* and the search for *essential variables* are standard operations in OBDD packages. Their time complexity is linear in the number of OBDD nodes.

4.5 Experiments

4.5.1 Benchmark and implementation settings

We carried out experiments on standard OPTW benchmarks[1] whose features are summarized in Table 4.1. As the number of customers increases, finding an optimal selection and sequencing of customers is more challenging. Overall, the best-known total reward for each instance is retrieved from Schmid & Ehmke (2017). We conduct experiments on Intel(R) Core(TM) i5-8265U 1.60 GHz processors with 32GB RAM. All implementations are written in C++ and compiled in a Linux environment with g++17.

Instance Set	#instances	#nodes	remark
Solomon 1 (c1*,r1*,rc1*)	29	100	-
Solomon 2 (c2*,r2*,rc2*)	27	100	wider TW
Cordeau 1 (pr01-pr10)	10	48-288	-
Cordeau 2 (pr11-pr20)	10	48-288	wider TW

Table 4.1 – Features of the OPTW benchmarks

As the implementation of the baseline LNS algorithm of Schmid & Ehmke (2017) is not available online, we re-implemented it. We recover a similar performance even if there are some differences with respect to the results provided in the original paper, possibly due to random seeds or to a lack of information concerning a reset parameter K (we set $K = 50$ in our LNS implementation i.e. Algorithm 4.1, Line 12). Anyway, our primary objective was to determine whether conflict generation can help a baseline algorithm, therefore the slight differences in performance are not a real issue.

Concerning the management of clauses, the three data structures proposed for CB are implemented as follows.

- For CB-UnitPropagation, the data structure is implemented from scratch.
- For CB-IncrementalSat, we directly reuse CryptoMiniSat[2] (Soos et al. (2009)) that supports incremental solving and won the Incremental Track in the SAT competition 2020.

[1] Instances are available at `https://www.mech.kuleuven.be/en/cib/op`

[2] `https://github.com/msoos/cryptominisat`

- For CB-OBDD, we reuse the CUDD library that offers many functions to manage OBDDs.[3] Notably, the variable ordering has a strong impact on the size of OBDDs, while finding an ordering giving a minimal size is NP-hard (Tani et al. (1996); Bollig & Wegener (1996)). Hence, we directly use the dynamic reordering operations available in CUDD employing Rudell's sifting algorithm (Rudell (1993)). Dynamic reordering can take some time but reducing the size of OBDDs can pay off in the long term.

4.5.2 Parameter tuning for lazy clause generation

As illustrated in Figure 4.5, we observe that the length of the time windows in OPTW instances has a large impact on the number of TW-conflicts generated for a given value of *maxConfSize*: many TW-conflicts are generated for the Solomon 1 & Cordeau 1 instances, contrarily to the case of Solomon 2 & Cordeau 2 instances that involve longer time windows. This is reasonable since longer time windows make the problem less constrained, which implies that a TW-conflict, if exists, usually has a large size. Besides, the complexity of the dynamic programming algorithm producing the TW-conflicts is exponential in *maxConfSize* and the computation times indeed strongly increase when *maxConfSize* is higher (Figure 4.6). Thus, we decided to set $maxConfSize = 4$ after the analysis of the global search efficiency.

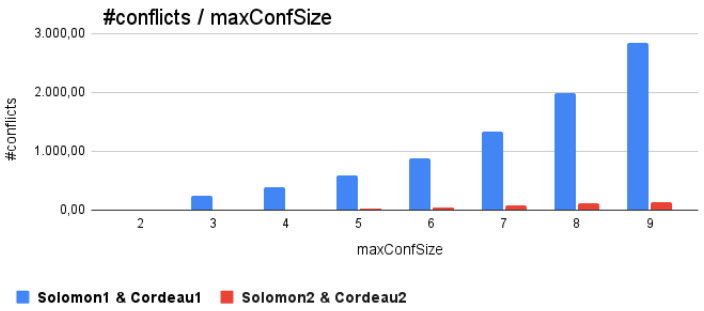


Figure 4.5 – Impact of *maxConfSize* on the number of TW-conflicts

[3] `https://github.com/ivmai/cudd`

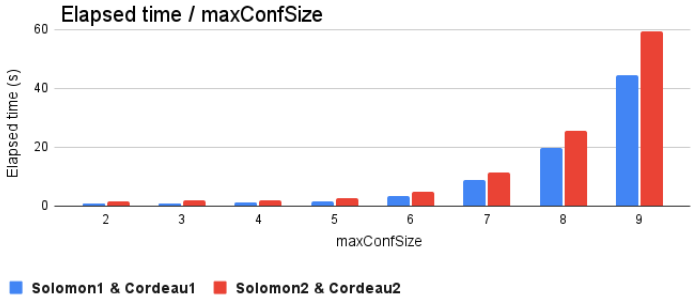


Figure 4.6 – Impact of *maxConfSize* on the clause generation time (within a 1-minute time limit for LNS)

However, generating TW-conflicts all the time can slow down the global search. Therefore, we define an *explanation quota* (denoted as *xpQuota*) for every node i to reduce the workload of function `extractMinTWconflicts`. This quota is decreased by one unit each time no TW-conflict explaining the absence of customer i in a locally optimal solution is found. When the quota of customer i becomes 0 after *xpQuota* searches for TW-conflicts related to i, the absence of customer i in a locally optimal solution is not explained anymore. With such an approach, there is somehow a warm-up phase where TW-conflicts are learned, followed by an exploitation phase of these conflicts. After performing tests with different values of $xpQuota \in \{20, 60, 100\}$, we decided to set $xpQuota = 20$.

Last, concerning the generation of Lopt-conflicts to diversify search, we need to forbid during *tabuSize* iterations a region around a locally optimal solution, where the region size is controlled by the *approxSize* parameter which defines the maximum size of the approximate Lopt-conflicts. After several tests performed with $approxSize \in \{3, 5, 7\}$ and $tabuSize \in \{10, 50, 100, 200\}$, we set $approxSize = 7$ and $tabuSize = 50$ for the experiments.

4.5.3 Performance analysis of the different CB data structures proposed

Experiments are performed for the three CB data structures presented above.

- For LNS-CB-UnitPropagation (or shortly LNS-CB-UP), we con-

sider two versions: one called LNS-CB-UP where no Lopt-conflict is generated, and another called LNS-CB-UP-LOPT where Lopt-conflicts are generated.

- For LNS-CB-INCREMENTALSAT (or shortly LNS-CB-SAT), we do not consider the case with Lopt-conflicts since the interface of CryptoMiniSat does not support clause removal.
- For LNS-CB-OBDD, we do not use the temporary Lopt-conflicts as it would require (a) maintaining an OBDD containing only permanent TW-conflicts, and (b) making time-consuming conjunctions with the temporary Lopt-clauses that are still active at the current iteration.

Overall performance To quickly compare the baseline incomplete search algorithm (called LNS-NOCB) and the versions using a CB, we first compute, for each solver and each instance, the average gap to the best known solution after five runs, each within 1 minute. This gap g_s for solver s is defined by

$$g_s = 100 * \frac{bk - bf_s}{bk}$$

where bf_s is the total reward of the best feasible solution found by s and bk is the best known objective value. Table 4.2 shows that for 1-minute time limit, using CB-UP globally improves the gaps (0.851% compared to 0.886% when using NOCB, while using CB-UP-LOPT also generates competitive results). On the contrary, CB-SAT and CB-OBDD deteriorate the average gap (mean gaps equal to 1.739% and 1.418% respectively).

Instance set	Variants of CB in LNS					
	noCB	UP	UP-Lopt	SAT	OBDD	simpleTabu
Solomon1	1.093	1.093	1.304	1.492	1.315	**0.083**
Solomon2	0.416	0.387	**0.345**	0.607	0.497	4.097
Cordeau1	0.139	**0.078**	0.351	1.125	0.903	1.540
Cordeau2	1.898	**1.846**	1.900	3.729	2.958	2.119
Grand mean	0.886	**0.851**	0.977	1.739	1.418	2.332

Table 4.2 – Average gap (%) over 5 runs (maxCPUtime=60s, best average gaps in **red bold**)

To further analyze the results, each version of the solver is executed during 10 000 LNS iterations and the total time elapsed over each set of instances is

measured. Then, a speed-up rate compared to the noCB version is computed by

$$speedUp_s = 100 * \frac{timeNoCB - timeWithCB_s}{timeNoCB}$$

Table 4.3 shows that the search process is accelerated with CB-UP and CB-UP-Lopt almost all the time, especially on the Cordeau instances where the speed-up reaches almost 50%. On the contrary, the search process is drastically slowed down with CB-Sat and CB-OBDD.

Instance set	Variant of CB in LNS			
	UP	UP-Lopt	SAT	OBDD
Solomon1	-8.83	-18.66	-2517.14	-646.14
Solomon2	**25.17**	**25.15**	-492.75	-163.62
Cordeau1	**48.66**	**47.04**	-2779.32	-2446.31
Cordeau2	**45.96**	**47.83**	-2092.35	-610.95

Table 4.3 – Speed-up (%) when solving during 10 000 LNS iterations

Slow convergence with CB-SAT and CB-OBDD. Despite the rapidity of incremental solving with CryptoMiniSat, the results obtained show that the search process is slower for the LNS-CB-Sat version. The main reason for this is that there are numerous calls to $\texttt{solve}(\mathcal{F}, \mathcal{A} \cup \{l\})$, and each call must either find a full solution or prove that none exists.

As for CB-OBDD, while querying in OBDD is fast, the results are not as good as expected. To understand this behavior, we analyzed the number of conflicts generated during search, the number of OBDD nodes, and the time consumed by the different OBDD functions. Table 4.4 shows that the OBDDs obtained are globally compact given the number of conflicts. But the reordering operations performed to get such a compactness can take a lot of time: on some instances, CB-OBDD spends more than 60% of the CPU time for reordering the variables. Alternatively, it is challenging to heuristically compute in advance a good static variable ordering for compiling OBDDs in a bottom-up manner, since we do not have the entire information about the conflicts as well as when a static ordering must be defined. Meanwhile, we tested eight problem-dependent heuristics (e.g., ordering the selection variables depending on the rewards, the time windows, etc.), and as shown in Table 4.5, the best heuristics are different in two cases, but overall these static orderings give poor results on some instances.

Instance set	#OPTW nodes	#conflicts (average)	#OBDDnodes (average)	reordering time (%)
Solomon1	100	509.66	257.59	34.15
Solomon2	100	19.19	11.19	6.91
Cordeau1	48-288	109.10	303.50	67.73
Cordeau2	48-288	0.40	1.70	2.15

Table 4.4 – Size of CB for each instance group (CPU time: 10s)

instance	LNS iteration	#conflicts	best-static-ordering		dynamic-ordering	
			#nodes	time(s)	#nodes	time(s)
pr01	1	0	1	0.0006	1	0.0012
	2	2	7	0.0013	5	0.0026
	3	8	36	0.0021	13	0.0041
	4	8	36	0.0030	12	0.0054
pr06	1	0	1	0.0885	1	0.4886
	2	55	69219	0.1803	477	2.5110
	3	80	6342191	19.1407	533	2.6108
	4	94	38250383	367.198	833	2.6422

Table 4.5 – Performance of the static and dynamic ordering strategies for OBDDs on two instances (pr01: 48 variables, best static order found = "increasing opening time"; pr06: 288 variables, best static order found = "decreasing rewards")

Better and faster search with CB-UP. Figures 4.7 and 4.8 detail the evolution of the mean gap during search over each set of instances. Globally, we observe that LNS is boosted by CB-UP over all sets. In particular, for the Cordeau 1 set which involves many TW-conflicts, the search process converges much more quickly with the support of CB-UP. This is because more LNS iterations are performed thanks to the effectiveness of neighborhood pruning through CB-UP. The strength of CB-UP-Lopt is particularly visible over instance sets Cordeau 2 and Solomon 2. In these cases, even with very few TW-conflicts, the approximate Lopt-conflicts help guide the search towards other interesting search regions.

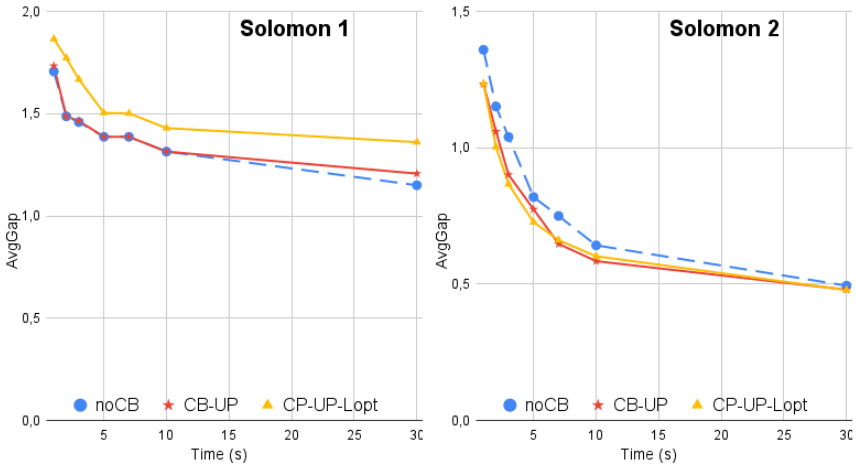


Figure 4.7 – Evolution of the average gaps for CB-UP and CB-UP-Lopt on Solomon instances

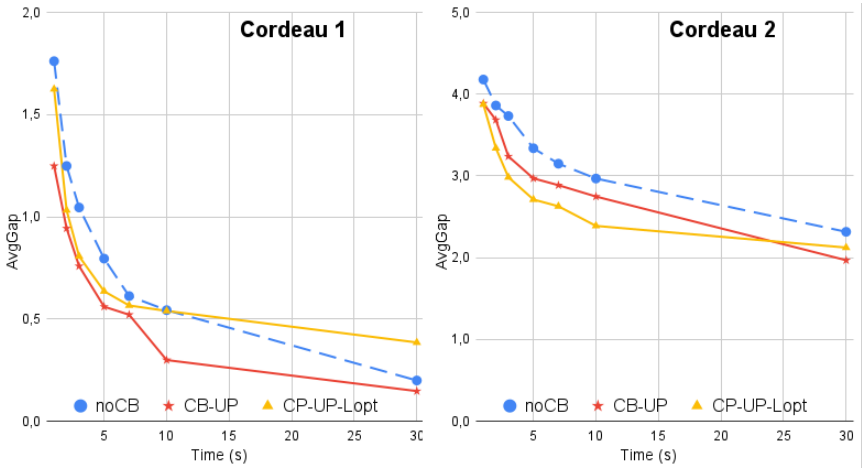


Figure 4.8 – Evolution of the average gaps for CB-UP and CB-UP-Lopt on Cordeau instances

4.6 Related works and discussion

Incomplete search and SAT/CP were combined in Large Neighborhood Search (LNS), where a sequence of destroy-repair operations is performed on an incumbent solution. The destroy phase unassigns a subset S of the decision variables, while the repair phase can be delegated to a SAT/CP engine ca-

pable of quickly exploring all possible reassignments of S given the current partial assignment. Some authors also proposed to represent specific neighborhood structures using a tailored CP model and to translate the solutions found for this model into changes at the level of the global solution (Pesant & Gendreau (1999)). In the same spirit, our CB is built to quickly detect inconsistent assignments at the selection level, therefore it can significantly reduce the neighborhood size to explore in the repair phase, but one difference is that we generate new conflicts during search and for the incomplete search process, CB only acts as a constraint propagation engine.

Other hybrid approaches exploit the strengths of incomplete search and complete SAT/CP techniques at different search phases. As an illustration, in SAT, *Stochastic Local Search* (SLS) has been combined with DPLL or *Conflict Directed Clause Learning* (CDCL) (Crawford (1996); Mazure et al. (1998); Audemard et al. (2010)). For the SLS-CDCL version, the idea is that on one side, SLS can be run first to help CDCL have a heuristic for choosing variable values or to help CDCL update the activities of the variables, and on the other side CDCL can help SLS espace local optima. Another example is the composition of traditional CP search and Constraint-Based Local Search (CBLS) (Hentenryck & Michel (2005)), where the two search approaches can exchange bounds, solutions, etc. In line with previous studies, inconsistency explanations generated at each iteration are stored in CB and then reused to help the search engine escape explored or invalid regions. In our case, by taking into account the current search state along with the clauses learned in the past iterations, CB may suggest mandatory assignments to quickly lead the search to promising regions.

Another technique uses inference methods such as unit propagation or constraint propagation initially developed for complete search strategies, to speed up the neighborhood exploration during local search. As already presented in Chapter 2, one typical example following this line for SAT is the *UnitWalk* algorithm (Li et al. (2003); Hirsch & Kojevnikov (2005)). At each iteration, it considers a complete variable assignment and performs a pass over this assignment to iteratively update the values of the variables with unit propagation. Compared to this work, one of the novelties in CB-UP is the decremental propagation aspect.

Last, the use of an external CB coupled with incomplete search can be compared with the use of a memory data structure in tabu search. On this point, instead of a simple list of forbidden local moves or forbidden variable assignments as in tabu search (Glover & Laguna (1998)), CB memorizes log-

ical formulas about the selection of nodes in a long-term way (possibly with some trashing when the size of CB becomes too large). CB is also equipped with efficient mechanisms to quickly reason about the formulas collected, instead of just reading explicit forbidden configurations. Another remark is that traditional tabu search is usually not recyclable i.e. the memory is reset at each resolution, while the time window conflicts stored in CB are easily recyclable for dynamic OPTWs where the reward associated with each node can change.

4.7 Conclusion

Our primary objective is to enhance memoryless incomplete search algorithms with a memory component and exploit reasoning techniques used in complete methods to improve the search performance. However, it is a challenge to efficiently manage and use this knowledge during search without increasing the computational resources. To address this challenge, in this chapter, we have described a hybrid optimization architecture combining an incomplete search process with clause generation techniques. More precisely, we have presented an application to the OPTWs with details on (1) a *specific* lazy clause generation procedure and (2) three *generic* clause managers inspired by techniques in SAT solving and knowledge compilation (e.g., ordered binary decision diagrams). The empirical study we have performed on the classical OPTW benchmark demonstrates the efficiency of the approach proposed, in particular when the clause base is exploited by using an enhanced version of watched literal techniques and unit propagation mechanism commonly used in SAT community. With a well-tuned configuration, we have observed that the clauses generated were able to help pruning inconsistent neighborhood as well as diversifying search without decreasing the search performance. The architecture proposed is also applicable to other combinatorial optimization problems, beyond OPTWs.

CHAPTER 5

Route recombination procedure for deterministic and non-deterministic scenarios

In this chapter, we describe our second contribution to hybrid optimization methods for routing problems with profits. Differently from the clause generation mechanism of the previous chapter that exploits *nogoods*, the contribution presented in this chapter aims at exploring *goods* that are features of high-quality solutions. More specifically, our goal is to post-optimize the quality of solutions obtained by classical meta-heuristics for the (Time-dependent) Orienteering Problem with Time Windows ((TD)OPTW). With this aim, we introduce a novel Route Recombination procedure that is able to take as an input a set of solutions and return the best combination containing at most k subsequences of customer visits from these solutions. This route recombination procedure is based on a dynamic programming algorithm enhanced with pruning strategies that significantly reduce the size of the search space. It is also able to deal with or without time-dependent transition times. The experiments show that the algorithm proposed can be used as a lightweight and efficient post-optimization procedure working on elite solutions provided by a standard incomplete (TD)OPTW solver. Moreover, it can be used in a non-deterministic context where the reward values are not precisely known in advance; in this case, (TD)OPTW solutions can be first generated for various reward scenarios during an offline phase, and then

combined during an online phase to quickly get a high-quality solution given the last-known reward values.

The rest of this chapter is organized as follows. Section 5.1 describes an incomplete baseline search algorithm used to generate a pool of good-quality solutions for a (TD)OPTW. Section 5.2 illustrates the idea of the Route Recombination (called RR) procedure with a simple example. Section 5.3 formalizes the RR procedure. Section 5.4 discusses possible usages of RR for deterministic and non-deterministic (TD)OPTW. Section 5.5 gives the experimental results obtained on classical OPTW benchmarks and on an additional benchmark related to satellite scheduling problems. Last, Section 5.6 discusses the contributions compared to other relevant works, and Section 5.7 concludes by providing several perspectives.

5.1 Generation of a pool of solutions

To quickly generate a pool of elite solutions for a given (TD)OPTW, we consider an incomplete algorithm that uses Iterated Local Search (ILS) and Large Neighborhood Search (LNS). These metaheuristics were shown to be efficient for solving both OPTWs (Gunawan et al. (2015a); Schmid & Ehmke (2017)) and TDOPTWs (Garcia et al. (2013)). As shown in Algorithm 5.1, the basic idea is to iteratively *destroy* (Line 2) and *repair* (Lines 3-8) a solution until the stopping criterion is met (Line 1).

More precisely, in the *destroy* phase, the algorithm removes up to $x\%$ of the customer nodes appearing in the current sequence of visits, referred to as σ. The number of nodes selected for removal, denoted n_R, is randomly chosen from a discrete uniform distribution $n_R \sim U(1, x \cdot n_\sigma)$, where n_σ is the number of customer nodes currently visited in σ, excluding depot nodes. As opposed to the LNS algorithm proposed by Schmid & Ehmke (2017) and the version we implemented in Chapter 4, the removal operator here is simplified by randomly choosing a node $i \in \sigma$ from which a subsequence of length n_R is removed. Compared to the ILS algorithm of Vansteenwegen et al. (2009c), the destroy phase is similar to a random *shake* operator.

After that, the partial solution obtained is *repaired*. For this, the algorithm first uses the *insert* operator to insert unvisited customer nodes back into the current solution (Line 3). Compared to the LNS we implemented in Chapter 4, the repair phase is enhanced with a local search procedure that exploits several neighborhoods (Lines 4-8). The idea is to locally improve

the current solution until no further improvement is found. As in the work of Gunawan et al. (2015a), the local search procedure alternates between applying the *swap* and *2-opt* operators to reduce the travel time, and applying the *replace* and *insert* operators to improve the solution quality in terms of collected rewards. All along the search process, a set of elite solutions of bounded size is updated (Line 9).

Algorithm 5.1: LNS for the (TD)OPTW

```
1  while cpu() < maxCPU do
2      destroy x%
3      insert
4      while improvement do
5          swap
6          2-opt
7          replace
8          insert
9      update elite pool
10 return elite pool
```

Globally, the structure of the LNS algorithm obtained directly reuses existing ideas from the literature and our implementations from Chapter 4, the only contribution here is that adaptations are made to handle both time-independent and time-dependent transitions. For the sake of simplification, we do not consider here the version combining LNS with conflict learning.

5.2 Route Recombination: an example

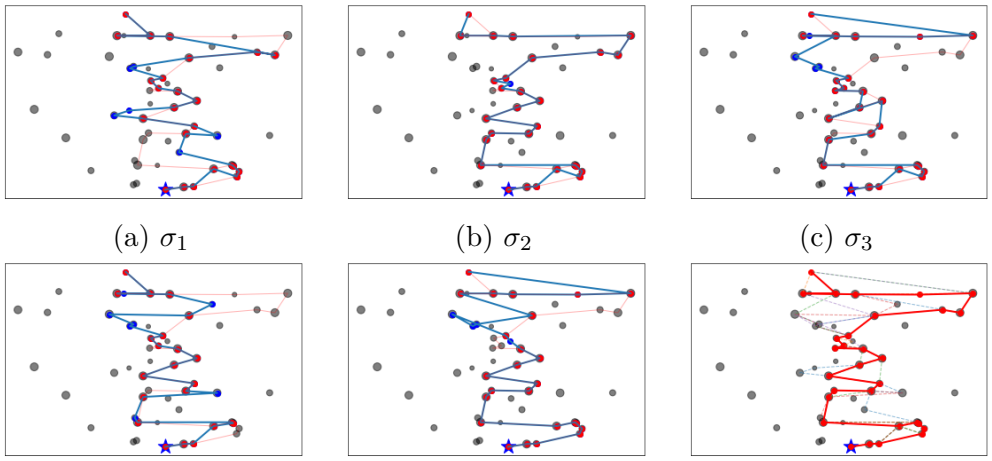
To illustrate RR, let us consider Table 5.1 that gives a pool of five locally optimal solutions $\sigma_1, \ldots, \sigma_5$ found by LNS for a specific OPTW instance. Each solution represents a feasible sequence of visits over a subset of customer nodes, where each node corresponds to a location on a map as shown in Figure 5.1. We observe that by combining subsequences available in elite solutions of the pool, it is possible to find a new solution σ^* that has a better global reward $R(\sigma^*)$ equal to 652.

However, finding such a recombination of subsequences is challenging in the general case, since exploring all possible combinations involving k subsequences is exponential in k. To reduce the complexity, we can limit the

maximum number of subsequences used for each combination, that is the maximum number of so-called *jumps* between different elite solutions, where a jump refers to a change in the elite solution followed at a given step.

id	sequence of visits	rewards
σ_1	0 5 4 7 28 10 42 20 46 12 39 44 45 41 24 16 37 21 50 1 18 32 13 48 40 53 8 36 27 22 0	633
σ_2	0 5 4 7 28 10 42 20 46 15 3 11 39 45 41 37 21 50 1 54 18 32 40 53 8 6 36 22 27 0	637
σ_3	0 5 4 7 28 10 42 20 46 15 3 39 45 21 41 37 50 25 1 18 32 13 48 17 22 33 27 36 38 6 0	634
σ_4	0 5 4 7 28 20 46 15 19 3 44 39 45 41 37 21 50 1 18 32 13 48 17 40 30 36 27 33 22 0	631
σ_5	0 5 4 7 10 42 20 46 28 15 3 11 39 45 41 37 21 50 54 32 17 13 48 40 27 22 36 38 6 0	634
σ^*	0 5 4 7 10 42 20 46 28 15 3 11 39 45 41 37 21 50 25 1 18 32 40 53 8 6 38 36 27 22 0	652

Table 5.1 – An improved solution (σ^*) that can be obtained by combining solutions $\sigma_1, \ldots, \sigma_5$ in the elite pool (results obtained on instance *ti-singlesat-ttf2.0-500-16* used in the experiments)



(a) σ_1 (b) σ_2 (c) σ_3

(d) σ_4 (e) σ_5 (f) σ^*

Figure 5.1 – For the example of Table 5.1, the above figures a-f provide the routes associated with the solutions of the pool (blue lines) and with the combined route σ^* (red line); the *star* represents the starting node; the red points represent the nodes visited in σ^*

From Table 5.1, we observe that it is possible to create σ^* by using 4 jumps and 5 subsequences highlighted in gray:

- (1) starting with σ_5, the first *forward jump* is made from σ_5 to σ_3 via node 50 to connect the red subsequence with the green one;

- (2) the second jump is made through node 32 to connect the green subsequence in σ_3 to the orange subsequence in σ_2;
- (3) the third jump, called a *backward jump*, is from σ_2 to σ_3 via node 6; it links the orange subsequence with the blue one, but traverses the blue subsequence in the backward direction;
- (4) the last jump is from σ_3 to σ_1 through node 36. All these jumps lead to solution σ^*.

It should be emphasized that a jump can be performed in the forward or backward direction, the semantics being that a backward jump traverses a subsequence of visits in the reverse order. Figure 5.1 details the sequences of visits used in each solution of the pool and in solution σ^*.

5.3 Dynamic programming formulation

In the following, we detail the Dynamic Programming (DP) algorithm used for the RR procedure. We first describe the DP system with the definition of states in Section 5.3.1 and state extension rules in Section 5.3.2. We then show a detailed pseudo-code of the recombination procedure in Section 5.3.3. To reduce the search space, we introduce several pruning strategies in Section 5.3.4 as well as a bounded-width version of RR in Section 5.3.5. We also provide a detailed analysis about the complexity of the RR procedure in Section 5.3.6.

For ease of understanding, we summarize in Table 5.2 several notations in the formulation of (TD)OPTW used in the following sections.

Symbol	Semantics
$i \in \{0, \ldots, N+1\}$	a customer node, where 0 and $N+1$ are depot nodes
R_i	a reward associated with node i, where $R_0 = R_{N+1} = 0$
$[C_i, O_i]$	a predefined time window during which node i can be visited
T_{max}	the limited time budget
$tt(i, j, \tau_i)$	minimum transition time from node i to node j, where τ_i is the departure time at node i
σ	a solution i.e. a sequence of visits

Table 5.2 – Summary of notations of (TD)OPTW

5.3.1 Search states

Formally, the RR procedure explores *search states* representing possible paths from the source depot to the sink depot. These paths are built by following the solutions of the pool. Formally, each search state is defined as a tuple $S = (V, i, r, d, n)$, where V is a set of visited customers, i is the last visited customer, r is the current route (or solution) being followed, $d \in \{FWD, BWD\}$ is the current exploration direction of r (forward or backward), and n is the number of jumps made so far. Intuitively, different states associated with the same last visited customer i correspond to different feasible paths reaching customer i.

Example 5.1. *Let us consider an example based on the solutions provided in Table 5.1. Let us assume that we initially follow solution σ_5, then make the first jump from σ_5 to σ_3 through node 50, and then continue the visits until node 32. At this point, the current search state is*

$$S = (\{0, 5, 4, \ldots, 18, 32\}, 32, \sigma_3, FWD, 1)$$

States are explored in a *level-by-level* manner, where the *level* of a state $S = (V, i, r, d, n)$ corresponds to the number of nodes visited in V. Initially, for each route r in the pool, we define a starting state $S_0^r = (\{0\}, 0, r, FWD, 0)$ where the starting point is the source depot and the forward direction is being followed. A state (V, i, r, d, n) is said to be *terminal* when it reaches the sink depot, *i.e.* $i = N + 1$. Between the initial and terminal states, the intermediate states represent incomplete solutions that visit a subset of customers but do not return to the depot yet.

The *reward* $R(S)$ of a state $S = (V, i, r, d, n)$ is computed as the sum of the rewards R_j of the customers j visited in V, *i.e.* $R(S) = \sum_{j \in V} R_j$.

The *cost* of a state $S = (V, i, r, d, n)$, denoted as $arr(S)$, is the minimal arrival time at node i if all the customers in V are already visited, route r is being followed in direction d, and n jumps have been performed so far. Initially, $arr(S)$ is set to 0 for the initial states and to $+\infty$ for the other states. During the exploration of the search space, $arr(S)$ is updated each time a new feasible path reaching S is found. Each non-initial state S also keeps track of a *parent state* to visit before S in order to get cost $arr(S)$. To build the best route from the start to the end depot, it suffices to select a terminal state that has a maximal reward and extract the corresponding best path by backward propagation through the parent states. In the following, we denote by J_{max} the maximum number of jumps allowed for each state.

5.3.2 Extension rules

As illustrated in Figure 5.2a, extending a state $S = (V, i, r, d, n)$ corresponds to selecting a route and a direction from which we try to find the next customer to visit after i. Formally, such an action is represented as a pair (ρ, δ) where ρ is a route containing customer i and δ is a direction (forward or backward). Applying action (ρ, δ) in state S leads to a new state $S' = (V \cup \{i'\}, i', \rho, \delta, n')$ where one new customer $i' \in \rho$ is visited and where $n' = n$ if $\rho = r$ and $n' = n + 1$ otherwise. Such a state transition is feasible only when all the following conditions are satisfied.

- To satisfy the limit on the maximum number of jumps, a jump to another solution (case $\rho \neq r$) is allowed only if $n < J_{max}$, and a jump in the backward direction is not allowed if there is only one jump left (case $n = J_{max} - 1$).
- Node i' is the next *feasible* visit after i in solution ρ. More precisely, let $\tau = arr(S)$ be the arrival time at the current state. If $\delta = FWD$ (resp. BWD), then i' is the first customer following i (resp. preceding i) in route ρ such that (1) i' is not visited yet ($i' \in \{1, \ldots, N\} \setminus V$), (2) it is possible to visit i' before its closing date ($\tau' = \tau + tt(i, i', \tau) \leq C_{i'}$), and (3) from i', it is possible to return to node $N + 1$ before T_{max} ($\tau' + tt(i', N + 1, \tau') \leq T_{max}$). Note that as shown in Figure 5.2b, this rule can skip customers in ρ that are already visited in V or cannot be visited due to the time window constraints. In other words, the rule is more flexible than just strictly following solution ρ.
- If the selected route is the same as the current route ($\rho = r$), then the state is feasible only if the direction does not change ($\delta = d$). In other words, moving in an opposite direction on the same sequence, that is making a *reverse jump*, is not allowed. This last rule is introduced to reduce the number of search states but can be omitted (more details on this point are empirically discussed in Section 5.5).

If the transition from state S to state S' is feasible, the arrival time obtained at customer i' by following this transition is $\tau' = \max(\tau + tt(i, i', \tau), O_{i'})$ where $\tau = arr(S)$. If $\tau' < arr(S')$, this means that a path reaching S' at a lower cost has been found. In this case, the cost of S' is updated by $arr(S') \leftarrow \tau'$ and S is set as the new parent of S'.

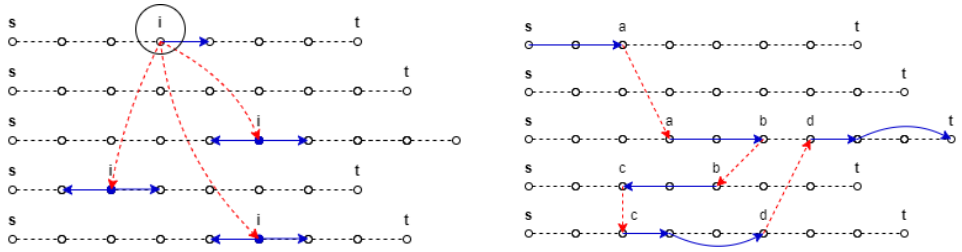


Figure 5.2 – (a) Possible actions (jump in red, direction in blue) given the last visited customer i; (b) a possible sequence generated with 4 jumps (in red) given a pool of 5 sequences of visits

5.3.3 Pseudocode of the route recombination procedure

Algorithm 5.2 details the RR procedure. For the moment, we ignore the lines highlighted in gray and parameter W_{max}. RR takes as an input a pool of P elite solutions and parameter J_{max} defining the maximum number of jumps. Initially, the best-known solution is initialized by computing the total reward provided by each individual solution in the elite pool (Line 1). Then, the algorithm iteratively extends the set of states $\mathcal{S}_l$ at each level l, to generate the set of states $\mathcal{S}_{l+1}$ at level $l+1$. Starting with the P initial states at level 0 (Line 2), the process is repeated until no more feasible state is found (Lines 4-16). For each state S reached at the current level, the algorithm explores all actions (ρ, δ) satisfying the transition rules (Line 7). For each new extended state S', the algorithm computes the earliest time at which S' can be reached and updates $arr(S')$ if needed, together with the parent of S' (Lines 6-10). States at level $l+1$ are pruned if necessary by calling the PRUNE function (Line 11, more details later on this point). If the new feasible state S' is not pruned, it is appended to $\mathcal{S}_{l+1}$. The best reward value found so far is updated if the new state is strictly better (Lines 12-13). Finally, the best state found is returned at the end of the recombination process (Line 17). Strictly speaking, given Lines 12-13 in the pseudocode of Algorithm 5.2, S^* is not a terminal state since it can be shown that it never ends with customer $N+1$ (Line 13); however, the state extension rules always ensure that coming back to node $N+1$ is feasible from S^*.

Algorithm 5.2: `recombine`$(\{r_1, \ldots, r_P\}, J_{max}, W_{max})$

```
1  S* ← preprocessing({r_1, ..., r_P})
2  S_0 ← {S_0^{r_1}, ..., S_0^{r_P}}
3  l ← 0
4  while S_l ≠ ∅ do
5      S_{l+1} ← ∅
6      for each state S = (V, i, r, d, n) ∈ S_l do
7          for each action (ρ, δ) feasible in state S given J_max and the rules
            of Section 5.3.2 do
8              (S', τ') ← extend(S, ρ, δ)                    /* Section 5.3.2 */
9              if τ' < arr(S') then
10                 arr(S') ← τ', parent(S') ← S
11                 S_{l+1} ← prune(S_{l+1}, S')              /* Section 5.3.4 */
12                 if R(S') > R(S*) then
13                     S* ← S'
14     if |S_{l+1}| > W_max then
15         S_{l+1} ← RESTRICT(S_{l+1}, W_max)                /* Section 5.3.5 */
16     l ← l + 1
17 return S*
```

5.3.4 Pruning strategies

The efficiency of dynamic programming algorithms often depends on the capacity to detect and eliminate states that cannot lead to an optimal solution or are unlikely to lead to an optimal solution. In function `prune` used at Line 11, two pruning strategies are considered.

Pruning by bound At each step, given the solutions of the pool and the states already reached, the algorithm disposes of the best-known state S^*. From this, a state S can be pruned if $R(S) + \overline{R}(S) < R(S^*)$, where $\overline{R}(S)$ is an upper bound on the maximum extra score that can be collected from state S to a terminal state. To compute $\overline{R}(S)$, it is possible to find a superset of customers *'visitable'* from state S, denoted by $\overline{V}(S)$, and compute $\overline{R}(S) = \sum_{j \in \overline{V}(S)} R_j$. We propose two ways to find $\overline{V}(S)$ depending on whether state $S = (V, i, r, d, n)$ has reached the maximum number of jumps:

- if S already uses the maximum number of jumps ($n = J_{max}$), the only possible action is to follow the current route r in the forward direction until reaching the end depot, hence we use $\overline{V}(S) = \{j \in r \mid (j \notin V) \wedge (j \text{ follows } i \text{ in route } r)\}$;
- otherwise ($n < J_{max}$), if node i is reached at time τ, then we use $\overline{V}(S) = \{j \in \{1, \ldots, N\} \mid (j \notin V) \wedge (\tau + tt(i, j, \tau) \leq C_j)\}$; this definition of $\overline{V}(S)$ guarantees that the optimal recombined route will not be pruned if both the triangular inequality and the FIFO properties mentioned in Section 3.2.4 are satisfied by tt (intuitively, the former means that the shortest path between two customers is always the direct one; whereas, the latter states that the earlier a transition starts, the earlier it ends).

Stronger bounds could be looked for, by exploiting linear programming models of a relaxed problem, but this is not straightforward as we deal with a black-box time-dependent transition function.

Pruning by dominance Given two states $S_1 = (V, i, r, d, n_1)$ and $S_2 = (V, i, r, d, n_2)$, we say that S_1 *strongly* dominates S_2 if $arr(S_1) \leq arr(S_2)$ and $n_1 < n_2$, which means that S_1 has a lower cost and more remaining jumps than S_2. In this case, state S_2 is not kept in the set of states to be extended. Given the conditions checked, the number of non-dominated states can however remain high. This is why, to reduce the number of states to be explored, we introduce a weaker dominance rule that allows us to compare two states $S_1 = (V, i, r_1, d, n_1)$ and $S_2 = (V, i, r_2, d, n_2)$ even if r_1 is distinct from r_2. We say that S_1 *weakly* dominates S_2 if either $arr(S_1) < arr(S_2)$, or $arr(S_1) = arr(S_2) \ \wedge \ n_1 < n_2$. Intuitively, if two states reach the same node i and visit the same set of nodes V, the weak dominance rule discards the state that has the highest cost *without* taking into account the current route being followed. Note that this dominance rule may prune states from which a better reward could be reached in the end, since state S_2 can be relevant even if $arr(S_1) < arr(S_2)$, especially if it uses fewer jumps than S_1 (*i.e.* $n_1 > n_2$).

5.3.5 Bounded-width recombination

Although the approach restricts the number of states by limiting the maximum number of jumps and by incorporating pruning strategies, the recombination procedure possibly requires a memory size that is exponential in J_{max}

(see Section 5.3.6). To address this issue, the algorithm employs an additional *width-restriction* mechanism that limits the number of states kept at each decision level. This mechanism corresponds to the part highlighted in gray in Algorithm 5.2. Basically, once all states in $\mathcal{S}_{l+1}$ have been generated, the state restriction phase is invoked. If $\mathcal{S}_{l+1}$ contains too many states according to an input parameter W_{max}, the algorithm heuristically selects the most promising ones and discards the others (Lines 14-15). Formally, for each state $S \in \mathcal{S}_{l+1}$, we compute a reward-to-cost ratio $h(S) = R(S) \ / \ arr(S)$. Based on this heuristic value, if $|\mathcal{S}_{l+1}| > W_{max}$, we keep only W_{max} states that have the highest heuristic values. Such a width-restriction strategy is similar to the approach proposed by Gillard & Schaus (2022) for LNS with decision diagrams.

5.3.6 Complexity results

For the theoretical complexity analysis, we only consider the pure dynamic programming algorithm and do not take into account the state pruning strategies.

Proposition 5.1. *The number of feasible search states using k jumps is bounded by $2^k * (NP)^{k+1}$, where P is the number of solutions in the elite pool and N is the number of customers.*

Proof. The statement holds for $k = 0$ because the only transitions that lead to a state with no jump are those that move forward by following one of the P routes of the solution pool. As these routes contain at most N transitions between successive customers (if we omit customer $N + 1$), the number of transitions leading to a state with no jump is less than $N * P$.

Assuming that the statement holds for k jumps, let us show that it also holds for $k + 1$ jumps. There are two different ways to generate a state with $k + 1$ jumps.

- The first way is to make a transition from a state S using k jumps to a new state S' using $k + 1$ jumps. This is done by jumping to another sequence in the pool. Note that each jump can be either forward or backward, resulting in at most $2(P - 1)$ such transitions for each state using k jumps. By using the satisfaction of the property at step k, the set of states T generated in this way has a cardinality bounded by $|T| \leq 2(P - 1) * 2^k * (NP)^{k+1} \leq 2^{k+1} * N^{k+1} * P^{k+2}$.

- The second way is to make transitions from states using $k+1$ jumps to new states using $k+1$ jumps. This is simply done by following the route r and direction d registered in each of the states in T. In the worst case, the number of such transitions is at most $N-1$, hence the number of states with $k+1$ jumps generated in this way is bounded by $|T| * (N-1)$.

By combining the two cases, the number of states using $k+1$ jumps is bounded by $|T| * N \leq 2^{k+1} * (NP)^{k+2}$. As the property holds at step $k+1$, we conclude that it holds for all k. □

Proposition 5.2. *The number of search states explored by the RR procedure is $\mathcal{O}((2NP)^{J_{max}+1})$, where N is the number of customers, P is the size of the elite pool, and J_{max} is the maximum number of jumps.*

Proof. The maximum number of states μ explored by RR can be computed as the sum of the number of states using k jumps where k ranges from 0 to J_{max}. Therefore, we can write $\mu \leq \sum_{k=0}^{J_{max}} (2^k * (NP)^{k+1}) = NP \sum_{k=0}^{J_{max}} (2NP)^k = NP \frac{(2NP)^{J_{max}+1}-1}{2NP-1} \leq (2NP)^{J_{max}+1}$. □

Proposition 5.3. *The number of search states explored by the RR procedure is $\mathcal{O}(NPW_{max})$ where N is the number of customers, P is the size of the elite pool, and W_{max} is the maximum width per level.*

Proof. The algorithm systematically explores states in a level-by-level manner. In the worst case, there are at most N extension levels to consider. At level l, for each state in $\mathcal{S}_l$, the number of possible actions (ρ, δ) is bounded by $2P$ (Line 7) since we can either follow the current route or jump to one of the $P-1$ solutions of the pool in the forward or backward direction. As the width of each level is bounded by W_{max} (Lines 14-15), the number of states explored by Algorithm 5.2 is bounded by $2NPW_{max}$. □

Proposition 5.4. *The number of search states explored by the RR procedure is $\mathcal{O}(min((2NP)^{J_{max}+1}, NPW_{max}))$ where N is the number of customers, P is the size of the elite pool, J_{max} is the maximum number of jumps, and W_{max} is the maximum width per level.*

Proof. A direct consequence of Propositions 5.2 and 5.3. □

Propositions 5.2, 5.3, and 5.4 directly provide the worst-case space complexity of RR. For the worst-case time complexity, we must take into account

the fact that for each transition between two states, RR looks for the next customer i' that can be visited. In the worst case, N customers are considered for each transition. As a result, by assuming that each call to transition function tt takes a time $O(1)$, it suffices to multiply the quantities given in Propositions 5.2, 5.3, and 5.4 by a factor N to get worst-case time complexity results.

From a general point of view, Proposition 5.2 shows that for a fixed value of J_{max}, the RR procedure has a complexity that is only polynomial in N and P. However, when J_{max} is increased, the complexity of RR rapidly grows and follows an exponential pattern. In this case, W_{max} quickly becomes the main factor that limits the complexity of RR, and J_{max} mainly serves as a heuristic for limiting the state exploration process, ensuring that the recombination does not deviate too much from the baseline solutions.

5.4 Usages of the RR procedure

5.4.1 Iterative Route Recombination (IRR)

When considering a fixed value of J_{max}, the generation of a new sequence with the RR procedure typically involves the combination of at most $J_{max}+1$ subsequences. This means that if J_{max} is small, the diversity of the combined sequences explored by RR is limited. However, it is possible to try to overcome this restriction by using an iterative process where after each iteration of RR leading to an improvement, the worst solution in the pool is replaced by the new combined solution. This process continues until a fixed point is reached, *i.e.* until no further improvement is found. In the end, even with a fixed number of jumps J_{max}, it is possible to iteratively create sequences containing more than $J_{max}+1$ subsequences contained in the solutions of the original pool. Indeed, at the first iteration of RR, the resulting combined solution incorporates components from at most $J_{max}+1$ original subsequences, but at the next iterations, the new sequences generated can themselves be combined with other solutions of the pool. The iterative version of RR is referred to as IRR in the following.

5.4.2 RR used for deterministic and non-deterministic problems

As mentioned previously, for standard (TD)OPTWs, RR can be used as a post-optimization step that takes as an input the set of elite solutions produced by the LNS algorithm and tries to produce an improved combined solution. This usage of RR is illustrated in Figure 5.3.

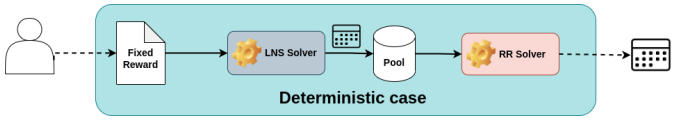


Figure 5.3 – Using RR in a deterministic context without reward uncertainty

In another direction, RR can be used to deal with non-deterministic (TD)OPTWs where there is uncertainty about the actual value of the reward associated with each customer and where updated reward values can be received during an online phase, just before the execution of the solutions. For this, as illustrated in Figure 5.4, we first build a pool of elite solutions during a training (offline) phase. For each training scenario, we change the customer rewards by $dR\%$, and run LNS to collect a pool containing $pool_{train}$ high-quality solutions for the updated scenario. Later on, in the execution (online) phase with new reward information, it is possible to use RR to try and generate a high-quality solution very quickly, based on the sequences available in the training pool.

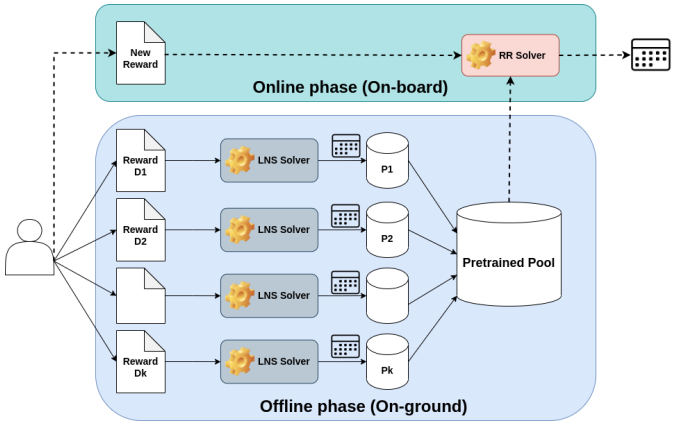


Figure 5.4 – Using RR as an online solver in the case of reward uncertainty

5.5 Experiments

5.5.1 Experimental settings

We carried out experiments on two benchmarks that are summarized in Table 5.3.

dataset	#instances	#nodes	is time-dependent ?
optw-solomon1	29	100	no
optw-solomon2	27	100	no
optw-cordeau1	10	48-288	no
optw-cordeau2	10	48-288	no
ti-singlesat-ttf1.0	108	46-95	no
ti-singlesat-ttf1.5	108	46-95	no
ti-singlesat-ttf2.0	108	46-95	no
td-singlesat-ttf1.0	108	46-95	yes
td-singlesat-ttf1.5	108	46-95	yes
td-singlesat-ttf2.0	108	46-95	yes

Table 5.3 – A summary of benchmark datasets

The first benchmark contains 76 classical OPTW instances from Solomon (1987) and Cordeau et al. (1997) already used in the previous chapter.[1] We recall that in the instances of sets Solomon 2 & Cordeau 2, the time windows are longer than those in the instances of sets Solomon 1 & Cordeau 1.

The second benchmark, called *singlesat*,[2] is a realistic dataset related to optimization for an Earth-observing satellite that can observe a set of ground targets (the customers). This benchmark contains 108 instances divided into three sets (500km, 700km, 800km) corresponding to different altitudes of the satellite. These instances were originally generated by Pralet (2023) for the Time-Dependent Traveling Salesman Problem with Time Windows (TSPTW), therefore there exists a sequence that can visit all the nodes. In order to diversify the experiments, for each instance, we multiply all the transition times by a factor $ttf \in \{1, 1.5, 2\}$ to reduce the number of feasible visits in a solution. Roughly speaking, factor $ttf = 2$ corresponds to a satellite rotating around its center of gravity at half its normal speed. Besides, since the

[1] Instances are available at `https://www.mech.kuleuven.be/en/cib/op`

[2] Instances are available on request

rewards are not available in the original dataset, we generated a random reward for each customer by using a uniform distribution $R \sim U(10, 30)$. Last, the experiments are run over both time-independent and time-dependent variants of the *singlesat* instances, referred to as *ti-singlesat* and *td-singlesat* respectively.[3]

All the experiments are performed on Intel(R) Core(TM) i5-1145G7 @ 2.60GHz processors with 64GB RAM. The implementation is in C++14 and compiled in a Linux environment with gcc 9.4.0. Table 5.4 summarizes the parameters used for these experiments and the values considered in the following. Two parameters rm and CPU_{max} are used for the LNS solver, while two other parameters J_{max} and W_{max} are used to control the complexity of the (I)RR procedure. Parameter rm defining the customer removal ratio of the LNS destroy phase is always set to 0.4. For each instance, we perform a single run of (I)RR since given a pool of elite solutions, the behavior of (I)RR is fully deterministic.

Parameter	Semantics	Values
rm	customer removal ratio (LNS destroy phase)	0.4
CPU_{max}	max. CPU time of each LNS run	$1s, 2s, 5s, 10s, 30s, 60s$
J_{max}	max. number of allowed jumps	$1, 2, 3, 4$
W_{max}	max. number of states explored in each level	$100, 1000, 5000, 10000, \infty$

Table 5.4 – Parameters used in all the experiments

In the following, we evaluate the performance of the RR procedure in two use cases: (1) as a post-optimization step in deterministic cases, and (2) as a solver in uncertain scenarios involving updated rewards.

5.5.2 Deterministic scenarios: using IRR as a post-optimizer

In this experiment, we run LNS for a limited CPU time. We then collect the 10 best solutions found by LNS. After that, we apply IRR over this elite pool to search for an improved solution.

[3]We did not perform experiments on the TDOPTW instances proposed by Verbeeck et al. (2017), for two main reasons. First, for small instances, LNS quickly finds the optimal solution, rendering RR unnecessary. Second, for the large instances, the authors used an additional procedure to modify the original instances so as to obtain known optimal solutions, but the modified instances are not available online.

Overall performance We first compare the results obtained by LNS and LNS-IRR (that uses IRR after LNS) over all instance sets. Here, LNS is run for 1 second (i.e. $CPU_{max} = 1$s) and IRR then uses $J_{max} = 2$ and $W_{max} = \infty$.

For the classical OPTW instances, we report the average gaps in percent compared to the best-known solutions available in the literature. Formally, the gap in percent for each instance is given by

$$gap = 100 * \frac{bk - bf}{bk}$$

where bf is the best profit found by each solver and bk is the best-known value found in the literature.

Table 5.5 shows that using IRR helps improve the average gaps over all instance sets. Precisely, the average mean over all instances is improved from 1.70% to 1.22% after using IRR, with a very short average computation time of 16.20 milliseconds for IRR. For the Solomon2 and Cordeau2 datasets, the execution time of IRR is slightly longer because the instances in these datasets are less constrained on the temporal aspect and the sequences in the elite pool are usually longer (up to 50 customers per sequence).

dataset	Average Gap (%)		t_{IRR}(ms)
	LNS	LNS-IRR	
optw-solomon1	0.00	**0.00**	0.87
optw-solomon2	2.39	**1.76**	37.92
optw-cordeau1	1.90	**1.01**	6.07
optw-cordeau2	4.55	**3.52**	12.10
All	1.70	**1.22**	16.20

Table 5.5 – Results obtained by using IRR as a post-optimization module for the LNS solver over classical OPTW instances ($J_{max} = 2$, $W_{max} = \infty$)

Table 5.6 summarizes the results obtained by LNS and LNS-IRR when dealing with the singlesat instances. Since the best-known solutions are unavailable, we report in the first two columns the best rewards collected by each solver, averaged over all instances of each dataset. The third column reports the time consumed by the IRR procedure. In addition, we measure, for each instance, the reward improvement of LNS-IRR by computing

$$\Delta R = 100 * \frac{R_{LNS-IRR} - R_{LNS}}{R_{LNS}}$$

where R_{LNS} and $R_{LNS-IRR}$ are the best profit found by LNS and LNS-IRR, respectively. Then, in the two following columns, we report ΔR_{avg} and ΔR_{max} as the average and the highest reward improvement over all instances. In the last column, we show the number of instances where IRR helps improve the quality of solutions found by LNS i.e. $\Delta R > 0$.

dataset	Rwd(LNS)	Rwd(LNS-IRR)	t_{IRR}(ms)	ΔR_{avg}(%)	ΔR_{max}(%)	nbImpr
ti-singlesat-ttf1.0	1311.06	**1315.59**	7.94	0.33	2.67	38/108
ti-singlesat-ttf1.5	1082.19	**1089.27**	17.88	0.63	3.36	72/108
ti-singlesat-ttf2.0	897.10	**903.42**	25.30	0.67	3.25	70/108
td-singlesat-ttf1.0	1319.76	**1325.31**	23.29	0.40	3.50	48/108
td-singlesat-ttf1.5	1091.44	**1101.86**	41.35	0.94	3.55	85/108
td-singlesat-ttf2.0	907.95	**917.65**	33.56	1.04	4.44	85/108
Grand mean	1101.59	**1108.85**	24.89	-	-	-

Table 5.6 – Results obtained by using IRR as a post-optimization module for the LNS solver over the singlesat datasets ($J_{max} = 2$, $W_{max} = \infty$)

As before, LNS-IRR outperforms LNS for all instance sets, and IRR only requires a very short time, with an average of 24.89 milliseconds to enhance the performance of LNS. Regarding the last three columns, we observe that IRR leads to higher improvements when dealing with the more constrained sets. In particular, for *ti-singlesat-ttf2.0*, IRR helps improve the best solution for 70 over 108 instances, with an average and the highest reward improvement equal to 0.67% and 3.25%, respectively. For the *td-singlesat-ttf2.0* dataset, the number of improving cases is up to 85 over 108 instances, with an average and the highest reward improvement equal to 1.06% and 4.44%, respectively.

For both benchmarks, results obtained show that using the IRR procedure after LNS is both light and efficient in most cases.

Impact of the elite pool To investigate the impact of the quality of the solutions in the pool on the final solution found by IRR, we analyze the evolution of the grand mean gap (for the classical OPTW datasets) or reward (for the singlesat datasets) over different values of CPU_{max} for LNS, varying from 1 second to 60 seconds.

In Table 5.7a, we observe that on classical OPTW datasets, LNS-IRR using 10 seconds of LNS and 21.64 milliseconds of IRR achieves almost the same average gap as pure LNS running during 60 seconds. As for the singlesat datasets, Table 5.7b shows that LNS-IRR only needs 10 seconds of LNS to get a slightly better average reward than LNS alone running during 60. In both

cases, we remark that LNS-IRR outperforms LNS i.e. LNS-IRR produces better results within a much shorter CPU time.

CPU_{max}(s)	Average Gap (%)			t_{IRR}(ms)
	LNS	LNS-IRR	ΔGap	
1	1.70	**1.22**	0.48	16.20
2	1.44	**1.12**	0.32	15.21
5	1.39	**1.03**	0.36	30.26
10	1.14	**0.88**	0.26	21.64
30	0.91	**0.69**	0.22	15.10
60	0.80	**0.62**	0.18	17.31

(a) OPTW

CPU_{max}(s)	Reward			t_{IRR}(ms)
	LNS	LNS-IRR	ΔR	
1	1101.59	**1108.85**	7.26	24.89
2	1107.03	**1112.94**	5.91	17.15
5	1109.66	**1114.94**	5.28	15.91
10	1111.82	**1116.68**	4.86	15.16
30	1114.81	**1117.99**	3.18	12.80
60	1115.99	**1119.09**	3.10	14.56

(b) singlesat

Table 5.7 – Different CPU_{max} time limits of LNS over OPTW and singlesat instances (IRR with $J_{max} = 2$, $W_{max} = \infty$)

Besides, Tables 5.7a and 5.7b also show that the improvement provided by IRR decreases if the computation time allocated to LNS is higher (see columns ΔGap and ΔR). This is reasonable since with a higher time limit, LNS can converge to a better local optimum, making it harder for IRR to find new improving solutions.

Impact of parameters J_{max} and W_{max} The complexity of RR is controlled by parameters, J_{max} and W_{max}. To study the impact of these parameters, we report results obtained by LNS-IRR for $J_{max} \in \{0, 1, 2, 3, 4\}$ ($J_{max} = 0$ means that we do not use IRR after running LNS) and $W_{max} \in \{100, 1000, 5000, 10000, \infty\}$. The timeout duration for IRR is set to 5 minutes.

As shown in Table 5.8, when $W_{max} = \infty$, increasing J_{max} can lead to higher improvements in solution quality. Indeed, for the classical OPTW instances, the average gap decreases from 1.70% without jumps to 1.05% with 3 jumps. However, the average gap obtained with 4 jumps is worse than the one with 3 jumps. This is because, on some instances, IRR using

$J_{max} = 4$ reaches the time limit and does not complete the recombination process. On this point, we additionally report in column #t.o. the number of instances over which a timeout of IRR occurs given the time limit of 5 minutes. Similar results are observed with both the OPTW and singlesat datasets: when we do not constrain the width of each level ($W_{max} = \infty$), IRR requires much more time to finish when J_{max} increases. This is due to the exponential increase in the number of states to consider during dynamic programming.

J_{max}	W_{max}	optw			singlesat		
		AvgGap%(LNS-IRR)	t_{IRR}(ms)	#t.o	AvgRwd(LNS-IRR)	t_{IRR}(ms)	#t.o
0	∞	1.70	0	0	1101.59	0	0
1	∞	1.63	1.48	0	1103.35	2.77	0
2	∞	1.22	16.20	0	1108.85	24.89	0
3	∞	1.05	1929.95	0	1112.26	554.14	0
4	∞	1.24	81370.61	17/76	**1113.75**	37561.21	31/648
4	10000	**0.89**	2321.45	0	1113.65	6694.53	0
4	5000	0.95	828.19	0	1113.13	3589.63	0
4	1000	1.11	129.03	0	1109.25	622.03	0
4	100	1.47	11.79	0	1103.49	43.80	0

Table 5.8 – Impact of parameters J_{max} and W_{max} ($CPU_{max} = 1$s)

However, limiting W_{max} can help overcome this issue. With 4 jumps, for the singlesat datasets, t_{IRR} decreases from 37 seconds to about 6 seconds after setting $W_{max} = 10000$ and there is no instance over which a timeout is reached. For the classical OPTW instances, the speed-up is even more significant: t_{IRR} decreases from 81 seconds to about 2 seconds with $W_{max} = 10000$, while the average gap is even better than when using configuration $J_{max} = 3, W_{max} = \infty$ (0.89% compared to 1.05%). Obviously, for smaller values of W_{max}, IRR is faster but provides lower rewards, since more states are pruned by the state restriction phase.

Effectiveness of the iterative recombinations Table 5.9 gives the total number of RR iterations done over all the instances of each dataset. For instance, column #RR=1 counts the number of instances for which the first iteration of RR brings no improvement. For the Solomon1 and Cordeau1 datasets, since the best solution provided by LNS is already near-optimal, we always have #RR≤ 2. For the singlesat datasets, there exist many instances where RR is applied more than 3 times. In these cases, combined solutions potentially involve the combination of more than three (or four) sequences of the original pool, even though the maximum number of jumps J_{max} is set to 2 (or 3).

dataset	#instances	#RR ($J_{max} = 2$)						#RR ($J_{max} = 3$)					
		1	2	3	4	5	6	1	2	3	4	5	6
optw-solomon 1	29	29	0	0	0	0	0	29	0	0	0	0	0
optw-solomon 2	27	12	10	4	1	0	0	7	14	6	0	0	0
optw-cordeau 1	10	4	6	0	0	0	0	4	6	0	0	0	0
optw-cordeau 2	10	4	4	1	1	0	0	4	6	0	0	0	0
ti-singlesat-ttf1.0	108	70	27	10	1	0	0	60	27	19	2	0	0
ti-singlesat-ttf1.5	108	36	48	19	3	2	0	31	54	11	7	4	1
ti-singlesat-ttf2.0	108	38	52	15	3	0	0	32	54	18	4	0	0
td-singlesat-ttf1.0	108	60	29	15	3	1	0	63	25	16	3	1	0
td-singlesat-ttf1.5	108	23	55	28	2	0	0	17	55	27	8	1	0
td-singlesat-ttf2.0	108	23	59	25	1	0	0	23	63	20	2	0	0

Table 5.9 – Number of RR iterations in the iterative recombination process ($CPU_{max} = 1$s, $W_{max} = \infty$)

Impact of the weak dominance rule Furthermore, to measure the strength of the pruning strategies, we run again the IRR procedure by replacing the weak dominance rule with the strong dominance rule. The two variants obtained are referred to as *IRR-weakDom* and *IRR-strongDom* respectively, IRR-weakDom being our default configuration. As indicated in Table 5.10, for the singlesat datasets, IRR-weakDom with $J_{max} = 4$ and $W_{max} = \infty$ reaches its time limit for 31 instances, compared to 99 instances for IRR-strongDom. Indeed, RR with the strong dominance rule requires exploring a higher number of states during dynamic programming. Moreover, we can observe that the execution time of IRR-strongDom and the number of timeouts significantly increase when using more than 2 jumps. Yet again, setting W_{max} can help overcome this issue (see the last column in Table 5.10).

dataset	IRR variant	$J_{max} = 2$ $W_{max} = \infty$		$J_{max} = 3$ $W_{max} = \infty$		$J_{max} = 4$ $W_{max} = \infty$		$J_{max} = 4$ $W_{max} = 10000$	
		t_{IRR}(ms)	#t.o	t_{IRR}(ms)	#t.o	t_{IRR}(ms)	#t.o	t_{IRR}(ms)	#t.o
optw	weakDom	16	0	1929	0	81370	17/76	2321	0
	strongDom	52	0	8149	0	105787	23/76	7180	0
	weakDom + revJump	135	0	62428	13/76	138709	34/76	6639	0
	strongDom + revJump	938	0	66670	14/76	137345	32/76	112208	0
singlesat	weakDom	24	0	554	0	37561	31/648	6687	0
	strongDom	61	0	1391	0	81575	99/648	12415	0
	weakDom + revJump	48	0	2486	0	140199	208/648	10581	0
	strongDom + revJump	127	0	14862	3/648	161587	258/648	15463	0

Table 5.10 – Impact of *weakDom* and *reverseJump* on the computation time of IRR ($CPU_{max} = 1$s)

Impact of reverse jumps As mentioned earlier in Section 5.3.2, we call *reverse jump* a jump within the same sequence but in an opposite direction.

The state extension rules forbid the reverse jumps, thereby reducing the number of search states. To illustrate this point, we compare the standard configuration with the one that allows reverse jumps (by omitting the last state extension rule). As reported in Table 5.10, the variants allowing reverse jumps, denoted shortly as IRR-revJump, require a larger amount of time in all cases. For example, for the singlesat datasets, IRR-revJump is about up to four times slower with $J_{max} = 4$ and reaches the timeout for more than 200 instances. Similarly, for the classical OPTW instances, IRR-revJump is slower and reaches the timeout for several instances, even with $J_{max} = 3$.

Instance	LNS	LNS-IRR weakDom			LNS-IRR strongDom			LNS-IRR weakDom-revJump			LNS-IRR strongDom-revJump		
	Rwd	Rwd	t_{IRR}(ms)	#RR	Rwd	t_{IRR}(ms)	#RR	Rwd	t_{IRR}(ms)	#RR	Rwd	t_{IRR}(ms)	#RR
optw-pr15	642	676	746.68	2	678	1409.79	3	698	3681.02	5	**703**	248592.00	6
optw-pr18	528	528	70.28	1	528	99.04	1	**531**	251.48	2	**531**	740.19	2
optw-pr20	621	628	165.44	2	629	414.09	2	**637**	10647.00	4	629	14856.00	2

Table 5.11 – Impact of *strongDom* and *reverseJump* for several instances ($CPU_{max} = 1$s, $J_{max} = 3$, $W_{max} = \infty$)

Essentially, reverse jumps are able to explore the sequences produced by the state-of-the-art *2-opt* operator and can be useful in some cases. As shown in Table 5.11, this is empirically observed for several optw-cordeau2 instances (e.g., instances pr15, pr18, and pr20) where IRR-revJump gives the best solution quality compared to other variants. However, IRR-revJump requires a much larger execution time, which is why we deactivate the reverse jumps in the baseline dynamic programming algorithm proposed.

Summary Globally, the results obtained show that using iterative RR (IRR) in a post-optimization phase can quickly improve the quality of solutions provided by a standard LNS solver. IRR is very fast with a small value of J_{max}. While using a larger value for J_{max} leads to further improvements, it also requires much more time. To overcome this issue, setting W_{max} becomes essential. Moreover, using the weak dominance test and omitting the reverse jumps help speed up the recombination phase, while inducing a slight decrease in the solution quality in some cases.

5.5.3 Uncertain scenarios: using RR as an online solver

The general scheme of this experiment corresponds to the approach provided earlier in Figure 5.4, which involves uncertain reward values that can be

updated at the last minute. In the following, we use a fixed configuration where $J_{max} = 2$ and $W_{max} = \infty$, and we run RR only once instead of using the iterative recombination version, to avoid modifying the sequences in the training pool. In other words, we use RR instead of IRR. The remaining parameters of the experiments are given in Table 5.12.

Parameter	Semantics	Values
dR	perturbation ratio for the rewards	$10\%, 20\%, 30\%, 50\%$
ns_{train}	number of training scenarios (*i.e.* number of reward scenarios)	$5, 10$
$pool_{train}$	max. number of solutions generated for each training scenario	$5, 10$

Table 5.12 – Parameters used in uncertain scenarios

Basically, we generate ns_{train} reward scenarios based on a perturbation ratio referred to as dR. More specifically, to generate one reward scenario, we randomly chose, for each customer i whose original reward is R_i in an instance, a reward in interval $[R_i * (1 - dR), R_i * (1 + dR)]$ following a uniform distribution. Then, for each reward scenario, we generate a pool of $pool_{train}$ elite solutions. After that, the online reward scenario is also built by choosing a reward $\tilde{R}_i$ following the same uniform distribution $\tilde{R}_i \sim U(R_i * (1 - dR), R_i * (1 + dR))$. With this configuration, the online rewards do not necessarily correspond to the expected mean of the rewards chosen in the scenarios.

Overall performance In this experiment, we set $dR = 20\%$, $ns_{train} = 10$, and $pool_{train} = 5$. As a result, the pool contains at most $ns_{train} * pool_{train} = 50$ sequences.

Table 5.13 compares the best-combined solutions obtained by RR and the best solution found by a run of LNS during 5 seconds given the online reward scenario. The online scenarios are created based on both the classical OPTW instances and the singlesat instances, but with $dR\%$ reward perturbation to introduce variability. Due to the unavailability of best-known solutions for these test instances, in the first two columns of the upper table, we report the average rewards obtained by each solver on the online reward scenario. In the next two columns, we report tf_{LNS}, the average time at which LNS produces its best solution, and t_{RR}, the computation time consumed by RR given the training pool. Besides, we also compute the reward gap (in %)

between two solvers, which is given by

$$\Delta R = 100 * \frac{R_{RR} - R_{LNS}}{R_{LNS}}$$

where R_{RR} and R_{LNS} are the total rewards provided by RR and LNS, respectively. This means that RR achieves a better solution quality if $\Delta R > 0$.

dataset	Rwd		Time(ms)	
	LNS	RR	tf_{LNS}(ms)	t_{RR}(ms)
optw-solomon1	**299.69**	**299.69**	105.99	**8.14**
optw-solomon2	905.04	**910.78**	**1915.68**	3375.48
optw-cordeau1	368.40	**372.50**	1876.66	**156.70**
optw-cordeau2	411.70	**418.60**	1791.92	**295.53**
optw-mean	495.05	**500.97**	1422.56	**958.96**
ti-singlesat-ttf1.0	1200.64	**1209.37**	1860.90	**103.99**
ti-singlesat-ttf1.5	1003.91	**1013.16**	2329.27	**400.76**
ti-singlesat-ttf2.0	841.07	**846.40**	1865.40	**408.91**
td-singlesat-ttf1.0	1213.16	**1221.11**	2058.36	**96.39**
td-singlesat-ttf1.5	1018.14	**1029.94**	2004.76	**268.49**
td-singlesat-ttf2.0	854.09	**862.17**	2082.80	**338.93**
singlesat-mean	1021.83	**1030.36**	2033.58	**269.58**

dataset	RR wins	RR=LNS	LNS wins	ΔR(%)		
				min	avg	max
optw-solomon1	0	**29**	0	0.00	0.00	0.00
optw-solomon2	**18**	6	3	-0.46	0.54	2.38
optw-cordeau1	**5**	**5**	0	0.00	0.92	2.88
optw-cordeau2	**7**	2	1	-0.20	1.48	5.32
optw-mean	-	-	-	-0.08	1.18	3.17
ti-singlesat-ttf1.0	**79**	28	1	-0.68	0.72	2.91
ti-singlesat-ttf1.5	**92**	12	4	-0.28	0.89	2.98
ti-singlesat-ttf2.0	**75**	26	7	-0.73	0.58	2.53
td-singlesat-ttf1.0	**77**	30	1	-0.39	0.63	2.41
td-singlesat-ttf1.5	**97**	7	4	-0.24	1.14	3.16
td-singlesat-ttf2.0	**85**	20	3	-0.61	0.88	5.46
singlesat-mean	-	-	-	-0.49	0.81	3.24

Table 5.13 – Comparison of results obtained by LNS and RR for a test scenario involving a reward perturbation of 20% (CPU_{max} = 5s, $J_{max} = 2$, $W_{max} = \infty$, $ns_{train} = 10$, $pool_{train} = 5$)

Once again, the results obtained indicate that RR outperforms LNS while requiring a lower computation time. Indeed, with the singlesat datasets, RR takes only 269.58 milliseconds on average and achieves a better grand mean (1030.36) than LNS alone run during 5 seconds (1021.83). This is also true for the classical OPTW datasets, where RR only requires 958.96 milliseconds on average. It is worth noting that there still exist a few cases where LNS obtains better solutions, but the gap compared to RR is relatively small. For example, for dataset *ti-singlesat-ttf2.0*, RR is only less effective in 7 out of 108

cases, the worst gap being just -0.73% compared to the solution found by LNS. Meanwhile, in the time-dependent version (*td-singlesat-ttf2.0*), there are only 3 instances where LNS is better than RR and the worst gap being just -0.61%.

Regarding the execution time (column t_{RR}, Table 5.13), RR consumes more time for the *optw-solomon2* instances, mainly because these instances are less constrained (the optimal solution visits around 50 over 100 customers). This leads to a larger number of possible combinations, resulting in an increase of processing times. Yet globally, these findings demonstrate that RR can effectively adapt the solutions to small changes in the rewards with the use of a pre-trained elite pool.

Impact of *weakDom* and *revJump* Once again, we aim to study the effectiveness of the proposed pruning strategies. Here, we evaluate these pruning strategies with $J_{max} = 2$, but dealing with a larger pool of solutions (50 solutions in the pool). Table 5.14 details the average rewards and execution times obtained with different variants of RR.

As before, using the exact dominance rule (RR-strongDom) instead of the weak version (RR-weakDom) gives a better solution quality, yet requires more time. In terms of runtime, the variant using both strongDom and revJump is the slowest version over both datasets (*e.g.*, up to 109 seconds for *optw-solomon2*). Overall, RR-weakDom is the fastest version, while RR using strongDom (with or without revJump) appears to be better in terms of solution quality.

dataset	RR-weakDom		RR-strongDom		RR-weakDom-revJump		RR-strongDom-revJump	
	Rwd	t(ms)	Rwd	t(ms)	Rwd	t(ms)	Rwd	t(ms)
optw-solomon1	**299.69**	**8.14**	**299.69**	11.40	**299.69**	17.11	**299.69**	13.84
optw-solomon2	910.78	**3375.48**	**911.00**	4327.26	885.30	24056.81	870.56	109487.10
optw-cordeau1	**372.50**	**156.70**	**372.50**	312.51	**372.50**	459.23	372.40	469.57
optw-cordeau2	418.60	**295.53**	**420.70**	715.02	419.90	2864.23	419.40	1972.55
optw-mean	500.39	**958.96**	**500.97**	1341.55	492.68	6489.14	490.51	27985.77
ti-singlesat-ttf1.0	1209.37	**103.99**	1212.67	781.40	1209.42	198.86	**1213.56**	1171.37
ti-singlesat-ttf1.5	1013.16	**400.76**	1014.47	1546.66	1011.68	834.61	**1014.57**	2546.70
ti-singlesat-ttf2.0	846.40	**408.91**	**846.78**	952.48	846.34	728.47	846.63	1523.38
td-singlesat-ttf1.0	1221.11	**96.39**	**1225.56**	788.31	1219.79	176.41	1225.34	1299.59
td-singlesat-ttf1.5	1029.94	**268.49**	**1031.47**	1438.11	1029.47	495.45	1031.44	2429.55
td-singlesat-ttf2.0	862.17	**338.93**	862.54	979.55	862.48	596.31	**862.70**	1407.89
singlesat-mean	1030.36	**269.58**	1032.25	1081.09	1029.86	505.02	**1032.37**	1248.22

Table 5.14 – Impact of *reverseJump* and *weakDom* ($CPU_{max} = 5$s, $J_{max} = 2$, $W_{max} = \infty$, $ns_{train} = 10$, $pool_{train} = 5$).

Impact of the training pool It is also natural to inquire about the number of training scenarios and solutions per scenario required to make RR efficient enough during the online phase.

Table 5.15 shows that the configuration using a larger pool produces a better average reward, though the difference is not significant. On the other hand, reducing the number of solutions in the pool by using either fewer scenarios or fewer sequences per scenario significantly reduces the execution time of RR without deteriorating the solution quality.

Indeed, for the singlesat instances and $ns_{train} = 10$, generating 5 sequences per scenario yields a similar solution quality compared to using 10 sequences per scenario (average rewards equal to 1030.36 and 1030.79 respectively), while being four times faster (269.58ms compared to 1318.42ms). For the classical OPTW instances, the speed-up ratio is even higher (about ten times).

ns_{train}	$pool_{train}$	OPTW		singlesat	
		Rwd(RR)	t_{RR}(ms)	Rwd(RR)	t_{RR}(ms)
5	5	499.15	136.65	1028.44	65.35
5	10	499.93	1230.83	1029.02	357.12
10	5	500.97	958.96	1030.36	269.58
10	10	501.09	11232.44	1030.79	1318.42

Table 5.15 – Impact of the number of solutions in the training pool ($CPU_{max} = 5$s, $J_{max} = 2$, $W_{max} = \infty$)

From a general point of view, having more solutions in the pool results in a longer execution time for RR, while having fewer solutions leads to a lower reward in the end. It is however counterproductive to keep too many solutions in the elite pool because this increases the execution time of RR without significantly improving the solution.

Impact of reward perturbations Last, to assess the adaptation capacity of the RR module, we run additional tests with different levels of reward perturbations.

From Table 5.16, we observe that RR becomes less effective when the reward perturbation is large, e.g., when dR is set to 50%. Intuitively, this is because the scenarios used to build the solutions in the training pool become too distinct from the scenario to consider during the online phase.

$dR(\%)$	meanRwd-OPTW		meanRwd-singlesat	
	LNS	RR	LNS	RR
10	512.90	**518.80**	1039.29	**1048.44**
20	495.05	**500.97**	1021.83	**1030.36**
30	547.31	**552.04**	1049.41	**1055.87**
50	**546.83**	544.10	**1045.12**	1044.14

Table 5.16 – Adaptation of RR with different perturbation ratios (CPU_{max} = 5s, $J_{max} = 2$, $W_{max} = \infty$, $ns_{train} = 10$, $pool_{train} = 5$)

Summary The experimental results show that for scenarios involving uncertain rewards, RR can be effectively used as a fast online solver working on a pre-trained pool of solutions. The size of the pool of solutions must not be too large to keep short computation times, and RR is less effective with large reward perturbations.

5.6 Related works and discussion

Deterministic (TD)OPTW In the literature, various approaches are introduced to solve (TD)OPTW (i.e. with or without time dependent transition times) in a deterministic case (see Section 3.2 for more details). With regards to these existing works, the approach proposed in this chapter is a post-optimization step that can take as an input good-quality solutions provided by any of the existing algorithms.

(TD)OPTW with uncertain rewards In this work, we also consider problems in which there is uncertainty about the precise values of the rewards, and where updated reward values can be received at the last minute.

From a general point of view, a few contributions were proposed in the literature to solve orienteering problems with uncertain profits. One contribution is a work by Ilhan et al. (2008), that considers the orienteering problem with stochastic profits where the goal is to maximize the probability to get a reward greater than a given value. In this case, a unique solution is built beforehand. Other authors considered orienteering problems where customers are present with a certain probability, the goal being then to maximize the total expected reward. For this, Angelelli et al. (2017) developed mathematical programming models and matheuristics, Zhang et al.

(2018) introduced a genetic algorithm optimizing the expected profit and the expected travel cost, and Chou et al. (2021) defined a tabu search algorithm. In all these works, a unique solution is built beforehand. In another direction, Evers et al. (2014) and Verbeeck et al. (2016) considered problems where there is uncertainty about the transition times, but we focus here on profit uncertainty.

This configuration is motivated by a kind of (TD)OPTW under uncertainty related to the satellite observation scheduling problem presented in Chapter 3. In this problem, the vehicle is an observation satellite and the customers are ground targets that should be observed. Each ground target can only be observed during the time window where the satellite overflies that target. The (time-dependent) minimum transition time between the visits of targets i and j corresponds to the time required by the satellite to move around its center of gravity from a configuration where it is pointed to target i to a configuration where it is pointed to target j. In this context, the reward associated with each customer depends on the cloud cover, since images full of clouds bring a lower reward compared to images of ground targets benefiting from clear sky conditions. The objective is then to select and schedule candidate acquisitions in order to maximize the total reward collected. In practice, the problem becomes more challenging due to the uncertainty about the cloud cover conditions.

As for the literature related to Earth observation satellites, several methods were proposed to deal with the uncertainty about the cloud cover, including stochastic optimization methods that associate a probability of success with each observation and possibly plan multiple observations of the same target (Wang et al. (2015)), multi-stage optimization methods that anticipate the possibility of generating new observation tasks to react to the actual weather conditions (Valicka et al. (2019)), as well as robust optimization approaches that exploit an uncertainty budget (Wang et al. (2015, 2019)). Besides, some approaches use the Markov Decision Process framework (Bensana et al. (1999b)) or deep reinforcement learning (Hadj-Salah et al. (2019); Lam et al. (2019)) to learn observation selection rules. However, none of these approaches exploits the last-minute weather data. For this, onboard decision-making techniques were proposed, mainly to adapt a baseline plan given updated weather data obtained just before executing the plan (Beaumet et al. (2011); Chien et al. (2014); Pralet et al. (2019)). But again, a unique baseline plan is considered in these works.

Contributions We employ the route recombination terminology by analogy with the post-processing module of the I3CH algorithm proposed by Hu & Lim (2014). With regards to the works discussed previously as well as the principle of solution merging using complete techniques presented in Section 2.3.1, the techniques proposed bring several contributions.

- First, the RR method proposed is based on a new dynamic programming algorithm that is able to efficiently explore various combinations of subsequences of visits involved in a set of baseline solutions. This differs from existing iterative methods that only explore a path between two baseline solutions, and from existing techniques that use mathematical programming to combine solutions. The dynamic programming algorithm proposed is enhanced with pruning strategies that allow the search space to be reduced. As shown in the experimental results, the algorithm obtained is both fast and efficient on standard OPTW benchmarks.

- Second, to the best of our knowledge, the approach proposed is the first route recombination algorithm that is applied to both time-independent and time-dependent problems, especially contrarily to recombination methods based on the mathematical programming model introduced by Hu & Lim (2014). Moreover, in the case of time-dependent problems, RR does not require the transition function to be piecewise linear.

- Third, the recombination method proposed can combine up to k subsequences of customer visits involved in the whole set of baseline solutions, unlike existing *path relinking* methods that only combine two solutions at a time. This broader exploration can yield larger improvements in solution quality. On this point, RR is inspired by the *k-opt moves* used for the Traveling Salesman Problem and its variants (Helsgaun (2017); Tinós et al. (2018)). Similiarly to k-opt, RR is able to traverse subsequences of visits both in the forward and backward directions. But unlike *k-opt moves* that work on a single solution, RR tries to find the best combination of multiple sequences of visits. Moreover, RR is able to deal with a selection problem where only a subset of customers is visited and where the goal is to maximize the total reward.

- Fourth, we provide worst-case time and space complexity results for the RR procedure, showing that the complexity of RR can be fully

controlled based on only two parameters, namely J_{max} the maximum number of subsequences that can be combined and W_{max} the maximum number of states maintained by the dynamic programming procedure at each step. With the latter parameter, each application of RR explores a number of states that is only linear in the number of customers.

- Fifth, the RR procedure is applied not only to a deterministic context i.e. to post-optimize a set of elite (TD)OPTW solutions, but also to non-deterministic scenarios involving uncertainty about the actual reward provided by each customer visit. For such scenarios, baseline solutions can first be produced during an offline phase for diverse rewards, and the RR module can then exploit these baseline solutions to try and produce a good sequence of visits given the last-known reward values, during an online phase. In particular, for the satellite benchmark, the idea is to build baseline solutions some hours in advance using randomized cloud cover scenarios, and quickly adapt the plan at the last minute given the updated weather forecast, possibly directly onboard the satellite that may have very limited computational capabilities (Figure 3.5).

5.7 Conclusion

This chapter presented a Route Recombination (RR) method applicable to both deterministic and non-deterministic (TD)OPTWs. This method exploits a pool of elite solutions and searches for the best recombination of subsequences of visits available in the pool. The algorithm proposed uses a dynamic programming approach whose complexity is controlled by two parameters, namely J_{max} the maximum number of jumps between baseline solutions, and W_{max} the maximum width of the dynamic programming process. Moreover, pruning rules were introduced to limit the number of states to consider. The experiments performed show that RR is very efficient in various situations. In deterministic cases, RR can be used as a lightweight post-optimization component for a standard incomplete solver. It can also be used to deal with uncertain rewards, by generating a pool of elite solutions from a set of training scenarios. Our experimental results indicate that, with appropriate parameter selections, RR outperforms a standard LNS solver while requiring a shorter CPU time.

CHAPTER 6

A generic framework for solving complex routing problems

In the previous chapters, several hybrid optimization methods have been introduced to deal with OPTWs and TDOPTWs. However, in practice, routing problems with profits may involve many other specifications. This is why in this chapter, we focus on complex scenarios possibly involving optional customers, multiple vehicles, multiple time windows for each customer, multiple knapsack constraints (e.g., capacity constraints), and/or time-dependent transition times modeling traffic conditions (e.g., congestion or rush hours) varying over time. For each problem variant, dedicated powerful heuristic, local search, and metaheuristic algorithms have been defined in the literature, especially in order to solve large-size instances. However, this raises some issues because each time a new problem is introduced, the powerful algorithms defined for the baseline problems need to be revised.

To avoid developing new specific algorithms for each problem variant, we present a hybrid approach for solving complex routing problems involving optional customers, while making an effort to minimize the potential performance loss associated with an increase in genericity. Precisely, we propose a modular architecture exploiting (1) on one side a sub-module managing all customer selection decisions, and (2) on the other side a sub-module efficiently managing customer sequencing decisions. On top of that, a generic solver interface can freely choose a specific metaheuristic while interacting with the low-level modules to search for solutions.

The rest of this chapter is organized as follows. First, the problem con-

sidered is formally defined in Section 6.1. Then, the generic framework is described in Section 6.2. After that, an implementation of two sub-modules is detailed in Section 6.3. A metaheuristic using this framework is presented in Section 6.4. The experiments presented in Section 6.5 show the efficiency and genericity of the approach proposed. Last, related works are discussed in Section 6.7, and the contribution is summarized in Section 6.8.

6.1 Complex orienteering problem formulation

To define the complex orienteering problem considered, we recall several basic notations introduced in Chapter 3. Formally, we consider here a fleet of vehicles $v \in \{1, \ldots, M\}$ and a set of customers $i \in \{1, \ldots, N\}$, plus two fictitious customers numbered 0 and $N+1$ that respectively represent the *source* depot from which each vehicle starts and the *sink* depot at which each vehicle must finish its tour. A reward R_i and possibly multiple successive and non-overlapping time windows $[O_{iw}, C_{iw}], w \in \{1, \ldots, W_i\}$ are associated with each customer $i \in \{0, \ldots, N+1\}$. By convention, $R_0 = R_{N+1} = 0$, $W_0 = W_{N+1} = 1$, and $[O_{0,0}, C_{0,0}] = [O_{N+1,0}, C_{N+1,0}] = [0, T_{max}]$, where T_{max} is the predefined limited time budget for each vehicle v.

In this formulation, we use a black-box transition function $tt(i, j, s_i)$ to cover both time-independent and time-dependent variants of the transition times between two distinct customers i and j when the transition starts at time s_i. For the sake of simplification, we assume that a visit duration of customer i is already included in the transition time from customer i to customer j. We also assume that tt returns only positive values i.e. $tt(i, j, s_i) > 0$.

A classic orienteering problem can be formulated as a mixed integer (non-linear) program using the following variables.

- $\forall i \in \{0, \ldots, N+1\}\ \forall v \in \{1, \ldots, M\}\ \ x_{iv} \in \{0, 1\}$: a binary decision variable taking value 1 if and only if customer i is visited by vehicle v;
- $\forall i \in \{0, \ldots, N\}\ \forall j \in \{1, \ldots, N+1\}\ \forall v \in \{1, \ldots, M\}\ y_{ijv} \in \{0, 1\}$: a binary decision variable taking value 1 if and only if customer i is immediately followed by customer $j \neq i$ in the sequence of visits of vehicle v;

- $\forall i \in \{0, \ldots, N+1\}\ \forall v \in \{1, \ldots, M\}\ s_{iv} \in [O_{i1}, C_{iW_i}]$: a continuous variable denoting the start of the visit at customer i by vehicle v, that is bounded by the start of the first time window of customer i and the end of its last time window.

Together with temporal constraints, several complex variants of this problem can involve additional side constraints in terms of customer selection. As an example, one may extend the OP with capacity constraints like in the capacitated vehicle routing problems. In such a case, each customer has a certain demand c_i and every vehicle has a restricted capacity C_v^{max} to serve these customers. Then, these capacity constraints can be formally represented as $\sum_{i=1}^{N} c_i x_{iv} \leq C_v^{max}\ \forall v \in \{1, \ldots, M\}$. In another direction, one may also have a constraint defining a minimum reward that must be collected, i.e., $\sum_{v=1}^{M} \sum_{i=1}^{N} R_i x_{iv} \geq LB$, as previously mentioned in Section 4.3.1. From a general point of view, these constraints can be expressed in the form of z linear pseudo-Boolean constraints, i.e. $\sum_{i,v} e_{ivz} x_{iv} \sim E_z$ where $\sim \in \{\leq, \geq\}$, $x_{iv} \in \{0, 1\}$, and $e_{ivz}, E_z \in \mathbb{N}$ for $z \in \{1, \ldots, Z\}$. Therefore, to cover all these points in a generic formulation, we introduce additional notations as follows.

- $\forall z \in \{1, \ldots, Z\}\ c_z$: a linear pseudo-Boolean constraint defined over binary variables x_{iv};

- $\forall i \in \{0, \ldots, N+1\}\ \forall v \in \{1, \ldots, M\}\ \forall z \in \{1, \ldots, Z\}\ e_{ivz} \in \mathbb{N}$: the integer coefficient associated with decision variable x_{iv} in linear constraint c_z;

- $E_z \in \mathbb{N}$: the right-hand side value associated with linear constraint c_z.

The model of the complex orienteering problem is described step-by-step as follows.

$$\max \sum_{v=1}^{M} \sum_{i=1}^{N} R_i x_{iv} \tag{6.1}$$

The objective function (Equation 6.1) maximizes the total reward collected by all vehicles.

$$\sum_{j=1}^{N+1} y_{0jv} = \sum_{i=0}^{N} y_{i(N+1)v} = 1; \quad \forall v \in \{1, \ldots, M\} \tag{6.2}$$

$$\sum_{v=1}^{M} x_{kv} \leq 1; \quad \forall k \in \{1, \ldots, N\} \tag{6.3}$$

$$\sum_{i=0}^{N} y_{ikv} = \sum_{j=1}^{N+1} y_{kjv} = x_{kv}; \quad \forall k \in \{1, \ldots, N\}; \ \forall v \in \{1, \ldots, M\} \tag{6.4}$$

Constraints 6.2 guarantee that all vehicles start and end at the depot nodes. Constraints 6.3 guarantee that each node is visited at most once by all vehicles in the case of the multi-vehicle variant. Constraints 6.4 ensure the consistency of the visit sequence of each vehicle, meaning that if a customer is visited by a vehicle, it is preceded and followed by exactly one other customer in the same visit sequence.

$$\begin{aligned}(y_{ijv} = 1) \rightarrow (s_{iv} + tt(i, j, s_{iv}) \leq s_{jv}); \\ \forall i \in \{0, \ldots, N\}; \ \forall j \in \{1, \ldots, N+1\}; \ \forall v \in \{1, \ldots, M\}\end{aligned} \tag{6.5}$$

$$\begin{aligned}\exists w \in \{1, \ldots, W_i\} \ : \ O_{iw} \leq s_{iv} \leq C_{iw}; \\ \forall i \in \{1, \ldots, N\}; \ \forall v \in \{1, \ldots, M\}\end{aligned} \tag{6.6}$$

Constraints 6.5 define the start time of each visit and Constraints 6.6 force the start of the service at a customer to be included in one of the predefined time windows. More precisely, in Constraints 6.5, if the vehicle makes a transition from customer i to customer j, then the start time of the visit at customer j should not be earlier than the start of the service at customer i plus the travel time between i and j. On the other hand, if the vehicle does not travel between i and j, then there is no direct constraint between s_{iv} and s_{jv}. It is also important to emphasize that the linearity of this formulation strongly depends on whether transition function tt is piecewise linear. In the case of a linear time-dependent transition function, we refer to the work of Verbeeck et al. (2013) for a precise MIP formulation.

$$\sum_{i=1}^{N} \sum_{v=1}^{M} e_{ivz} x_{iv} \sim E_z; \quad \sim \in \{\leq, \geq\}; \quad \forall z \in \{1, \ldots, Z\} \tag{6.7}$$

Constraints 6.7 describe linear side constraints in terms of customer selection. Such constraints can be used to formalize several variants of the OP. As an example, Constraints 6.7 using relation "$\leq$" can be used to express the capacity constraints (e.g., Selective VRPTW (Boussier et al. (2007))) or budget limitations (e.g., Tourist Trip Decision Problem (Souffriau (2010))). Meanwhile, Constraints 6.7 using relation "$\geq$" are useful for setting a lower bound on the reward collected (e.g., Prize-Collecting TSP (Balas (1989))) or expressing the selection of mandatory customers (e.g., OP with compulsory vertices (Gendreau et al. (1998a))).

In particular, the *multiple-time-window* variant expressed in Constraints 6.6 can be reformulated through a simple problem transformation. Precisely, each customer i having a set of W_i time windows $[O_{iw}, C_{iw}], w \in \{1, \ldots, W_i\}$ can be replaced by a set of W_i atomic customers S_i (one customer per time window) having the same location and the same properties. Then, a constraint is added to guarantee that at most one of these atomic customers is visited:

$$\sum_{v=1}^{M} \sum_{j \in S_i} x_{jv} \leq 1 \tag{6.8}$$

Summary The mathematical model using Equations 6.1 to 6.8 described above can cover various variants of the orienteering problems, which possibly involve:

- multiple vehicles (when $M > 1$),
- multiple time windows for each customer (Constrains 6.8),
- multiple linear selection constraints (Constraints 6.7),
- and/or time-dependent transition times (Constraints 6.5).

Problem decomposition As mentioned in Chapter 3, the orienteering problem can be decomposed into two subproblems:

- a *selection subproblem* dedicated to the selection of the subset of customers to visit and to the assignment of a vehicle to each customer.

Based on the formulation given before, the selection subproblem can be formulated as follows.

$$\max \sum_{v=1}^{M} \sum_{i=1}^{N} R_i x_{iv}$$

subject to:

Constraints 6.7

$x_{iv} \in \{0,1\}; \ \ \forall i \in \{1,\ldots,N\}; \ \forall v \in \{1,\ldots M\}$

Here, Constraints 6.3 and 6.8 are implicitly expressed as a special case of Constraints 6.7.

- a *sequencing subproblem* used for determining a visit sequence traversing a specific set of customers while respecting the temporal constraints, which can be formulated as a Constraint Satisfaction Problem as below.

  Constraints 6.2, 6.4, 6.5

  $y_{ijv} \in \{0,1\}; \ \ \forall i \in \{0,\ldots,N\}; \forall j \in \{1,\ldots,N+1\}; \forall v \in \{1,\ldots,M\}$

  $s_{iv} \in [O_i, C_i]; \ \ \forall i \in \{0,\ldots,N+1\}; \ \forall v \in \{1,\ldots,M\}$

  In the above formulation, we assume that thanks to the transformation expressed in Constraint 6.8, we only deal with *atomic* customers (i.e. having only one time window). Therefore, Constraints 6.6 can be discarded from the model of the problem.

6.2 A generic solving framework

To search for near-optimal solutions to complex orienteering problems, we propose the modular architecture provided in Figure 6.1. This architecture is composed of three components, namely one main solver interface (called *GenOP*), one reasoning engine (called *selMgr*) dedicated to the selection (or assignment) subproblem, and one reasoning engine (called *routingMgr*) dedicated to the routing (or sequencing) subproblem. The primary idea here is to be able to directly reuse low-level optimization techniques related to selection subproblems and routing subproblems without optional customers. Technically, the *GenOP* solver contains the main solving function and interacts with the low-level reasoners through queries and operations, which are described below.

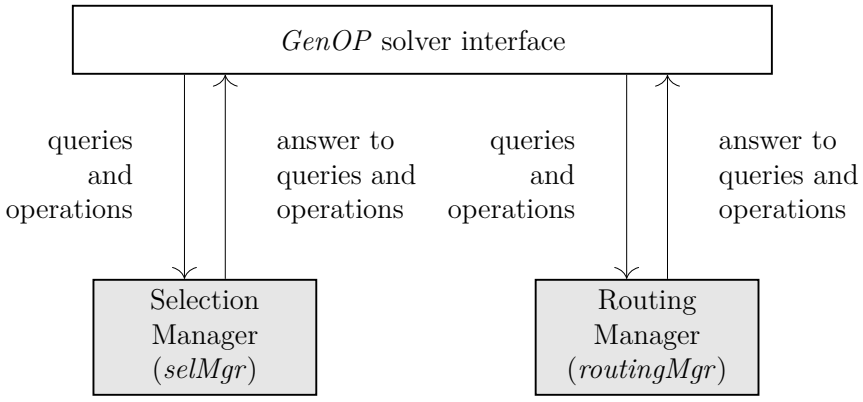


Figure 6.1 – A general architecture for solving complex orienteering problems

Possible functionalities at the selection level The *selMgr* module concentrates on managing the constraints of the problems at the selection level. In principle, *selMgr* must keep track of every decision made by *GenOP* and update the constraints accordingly. Moreover, *selMgr* should be able to quickly answer queries sent by *GenOP*, regarding the feasibility or optimality of the selection problem. Technically, several functions of *selMgr* can be considered for these purposes.

- `addConstraint`(c_z), to add a selection constraint c_z to the selection manager (i.e. constraints of the form given in Equation 6.7);
- `removeConstraint`(c_z), to remove an existing selection constraint c_z;
- `addObjective`(c_{obj}), to set the objective function at the selection level;
- `setBoundOnObjective`(LB), to update the lower bound for the objective function;
- `assign`(x_{iv}, b), to specify an assignment $[x_{iv} = b]$ for a selection variable x_{iv} of the selection subproblem;
- `unassign`(x_{iv}), to declare that selection decision x_{iv} is unassigned;
- `propagate`(), to check the consistency of the selection subproblem and find propagated decisions;
- `isAllowed`(x_{iv}, b), to evaluate whether value b is still a possible value for selection variable x_{iv} according to the selection manager;

- `resetAssignment`(), to reset the domain of all the selection variables and other related information (i.e. a call to this function expresses that all the variables are unassigned).

Possible functionalities at the routing level Module *routingMgr* is responsible for evaluating the feasibility (or the optimality) at the sequencing level. This module disposes of several basic functions such as:

- `resetSequences`(), to reinitialize the sequences of visits by removing all customers visited;
- `insert`(i, v), to evaluate whether a customer i can be inserted into the visit sequence of vehicle v and incrementally apply the insertion if feasible;
- `remove`(i, v), to incrementally remove customer i from the visit sequence of vehicle v.

The list of functions mentioned above is not exhaustive, and there may be other relevant functions depending on the search strategy used by *GenOP*. In the following, we detail the functions associated with each sub-module in Section 6.3, and show how we design a metaheuristic for the *GenOP* solver at the upper level in Section 6.4.

6.3 Definition of the low-level reasoners

In this section, we detail the techniques and functions defined for each sub-module, to exchange information with *GenOP* as *incrementally* as possible.

6.3.1 Selection manager

Globally, the selection subproblem involves a set of linear pseudo-Boolean constraints of the form $\sum_{i,v} e_{ivz} x_{iv} \sim E_z$ where $\sim \in \{\leq, \geq\}$, $x_{iv} \in \{0, 1\}$, and $e_{ivz}, E_z \in \mathbb{N}$. The primary objective here is to find an assignment A of values to variables x_{iv} such that all these constraints are satisfied. In the following, we denote $D(x_{iv}) = \{0, 1\}$ as the domain of variable x_{iv} and $A(x_{iv}) \in \{nil, 0, 1\}$ as the truth value of variable x_{iv} in the assignment A, where $A(x_{iv}) = nil$ means that x_{iv} is not assigned yet in A.

In principle, *selMgr* aims to perform operations and to answer queries requested by the *GenOP* solver as incrementally as possible. Globally, the goal of *selMgr* is to enforce the consistency of the selection problem. In fact, there exist various solvers available online developed for pseudo-Boolean constraint solving (Sheini & Sakallah (2005); Piotrów (2020)), making them suitable for fulfilling the role of *selMgr*. However, to fully control and exploit information obtained at the selection level in a flexible manner, we decided to construct our own *selMgr* instead of relying on the existing ones, given that the implementation of the basic mechanisms considered here is not particularly challenging and it also allows us to define incremental computation methods.

For ease of understanding, we first provide some background related to the pseudo-Boolean constraints before detailing the implementation of the necessary functions of the *selMgr* module.

Preliminaries In the literature, various works exploit consistency checking and unit propagation to solve problems involving pseudo-Boolean constraints (Dixon & Ginsberg (2002); Chai & Kuehlmann (2003); Sheini & Sakallah (2005)). In principle, inconsistent values for unassigned variables can be efficiently filtered by regarding the slack values of a pseudo-Boolean constraint, which can be formally defined as follows.

Definition 6.1. *Given a pseudo-Boolean constraint (called LE constraint for Less than or Equal)*

$$c_z : \sum_{v=1}^{M} \sum_{i=1}^{N} e_{ivz} x_{iv} \leq E_z$$

where $e_{ivz}, E_z \in \mathbb{N}$*, and the current assignment* A*, the slack value of* c_z *is formally defined by*

$$slack_z = E_z - \sum_{A(x_{iv})=1} e_{ivz}$$

Definition 6.2. *Given a pseudo-Boolean constraint (called GE constraint for Greater than or Equal)*

$$c_z : \sum_{v=1}^{M} \sum_{i=1}^{N} e_{ivz} x_{iv} \geq E_z$$

where $e_{ivz}, E_z \in \mathbb{N}$*, and the current assignment* A*, the slack value of* c_z *is formally defined by*

$$slack_z = \sum_{A(x_{iz})\neq 0} e_{iz} - E_z$$

Intuitively, the slack value of a constraint c_z measures how much margin is left to satisfy the constraint. Based on the slack values, inconsistent assignments can be detected and such values can be filtered from the variable domains.

Proposition 6.1. *Given a pseudo-Boolean constraint*

$$c_z : \sum_{v=1}^{M} \sum_{i=1}^{N} e_{ivz} x_{iv} \sim E_z$$

where $\sim \in \{\leq, \geq\}$, $x_{iv} \in \{0, 1\}$, *and* $e_{ivz}, E_z \in \mathbb{N}$. *Then,* c_z *is satisfied if and only if* $slack_z \geq 0$.

Proposition 6.2. *Given an LE constraint* c_z, *if there exist an unassigned variable* x_{iv} *(i.e.* $A(x_{iv}) = nil$*) such that* $slack_z < e_{ivz}$, *then value* 1 *can be pruned from the domain of* x_{iv}.

Proposition 6.3. *Given a GE constraint* c_z, *if there exist an unassigned variable* x_{iv} *(i.e.* $A(x_{iv}) = nil$*) such that* $slack_z < e_{ivz}$, *then value* 0 *can be pruned from the domain of* x_{iv}.

Example 6.1. *Let us consider the two following pseudo-Boolean constraints:*

$$c_1 : 2x_1 + 2x_2 + x_3 \leq 3$$
$$c_2 : x_2 + 2x_3 + x_4 \geq 2$$

At initialization, no variable is assigned then $slack(c_1) = 3$ *and* $slack(c_2) = 2$. *Assume that we assign* $[x_1 = 1]$, *then the slack value of constraint* c_1 *can be updated incrementally by*

$$slack(c_1) \leftarrow slack(c_1) - 2 = 1$$

As $slack(c_1) = 1 < 2$, *we can propagate* $x_2 \neq 1$, *or equivalently we must assign* $[x_2 = 0]$ *to avoid the violation of constraint* c_1. *Once again, the slack value of constraints* c_1 *and* c_2 *are incrementally updated by*

$$slack(c_1) \leftarrow 1$$
$$slack(c_2) \leftarrow slack(c_2) - 1 = 1$$

Similarly, as $slack(c_2) = 1 < 2$, *we can infer that* $x_3 \neq 0$, *or equivalently, we must assign* $[x_3 = 1]$ *to avoid the violation of constraint* c_2.

Detailed implementation For the `resetAssignment`() function, as illustrated in Algorithm 6.1, we simply reinitialize the domain and the assignment of all decision variables (Lines 1-3), together with the slack value of all constraints (Lines 4-8).

Algorithm 6.1: selMgr::`resetAssignment`()

```
1 for all variables x_iv do
2 |   A(x_iv) ← nil
3 |   D(x_iv) ← {0, 1}
4 for all constraints c_z do
5 |   if c_z is an LE constraint then
6 |   |   slack_z ← E_z
7 |   else if c_z is a GE constraint then
8 |   |   slack_z ← Σ_{i=1}^{N} Σ_{v=1}^{M} e_ivz − E_z
```

Next, the `assign`(x_{iv}, b) function described in Algorithm 6.2 tries to add assignment decision $[x_{iv} = b]$ and returns value *true* if and only if this change is accepted based on its compatibility with the current state of the selection manager. In particular, there is no update required when variable x_{iv} is already assigned (Line 1). Otherwise, besides setting value $b \in \{0, 1\}$ for variable x_{iv} (Line 2), the function also incrementally updates the slack values of the constraints (Lines 5, 11). Then, these slack values can be used to reduce the domain of other variables (Lines 8, 14). Once an empty domain is found, this function immediately returns value *false* (Lines 9, 15).

It is worth noting that the `assign` function described above only performs some domain reductions and no actual full propagation mechanism. On the latter point, to enhance the reasoning power of *selMgr*, we consider the `propagate`() function presented in Algorithm 6.3. This function performs unit propagation, a key technique used in the context of SAT solving and pseudo-Boolean solving, to simplify the formula and check its consistency. Precisely, while there exists an unassigned variable x_{iv} having a singleton domain (Line 2), the `propagate`() function can successfully infer the value of x_{iv} (Line 3) or detect inconsistency if this assignment is not accepted (Lines 4-5). Moreover, this function also records all positive assignments, i.e. $[x_{iv} = 1]$, that are derived during the propagation phase (Line 6). This point is very useful in our context. Indeed, for complex orienteering problems,

Algorithm 6.2: selMgr::`assign`(x_{iv}, b)

```
1  if A(x_iv) ≠ nil then return true
2  A(x_iv) ← b
3  for every constraint c_z s.t. x_iv ∈ c_z do
4      if (c_z is a LE constraint) and (b = 1) then
5          slack_z ← slack_z − e_iv
6          for x_i'v' ∈ c_z s.t A(x_i'v') = nil do
7              if slack_z < e_i'v' then
8                  D(x_i'v') ← D(x_i'v') \ {1}          /* Proposition 6.2 */
9                  if D(x_i'v') = ∅ then return false
10     else if (c_z is a GE constraint) and (b = 0) then
11         slack_z ← slack_z − e_iv
12         for x_i'v' ∈ c_z s.t A(x_i'v') = nil do
13             if slack_z < e_i'v' then
14                 D(x_i'v') ← D(x_i'v') \ {0}          /* Proposition 6.3 */
15                 if D(x_i'v') = ∅ then return false
16 return true
```

such an assignment $[x_{iv} = 1]$ means that customer i has to be visited by vehicle v to avoid the violation of selection constraints. Therefore, *selMgr* can identify mandatory selections and notify them to *GenOP* for guiding the search process. More details are provided later on this point (Section 6.4.4).

From a technical point of view, the `assign` and the `propagate` functions are implemented in an incremental manner to optimize the performance. The key technique is to use, for instance, FIFO queues to incrementally revise modifications and to avoid repeating redundant works in the assignment or the propagation procedure. For example, when multiple `assign` operations are consecutively applied, instead of checking for domain reductions after each assignment one-by-one, we first store all the constraints involving the newly assigned variables in a queue, then update the slack values of these constraints and finally search for the domain reductions. Similarly, for the `propagate` function, we use a queue to store all unassigned variables having a singleton domain. This queue is updated each time such a variable is found, and the propagation process repeats until the queue becomes empty.

Algorithm 6.3: selMgr::`propagate`()

```
1 Q ← ∅
2 while ∃x_iv s.t. A(x_iv) = nil and D(x_iv) = {b} do
3 |   consistent ← assign(x_iv, b)
4 |   if ¬consistent then
5 |   |   return (¬consistent, Q)
6 |   if b = 1 then Q ← Q ∪ {x_iv}
7 return (true, Q)
```

For other functions such as `addConstraint`(c_z), `addObjective`(c_{obj}), `setBoundOnObjective`(LB), `isAllowed`(x_{iv}, b), their implementations are quite straightforward. Remarkably, the `unassign` function is not implemented yet in our current *selMgr*, but it would be useful to quickly undo previous decisions, for example, when some of the customers are individually removed from the current sequences of visits.

6.3.2 Routing manager

Basically, the *routingMgr* module is responsible for managing all temporal constraints of the problem. To increase the generality of *GenOP*, the *routingMgr* should be able to handle a TSP in a flexible manner, regardless of whether the problem involves time window constraints and/or time-dependent transition times.

For this purpose, we can fully exploit the specificity and efficient OR techniques for a standard Time-Dependent TSP with Time Windows (TDTSPTW). In this contribution, we directly reuse a state-of-the-art TDTSPTW solver, called ImaxLNS, recently presented by Pralet (2023), as our *routingMgr* module. Besides using this solver to find the optimal visit sequence given a specific set of customers, we make some adaptations to this algorithm to use it in a dynamic context where customers can be iteratively added or removed, as requested by the *GenOP* solver.

In principle, the ImaxLNS solver was originally developed to handle problems involving a single vehicle. Hence, to deal with the multi-vehicle variant, we employ multiple ImaxLNS solvers associated with different vehicles within the *routingMgr* module. In the following, we denote $\sigma[v]$ as the visit sequence of vehicle v, and $tdtsptwSolver[v]$ as the ImaxLNS solver for vehicle v.

Then, the implementation of the `insert`(i, v) function is straightforward as illustrated in Algorithm 6.4. In principle, this function checks whether a customer i can be inserted in the current visit sequence of vehicle v. The insertion method used in Line 1 is a fast greedy insertion algorithm available in ImaxLNS, that simply tries to heuristically insert customer i at the *best* possible position in the current sequence of visits of vehicle v. In this context, the term '*best*' refers to the position that yields the least increase in the total traveling time. Functions `getSequence()` and `setSequence()` allow us to get the current sequence of visits manipulated by the ImaxLNS solvers and to update this sequence.

Algorithm 6.4: routingMgr::`insert`(i, v)

```
1 insertOK ← tdtsptwSolver[v].insert(i)
2 if insertOK then
3     σ[v] ← tdtsptwSolver[v].getSequence()
4 else
5     tdtsptwSolver[v].setSequence(σ[v])
6 return insertOK
```

The *routingMgr* module also disposes of a `remove`(i, v) function that allows us to *incrementally* remove customer i from the current sequence of visits of vehicle v. A key point here is that the insertion and removal operations can be requested by *GenOP* in *any* order. For the `resetSequences`() function, *routingMgr* simply restarts all the ImaxLNS solvers from an empty sequence of visits.

More than that, ImaxLNS also offers other advanced functions that can help optimize the sequencing decision of the *routingMgr* module. Several typical functions are the well-tuned specific insertion method rather than greedy heuristics, or the reordering of the visit sequence. The latter operation can be used to search for a new sequence of visits that is compatible with the time window constraints or to minimize the total traveling time. These points will be exploited later in Section 6.6.

6.4 A metaheuristic for the high-level GenOP

In principle, the *GenOP* solver exploits the functions available in each submodule to try and build a customer selection strategy in order to find sequences of visits maximizing the total reward collected. The search at the level of *GenOP* can be enhanced thanks to the feedback (or advice) received from the selection module and the feedback received from the routing module concerning the feasibility of the sequences of visits. On this point, there exist several candidate metaheuristics inspired by the techniques available in the literature for solving variants of OP (see Chapter 3).

6.4.1 Solution representation

In the *GenOP* solver, a solution is represented as M sequences of visits associated with M vehicles or formally $\sigma = \{\sigma[1], \ldots, \sigma[M]\}$. The total reward collected through all the vehicles can be denoted as $R(\sigma) = \sum_{v=1}^{M} \sum_{i=1}^{N} R_i x_{iv}$.

As explained before, at the lower level, *selMgr* manipulates Boolean variables x_{iv} to represent the assignment of the customers i to the vehicles v during the search process, whereas the routing information (e.g., the starting time of a customer visit) is implicitly stored in the *routingMgr* and used to evaluate the feasibility of a sequence of visits with respect to the temporal constraints.

6.4.2 Multi-start Large Neighborhood Search

In this contribution, we propose an example of a metaheuristic for the *GenOP* solver, the so-called Multi-Start Large Neighborhood Search (MSLNS), to find a high-quality solution for complex orienteering problems. As shown in Algorithm 6.5, our implementation of MSLNS for *GenOP* relies on LNS as the core search technique to intensify the search (Lines 10-22) and performs multiple restarts from scratch to diversify the search (Lines 6-9).

Precisely, the algorithm first declares the Boolean variables x_{iv} and adds the initial selection constraints and the objective function to the *selMgr* module (Lines 2-4). For each different starting point, the algorithm simply re-initializes the solution representation σ (Line 6), the sequencing data in *routingMgr* (Line 7), and the customer selection in *selMgr* (Line 8). It is important to note that *selMgr* still keeps additional selection constraints

Algorithm 6.5: GenOP::MSLNS($maxCpu, maxNoImpr, rmRatio$)

```
1  σ* ← nil; σ ← nil
2  selMgr.declareVar()
3  for each constraint c_z do selMgr.addConstraint(c_z)
4  selMgr.addObjective()
5  while cpu() < maxCpu do
       /* Restart from scratch                                   */
6      for v ∈ [1..M] do σ[v] ← [0, N + 1]
7      routingMgr.resetSequences()
8      selMgr.resetAssignment()
9      selMgr.setBoundOnObjective(1)
       /* LNS procedure                                          */
10     σ_Lopt ← σ
11     while cpu() < maxCpu and noImpr < maxNoImpr do
12         σ ← destroy(σ, rmRatio)              /* Section 6.4.3 */
13         σ ← repair(σ)                        /* Section 6.4.4 */
14         if R(σ) ≥ R(σ_Lopt) then
15             if R(σ) > R(σ_Lopt) then
16                 selMgr.setBoundOnObjective(R(σ))
17                 noImpr ← 0
18             else noImpr ← noImpr + 1
19             σ_Lopt ← σ
20         else
21             σ ← σ_Lopt
22             noImpr ← noImpr + 1
       /* Update best found solution                             */
23     if R(σ_Lopt) > R(σ*) then σ* ← σ_Lopt
24 return σ*
```

possibly generated during the search as they are useful for the subsequent steps.

For the LNS procedure, the algorithm employs successive destroy and repair operators to hopefully search for a better quality solution (Lines 12-13, more details later on this point). If the new solution obtained, σ, is better than or equal to the current local optimum σ_{Lopt}, it is used for the next LNS

iteration (Lines 14, 19) and the bound on the objective value can be tightened (Line 16). Otherwise, the algorithm comes back to the solution available before the destroy-repair step (Line 21). These procedures are repeated until no strict improvement is found after $maxNoImpr$ iterations or the global computational time $maxCpu$ is reached.

Remarkably, preliminary tests show that keeping the strictly tightened bound on the objective value, i.e. $\sum_{v=1}^{M}\sum_{i=1}^{N} R_i x_{iv} \geq R(\sigma^*) + 1$, does not produce the better results since this condition is too restrictive and limits the diversification degree in the context of incomplete search. This is why we decide to relax the constraint on the objective value in *selMgr* by:

- setting $\sum_{v=1}^{M}\sum_{i=1}^{N} R_i x_{iv} \geq R(\sigma)$ once a new strict improvement is found within the LNS procedure (Line 16);
- and resetting $\sum_{v=1}^{M}\sum_{i=1}^{N} R_i x_{iv} \geq 1$ at each restart (Line 9).

6.4.3 Destroy procedure

Function `destroy`$(\sigma, rmRatio)$ shown in Algorithm 6.6 takes as an input a set of visit sequences σ (one sequence per vehicle) and a parameter *rmRatio* indicating the maximum destroy degree.

Algorithm 6.6: GenOP::`destroy`$(\sigma, rmRatio)$

```
1 selMgr.resetAssignment()
2 for v ∈ [1..M] do
3 |   F ← select k consecutive customers in σ[v] to remove given
  |     rmRatio
4 |   σ[v] ← σ[v] \ F
5 |   for i ∈ σ[v] do
6 |   |   selMgr.assign(x_iv, 1)
7 |   for i ∈ R do
8 |   |   routingMgr.remove(i, v)
9 return σ
```

Similarly to the principle of the destroy operators presented in Sections 4.2 and 5.1, for each vehicle v, *GenOP* applies a random *shake* operator to remove a subset F of consecutive customers from the current sequence of visits $\sigma[v]$

(Lines 2-4). The number of customers selected for removal, denoted k, is randomly chosen from a discrete uniform distribution $k \sim U(1, rmRatio*N_v)$, where N_v is the number of customers visited by vehicle v (as in Section 5.1).

For the selection aspect, as the *selMgr* that we developed does not dispose of an `unassign`(x_{iv}) function to incrementally unassign the customer selection, the destroy procedure simply resets the assignment in *selMgr* (Line 1) and performs again the assignments from scratch given the new current solution (Lines 5-6). For the *routingMgr* module, the sequencing data of each vehicle v is incrementally updated using the remove functions offered by the ImaxLNS solvers (Lines 7-8).

6.4.4 Repair procedure

One of the core components of the MSLNS algorithm is the repair procedure specifying how to re-optimize the current solution through iterative customer selections and insertions. In this procedure, *GenOP* proposes a customer selection strategy and employs *selMgr* and *routingMgr* to verify the feasibility at the selection and sequencing levels, respectively. The pseudocode of the repair procedure is outlined in Algorithm 6.7. For the sake of simplification, we can temporarily ignore the lines highlighted in gray (Lines 7, 13, and 17), which will be detailed later in Section 6.4.5.

Initially, *GenOP* defines a naive selection order by sorting unassigned selection variables x_{iv} in a priority queue Q (Line 1). In this list, the x_{iv} variables are sorted in a decreasing order based on the value $f_i \sim U(1, R_i)$, corresponding to a random reward uniformly distributed in interval $[1, R_i]$.

Based on this selection order, the algorithm repeatedly evaluates candidate selections in Q until Q is empty, while ensuring consistency at the level of the selection subproblem (Lines 2-16). The latter is examined by calling the `propagate`() function available in *selMgr*, and in case of consistency, a set of *mandatory* assignments at the selection level is returned (Lines 2, 16). For every mandatory assignment, the algorithm invokes the *routingMgr* to evaluate the sequencing feasibility and tries to apply such an assignment if possible. In case of failure, the algorithm immediately stops the repairing procedure (Lines 6-8).

If there is no such mandatory assignment, the algorithm proceeds to evaluate a candidate selection x_{iv} retrieved from Q (Line 9). If decision $[x_{iv} = 1]$ is allowed by *selMgr* (Line 10), the *routingMgr* module attempts to insert customer i into the visit sequence of vehicle v (Line 11). The *selMgr* mod-

Algorithm 6.7: GenOP::`repair`(σ)

```
1  Q ← sort({x_iv | i ∈ {1,...,N}, i ∉ σ, v ∈ {1,...,M}}) based on
     noisy reward
2  (isConsistent, mandatoryQueue) ← selMgr.propagate()
3  while isConsistent and Q ≠ ∅ do
       /* Evaluate ALL mandatory selection                          */
4      while mandatoryQueue ≠ ∅ do
5          x_iv ← mandatoryQueue.pop()
6          if routingMgr.insert(i, v) = false then
7              analyzeTWconflict(σ[v], i)
8              return σ
       /* Evaluate a candidate selection                            */
9      x_iv ← Q.pop()
10     if selMgr.isAllowed(x_iv, 1) then
11         if routingMgr.insert(i, v) = false then
12             selMgr.assign(x_iv, 0)
13             analyzeTWconflict(σ[v], i)
14         else
15             selMgr.assign(x_iv, 1)
16         (isConsistent, mandatoryQueue) ← selMgr.propagate()
17 analyzeLopt(σ)
18 return σ
```

ule also updates the current assignment based on the result returned by *routingMgr* (Lines 12, 15).

Overall, the repair procedure terminates in three cases: when there is a disagreement between the *selMgr* and the *routingMgr* (Line 8), when there is no candidate selection left, or when an inconsistency is detected (Line 18).

6.4.5 Conflict analysis procedure

The last point concerns the generation of additional selection constraints during the search procedure to enhance the knowledge of the selection aspect. This can be done by analyzing insertion failures based on temporal constraints (Algorithm 6.7 - Lines 7, 13) and/or by analyzing the solution

obtained after the repair phase (Algorithm 6.7 - Line 17), following the ideas presented in Chapter 4.

The constraint generation procedures outlined in Algorithms 6.8 and 6.9 are mainly derived from the clause generation procedure described in Section 4.3. Once difference here is that the conflicts generated are not represented as clauses, but rather as pseudo-Boolean constraints. Precisely, a TW-conflict $S_c = \{i_1, \ldots, i_k\} \subseteq \{1, \ldots, N\}$ can be applied for all vehicles using v constraints, called *TW-constraints*:

$$\sum_{i \in S_c} x_{iv} \leq |S_c| - 1 \; \forall v \in \{1, \ldots, M\} \tag{6.9}$$

Notably, in Equation 6.9, we assume that all the vehicles are homogeneous (i.e. having the same transition function) and that all customers have the same time windows for all vehicles. Otherwise, only one TW-conflict is added to *selMgr* and the framework still works.

Algorithm 6.8: GenOP::`analyzeTWconflict`$(\sigma[v], i)$

```
1 V ← select(σ[v], i, maxConfSize)
2 C ← extractMinTWconflicts(V ∪ {i})          /* Section 4.3.2 */
3 for S_c ∈ C do
4     for v ∈ {1, ..., M} do
5         c_TW ← Σ_{i∈S_c} x_iv ≤ |S_c| − 1
6         selMgr.addConstraint(c_TW)
```

Similarly, an approximate Lopt-conflict, that requires at least one vehicle to visit at least one customer $i \in Y$ (where Y is the set of unvisited customers having high rewards) can also be expressed as the following pseudo-Boolean constraint, called an *Lopt-constraint*:

$$\sum_{v=1}^{M} \sum_{i \in Y} x_{iv} \geq 1 \tag{6.10}$$

An interesting point is that such Lopt-constraints are stored in *selMgr* in a *temporary* manner by using a specific `addTmpConstraint` function. This is done by using a *tabu list* that only keeps Lopt-constraints in *selMgr*. When

Algorithm 6.9: GenOP::`analyzeLopt`(σ^*)

1 $U \leftarrow$ `sort`($\{i \mid i \in \{1,\ldots,N\}, i \notin \sigma^*\}$) based on their rewards
2 $Y \leftarrow$ *approxSize* elements having the highest rewards in U
3 $c_{lOpt} \leftarrow \sum_{v=1}^{M} \sum_{i \in Y} x_{iv} \geq 1$
4 *selMgr*.`addTmpConstraint`(c_{lOpt}, *tabuTenure*)

this tabu list reaches its maximum size, denoted as *tabuTenure*, the oldest Lopt-constraint is discarded, as in tabu search. This point can be easily done since we can fully control the content of the *selMgr* module.

6.4.6 Search parameters

As usual in metaheuristic search, the algorithm proposed has several parameters summarized in Table 6.1, which also provides the values chosen after preliminary tests. Since the conflict analysis procedure is optional, the MSLNS metaheuristic proposed for *GenOP* has different variants, depending on whether additional constraints are generated during the main search procedure or not.

Component	Parameter	Semantics	Values
MSLNS	*maxNoImpr*	Max. number of LNS iterations without improvement	$1000 * M$
	rmRatio	Max. removal ratio used in the LNS destroy phase	40%
xpTW	*maxNbTwCtr*	Max. number of TW-constraints stored in *selMgr*	$20000 * M$
	maxConfSize	Max. cardinality of a TW-conflict	4
	xpQuota	Max. number of successive useless explanation attempts for each customer	20
xpLopt	*approxSize*	Max. number of customers in an approximate Lopt-conflict	7
	tabuTenure	A number of iterations during which an Lopt-constraint is available in *selMgr*	50

Table 6.1 – Parameters used for variants of GenOP-MSLNS

Interestingly, the simplest version of MSLNS has only two parameters: *maxNoImpr* and *rmRatio*, which allow us to easily adjust the balance between search intensification (large value of *maxNoImpr* and/or small value of *rmRatio*) and search diversification (small value of *maxNoImpr* and/or large

value of *rmRatio*). In the following experiments, we always use $rmRatio = 40\%$ as in the work of Schmid & Ehmke (2017) and our experiments in Section 5.5. As for parameter *maxNoImpr*, we empirically observe that the problem becomes more complex when considering multiple vehicles, as we have to deal with more variables and constraints in the *selMgr* module. Therefore, to intensify the search in this case, we use $maxNoImpr = 1000 * M$ to adjust parameter *maxNoImpr* according to the number of vehicles (M).

Besides, *GenOP*-MSLNS can be further enhanced with the conflict analysis procedure given in Section 6.4.5. There are several parameters for controlling the use of each type of constraint (TW-constraint or Lopt-constraint) in the *selMgr* module. The values of these parameters are chosen based on the experiments from our previous work in Section 4.5.

6.5 Experiments

To demonstrate the genericity of the *GenOP* solver for solving complex variants of the orienteering problem, we carry out experiments on various benchmarks including[1]

- (1) TOPTW instances involving multiple vehicles ($M = \{1, \ldots, 4\}$),
- (2) TOPTW instances involving multiple knapsack constraints (denoted as MC-TOPTW),
- (3) TOPTW instances involving both multiple knapsack constraints and multiple time windows (denoted as MC-TOP-MTW),
- and (4) generated OPTW instances that cover the time-dependency and multiple time windows aspects (denoted as TD-OP-MTW).

For the *GenOP* solver, three variants of MSLNS are considered. The simplest version denoted as MSLNS-basic, does not invoke the conflict analysis procedure and only uses a *greedy randomized* selection strategy (i.e. based on the noisy reward function) as well as the greedy insertion heuristics concerning the sequencing aspect. On the other hand, two other variants of MSLNS consider the conflict analysis procedure to generate additional

[1]Instances for TOPTW, MC-TOPTW, MC-TOP-MTW are available at `https://www.mech.kuleuven.be/en/cib/op`

selection constraints during search. For these two variants, one exploits TW-constraints only (denoted as MSLNS+TW), and another one exploits both TW-constraints and Lopt-constraints (denoted as MSLNS+TW+Lopt). The motivation behind this is derived from our previous work in Chapter 4, whose objective is to enhance the knowledge in *selMgr* with the hope of guiding the selection strategy in *GenOP*.

The effectiveness of the *GenOP*-MSLNS algorithm can be evaluated by comparing the results obtained by other relevant methods dedicated to each specific problem in the literature. For measuring the quality of the solutions obtained, we can compute the average gap compared to the best-known solutions available in the literature. For each solver, this *quality gap* in percent (%) is formally given by

$$gap = 100 * \frac{R_{bk} - R_{bf}}{R_{bk}}$$

where R_{bk} (or R_{bf}) is the total reward of the best-known (respectively, best-found) solution. Then, a smaller gap indicates a more effective solver. In particular, a *negative* gap signifies that the solver discovers a new best-known solution. Also, it is crucial to report the computational time required by each solver to produce a low quality gap.

However, we need to be careful when comparing the performance of different solvers (including both the quality gap and the computational time), since the results reported in the literature correspond to the best and/or the average solutions obtained by using a varying number of runs and/or different termination conditions. On this point, there are two possible ways commonly used in the literature.

- A direct way to compare a single-run algorithm with a multiple-run algorithm is to report the *average* results and the *average* computational time.
- In the case where the *best* results of a multiple-run algorithm are used for comparison, we should consider the *total* computational time of all runs. This is analogous to a multi-start algorithm which requires the total time for performing all runs to obtain the best result.

Since the MSLNS algorithm dynamically performs restarts during search, all the experiments for the MSLNS variants are executed only once by using a fixed CPU time (or timeout), and the best result is returned. Said

differently, MSLNS is akin to a single-run algorithm mentioned in the above discussion. Also, the computational time of MSLNS reported in the following experiments is the time spent to find the best solution before the timeout is reached.

Last, all the implementations are written in C++ and compiled in a Linux environment with gcc-9.4.0. All the experiments are performed over one thread on a processor Intel(R) Xeon(R) CPU E5-2660 v3 @ 2.60GHz.

6.5.1 TOPTW benchmark

This first benchmark contains 76 standard TOPTW instances generated based on the datasets of Solomon (1987) and Cordeau et al. (1997), where the number of vehicles ranges from 1 to 4. We recall that all instances defined by Solomon (1987) contain 100 customers, while the number of customers in the instances defined by Cordeau et al. (1997) varies between 48 and 288. All the best-known solutions are retrieved from the work of Schmid & Ehmke (2017).

Experimental settings The standard TOPTW instances considered here were tackled by several authors in the past, which gives us a baseline provided by specific TOPTW solvers. Yet, we only consider here the hybrid SAILS of Gunawan et al. (2015b) and the effective LNS (eLNS) of Schmid & Ehmke (2017) as reference points for *GenOP*-MSLNS. Besides having a similar search behavior (i.e. destroy and repair mechanisms), these two metaheuristics are shown to be very effective not only for TOPTW but also for several OP variants. We briefly recall that, in SAILS, the authors exploit different local search operators including the *insert*, *replace*, *swap*, and *exchange* moves. These moves update either each individual route or multiple routes at the same time. For eLNS, the algorithm iteratively performs customer removals and insertions based on well-designed heuristics.

It is also important to note that the *GenOP*-MSLNS variants are run within a predefined limited CPU time. Meanwhile, reference solvers (SAILS and eLNS) perform multiple independent runs and use a termination condition corresponding to a maximum number of iterations without improvement. To achieve a comparison that is as fair as possible, for SAILS and eLNS, we directly use both the average and the best results of different independent runs (10 runs for SAILS, 5 runs for eLNS) presented in the original papers of Gunawan et al. (2015b) and Schmid & Ehmke (2017). Based on the largest

computational time used by SAILS and eLNS for hard instances (around hundreds of seconds), we decided to set the timeout to 5 minutes for each variant of *GenOP*-MSLNS in the following experiments.

Nonetheless, a potential issue of this comparison is that we do not take into account the speed difference between the different computation setups. This problem can be handled by using the same approach as Hu & Lim (2014), namely the *SuperPi* benchmark, whose basic idea is to adjust the computational time based on the single-thread performance of the computers used. However, we do not consider this point in our experiments as we lack enough information on the experimental setup used by eLNS, for instance, to be able to estimate their SuperPi scores. Therefore, we assume that all experimental setups have a similar performance. This also means that all the comparisons of the computational times between different solvers are only *relative*.

Numerical analysis Table 6.2 shows the comparison of different solving approaches in terms of solution quality and computational time. The first column indicates the number of vehicles M and the second column gives the name of the instance set over which the results are aggregated. The following columns present the quality gaps obtained by SAILS, eLNS, and the three variants of *GenOP*-MSLNS. In the upper table, the quality gaps are reported over a total of 16 cases, given M ranging from 1 to 4 and 4 sets of instances (Solomon1, Solomon2, Cordeau1, Cordeau2). The lower table summarizes the average quality gap and the average computational time of the solver in each column over two large sets (i.e. Solomon and Cordeau) as well as all instance sets (Line 'All'), taking into account different numbers of vehicles. Notably, the computational time given here represents the average time required to reach the best-found solution.[2]

Focusing on the lower part of Table 6.2, we observe that the three *GenOP*-MSLNS variants are very competitive compared to the other two reference solvers. On average, MSLNS-basic achieves the best quality gap over all instances, though the difference is not significant (0.69% for MSLNS-basic compared to 0.71% for eLNS-best and 0.81% for the SAILS-best). This is mainly because MSLNS-basic performs better than the other solvers on Cordeau instances. Overall, all variants of MSLNS can quickly achieve a low average gap (less than 1%) for most cases, with an average computational

[2]For SAILS and eLNS, computational times are reported from the original papers.

M	Instance set	Gunawan et al. (2015c)		Schmid & Ehmke (2017)		*GenOP*-MSLNS (5mins)		
		SAILS (avg)	SAILS (best)	eLNS (avg)	eLNS (best)	basic	TW	TW+Lopt
1	Solomon1	0.02	**0.00**	0.10	**0.00**	0.14	0.14	0.14
	Solomon2	1.06	0.36	0.20	**0.06**	0.25	0.24	0.24
	Cordeau1	0.93	0.44	0.20	0.10	**0.00**	**0.00**	0.02
	Cordeau2	2.31	1.17	2.10	1.44	1.05	**1.04**	1.05
2	Solomon1	0.02	**0.00**	0.20	0.11	0.21	0.20	0.20
	Solomon2	1.66	0.73	0.14	**0.02**	0.53	0.47	0.49
	Cordeau1	2.19	0.66	1.00	0.58	**0.35**	0.47	0.49
	Cordeau2	3.44	1.74	2.60	1.69	**1.18**	1.88	1.74
3	Solomon1	0.52	0.09	0.40	**0.07**	0.19	0.25	0.25
	Solomon2	1.90	0.16	0.04	**0.02**	0.09	0.11	0.11
	Cordeau1	2.86	1.27	1.60	0.89	**0.73**	0.74	0.79
	Cordeau2	3.39	2.02	2.70	1.92	1.70	**1.69**	1.74
4	Solomon1	1.28	0.32	0.60	**0.28**	0.40	0.52	0.59
	Solomon2	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**
	Cordeau1	3.58	2.08	2.60	**1.72**	2.03	2.15	2.15
	Cordeau2	3.97	**1.88**	3.30	2.52	2.24	2.60	2.60
Summary								
Gap(%)	Solomon	0.81	0.21	0.21	**0.07**	0.22	0.24	0.25
	Cordeau	2.83	1.41	2.01	1.36	**1.16**	1.32	1.32
	All	1.82	0.81	1.11	0.71	**0.69**	0.78	0.79
Time (s)	Solomon	98.76	-	9.59	29.00	47.88	49.13	52.24
	Cordeau	198.94	-	75.59	129.39	107.38	101.37	93.25
	All	148.85	-	42.59	79.19	77.63	75.25	72.75

Table 6.2 – Results obtained on the TOPTW benchmark: quality gaps (%) obtained by different algorithms (upper table); average quality gaps (%) and average computational times (seconds) per instance set (lower table)

time of only 80 seconds, similarly to the best version of eLNS.

Our three MSLNS variants outperform SAILS in terms of both quality gap and computational time for most cases. Regarding computational time, SAILS takes about 100 seconds for Solomon instances and 200 seconds for Cordeau instances, on average. MSLNS, on the other hand, needs only half as much time, i.e. about 50 seconds and 100 seconds on average for Solomon and Cordeau instances, respectively. Besides, MSLNS gives better quality gaps than the best version of SAILS in 11 out of 16 cases. For SAILS-best, although the total time for obtaining the best results is not reported in the literature, it is obviously larger than their average time.

On the other hand, the performance of MSLNS-basic is somehow equivalent to the performance of eLNS-best. Comparing the quality gaps obtained for 16 cases, MSLNS-basic has better results in 8 out of 16 cases, while eLNS-best outperforms MSLNS-basic in 7 out of 16 cases (both solvers achieve the optimal solutions on Solomon2 instances using 4 vehicles). In general, for different numbers of vehicles, we remark that eLNS performs better on Solomon instances, whereas MSLNS-basic is better on Cordeau instances. Concretely, on Solomon instances, eLNS-best achieves an average gap of 0.07% with an average computational time of 29.00 seconds, while

MSLNS-basic has a slightly higher average gap of 0.22% and a longer average computational time of 47.88 seconds. On the contrary, when dealing with Cordeau instances, eLNS-best shows a higher average gap of 1.36% and a longer average runtime of 129.39 seconds, while MSLNS-basic achieves a somewhat lower average gap of 1.16% with a shorter average time of 107.38 seconds. These differences in performance showcase the sensitivity of each algorithm to the specific problem instances it deals with.

Besides, we remark that the impact of using additional constraints is not significant, contrarily to what was observed in Chapter 4. There are several possible explanations for this.

- First, in the LNS algorithm presented in Chapter 4, the use of TW-constraints for neighborhood pruning is efficient since at each construction step, the algorithm explores, for *all* unvisited customers, their insertion at each position in the visit sequence before choosing the best one according to a specific heuristic function. Meanwhile, in MSLNS, the algorithm selects a customer first and subsequently tests its insertion. As a result, there are fewer insertion tests. Also, in MSLNS, each TW-constraint is duplicated for every vehicle, which leads to a significant increase in the number of constraints in *selMgr* when using multiple vehicles. This implies that *selMgr* also requires more effort for the unit propagation procedure at the selection level.

- Second, using approximate Lopt-constraints can be efficient when customers having a high reward are not visited in the locally optimal solution. However, for problems involving multiple vehicles, more customers are selected. This implies that only less favorable customers (having a low reward) remain in the approximate Lopt-constraints.

To further compare the MSLNS variants with several relevant metaheuristics for the TOPTW, we show in Table 6.3 the percentage of best-known solutions obtained per number of vehicles. In general, eLNS gives the best results in all cases compared to the other solvers reported in this table. However, we can observe that the MSLNS used by *GenOP*, which employs a randomized customer selection strategy, provides a similar proportion of best-known solutions as other methods involving specific heuristics developed for this problem, like ILS or SAILS.

Overall, these findings showcase that the MSLNS metaheuristic used by the *GenOP* solver performs very well on the standard TOPTW instances.

Reference	Algorithm	Percentage of best known solutions				Average
		$M = 1$	$M = 2$	$M = 3$	$M = 4$	
Labadie et al. (2011)	GRASP-ELS	50.0	21.1	32.9	46.1	37.5
Lin & Vincent (2012)	SSA	51.3	34.2	39.5	56.6	45.5
Labadie et al. (2012)	GVNS	36.8	30.3	40.8	44.7	38.2
Souffriau et al. (2013)	GRILS	51.3	15.8	22.4	39.5	32.5
Hu & Lim (2014)	I3CH	43.4	34.2	57.9	55.3	47.7
Cura (2014)	ABC	48.7	36.8	46.1	48.7	45.1
Gunawan et al. (2015c)	ILS	48.7	36.5	46.1	48.7	45.1
Gunawan et al. (2015b)	SAILS	67.1	50.0	57.9	53.9	57.2
Schmid & Ehmke (2017)	eLNS	82.9	71.1	65.8	60.1	70.0
GenOP (Ours)	MSLNS-basic	75.9	43.2	63.1	52.6	58.5
	MSLNS-TW	75.0	46.1	61.8	48.7	57.9
	MSLNS-TW-Lopt	73.7	43.4	60.5	47.3	56.3

Table 6.3 – Percentage (%) of best-known solutions obtained by different algorithms on the TOPTW benchmark, extending the summary of Schmid & Ehmke (2017)

It should also be emphasized that our primary objective is to evaluate the *genericity* of the framework proposed, not to outperform the latest state-of-the-art solvers for TOPTW in the literature. In other words, this approach is not particularly designed only for the TOPTW problem but also targets different complex variants of OP, thanks to the modularity of the algorithmic architecture.

6.5.2 MC-TOPTW benchmark

A MC-TOPTW benchmark is designed by Souffriau (2010) based on instances from Solomon1 and Cordeau1 sets, where each instance involves 11 extra knapsack constraints on the selection aspect. Specifically, these constraints are designed so that high-quality (T)OPTW solutions are also valid high-quality solutions for the new MC-TOPTW problem. In short, optimal solutions are known for $M = 1$, and for other values of $M = \{2, 3, 4\}$, only good-quality solutions are known but they can be used as a reference for comparison.

Experimental settings To the best of our knowledge, only the works of Souffriau (2010) and Souffriau et al. (2013) deal with this benchmark. Therefore, we consider here as a baseline the GRILS algorithm proposed by Souffriau et al. (2013), which is a hybrid metaheuristic combining GRASP and ILS. In this work, the authors empirically study advanced but complex

insertion heuristics based on the reward gain, the time shift, and information related to the knapsack constraints.

Each experiment of GRILS is performed based on 10 independent runs on an Intel Xeon @ 2.5GHz (since the series details are not provided, we cannot compare the two different experimental environments). Therefore, to quickly compare our MSLNS variants with GRILS, we set a timeout of 1 minute for all variants of MSLNS to obtain a computational time that is similar to the time allocated to GRILS in the results provided by Souffriau et al. (2013). Then, we report here the best results obtained by each solver and the corresponding time that is spent to find the best solution.[3] Also, to look for new best-known solutions if any, we extend the time limit up to 5 minutes for *GenOP* in the second experiment.

Numerical analysis Table 6.4 presents the results of different solvers on 8 cases, with the number of vehicles ranging from 1 to 4 (column 1) and 2 instance sets (column 2). From the above table, we can observe that all three variants of MSLNS produce negative gaps in most cases, meaning that many best-known solutions are found by MSLNS. In general, MSLNS-basic seems to be the best version with the lowest quality gap obtained over all

M	Instance set	Souffriau et al. (2013) GRILS (best)	*GenOP*-MSLNS (1min)		
			basic	TW	TW+Lopt
1	Solomon	0.59	**0.58**	**0.58**	**0.58**
	Cordeau	4.24	**0.00**	**0.00**	**0.00**
2	Solomon	1.46	**-0.13**	-0.07	-0.06
	Cordeau	1.91	-3.23	**-3.41**	-3.36
3	Solomon	1.80	**-0.33**	-0.29	-0.26
	Cordeau	2.57	**-3.50**	-3.26	-3.22
4	Solomon	2.85	**-0.68**	-0.22	-0.15
	Cordeau	2.86	-2.11	**-2.38**	**-2.38**
Summary					
Gap (%)	Solomon	1.68	**-0.14**	0.00	0.03
	Cordeau	2.89	-2.21	**-2.26**	-2.24
	All	2.28	**-1.17**	-1.13	-1.11
Time (s)	Solomon	8.35	14.23	13.82	14.36
	Cordeau	19.24	19.94	23.30	24.23
	All	13.79	17.08	18.56	19.29

Table 6.4 – Results obtained on the MC-TOPTW benchmark: quality gaps (%) obtained by different algorithms (upper table); average quality gaps (%) and average computational times (seconds) per instance set (lower table)

[3]For GRILS, computational times are reported from the original paper.

cases. Yet, the effect of additional constraints generated during the search process is not negligible. Indeed, the MSLNS+TW variant gives the best results when aggregating over all Cordeau instances.

Based on the results obtained, we remark that all three variants of MSLNS outperform GRILS in all cases within a similar computational time. A possible explanation is that GRILS spends more effort in the insertion procedure by using a more fine-tuned but complicated heuristic. Precisely, the optimal insertion move is chosen by evaluating the insertion of every unvisited customer at all possible positions while satisfying the constraints. On the other hand, MSLNS first focuses on the customer selection constraints to find a feasible selection and then uses greedy heuristics for testing insertions. Thus, MSLNS seems to be faster in each insertion phase and can restart more frequently, leading to more diversification.

Next, Table 6.5 reports the quality gap obtained when a higher CPU time limit is allowed, i.e. 5 minutes. We observe that the quality gap is further improved in most cases, where the computational time required to find the best-found solutions is about 70 seconds, on average.

M	Instance set	*GenOP*-MSLNS (5min)		
		basic	TW	TW+Lopt
1	Solomon	**0.58**	**0.58**	**0.58**
	Cordeau	**0.00**	**0.00**	**0.00**
2	Solomon	**-0.16**	**-0.16**	**-0.16**
	Cordeau	-3.66	-3.64	**-3.73**
3	Solomon	**-0.73**	-0.69	-0.68
	Cordeau	-4.17	**-4.23**	-4.12
4	Solomon	**-1.51**	-1.32	-1.27
	Cordeau	**-3.22**	-3.18	-3.18
Summary				
Gap (%)	Solomon	**-0.45**	-0.40	-0.38
	Cordeau	**-2.76**	**-2.76**	**-2.76**
	All	**-1.61**	-1.58	-1.57
Time (s)	Solomon	48.38	52.43	53.46
	Cordeau	92.12	76.56	77.49
	All	70.25	64.49	65.48

Table 6.5 – Quality gaps (%) and computational times (seconds) of the MSLNS variants on the MC-TOPTW benchmark (5-minute timeout)

In addition, we report in Table 6.6 the percentage of best-known solutions found by all solvers, for different numbers of vehicles, given the baseline provided by Souffriau (2010). Notably, for each solver, we distinguish the cases where the given high-quality solution is found (Line %BK) and the case where this solution is improved (Line %newBK, highlighted in green).

Reference	Algorithm		Percentage of (new) best known solutions $M=1$	$M=2$	$M=3$	$M=4$	Average
Souffriau et al. (2013)	GRILS	%BK	70.27	21.62	8.11	8.11	35.81
		%newBK		16.22	13.51	5.41	
GenOP (Ours, 5mins)	MSLNS-basic	%BK	86.48	56.75	13.51	8.11	89.85
		%newBK		37.83	75.67	81.06	
	MSLNS-TW	%BK	86.48	54.05	16.21	8.11	89.85
		%newBK		40.54	72.97	81.06	
	MSLNS-TW-Lopt	%BK	86.48	54.05	13.51	10.81	89.16
		%newBK		40.54	72.97	78.31	

Table 6.6 – Percentage of best-known solutions obtained by different algorithms on the MC-TOPTW benchmark

Overall, MSLNS outperforms GRILS in all cases, yet this comparison is not fair due to the significant difference in computational time. Despite that issue, we remark that the three MSLNS variants achieve very good results, where nearly 90% of best-known or *new* best-known solutions are found, on average. Also, MSLNS performs equivalently well for different numbers of vehicles (M). Besides, we observe that the use of additional constraints in MSLNS+TW or MSLNS+TW+Lopt is slightly effective when $M = 2$. In summary, the MSLNS variants also manage to find new best-known solutions for up to 73 out of 148 cases, which are detailed in Table 6.7.

M	Inst.	Old BK	New BK	M	Inst.	Old BK	New BK	M	Inst.	Old BK	New BK
2	c103	700	720	3	pr07	713	737	4	c108	1100	1120
2	c108	670	680	3	pr08	1082	1120	4	pr02	1014	1073
2	pr01	471	493	3	pr09	1144	1223	4	pr03	1162	1207
2	pr02	660	700	3	r103	720	723	4	pr04	1452	1525
2	pr03	714	728	3	r104	765	767	4	pr05	1665	1723
2	pr04	863	918	3	r105	609	611	4	pr07	840	861
2	pr05	1011	1080	3	r107	747	750	4	pr08	1267	1319
2	pr07	552	558	3	r109	699	702	4	pr09	1460	1503
2	pr08	796	813	3	r110	711	734	4	r102	807	823
2	pr09	867	874	3	r111	764	765	4	r103	878	888
2	r107	529	532	3	r112	758	768	4	r104	941	969
2	r108	549	554	3	rc101	604	616	4	r105	735	765
2	r110	515	518	3	rc102	698	707	4	r106	870	893
2	r112	515	517	3	rc103	747	757	4	r109	866	884
2	rc102	494	495	3	rc105	654	657	4	r110	870	905
3	c102	890	900	3	rc106	678	690	4	r111	935	937
3	c103	960	980	3	rc107	745	753	4	r112	939	957
3	c106	840	870	3	rc108	757	765	4	rc103	947	967
3	c108	900	910	4	c101	1000	1010	4	rc104	1019	1050
3	c109	950	960	4	c102	1090	1140	4	rc105	841	843
3	pr01	598	613	4	c103	1150	1180	4	rc106	874	895
3	pr02	899	938	4	c104	1220	1230	4	rc107	951	961
3	pr03	946	992	4	c105	1030	1040	4	rc108	998	1016
3	pr04	1195	1232	4	c106	1040	1060				
3	pr05	1356	1446	4	c107	1100	1110				

Table 6.7 – New best-known solutions found by different MSLNS variants on the MC-TOPTW benchmark (73 out of 148 cases)

6.5.3 MC-TOP-MTW benchmark

To generate a benchmark for the case of MC-TOP-MTW, Souffriau et al. (2013) extend the existing MC-TOPTW benchmark instances by modifying the time windows associated with each customer. More precisely, each time window is divided into four equally reduced time windows. Additional conditions are set to ensure that a high-quality TOPTW solution is still feasible for the case of MC-TOP-MTW. For detailed description of the generation of these benchmark instances, we refer to the work of Souffriau et al. (2013).

Experimental settings We consider here two metaheuristics developed for the MC-TOP-MTW benchmark including the GRILS of Souffriau et al. (2013) and the SA metaheuristic presented by Lin & Vincent (2015). More precisely in the work of Lin & Vincent (2015), the authors experiment with four different variants of the SA algorithm based on whether the acceptance probability of a worse solution is determined by using Boltzmann function (BF) or Cauchy function (CF), and with or without restart (RS). Their report indicates that SA with a restart strategy is better than the no-restarting version, hence we report here only the results of their two best variants, namely SA-RSBF and SA-RSCF. In the following experiments, we analyze the best solution found and the corresponding total computational time for each solver.[4] The baseline results are retrieved from the work of Lin & Vincent (2015).

For technical information, the experiments of SA are run on an Intel Core 2 @ 2.5 GHz but no further details are provided; thus, we do not discuss here the difference in the computational setups. Similarly to the previous experiments on the MC-TOPTW benchmark, we simply run *GenOP*-MSLNS for a *1-minute timeout* to obtain results within a similar computational time. Then, the *GenOP* solver is given more time (5 minutes) to hopefully find new improved solutions and/or to assess the difficulty of each problem instance.

Numerical analysis Table 6.8 shows the results obtained by all solvers within a similar computational time and on different cases. Regarding the quality gap, MSLNS-basic is very competitive with the other solvers. Interestingly, we see that GRILS performs better on Solomon instances, while

[4]For GRILS and the variants of SA, computational times are reported from the original papers.

M	Instance set	Souffriau et al. (2013) GRILS	SA(Lin & Vincent (2015)) RSBF	SA(Lin & Vincent (2015)) RSCF	*GenOP*-MSLNS (1min) basic	*GenOP*-MSLNS (1min) TW	*GenOP*-MSLNS (1min) TW+Lopt
1	Solomon	**0.23**	0.47	0.47	1.41	1.41	2.52
	Cordeau	3.53	1.82	2.44	**0.00**	**0.00**	2.47
2	Solomon	**1.93**	3.86	2.26	4.26	4.66	4.84
	Cordeau	6.21	4.16	3.70	4.56	5.82	**3.11**
3	Solomon	4.15	3.50	**2.77**	6.59	6.64	8.11
	Cordeau	6.07	4.67	4.75	**3.76**	6.04	6.25
4	Solomon	5.40	4.17	**3.53**	6.02	9.62	7.71
	Cordeau	6.93	4.51	**3.98**	5.34	6.85	7.14
Summary							
Gap (%)	Solomon	2.93	3.00	**2.26**	4.57	5.58	5.79
	Cordeau	5.69	3.79	4.12	**3.20**	4.36	5.42
	All	4.31	3.40	**3.19**	3.88	4.97	5.61
Time (s)	Solomon	11.93	13.47	13.17	19.43	22.65	19.98
	Cordeau	27.28	22.05	22.67	24.00	26.38	25.49
	All	19.61	17.76	17.92	21.71	24.51	22.73

Table 6.8 – Results obtained on the MC-TOP-MTW benchmark: quality gaps (%) obtained by different algorithms (upper table); average quality gaps (%) and average computational times (seconds) per instance set (lower table)

MSLNS gives better results on Cordeau instances. MSLNS-basic even gives the best gap when averaging over Cordeau instances. Specifically, MSLNS can quickly find the optimal solutions on Cordeau instances using 1 vehicle, while the other solvers are still far from the optimum.

In general, MSLNS-basic seems to be the best version among the three variants of MSLNS. The impact of the conflict analysis procedure is also negligible when dealing with this benchmark. Although MSLNS+TW+Lopt running within 1 minute outperforms the other variants on Cordeau instances using $M = 2$, MSLNS-basic outperforms the other variants when allowing a larger CPU time limit of 5 minutes, as shown in Table 6.9. Also, for all variants of MSLNS, the computational time required to find the best solution is about 100 seconds, on average.

Moreover, Table 6.10 summarizes the average percentage of best-known solutions found by each solver and per number of vehicles. Also, we present the percentage of cases where a new best-known solution is found by our MSLNS variants, which are highlighted in green (Line %newBK). Overall, MSLNS-basic gives the best average of 45.94% of best-known or new best-known solutions over all instances, but this does not imply that MSLNS outperforms the others due to the differences in computational time. Besides, the results obtained show that MSLNS performs worse than the other solvers in the mono-vehicle case ($M = 1$), but is particularly effective when dealing

M	Instance set	*GenOP*-MSLNS (5min)		
		basic	TW	TW+Lopt
1	Solomon	**1.41**	**1.41**	2.52
	Cordeau	**0.00**	**0.00**	2.13
2	Solomon	**0.81**	1.62	1.28
	Cordeau	**0.70**	2.15	1.92
3	Solomon	**2.97**	3.60	4.33
	Cordeau	**1.88**	2.07	2.84
4	Solomon	**3.97**	6.08	5.70
	Cordeau	**2.55**	4.87	4.26
Summary				
Gap (%)	Solomon	**2.29**	3.18	3.46
	Cordeau	**1.28**	2.27	2.79
	All	**1.78**	2.73	3.12
Time (s)	Solomon	84.21	85.39	92.77
	Cordeau	110.57	117.93	101.15
	All	97.39	101.66	96.96

Table 6.9 – Quality gaps (%) and computational times (seconds) of the MSLNS variants on the MC-TOP-MTW benchmark (5-minute timeout)

Reference	Algorithm		Percentage of (new) best known solutions				Average
			$M=1$	$M=2$	$M=3$	$M=4$	
Souffriau et al. (2013)	GRILS	%BK	83.78	37.83	10.81	8.10	35.10
Lin & Vincent (2015)	SA-RSBF	%BK	86.48	40.54	18.91	8.10	38.51
	SA-RSCF	%BK	86.48	51.35	18.91	8.10	41.21
GenOP (Ours, 5mins)	MSLNS-basic	%BK	78.37	67.57	18.91	8.10	45.94
		%newBK		2.70	2.70	5.40	
	MSLNS-TW	%BK	78.37	62.16	8.10	2.70	39.18
		%newBK			2.70	2.70	
	MSLNS-TW-Lopt	%BK	67.56	59.45	18.91	10.81	40.53
		%newBK		2.70		2.70	

Table 6.10 – Percentage of best-known solutions obtained by different algorithms on the MC-TOP-MTW benchmark (the percentage of new best-known solutions is highlighted in green)

with $M = 2$ (up to 67.57% of best-known solutions are found by MSLNS-basic while the best version of SA, SA-RSCF, gets an average of 51.35%). Overall, several new best-known solutions are found by different variants of MSLNS. These solutions are detailed in Table 6.11.

M	Inst.	Old BK	New BK	M	Inst.	Old BK	New BK
2	pr02	660	663	4	c102	1110	1120
3	c106	840	850	4	pr02	1014	1018
3	rc105	654	657				

Table 6.11 – New best-known solutions found by one of the MSLNS variants on the MC-TOP-MTW benchmark (5 out of 148 cases)

6.5.4 TD-OP-MTW benchmark

This benchmark contains TD-OP-MTW instances where customers possibly have *multiple time windows* and the transition function is *time-dependent*. It is generated from an existing TD-TSP with multiple time windows benchmark (called a TD-TSP-MTW) related to urban delivery problems.[5]

Experimental settings In this experiment, we consider the case of a mono-vehicle problem and test with 20 instances, each containing 100 customers. Remarkably, for each existing TD-TSP-MTW instance, there is a best-known makespan m during which all the customers can be visited with only 1 vehicle. This problem can be seen as a TD-OP-MTW by defining a unit reward for each customer (i.e. $R_i = 1$) and by defining $T_{max} = m$. Therefore, in this case, the *GenOP* solver should be able to visit *all* customers. The instances are generated based on a transition duration function obtained from data $matrix00.txt$ available in the initial benchmark.

For these new TD-OP-MTW instances, there is no competitor since we are not aware of a solver available for such a kind of problems. Therefore, we conduct experiments using only MSLNS with a timeout of 1 minute. For this benchmark, we only test MSLNS-basic (the simplest version) due to the results obtained on previous benchmarks compared to the two other variants.

Numerical analysis As shown in Table 6.12, *GenOP*-MSLNS can quickly find a feasible TSP solution (i.e. a visit sequence that can traverse all the customers) for several instances within a reasonable computational time. However, there exist difficult cases where *GenOP* is still far from the optimal solution (about 70 out of 100 customers are visited). On this point, a possible explanation is that the customer selection strategy of MSLNS is randomized and the insertion method used is based on a local greedy heuristic. In addition, the transition function for this benchmark does not necessarily satisfy the triangular inequality meaning that the shortest path between two customers may not be the direct one. Therefore, the *GenOP* solver that inserts customers one by one using local information can get stuck at some point while a global solver considering all customers simultaneously can manage to find a valid solution.

[5] Available at `http://perso.citi-lab.fr/csolnon/TDTSP.html`

Instance	#nVisits	time (s)	Instance	#nVisits	time (s)
inst_100_1	**100**	16.61	inst_100_11	63	2.13
inst_100_2	63	13.52	inst_100_12	**100**	1.42
inst_100_3	**100**	4.12	inst_100_13	**100**	16.14
inst_100_4	68	1.34	inst_100_14	69	7.21
inst_100_5	99	0.79	inst_100_15	99	10.23
inst_100_6	64	2.81	inst_100_16	70	0.29
inst_100_7	63	7.53	inst_100_17	**100**	1.29
inst_100_8	**100**	30.72	inst_100_18	67	3.86
inst_100_9	62	18.77	inst_100_19	64	59.38
inst_100_10	**100**	1.28	inst_100_20	**100**	4.13

Table 6.12 – Results obtained by MSLNS-basic on 20 TD-OP-MTW instances (1-minute timeout); each instance involves 100 customers

Overall, the results obtained on this benchmark indicate that *GenOP* is applicable to the time-dependent versions of the OP problems. This opens up opportunities for testing the framework on the standard TDOP or TDOPTW instances of Verbeeck et al. (2013, 2014) as well as practical instances of the satellite scheduling problem generated by Pralet (2023).

6.6 Enhancements of the routing module

Apart from the enhancements of the knowledge in the *selMgr* module, we can also consider the improvement of the *routingMgr* module to make better decisions at the sequencing level. In fact, the ImaxLNS solver employed within *routingMgr* offers several advanced functions that can be used to enhance the customer insertion procedure, which are briefly described below. We refer to the work of Pralet (2023) for further details.

Enhanced insertion method Instead of using the greedy insertion heuristic, the ImaxLNS solver offers a well-tuned insertion procedure by exploiting the precedence graph of the problem. The latter expresses that some customers must necessarily be visited before others due to the time window constraints and the transition time function. In short, the reinsertion step is achieved by a dynamic programming algorithm whose complexity is bounded by a specific parameter.

Reordering procedure A reordering operation, namely a `reorder`() function, is also offered in the existing ImaxLNS solver, whose objective is to minimize first the makespan (time at which the vehicle arrives at the final

depot) and then, as a secondary objective, the sum of the transition times required between the successive visits. This reordering procedure can help us gain some free time to insert new customers. But one key point is that we need to avoid re-initializing all data structures after each customer insertion or reordering procedure. So, in ImaxLNS, the algorithm incrementally updates the precedence graph of the problem, to take into account all customer removals and insertions made since the last call to the `reorder`() function.

Modified repair procedure for GenOP-MSLNS The advanced procedures described above are shown to be very effective for solving TSPTWs both with or without time-dependent transition times. In essence, it would be possible to invoke the reordering function and/or the enhanced insertion function for each customer insertion operation in MSLNS. The intuitive idea is to make the best sequencing decision as possible at each step. However, we empirically observe that this is not efficient, especially for instances containing hundreds of customers. Based on this observation, we consider two key modifications to the previous MSLNS procedures shown in Algorithms 6.5 and 6.7.

First, we can allocate more effort to the insertion of mandatory customers (i.e., $[x_{iv} = 1]$) suggested by *selMgr* (Algorithm 6.7, Lines 4-8). Precisely, instead of testing the insertion of a mandatory customer using a greedy heuristic (Algorithm 6.7, Line 6), we first rearrange the visit sequence (by using the `reorder`() function) and then insert the mandatory customer in the reordered visit sequence. For testing the insertion of other (non-mandatory) candidate customers, only the greedy heuristic is used to get quick updates.

Second, the reordering procedure is also called when a new locally optimal solution is found (Algorithm 6.5, Line 19), which increases the possibility of obtaining a new best solution from this local optimum.

Experimental settings To better observe the impact of the enhancements detailed before, we conduct experiments only on the MC-TOP-MTW benchmark as the quality gap is still far from the best-known solutions in this case. The tests are performed only with the MSLNS-basic variant since the benefits of the conflict analysis procedure are not evident in the previous experiments. For ImaxLNS, we directly reuse the best configuration provided by Pralet (2023).

Numerical analysis Table 6.13 shows the quality gaps obtained by the standard version and the enhanced version of the MSLNS-basic algorithm for different time limits (1 or 5 minutes). From this table, we observe that, within a 1-minute timeout, the enhanced version outperforms the standard version of MSLNS-basic on Cordeau instances for different numbers of vehicles. However, when using a larger time limit of 5 minutes, the standard version performs better than the enhanced version in most cases. As a result, it appears that investing effort in the routing aspect is beneficial when there is a short time limit, but the benefit becomes negligible when allowing for a larger CPU time.

		GenOP-MSLNS-basic (1min)		*GenOP*-MSLNS-basic (5 mins)	
M	Instance set	standard	enhanced	standard	enhanced
1	Solomon	**1.41**	**1.41**	**1.41**	**1.41**
	Cordeau	**0.00**	**0.00**	**0.00**	**0.00**
2	Solomon	4.26	**4.18**	**0.81**	1.16
	Cordeau	3.70	**2.85**	**0.70**	1.34
3	Solomon	**6.59**	7.33	**2.97**	3.12
	Cordeau	3.76	**3.58**	**1.88**	2.18
4	Solomon	**6.02**	6.13	3.97	**3.94**
	Cordeau	5.34	**4.01**	**2.55**	2.57
Summary					
Gap (%)	Solomon	**4.57**	4.77	**2.29**	2.41
	Cordeau	3.20	**2.61**	**1.28**	1.52
	All	3.88	**3.69**	**1.78**	1.97

Table 6.13 – Impact of the enhancements of the routing module on the MC-TOP-MTW benchmark (gap in %)

Based on these findings, we can observe the impact of the intensification and diversification in metaheuristics. In fact, it is hard to determine whether using more intensification (focusing on optimizing the routing part frequently) or more diversification (using greedy routing heuristics and quick restarts) is the best approach. In the broader sense, this concept is also somewhat related to the effort required in the propagation phase in CP solving: using light constraint propagation rules is fast (as our greedy insertion procedure) while using advanced constraint propagation rules is more accurate in terms of inconsistency detection (as our insertion procedure exploiting reordering operations).

6.7 Related works and discussion

The techniques introduced in this chapter can be compared with the existing heuristics, local search, and metaheuristics proposed for OPTW and its extensions (Kantor & Rosenwein (1992); Solomon (1987); Vansteenwegen et al. (2009c)). In fact, the *GenOP* solver does not use fine-tuned heuristics like other approaches for OP and its variants, which rely on information such as the increase in rewards for a customer selection, the time shift when inserting a customer into a visit sequence, and/or the slack consumption of knapsack constraints (Souffriau et al. (2013); Gunawan et al. (2015b); Schmid & Ehmke (2017)). Rather, the *GenOP* solver intends to balance the work performed for selecting the right customers and the work performed for finding feasible sequences of visits. In general, our main contributions are that the solver proposed (1) is applicable to a larger class of problems (complex variants of orienteering problems), (2) has a modular conception that allows state-of-the-art methods for TDTSPTW and selection problems to be directly reused.

From a wider perspective, the idea of decomposing a problem into several sub-problems represented in different modeling frameworks is not new in optimization. In mathematical programming, such an approach is used in Logic-Based Benders Decomposition (Hooker & Ottosson (2003)), where solutions obtained by a master LP solver are analyzed by a solver tackling a sub-problem. If this solution is infeasible, the solver for the sub-problem returns *Benders cuts* (new linear constraints) that are integrated into the master solver. Hence, at each iteration, the master solver somehow learns new constraints. The analogy with the *GenOP* solver is that the conflicts sent to the selection problem from the analyses performed by the TDTSPTW manager are kinds of cuts that are learned following selection mistakes. In the same spirit, SAT Modulo Theory (SMT) (Barrett et al. (2009)) also uses a hybrid approach when SAT techniques are combined with techniques developed for specific *theories* such as the linear arithmetic theory. In this case, the SAT solver selects the linear constraints that should be satisfied and sends them to the theory reasoner, which possibly returns inconsistency explanations. In our case, *selMgr* can play a master role by proposing customer selections to the *routingMgr* modules. The latter is then used for checking feasibility at the *theory* level, which is a standard routing problem. If an inconsistency is found, several explanations might be returned to enhance

the constraint base in *selMgr*.

Last, the modularity of the *GenOP* solver architecture proposed and the propagation performed by the *selMgr* and *routingMgr* modules can be related to Constraint Programming (CP). In CP, models are expressed by a set of constraints and there exists a specific reasoning method for each constraint. We use the same kind of approach here in the sense that the selection subproblem and the routing subproblem can be seen as two global constraints for which specific and efficient reasoning mechanisms can be used. One difference though is that we exploit local search techniques instead of tree search with backtracking. In some senses, our work is actually closer to Constraint-Based Local Search (CBLS (Hentenryck & Michel (2005))), which combines constraint-based modeling and local search techniques. With regards to CBLS, the neighborhoods considered in the *GenOP* solver are much larger, especially the neighborhood that tries to reorder the sequences of visits at the routing level.

6.8 Conclusion

This chapter introduced a hybrid resolution approach for solving a general class of complex routing problems, possibly involving multiple vehicles, multiple time windows, multiple side constraints, and/or time-dependent transition times. This approach is based on a modular architecture where (1) the *GenOP* solver at the upper level is free to choose any metaheuristic to search for high-quality solutions, (2) efficient solvers at lower levels dedicated to specific sub-problems are exploited, with an effort to adapt these solvers to a dynamic context where customers can be added or removed in any order, (3) inconsistency explanations are possibly generated and recorded during search for enhancing the knowledge available in the selection sub-module. The experimental results show that despite being quite general, the approach proposed obtains competitive results on various benchmarks of OP variants including standard OPTW, MC-TOPTW, MC-TOP-MTW, and generated TD-OP-MTW instances. The approach also manages to find new best-known solutions on numerous instances.

CHAPTER 7

Conclusion and perspectives

In a general sense, our primary research objective was to enhance incomplete search algorithms for vehicle routing problems with profits. For this purpose, our study focused on developing hybrid optimization methods that exploits techniques commonly used in complete search.

More specifically, we first studied the generation of *'nogoods'* (inspired by the term *nogood* constraints) to help a specific incomplete search process. The core principle is to develop a *hybrid (integrative) approach* between incomplete search and satisfaction techniques, where several *clauses* are *lazily* generated and *incrementally* managed during the search process. In the context of an OPTW, we proposed a dynamic programming-based algorithm to extract clauses that represent several types of constraints/conflicts, namely (i) *TW-conflicts* that are generated based on the temporal constraints, and (ii) *Lopt-conflicts* that are generated based on the local optima obtained. The former type of conflict was used for neighborhood pruning while the latter one was used for search diversification. In addition, we performed an empirical study on three data structures with the objective of managing the clauses in an efficient way i.e. without diminishing the performance of the main search process. The numerical results obtained showed that the management employing two-watched literal techniques (commonly used in SAT) along with some enhancements yields the best performance, i.e. improves the solution quality and speeds up the search process. Meanwhile, the two other techniques (the use of a powerful incremental SAT solver and the compilation into an Ordered Binary Decision Diagram), appear to be ineffective

in this context, despite their effectiveness in related domains.

In the opposite direction, we studied the exploitation of *'goods'* taking the form of a pool containing high-quality solutions obtained from a specific incomplete search algorithm for the (TD)OPTW (i.e. with or without time-dependent transition times). Precisely, we defined a novel Route Recombination (RR) procedure based on dynamic programming to search for an optimal combination of k visit subsequences. The complexity of this procedure is controlled by using only two parameters. From a technical standpoint, the RR procedure can be hybridized with other black box incomplete search techniques in a *collaborative* way, where RR can post-optimize the solutions provided by a metaheuristic with the hope of finding a new solution having a higher quality. From a knowledge compilation perspective (Bryant (1986); Darwiche & Marquis (2002)), the RR procedure proposed shares similarities with algorithms used to recursively build so-called restricted decision diagrams. In this study, the effectiveness of the approach proposed was evaluated on two mono-vehicle benchmarks including the standard OPTW instances and a realistic dataset, named *singlesat*, corresponding to the satellite scheduling problem with or without time-dependent transition times. More specifically, experiments were conducted for both *deterministic* and *non-deterministic* use cases where in the latter, some information (e.g. the rewards associated with the customers) may be perturbed in an online scenario. Our experimental results showed that: (i) in the deterministic case, RR quickly improves the solution quality obtained by a standard LNS metaheuristic, when being used as a post-optimization module; (ii) in the case of non-deterministic scenarios, by using a 'pretrained' pool containing high-quality solutions generated from similar scenarios in the offline phase, RR also outperforms a standard LNS metaheuristic within a short time limit in the online phase.

Last, we developed a generic framework, called the GenOP solver, to solve complex variants of the vehicle routing problem with profits that possibly involve multiple vehicles, multiple time windows, multiple side constraints, and/or time-dependent transition times. We implemented this framework in a modular architecture that involves two low-level reasoners: one reasoner dedicated to the selection of customers, and another one dedicated to the sequencing problems given a specific set of customers. At the upper level, a generic GenOP solver can freely choose a specific metaheuristic while

interacting with the low-level reasoners. Technically, for the selection subproblem, we managed selection constraints as pseudo-Boolean constraints, whereas for the sequencing subproblem, we employed a state-of-the-art TDT-SPTW solver. From a constraint programming perspective, the selection and sequencing subproblems can be seen as two global constraints for which efficient reasoning mechanisms can be used to support the core search. Also, the GenOP solver can be related to Constraint-Based Local Search (CBLS) which integrates constraint-based modeling and local search techniques. From a wider perspective, the modular framework proposed is somehow in the same spirit as SAT Modulo Theory (SMT), which combines SAT techniques with powerful techniques developed for a specific theory. In our case, the counterpart of the SAT problem is the selection problem and the counterpart of the theory solver is the module that handles the sequencing problems (i.e. TDT-SPTWs). Overall, the experimental results obtained on various benchmarks highlighted the effectiveness of the approach proposed despite its genericity.

Perspectives

This dissertation has explored various hybrid approaches that could be further investigated. Each of the questions tackled in this work offers several potential implications and extensions for future research.

Local search and nogood learning In the literature, nogood learning appears to be effective in improving the performance of complete techniques for SAT/CSP solving. However, there are few works exploiting the nogood learning mechanism to boost incomplete search methods on specific problems considered in the Operations Research community. Concerning our first contribution (Chapter 4) and the nogood management techniques, several perspectives can be listed.

- First, it is promising to apply the framework proposed to other problems involving two decision levels, for instance, the flexible scheduling problems (Chaudhry & Khan (2016)). Briefly, such problems can be also decomposed into two subproblems: (1) an assignment subproblem that consists in assigning operations to machines (one machine-choice variable $x_i \in [1..M_i]$ per operation i), and (2) a scheduling subproblem corresponding to the sequencing of operations on machines, given an

upper bound UB on the makespan. Given that the generic clause managers are defined, the main effort to tackle a new problem is to define the problem-dependent clause generation procedure.

- Second, rather than exploiting nogood information only in an on-the-fly manner, some conflicts could be compiled into a compact form during a preprocessing step and reused to help the search, e.g. for neighborhood pruning and/or search guidance.

- Third, it would be interesting to explore other clause managers (e.g. based on 0/1 linear programming and reduced-cost filtering) or other knowledge bases represented as exact, relaxed, or restricted decision diagrams (Andersen et al. (2007); Bergman et al. (2016a)).

Route recombination Regarding our second contribution (Chapter 5), the route recombination (RR) algorithm proposed can be further improved or hybridized with other methods for solving variants of the routing problems (with or without uncertainty).

- A direct future work is to integrate RR into the previous framework (i.e. incomplete search using clause bases) to improve further the quality of solutions found for a (TD)OPTW. This would also require adapting the clause generation procedure to the case of time-dependent transition times. Besides the collaborative hybridization, RR could be hybridized with other search methods in an *integrative* way, for instance, to get more powerful crossover operations within a genetic algorithm.

- Another perspective is to develop an *Adaptive RR* procedure, where several parameters could be learned, such as (1) parameters referred to as J_{max} and W_{max}, (2) the choice of the dominance rule between states, or (3) the usage of reverse jumps.

- To better evaluate the effectiveness of the online RR solver under reward uncertainty, one has to experiment with real-life data, beyond the benchmark generated in this study. There are also possible avenues to enhance the adaptation to real-world situations with the presence of other sources of uncertainty:

(1) A first example is in logistic problems, where the transition times can change due to traffic, weather, road conditions, or other factors that affect the speed of the vehicle. The adaptation is quite straightforward as the new transition times can be taken into account when extending the search states in the dynamic programming algorithm achieved by RR.

(2) Another possible scenario is that new urgent customers may arrive during the online phase. This can happen in the case of the Earth Observation Satellite scheduling when new requests are sent to the satellite just before the start of an orbit. In such a situation, one needs to adjust the schedule of the satellite and prioritize the new requests over the old ones. However, the RR method proposed may have some issues that need to be addressed. The main reason is that the online RR solver produces a new solution based on a pool containing only old customers without knowing any information about new ones.

Generic framework for complex routing problems Concerning our third contribution (Chapter 6), further enhancements can be considered to increase both the genericity and efficiency of the GenOP solver proposed.

- First, as the GenOP solver can freely choose a search strategy, other advanced metaheuristics can be tested to try to achieve a better performance. Given that fine-tuning parameters for a specific metaheuristic as well as for low-level reasoners may require a lot of work, several parameters could be learned or automatized. For instance, it would be relevant to automatically determine when to reorder the visit sequence at the routing level.

- Another perspective to explore is the design of fine-tuned selection heuristics for the GenOP solver at the upper level. The key idea is to evaluate the long-term impact of the decisions on the selection and sequencing levels, based on inputs provided by the low-level reasoners. For instance, in the presence of multiple side constraints, the heuristic search should not only prioritize increasing the profit and minimizing resource consumption for each decision but also consider the importance of the different constraints to guide the search. As pointed out

by Souffriau et al. (2013), a constraint can be seen as important if it has a high-profit potential but is challenging to gain this profit.

- Last, regarding the genericity, one can adapt the GenOP solver to tackle other variants of the routing problems, such as variants considering time-dependent profits (Peng et al. (2019)) or non-linear profit function (Wang et al. (2008)). This can be done using the same functions implemented in the low-level selection and sequencing sub-modules. Yet, it could be necessary to adjust the selection strategy to achieve a better result with the new profit function. Last but not least, the architecture proposed could be also considered for solving other types of problems involving two levels of decisions (e.g., flexible job-shop scheduling).

Part III

Appendix

ANNEXE A

Résumé étendu

A.1 Introduction

Problématique L'optimisation combinatoire se concentre sur la recherche des solutions optimales parmi un ensemble fini de combinaisons possibles, tout en respectant un ensemble de contraintes et en maximisant ou minimisant une fonction objectif. Pour résoudre ces problèmes, les méthodes incomplètes sont souvent utilisées en pratique, car elles peuvent produire rapidement des solutions de haute qualité, ce qui est un point critique dans de nombreuses applications. Pourtant, ces algorithmes ont également de nombreuses limitations : ils n'offrent pas la garantie de trouver une solution optimale et peuvent rester bloqués dans des optima locaux, et la qualité de la solution obtenue dépend fortement du choix de l'heuristique et des stratégies de recherche. C'est pourquoi il est utile de développer des techniques permettant d'éviter rapidement les optima locaux et de guider la recherche vers des régions prometteuses de l'espace des solutions. Pour cela, dans la littérature, de nombreuses approches hybrides sont proposées pour accroître l'efficacité des méthodes incomplètes en exploitant diverses idées algorithmiques issues des méthodes complètes.

Techniques hybrides mises en œuvre Le but de cette thèse est de développer des approches hybrides qui permettent d'améliorer la recherche incomplète en exploitant les méthodes complètes. Plus précisément, cette thèse présente trois contributions principales :

- Une approche hybride intégrant l'apprentissage de clauses pour améliorer l'efficacité d'une méthode incomplète (Section A.2).

- Une hybridation avec la programmation dynamique afin de produire de nouvelles solutions de meilleure qualité (Section A.3).

- Un solveur générique couplé avec des mécanismes de propagation de contraintes pour résoudre efficacement des problèmes de routage complexes (Section A.4).

Cas d'étude Les approches hybrides proposées sont appliquées aux *problèmes de routage avec profits* et avec clients optionnels. En particulier, nous nous intéressons à une application pertinente dans le domaine aérospatial qui est le problème d'ordonnancement de prises de vue pour les satellites d'observation de la Terre, incluant éventuellement des profits incertains.

Dans un le problème de routage avec profits, on considère un ensemble de N clients optionnels pouvant être visités, chacun apportant un certain profit. Parmi les variantes de ce problème dans la littérature, le *problème d'orienteering (OP)* semble être le problème le plus étudié. Dans ce problème, l'objectif est de maximiser le profit total collecté en visitant les clients plutôt que de minimiser le coût total (distance ou temps de trajet). Ce problème peut être défini formellement comme suit.

Considérons $I = \{0, \ldots, N+1\}$ un ensemble de nœuds (ou clients) qui peuvent être visités, où 0 et $N+1$ sont considérés comme les nœuds de départ et d'arrivée. Chaque nœud $i \in I$ est associé à une récompense (ou profit) non négative R_i, avec $R_0 = R_{N+1} = 0$. Pour les contraintes temporelles, chaque nœud $i \in I$ peut avoir une (ou plusieurs) fenêtre(s) temporelle(s) $[O_{ik}, C_{ik}]$ autorisé(s) pour commencer la visite. De plus, une durée de transition non négative entre les nœuds i et j existe pour chaque paire de nœuds distincts $i \in I, j \in I$. Cette durée de transition peut être *"time-dependent"*, c'est-à-dire qu'elle peut dépendre de la date précise à laquelle la transition est réalisée. Une durée de visite d_i peut être considérée pour chaque nœud $i \in I$, cependant nous supposons que ces durées de visite sont déjà incluses dans les coûts de déplacement. En plus des contraintes temporelles, plusieurs variantes complexes de ce problème peuvent également impliquer des contraintes supplémentaires au niveau de sélection, telles que des contraintes de capacité.

En général, une solution de l'OP est une séquence $\sigma = [i_0, \ldots, i_{q+1}]$ qui commence et se termine aux deux dépôts ($i_0 = 0$, $i_{q+1} = N+1$), et qui

visite un sous-ensemble de nœuds distincts $S = \{i_1, \ldots, i_q\} \subseteq \{1, \ldots, N\}$. Pour la variante impliquant plusieurs véhicules, une solution correspond à un ensemble de M séquences de visites $\{\sigma_1, \ldots, \sigma_M\}$, où chaque nœud peut être visité au plus une fois. Une solution est dite *faisable* si et seulement si chaque nœud est visité dans une fenêtre temporelle autorisée, et le coût total de chaque trajet ne dépasse pas T_{max}, un budget limité prédéfini. L'objectif est alors de trouver une solution optimale σ^*, qui soit faisable et qui maximise la récompense totale collectée (c'est-à-dire $\sum_{v=1}^{M} \sum_{i \in \sigma_v} R_i$). Globalement, ce problème implique deux niveaux de décision, avec d'un côté la sélection des clients à visiter, et de l'autre côté le séquencement des visites des clients sélectionnés.

A.2 Recherche incomplète aidée par une base de clauses

A.2.1 Motivation et schéma général

Notre première contribution combine la recherche incomplète avec des techniques d'apprentissage de clauses souvent utilisées en SAT. L'idée principale est d'exploiter une mémoire long terme pour améliorer l'efficacité de la recherche, un peu comme dans une recherche tabou (Glover (1989)), mais avec des mécanismes qui s'inspirent de méthodes de recherche complète efficaces exploitant la génération de clauses (Marques-Silva et al. (2009); Schutt et al. (2013)). À partir de ces idées, nous proposons une architecture hybride décrite à la figure A.1. Cette architecture se compose de trois éléments principaux : une *procédure de recherche incomplète* (ISP), un *générateur de clauses* (CG), et une *base de clauses* (CB).

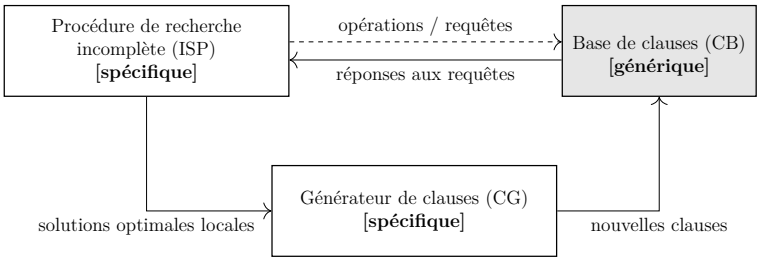


FIGURE A.1 – Recherche incomplète aidée par une base de clauses

Le schéma de recherche global fonctionne comme suit. Chaque fois que le moteur de recherche principal (ISP) converge vers une solution localement optimale, le module CG analyse cette solution et produit des clauses portant sur des variables de décision booléennes du problème. Les clauses générées représentent soit les raisons pour lesquelles la solution actuelle ne peut pas être améliorée, soit des conditions interdisant l'optimum local trouvé d'être rencontré à nouveau dans le futur. Ces clauses sont ensuite envoyées à une base de clauses (CB). Cette dernière est responsable du stockage des clauses sur le long terme. Elle est aussi exploitée pour répondre à diverses requêtes qui sont pertinentes pour la recherche incomplète principale ISP afin d'élaguer ou de guider l'exploration du voisinage d'une solution courante. Dans cette architecture, les clauses sont générées de manière paresseuse (lazy), c'est-à-dire uniquement pour les parties de l'espace de recherche que ISP décide d'explorer. Au final, cette architecture implique d'un côté des interactions très fréquentes entre les modules ISP et CB, et de l'autre une phase de génération de clauses moins fréquente afin de ne pas ralentir la recherche principale.

Application au problème OPTW Pour évaluer cette approche, nous considérons ici le problème d'orienteering avec fenêtres temporelles (OPTW). Pour un OPTW, une méthode de référence est l'algorithme LNS proposé par Schmid & Ehmke (2017) qui consiste à itérer des phases de destruction et de réparation de la solution courante. Ces itérations sont généralement effectuées de façon totalement indépendantes, c'est-à-dire qu'une itération donnée n'exploite pas du tout des connaissances qu'il aurait été possible d'acquérir lors des itérations précédentes. Partant de ce constat, nous envisageons d'*extraire* des connaissances au fur et à mesure des itérations (A.2.2) et de *mémoriser* ces connaissances acquises (A.2.3) pour éviter de réexplorer des configurations non admissibles ou pour identifier les zones les plus prometteuses dans l'espace de recherche.

A.2.2 Génération de clauses pour un OPTW

Dans ce travail, nous ne considérons que les clauses relatives aux décisions de sélection, et non les clauses relatives aux décisions de séquencement définissant l'ordre des visites. En fait, l'élagage de voisinage au niveau de la sélection peut contribuer à réduire de manière significative l'espace de recherche.

En ce qui concerne l'aspect sélection, nous introduisons une variable de décision booléenne $x_i \in \{0, 1\}$ associée à chaque nœud client i, où $x_i = 1$ signifie que le nœud i est visité. Plusieurs types de clauses peuvent être générés au cours de la recherche et la génération de ces clauses exploite des techniques *dépendantes du problème*, comme pour les coupes générées dans la décomposition de Benders logique (Hooker & Ottosson (2003)). Pour les OPTW, nous considérons deux types de conflits possibles et nous les générons de manière "lazy", c'est-à-dire uniquement lorsqu'un minimum local σ^* est atteint.

Conflits basés sur les contraintes temporelles (conflits TW) Une première idée est d'essayer de trouver des explications minimales pour les visites localement infaisables. Plus précisément, après chaque convergence vers une solution localement optimale, l'algorithme cherche à expliquer pourquoi les clients non visités ne peuvent pas être insérés dans le plan courant. De manière intuitive, un conflit est un ensemble des clients S_c qui exprime qu'il est impossible de visiter tous les clients $i \in S_c$ en respectant leurs fenêtres temporelles. Un tel conflit peut être vu formellement comme une clause $\bigvee_{j \notin S_c} \neg x_j$ spécifiant qu'au moins un client de S_c ne doit pas être sélectionné. Autrement dit, si tous les clients de $S_c \backslash \{i\}$ sont visités, un tel conflit peut être utilisé pour détecter rapidement que le client i ne peut pas être sélectionné pendant le processus de construction d'une solution, sans examiner explicitement toutes les insertions possibles de i à chaque position dans la séquence actuelle de visites σ. Pour cela, dans l'algorithme 4.5 (voir la partie du manuscrit en anglais), nous décrivons une procédure pour générer des clauses basées sur les conflits de fenêtres temporelles en utilisant la *programmation dynamique*. Néanmoins, énumérer tous ces conflits de fenêtres temporelles est impossible. C'est pourquoi nous fixons une taille maximum pour les conflits recherchés afin de limiter le temps dédié à la procédure d'explication. De plus, nous définissons pour chaque client j un nombre maximum de tentatives d'explication de l'absence du client j dans une solution localement optimale.

Conflits basés sur les optima locaux (conflits Lopt) Une remarque importante est que, pour les instances ayant de longues fenêtres temporelles, les conflits temporels impliquent souvent un plus grand nombre de clients. Autrement dit, il existe très peu de conflits TW de petite taille générés dans ces cas. Partant de ce constat, notre idée est de diversifier la recherche en

générant une autre type de clause basé sur les solutions localement optimales σ^* explorées dans le passé. Pour cela, nous définissons un conflit Lopt comme une clause $\bigvee_{j \notin \sigma^*} x_j$ qui favorise la sélection de nœuds non visités dans σ^* pour les prochaines itérations. En pratique, ces conflits Lopt sont de tailles très longues s'il existe seulement un ensemble restreint de clients visités dans chaque tournée. Dans ce cas, ces conflits sont moins efficaces pour guider la recherche vers des régions de solutions non-explorées. Afin de parvenir à intensifier la recherche, nous considérons des conflits de taille plus petite en sélectionnant seulement K des clients non-visités dans σ^*. Cependant, un grand nombre de clauses Lopt ainsi réduites peut également augmenter le risque d'élaguer des solutions optimales. Pour surmonter ce problème, ces clauses approximatives sont gardées dans CB uniquement pendant un certain nombre d'étapes appelé *tabuSize*, de la même manière qu'une recherche tabou à court terme.

A.2.3 Gestion de la base de clauses

La partie CB est responsable du stockage des clauses générées étape par étape pendant la recherche. Idéalement, le module CB doit intégrer rapidement ces clauses et les représenter de manière compacte. De plus, comme ISP doit interroger fréquemment CB, nous avons besoin d'interactions *continues* et *incrémentales* entre ces deux composants. En d'autres termes, CB doit mettre à jour son état courant à chaque fois qu'une décision est prise par ISP. De même, CB doit également répondre rapidement aux requêtes formulées par ISP, telles que "*évaluer si la décision* $[x_i = 1]$ *est réalisable*". Pour un OPTW, si CB prouve que cette décision est infaisable compte tenu de l'affectation actuelle des variables de décision et des clauses générées par le passé, il n'est pas nécessaire de tester l'insertion du nœud i dans la solution courante σ (*élagage du voisinage*). Un autre type de requête est : "*évaluer si la décision* $[x_i = 1]$ *est obligatoire*". Si c'est le cas, le nœud i doit être inséré immédiatement dans σ (*guidage de la recherche*). En général, les opérations et les requêtes mentionnées précédemment ne sont pas spécifiques aux OPTW, donc dans ce travail, nous étudions trois approches *génériques* pour définir CB.

- CB-UnitPropagation (CB-UP) : La première approche mémorise simplement les conflits sous forme d'une liste de clauses. Elle utilise ainsi des mécanismes de *propagation unitaire* pour évaluer la cohérence

d'une prochaine décision. Plus précisément, une clause est dite "unitaire" lorsqu'il ne reste qu'un seul littéral l non assigné dans la clause, nécessitant que l soit *vrai* pour satisfaire la clause. Dans ce cas, $\neg l$ est *faux* et la propagation unitaire est utilisée pour détecter d'autres décisions propagées dans les autres clauses. Comme en SAT, la propagation unitaire peut être réalisée en se basant sur la technique des *"two-watched literals"* (Moskewicz et al. (2001)). Une contribution clé ici est l'introduction de la propagation unitaire *décrémentale* pour gérer les "backtracking" aléatoires lors de la recherche incomplète (possible pour ISP d'ajouter et retirer des décisions dans n'importe quel ordre).

- CB-IncrementalSAT (CB-SAT) : La deuxième approche mémorise toujours une simple liste de clauses mais utilise directement un solveur SAT incrémental pour déterminer très rapidement et précisément la cohérence de décision proposées par ISP vis-à-vis de la base de clauses. L'idée d'utiliser la résolution SAT incrémentale a été proposée pour la première fois par Audemard et al. (2013) dans le but d'améliorer l'efficacité de la recherche en réutilisant le plus d'informations possibles entre les résolutions successives de problèmes SAT similaires.

- CB-OBDD : La troisième approche implique l'utilisation d'un diagramme de décision binaire ordonné (OBDD (Bryant (1986))) pour stocker des conflits. Les bonnes propriétés des OBDDs permettent d'extraire une sélection optimale de clients en temp linéaire en la taille de l'OBDD, cependant cette taille peut croître très rapidement lors des ajouts successifs de clauses.

A.2.4 Résultats expérimentaux

L'approche proposée a été évaluée sur des instances classiques du problème OPTW, décomposées en quatre groupes (Solomon1, Solomon2, Cordeau1 et Cordeau2), les instances des groupes Solomon1 et cordeau1 impliquant des fenêtres plus courtes que les instances des groupes Solomon2 et Cordeau2. Concernant la partie gestion des clauses, nous considérons l'utilisation des trois structures de données proposées : CB-UP, CB-SAT et CB-OBDD. Pour chaque variante, nous mémorisons sur le long terme seulement les clauses basées sur des conflits temporels (conflits TW). Pour CB-UP, nous considérons en plus la génération des clauses provenant de conflits Lopt

afin d'évaluer leur impact sur la diversification de la recherche. Le tableau A.1 donne, sur la base de 5 résolutions avec un temps de calcul de *1 minute* pour chaque instance, les écarts moyens obtenus par rapport à la meilleure solution connue. D'autre part, le tableau A.2 donne le taux d'accélération obtenu avec chaque CB en réalisant 10 000 itérations de LNS, où un taux positif indique que la recherche est plus rapide.

Instance set	Variants of CB in LNS				
	noCB	UP	UP-Lopt	SAT	OBDD
Solomon1	**1.093**	**1.093**	1.304	1.492	1.315
Solomon2	0.416	0.387	**0.345**	0.607	0.497
Cordeau1	0.139	**0.078**	0.351	1.125	0.903
Cordeau2	1.898	**1.846**	1.900	3.729	2.958
Grand mean	0.886	**0.851**	0.977	1.739	1.418

TABLE A.1 – Écarts moyens (%) obtenus en *1 minute* en utilisant chaque variante de CB (meilleurs écarts moyens en **rouge gras**)

Instance set	Variant of CB in LNS			
	UP	UP-Lopt	SAT	OBDD
Solomon1	-8.83	-18.66	-2517.14	-646.14
Solomon2	**25.17**	**25.15**	-492.75	-163.62
Cordeau1	**48.66**	**47.04**	-2779.32	-2446.31
Cordeau2	**45.96**	**47.83**	-2092.35	-610.95

TABLE A.2 – Taux d'accélération (%) suivant l'utilisation de chaque CB

En général, la gestion des clauses avec CB-SAT et CB-OBDD détériore les performances, car elle exige beaucoup d'effort de raisonnement sur les conflits et diminue donc fortement le nombre d'itérations de recherche principale réalisées dans un temps de calcul donné. En revanche, l'utilisation de la base de clauses CB-UP a permis d'accélérer globalement la recherche et de réduire les écarts aux meilleures solutions connues sur tous les groupes d'instances. Cela s'explique par le fait que des conflits TW appris ont aidé à éviter un grand nombre d'essais d'insertion de clients qui sont incompatibles avec des clients déjà présents dans la solution courante. Aussi, l'utilisation de CB-UP-LOPT génère des résultats très compétitifs par rapport à la version de base (noCB), ce qui montre que les conflits Lopt peuvent aider à

diversifier la recherche, notamment avec des instances du groupe Solomon2. En résumé, nous constatons que la recherche aidée par CB-UP ou CB-UP-Lopt est plus rapide, et elle est efficace en termes de qualité de solutions obtenues.

A.2.5 Perspectives

Dans ce travail, nous exploitons le mécanisme d'apprentissage des "*nogoods*" pour améliorer les méthodes de recherche incomplète sur des problèmes spécifiques de la communauté Recherche Opérationnelle. Sur cette base, plusieurs perspectives peuvent être énumérées.

- Il serait intéressant d'appliquer le cadre proposé à d'autres problèmes impliquant deux niveaux de décision, tels que les problèmes d'ordonnancement avec resources flexibles (Chaudhry & Khan (2016)). Comme les gestionnaires de clauses génériques sont disponibles, l'effort principal consisterait à définir la procédure de génération de clauses pour ce problème spécifique.
- Deuxièmement, au lieu d'exploiter les connaissances uniquement à la volée, certains conflits pourraient être compilés sous une forme compacte dans une phase hors ligne et réutilisés pour faciliter la recherche dans une phase en ligne, par exemple pour élaguer les voisinages et/ou guider la recherche.
- Il serait également possible d'explorer d'autres types de gestion de la base de clauses (par exemple, basés sur la programmation linéaire 0/1 et le filtrage par coûts réduits), ou d'autres bases de connaissances représentées sous forme de diagrammes de décision exacts, relâchés ou restreints (Andersen et al. (2007); Bergman et al. (2016a)).

A.3 Recombinaison des routes par programmation dynamique

A.3.1 Motivation

Contrairement au mécanisme de génération de clauses qui exploite des *nogoods*, nous visons également à explorer des *goods*, c'est-à-dire des bonnes

caractéristiques de solutions de haute qualité. Plus précisément, notre objectif est d'optimiser a posteriori la qualité des solutions obtenues par les méta-heuristiques classiques pour le problème d'orienteering avec fenêtres temporelles (OPTW) et sa version *time-dependent* (TDOPTW). Dans ce but, nous introduisons une nouvelle procédure de recombinaison capable de prendre en entrée un ensemble de solutions et de retourner la meilleure combinaison contenant des sous-séquences de visites de clients obtenues à partir de ces solutions. Par exemple, la procédure de recombinaison peut prendre en entrée trois solutions $\sigma_1 = [\mathbf{0}, \mathbf{2}, \mathbf{3}, 4, 6, 8, 10]$, $\sigma_2 = [0, 4, \mathbf{3}, \mathbf{5}, \mathbf{8}, 1, 10]$ et $\sigma_3 = [0, 1, \mathbf{8}, \mathbf{7}, \mathbf{6}, \mathbf{10}]$, et renvoyer une solution recombinée $\sigma = [\mathbf{0}, \mathbf{2}, \mathbf{3}, \mathbf{5}, \mathbf{8}, \mathbf{7}, \mathbf{6}, \mathbf{10}]$ qui suit d'abord σ_1 puis σ_2 puis σ_3.

Toutefois, la recherche d'une telle recombinaison de sous-séquences est difficile dans le cas général, étant donné que le nombre de toutes les combinaisons possibles impliquant k sous-séquences est exponentiel en k. Pour réduire la complexité, nous pouvons limiter le nombre maximal de sous-séquences utilisées pour chaque combinaison, c'est-à-dire le nombre maximal de *jumps* entre différentes solutions élites, où un *jump* correspond à un changement dans la solution d'élite suivie à une étape donnée. Dans la suite, nous désignons par J_{max} le nombre maximum de *jumps* autorisés pour chaque combinaison. De plus, un *jump* peut être effectué vers l'avant ou vers l'arrière, la sémantique étant qu'un *jump* vers l'arrière traverse une sous-séquence de visites dans l'ordre inverse. Cette procédure de recombinaison repose sur un algorithme de *programmation dynamique* amélioré par des *stratégies d'élagage* qui réduisent de manière significative la taille de l'espace de recherche.

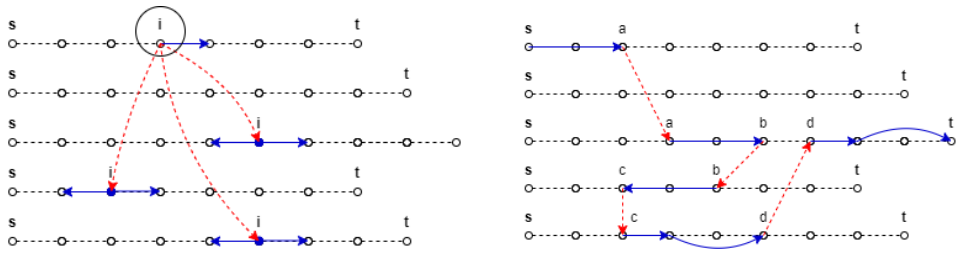


FIGURE A.2 – (a) Actions possibles (*jump* en red, direction en blue) compte tenu du dernier client visité i ; (b) une séquence possible générée avec 4 *jumps* (dans red) étant donné un ensemble de 5 séquences de visites.

A.3.2 Procédure de recombinaison (RR)

Dans ce qui suit, nous détaillons le système de programmation dynamique (DP) utilisé pour la procédure de recombinaison (appelée RR) en décrivant les *états de recherche*, les *règles d'extension* des états et les *stratégies d'élagage* pour réduire l'espace de recherche.

Notion d'état La procédure RR explore des *états de recherche* représentant les chemins faisables qui commencent et retournent au dépôt. Ces chemins sont construits en suivant des solutions d'un pool de solutions donné. De manière formelle, chaque état est défini comme un tuple $S = (V, i, r, d, n)$, où V est un ensemble de clients visités, i est le dernier client visité, r est l'itinéraire (ou la solution) actuellement suivi, $d \in \{FWD, BWD\}$ est la direction d'exploration actuelle de r (vers l'avant ou vers l'arrière), et n est le nombre de *jumps* effectués jusqu'à présent. Intuitivement, les différents états associés au même dernier client visité i correspondent à différents chemins possibles pour atteindre le client i. Pour l'évaluation de chaque état, nous considérons les deux critères définis ci-dessous :

- La récompense $R(S)$ d'un état $S = (V, i, r, d, n)$ est calculée comme la somme des récompenses R_j des clients j visités dans V, *i.e.* $R(S) = \sum_{j \in V} R_j$.

- Le *coût* d'un état $S = (V, i, r, d, n)$, noté $arr(S)$, est le temps d'arrivée minimal au nœud i si tous les clients de V ont déjà été visités. Au cours de l'exploration de l'espace de recherche, $arr(S)$ est mis à jour chaque fois qu'un nouveau chemin réalisable atteignant S est trouvé. Chaque état non initial S garde également la trace d'un *état parent* à visiter avant S afin d'obtenir le coût $arr(S)$.

Un état (V, i, r, d, n) est dit *terminal* lorsqu'il atteint le dépôt final, *i.e.* $i = N + 1$. Entre l'état initial et l'état terminal, les états intermédiaires représentent des solutions incomplètes qui visitent un sous-ensemble de clients mais ne retournent pas encore au dépôt. Pour construire le *meilleur* itinéraire entre le dépôt initial et le dépôt final, il suffit de sélectionner un état terminal qui a une récompense maximale et d'extraire le meilleur chemin correspondant par rétropropagation à travers les états parents.

Règles d'extension Les états sont explorés *niveau par niveau*, où le *niveau* d'un état $S = (V, i, r, d, n)$ correspond au nombre de nœuds visités dans V. L'extension d'un état $S = (V, i, r, d, n)$ correspond à la sélection d'une séquence de visite et d'une direction, à partir desquelles nous essayons de trouver le prochain client à visiter après i. Formellement, une telle action est représentée par une paire (ρ, δ) où ρ est une séquence du pool de solutions contenant le client i et δ est une direction (vers l'avant ou vers l'arrière). L'application de l'action (ρ, δ) depuis l'état S conduit à un nouvel état $S' = (V \cup \{i'\}, i', \rho, \delta, n')$ où un nouveau client $i' \in \rho$ est visité et où $n' = n$ si $\rho = r$ et $n' = n + 1$ dans le cas contraire. Une telle transition d'état n'est possible que si *toutes* les conditions suivantes sont remplies.

- L'état obtenu doit respecter la limite sur le nombre maximal de *jumps* J_{max}. Autrement dit, un *jump* vers une autre solution (c'est-à-dire, $\rho \neq r$) n'est autorisé que si $n < J_{max}$. En particulier, un *jump* vers arrière n'est pas autorisé s'il ne reste qu'un seul *jump* possible (cas $n = J_{max} - 1$).

- La visite du nœud i' après i est *faisable*. Plus précisément, soit $\tau = arr(S)$ l'heure d'arrivée à l'état actuel. Si $\delta = FWD$ (ou BWD), alors i' est le premier client qui suit i (ou qui précède i) dans l'itinéraire ρ tel que (1) i' n'est pas encore visité ($i' \in \{1, \ldots, N\} \setminus V$), (2) il est possible de visiter i' avant sa date de fermeture $C_{i'}$ ($\tau' = \tau + tt(i, i', \tau) \leq C_{i'}$), et (3) il est possible de retourner au nœud dépôt $N + 1$ depuis i' avant T_{max} ($\tau' + tt(i', N + 1, \tau') \leq T_{max}$).

- Si l'itinéraire sélectionné est le même que celui actuel ($\rho = r$), l'action est autorisée uniquement si la direction ne change pas ($\delta = d$). En d'autres termes, il n'est pas permis de se déplacer dans une direction opposée sur la même séquence, ce qui est appelé un *jump renversé*. Cette dernière règle peut être négligée, mais elle est introduite dans ce travail pour réduire le nombre d'états de recherche.

Stratégies d'élagage Afin d'accélérer le processus de recombinaison, deux stratégies d'élagage sont utilisées pour éliminer les états qui sont sous-optimaux ou semblent sous-optimaux en termes de récompenses.

- Premièrement, en considérant le meilleur état S^* rencontré jusqu'alors, un état S peut être élagué si $R(S) + \overline{R}(S) < R(S^*)$, où $\overline{R}(S)$ est une

borne supérieure sur le score supplémentaire maximum qui peut être collecté depuis l'état S jusqu'à un état terminal.

- La deuxième stratégie repose sur la *règle de dominance*. Intuitivement, étant donné deux états S_1 et S_2, S_2 est dit *faiblement dominé* par S_1 si ces deux états atteignent le même nœud i et visitent le même ensemble de nœuds V, mais S_1 a un coût plus faible et plus de *jumps* restants que S_2, *sans* tenir compte de l'itinéraire actuellement suivi. Dans ce cas, l'état S_2 est jugé comme moins favorable que S_1 et l'algorithme de recombinaison considère qu'il n'est pas nécessaire de développer S_2. Il convient de noter que cette règle de dominance est dite *faible* car elle peut élaguer éventuellement des états à partir desquels une meilleure récompense pourrait être obtenue à la fin.

A.3.3 Variantes de la procédure de recombinaison

Version restreinte Pour accélérer davantage le processus de recombinaison, l'algorithme utilise une stratégie supplémentaire qui limite le nombre d'états conservés à chaque niveau de décision (*largeur restreinte*). Techniquement, une fois que l'ensemble des états $\mathcal{S}_L$ au niveau L a été généré, si $\mathcal{S}_L$ contient trop d'états selon un paramètre d'entrée W_{max} (c'est-à-dire $|\mathcal{S}_L| > W_{max}$), l'algorithme sélectionne de manière heuristique les états les plus prometteurs à explorer et rejette les autres. Cette stratégie de restriction de la largeur est similaire à l'approche proposée par Gillard & Schaus (2022) concernant l'algorithme LNS couplé avec des diagrammes de décision.

Version itérative (IRR) L'utilisation d'un processus itératif peut contribuer à accroître la diversité des séquences combinées explorées par RR, même en considérant une valeur fixe de J_{max}. Techniquement, après chaque itération de RR conduisant à une amélioration, la pire solution du pool en termes de récompense est remplacée par la nouvelle solution combinée. Ce processus se poursuit jusqu'à ce qu'un point fixe soit atteint, c'est-à-dire jusqu'à ce qu'aucune amélioration ne soit trouvée. En fin de compte, même avec un nombre fixe de *jumps* J_{max}, il est possible de créer itérativement des séquences contenant plus de $J_{max} + 1$ sous-séquences contenues dans les solutions du pool original. En effet, à la première itération de RR, la solution combinée résultante incorpore des composants provenant d'au plus $J_{max} + 1$ sous-séquences

originales, mais aux itérations suivantes, les nouvelles séquences générées peuvent elles-mêmes être combinées avec d'autres solutions du pool.

A.3.4 Résultats de complexité

La complexité de RR reste efficace sous réserve que J_{max} et W_{max} sont bornées : le nombre d'états explorés par RR est polynomiale avec J_{max} borné et linéaire avec W_{max}.

A.3.5 Expérimentations

Des expérimentations ont été réalisées sur deux jeux de données. Le premier contient des instances OPTW classiques et le second implique des données réelles générées pour la planification d'observation pour un satellite d'observation (instances *singlesat*). De plus, il est important de souligner que les tests sont effectués avec des variantes dépendantes et indépendantes du temps (c'est-à-dire problèmes *time-dependent* ou non). Nous évaluons les performances de la procédure RR dans deux cas d'utilisation : (1) en tant qu'étape de post-optimisation pour des problèmes déterministes, et (2) en tant que solveur pour des OPTW non déterministes impliquant des perturbations au niveau des récompenses.

Scénarios déterministes : utilisation d'IRR comme post-optimisateur Dans ce cas, nous collectons d'abord un pool contenant les meilleures solutions trouvées par une recherche LNS, puis nous appliquons la procédure RR itérative (IRR) à ce pool de solutions élites pour rechercher une solution améliorée (voir figure A.3).

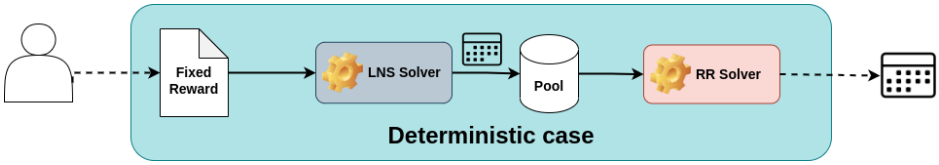


FIGURE A.3 – Utilisation d'IRR pour post-optimiser les solutions fournies par LNS dans un contexte déterministe

Globalement, les résultats décrits dans le tableau A.3 montrent que le processus de recombinaison IRR nécessite très peu de temps pour améliorer

la qualité des solutions fournies par un solveur LNS standard. Par ailleurs, l'amélioration apportée par IRR diminue lorsque le temps de calcul alloué à LNS augmente (voir les colonnes ΔGap et ΔR). Cela est raisonnable, car avec un temps de calcul plus élevé, LNS peut converger vers un meilleur optimum local, ce qui rend plus difficile pour IRR la découverte de nouvelles solutions de meilleure qualité.

CPU_{max}(s)	OPTW | AvgGap (%)			t_{IRR}(ms)	singlesat | AvgRwd			t_{IRR}(ms)
	LNS	LNS-IRR	ΔGap		LNS	LNS-IRR	ΔR	
1	1.70	**1.22**	0.48	16.20	1101.59	**1108.85**	7.26	24.89
2	1.44	**1.12**	0.32	15.21	1107.03	**1112.94**	5.91	17.15
5	1.39	**1.03**	0.36	30.26	1109.66	**1114.94**	5.28	15.91
10	1.14	**0.88**	0.26	21.64	1111.82	**1116.68**	4.86	15.16
30	0.91	**0.69**	0.22	15.10	1114.81	**1117.99**	3.18	12.80
60	0.80	**0.62**	0.18	17.31	1115.99	**1119.09**	3.10	14.56

TABLE A.3 – Résultats obtenus avec différents temps limités CPU_{max} pour LNS sur les instances OPTW et singlesat (IRR avec $J_{max} = 2$, $W_{max} = \infty$)

De plus, vu les résultats donnés dans le tableau A.4, nous constatons que IRR est très rapide avec une petite valeur de J_{max}. D'autre part, l'augmentation de J_{max} entraîne une amélioration plus significative de la qualité de la meilleure solution recombinée, mais nécessite également beaucoup plus de temps, simplement en raison de la croissance exponentielle des états dans le système de programmation dynamique. Toutefois, la limitation de W_{max} permet alors de surmonter ce problème (voir le cas $J_{max} = 4$). En général, avec des valeurs plus petites de W_{max}, IRR est plus rapide mais il fournit également un gain plus faible, car de nombreux états sont élagués par la phase de restriction selon la paramètre W_{max}.

J_{max}	W_{max}	optw			singlesat		
		AvgGap%(LNS-IRR)	t_{IRR}(ms)	#t.o	AvgRwd(LNS-IRR)	t_{IRR}(ms)	#t.o
0	∞	1.70	0	0	1101.59	0	0
1	∞	1.63	1.48	0	1103.35	2.77	0
2	∞	1.22	16.20	0	1108.85	24.89	0
3	∞	1.05	1929.95	0	1112.26	554.14	0
4	∞	1.24	81370.61	17/76	**1113.75**	37561.21	31/648
4	10000	**0.89**	2321.45	0	1113.65	6694.53	0
4	5000	0.95	828.19	0	1113.13	3589.63	0
4	1000	1.11	129.03	0	1109.25	622.03	0
4	100	1.47	11.79	0	1103.49	43.80	0

TABLE A.4 – Impact des paramètres J_{max} et W_{max} ($CPU_{max} = 1$s)

Scénarios incertains : utilisation de RR pour la résolution en ligne
En pratique, le problème devient plus difficile dans un contexte incertain concernant la récompense associée à chaque client. Dans le contexte des satellites d'observation, la récompense associée à chaque client dépend fortement de la couverture nuageuse. Par exemple, pendant une journée nuageuse, les images remplies de nuages rapportent une récompense plus faible par rapport aux images des cibles au sol bénéficiant de conditions de ciel dégagé. Dans de tels scénarios, il se peut que la solution actuelle ne soit plus adaptée et qu'il faille retrouver une meilleure solution en fonction des dernières valuers connues pour les récompenses (voir figure A.4).

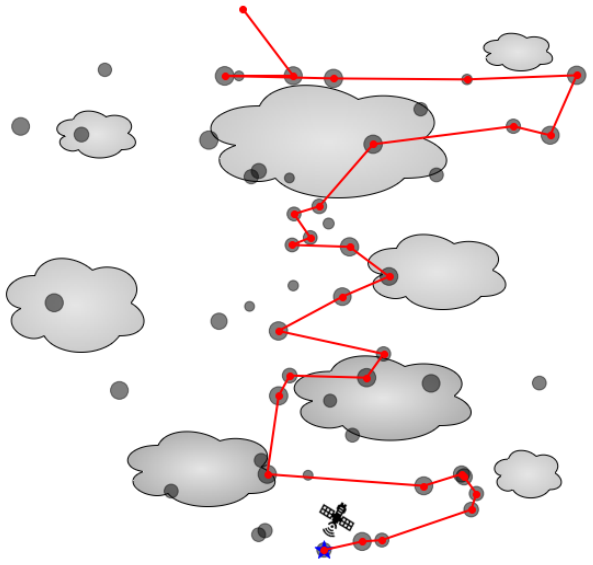


FIGURE A.4 – Un plan d'activités existant (ligne rouge) doit être révisé pendant une journée nuageuse.

La procédure RR peut être utile dans ce cas. En effet, nous pouvons d'abord générer un pool de solutions de haute qualité à partir de différents scénarios de récompense dans une phase hors ligne. Ensuite, dans la phase en ligne, nous utilisons RR directement comme un outil de résolution pour trouver la meilleure combinaison de ces solutions, basée sur les nouvelles informations de récompense, sans avoir besoin de réutiliser le solveur classique comme LNS (voir figure A.5).

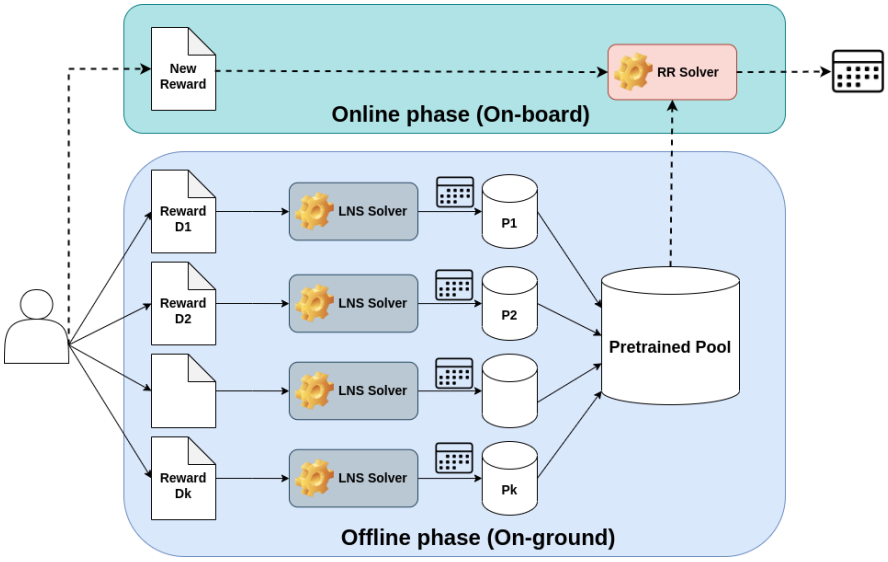


FIGURE A.5 – Résolution en ligne avec RR en cas d'incertitude des récompenses

Globalement, le tableau A.5 montre que pour les scénarios impliquant des récompenses incertaines, l'application de RR sur un ensemble de solutions pré-entraînées nécessite moins de temps pour obtenir de meilleures performances par rapport au solveur LNS classique. Dans le tableau A.6, nous avons étudié le nombre de scénarios d'entraînement et de solutions par scénario nécessaires pour maximiser l'efficacité de RR pendant la phase en ligne. Au final, avoir plus de solutions dans le pool entraîne un temps d'exécution plus long pour RR, tandis que le maintien de moins de solutions conduit à une récompense plus faible. Il est cependant contre-productif de conserver trop de solutions dans le pool de solutions élites, car cela augmente le temps d'exécution de RR sans améliorer significativement la solution, Ainsi, il est nécessaire de trouver un compromis pour équilibrer qualité de la solution combinée et temps d'exécution de RR.

En ce qui concerne l'effet de la perturbation de la récompense sur l'efficacité de RR, les résultats du tableau A.7 révèlent que RR reste efficace avec une perturbation modérée, par exemple moins de 30%. Cela s'explique par le fait que si la perturbation est trop large, les scénarios utilisés pour construire le pool d'entraînement deviennent trop différents du scénario à prendre en compte pendant la phase en ligne.

dataset	Rwd		Time(ms)	
	LNS	RR	tf_{LNS}(ms)	t_{RR}(ms)
optw-solomon1	**299.69**	**299.69**	105.99	**8.14**
optw-solomon2	905.04	**910.78**	**1915.68**	3375.48
optw-cordeau1	368.40	**372.50**	1876.66	**156.70**
optw-cordeau2	411.70	**418.60**	1791.92	**295.53**
optw-mean	495.05	**500.97**	1422.56	**958.96**
ti-singlesat-ttf1.0	1200.64	**1209.37**	1860.90	**103.99**
ti-singlesat-ttf1.5	1003.91	**1013.16**	2329.27	**400.76**
ti-singlesat-ttf2.0	841.07	**846.40**	1865.40	**408.91**
td-singlesat-ttf1.0	1213.16	**1221.11**	2058.36	**96.39**
td-singlesat-ttf1.5	1018.14	**1029.94**	2004.76	**268.49**
td-singlesat-ttf2.0	854.09	**862.17**	2082.80	**338.93**
singlesat-mean	1021.83	**1030.36**	2033.58	**269.58**

dataset	RR wins	RR=LNS	LNS wins	ΔR(%)		
				min	avg	max
optw-solomon1	0	**29**	0	0.00	0.00	0.00
optw-solomon2	**18**	6	3	-0.46	0.54	2.38
optw-cordeau1	**5**	**5**	0	0.00	0.92	2.88
optw-cordeau2	**7**	2	1	-0.20	1.48	5.32
optw-mean	-	-	-	-0.08	1.18	3.17
ti-singlesat-ttf1.0	**79**	28	1	-0.68	0.72	2.91
ti-singlesat-ttf1.5	**92**	12	4	-0.28	0.89	2.98
ti-singlesat-ttf2.0	**75**	26	7	-0.73	0.58	2.53
td-singlesat-ttf1.0	**77**	30	1	-0.39	0.63	2.41
td-singlesat-ttf1.5	**97**	7	4	-0.24	1.14	3.16
td-singlesat-ttf2.0	**85**	20	3	-0.61	0.88	5.46
singlesat-mean	-	-	-	-0.49	0.81	3.24

TABLE A.5 – Résultats obtenus par LNS et RR pour un scénario de test impliquant une perturbation de la récompense de 20% ($CPU_{max} = 5$s, $J_{max} = 2$, $W_{max} = \infty$, $ns_{train} = 10$, $pool_{train} = 5$).

ns_{train}	$pool_{train}$	OPTW		singlesat	
		Rwd(RR)	t_{RR}(ms)	Rwd(RR)	t_{RR}(ms)
5	5	499.15	**136.65**	1028.44	**65.35**
5	10	499.93	1230.83	1029.02	357.12
10	5	500.97	958.96	1030.36	269.58
10	10	**501.09**	11232.44	**1030.79**	1318.42

TABLE A.6 – Impact du nombre de solutions dans le pool d'entraînement ($CPU_{max} = 5$s, $J_{max} = 2$, $W_{max} = \infty$)

dR(%)	avgRwd-OPTW		avgRwd-singlesat	
	LNS	RR	LNS	RR
10	512.90	**518.80**	1039.29	**1048.44**
20	495.05	**500.97**	1021.83	**1030.36**
30	547.31	**552.04**	1049.41	**1055.87**
50	**546.83**	544.10	**1045.12**	1044.14

TABLE A.7 – Adaptation de RR avec différents niveaux de perturbation ($CPU_{max} = 5$s, $J_{max} = 2$, $W_{max} = \infty$, $ns_{train} = 10$, $pool_{train} = 5$)

A.3.6 Conclusion et perspectives

En résumé, l'algorithme de recombinaison des routes (RR) proposé peut être utilisé comme une procédure de post-optimisation légère et efficace travaillant sur des solutions élites fournies par un solveur (TD)OPTW incomplet standard. De plus, il peut être utilisé dans un contexte non déterministe où les valeurs des récompenses ne sont pas précisément connues à l'avance. Dans ce cas, les solutions (TD)OPTW peuvent d'abord être générées pour différents scénarios de récompense au cours d'une phase hors ligne, puis combinées au cours d'une phase en ligne pour obtenir rapidement une solution de haute qualité compte tenu des dernières valeurs de récompense connues. Divers travaux futurs sont envisagés :

- Un travail futur direct consiste à combiner RR avec les méthodes de la section A.2, c'est-à-dire avec la recherche incomplète aidée par une base de clauses, afin d'améliorer encore la qualité des solutions trouvées pour un (TD)OPTW. Cela nécessiterait également d'adapter la procédure de génération de clauses au cas des temps de transition dépendant du temps. RR pourrait aussi être hybridé avec d'autres méthodes de recherche, par exemple pour réaliser des opérations de croisement plus puissantes dans un algorithme génétique.

- Une autre perspective consiste à développer une procédure *RR Adaptative*, où plusieurs paramètres pourraient être appris, tels que (1) les paramètres appelés J_{max} et W_{max}, (2) le choix de la règle de dominance entre les états, ou (3) l'utilisation de *jumps renversés*.

- Pour mieux évaluer l'efficacité du solveur RR en cas d'incertitude sur les récompenses, il serait nécessaire d'expérimenter RR avec des données réelles. De plus, il serait possible de gérer d'autres sources d'incertitude, telles que des temps de transition incertains ou l'arrivée de clients urgents pendant la phase en ligne.

A.4 Résolution de problèmes de routage complexes par décomposition

A.4.1 Motivation

En pratique, les problèmes de tournées peuvent être complexes, impliquant éventuellement des clients optionnels, de nombreux véhicules, plusieurs fenêtres temporelles pour chaque client, des contraintes de sélection telles que des contraintes de capacité, et/ou des temps de transition *"time-dependent"* qui modélisent par exemple les conditions de circulation liées avec embouteillages. Pour résoudre rapidement ces problèmes, notamment avec des instances de taille volumineuse, de nombreux algorithmes (méta)heuristiques spécifiques pour chaque variante ont été proposés dans la littérature. Toutefois, cela pose un défi, car chaque fois qu'une nouvelle variante est introduite, les algorithmes performants définis pour les problèmes de base doivent être révisés.

Afin d'éviter de développer de nouveaux algorithmes spécifiques pour chaque variante du problème, nous proposons une approche hybride et générique qui est capable de résoudre efficacement diverses variantes complexes du problème. Plus précisément, cette approche repose sur une architecture modulaire exploitant (1) d'une part un sous-module gérant toutes les décisions de sélection des clients, et (2) d'autre part un sous-module gérant efficacement les décisions de séquencement des clients sélectionnés. En outre, une interface de résolution générique peut librement choisir une métaheuristique tout en interagissant avec les modules de bas niveau pour rechercher des solutions.

A.4.2 Un cadre générique : solveur GenOP

Pour rechercher des solutions quasi optimales pour des variantes complexes du problème d'orienteering, nous proposons l'architecture modulaire présentée à la figure A.6. Cette architecture est composée de trois éléments : une interface de résolution principale (appelée *GenOP*), un moteur de raisonnement (appelé *selMgr*) s'occupant du sous-problème de sélection de clients (ou d'affectation des clients aux véhicules), et un moteur de raisonnement (appelé *routingMgr*) dédié au sous-problème de routage (ou de séquencement).

L'idée principale ici est de pouvoir réutiliser directement les techniques

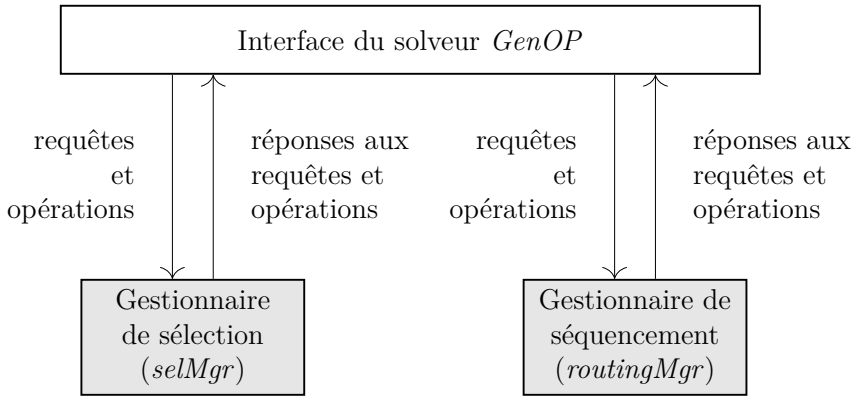


FIGURE A.6 – Une architecture générale pour résoudre des variantes complexes du problème d'orienteering

d'optimisation de bas niveau liées au sous-problème de sélection et au sous-problème de routage sans clients optionnels. En d'autres termes, *GenOP* conduit la recherche grâce aux conseils reçus du module de sélection et aux réponses reçues du module de routage concernant la faisabilité des séquences de visites. Techniquement, le solveur *GenOP* contient la fonction de résolution principale et interagit avec les modules de raisonnement bas niveau par des requêtes et opérations spécifiques.

A.4.3 Gestionnaire de sélection

Le module *selMgr* se concentre sur la gestion des contraintes des problèmes au niveau de la sélection. En géneral, *selMgr* doit prendre en compte chaque décision prise par *GenOP* et mettre à jour les contraintes en fonction de ces décisions. De plus, *selMgr* doit être en mesure de répondre rapidement aux requêtes envoyées par *GenOP* concernant la faisabilité ou l'optimalité du problème de sélection.

Formellement, le sous-problème de sélection implique des contraintes linéaires pseudo-booléennes de la forme $\sum_{i,v} e_{ivz} x_{iv} \leq E_z$ ou $\sum_{i,v} e_{ivz} x_{iv} \geq E_z$ où les termes $e_{ivz}, E_z \in \mathbb{N}$ sont des constantes, et où les termes $x_{iv} \in \{0,1\}$ sont des variables de décisions. Par exemple, les contraintes utilisant la relation "$\leq$" peuvent être utilisées pour exprimer les contraintes de capacité (Boussier et al. (2007)) ou les limitations budgétaires (Souffriau (2010)). De l'autre côté, les contraintes utilisant la relation "$\geq$" sont utiles pour fixer

une borne inférieure à la récompense collectée (Balas (1989)) ou pour exprimer la sélection de clients obligatoires (Gendreau et al. (1998a)). L'objectif principal est ici de trouver une affectation A de valeurs aux variables x_{iv} telle que toutes ces contraintes soient satisfaites. L'idée centrale est d'exploiter les techniques de propagation de contrainte pour raisonner sur des problèmes impliquant des contraintes pseudo-booléennes (Dixon & Ginsberg (2002); Chai & Kuehlmann (2003); Sheini & Sakallah (2005)). Par exemple, considérons une contrainte $c: \ 2x_1 + 2x_2 + x_3 \leq 3$, si nous assignons $[x_1 = 1]$, alors nous pouvons déduire que x_2 doit être fixé à 0 pour éviter la violation de la contrainte c. En fait, plusieurs solveurs dans la littérature ont été développés pour cela (Sheini & Sakallah (2005); Piotrów (2020)), et pourraient remplir le rôle de *selMgr*. Toutefois, pour contrôler et exploiter pleinement les informations obtenues au niveau de la sélection de manière flexible, nous avons décidé de construire notre propre *selMgr* au lieu de nous appuyer sur les mécanismes existants. Cela nous permet de définir des méthodes de calcul incrémentales adaptées à nos besoins.

A.4.4 Gestionnaire de séquencement

Le module *routingMgr* est chargé d'évaluer la faisabilité (ou l'optimalité) au niveau du séquencement, c'est-à-dire de déterminer s'il existe une séquence de visites traversant un ensemble spécifique de clients sélectionnés tout en respectant les contraintes temporelles.

Afin d'augmenter la généricité de *GenOP*, *routingMgr* doit être capable de traiter des problèmes de voyageur de commerce (TSP) impliquant des contraintes de fenêtre temporelle, avec ou sans temps de transition "*time-dependent*". Pour ce faire, nous pouvons exploiter les techniques efficaces pour traiter des TSP "*time-dependent*" avec des fenêtres temporelles (TDTSPTW). Dans ce travail, le module *routingMgr* réutilise directement un solveur de l'état de l'art pour les TDTSPTW, appelé ImaxLNS (Pralet (2023)). Outre l'utilisation de ce solveur pour trouver une séquence de visites faisable pour un ensemble spécifique de clients, nous adaptons cet algorithme pour l'utiliser dans un contexte dynamique où les clients peuvent être ajoutés ou supprimés de manière itérative, conformément aux demandes envoyées depuis l'interface du solveur *GenOP*.

A.4.5 MSLNS : une métaheuristique pour *GenOP*

Pour le solveur maître *GenOP*, il existe plusieurs métaheuristiques candidates inspirées des techniques disponibles dans la littérature pour résoudre des variantes de l'OP (voir le chapitre 3). Dans ce travail, nous proposons une métaheuristique pour le solveur *GenOP* appelée "Multi-Start Large Neighborhood Search" (MSLNS), afin de trouver une solution de haute qualité pendant un temps de calcul limité. L'algorithme MSLNS pour *GenOP* s'appuie sur une méthode LNS en tant que technique de recherche principale pour intensifier la recherche et effectue de temps en temps des redémarrages à partir de zéro pour diversifier la recherche. Au cours de la recherche, le solveur *GenOP* exploite les fonctions disponibles dans chaque sous-module pour essayer de sélectionner des clients et de construire une séquence de visites, tout en respectant des contraintes de sélection et des contraintes temporelles.

Dans la procédure LNS, l'algorithme utilise itérativement des opérateurs de destruction et de réparation pour rechercher une solution de meilleure qualité. Plus précisément, dans la phase de *destruction*, pour chaque véhicule, un sous-ensemble F de clients consécutifs est supprimé de la séquence de visites en cours. Le module *GenOP* notifie ce changement aux modules bas niveau *selMgr* et *routingMgr* afin qu'ils puissent mettre à jour leur état. L'une des composantes essentielles de l'algorithme MSLNS est la procédure de *réparation* qui spécifie comment réoptimiser la solution actuelle par le biais de sélections et d'insertions itératives de clients. Dans cette procédure, *GenOP* exploite une stratégie de sélection des clients et essaie de les insérer un par un tout en assurant la cohérence des sous-problèmes de sélection et de séquencement. Contrairement aux méthodes d'insertion traditionnelles, qui utilisent des heuristiques bien réglées pour vérifier l'insertion d'un client à toutes les positions possibles tout en respectant les contraintes, MSLNS se concentre d'abord sur les contraintes de sélection des clients pour trouver une sélection réalisable, et utilise ensuite des heuristiques gloutonnes pour tester les insertions. Ainsi, MSLNS peut être plus rapide pour chaque phase d'insertion et peut redémarrer plus fréquemment, ce qui conduit à une plus grande diversification

De plus, nous pouvons envisager la génération de contraintes de sélection apprises en cours de recherche afin d'enrichir la connaissance du module *selMgr* et d'améliorer le guidage au niveau de la sélection. Cela peut se faire en analysant les échecs d'insertion basés sur contraintes temporelles (c'est-à-dire la génération des conflits TW) et/ou en examinant la solution

localement optimale obtenue après chaque phase de réparation (c'est-à-dire la génération des conflits Lopt), en suivant les idées présentées précédemment dans la section A.2.

A.4.6 Expérimentations et analyses

Pour les expérimentations, nous avons considéré trois variantes de MSLNS pour le solveur *GenOP*. La version la plus simple, appelé MSLNS-basic, n'utilise qu'une heuristique de sélection *gloutonne aléatoire* et une heuristique d'insertion *gloutonne*. Les deux autres variantes, appelées MSLNS+TW et MSLNS+TW+Lopt, prennent également en compte la génération de conflits en cours de recherche pour ajouter des contraintes au niveau du module de sélection.

Afin d'évaluer la généricité des solveurs *GenOP* pour résoudre des variantes complexes du problème d'orienteering, nous comparons les résultats obtenus par le solveur GenOP avec ceux de solveurs spécifiques pour chaque variante. Cette comparaison est résumée à travers les tableaux A.8, A.9, A.10, et A.11, à partir d'expérimentations menées sur différents jeux de données :

(1) les instances TOPTW classiques (Team OPTW) qui impliquent plusieurs véhicules, allant de un à quatre ;

(2) les instances TOPTW qui comprenent plusieurs contraintes de type "knapsack" sous la forme $\sum_{i,v} e_{ivz} x_{iv} \leq E_z$ (dénotées par MC-TOPTW) ;

(3) les instances TOPTW qui présentent à la fois plusieurs contraintes de type "knapsack" et plusieurs fenêtres temporelles pour chaque client (dénotées par MC-TOP-MTW) ;

(4) des instances OPTW que nous avons générées et qui couvrent les aspects "*time-dependent*" et des fenêtres temporelles multiples pour chaque client (désignées par TD-OP-MTW).

Globalement, nous constatons que la métaheuristique MSLNS utilisée par le solveur *GenOP* donne de bons résultats sur différents jeux de données. Plus précisément, la métaheuristique GenOP-MSLNS est très compétitive par rapport à d'autres métaheuristiques classiques conçues spécifiquement pour chaque problème dans la littérature. En particulier, avec les instances MC-TOPTW, GenOP-MSLNS parvient à trouver de nouvelles meilleures

solutions connues pour 73 des 148 instances de la litérature dans un temps de calcul raisonnable.

	Instance set	Gunawan et al. (2015c)		Schmid & Ehmke (2017)		MSLNS (5mins)		
		SAILS (avg)	SAILS (best)	eLNS (avg)	eLNS (best)	basic	TW	TW+Lopt
Gap(%)	Solomon	0.81	0.21	0.21	**0.07**	0.22	0.24	0.25
	Cordeau	2.83	1.41	2.01	1.36	**1.16**	1.32	1.32
Time (s)	Solomon	98.76	-	9.59	29.00	47.88	49.13	52.24
	Cordeau	198.94	-	75.59	129.39	107.38	101.37	93.25

TABLE A.8 – Ecarts obtenus (%) sur les instances TOPTW

	Instance set	Souffriau et al. (2013)	MSLNS (1min)		
		GRILS (best)	basic	TW	TW+Lopt
Gap (%)	Solomon	1.68	**-0.14**	0.00	0.03
	Cordeau	2.89	-2.21	**-2.26**	-2.24
Time (s)	Solomon	8.35	14.23	13.82	14.36
	Cordeau	19.24	19.94	23.30	24.23

TABLE A.9 – Ecarts obtenus (%) sur les instances MC-TOPTW

	Instance set	Souffriau et al. (2013)	Lin & Vincent (2015)		MSLNS (1min)		
		GRILS	SA-RSBF	SA-RSCF	basic	TW	TW+Lopt
Gap (%)	Solomon	2.93	3.00	**2.26**	4.57	5.58	5.79
	Cordeau	5.69	3.79	4.12	**3.20**	4.36	5.42
Time (s)	Solomon	11.93	13.47	13.17	19.43	22.65	19.98
	Cordeau	27.28	22.05	22.67	24.00	26.38	25.49

TABLE A.10 – Ecarts obtenus (%) sur les instances MC-TOP-MTW

Instance	#nVisits	time (s)	Instance	#nVisits	time (s)
inst_100_1	**100**	16.61	inst_100_11	63	2.13
inst_100_2	63	13.52	inst_100_12	**100**	1.42
inst_100_3	**100**	4.12	inst_100_13	**100**	16.14
inst_100_4	68	1.34	inst_100_14	69	7.21
inst_100_5	99	0.79	inst_100_15	99	10.23
inst_100_6	64	2.81	inst_100_16	70	0.29
inst_100_7	63	7.53	inst_100_17	**100**	1.29
inst_100_8	**100**	30.72	inst_100_18	67	3.86
inst_100_9	62	18.77	inst_100_19	64	59.38
inst_100_10	**100**	1.28	inst_100_20	**100**	4.13

TABLE A.11 – Ecarts obtenus (%) par MSLNS-basic sur 20 instances TD-OP-MTW en 1 minute ; chaque instance implique 100 clients

Analyses complémentaires En général, nous remarquons que l'impact de l'utilisation des conflits TW et Lopt n'est pas significatif, contrairement à ce qui a été observé dans la section A.2. Plusieurs facteurs peuvent expliquer cette observation dans notre contexte. Tout d'abord, l'utilisation des

contraintes TW pour l'élagage du voisinage est moins efficace ici, étant donné qu'il y a moins de tests d'insertion à chaque étape de construction de MSLNS. Ainsi, vu que chaque conflit TW est dupliqué pour chaque véhicule, il conduit à une augmentation significative du nombre de contraintes dans *selMgr*, et il demande donc plus d'efforts pour le raisonnement au niveau de la sélection. De plus, l'efficacité des conflits Lopt est limitée dans les problèmes impliquant plusieurs véhicules, où la plupart des clients sont sélectionnés et visités. Dans ce cas, seuls les clients moins favorables, ayant une faible récompense, restent dans les contraintes Lopt approximatives, réduisant ainsi leur impact sur la diversification de la recherche.

En outre, nous envisageons l'amélioration du module *routingMgr* afin de prendre de meilleures décisions au niveau du séquencement en exploitant les fonctions avancées du solveur ImaxLNS. Plus précisément, l'idée principale consiste à optimiser le séquencement en utilisant la procédure de *réorganisation* de la séquence de visites afin de gagner du temps pour l'insertion de nouveaux clients. Les résultats expérimentaux donnés au tableau A.12 indiquant que l'optimisation au niveau du routage semble plus efficace avec un temps limité et devient moins efficace lorsque le temps est plus large. Cela illustre également l'analogie entre le compromis entre intensification et diversification dans les métaheuristiques, et entre la propagation simple et avancée dans la programmation par contraintes.

		GenOP-MSLNS-basic (1min)		*GenOP*-MSLNS-basic (5 mins)	
M	Instance set	standard	enhanced	standard	enhanced
1	Solomon	**1.41**	**1.41**	**1.41**	**1.41**
	Cordeau	**0.00**	**0.00**	**0.00**	**0.00**
2	Solomon	4.26	**4.18**	**0.81**	1.16
	Cordeau	3.70	**2.85**	**0.70**	1.34
3	Solomon	**6.59**	7.33	**2.97**	3.12
	Cordeau	3.76	**3.58**	**1.88**	2.18
4	Solomon	**6.02**	6.13	3.97	**3.94**
	Cordeau	5.34	**4.01**	**2.55**	2.57
Summary					
Gap (%)	Solomon	**4.57**	4.77	**2.29**	2.41
	Cordeau	3.20	**2.61**	**1.28**	1.52
	All	3.88	**3.69**	**1.78**	1.97

TABLE A.12 – Impact des améliorations du module de routage sur le benchmark MC-TOP-MTW (écart en %)

A.4.7 Perspectives

En ce qui concerne notre troisième contribution (chapitre 6), plusieurs améliorations peuvent être envisagées pour accroître à la fois la généricité et l'efficacité du solveur GenOP proposé.

- Comme le solveur GenOP peut choisir librement une stratégie de recherche, il est possible d'explorer d'autres métaheuristiques avancées pour tenter d'améliorer ses performances. Cela pourrait impliquer le réglage fin des paramètres d'une métaheuristique spécifique et des paramètres des raisonneurs de bas niveau, réglage qui pourraient être appris ou adapté en continu. Par exemple, il serait utile de déterminer automatiquement quand autoriser la réorganisation de la séquence des visites au niveau du routage.

- Une autre perspective consiste à recherche des heuristiques de sélection de clients efficaces pour le solveur GenOP. Une idée consisterait à apprendre l'impact à long terme des décisions de sélection de clients ou d'affectation de clients à des véhicules.

- Concernant la généricité de l'approche proposée, il serait possible d'adapter le solveur GenOP pour traiter d'autres variantes des problèmes de routage, telles que les variantes tenant compte des profits dépendant du temps (Peng et al. (2019)) ou les variantes faisant intervenir une fonction d'évaluation des profits non linéaire (Wang et al. (2008)). Enfin, l'architecture proposée pourrait également être envisagée pour résoudre d'autres problèmes impliquant plusieurs types de contraintes.

Bibliographie

Abbaspour, R. A. & Samadzadegan, F. (2011). Time-dependent personal tour planning and scheduling in metropolises. *Expert Systems with Applications*, 38(10), 12439–12452.

Aggarwal, C. C., Orlin, J. B., & Tai, R. P. (1997). Optimized crossover for the independent set problem. *Operations research*, 45(2), 226–234.

Ahuja, R. K., Ergun, Ö., Orlin, J. B., & Punnen, A. P. (2002). A survey of very large-scale neighborhood search techniques. *Discrete Applied Mathematics*, 123(1-3), 75–102.

Ahuja, R. K., Orlin, J. B., & Sharma, D. (2000a). Very large-scale neighborhood search. *International Transactions in Operational Research*, 7(4-5), 301–317.

Ahuja, R. K., Orlin, J. B., & Sharma, D. (2001). Multi-exchange neighborhood structures for the capacitated minimum spanning tree problem. *Mathematical Programming*, 91, 71–97.

Ahuja, R. K., Orlin, J. B., & Tiwari, A. (2000b). A greedy genetic algorithm for the quadratic assignment problem. *Computers & Operations Research*, 27(10), 917–934.

Aiex, R. M., Resende, M. G., Pardalos, P. M., & Toraldo, G. (2005). GRASP with path relinking for three-index assignment. *INFORMS Journal on Computing*, 17(2), 224–247.

Andersen, H. R., Hadzic, T., Hooker, J. N., & Tiedemann, P. (2007). A constraint store based on multivalued decision diagrams. *Proceedings of*

the 13th International Conference on Principles and Practice of Constraint Programming, 118–132.

Andrade, D. V. & Resende, M. G. (2007). GRASP with path-relinking for network migration scheduling. *Proceedings of the International Network Optimization Conference*, 1–7.

Angel, E. & Bampis, E. (2005). A multi-start dynasearch algorithm for the time-dependent single-machine total weighted tardiness scheduling problem. *European Journal of Operational Research*, 162(1), 281–289.

Angelelli, E., Archetti, C., Filippi, C., & Vindigni, M. (2017). The probabilistic orienteering problem. *Computers & Operations Research*, 81, 269–281.

Angelelli, E., Archetti, C., & Vindigni, M. (2014). The clustered orienteering problem. *European Journal of Operational Research*, 238(2), 404–414.

Applegate, D., Bixby, R., Cook, W., & Chvátal, V. (1998). On the solution of traveling salesman problems. *Documenta Mathematica*, 645–656.

Applegate, D. & Cook, W. (1991). A computational study of the job-shop scheduling problem. *ORSA Journal on Computing*, 3(2), 149–156.

Archetti, C., Carrabs, F., & Cerulli, R. (2018). The set orienteering problem. *European Journal of Operational Research*, 267(1), 264–272.

Archetti, C., Feillet, D., Hertz, A., & Speranza, M. G. (2009). The capacitated team orienteering and profitable tour problems. *Journal of the Operational Research Society*, 60, 831–842.

Archetti, C., Hertz, A., & Speranza, M. G. (2007). Metaheuristics for the team orienteering problem. *Journal of Heuristics*, 13, 49–76.

Audemard, G., Lagniez, J.-M., Mazure, B., & Saïs, L. (2010). Boosting local search thanks to CDCL. *Proceedings of the 17th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning*, 474–488.

Audemard, G., Lagniez, J.-M., & Simon, L. (2013). Improving Glucose for incremental SAT solving with assumptions : Application to MUS extraction. *Proceedings of the International Conference on Theory and Applications of Satisfiability Testing*, 309–317.

Bäck, T., Fogel, D. B., & Michalewicz, Z. (1997). *Handbook of evolutionary computation* (1st ed.). Taylor & Francis.

Backer, B. D., Furnon, V., Shaw, P., Kilby, P., & Prosser, P. (2000). Solving vehicle routing problems using constraint programming and metaheuristics. *Journal of Heuristics*, 6, 501–523.

Balas, E. (1989). The prize collecting traveling salesman problem. *Networks*, 19(6), 621–636.

Barrett, C. W., Sebastiani, R., Seshia, S. A., & Tinelli, C. (2009). Satisfiability modulo theories. *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, 825–885. IOS Press.

Beasley, J. E. (1990). A lagrangian heuristic for set-covering problems. *Naval Research Logistics (NRL)*, 37(1), 151–164.

Beaumet, G., Verfaillie, G., & Charmeau, M.-C. (2011). Feasibility of autonomous decision making on board an agile Earth-observing satellite. *Computational Intelligence*, 27(1), 123–139.

Bellman, R. (1962). Dynamic programming treatment of the travelling salesman problem. *Journal of the ACM (JACM)*, 9(1), 61–63.

Bellman, R. (1966). Dynamic programming. *Science*, 153(3731), 34–37.

Bellmore, M. & Nemhauser, G. L. (1968). The traveling salesman problem : a survey. *Operations Research*, 16(3), 538–558.

Benoist, T. & Bourreau, E. (2003). Improving global constraints support by local search. *Proceedings of the CP 2003 Workshop on Cooperative Solvers in Constraint Programming*.

Bensana, E., Lemaitre, M., & Verfaillie, G. (1999a). Earth observation satellite management. *Constraints*, 4, 293–299.

Bensana, E., Verfaillie, G., Michelon-Edery, C., & Bataille, N. (1999b). Dealing with uncertainty when managing an earth observation satellite. *Artificial Intelligence, Robotics and Automation in Space*, volume 440, 205.

Bergman, D., Cire, A. A., Van Hoeve, W.-J., & Hooker, J. (2016a). *Decision diagrams for optimization*, volume 1. Springer.

Bergman, D., Cire, A. A., Van Hoeve, W.-J., & Hooker, J. N. (2016b). Discrete optimization with decision diagrams. *INFORMS Journal on Computing*, 28(1), 47–66.

Beyer, H.-G. & Schwefel, H.-P. (2002). Evolution strategies–a comprehensive introduction. *Natural Computing*, 1, 3–52.

Binato, S., Faria Jr, H., & Resende, M. G. (2001). Greedy randomized adaptive path relinking. *Proceedings of the 4th Metaheuristics International Conference*, 393–397.

Blum, C. (2005). Ant colony optimization : Introduction and recent trends. *Physics of Life reviews*, 2(4), 353–373.

Blum, C. (2006). A new hybrid evolutionary algorithm for the huge k-cardinality tree problem. *Proceedings of the 8th Annual Conference on Genetic and Evolutionary Computation*, 515–522.

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

Bollig, B. & Wegener, I. (1996). Improving the variable ordering of OBDDs is NP-complete. *IEEE Transactions on Computers*, 45(9), 993–1002.

Bouly, H., Dang, D.-C., & Moukrim, A. (2010). A memetic algorithm for the team orienteering problem. *4OR*, 8, 49–70.

Boussier, S., Feillet, D., & Gendreau, M. (2007). An exact algorithm for team orienteering problems. *4OR*, 5, 211–230.

Brimberg, J., Mladenović, N., & Urošević, D. (2015). Solving the maximally diverse grouping problem by skewed general variable neighborhood search. *Information Sciences*, 295, 650–675.

Bryant, R. E. (1986). Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, 100(8), 677–691.

Butt, S. E. & Cavalier, T. M. (1994). A heuristic for the multiple tour maximum collection problem. *Computers & Operations Research*, 21(1), 101–111.

Butt, S. E. & Ryan, D. M. (1999). An optimal solution procedure for the multiple tour maximum collection problem using column generation. *Computers & Operations Research*, 26(4), 427–441.

Campos, V., Martí, R., Sánchez-Oro, J., & Duarte, A. (2014). GRASP with path relinking for the orienteering problem. *Journal of the Operational Research Society*, 65, 1800–1813.

Caseau, Y. & Laburthe, F. (1996). Improving branch and bound for jobshop scheduling with constraint propagation. *Proceedings of the 8th Franco-Japanese and 4th Franco-Chinese Conference on Combinatorics and Computer Science*, 129–149.

Chai, D. & Kuehlmann, A. (2003). A fast pseudo-boolean constraint solver. *Proceedings of the 40th annual Design Automation Conference*, 830–835.

Chao, I.-M., Golden, B. L., & Wasil, E. A. (1996a). A fast and effective heuristic for the orienteering problem. *European Journal of Operational Research*, 88(3), 475–489.

Chao, I.-M., Golden, B. L., & Wasil, E. A. (1996b). The team orienteering problem. *European Journal of Operational Research*, 88(3), 464–474.

Chaudhry, I. A. & Khan, A. A. (2016). A research survey : review of flexible job shop scheduling techniques. *International Transactions in Operational Research*, 23(3), 551–591.

Chen, X. & Van Beek, P. (2001). Conflict-directed backjumping revisited. *Journal of Artificial Intelligence Research*, 14, 53–81.

Chien, S., Doubleday, J., Thompson, D. R., Wagstaff, K. L., Bellardo, J., Francis, C., Baumgarten, E., Williams, A., Yee, E., Fluitt, D., Stanton, E., & Piug-Suari, J. (2014). Onboard autonomy on the intelligent payload experiment (IPEX) cubesat mission : a pathfinder for the proposed hyspiri mission intelligent payload module. *Proceedings of the International Symposium on Artificial Intelligence, Robotics and Automation in Space*.

Chou, X., Gambardella, L. M., & Montemanni, R. (2021). A tabu search algorithm for the probabilistic orienteering problem. *Computers & Operations Research*, 126, 105107.

Clements, D., Crawford, J., Joslin, D., Nemhauser, G., Puttlitz, M., & Savelsbergh, M. (1997). Heuristic optimization : A hybrid AI/OR approach. *Proceedings of the Workshop on Industrial Constraint-Directed Scheduling*, 33.

Congram, R. K. (2000). *Polynomially searchable exponential neighbourhoods for sequencing problems in combinatorial optimisation*. Doctoral Dissertation, University of Southampton.

Congram, R. K., Potts, C. N., & van de Velde, S. L. (2002). An iterated dynasearch algorithm for the single-machine total weighted tardiness scheduling problem. *INFORMS Journal on Computing*, 14(1), 52–67.

Cordeau, J.-F., Gendreau, M., & Laporte, G. (1997). A tabu search heuristic for periodic and multi-depot vehicle routing problems. *Networks : An International Journal*, 30(2), 105–119.

Cotta, C. & Troya, J. M. (2003). Embedding branch and bound within evolutionary algorithms. *Applied Intelligence*, 18, 137–153.

Crawford, J. (1996). Solving satisfiability problems using a combination of systematic and local search. *Second Challenge on Satisfiability Testing organized by Center for Discrete Mathematics and Computer Science of Rutgers University*.

Crawford, J. M. (1993). Solving satisfiability problems using a combination of systematic and local search. *Second DIMACS Challenge : cliques, coloring, and satisfiability*.

Cura, T. (2014). An artificial bee colony algorithm approach for the team orienteering problem with time windows. *Computers & Industrial Engineering*, 74, 270–290.

Dang, D.-C., El-Hajj, R., & Moukrim, A. (2013a). A branch-and-cut algorithm for solving the team orienteering problem. *Proceedings of the 10th International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, 332–339.

Dang, D.-C., Guibadj, R. N., & Moukrim, A. (2011). A PSO-based memetic algorithm for the team orienteering problem. *Applications of Evolutionary Computation : EvoApplications 2011*, 471–480.

Dang, D.-C., Guibadj, R. N., & Moukrim, A. (2013b). An effective pso-inspired algorithm for the team orienteering problem. *European Journal of Operational Research*, 229(2), 332–344.

Dantzig, G. B. (1951). Maximization of a linear function of variables subject to linear inequalities. *Activity Analysis of Production and Allocation*, 13, 339–347.

Darwiche, A. & Marquis, P. (2002). A knowledge compilation map. *Journal of Artificial Intelligence Research*, 17, 229–264.

De Backer, B., Furnon, V., Prosser, P., Kilby, P., & Shaw, P. (1997). Local search in constraint programming : Application to the vehicle routing problem. *Proceedings of CP-97 Workshop Industrial Constraint-Directed Scheduling*, 1–15.

De Kleer, J. (1986). An assumption-based truth maintenance system. *Artificial intelligence*, 28(2), 127–162.

Dell'Amico, M. & Lodi, A. (2005). On the integration of metaheuristic strategies in constraint programming. *Metaheuristic Optimization via Memory and Evolution : Tabu Search and Scatter Search*, 357–371.

Dell'Amico, M., Maffioli, F., & Värbrand, P. (1995). On prize-collecting tours and the asymmetric travelling salesman problem. *International Transactions in Operational Research*, 2(3), 297–308.

Dixon, H. E. & Ginsberg, M. L. (2002). Inference methods for a pseudo-boolean satisfiability solver. *Proceedings of the Eighteenth National Conference on Artificial Intelligence*, 635–640.

Dorigo, M., Birattari, M., & Stutzle, T. (2006). Ant colony optimization. *IEEE Computational Intelligence Magazine*, 1(4), 28–39.

Dorigo, M. & Gambardella, L. M. (1997). Ant colony system : a cooperative learning approach to the traveling salesman problem. *IEEE Transactions on Evolutionary Computation*, 1(1), 53–66.

Duarte, A. R., Ribeiro, C. C., & Urrutia, S. (2007). A hybrid ILS heuristic to the referee assignment problem with an embedded MIP strategy. *Hybrid Metaheuristics : 4th International Workshop*, 82–95.

Duque, D., Lozano, L., & Medaglia, A. L. (2015). Solving the orienteering problem with time windows via the pulse framework. *Computers & Operations Research*, 54, 168–176.

Eén, N. & Sörensson, N. (2003). Temporal induction by incremental SAT solving. *Electronic Notes in Theoretical Computer Science*, 89(4), 543–560.

El-Hajj, R., Moukrim, A., Chebaro, B., & Kobeissi, M. (2015). A column generation algorithm for the team orienteering problem with time windows. *Proceedings of the 45th International Conference on Computers & Industrial Engineering*.

Erdoğan, G. & Laporte, G. (2013). The orienteering problem with variable profits. *Networks*, 61(2), 104–116.

Eremeev, A. V. (2008). On complexity of optimal recombination for binary representations of solutions. *Evolutionary Computation*, 16(1), 127–147.

Evers, L., Glorie, K., Van Der Ster, S., Barros, A. I., & Monsuur, H. (2014). A two-stage approach to the orienteering problem with stochastic weights. *Computers & Operations Research*, 43, 248–260.

Fang, H. & Ruml, W. (2004). Complete local search for propositional satisfiability. *Proceedings of the Nineteenth National Conference on Artificial Intelligence*, volume 4, 161–166.

Feillet, D., Dejax, P., & Gendreau, M. (2005). Traveling salesman problems with profits. *Transportation Science*, 39(2), 188–205.

Feltl, H. & Raidl, G. R. (2004). An improved hybrid genetic algorithm for the generalized assignment problem. *Proceedings of the 2004 ACM Symposium on Applied Computing*, 990–995.

Feo, T. A. & Resende, M. G. (1995). Greedy randomized adaptive search procedures. *Journal of Global Optimization*, 6, 109–133.

Ferreira, J., Quintas, A., Oliveira, J. A., Pereira, G. A., & Dias, L. (2014). Solving the team orienteering problem : developing a solution tool using a genetic algorithm approach. *Proceedings of the 17th Online World Conference on Soft Computing in Industrial Applications*, 365–375.

Focacci, F., Laburthe, F., & Lodi, A. (2003). Local search and constraint programming. *Handbook of Metaheuristics*, 369–403. Springer.

Fogel, D. B. (1998). *Artificial intelligence through simulated evolution*, 227–296. Wiley-IEEE Press.

Fogel, D. B. (1999). An overview of evolutionary programming. *Evolutionary Algorithms*, 89–109.

Fogel, L. J. (1962). Toward inductive inference automata. *Communications of the ACM*, volume 5, 319–319.

Fomin, F. V. & Lingas, A. (2002). Approximation algorithms for time-dependent orienteering. *Information Processing Letters*, 83(2), 57–62.

Fonlupt, C., Robilliard, D., Preux, P., & Talbi, E.-G. (1999). Fitness landscapes and performance of meta-heuristics. *Meta-heuristics : Advances and Trends in Local Search Paradigms for Optimization*, 257–268.

Gambardella, L. M., Montemanni, R., & Weyland, D. (2012). Coupling ant colony systems with strong local searches. *European Journal of Operational Research*, 220(3), 831–843.

Garcia, A., Arbelaitz, O., Vansteenwegen, P., Souffriau, W., & Linaza, M. T. (2010). Hybrid approach for the public transportation time dependent orienteering problem with time windows. *Proceedings of the 5th International Conference on Hybrid Artificial Intelligence Systems*, 151–158.

Garcia, A., Vansteenwegen, P., Arbelaitz, O., Souffriau, W., & Linaza, M. T. (2013). Integrating public transportation in personalised electronic tourist guides. *Computers & Operations Research*, 40(3), 758–774.

Garey, M. R. & Johnson, D. S. (1979). *Computers and intractability : A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company.

Gavalas, D., Konstantopoulos, C., Mastakas, K., Pantziou, G., & Tasoulas, Y. (2013). Cluster-based heuristics for the team orienteering problem with time windows. *Proceedings of the 12th International Symposium on Experimental Algorithms*, 390–401.

Gavalas, D., Konstantopoulos, C., Mastakas, K., Pantziou, G., & Vathis, N. (2014). Efficient heuristics for the time dependent team orienteering problem with time windows. *Proceedings of the First International Conference on Applied Algorithms*, 152–163.

Gedik, R., Kirac, E., Milburn, A. B., & Rainwater, C. (2017). A constraint programming approach for the team orienteering problem with time windows. *Computers & Industrial Engineering*, 107, 178–195.

Geem, Z. W., Tseng, C.-L., & Park, Y. (2005). Harmony search for generalized orienteering problem : best touring in china. *Proceedings of the International Conference on Natural Computation*, 741–750.

Gendreau, M., Laporte, G., & Semet, F. (1998a). A branch-and-cut algorithm for the undirected selective traveling salesman problem. *Networks : An International Journal*, 32(4), 263–273.

Gendreau, M., Laporte, G., & Semet, F. (1998b). A tabu search heuristic for the undirected selective travelling salesman problem. *European Journal of Operational Research*, 106(2-3), 539–545.

Gillard, X. & Schaus, P. (2022). Large neighborhood search with decision diagrams. *Proceedings of the 31st International Joint Conference on Artificial Intelligence*, 4754–4760.

Ginsberg, M. L. & McAllester, D. (1994). GSAT and dynamic backtracking. *Proceedings of the International Workshop on Principles and Practice of Constraint Programming*, 243–265.

Glover, F. (1989). Tabu search—part I. *ORSA Journal on Computing*, 1(3), 190–206.

Glover, F. (1990). Tabu search—part II. *ORSA Journal on Computing*, 2(1), 4–32.

Glover, F. (1997a). Tabu search and adaptive memory programming—advances, applications and challenges. *Interfaces in Computer Science and Operations Research : Advances in Metaheuristics, Optimization, and Stochastic Modeling Technologies*, 1–75.

Glover, F. (1997b). A template for scatter search and path relinking. *Proceedings of the European Conference on Artificial Evolution*, 1–51.

Glover, F. & Laguna, M. (1998). *Tabu search*. Springer US.

Glover, F., Laguna, M., & Martí, R. (2000). Fundamentals of scatter search and path relinking. *Control and Cybernetics*, 29(3), 653–684.

Glover, F. W. & Kochenberger, G. A. (2006). *Handbook of metaheuristics*, volume 57. Springer Science & Business Media.

Golden, B. L., Levy, L., & Vohra, R. (1987). The orienteering problem. *Naval Research Logistics (NRL)*, 34(3), 307–318.

Golomb, S. W. & Baumert, L. D. (1965). Backtrack programming. *Journal of the ACM (JACM)*, 12(4), 516–524.

Grosso, A., Della Croce, F., & Tadei, R. (2004). An enhanced dynasearch neighborhood for the single-machine total weighted tardiness scheduling problem. *Operations Research Letters*, 32(1), 68–72.

Gunawan, A., Lau, H. C., & Lu, K. (2015a). An iterated local search algorithm for solving the orienteering problem with time windows. *Proceedings of the 15th European Conference on Evolutionary Computation in Combinatorial Optimization*, 61–73.

Gunawan, A., Lau, H. C., & Lu, K. (2015b). SAILS : hybrid algorithm for the team orienteering problem with time windows. *Proceedings of the 7th Multidisciplinary International Scheduling Conference*.

Gunawan, A., Lau, H. C., & Lu, K. (2015c). Well-tuned ils for extended team orienteering problem with time windows. *Living Analytics Research Center, Singapore, Technical Report*.

Gunawan, A., Lau, H. C., & Vansteenwegen, P. (2016). Orienteering problem : A survey of recent variants, solution approaches and applications. *European Journal of Operational Research*, 255(2), 315–332.

Gunawan, A., Yuan, Z., & Lau, H. C. (2014). A mathematical model and metaheuristics for time dependent orienteering problem. *Proceedings of the 10th International Conference of the Practice and Theory of Automated Timetabling*.

Hadj-Salah, A., Verdier, R., Caron, C., Picard, M., & Capelle, M. (2019). Schedule earth observation satellites with deep reinforcement learning. *Proceedings of the International Workshops on Planning and Scheduling for Space*.

Hansen, P. & Mladenović, N. (1999). An introduction to variable neighborhood search. *Meta-heuristics : Advances and Trends in Local Search Paradigms for Optimization*, 433–458. Springer.

Hansen, P. & Mladenović, N. (2001). Variable neighborhood search : Principles and applications. *European Journal of Operational Research*, 130(3), 449–467.

Hansen, P., Mladenović, N., Brimberg, J., & Pérez, J. A. M. (2019). *Variable neighborhood search*. Springer International Publishing.

Hapsari, I., Surjandari, I., & Komarudin, K. (2018). Solving multi objectives team orienteering problem with time windows using multi integer linear programming. *Proceedings of the International Conference on Industrial Revolution for Polytechnic Education*.

Harvey, W. D. & Ginsberg, M. L. (1995). Limited discrepancy search. *Proceedings of the 14th International Joint Conference on Artificial Intelligence*, 607–615.

Held, M. & Karp, R. M. (1962). A dynamic programming approach to sequencing problems. *Journal of the Society for Industrial and Applied mathematics*, 10(1), 196–210.

Helsgaun, K. (2017). An extension of the Lin-Kernighan-Helsgaun TSP solver for constrained traveling salesman and vehicle routing problems : Techincal report. *Roskilde : Roskilde University*, 12.

Hentenryck, P. V. & Michel, L. (2005). *Constraint-based local search*. MIT Press, Cambridge, MA.

Hirsch, E. A. & Kojevnikov, A. (2005). UnitWalk : A new SAT solver that uses local search guided by unit clause elimination. *Annals of Mathematics and Artificial Intelligence*, 43, 91–111.

Holland, J. H. (1975). Adaptation in natural and artificial systems. *University of Michigan Press, Ann Arbor*, 7, 390–401.

Hooker, J. N. & Ottosson, G. (2003). Logic-based Benders decomposition. *Mathematical Programming*, 96(1), 33–60.

Howard, R. A. (1960). *Dynamic programming and markov processes*. MIT Press, Cambridge, MA.

Hu, B., Leitner, M., & Raidl, G. R. (2008). Combining variable neighborhood search with integer linear programming for the generalized minimum spanning tree problem. *Journal of Heuristics*, 14, 473–499.

Hu, Q. & Lim, A. (2014). An iterative three-component heuristic for the team orienteering problem with time windows. *European Journal of Operational Research*, 232(2), 276–286.

Humeau, J., Liefooghe, A., Talbi, E. G., & Verel, S. (2013). Paradiseo-mo : From fitness landscape analysis to efficient local search algorithms. *Journal of Heuristics*, 19, 881–915.

Ignatiev, A. & Semenov, A. (2011). DPLL+ ROBDD derivation applied to inversion of some cryptographic functions. *Proceedings of the International Conference on Theory and Applications of Satisfiability Testing*, 76–89.

Ilhan, T., Iravani, S. M., & Daskin, M. S. (2008). The orienteering problem with stochastic profits. *IIE Transactions*, 40(4), 406–421.

Jones, T. et al. (1995). *Evolutionary algorithms, fitness landscapes and search*. Doctoral Dissertation, The University of New Mexico.

Jussien, N. & Lhomme, O. (2002). Local search with constraint propagation and conflict-based heuristics. *Artificial Intelligence*, 139(1), 21–45.

Kantor, M. G. & Rosenwein, M. B. (1992). The orienteering problem with time windows. *Journal of the Operational Research Society*, 43(6), 629–635.

Karbowska-Chilinska, J. & Zabielski, P. (2014). Genetic algorithm with path relinking for the orienteering problem with time windows. *Fundamenta Informaticae*, 135(4), 419–431.

Ke, L., Archetti, C., & Feng, Z. (2008). Ants can solve the team orienteering problem. *Computers & Industrial Engineering*, 54(3), 648–665.

Ke, L., Zhai, L., Li, J., & Chan, F. T. (2016). Pareto mimic algorithm : An approach to the team orienteering problem. *Omega*, 61, 155–166.

Khodadadian, M., Divsalar, A., Verbeeck, C., Gunawan, A., & Vansteenwegen, P. (2022). Time dependent orienteering problem with time windows and service time dependent profits. *Computers & Operations Research*, 143, 105794.

Kirkpatrick, S., Gelatt Jr, C. D., & Vecchi, M. P. (1983). Optimization by simulated annealing. *Science*, 220(4598), 671–680.

Klau, G. W., Ljubić, I., Moser, A., Mutzel, P., Neuner, P., Pferschy, U., Raidl, G., & Weiskircher, R. (2004). Combining a memetic algorithm with integer programming to solve the prize-collecting steiner tree problem. *Proceedings of the Genetic and Evolutionary Computation Conference*, 1304–1315.

Koza, J. R. et al. (1994). *Genetic programming II*, volume 17. MIT Press, Cambridge, MA.

Labadie, N., Mansini, R., Melechovskỳ, J., & Calvo, R. W. (2012). The team orienteering problem with time windows : An lp-based granular variable neighborhood search. *European Journal of Operational Research*, 220(1), 15–27.

Labadie, N., Melechovskỳ, J., & Wolfler Calvo, R. (2011). Hybridized evolutionary local search algorithm for the team orienteering problem with time windows. *Journal of Heuristics*, 17, 729–753.

Laguna, M. & Marti, R. (1999). GRASP and path relinking for 2-layer straight line crossing minimization. *INFORMS Journal on Computing*, 11(1), 44–52.

Lam, J. T., Rivest, F., & Berger, J. (2019). Deep reinforcement learning for multi-satellite collection scheduling. *Proceedings of 8th International Conference on the Theory and Practice of Natural Computing*, 184–196.

Laporte, G. & Martello, S. (1990). The selective travelling salesman problem. *Discrete Applied Mathematics*, 26(2-3), 193–207.

Lawler, E. L. & Wood, D. E. (1966). Branch-and-bound methods : A survey. *Operations Research*, 14(4), 699–719.

Lemaıtre, M., Verfaillie, G., Jouhaud, F., Lachiver, J.-M., & Bataille, N. (2002). Selecting and scheduling observations of agile satellites. *Aerospace Science and Technology*, 6(5), 367–381.

Lever, J. (1996). Resource reallocation : a preprocessing role for constraint programming. *Proceedings of Practical Applications of Constraint Technology*.

Lever, J. (2005). A local search/constraint propagation hybrid for a network routing problem. *International Journal on Artificial Intelligence Tools*, 14, 43–60.

Li, X. Y., Stallmann, M. F., & Brglez, F. (2003). QingTing : a fast SAT solver using local search and efficient unit propagation. *Sixth International Conference on Theory and Applications of Satisfiability Testing*, volume 1.

Liang, Y.-C., Kulturel-Konak, S., & Lo, M.-H. (2013). A multiple-level variable neighborhood search approach to the orienteering problem. *Journal of Industrial and Production Engineering*, 30(4), 238–247.

Liao, D.-Y. & Yang, Y.-T. (2007). Imaging order scheduling of an earth observation satellite. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, 37(5), 794–802.

Lin, S. & Kernighan, B. W. (1973). An effective heuristic algorithm for the traveling-salesman problem. *Operations Research*, 21(2), 498–516.

Lin, S.-W. & Vincent, F. Y. (2012). A simulated annealing heuristic for the team orienteering problem with time windows. *European Journal of Operational Research*, 217(1), 94–107.

Lin, S.-W. & Vincent, F. Y. (2015). A simulated annealing heuristic for the multiconstraint team orienteering problem with multiple time windows. *Applied Soft Computing*, 37, 632–642.

Liu, X., Laporte, G., Chen, Y., & He, R. (2017). An adaptive large neighborhood search metaheuristic for agile satellite scheduling with time-dependent transition time. *Computers & Operations Research*, 86, 41–53.

Lourenço, H. R., Martin, O., & Stützle, T. (2001). A beginner's introduction to iterated local search. *Proceedings of the 4th Metaheuristics Conference*, volume 2, 1–6.

Lourenço, H. R., Martin, O. C., & Stützle, T. (2003). Iterated local search. *Handbook of Metaheuristics*, 320–353. Springer.

Lynce, I. & Marques-Silva, J. (2007). Random backtracking in backtrack search algorithms for satisfiability. *Discrete Applied Mathematics*, 155(12), 1604–1612.

Maniezzo, V. (1999). Exact and approximate nondeterministic tree-search procedures for the quadratic assignment problem. *INFORMS Journal on Computing*, 11(4), 358–369.

Maniezzo, V. & Carbonaro, A. (2000). An ants heuristic for the frequency assignment problem. *Future Generation Computer Systems*, 16(8), 927–935.

Maniezzo, V., Carbonaro, A., Golfarelli, M., & Rizzi, S. (2001). An ANTS algorithm for optimizing the materialization of fragmented views in data warehouses : Preliminary results. *Proceedings of the Applications of Evolutionary Computing*, 80–89.

Mansini, R., Pelizzari, M., & Wolfer, R. (2006). A granular variable neighbourhood search heuristic for the tour orienteering problem with time windows. Technical report, University of Brescia, Italy.

Marinakis, Y., Politis, M., Marinaki, M., & Matsatsinis, N. (2015). A memetic-GRASP algorithm for the solution of the orienteering problem. *Proceedings of the 3rd International Conference on Modelling, Computation and Optimization in Information Systems and Management Sciences*, 105–116.

Marques-Silva, J., Lynce, I., & Malik, S. (2009). Conflict-driven clause learning SAT solvers. *Handbook of Satisfiability*, 131–153. IOS Press.

Mautor, T. & Michelon, P. (1997). MIMAUSA : A new hybrid method combining exact solution and local search. *Proceedings of the 2nd International Conference on Meta-heuristics*.

Mazure, B., Sais, L., & Grégoire, É. (1998). Boosting complete techniques thanks to local search methods. *Annals of Mathematics and Artificial Intelligence*, 22, 319–331.

Metropolis, N., Rosenbluth, A. W., Rosenbluth, M. N., Teller, A. H., & Teller, E. (1953). Equation of state calculations by fast computing machines. *The Journal of Chemical Physics*, 21(6), 1087–1092.

Meyer, B. (2008). Hybrids of constructive metaheuristics and constraint programming : A case study with ACO. *Hybrid Metaheuristics : An Emerging Approach to Optimization*, 151–183.

Minton, S., Johnston, M. D., Philips, A. B., & Laird, P. (1992). Minimizing conflicts : a heuristic repair method for constraint satisfaction and scheduling problems. *Artificial Intelligence*, 58(1-3), 161–205.

Montemanni, R., Weyland, D., & Gambardella, L. (2011). An enhanced ant colony system for the team orienteering problem with time windows. *Proceedings of the International Symposium on Computer Science and Society*, 381–384.

Moskewicz, M. W., Madigan, C. F., Zhao, Y., Zhang, L., & Malik, S. (2001). Chaff : Engineering an efficient SAT solver. *Proceedings of the 38th annual Design Automation Conference*, 530–535.

Muthuswamy, S. & Lam, S. S. (2011). Discrete particle swarm optimization for the team orienteering problem. *Memetic Computing*, 3, 287–303.

Nadel, A. & Ryvchin, V. (2012). Efficient SAT solving under assumptions. *Proceedings of the International Conference on Theory and Applications of Satisfiability Testing*, 242–255.

Nuijten, W. & Le Pape, C. (1998). Constraint-based job shop scheduling with ILOG SCHEDULER. *Journal of Heuristics*, 3, 271–286.

Papadimitriou, C. H. & Steiglitz, K. (1998). *Combinatorial optimization : algorithms and complexity*. Courier Corporation, United States.

Peng, G., Dewil, R., Verbeeck, C., Gunawan, A., Xing, L., & Vansteenwegen, P. (2019). Agile earth observation satellite scheduling : An orienteering problem with time-dependent profits and travel times. *Computers & Operations Research*, 111, 84–98.

Peng, G., Song, G., Xing, L., Gunawan, A., & Vansteenwegen, P. (2020). An exact algorithm for agile earth observation satellite scheduling with time-dependent profits. *Computers & Operations Research*, 120, 104946.

Pesant, G. & Gendreau, M. (1996). A view of local search in constraint programming. *Proceedings of the International Conference on Principles and Practice of Constraint Programming*, 353–366.

Pesant, G. & Gendreau, M. (1999). A constraint programming framework for local search methods. *Journal of Heuristics*, 5, 255–279.

Piotrów, M. (2020). Uwrmaxsat : Efficient solver for MAXSAT and pseudo-boolean problems. *Proceedings of the 32nd International Conference on Tools with Artificial Intelligence*, 132–136.

Pitsoulis, L. S. & Resende, M. G. (2002). Greedy randomized adaptive search procedures. *Handbook of Applied Optimization*, 168–183.

Plateau, A., Tachat, D., & Tolla, P. (2002). A hybrid search combining interior point methods and metaheuristics for 0–1 programming. *International Transactions in Operational Research*, 9(6), 731–746.

Poggi, M., Viana, H., & Uchoa, E. (2010). The team orienteering problem : Formulations and branch-cut and price. *Proceedings of the 10th Workshop on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems*.

Poojari, C. A. & Beasley, J. E. (2009). Improving benders decomposition using a genetic algorithm. *European Journal of Operational Research*, 199(1), 89–97.

Potts, C. N. & van de Velde, S. L. (1995). Dynasearch-iterative local improvement by dynamic programming : Part I, the traveling salesman problem. *Laboratory of Production and Operations Management Report, University of Twente*, 95.

Povéda, G., Regnier-Coudert, O., Teichteil-Königsbuch, F., Dupont, G., Arnold, A., Guerra, J., & Picard, M. (2019). Evolutionary approaches to dynamic earth observation satellites mission planning under uncertainty. *Proceedings of the Genetic and Evolutionary Computation Conference*, 1302–1310.

Pralet, C. (2023). Iterated maximum large neighborhood search for the traveling salesman problem with time windows and its time-dependent version. *Computers & Operations Research*, 150, 106078.

Pralet, C., Lesire, C., & Jaubert, J. (2019). An autonomous mission controller for earth observing satellites. *Proceedings of the 11th International Workshop on Planning and Scheduling for Space*.

Prandtstetter, M. & Raidl, G. R. (2008). An integer linear programming approach and a hybrid variable neighborhood search for the car sequencing problem. *European Journal of Operational Research*, 191(3), 1004–1022.

Prestwich, S. (2000). A hybrid search architecture applied to hard random 3-sat and low-autocorrelation binary sequences. *Proceedings of the International Conference on Principles and Practice of Constraint Programming*, 337–352.

Prestwich, S. (2002). Combining the scalability of local search with the pruning techniques of systematic search. *Annals of Operations Research*, 115, 51–72.

Prestwich, S. (2008). The relation between complete and incomplete search. *Hybrid Metaheuristics : An Emerging Approach to Optimization*, 63–83. Springer.

Puchinger, J. & Raidl, G. R. (2004). An evolutionary algorithm for column generation in integer programming : an effective approach for 2d bin packing. *Proceedings of the International Conference on Parallel Problem Solving from Nature*, 642–651.

Puchinger, J. & Raidl, G. R. (2005). Combining metaheuristics and exact algorithms in combinatorial optimization : A survey and classification. *Proceedings of the International Work-conference on the Interplay between Natural and Artificial Computation*, 41–53.

Puchinger, J. & Raidl, G. R. (2008). Bringing order into the neighborhoods : relaxation guided variable neighborhood search. *Journal of Heuristics*, 14(5), 457–472.

Puchinger, J., Raidl, G. R., & Pferschy, U. (2006). The core concept for the multidimensional knapsack problem. *Proceedings of the 6th European Conference on Evolutionary Computation in Combinatorial Optimization*, 195–208.

Raidl, G. R. (1998). An improved genetic algorithm for the multiconstrained 0-1 knapsack problem. *Proceedings of the International Conference on Evolutionary Computation*, 207–211.

Raidl, G. R. (2006). A unified view on hybrid metaheuristics. *Proceedings of the International Workshop on Hybrid Metaheuristics*, 1–12.

Raidl, G. R., Baumhauer, T., & Hu, B. (2014). Speeding up logic-based benders' decomposition by a metaheuristic for a bi-level capacitated vehicle routing problem. *Proceedings of the International Workshop on Hybrid Metaheuristics*, 183–197.

Ramesh, R. & Brown, K. M. (1991). An efficient four-phase heuristic for the generalized orienteering problem. *Computers & Operations Research*, 18(2), 151–165.

Reeves, C. R. & Yamada, T. (1998). Genetic algorithms, path relinking, and the flowshop sequencing problem. *Evolutionary Computation*, 6(1), 45–60.

Resende, M. G., Martí, R., Gallego, M., & Duarte, A. (2010). GRASP and path relinking for the max–min diversity problem. *Computers & Operations Research*, 37(3), 498–508.

Resende, M. G. & Ribeiro, C. C. (2003). A GRASP with path-relinking for private virtual circuit routing. *Networks : An International Journal*, 41(2), 104–114.

Resende, M. G. & Ribeiro, C. C. (2010). *Greedy randomized adaptive search procedures : Advances, hybridizations, and applications*, 283–319. Handbook of Metaheuristics, Springer New York.

Resende, M. G. & Werneck, R. F. (2004). A hybrid heuristic for the p-median problem. *Journal of Heuristics*, 10, 59–88.

Resendel, M. G. & Ribeiro, C. C. (2005). GRASP with path-relinking : Recent advances and applications. *Metaheuristics : Progress as Real Problem Solvers*, 29–63.

Ribeiro, C. C., Hansen, P., Cung, V.-D., Martins, S. L., Ribeiro, C. C., & Roucairol, C. (2002a). Strategies for the parallel implementation of metaheuristics. *Essays and Surveys in Metaheuristics*, 263–308.

Ribeiro, C. C., Hansen, P., Festa, P., & Resende, M. G. (2002b). GRASP : An annotated bibliography. *Essays and Surveys in Metaheuristics*, 325–367.

Ribeiro, C. C., Hansen, P., & Grünert, T. (2002c). Lagrangean tabu search. *Essays and Surveys in Metaheuristics*, 379–397.

Ribeiro, C. C. & Rosseti, I. (2007). Efficient parallel cooperative implementations of GRASP heuristics. *Parallel Computing*, 33(1), 21–35.

Ribeiro Filho, G. & Lorena, L. N. (2000). Constructive genetic algorithm and column generation : an application to graph coloring. *Proceedings of the 5th Conference of the Association of Asian-Pacific Operations Research Societies*, 35.

Richards, E. T. & Richards, B. (2000). Nonsystematic search and no-good learning. *Journal of Automated Reasoning*, 24, 483–533.

Righini, G. & Salani, M. (2009). Decremental state space relaxation strategies and initialization heuristics for solving the orienteering problem with time windows with dynamic programming. *Computers & Operations Research*, 36(4), 1191–1203.

Righini, G., Salani, M., et al. (2006). Dynamic programming for the orienteering problem with time windows. Technical report, Università degli Studi di Milano-Polo Didattico e di Ricerca di Crema.

Rojanasoonthon, S., Bard, J. F., & Reddy, S. D. (2003). Algorithms for parallel machine scheduling : a case study of the tracking and data relay satellite system. *Journal of the Operational Research Society*, 54(8), 806–821.

Rosenkrantz, D. J., Stearns, R. E., & Lewis, II, P. M. (1977). An analysis of several heuristics for the traveling salesman problem. *SIAM Journal on Computing*, 6(3), 563–581.

Rudell, R. (1993). Dynamic variable ordering for ordered binary decision diagrams. *Proceedings of 1993 International Conference on Computer-Aided Design*, 42–47.

Salkin, H. M. & De Kluyver, C. A. (1975). The knapsack problem : a survey. *Naval Research Logistics Quarterly*, 22(1), 127–144.

Sampson, J. R. (1976). Adaptation in natural and artificial systems (John H. Holland). *SIAM Review*, 18(3), 529–530.

Savelsbergh, M. W. (1985). Local search in routing problems with time windows. *Annals of Operations Research*, 4(1), 285–305.

Schilde, M., Doerner, K. F., Hartl, R. F., & Kiechle, G. (2009). Metaheuristics for the bi-objective orienteering problem. *Swarm Intelligence*, 3, 179–201.

Schmid, V. & Ehmke, J. F. (2017). An effective large neighborhood search for the team orienteering problem with time windows. *Proceedings of the 8th International Conference on Computational Logistics*, 3–18.

Schutt, A., Feydy, T., Stuckey, P., & Wallace, M. (2013). Solving RCPSP/max by lazy clause generation. *Journal of Scheduling*, 16(3), 273–289.

Selman, B., Kautz, H., et al. (1993). Domain-independent extensions to GSAT : Solving large structured satisfiability problems. *Proceedings of the 13th international joint conference on Artificial intelligence*, volume 93, 290–295.

Şevkli, A. Z. & Sevilgen, F. E. (2010). StPSO : Strengthened particle swarm optimization. *Turkish Journal of Electrical Engineering and Computer Sciences*, 18(6), 1095–1114.

Şevkli, Z. & Sevilgen, F. E. (2010). Discrete particle swarm optimization for the orienteering problem. *Proceedings of the IEEE Congress on Evolutionary Computation*, 1–8.

Shaw, P. (1998). Using constraint programming and local search methods to solve vehicle routing problems. *Proceedings of the International Conference on Principles and Practice of Constraint Programming*, 417–431.

Shaw, P., De Backer, B., & Furnon, V. (2002). Improved local search for CP toolkits. *Annals of Operations Research*, 115, 31–50.

Sheini, H. M. & Sakallah, K. A. (2005). Pueblo : A modern pseudo-Boolean SAT solver. *Proceedings of the Design, Automation, and Test in Europe*, 684–685.

Solnon, C. (2010). *Ant colony optimization and constraint programming*. John Wiley & Sons, Inc., London.

Solomon, M. M. (1987). Algorithms for the vehicle routing and scheduling problems with time window constraints. *Operations Research*, 35(2), 254–265.

Soos, M., Nohl, K., & Castelluccia, C. (2009). Extending SAT solvers to cryptographic problems. *Proceedings of the 12th International Conference on Theory and Applications of Satisfiability Testing (SAT)*, 244–257.

Souffriau, W. (2010). *Automated tourist decision support*. Doctoral Dissertation, Katholieke Universiteit Leuven.

Souffriau, W., Vansteenwegen, P., Berghe, G. V., & Van Oudheusden, D. (2010). A path relinking approach for the team orienteering problem. *Computers & Operations Research*, 37(11), 1853–1859.

Souffriau, W., Vansteenwegen, P., Vanden Berghe, G., & Van Oudheusden, D. (2013). The multiconstraint team orienteering problem with multiple time windows. *Transportation Science*, 47(1), 53–63.

Souffriau, W., Vansteenwegen, P., Vertommen, J., Berghe, G. V., & Oudheusden, D. V. (2008). A personalized tourist trip design algorithm for mobile tourist guides. *Applied Artificial Intelligence*, 22(10), 964–985.

Srinivas, M. & Patnaik, L. M. (1994). Genetic algorithms : A survey. *IEEE Computer*, 27(6), 17–26.

Stützle, T. (1999). *Local search algorithms for combinatorial problems : analysis, improvements, and new applications*. Doctoral Dissertation, Darmstadt University of Technology.

Stützle, T. & Hoos, H. H. (2000). MAX–MIN ant system. *Future Generation Computer Systems*, 16(8), 889–914.

Su, X. & Nan, H. (2023). An enhanced heuristic for the team orienteering problem with time windows considering multiple deliverymen. *Soft Computing*, 27(6), 2853–2872.

Tae, H. & Kim, B.-I. (2015). A branch-and-price approach for the team orienteering problem with time windows. *International Journal of Industrial Engineering*, 22(2).

Talbi, E.-G. (2002). A taxonomy of hybrid metaheuristics. *Journal of Heuristics*, 8, 541–564.

Tang, H. & Miller-Hooks, E. (2005). A tabu search heuristic for the team orienteering problem. *Computers & Operations Research*, 32(6), 1379–1407.

Tani, S., Hamaguchi, K., & Yajima, S. (1996). The complexity of the optimal variable ordering problems of a shared binary decision diagram. *IEICE Transactions on Information and Systems*, 79(4), 271–281.

Thomadsen, T. & Stidsen, T. (2003). The quadratic selective travelling salesman problem. Technical report, Technical University of Denmark.

Tinós, R., Helsgaun, K., & Whitley, D. (2018). Efficient recombination in the Lin-Kernighan-Helsgaun traveling salesman heuristic. *Proceedings of the 15th International Conference on Parallel Problem Solving from Nature*, 95–107.

Tricoire, F., Romauch, M., Doerner, K. F., & Hartl, R. F. (2010). Heuristics for the multi-period orienteering problem with multiple time windows. *Computers & Operations Research*, 37(2), 351–367.

Tsiligirides, T. (1984). Heuristic methods applied to orienteering. *Journal of the Operational Research Society*, 35, 797–809.

Valicka, C. G., Garcia, D., Staid, A., Watson, J.-P., Hackebeil, G., Rathinam, S., & Ntaimo, L. (2019). Mixed-integer programming models for optimal constellation scheduling given cloud cover uncertainty. *European Journal of Operational Research*, 275(2), 431–445.

Vansteenwegen, P. & Gunawan, A. (2019). *Orienteering Problems : Models and Algorithms for Vehicle Routing Problems with Profits*. Springer Nature Switzerland.

Vansteenwegen, P., Souffriau, W., Berghe, G. V., & Oudheusden, D. V. (2009a). Metaheuristics for tourist trip planning. *Metaheuristics in the Service Industry*, 15–31.

Vansteenwegen, P., Souffriau, W., Berghe, G. V., & Van Oudheusden, D. (2009b). A guided local search metaheuristic for the team orienteering problem. *European Journal of Operational Research*, 196(1), 118–127.

Vansteenwegen, P., Souffriau, W., Berghe, G. V., & Van Oudheusden, D. (2009c). Iterated local search for the team orienteering problem with time windows. *Computers & Operations Research*, 36(12), 3281–3290.

Vansteenwegen, P., Souffriau, W., & Van Oudheusden, D. (2011). The orienteering problem : A survey. *European Journal of Operational Research*, 209(1), 1–10.

Vasquez, M., Hao, J.-K., et al. (2001). A hybrid approach for the 0-1 multidimensional knapsack problem. *Proceedings of the International Joint Conference on Artificial Intelligence*, 328–333.

Vasquez, M. & Vimont, Y. (2005). Improved results on the 0–1 multidimensional knapsack problem. *European Journal of Operational Research*, 165(1), 70–81.

Verbeeck, C., Aghezzaf, E.-H., & Vansteenwegen, P. (2013). A fast solution method for the time-dependent orienteering problem with time windows. *Proceedings of the 6th Multidisciplinary International Conference on Scheduling : Theory and Applications*, 1–4.

Verbeeck, C., Sörensen, K., Aghezzaf, E.-H., & Vansteenwegen, P. (2014). A fast solution method for the time-dependent orienteering problem. *European Journal of Operational Research*, 236(2), 419–432.

Verbeeck, C., Vansteenwegen, P., & Aghezzaf, E.-H. (2016). Solving the stochastic time-dependent orienteering problem with time windows. *European Journal of Operational Research*, 255(3), 699–718.

Verbeeck, C., Vansteenwegen, P., & Aghezzaf, E.-H. (2017). The time-dependent orienteering problem with time windows : a fast ant colony system. *Annals of Operations Research*, 254, 481–505.

Vilím, P., Laborie, P., & Shaw, P. (2015). Failure-directed search for constraint-based scheduling. *Proceedings of the 12th Internation Conference on Integration of AI and OR Techniques in Constraint Programming*, 437–453.

Voudouris, C. (1997). *Guided local search for combinatorial problems*. Doctoral Dissertation, University of Essex.

Voudouris, C. & Tsang, E. (1999). Guided local search and its application to the traveling salesman problem. *European Journal of Operational Research*, 113(2), 469–499.

Voudouris, C., Tsang, E. P., & Alsheddy, A. (2010). Guided local search. *Handbook of metaheuristics*, 321–361. Springer New York.

Wallace, M. (1996). Practical applications of constraint programming. *Constraints*, 1, 139–168.

Wang, J., Zhu, X., Yang, L. T., Zhu, J., & Ma, M. (2015). Towards dynamic real-time scheduling for multiple earth observation satellites. *Journal of Computer and System Sciences*, 81(1), 110–124.

Wang, Q., Sun, X., & Golden, B. (1996). Using artificial neural networks to solve generalized orienteering problems. *Intelligent Engineering Systems through Artificial Neural Networks*, 6, 1063–1068.

Wang, X., Golden, B. L., & Wasil, E. A. (2008). Using a genetic algorithm to solve the generalized orienteering problem. *The vehicle routing problem : latest advances and new challenges*, 263–274.

Wang, X., Song, G., Leus, R., & Han, C. (2019). Robust earth observation satellite scheduling with uncertainty of cloud coverage. *IEEE Transactions on Aerospace and Electronic Systems*, 56(3), 2450–2461.

Wolsey, L. A. & Nemhauser, G. L. (1999). *Integer and combinatorial optimization*, volume 55. John Wiley & Sons, New York.

Woodruff, D. L. (1999). A chunking based selection strategy for integrating meta-heuristics with branch and bound. *Meta-Heuristics : Advances and Trends in Local Search Paradigms for Optimization*, 499–511. Springer, Boston, MA.

Wright, S. (1932). The roles of mutation, inbreeding, crossbreeding, and selection in evolution. *Proceedings of the Sixth International Congress on Genetics*, volume 1, 356, 366.

Yokoo, M. (1994). Weak-commitment search for solving constraint satisfaction problems. *Proceedings of the Twelfth National Conference on Artifical Intelligence*, 313–318.

Yu, Q., Fang, K., Zhu, N., & Ma, S. (2019). A matheuristic approach to the orienteering problem with service time dependent profits. *European Journal of Operational Research*, 273(2), 488–503.

Zhang, M., Qin, J., Yu, Y., & Liang, L. (2018). Traveling salesman problems with profits and stochastic customers. *International Transactions in Operational Research*, 25(4), 1297–1313.

RÉPUBLIQUE FRANÇAISE
Liberté
Égalité
Fraternité

Université de Toulouse

Titre : Méthodes d'optimisation hybrides pour des problèmes de routages avec profits

Mots clés : recherche opérationnelle, optimisation combinatoire, méthodes hybrides complètes/incomplètes, apprentissage de clauses, programmation dynamique, routage avec profits

Résumé : L'optimisation combinatoire est une branche de l'optimisation mathématique qui se concentre sur la recherche de solutions optimales parmi un ensemble fini de combinaisons possibles, tout en respectant un ensemble de contraintes et en maximisant ou minimisant une fonction objectif. Pour résoudre ces problèmes, les méthodes incomplètes sont souvent utilisées en pratique, car ces dernières peuvent produire rapidement des solutions de haute qualité, ce qui est un point critique dans de nombreuses applications. Dans cette thèse, nous nous intéressons au développement d'approches hybrides qui permettent d'améliorer la recherche incomplète en exploitant les méthodes complètes. Pour traiter en cas pratique, nous considérons ici le problème de tournées de véhicules avec profits, dont l'objectif est de sélectionner un sous-ensemble de clients à visiter par des véhicules de manière à maximiser la somme des profits associés aux clients visités. Plus précisément, nous visons tout d'abord à améliorer les algorithmes de recherche incomplets en exploitant les connaissances acquises dans le passé. L'idée centrale est de: (i) apprendre des conflits (combinaisons de décisions qui conduisent à une violation de certaines contraintes ou à une sous-optimalité des solutions) et les utiliser pour éviter de réexaminer les mêmes solutions et guider la recherche, et (ii) exploiter les bonnes caractéristiques de solutions élites afin de produire de nouvelles solutions ayant une meilleure qualité. En outre, nous étudions le développement d'un solveur générique pour des problèmes de routage complexes pouvant impliquer des clients optionnels, des véhicules multiples, des fenêtres temporelles multiples, des contraintes supplémentaires, et/ou des temps de transition dépendant du temps. Le solveur générique proposé exploite des sous-problèmes pour lesquels des méthodes de raisonnement dédiées sont disponibles. L'efficacité des approches proposées est évaluée par diverses expérimentations sur des instances classiqueset sur des données réelles liées à un problème d'ordonnancement pour des satellites d'observation de la Terre, qui inclut éventuellement des profits incertains.

Title: Hybrid optimization approaches for vehicle routing problems with profits

Key words: operations reasearch, combinatorial optimization, hybrid complete/incomplete approaches, clause learning, dynamic programming, routing with profits

Abstract: Combinatorial optimization is an essential branch of computer science and mathematical optimization that deals with problems involving a discrete and finite set of decision variables. In such problems, the main objective is to find an assignment that satisfies a set of specific constraints and optimizes a given objective function. One of the main challenges is that these problems can be hard to solve in practice. In many cases, incomplete methods are preferred to complete methods since the latter may have difficulties in solving large-scale problems within a limited amount of time. On the other hand, incomplete methods can quickly produce high-quality solutions, which is a critical point in numerous applications. In this thesis, we investigate hybrid approaches that enhance incomplete search by exploiting complete search techniques. For this, we deal with a concrete case study, which is the vehicle routing problem with profits. In particular, we aim to boost incomplete search algorithms by extracting some knowledge during the search process and reasoning with the knowledge acquired in the past. The core idea is two-fold: (i) to learn conflicting solutions (that violate some constraints or that are suboptimal) and exploit them to avoid reconsidering the same solutions and guide search, and (ii) to exploit good features of elite solutions in order to hopefully generate new solutions having a higher quality. Furthermore, we investigate the development of a generic framework by decomposing and exchanging information between sub-modules to efficiently solve complex routing problems possibly involving optional customers, multiple vehicles, multiple time windows, multiple side constraints, and/or time-dependent transition times. The effectiveness of the approaches proposed is shown by various experiments on both standard benchmarks (e.g., the Orienteering Problem and its variants) and real-life datasets from the aerospace domain (e.g., the Earth Observation Satellite scheduling problem), and possibly involving uncertain profits.