A dynamic programming operator for metaheuristics to solve vehicle routing problems with optional visits

Leticia Gloria Vargas Suarez

THÈSE

En vue de l'obtention du

DOCTORAT DE L'UNIVERSITÉ FÉDÉRALE TOULOUSE MIDI-PYRÉNÉES

Délivré par :

l'Institut National des Sciences Appliquées de Toulouse (INSA de Toulouse)

Présentée et soutenue le *24/06/2016* **par :**

Leticia VARGAS

A dynamic programming operator for metaheuristics to solve vehicle routing problems with optional visits

---

JURY

Dominique FEILLET	Professeur	École des Mines St-Étienne
Laetitia JOURDAN	Professeur des Universités	Université Lille 1
Nicolas JOZEFOWIEZ	Maître de Conférences	INSA Toulouse
Sandra U. NGUEVEU	Maître de Conférences	INP Toulouse
Caroline PRODHON	Maître de Conférences	Université de Troyes
Christos TARANTILIS	Professeur	Athens University

École doctorale et spécialité :

EDSYS : Informatique 4200018

Unité de Recherche :

LAAS-CNRS Toulouse

Directeur(s) de Thèse :

Nicolas JOZEFOWIEZ et *Sandra Ulrich NGUEVEU*

Rapporteurs :

Laetitia JOURDAN et *Christos TARANTILIS*

Acknowledgments

First and foremost, I thank my thesis advisor M. Nicolas Jozefowiez, Maître de Conférences INSA-Toulouse, for his relentless support, advice and guidance along these three years. A great thank you also goes to my second thesis supervisor Mlle. Sandra Ulrich Ngueveu, Maître de Conférences INP-Toulouse, for her continuous encouragement. It was a great experience working with both of you.

Let me express my sincere gratitude to my jury: Professeur D. Feillet, Professeur de Universités L. Jourdan, Maître de Conférences C. Prodhon, and Professor C. Tarantilis. Their comments, remarks and thought-provoking questions help to improve my work and writings.

I am very grateful to our team leader M. Christian Artigues, Directeur de Recherche at LAAS-CNRS, who was always very helpful and supportive aside from being a very charismatic personality always seeking the cohesion of the team and the success of each of its members. I also thank him for always making me feel welcome in a foreign country.

I am specially grateful to Mme. Christèle Mouclier, Assistant ROC team, who helped me out so many times in diverse matters with a very kind and warm demeanour.

I would also like to thank all the permanent members of the ROC team as well as all my colleagues for their help in diverse situations, for their patience with my limited French skills, and for making the average work day more fun and interesting.

I am specially grateful to Erasmus Mundus Project LAMENITEC which financially supported my studies, fellowship 372362-1-2012-1-ES, and made this endeavor possible. I thank the French government as well for their support. This work was partially supported by the French National Research Agency through the ATHENA project under the grant ANR-13-BS02-0006.

I also gratefully acknowledge the economic support received from the Mexican National Council of Science and Technology (CONACyT) as well as from the Instituto de Innovación y Transferencia de Tecnología de Nuevo León.

Last but not least, I would like to thank my friends and family for their support. I especially wish to thank my parents for their inconditional love and support throughout my life. A very special thank you is for my children, Tavo and Tani, and my husband who have always been my greatest advocates and cheerleaders.

Abstract

Metaheuristics are problem independent optimisation techniques. As such, they do not take advantage of any specificity of the problem and, therefore, can provide general frameworks that may be applied to many problem classes. These iterative upper level methodologies can furnish a guiding strategy in designing subordinate heuristics to solve specific optimisation problems. Their use in many applications shows their efficiency and effectiveness to solve large and complex problems. Nowadays, metaheuristics applied to the solution of optimisation problems have shifted towards integrating other optimisation techniques, so that solution methods benefit from the advantages each offers. This thesis seeks to contribute to the study of vehicle routing problems with optional visits by providing a dynamic programming-based operator that works embedded into a generic metaheuristic. The operator retrieves the optimal tour of customers to visit, satisfying the side constraints of the problem, while optimising the defined objective. The operator formulates the problem of selecting the best customers to visit as a *Resource Constrained Elementary Shortest Path Problem* on an auxiliary directed acyclic graph where the side restrictions of the problem considered act as the constraining resource. In vehicle routing problems with optional visits, the customers to serve are not known a priori and this fact leaves a more difficult to solve problem than a classic routing problem, which per se is already NP-hard. Routing problems with optional visits find application in multiple and diverse areas such as bimodal distribution design, humanitarian logistics, health care delivery, tourism, recruitment, hot rolling production, selected collection or delivery, and urban patrolling among others.

Keywords: Metaheuristics, dynamic programming, vehicle routing.

Résumé

Les métaheuristiques sont des techniques d'optimisation indépendantes des problèmes traités. Elles ne profitent pas d'une spécificité du problème et, par conséquent, peuvent fournir des cadres généraux qui peuvent être appliqués à de nombreuses classes de problèmes. Les métaheuristiques peuvent fournir une stratégie de guidage dans la conception des heuristiques pour résoudre des problèmes d'optimisation spécifiques. Leur utilisation dans de nombreuses applications montre leur efficacité pour résoudre des problèmes importants et complexes. De nos jours, les métaheuristiques appliquées à la solution des problèmes d'optimisation ont évolué vers l'intégration d'autres techniques d'optimisation, de sorte que les méthodes de résolution peuvent bénéficier des avantages de chacune des composantes. Le travail dans cette thèse vise à contribuer à l'étude des problèmes de tournées de véhicules avec des visites optionnelles en fournissant un opérateur à base de programmation dynamique intégré dans un processus métaheuristique générique. L'opérateur récupère le tour optimal de clients à visiter, répondant aux contraintes du problème, tout en optimisant l'objectif défini. L'opérateur pose le problème de la sélection des meilleurs clients à visiter comme un problème de plus court chemin avec contraintes de ressources sur un graphe auxiliaire dirigé acyclique représentant les choix de visite possibles. Dans les problèmes de tournées de véhicules avec des visites optionnelles, les clients à servir ne sont pas connus a priori et cela rend plus difficile à résoudre le problème qu'un problème de routage classique qui est lui-même déjà NP-difficile. Les problèmes de tournées avec des visites optionnelles trouvent des applications dans des domaines multiples et variés tels que la conception de la distribution, la logistique humanitaire, la prestation des soins de santé, le tourisme, le recrutement, la collection ou la livraison de marchandises et patrouille en milieu urbain.

Mots-clés: Métaheuristiques, programmation dynamique, tournées de véhicules.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

Contents

1.1 Affiliation

The thesis work took place at the research centre LAAS (Laboratoire d'Analyse et d'Architecture des Systèmes) of Toulouse which belongs to the CNRS (the French National Centre for Scientific Research). Both the thesis advisors and the author of this manuscript belong to the team named ROC (Recherche Opérationnelle, Optimisation Combinatoire et Contraintes which translates into Operations Research, Combinatorial Optimisation and Constraints). The research domains considered by this team are branches of Operational Research and/or Artificial Intelligence (more specifically Constraint Programming).

The team conducts research on models and methods for solving efficiently combinatorial (or discrete) optimisation problems and constraint satisfaction problems. To achieve this aim, the team develops, on the one hand, studies on the structure of fundamental problems in graph theory, scheduling, constraint satisfaction, and integer programming. On the other hand, the team aims at designing and evaluating generic solution methodologies to cope with the combinatorial explosion of the search space.

1.2 Motivation

Enterprises face a heavy economic burden when it comes to pay for the costs of transportation such as fuel, vehicles, maintenance, and wages. As a result, optimisation within transportation is a subject that is used and sought-after in the business world and not just a topic studied in academia. The industry has looked into the operational research (OR) and mathematical programming techniques because the use of computerized procedures for planning the distribution process leads to substantial savings (Toth and Vigo, 2002). These authors estimate that the savings could be from 5% to 20%, so studying and applying such procedures seems worthwhile. Furthermore, the road transportation sector is responsible for a high percentage of the CO_2 emissions,

and, hence, for high fuel consumption (Pedersen, 2005). According to US Environmental Protection Agency (2015), approximately 82.5% of total greenhouse gas emissions by human activities was CO_2. Along with passenger cars, which generated 42.7% of CO_2 emissions, the second largest source of CO_2 emissions in transportation was freight trucks (22.8%). Hence, improvements in distribution planning might help ease the strain caused on the environment by the transportation of goods and passengers. Nonetheless, reducing CO_2 emissions goes beyond being environmentally friendly. It also translates into a financial benefit.

Due to this major economic and social importance, the academic community has given a lot of attention to the problems concerning the distribution of goods between depots and final users. This class of problems is known as *Vehicle Routing Problems* (VRP), and they constitute a key component of transportation optimisation. Toth and Vigo (2002) explain that the distribution of goods concerns the service, in a given time period, of a set of customers by a set of vehicles, which are located in one or more depots, are operated by a set of crews (drivers), and perform their movements by using an appropriate road network. In particular, the solution of a VRP calls for the determination of a set of routes, each performed by a single vehicle that starts and ends at its own depot, such that all the requirements of the customers are fulfilled, all the operational constraints are satisfied, and the global transportation cost is minimised.

The vehicle routing software survey that recently appeared in OR/MS Today (Hall and Partyka, 2016) lists 22 vendors of vehicle routing software world-wide. The survey should not be considered as comprehensive, but rather as a representation of available vehicle routing packages. All of these companies offer software to solve variations of the *Vehicle Routing Problem*—finding an assignment of customers to vehicles, as well as the sequence and schedule of customers served by each vehicle. The aim is to minimise transportation costs while satisfying feasibility constraints as to when and where stops are visited, what can be loaded in each vehicle, and what routes drivers can serve. Solutions are usually generated in advance and executed as planned, though sometimes routes are dynamically updated throughout the day. The platforms where the different software applications can be used range from smart phones to Linux-based servers. Routing software is used to plan deliveries from central locations, pick-ups from shippers, routes of service fleets (e.g., appliance repair), and bus and taxi schedules. The companies that use routing software vary greatly in size, ranging from small businesses with a fleet of ten vans or fewer, to large corporations routing thousands of trucks. What these companies have in common is the need to coordinate and sequence tasks across multiple drivers and stops, ensuring predictable and expedient customer service at the lowest cost.

The market seeks solution methods that consider the following traits.

i. Simplicity. Cumbersome, difficult to understand procedures are not easy to implement, test and maintain.

ii. Rapid response. The faster the software provides a solution, the better.

iii. Precise. The more accurate the yielded results are, the higher the potential is for accurate budget forecast.

iv. Easiness in adapting to a variety of problem features. It is too costly to develop software from scratch every time a company wants an application for a new class of routing problem.

v. Robustness. It is better to have a method that produces fairly good results for all problem instances, than one that produces very good results for just a portion of the instances, and very poor results for the rest.

These characteristics may be somewhat in conflict, so some kind of trade-off might be necessary. The scientific literature usually evaluates the solution methods proposed in terms of execution speed and solution quality. Robustness is sometimes considered, but simplicity and easiness to adapt to a variety of problems rarely receive attention.

Within the family *Vehicle Routing Problems*, an important subclass is routing problems with *optional visits*, since a wide variety of applications can be modelled as problems of this subclass. In these problems, the assumptions, restrictions, needs and objectives are very similar to those of classical routing problems. However, the main difference is that the set of customers to serve is not known a priori. This situation gives rise to a problem which is more difficult to solve since there is an additional level of decision-making. Now, one has to *select* which customers to serve, *cluster* them in routes and *order* them in the best visitation sequence possible within each route. In this subclass, there are two kinds of problems. The first kind considers explicit and implicit visits of customers and the second one considers gaining a profit upon visiting a customer.

Many real-world routing problems exist for which the classic assumption of visiting every customer is not valid. For example, time or budget constraints may prevent a service technician or sales person from visiting all of his customers, or a company may be able to serve only the more urgent/profitable requests at a given time because the available number of resources is not sufficient to visit all customers and satisfy their demands. Another application emerges when delivering rural health care or disaster relief. It might not be possible to visit each village directly, but rather it may be sufficient for all villages to be near a stop on the route. Inhabitants of villages not on the route would be expected to travel to the nearest stop. A parallel situation emerges when providing a service such as post boxes or when designing tours in the entertaining industry. In general, these situations may be grouped as bimodal distribution design. An important application appears in the tourist industry. Due to budget and time constraints, visitors must select what they believe to be the most interesting attractions. A similar necessity, but in a different context emerges in the military or scientific exploration. The expedition of an unmanned aircraft or submarine involved in surveillance activities is constrained by its fuel supply, so it needs to select the best sites to visit or photograph.

In wide contrast with classical VRP, commercial applications seldom consider routing problems with optional visits. On the other hand, in academia, some of the routing

problems with optional visits have been only scarcely studied. Therefore, in spite of the several decades of research in vehicle routing, there still exists ample room for further investigation and improvement in the solution methods employed, and OR methods and techniques could be applied to a wider array of problems encountered within the transportation industry.

1.3 Thesis Proposal

VRP are difficult to solve problems since the set of solutions grows exponentially in function of the size of the instance to solve. Medium-sized instances may be solved with exact methods, but larger instances often require approximate methods also known as heuristics. This is usually the case for real-world applications.

The focus of this PhD. thesis is vehicle routing problems with optional visits (VRPOV). The problems studied in this work have been inspired from real-world applications, albeit they are not real-world problems themselves. More specifically, the aim of the thesis is to develop a heuristic method that addresses the solution of VRPOV in a unified manner.

The solution methodology developed is based upon the seminal approach of Beasley (1983) to solve the *Capacitated Vehicle Routing Problem* (CVRP). He indicated that a given sequence of the customers to visit (permutation known as giant tour) could be optimally splitted into feasible vehicle routes by finding the least cost path in the auxiliary acyclic graph that represents the set of n customers. A problem that can be solved in polynomial time. Beasley named his approach *route first–cluster second*, indicating in this way the tasks that need to be accomplished. First, determine a visitation order of all the customers, and then cluster them in feasible routes. Nonetheless, he did not explore the two stages of the method since the routing part was left aside. He provided few computational results for his proposal, and the method neither outperformed more traditional CVRP heuristics nor was it given adequate recognition (Laporte and Semet, 2002).

However, Beasley's seminal method has led to several successful metaheuristics for diverse VRP beginning notably with the genetic algorithm of Prins (2004) for the CVRP. This algorithm provided the first computational implementation of the splitting method suggested by Beasley, the *Split* operator. This kind of methods explores only the space of giant tours, but evaluates them using a splitting operator able to optimally segment the tour into feasible routes. A reason for the success of these metaheuristics is that a smaller solution space is searched, since the search is done over the set of giant tours rather than over the much larger set of VRP solutions. In some implementations, the giant tour is replaced by an ordering of the customers or by a priority list. This explains the more general name of *order first–split second* given to this class of heuristics by Prins et al. (2014) in their recent survey of more than 70 algorithms which follow this idea.

Unified or general-purpose solvers are algorithms that can be used to address large classes of problem settings without requiring extensive adaptations. Several general vehicle routing metaheuristics have been proposed in the literature. Among the most

recent ones: Ropke and Pisinger (2006), Pisinger and Ropke (2007), and Subramanian et al. (2013). These methods have addressed a single compound problem formulation including several variants as special cases. However, Vidal et al. (2014) have contributed the most problem-independent and efficient general-purpose solver for a very broad and diverse set of Multi-Attribute Vehicle Routing Problems: the unified hybrid genetic search metaheuristic, a considerable research challenge.

This thesis proposes a method following Beasley's approach for solving VRPOV in a unified fashion. The clustering is handled by a novel dynamic programming-based operator, named *Selector*, that allows to optimally separate a given ordering of customers into visited and non-visited subsequences. Meanwhile, the routing is solved via a generic metaheuristic which produces a number of different giant tours from which *Selector* extracts a VRPOV solution. In this case, an Adaptive Large Neighborhood Search (ALNS), developed by Ropke and Pisinger (2006) and Pisinger and Ropke (2007), was used. Our proposal solved very successfully three VRPOV: the *Covering Tour Problem*, both the single and the multi-vehicle version, and the *Orienteering Problem*.

1.4 Overview of the PhD. Thesis

The manuscript is divided into five technical chapters that firstly provide some background for the method proposed and for the problems treated. Next, the proposed methodology is detailed, and later the application of the methodology to solve different VRPOV is explained. The manuscript finishes with some general concluding remarks as well as future research perspectives.

- **Chapter 2: Routing Problems and Solution Methods**. It explains the subclass of problems known as VRPOV. It provides numerous examples of these problems and identifies some of the solution approaches presented in the literature to solve them. Next, it presents the solution methods available in OR to solve VRPOV. Emphasis is placed on heuristic methods and it expands on large-scale neighborhood search metaheuristics.

- **Chapter 3: Solution Methodology Developed**. It starts by presenting the foundations of the method developed, and proceeds to discuss the backbone of the *Selector* operator. Later, it presents the performance enhancements done to the basic algorithm, and the version developed to solve multi-vehicle problems. It finishes with a presentation of the version of the ALNS implemented.

- **Chapter 4: Solving the Covering Tour Problem with *Selector***. It explains the particular facts of how the method is applied to solve a particular VRPOV that includes implicit customer visits. It provides computational results that give evidence of the efficiency of the method. A paper based on preliminary work done on this problem was published in the Proceedings of the 9^{th} Learning and Intelligent OptimizatioN Conference (LION9) (Vargas et al., 2015a). A second paper that considers all the development done for the problem has been

submitted to the Journal of Heuristics and it has undergone a first round of revision (Vargas et al., 2015b).

- **Chapter 5: Solving the Multi-Vehicle Covering Tour Problem with** m-***Selector***. It discusses the implementation done for this problem. Computational results are presented and they show the method is competitive with the state-of-the-art metaheuristic and exact algorithm. A paper based on this work has been submitted to Computers & Operations Research (Vargas et al., 2016).
- **Chapter 6: Solving the Orienteering Problem with *Selector***. It explains how the method is applied to solve a VRPOV where profit is gained upon visitation of a customer. It includes computational results that demonstrate the quality of the suggested method. A technical report based on this work is available and will be submitted to an international journal (Vargas et al., 2015c).

The methodology developed and its application to different problems have been presented in different international conferences with or without proceedings. The following studies are co-authored with Nicolas Jozefowiez and Sandra Ulrich Ngueveu.

- Covering Tour Problem for Emergency Logistics. In the book of abstracts of the XVII Conferencia Latinoamericana en Investigación de Operaciones (CLAIO 2014) Monterrey, México, October 2014.
- A Selector Operator-Based Adaptive Large Neighborhood Search for the Covering Tour Problem. In the proceedings of the Learning and Intelligent Optimisatio N 9 Conference (LION 9) Lille, France, January 2015.
- A Selector Operator-Based Adaptive Large Neighborhood Search for the Orienteering Problem. In the book of abstracts of VeRoLog 2015 Austria, Vienna, June 2015.
- Resolviendo el Problema de Senderismo Usando Búsqueda Adaptativa de Vecindad Amplia Basada en el Operador Selector. In the book of abstracts of the IV Congreso de la Sociedad Mexicana de Investigación de Operaciones Cd. Juárez, México, October 2015.
- Solving the Single and Multi-Vehicle Covering Tour Problem with a General Purpose Operator. In the book of abstracts of the 17ème conférence de la société Française de Recherche Opérationnelle et d'Aide à la Décision (ROADEF 2016) Compiègne, France, February 2016.
- A Dynamic Programming-Based Metaheuristic to Solve the Single and Multi-Vehicle Covering Tour Problems. In the book of abstracts of VeRoLog 2016 Nantes, France, June 2016.
- Solving the Multi-Vehicle Covering Tour Problem with a Dynamic Programming-Based Operator. To be presented in the V Congreso de la Sociedad Mexicana de Investigación de Operaciones Cd.Victoria, Tamaulipas, México, October 2016.

Chapter 2

Routing Problems and Solution Methods

Contents

2.1 Introduction

During the past 30 years, heuristic solution methods, in particular metaheuristics, have achieved impressive results solving very difficult optimisation problems in the area of transportation as well as in others. There is a wealth of publications on heuristics for vehicle routing, so this chapter by no means pretends to provide yet another introduction to the topic. We would not contribute anything new. The aims of the chapter include: (i) present the class of problems treated in this thesis; (ii) provide a frame of reference for the concepts applied to devise the heuristic solution method presented in this study; and (iii) provide some insight into the research that has been reported in the scientific literature for the problems studied.

The remainder of the chapter is organised as follows. Section 2.2 provides an overview of the class of problems treated in this thesis. Section 2.3 presents the possible solution methods for such problems. The chapter then focuses on explaining the classic heuristic methods available for solving routing problems in Section 2.4. Next, in Sections 2.5 and 2.6, it proceeds with the explanation of the family of metaheuristic procedures used in this study. The final remarks are presented in Section 2.7.

2.2 Routing Problems

Due to the economic importance that the transportation, either of passengers or goods, has in modern societies, great effort has been put into finding the most efficient ways to perform it. For this purpose, operational research (OR) has been a great ally. It could be said that solving transportation-related problems is one of the greatest achievements of OR. Its techniques have been successfully applied, among others, in crew rostering, timetable design, pickup and delivery of goods, vehicle routing, facility location, network design. The wide applicability and economic importance of transportation problems has led to the creation of several abstractions of real-life problems that have been intensively studied. This section explains two that serve as a basis to later present the problems addressed in this work.

The thesis studied a type of transportation problems known as vehicle routing problems, and within this class, it focused in routing problems where it is not compulsory to visit all the given locations. This family is named *Tour Location Problems* (Laporte, 1997) or *Vehicle Routing-Allocation Problems* (Beasley and Nascimento, 1996). They all are combinatorial optimisation problems. A combinatorial problem is one whose decision variables have discrete and finite domains. As a consequence, a combinatorial problem has a finite number of solutions, although typically exponential in the number of variables. A combinatorial optimisation problem (COP) searches for the "best" configuration of the set of variables. Papadimitriou and Steiglitz (1982) provide the following formal definition.

Definition 1. A *Combinatorial Optimisation Problem* $P = (\mathcal{S}, f)$ can be defined by:

- a set of variables $X = \{x_1, \ldots, x_n\}$;
- variable domains $D_1, \ldots, D_n$;
- constraints among variables;
- an objective function f to be minimised[1], where $f : D_1 \times \cdots \times D_n \to \mathbb{R}^+$.

The set of all possible feasible assignments is $\mathcal{S} = \{s = \{(x_1, v_1), \ldots, (x_n, v_n)\} \mid v_i \in D_i,$ s satisfies all the constraints$\}$.

$\mathcal{S}$ is usually called a *search (or solution) space*, as each element of the set can be seen as a candidate solution. The objective function associates a value to each solution $s \in \mathcal{S}$. We refer to $f(s)$ as the objective value of a solution $s \in \mathcal{S}$. To solve a COP one has to

[1]Without any loss of generality, since maximising objective function f is equal to minimising $-f$

find a solution $s^* \in \mathcal{S}$ with minimum objective value, that is $f(s^*) \leq f(s) \quad \forall s \in \mathcal{S}$. A solution s^* is called a globally optimal solution of $(\mathcal{S}, f)$, and the set $\mathcal{S}^* \subseteq \mathcal{S}$ is called the set of globally optimal solutions. Thus, in COP we are looking for an object from a finite—but usually extremely large—set. This object is typically an integer number, a subset, a permutation, or a graph structure. In the problems treated in this work we are looking for a permutation on a set of objects.

In general, COP are the most difficult to solve problems. The difficulty arises from the following. First, as mentioned, the solution space grows exponentially in the instance size. Even for moderate instance sizes, the solution space can reach an astronomical size. Second, no exact algorithm with a number of steps polynomial in the size of the instances is known for solving them. This means large-sized instances are not solvable in reasonable time. Some COP can be solved by efficient algorithms, however, these cases are exceptions to the rule. The *Shortest Path Problem*, the *Minimum Spanning Tree Problem*, or *The Assignment of Jobs to Workers Problem* belong to these exceptions.

2.2.1 The Travelling Salesman Problem

The *Travelling Salesman Problem* (TSP) is one of the most widely studied combinatorial problems. Starting with the seminal publication of Dantzig et al. (1954), its solution methods have now reached a very high level, and the original problem has been extended to other decision settings (e.g., DNA sequencing and chip design). The TSP is quite simple to define, yet very difficult to solve. A set of n cities is given together with a way of determining the distances between each city. The aim is to find the shortest tour visiting each city exactly once (Hamiltonian cycle). For coming explanations, consider this note about nomenclature: the visitation sequence of the salesman, which in essence is a permutation, is named giant TSP tour or simply giant tour.

Depending on the properties the distances satisfy, there are different versions. If the distance from city i to city j is the same as the distance from city j to city i for all cities i and j, then the problem is said to be symmetric. Otherwise, it is named asymmetric. A problem is said to be Euclidean if the cities are located in $\mathbb{R}^2$ and the distance between two cities is the Euclidean distance. Lawler et al. (1985) provide a good introduction to the problem. New variants have been introduced such as the Selective TSP (Laporte and Martello, 1990); the Generalized TSP (Fischetti et al., 1997); the TSP with profits (Feillet et al., 2005) among others. Gutin and Punnen (2002) provide an in-depth explanation of the different variants and solution procedures.

Nowadays, very large Euclidean instances can be solved to optimality with the Concorde code (branch-and-cut method) of Applegate et al. (2007). The largest instance solved by this group has 85,900 locations (chip-design application) and the largest Euclidean instance has 24,978 cities[2]. Another impressive figure is the 0.0474% gap between the currently best known upper and lower bounds for a 1.9-million-city

[2] www.math.uwaterloo.ca/tsp/sweden/index.html

instance obtained by the LKH heuristic of Helsgaun[3]. It is worth mentioning these advancements because they are encouraging as they give hope for significant improvements in solution methods for other routing problems.

2.2.2 The Capacitated Vehicle Routing Problem

The *Vehicle Routing Problem* (VRP) is a generic name given to a whole class of problems. In this family of problems, different vocabulary is used. Cities are now called customers, the salespersons are named vehicles, and the starting point is known as depot. The vehicles (all equal) start at a given depot, visit customers at given locations in individual routes and then return to the depot. Routes for the vehicles are designed to minimise some objective such as the total distance travelled. The objective function may also consider other costs linked to the use of a vehicle. Then, the given VRP should solve for each route (or vehicle) not only the problem of finding the shortest visitation order, but also the one of assigning, in the best way possible, the customers it will visit, see Figure 2.1.

The most basic version of the VRP is the *Capacitated Vehicle Routing Problem* (CVRP) which introduced the idea of delivering goods to customers. For such purpose, every customer has a certain associated demand for a commodity aside from a given location, and every delivery vehicle a given capacity. In the CVRP one is given a depot, a set of n customers, a set of m vehicles, a vehicle capacity Q and every customer $i \in \{1, \ldots, n\}$ has a demand q_i. The problem calls to find m vehicle routes where all customers are served exactly once within the capacity of the vehicle assigned, while minimising the total transportation cost. Variants of the problem include relaxing the constraint on the number of vehicles used to at most m vehicles or no restriction at all on this number. Other variants place an upper bound on the length of a route.

Theoretical research and practical applications in the field of vehicle routing started with the work of Dantzig and Ramser (1959) who posed the truck dispatching problem for a fleet of gasoline delivery trucks. The area has been intensively researched since then. To better address customer requirements, several other variants of the basic VRP have been introduced such as consideration of time windows, pickup and delivery, backhauls, split delivery, or multiple depots, for example. The reader may refer to the taxonomic review presented by Eksioglu et al. (2009). Applications are varied and include solid-waste collection, street cleaning, school bus routing, delivery of goods to retailers, dial-a-ride systems, health care logistics among others. For further information, Golden et al. (2008) and Toth and Vigo (2014) are excellent sources.

2.2.3 The Vehicle Routing-Allocation Problem

Vehicle Routing-Allocation Problems (VRAP) (Beasley and Nascimento, 1996) differ from classic vehicle routing problems since the assumption shared by problems of the TSP and VRP families is that *all* customers should be served. In VRAPs, some customers may be left unvisited, refer to Figure 2.1. In other words, one does not

[3] www.math.uwaterloo.ca/tsp/world/index.html

know, a priori, which customers will be on the tour. Many real-world routing problems exist for which the classic assumption of visiting every customer is not valid. For example, time or budget constraints may prevent a serviceperson from visiting all of his customers, or a company may be able to serve only the more urgent/profitable requests at a given time because the available number of vehicles is not sufficient to visit all customers and satisfy their demands. Another application emerges when delivering rural health care. It might not be possible to visit each village directly, but rather it may be sufficient for all villages to be near a stop on the route. Inhabitants of villages not on the route would be expected to travel to the nearest stop.

Compared to the TSP and VRP families, an additional decision level has been introduced, since now one has to *select* which customers to serve, *cluster* them in routes and *order* them in the best visitation sequence possible within each route. This family of problems has been the subject of a well-developed literature since the 1980s. Beasley and Nascimento (1996) presented a paper where they concentrate their discussion on the single-vehicle version of the VRAP (SVRAP) and show how the SVRAP provides a unifying framework for understanding a number of papers and problems presented in the literature such as the: covering tour problem, the covering salesman problem, the median tour problem, the selective travelling salesman problem, the prize collecting travelling salesman problem, among others. In other words, many of the problems presented in the literature under different names are special cases, or variants, of a single general problem, namely the SVRAP. In addition to routing costs, allocation costs incurred by the customers not on the tour, together with penalty costs incurred by the customers left isolated are defined in this framework. Depending on the values given to these three different costs, diverse problems considered in the literature can be modelled. The focus of their discussion is theoretical rather than computational. Years later, Vogt et al. (2007) presented a tabu search algorithm to solve the SVRAP.

Beasley and Nascimento choose to consider the problem as a routing-allocation problem, i.e. to decide a route and an appropriate allocation for each customer (on, off the route or isolated). Deciding an allocation may well also involve deciding an underlying location for some facility (e.g., the location of collection points in the design of postal collection routes). For this reason, this class of problems has also been labelled using some combination of the words routing, location and allocation. Laporte (1988), for example, refers to some of the problems considered in the study of the SVRAP as location-routing problems. Later on, the same author studied them as tour-location problems (Laporte, 1997). For the sake of simplicity, in this study, they are referred as vehicle routing problems with optional visits (VRPOV). Figure 2.1 provides a pictorial representation of the problems formerly explained.

Henceforth, some of the most representative VRPOV are reviewed. The problems are not presented under the approach of Beasley and Nascimento (1996). Instead, they are explained in a more simple form as they have been presented in the literature. To perform the explanation I divided them into two subclasses according to a shared characteristic: (1) covering problems, which consider explicit and implicit visits, and (2) profit problems, in which a profit is gained upon visitation. Compared to covering problems, routing problems with profits have been much more intensively researched.

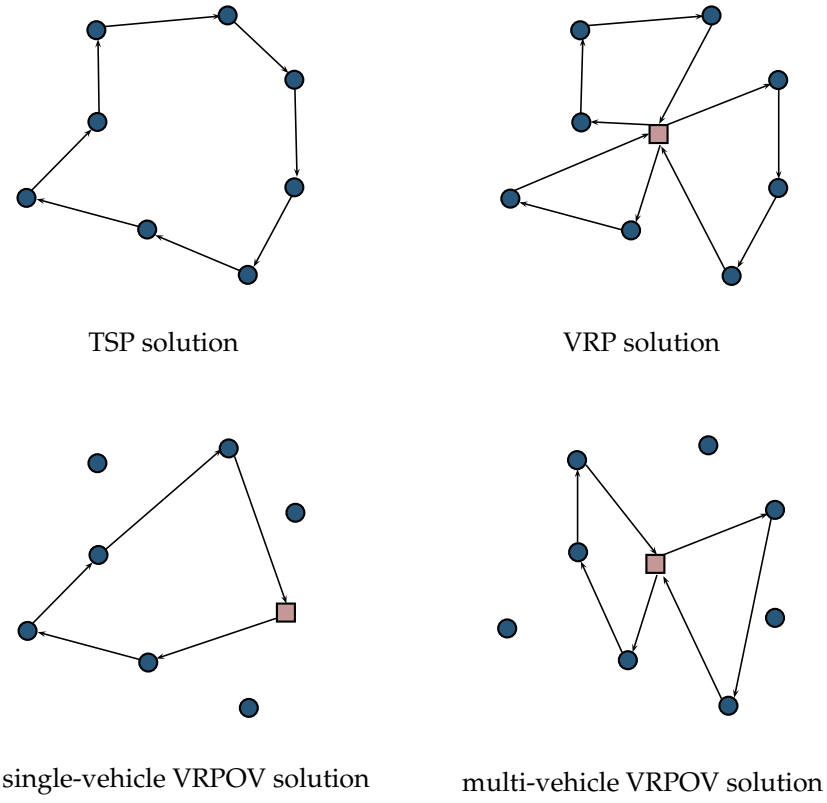

Figure 2.1: Solutions for routing problems with compulsory visits (top) and optional visits (bottom).

This work only considers problems where the customers are represented as vertices of a graph. The problems where customers are represented as edges or arcs of a graph (arc routing problems) are out of the scope of this study. In the following explanation, the first vertex is named depot, and the rest of the vertices may also be named customers, since we may think of the customers as being located on a network.

Covering Problems. The concept that Beasley and Nascimento (1996) explain in their framework as "customers not visited have to be allocated to some customer on one of the vehicle tours" is named *covered* vertex in some problems of this subclass. A vertex is considered covered if it lies within a given radius from a visited vertex. This travel distance from a visited stop is named *covering distance*. A covered vertex is, then, implicitly visited. This notion of covering is introduced in the literature to model situations in which unavailability of roads or restrictions on resources prevent from visiting a customer. In this case, the demand of an unvisited (covered) customer is delivered at a place located within an acceptable distance from it. This concept has a wide variety of applications as shown in Tableau 2.1.

Current and Schilling (1989) extended the concept of covering from the facility location literature to the TSP and introduced the *Covering Salesman Problem* (CSP).

Table 2.1: Applications of covering problems.

Application	Publication
location of post boxes	Labbé and Laporte (1986) Laporte et al. (1989)
bimodal distribution systems	Current and Schilling (1989)
health care delivery	Current and Schilling (1989) Hodgson et al. (1998)
design of tours in the entertaining industry	ReVelle and Laporte (1993)
humanitarian logistics	Naji-Azimi et al. (2012)
design of urban patrolling services	Oliveira et al. (2013)

One of the first routing problems in which visiting each of the given customers is not necessary. In the CSP, a network of n vertices and a value $p \leq n$ are given. The aim is to identify a least-cost Hamiltonian tour which visits p of the n vertices and covers all the $n - p$ vertices not on the tour. In essence, the tour must cover each vertex rather than visit it directly. They provide an integer linear formulation, and a two-step heuristic based upon solution procedures for the *Set Covering Problem* (SCP) and the TSP. In the first step, a SCP is solved in order to find the minimum number of vertices that cover all the vertices of the problem. In the second step, the optimal TSP tour is calculated over the set of vertices obtained in the first step.

Later on, Current and Schilling (1994) presented two bi-objective variants of the CSP: the *Median Tour Problem* (MTP) and the *Maximal Covering Tour Problem* (MCTP). In both problems, the tour must visit only p of the vertices, and the length of this tour is minimised (as in the CSP). In the MTP, the second objective is to minimise the total distance between each unvisited vertex and the nearest visited vertex. For the MCTP, the second objective is to maximise the total demand within some prespecified maximal covering distance. The second objective in both problems maximises access to the tour for the vertices not directly on it. To generate a good approximation of the efficient frontier, a heuristic procedure is used since the number of efficient solutions may grow exponentially with the number of points in the particular problem instance.

Golden et al. (2012) developed a generalisation of the CSP—the *Generalized CSP*—on the basis that in some applications it might not be possible to meet the demand of some customers by visiting or covering them only once, and each unvisited vertex i has to be covered k_i times. Besides the usual routing cost, they consider a cost associated with visiting a vertex. The problem seeks a minimum-cost tour such that each unvisited vertex i is covered at least k_i times by the tour. They define three variants of the problem and propose two local search heuristics based on classic TSP improvement procedures.

Salari and Naji-Azimi (2012) presented an algorithm to solve the CSP which combines heuristic search with integer linear programming (ILP) techniques. Given an initial feasible solution, the tour length is improved using a destroy-and-repair method. Vertices are removed from the tour and reassigned by solving an ILP-based model to optimality. Their ILP-based methodology outperformed the algorithms of Current and Schilling (1989) and Golden et al. (2012). Salari et al. (2015) approached the CSP with a hybrid heuristic algorithm which combines ant colony optimisation (ACO) and dynamic programming techniques that improve the quality of the solution. They also proposed a polynomial-size mathematical formulation for the studied problem. Their method outperforms all previous CSP algorithms.

Gendreau et al. (1997) proposed a generalisation of the CSP called the *Covering Tour Problem* (CTP). The CTP is defined on a weighted graph $G = (V \cup W, E)$, where V is the set of vertices that *can* be visited, $T \subseteq V$ is a set of vertices that *must* be visited but provide no covering, and W is a set of vertices that *must* be covered. The CTP calls to find a minimum-length Hamiltonian cycle on a subset of V such that every member of T is on the tour and every member of W is covered. Unlike the CSP, in this problem the total number of vertices to visit is not fixed. When $T = \emptyset$, the CTP reduces to a CSP, and when $T = V$, the CTP reduces to a TSP. They present a heuristic algorithm and a branch-and-cut method to solve it. The solution procedure developed in this thesis was applied to this problem, so a more in-depth explanation of the problem and additional solution approaches are provided in Chapter 4.

Hachicha et al. (2000) introduced a multi-vehicle version of the CTP (m-CTP) in which the goal is to find m Hamiltonian cycles over a subset of eligible vertices to visit such that all of the vertices not on the routes are covered and the total distance travelled is minimised. They present three heuristic algorithms to solve it. Their solution procedures are based upon solution heuristics for the SCP, the TSP and classic construction and improvement algorithms for the VRP. The solution methodology developed in this work was also applied to this problem, further details and solution proposals are given in Chapter 5.

Another problem closely related to the CSP is the *Generalized Travelling Salesman Problem* (GTSP). In this problem, the vertices are clustered into disjoint sets, and the aim is to identify the shortest route passing through each cluster at least once and through each vertex at most once. Fischetti et al. (1997) proposed a branch-and-cut procedure to solve it, while Karapetyan and Reihaneh (2012) presented a hybrid ACO algorithm. The latter is a modification of a simple ACO for the TSP improved by an efficient GTSP-specific local search procedure. Their computational experimentation shows that the local search procedure dramatically improves the performance of the ACO algorithm, making it one of the most successful GTSP metaheuristics to date.

Another related problem is the *Ring Star Problem* (RSP) which finds application in telecommunications network design and in rapid transit systems planning as explained in Labbé et al. (2004) and in Kedad-Sidhoum and Nguyen (2010). The two previous publications propose an exact solution method. This problem is an extension of the classical location–allocation problem introduced in the early 1960s. Each vertex is either visited by the tour (generating a routing cost) or is assigned to its closest vertex

on the tour (generating an assignment cost). The aim is to locate a simple cycle through a subset of vertices minimising the sum of routing and assignment costs. The notion of covering distance as defined previously is not present. A variant of the RSP is the *Median Cycle Problem* (MCP) which places an upper bound on the total assignment cost. The objective is to minimise the routing cost, subject to this upper bound. Labbé et al. (2005) proposed a branch-and-cut method which solves up to medium-size library instances and a medium-size real instance of the city of Milan, Italy. Moreno Pérez et al. (2003) implemented a variable neighborhood tabu search that solves both versions of the problem for library data sets. For some identified instances, their method obtained better solutions than the then state-of-the-art approach. However, Renaud et al. (2004) presented an evolutionary algorithm which outperformed the more complex heuristic of Moreno Pérez et al. (2003).

Table 2.2 offers a summary of the problems explained. The label for the first four columns is self-explanatory. The following two show if a covering distance is considered in the problem, and if the vertices are visited exactly once (H=Hamiltonian) respectively. The last one indicates the number of vehicles the problem considers.

Table 2.2: Summary of covering problems.

Problem name	Objective function	No. Vertices in tour	Type of vertex	Covering distance	H. tour	Vehicles
CSP	min distance	known, p	one	√	√	single
MTP	Z_1 min distance Z_2 min allocation distance	known, p	one		√	single
MCTP	Z_1 min distance Z_2 max demand within a covering distance	known, p	one	√	√	single
GCSP	min (distance + visiting cost)	known, p	one	√		single
CTP	min distance while covering every $w_i \in W$	unknown	three	√	√	single
m-CTP	min distance while covering every $w_i \in W$	unknown	three	√	√	multiple, m
GTSP	min distance while visiting every cluster	unknown	one	√	√	single
RSP	min (distance + allocation cost)	unknown	one		√	single
MCP	min distance subject to a max allocation cost	unknown	one		√	single

Profit Problems. As in the covering problems, their key characteristic is that the set of customers to serve is not known a priori. However, a profit value is associated with each customer. This figure places some level of attractiveness on it. Therefore, the route or set of routes can be measured both in terms of distance travelled and in terms of profit collected. Feillet et al. (2005) gather the single-vehicle version of these problems under the umbrella name *Travelling Salesman Problems with Profits* (TSPP). Their counterpart, the multi-vehicle version, is named routing problems with profits. However, Archetti et al. (2014) name both versions vehicle routing problems with profits. The following explanations follow the nomenclature of Feillet et al. (2005).

Numerous practical applications can be modelled by this kind of problems as shown in Tableau 2.3.

Table 2.3: Applications of routing problems with profits.

Application	Publication
deliver home heating fuel where the customer's tank level sets the urgency of service which translates into a score	Golden et al. (1987)
route customer visits to maximise the sales score	Ramesh and Brown (1991)
recruit high school athletes for college teams	Butt and Cavalier (1994)
delegate unprofitable customers to an external logistics provider	Chu (2005) Bolduc et al. (2008)
schedule hot rolling production	Tang and Wang (2006) Zhang et al. (2009)
design time-constrained personalised tourist routes to maximise the value of the visited attractions	Vansteenwegen and Van Oudheusden (2007) Vansteenwegen et al. (2011b)
with a limited number of auditors, select the suppliers to visit in order to maximise recovered inventory claims	Ilhan et al. (2008)
select customers for waste collection	Aksen et al. (2014)
design a multi-day tourist or sales trip where hotel choosing is also considered	Divsalar et al. (2014)

TSPP may be seen as bi-objective TSP with two opposite objectives: one pushing the vehicle to travel in order to collect profit, and the other one encouraging it to minimise travel costs with the option of skipping vertices. It is a combination of vertex selection and determination of the shortest elementary cycle among the selected vertices. Nevertheless, most researchers address them as single-criterion versions where either the two objectives are combined linearly in a single function or one of the objectives is treated as a constraint with a specified bound value. Depending on the way the two objectives are addressed, Feillet et al. (2005) define three generic TSPP:

i. Both objectives are combined in the objective function, and the goal is to find a tour that minimises travel costs minus collected profit. The problem has been defined as the *Profitable Tour Problem* (PTP) by Dell'Amico et al. (1995). Though seldom studied as such, it often appears as a subproblem in solution schemes based on column generation for a variety of routing problems.

ii. The travel cost objective is treated as a constraint, and the goal is to find a circuit that maximises the collected profit but whose length does not exceed a given bound $L_{\max}$. This problem is known in the literature as the *Orienteering Problem* (OP) as in Tsiligirides (1984) and Golden et al. (1987). Also, as the *Selective Traveling Salesman Problem* (STSP) as in Laporte and Martello (1990),

Gendreau et al. (1997), and Thomadsen and Stidsen (2003). Kataoka and Morito (1988) name it the *Maximum Collection Problem* and Awerbuch et al. (1998) the *Bank Robber Problem*. The solution approach presented in this thesis was applied to this problem, a detailed presentation is provided in Chapter 6.

iii. The profit objective is treated as a constraint, and the goal is to find a circuit that minimises travel costs but whose collected profit is above a given bound $P_{\min}$. Such problem is called the *Prize Collecting TSP* (PCTSP) as in Balas (1989) and Fischetti et al. (1997).

Multi-vehicle versions of the three problems mentioned above can be defined. However, the only routing problem with profits that has been widely studied is the multi-vehicle version of the OP, the *Team Orienteering Problem* (TOP). Butt and Cavalier (1994) initiated its study out of the necessity to model the recruitment of high school athletes in college teams, and suggested a heuristic procedure to solve it. Their paper presented it as the *Multiple Tour Maximum Collection Problem* following the nomenclature of Kataoka and Morito (1988) for the single-vehicle version. The name TOP starts with the paper of Chao et al. (1996a) as a way to highlight that the problem is a multi-vehicle version of the OP. In fact, a paper for each version was published simultaneously, Chao et al. (1996b) and Chao et al. (1996a). For both versions, they presented a quite simple heuristic solution method based on the geometry of the problem. The other two multi-vehicle versions considered in the literature are the *Capacitated Profitable Tour Problem* (CPTP), and the so-called *VRP with Private Fleet and Common Carrier* (VRPPFCC). The latter occurs when the total demand is greater than the whole capacity of owned trucks and some customers can be serviced by an external logistics provider. Two kinds of tabu search algorithms and a VNS-based algorithm were proposed by Archetti et al. (2009) to solve the CPTP and the TOP, while Vidal et al. (2015) presented three heuristic methods to solve multi-vehicle routing problems. Li and Tian (2016) present a variant of the CPTP which they name the *Prize Collecting Vehicle Routing Problem* (PCVRP). This problem is derived from the practical hot rolling production process. It adds the constraint that the total demands of served customers should not be less than a prespecified value. They propose a two-level self-adaptive variable neighborhood search to solve it.

Variants for the explained problems with profits have been introduced, a good study is that of Archetti et al. (2014) where both single and multi-vehicle versions are thoroughly explained as well as their variants. Feillet et al. (2005) presented an in-depth explanation of the works related to TSPP, and Vansteenwegen et al. (2011a) offered an extensive survey that focuses on the studies that have been done for the OP, the TOP and their variants. Table 2.4 depicts a summary of the problems with profits explained. The acronyms in parenthesis indicate alternative names given to the problem.

Table 2.4: Summary of problems with profits.

Problem name	Objective function	Constraints	Vehicle(s)
PTP	min (cost - profit)	none	single
OP (STSP, MCP, BRP)	max profit	route length	single
PCTSP	min cost	route profit	single
CPTP	min (cost - profit)	capacity	multi
TOP (MTMCP)	max profit	route length	multi
VRPPFCC	min cost	limited fleet capacity	multi
PCVRP	min (cost - profit)	capacity total demand	multi

2.3 Overview of Solution Methods

Since the set of solutions of a COP is a finite set, a naive proposition to solve it would be to do an exhaustive enumeration of this set and select the best solution. However, this set is frequently so large that complete enumeration of all possible solutions is not doable even in the fastest computers available. An example with the simplest problem explained—the TSP—gives a dimension to the former statement. The size of the search space in this problem is $n!$ possible solutions. Given different values of n, Table 2.5 shows the combinatorial explosion of the number of solutions as the number of cities grows. Consequently, we need to turn to smarter methods.

The COP explained in Section 2.2 are NP-hard. This is to say that, no polynomial time algorithm has been designed, or is expected to be designed, to solve them to optimality. These problems are also referred as being computationally intractable. Depending on their size, two approaches can be considered to search their solution space and solve them: exact and approximate, refer to Fig 2.2. Nevertheless, a solution methodology may combine both strategies at different stages. This thesis presents a methodology of this type.

2.3.1 Exact Methods

These procedures, also known as complete methods, explore the search space exhaustively, but not by merely enumerating all the solutions. A common technique to implement this idea is to keep an incumbent solution, and try to prove that specific regions of the search space cannot contain better solutions, and as a consequence, can be avoided (pruned). However, if the search strategy is unfortunate for the problem

Table 2.5: Effect of the size of the instance on the number of feasible solutions.

Number of cities, n	Size of the search space
5	120
10	3,628,800
75	2.5×10^{109}
100	9.3×10^{157}

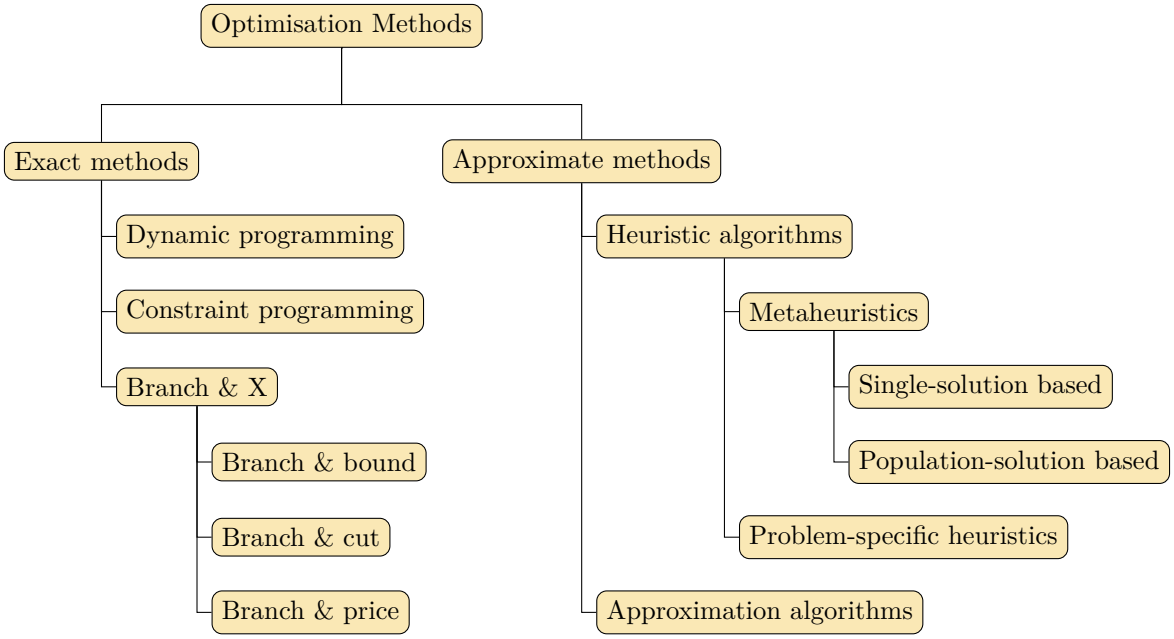

Figure 2.2: Classical optimisation methods for COP.

being solved, an exhaustive or almost exhaustive search can occur. A very important advantage of exact methods is that they guarantee that the solution found is optimal if the procedure is given sufficient time and space.

The worst-case running time of an exact method for large-sized instances (or even medium-sized) of many NP-hard problems is still very poor (exponential). They may take numerous hours to find moderately decent, let alone optimal, solutions. However, the size of the instance is not the unique indicator to describe the difficulty of the problem, but also its structure. It is possible that for a given problem some small instances are very hard to solve to optimality while larger instances may be solved exactly more easily. Since most vehicle routing problems are computationally intractable, the current limit of exact algorithms is a network of around one hundred customers. Approximate approaches are applied when solving the much larger instances met in many

industries. The following paragraphs explain the classical exact algorithms: dynamic programming, the branch & X family and constraint programming.

Dynamic Programming. The idea underlying this technique is to represent a problem by a decision process (minimisation or maximisation) which proceeds in a series of stages. Dynamic programming (DP) decomposes a problem into a number of smaller subproblems (also known as states) each of which is further decomposed until subproblems have trivial solutions. In this decomposition process, partial results are stored, and later, whenever they are needed, they are looked up instead of recomputed. Thus, it gains efficiency by avoiding solving common subproblems many times. The term "dynamic programming" stems from control theory. Programming means that an array or tableau is used to store partial results which lead to the construction of a solution. Therefore, it is a bottom-up approach. The procedure avoids total enumeration of the search space by pruning partial results that cannot lead to the optimal solution. Although it may seem that any optimisation problem can be solved using DP, this is not the case. The optimisation problem must exhibit two properties:

(1) Bellman's principle of optimality—an optimal solution to an instance of a problem always contains optimal solutions to all subinstances—must apply in the problem (Bellman, 1957).

(2) The space of subproblems should be relatively small. The decomposition done by this technique seems to lead to an exponential-time algorithm, which is indeed true in the TSP (Neapolitan and Naimipour, 1998). Typically, however, the total number of distinct subproblems is polynomial in the input size.

For instance, the *Shortest Path Problem* complies with Bellman's principle, and DP has become the standard method to solve it. In VRP, DP is usually utilized together with other solution methods, for example metaheuristics, in order to solve some specific portion of the problem.

Branch & X Family. These procedures are based on an implicit enumeration of all the solutions of the problem considered. A branch & X algorithm uses a divide and conquer strategy to partition the solution space into subproblems and then optimises each subproblem individually. The search space is explored by dynamically building a tree whose root node represents the problem being solved and its whole associated search space. The leaf nodes are the potential solutions and the internal nodes are subproblems of the total solution space. In this tree, the solution set of the child nodes is equal to the solution set of the parent nodes. In branch-and-bound (B&B), the pruning of the search tree is based on a bounding function that prunes subtrees that do not contain any optimal solution. However, the efficiency of the method relies upon its ability to prune nodes early in the tree, before they produce too many child nodes. A branch-and-cut method is similar, but valid inequalities are added to the mathematical description of the problem to further prune non-promising areas of the

search space. When B&B considers column generation methods, a branch-and-price scheme appears.

These procedures, commonly referred as integer linear programming, have been used extensively in the solution of vehicle routing problems, but with the limitation that they can solve efficiently only small or medium-sized instances. Toth and Vigo (2014) provide a good introduction to the application of these methods on the CVRP, while Nemhauser and Wolsey (1999) provide an in-depth explanation of these methodologies applied to different combinatorial optimisation problems.

Constraint Programming. Constraint Programming (CP) (Rossi et al., 2006) is a paradigm for representing and solving a wide variety of problems. CP combines techniques from artificial intelligence, logic programming, and operations research. CP is traditionally an exact approach. However, by virtue of the decoupling between modelling of the problem and searching for a solution that it is capable of doing, many alternative search methods, not necessarily complete, have been devised.

Problems are modeled in terms of decision variables, domains for those variables and constraints between the variables. In addition to expressions involving the usual arithmetic and logical operators, complex symbolic constraints can be used to describe a problem. The problems are then solved using complete search techniques such as depth-first search (for satisfaction) and B&B (for optimisation). However, for routing problems of practical size, complete search methods cannot produce solutions in a short time. By contrast, iterative improvement methods (refer to 2.4.2) have proved very successful in this regard. Iterative improvement methods operate by changing small parts of the solution, for instance moving a visit from one route to another. CP enhances the search using constraint propagation. If bounds or constraints on a variable can be inferred, or are tentatively set, these changes are "propagated" through all the constraints to reduce the domains of constrained variables.

CP has proved to be a very efficient approach for solving job scheduling problems, but such is not the case for VRP. Very few efficient methods based on CP have been published for solving routing problems. Shaw (1998) coupled a local search method he termed Large Neighborhood Search (LNS) with constraint-based technology to solve vehicle routing problems. LNS explores a large neighborhood of the current solution by removing customer visits from the set of planned routes, and re-inserting these visits using a constraint-based tree search. His results over benchmark problems showed his method to be competitive with OR metaheuristic procedures. De Backer et al. (2000) introduced a CP model for vehicle routing, and a system for integrating CP and iterative improvement techniques. They coupled their iterative improvement technique with a metaheuristic to avoid the search getting trapped in local minima. Two metaheuristics were investigated: a simple tabu search procedure and guided local search. An empirical study over benchmark problems shows the relative merits of their techniques.

2.3.2 Approximate Methods

These procedures represent a possibility to overcome the performance issue of exact methodologies. Approximate methods, also known as incomplete, explore the search space in a non-exhaustive (even, possibly, stochastic) way usually improving an incumbent solution iteratively. Many approximate methods cannot offer any guarantee about the quality of the solutions found, because they cannot detect when the search space has been completely explored, which might never happen. On the other hand, they have two important advantages. First, since they do not have to be exhaustive, they can explore promising regions of the search space much earlier than exact methods. Second, since they can move freely through the search space, without sticking to a fixed systematic exploration rule, they usually exhibit better anytime properties, i.e., the more time is given to the search method, the better is the returned solution. Two kinds of algorithms compose this class: approximation algorithms and heuristic algorithms.

Approximation Algorithms. Unlike heuristics, which usually find good quality solutions in a reasonable time but without guarantees, approximation algorithms provide provable solution quality and provable run-time bounds. For example, an ϵ-method guarantees that the solution obtained is at most ϵ times more costly than the best solution attainable. The aim in designing an approximation algorithm for a problem is to find tight worst-case bounds. It gives more knowledge on the difficulty of the problem and can help to design efficient heuristics. However, this type of algorithms are problem-dependent and this limits their applicability. Moreover, for real-life applications it might not be possible to design a polynomial-time approximation algorithm with constant error guarantee or approximation can be impractical. This is to say, error guarantee may be too poor or the running time of the algorithm may be too high. For example, when solving the TSP using benchmark instances, the best approximation ratio is 1.5, whereas the best heuristics are less than 1% away from optimality. Vazirani (2003) provides an in-depth presentation of these algorithms.

Heuristics. The concept of heuristic search as an aid to problem solving was firstly introduced by Polya (1945). A heuristic is a solution method which seeks (and hopefully finds) a feasible solution of good quality within modest computing time. Modest is defined more exactly as a time which is bounded by a low-order polynomial function. A heuristic is approximate in the sense that it provides (hopefully) a good solution for relatively little effort, but it does not guarantee optimality. Neither does it provide an approximation guarantee on the obtained solutions. These procedures are tested empirically and the quality of the solutions obtained can be evaluated by comparing them against the output of an exact method when possible, or against the output of other heuristic approach. This implies the existence of standard data sets. Heuristics are conspicuously preferable in practical applications because of their easiness to adapt them to diverse and complex constraints (usually present in real problems), and their ability to solve large instances in polynomial time. These techniques typically require an ad hoc construction of the algorithm applied.

Heuristics can be categorised broadly into two types: problem-specific and metaheuristics. *Problem-specific heuristics* are problem-dependent algorithms. As such, they usually are adapted to the problem at hand and they try to take full advantage of the particularities or structure of the problem. However, when these algorithms are followed by an improvement procedure (local search), they perform a relatively limited exploration of the search space, and usually exhibit the limitation of getting trapped in a local optimum and thus fail, in general, to obtain the global optimum solution.

Metaheuristics, on the other hand, are problem-independent techniques. As such, they do not take advantage of any specificity of the problem and, therefore, can provide general frameworks that may be applied to many problem classes. Blum and Roli (2003) compile the formal definitions that several researchers have tried to give of these procedures. In short, one may view them as iterative upper level general methodologies that can furnish a guiding strategy in designing subordinate heuristics to solve specific optimisation problems. The underlying heuristics may be high (or low) level procedures, or a simple local search, or just a construction method.

Compared to problem-specific heuristics, metaheuristics incorporate strategies which allow them to explore more thoroughly the most promising regions of the solution space. Consequently, solution quality is much higher (many times even finding the global optimum), but so is the computing time. Even though a metaheuristic is a problem-independent technique, it is nonetheless necessary to fine tune its intrinsic parameters in order to adapt the technique to the problem at hand.

Metaheuristics have been heavily researched in the past 30 years and very impressive results have been attained using these heuristics. Their use in many applications shows their efficiency and effectiveness to solve large and complex problems. The family of metaheuristics includes, but is not limited to, adaptive memory procedures, tabu search, ant systems, greedy randomized adaptive search, variable neighborhood search, large neighborhood search, evolutionary methods, genetic algorithms, scatter search, neural networks, simulated annealing. Voß (2001), Blum and Roli (2003) and Gendreau and Potvin (2005) provide an introduction to these methods, while Talbi (2009) presents a more in-depth explanation. The recently released book of Labadie et al. (2016) is specifically dedicated to the application of metaheurisics to solve VRP. The aim of the authors is to provide a book for people wishing to discover and quickly master metaheuristics dedicated to VRP. They combine tutorials with algorithms, examples, and a quick overview of the state-of-the-art for these methods in the context of the CVRP and some of its variants.

Different criteria exist to classify metaheuristics. I mention only one: number of solutions used at the same time. Population-based perform search processes which describe the evolution of a set of points in the search space, while single-solution, as the name says, work on only one at a time.

2.4 Classical Heuristics in Vehicle Routing

Laporte and Semet (2002) name the problem-specific heuristics for vehicle routing *clas-*

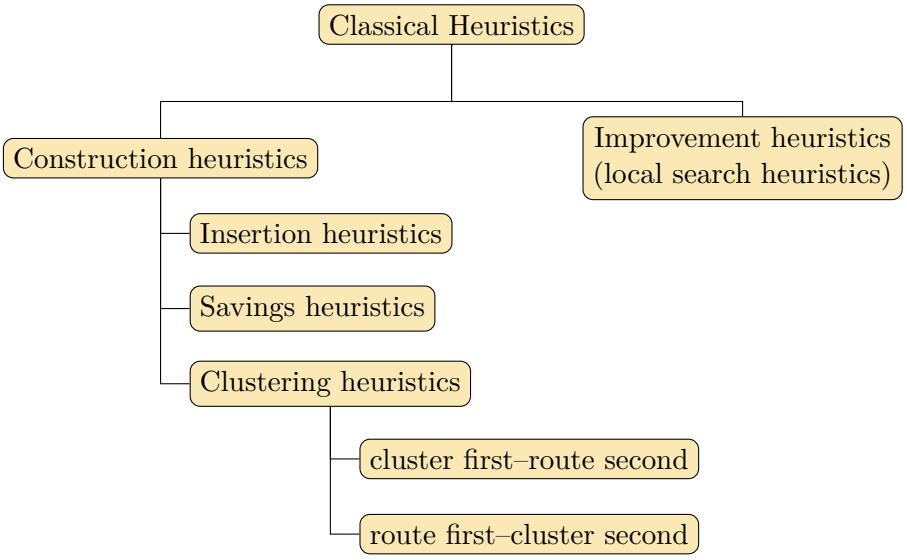

Figure 2.3: Families of problem-specific heuristics proposed for solving the VRP.

sical heuristics, and identify three categories within this type: constructive heuristics, two-phase heuristics and improvement methods. The category two-phase heuristics includes those algorithms that divide the construction into two phases: a *routing* phase and a *clustering* phase. In fact, this class can be seen as a subset of the construction heuristics: one stage is in charge of route construction, while the other one constructs clusters of vertices. This is how two-phase methods are considered in this manuscript, see Figure 2.3.

A constructive algorithm builds a solution from scratch by assigning values to one or more decision variables at a time; while an improvement algorithm generally starts with a constructed feasible solution and iteratively attempts to obtain a better solution. No consideration was made in this classification of the destruction heuristics that are nowadays frequently used to solve routing problems.

2.4.1 Construction Heuristics

Laporte and Semet (2002) define them as: procedures that gradually build a feasible solution while keeping an eye on solution cost, but they do not contain an improvement phase per se. Since the 1960s, a great deal of constructive heuristics have been proposed for vehicle routing problems. However, as metaheuristics became more dominant due to their superior performance in terms of solution quality, the popularity of this type of heuristics as stand-alone procedures has somewhat faded in the scientific literature. They are now more used as subroutines inside more time consuming metaheuristics. This approach is used in the solution methodology proposed in this thesis.

Many construction heuristics for vehicle routing problems fall into one of the three classes: insertion heuristics, savings heuristics and clustering heuristics.

Insertion Heuristics. These procedures build a solution by inserting one vertex at a time. They can construct one route at a time (sequential insertion heuristics) or build many or all routes in parallel (parallel insertion heuristics). The choice of which vertex to insert and where to insert it is what differentiates insertion heuristics. In this work, for example, an insertion heuristic which inserts the vertex that increases the overall cost the least is used.

Savings Heuristics. These heuristics initially build a solution by assigning each vertex to its own route. Routes are then merged one by one according to some criteria. Savings algorithms vary by the criterion used to merge routes (amount of savings obtained by merging two routes). Clarke and Wright (1964) were the first ones to propose a savings heuristic, and many variants and improvements of the algorithm have been published since then.

Clustering Heuristics. They are two-phase algorithms based on the idea that solving a vehicle routing problem involves two decisions: partitioning the customers into clusters compatible with vehicle capacity, and ordering the customers in each cluster to get a route. Therefore, the clustering phase consists of grouping the vertices into subsets (clusters) where each is served by one feasible route (vehicle), and the routing phase then finds the best visitation order for each cluster. Feedback loops between the two stages might exist. The *cluster first–route second* approach determines the partition first, and then a TSP is solved for each cluster, refer to Figure 2.4 where the circles represent customers and the square the depot. This technique includes, among others, the Gillett and Miller (1974) sweep algorithm where vertices are clustered in sectors of a circle whose center contains the depot, and customers in each sector of the circle are served by one route. Also, the Fisher and Jaikumar (1981) algorithm which builds the clusters by solving a *Generalized Assignment Problem* instead of using a geometric method. It also includes the petal algorithms which are an extension of the sweep algorithm. In these algorithms routes are named petals, and they are formed by solving a *Set Partitioning Problem*.

In the *route first–cluster second* approach the phases are inverted. A tour is first built on all vertices, and it is then partitioned into a set of feasible vehicle routes, see Figure 2.5. The idea was firstly proposed by Beasley (1983) when solving the CVRP. He noticed that the giant tour could be optimally segmented by finding the least cost path in the auxiliary acyclic graph that represents the set of n customers. This standard *Shortest Path Problem* (SPP) can be solved in $O(n^2)$ time using, for example, Dijkstra's algorithm or the Bellman-Ford algorithm. This possibility gives this approach an advantage over cluster first–route second: the clustering algorithm is an exact polynomial-time algorithm.

Beasley provided few computational results for his proposal, and the method neither outperformed more traditional CVRP heuristics nor was it given adequate recognition, see Laporte and Semet (2002). However, when Beasley's seminal method was efficiently implemented within a genetic algorithm (GA) (Prins, 2004), it proved to

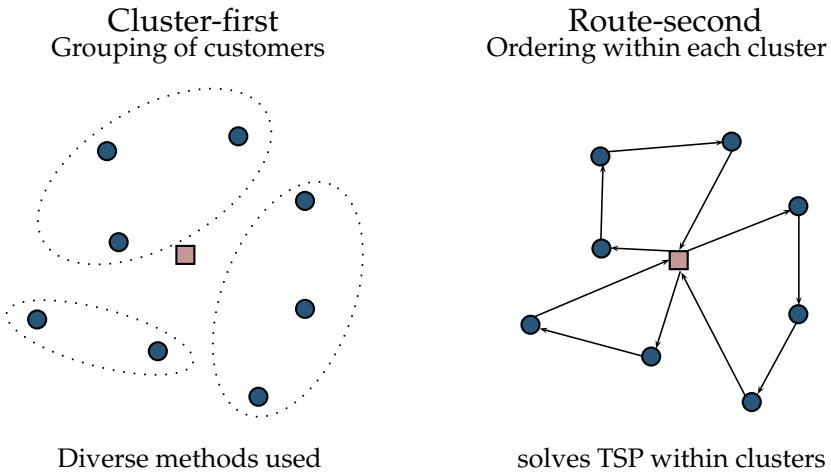

Figure 2.4: A cluster first–route second approach in vehicle routing problems.

be the first GA able to compete with the best methods available at that time for the solution of the CVRP, i.e., tabu search heuristics. The clustering algorithm is since known as the basic *Split* procedure. Other versions of the *Split* operator have also been developed in order to tackle additional constraints, (Prins et al., 2009).

In the last decade, the route first–cluster second approach has led to successful constructive heuristics and metaheuristics for routing problems as explained in Prins et al. (2014), where a more general name—order first–split second—is given to the methodology, and an analysis of 70 articles involving splitting procedures is made. A reason for this growing success is that a smaller solution space is searched since the search is done over the set of giant tours rather than over the much larger set of VRP solutions.

2.4.2 Improvement Heuristics

Laporte and Semet (2002) explain that improvement algorithms attempt to upgrade a feasible solution by performing a sequence of arc or vertex exchanges within or between vehicle routes. Tour improvement heuristics are based on the heuristic of Lin and Kernighan (1973) for the TSP. Here, λ edges are removed from the tour, and the λ remaining segments are reconnected in all possible ways. If any profitable reconnection is identified, it is implemented. The procedure stops when no further improvements can take place. The Lin-Kernighan heuristic modifies λ dynamically throughout the search.

Multi-route improvement heuristics for the VRP operate on each vehicle route, but taking on several routes at a time. Descriptions of multi-route edge exchanges for the VRP can be found in these three references. Thompson and Psaraftis (1993) describe a general "b-cyclic, k-transfer" scheme in which a circular permutation of b routes is considered and k customers from each route are shifted to the next route of the cyclic permutation. The authors show that applying specific sequences of b-cyclic, k-

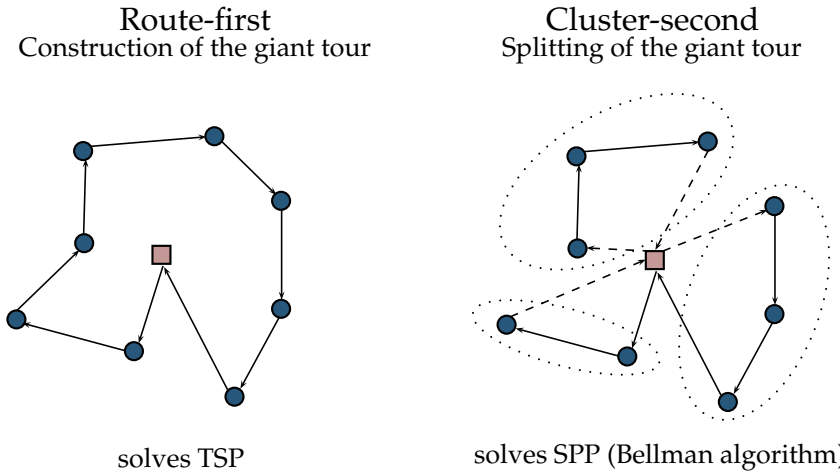

Figure 2.5: A route first–cluster second approach in vehicle routing problems.

transfer exchanges (with b = 2 or b variable, and k = 1 or 2) yields interesting results. Van Breedam (1994) classifies the improvement operations as "string cross", "string exchange", "string relocation", and "string mix", which can all be viewed as special cases of 2-cyclic exchanges, and provides a computational analysis on a restricted number of test problems. Kindervater and Savelsbergh (1997) define similar operations and perform experiments mostly in the context of the VRP with time windows.

The term improvement heuristic used by Laporte and Semet (2002) describes a *local search* heuristic that only performs moves (operations) that lead to a betterment of the objective function value. Local search heuristics start from a complete solution (obtained by a constructive heuristic or even randomly), modify a rather small part of it by performing a sequence of operations (changing the values of some variables), and produce a new, hopefully, improved solution. In more advanced local search heuristics, the algorithm sometimes performs changes that lead to a solution that is worse than the current. This is done as one can hope to find an even better solution after a few more changes. In the following section, local search heuristics are introduced more formally.

2.5 Large Neighborhood Search Metaheuristics

The focus of this section is a particular very large-scale neighborhood search metaheuristic called large neighborhood search (LNS) and specifically an extension of it named adaptive large neighborhood search (ALNS). However, some fundamentals will be presented first.

An important concept in local search heuristics is that of *move*. A move is an operation that transforms a solution s into another solution s' that shares some characteristics of s. Local search algorithms, alternatively called neighborhood search algorithms, are a wide class of improvement algorithms where at each iteration an improving solu-

tion is found by searching the *neighborhood* of the current solution. That is, this kind of methods try to improve an initial solution of an optimisation problem by repeatedly applying a local change (move). In each iteration, the heuristic replaces the current solution by some neighboring one which improves the objective function value. However, some schemes allow the acceptance of solutions that are worse than the current one as explained before. Two key issues in local search algorithms are the definition of a neighborhood and the way to examine it.

2.5.1 Neighborhoods

Let $\mathcal{I}$ be an instance of a combinatorial optimisation problem, and $\mathcal{S}(\mathcal{I})$ be the set of feasible solutions of $\mathcal{I}$. In order to have a quality measure for a solution, a cost function $c : \mathcal{S}(\mathcal{I}) \to \mathbb{R}$ is defined. For each solution $s \in \mathcal{S}(\mathcal{I})$ all the candidate solutions that are reachable in one logical step of the algorithm are defined as the *neighborhood* of s. More formally, a neighborhood $\mathcal{N}(s)$ can be defined as a function $\mathcal{N} : \mathcal{S}(\mathcal{I}) \to \mathcal{P}(\mathcal{S}(\mathcal{I}))$ that assigns to every candidate solution s in the search space $\mathcal{S}(\mathcal{I})$ a set of neighbors $\mathcal{N}(s) \subseteq \mathcal{S}(\mathcal{I})$. Frequently, modifications to obtain neighboring solutions introduce rather small changes. The simplest local search algorithm only accepts better neighbors and for this reason it is called iterative improvement. It terminates once no improving neighbor is available anymore. In such case, a local optimum has been found. More formally, a solution s is a *local optimum* with respect to a neighborhood $\mathcal{N}$ if $c(s) \leq c(s') \quad \forall s' \in \mathcal{N}(s)$. The vast number of solutions contained in $\mathcal{S}(\mathcal{I})$ makes it impossible in practice to search through it completely.

A simple example of a neighborhood for the TSP is the 2-*opt* neighborhood. Here, for a solution s, the neighborhood consists of the set of solutions that can be reached from s by deleting two edges in s and adding two other edges in order to reconnect the tour. Another example, which can be applied to only one route or to several, is the relocate neighborhood where $\mathcal{N}(s)$ is defined as the set of solutions that can be created from s by relocating a single customer. The customer can be moved to other position in its current route or to other route. These two neighborhoods are used in the metaheuristic algorithm proposed in this thesis. A broad presentation of neighborhoods for vehicle routing is provided by Funke et al. (2005).

Variants of local search may be distinguished according to the order in which the neighboring solutions are generated (deterministic/stochastic) and the selection strategy of the neighboring solution (best/first/random).

2.5.2 Neighborhood Search

The iterative improvement algorithm formerly outlined can be described with the above definitions. It takes an initial solution $s_{\min}$ as input. It finds the cheapest solution s' in the neighborhood of $s_{\min}$. If the cost of solution s' is better than the best known, $s_{\min}$ is updated . The neighborhood of the new solution $s_{\min}$ is searched for an improving solution and this is repeated until a local optimum $s_{\min}$ is reached. The algorithm is denoted *steepest descent* as it always chooses the best solution in the neighborhood.

Another strategy is to choose the first improving solution observed in the neighborhood. Such an algorithm would be a *descent algorithm*.

Algorithm 1 : Neighborhood Search (steepest descent)

Input: $s \in \mathcal{S}(\mathcal{I})$ initial solution
Output: $s_{\min} \in \mathcal{S}(\mathcal{I})$ best solution found

```
1: s_min ← s
2: improving ← true
3: while ( improving ) do
4:    s' ← arg min_{ŝ∈N(s_min)}{c(ŝ)}
5:    if c(s') < c(s_min) then
6:       s_min ← s'
7:    else
8:       improving ← false
9:    end if
10: end while
11: return s_min
```

The main disadvantage of neighborhood search algorithms is the convergence towards local optima, so different alternatives have been proposed to avoid becoming trapped at local optima. These alternatives include (1) accepting non-improving neighbors (used in simulated annealing); (2) iterating from different initial solutions (used in GRASP); (3) changing the neighborhood (used in variable neighborhood search); and (4) changing the objective function (used in guided local search).

2.5.3 Large Neighborhood Search

As formerly stated, a neighborhood search approach requires a good choice of the neighborhood structure. Ahuja et al. (2002) explain that as a rule of thumb, the larger the neighborhood, the better the quality of the locally optimal solutions, and as a consequence, the greater the accuracy of the final solution obtained. Additionally, the possibility to make larger modifications to candidate solutions allows to traverse the search space in less steps. However, the larger the neighborhood, the longer it takes to search it. For this reason, a larger neighborhood does not necessarily produce a more effective heuristic unless it can be searched in a very efficient manner. To meet this challenge, algorithms that search large neighborhoods typically implement filtering techniques to limit them to a subset of the solutions which can be searched efficiently. Another approach is to define neighborhoods that although exponential in size admit polynomial-time algorithms for their exploration, or the exploitation of problem-specific knowledge to speed up the search as much as possible. One example of these polynomial search techniques is given by dynamic programming approaches known as *dynasearch*. The interested reader may consult Chiarandini et al. (2008) for further explanations.

A precise definition of when a neighborhood is large or small is not simple to give. Ahuja et al. (2002) define large neighborhoods as those whose size grows exponentially

with the number of solution components that are allowed to be changed by one single search step. Nevertheless, they also consider those whose size is simply too large and cannot be searched explicitly in practice. Ahuja et al. (2002) name the algorithms that search this kind of neighborhoods *very large-scale neighborhood search* (VLSN) techniques. They identify three categories of VLSN algorithms: (1) variable-depth methods in which large neighborhoods are searched heuristically; (2) network flow-based or dynamic programming-based neighborhood search methods; and (3) problem-restricted methods solvable in polynomial time. Even though the concept of VLSN was formally defined recently, former methods widely used in operations research can be seen as based on the same principles. For example, if the simplex algorithm used for solving linear programs is viewed as a neighborhood search algorithm, then column generation is a VLSN method.

The Large Neighborhood Search (LNS) metaheuristic, originally proposed by Shaw (1997), does not fit well into any of the three categories defined by Ahuja et al. (2002), but it definitely belongs to the class of VLSN algorithms as it explores a very large neighborhood that cannot be searched fully. The LNS metaheuristic defines the neighborhood through one destroy and one repair method. These methods are usually problem-specific heuristics which replace the former neighborhood search function of Algorithm 1 (line 4). They specify the way in which the search is carried out.

A destroy method, as the name indicates, partially destroys an incumbent solution (removes some members), and a repair method then rebuilds it (reinserts the members back) to obtain a new feasible solution. Thus, the neighborhood $\mathcal{N}(s)$ of a solution is the set of solutions that can be obtained by applying a destroy-repair pair. The neighborhood $\mathcal{N}(s)$ contains a very large number of solutions since a destroy method can remove a large part of the solution. Consider an instance of a giant tour with 100 customers, and a method that removes 25% of the customers. There are $C(100, 25) = 100!/(25! \times 75!) = 2.4 \times 10^{23}$ different ways to select the customers to be removed, and for each removal choice there are many ways of repairing the solution. Of course, different removal choices can yield the same solution after the repair, but still the size of $\mathcal{N}(s)$ is huge.

Algorithm 2 depicts the LNS metaheuristic for a minimisation problem. The initial solution may be obtained with a constructive heuristic and parameter k sets the scope of the search. Three variables are used: (1) $s_{\min}$ stores the best solution found during the search; (2) s maintains the current solution; and (3) s' keeps the new solution obtained by applying the destroy-repair methods, and it can be promoted to current solution. Function $d(s, k)$ destroys a k portion of solution s, and function $r(\cdot)$ reconstructs it to obtain a new feasible solution, s'. Elements of stochasticity are introduced so that different parts of the solution are destroyed at every call of the method. The new solution s' is evaluated and the algorithm determines if it should be promoted to the current solution s. The *accept* function does not need to accept only improving solutions like in Shaw's original paper. Other possibilities are explained in the ensuing section. Function $c(\cdot)$ denotes the objective function. In line 7, the algorithm updates the best solution found if necessary. The typical termination criteria are a given number of iterations or a time limit. The algorithm returns the best solution found. One can

observe that the LNS metaheuristic does not search the entire neighborhood of a solution, but rather it merely samples this neighborhood.

The destroy–repair idea has been proposed in computer science under different names: remove and insert (Ropke and Pisinger, 2006), fix and optimise (Pisinger and Ropke, 2007), ruin and recreate (Schrimpf et al., 2000), ripup and reroute (Dees Jr and Karger, 1982), and remove and reinsert (Duff, 1967).

Algorithm 2 : Large Neighborhood Search

Input: $s \in \mathcal{S}(\mathcal{I})$ initial solution, $k \in \mathbb{N}$ parameter that determines the size of the neighborhood

Output: $s_{\min} \in \mathcal{S}(\mathcal{I})$ best solution found

1: $s_{\min} \leftarrow s$
2: **while** (stopping criterion not met) **do**
3: $s' \leftarrow r(d(s,k))$
4: **if** accept(s, s') **then**
5: $s \leftarrow s'$
6: **if** $c(s) < c(s_{\min})$ **then**
7: $s_{\min} \leftarrow s$
8: **end if**
9: **end if**
10: **end while**
11: **return** $s_{\min}$

2.5.4 Adaptive Large Neighborhood Search

Adaptive Large Neighborhood Search (ALNS) extends Shaw's LNS metaheuristic by allowing the use of several destroy and repair methods within the same search process. Also, it adds an adaptive layer which uses statistics gathered during the search to adjust the probability of choosing a destroy or repair method every defined number of iterations. In this way, it randomly controls which methods to choose according to their past performance (score). Therefore, ALNS operates on multiple large neighborhoods corresponding to the destroy (removal) and repair (insertion) sub-heuristics. Like in the LNS algorithm, the neighborhoods are not necessarily well defined in a formal mathematical sense, but, rather, are defined by the corresponding heuristic algorithms.

Algorithm 3 shows the pseudocode for the ALNS metaheuristic. Let $D = \{d_i | i = 1, \ldots, k\}$ be the set of k destroy sub-heuristics, and $R = \{r_i | i = 1, \ldots, l\}$ the set of l repair sub-heuristics. The probability vectors $P_d \in \mathbb{R}^{|D|}$ and $P_r \in \mathbb{R}^{|R|}$ store the probability of choosing each destroy and repair method respectively. Initially all methods are equally likely to be chosen. Variable ς stores the number of iterations performed before updating the probability vectors. Aside from the choice of a destroy-repair pair and the periodic probability updating, the structure of this algorithm is equal to the one of the LNS metaheuristic shown in Algorithm 2.

At each iteration, the algorithm selects, according to the adaptive probabilistic mechanism, a destroy-repair pair with which it destroys part of the current solution s,

Algorithm 3 : Adaptive Large Neighborhood Search

Input: $s \in \mathcal{S}(\mathcal{I})$ initial solution, $k \in \mathbb{N}$ parameter that determines the size of the neighborhood
Output: $s_{\min} \in \mathcal{S}(\mathcal{I})$ best solution found

```
1: s_min ← s; P_d = (1, ..., 1); P_r = (1, ..., 1)
2: while ( stopping criterion not met ) do
3:   for i = 1 to ς do
4:     select d ∈ D and r ∈ R according to probabilities P_d and P_r
5:     s' ← r(d(s, k))
6:     if accept(s, s') then
7:       s ← s'
8:       if c(s) < c(s_min) then
9:         s_min ← s
10:      end if
11:    end if
12:    update scores of sub-heuristics chosen
13:  end for
14:  update probability vectors P_d and P_r
15: end while
16: return s_min
```

and repairs it to generate a new solution s'. This new solution is accepted according to a criterion defined by a search mechanism applied at the master level, for example, the one proposed by simulated annealing (SA) where if s' is better than s, the search continues from s', otherwise, it continues from s' with some probability. Implementing a SA algorithm is straightforward as one solution is sampled in each iteration of the ALNS, but other choices are possible. Table 2.6 summarizes different ideas presented in the literature (Schrimpf et al. (2000), Dueck (1993), Ropke and Pisinger (2006), Hu et al. (1995)) for the *accept* function of Algorithm 3. All methods accept improving solutions, but they differ in the way they accept non-improving solutions.

ALNS competes strongly with genetic algorithms (GA) in vehicle routing. However, the efficiency of GAs relies on sophisticated local search procedures and population management techniques, while in ALNS, the neighborhoods are searched by simple and fast heuristics. The ALNS framework has several advantages. For most optimisation problems, a number of well-performing heuristics is already known and they can form the core of an ALNS algorithm. Also, due to the large size and diversity of the neighborhoods, the ALNS algorithm will explore large parts of the solution space in a structured way. Another advantage is its adaptive layer which allows the algorithm to work, for the instance at hand, with the best method among the ones available. Then, the resulting algorithm becomes very robust as it is able to adapt to various characteristics of the individual instances, and is seldom trapped in a local optimum. ALNS has provided very competitive solutions for a wide variety of routing problems. Some are explained in the following paragraphs.

Ropke and Pisinger (2006) used it to solve the *Pickup and Delivery Problem with*

Table 2.6: Different propositions for the *accept* function.

Method	Description
Random Walk	Every new solution s' is accepted.
Greedy Acceptance	The new solution s' is accepted only if $c(s') < c(s)$. The steepest descent method is of this kind.
Simulated Annealing	Every improving solution s' is always accepted. Otherwise s' is accepted with probability $e^{(\frac{c(s) - c(s')}{T})}$, where T is the so-called temperature. T decreases by a factor β at every iteration.
Threshold Accepting	A non-improving solution s' is accepted if $c(s') - c(s) < T$, where T is a threshold. T decreases by a factor α at every iteration.
Old Bachelor Acceptance	A non-improving solution s' is accepted if $c(s') - c(s) < T$, where T is a threshold. T decreases by a factor ϕ at every acceptance, and increases by a factor φ after every rejection.
Great Deluge Algorithm	A non-improving solution s' is accepted if $c(s') < L$, where L represents a level. L decreases by a factor ϑ only if the solution is accepted.

Time Windows. They tested their heuristic on more than 350 benchmark instances with up to 500 requests. Their method improved the best-known solutions from the literature for more than 50% of the problems. Pisinger and Ropke (2007) presented an ALNS-based unified heuristic able to solve five different variants of the VRP. All problem variants are transformed into a rich pickup and delivery model and solved using the ALNS framework. Ribeiro and Laporte (2012) developed an ALNS algorithm to solve the *Cumulative Capacitated Vehicle Routing Problem*, a variation of the classical CVRP, in which the objective is the minimisation of the sum of arrival times at the customers instead of the total routing cost. Their approach outperformed the then available heuristics.

Demir et al. (2012) published an ALNS heuristic for the *Pollution-Routing Problem* which determines the speed of the vehicles on each route segment so as to minimise a function comprising fuel, emission and driver costs. Kovacs et al. (2012) approached the service technician routing and scheduling problems which appear in infrastructure service and maintenance provision. The objective is to minimise the sum of the total routing and outsourcing costs. They solved both problem versions by means of an ALNS algorithm. It is tested on both artificial and real-world instances, and high quality solutions are obtained within short computation times.

Aksen et al. (2014) studied a selective and periodic inventory routing problem and developed an ALNS algorithm for its solution. The problem concerns a biodiesel

production facility which collects used oil from sources, such as restaurants, catering companies and hotels that produce waste vegetable oil in considerable amounts. The facility reuses the collected waste oil as raw material to produce biodiesel. The objective is to minimise the total collection, inventory and purchasing costs while meeting the raw material requirements and operational constraints.

Azi et al. (2014) solved *The Vehicle Routing Problem with Multiple Routes* (each vehicle can perform multiple routes during its operations day). This problem is relevant in applications where the duration of each route is limited, for example, when perishable goods are transported. The objective is first to maximise the number of served customers and then, to minimise the total distance traveled by the vehicles. An ALNS is proposed for solving this problem. Computational results on Euclidean instances, derived from well-known benchmark instances, demonstrate the benefits of this approach.

Tunalioglu et al. (2016) analysed a problem related to the olive oil production process. This process yields two by-products, one of which is the brown-coloured Olive Oil Mill Wastewater (OMWW) and has no direct use. OMWW is generally disposed of into soil or rivers, potentially polluting the environment. OMWW can be treated using ultrafiltration facilities, but this requires that it is collected from oil mills and delivered to the treatment facilities using a fleet of vehicles in an economically viable manner. Such considerations give rise to a multiperiod location-routing problem. They formally introduce the problem and propose an ALNS metaheuristic for its solution. The algorithm is applied on a case study drawn from one of the major olive oil producing countries.

2.6 Hybrid Metaheuristics

Nowadays, metaheuristics applied to the solution of combinatorial optimisation problems have shifted towards the hybridization of these procedures with other optimisation techniques. These hybrids are also named cooperative approaches. The realization that a performance limit had been hit or encountering convergence issues led researchers towards the exploration of integrating metaheuristics with other techniques. This trend originated the field of hybrid metaheuristics, which brought together practitioners from many optimisation paradigms, e.g., constraint programming, mathematical programming, machine learning. The hybrid metaheuristics community has now come of age, and has its own set of conferences and journals. Moreover, many well-established hybrid search techniques, which simultaneously exploit the advantages of a wide range of algorithms, have been developed.

The main motivation behind this tendency is to benefit from synergy. This is, choosing an adequate combination of complementary algorithmic concepts might be the key for achieving top performance in solving NP-hard optimisation problems. However, developing an effective hybrid approach is challenging since it requires expertise from different areas of optimisation. There are several possible hybridization schemes:

- combining metaheuristics with (meta)heuristics,

- combining metaheuristics with exact methods from mathematical programming approaches (hybrids are known as matheuristics),
- combining metaheuristics with constraint programming approaches,
- combining metaheuristics with dynamic programming,
- combining metaheuristics with data mining and machine learning techniques

An explanation of these topics is beyond the scope of this chapter. The interested reader may consult the papers of Blum et al. (2008) Jourdan et al. (2009), Blum et al. (2011), and Milano and Van Hentenryck (2011), and the PhD. thesis of Urli (2014). Archetti and Speranza (2014) present a survey of matheuristics for routing problems.

2.7 Conclusions

This chapter presents a summary of the problems studied, which are VRPOV. Two families of problems may be identified within this subclass: covering problems and profit problems. Both exact and approximate methods can be used to solve them. The size of the instances to solve determines the strategy to use. Among the approximate methods, metaheuristics are the preferred choice since they can search more thoroughly the solution space. They define a general-purpose iterative master process which guides and modifies the operations of subordinate heuristics to efficiently produce high-quality solutions. Nevertheless, the actual trend is to implement hybrid metaheuristics that take advantage of both exact and heuristic methods for specific purposes.

Local search heuristics are often built on moves that make small changes to the current solution, so they are able to investigate a large number of solutions in a short time. Since the solution changes only slightly, such heuristics can have difficulties in moving from one promising area of the search space to another. For this reason, instead of using small conventional moves, a heuristic that explores very large moves that can rearrange a high percentage of the solution is advantageous. However, larger moves require more computational time to perform and evaluate, but the metaheuristic may evaluate only a fraction of the solutions that could be evaluated by a standard heuristic and still exhibit very good performance as observed in the solution approaches reported in the literature.

Chapter 3

Solution Methodology Developed

Contents

3.1 Introduction

In this chapter, the unified solution approach proposed to solve VRPOV is explained. The approach is based on the route first–cluster second idea introduced by Beasley (1983) for solving the *Capacitated Vehicle Routing Problem* (CVRP). As a matter of fact, Beasley coined the name of the approach based on the tasks required to be accomplished at each stage, but he did not propose any method for the routing phase. When solving the CVRP, he considered a TSP tour was made available by some procedure (exact or heuristic), and only proposed a way to optimally segment the given TSP tour into feasible vehicle routes: find the shortest path on the network that represents the set of customers. Prins (2004) implemented this idea on the *Split* algorithm and used it to evaluate a given giant tour (a tour that visits all the vertices of the given customer

network) within a population solution-based metaheuristic, a genetic algorithm. Since then, very successful metaheuristics based on this route first–cluster second idea have been implemented to solve diverse vehicle routing problems. A reason for this success is that a smaller solution space is searched, since the search is done over the set of giant tours rather than over the much larger set of VRP solutions. In some implementations, the giant tour is replaced by an ordering of the customers or by a priority list which explains the more general name of order first–split second given to this class of heuristics by Prins et al. (2014) in their recent survey.

This thesis work developed a hybrid metaheuristic of this family for solving VRPOV. Its main feature is a dynamic programming-based operator aimed at extracting VRPOV solutions. The routing phase of the unified solution approach proposed is handled by a generic metaheuristic which produces high quality giant tours using subordinate heuristics. The applied metaheuristic for this work is the adaptive large neighborhood search introduced by Ropke and Pisinger (2006). The clustering phase is solved by the dynamic programming-based operator which is embedded into the metaheuristic. The operator, named *Selector*, accomplishes the following tasks: (i) optimally selects the vertices to visit in the giant tour formed by the sub-heuristics (clusters the vertices into visited and non-visited subsequences); (ii) evaluates route costs and (iii) in multi-vehicle problems, co-works with the *Split* operator to optimally segment a selected set of vertices into feasible vehicle routes. All in all, the overall task of the metaheuristic is to build good quality giant tours from which the *Selector* operator retrieves efficient VRPOV solutions.

The *Selector* operator is a novel algorithm introduced in this thesis work. Preliminary work on it was presented in the XVII Conferencia Latino Americana en Investigación de Operaciones (CLAIO 2014), Monterrey, Mexico, October 6-10, 2014. Further work was later presented in the 9$^{\text{th}}$ Learning and Intelligent OptimizatioN Conference (LION9), Lille, France, January 12-15, 2015, and a paper was published in the conference proceedings, (Vargas et al., 2015a). *Selector* was firstly applied to the solution of the CTP, and later on, the version implemented to solve the OP was presented in the 4$^{\text{th}}$ meeting of the EURO Working Group on Vehicle Routing and Logistics Optimization (VeRoLog 2015) in Vienna, Austria, June 8-10, 2015. The version of *Selector* aimed at solving multi-vehicle VRPOV was applied to the solution of the m-CTP, and the results were presented at the 17ème Conférence de la Société Française de Recherche Opérationnelle et d'Aide à la Décision (ROADEF 2016) in Compiègne, France, February 10-12, 2016.

Vidal et al. (2015) recently published a large neighborhood search for multi-vehicle routing problems with profits based on an algorithm which follows similar principles as *Selector*. They heuristically produce a conventional VRP solution, which they name exhaustive solution representation because it visits all customers, and then repeatedly apply their select algorithm on each new route in order to retrieve the optimal subsequences of visits to customers. They test their neighborhood structure within three heuristic frameworks: a multi-start local search, a multi-start iterated local search based on the method of Prins (2009), and a hybrid genetic search (UHGS) derived directly from the general framework of Vidal et al. (2014). *Selector* is envisioned as a

unified operator to solve VRPOV, whereas their select algorithm is conceived only in the context of multi-vehicle routing problems with profits.

The remainder of the chapter is organised as follows. The starting point of the explanation is a brief presentation, in Section 3.2, of the *Split* operator of Prins (2004) and of the labelling algorithm of Desrochers (1988). Next, a detailed explanation of the *Selector* algorithm is provided in Section 3.3. Sections 3.4 through 3.6 present modifications done to the basic *Selector* algorithm to fulfill specific purposes. The implementation made of the selected metaheuristic is later detailed in Section 3.7. The final remarks are presented in Section 3.8.

3.2 Foundations of *Selector*

When solving a VRPOV, the *Selector* operator optimally splits a giant tour into subsequences of visited and non-visited vertices in a similar way as the *Split* operator optimally segments a giant tour into feasible vehicle routes when applied to solve the CVRP as proposed by Prins (2004). Splitting the giant tour into vehicle routes entails solving a *Shortest Path Problem* (SPP). However, in the case of a VRPOV, the side restriction(s) considered act(s) as a constraining resource and the problem to be solved then becomes a *Resource-Constrained Elementary Shortest Path Problem* (RCESPP).

The *Selector* operator shares similarities with the *Split* operator. However, the fact of not knowing a priori which vertices constitute the tour leaves a more difficult to solve problem. Nonetheless, the RCESPP for a VRPOV can be solved quickly enough in practice by adapting the algorithm devised by Desrochers (1988) for the *Resource Constrained Shortest Path Problem* (RCSPP). The elementary requirement is guaranteed by the fact that the order given by the giant tour is respected.

This section provides an explanation of the mechanics of the *Split* process and of the pseudocode of its algorithm together with the fundamental principles of the algorithm of Desrochers (1988).

3.2.1 Split Operator

As stated before, Prins (2004) provided the first computational implementation of the clustering idea of Beasley (1983) for finding the optimal routes in a CVRP. Using any TSP algorithm, the first phase of *Split* computes a giant tour of all customers $T = (T_1, T_2, , \ldots, T_n)$ by relaxing vehicle capacity Q and maximum route length L. The second phase builds an auxiliary acyclic graph H from T, and applies the well established Bellman-Ford shortest-path algorithm to obtain an optimal partition of the giant tour into least-cost, capacity-feasible vehicle routes.

The specifics are as follows. H contains $n + 1$ vertices indexed from 0 to n. Each cluster of customers $(T_i, T_{i+1}, \ldots, T_j)$ that can be treated as one feasible trip, i.e., a trip for which all the problem constraints are satisfied, is modelled by one arc $(i-1, j)$ in H. The weight of the arc is given by the trip cost $cost(i, j) = c(0, T_i) + \sum_{k=i}^{j-1}(s(T_k) + c(T_k, T_k + 1)) + s(T_j) + c(T_j, 0)$, where $s(T_i)$ represents the service cost of customer T_i. To find the shortest path, the algorithm inserts visits to the depot within the

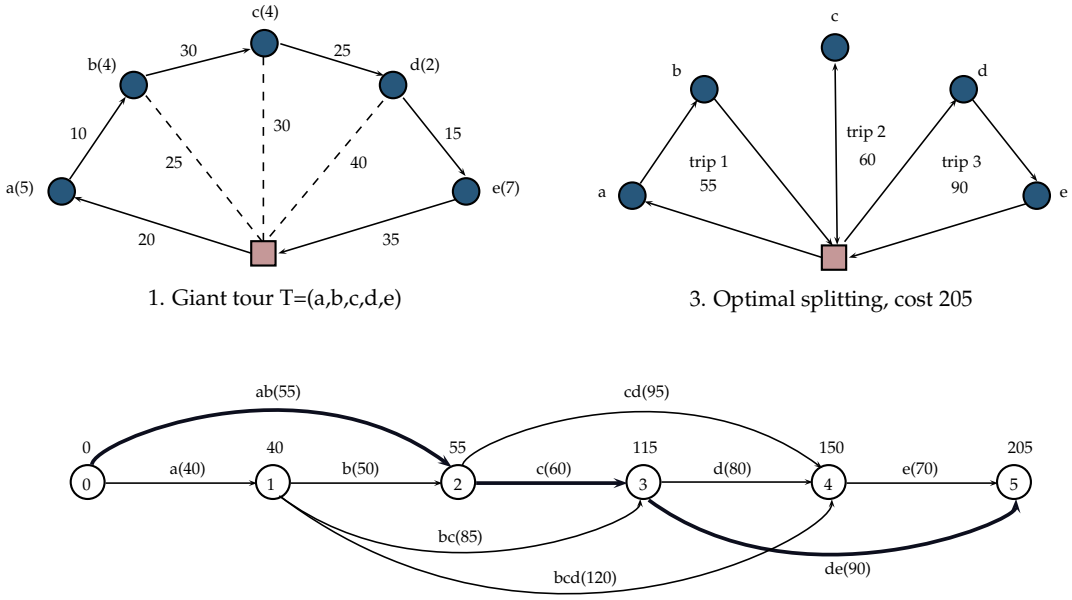

Figure 3.1: Example of a splitting procedure.

sequence, so the depot is first considered at vertex v_0, and then it is moved iteratively to vertices $v_1, v_2, \ldots, v_{n-1}$. Thus, when reading an arc, interpret the vertex where the tail lies as the depot rather than as a customer vertex. A shortest path from vertex v_0 to vertex v_n in H corresponds to an optimal splitting of T. The splitting is optimal for the given customer ordering. Figure 3.1 illustrates an example with five customers. In drawing 1, the set of customers is represented as the giant tour $T = (a, b, c, d, e)$, and the figures represent the traveling costs between customers and between customers and the depot, the ones in brackets are the customer demands. The graph depicted in drawing 2 is built assuming $Q = 10$, and $L = \infty$. For example, arc *cd* models a trip visiting customers c and d, with cost 95, while arc *cde* does not exist because the trip is not feasible due to excessive demand (13). The shortest path has three arcs (ab,c,de) and a cost of 205. The computation of the shortest path is reasonably fast because H is circuitless and the vertex numbering provides a natural topological ordering.

The *Split* procedure, shown in Algorithm 4, is a version in $O(n)$ space which does not generate H explicitly. Instead, for each vertex $j = 1, 2, \ldots, n$ in H, it maintains two labels: (1) V_j, the cost of the shortest path from vertex v_0 to vertex v_j in H, and (2) P_j, the predecessor of vertex v_j on this path. The *repeat* loop enumerates all feasible subsequences $T_i \ldots T_j$, and instead of storing the arcs of feasible trips, the labels of vertex v_j are directly updated when improved (lines 17 and 18). At the end, the total cost is stored in V_n and the set of routes is obtained from the information stored in vector P. For a given i—pointer which indicates the current position of

the depot—note that j is increased until the route length L is exceeded: no feasible trip is discarded since the triangle inequality holds. The resulting algorithm is simple, compact and fast, for further details see Prins (2004).

Algorithm 4 : Clustering algorithm for *Split* procedure

Input: giant tour
Output: optimal set of routes for the given giant tour, total route length

$V_0 \leftarrow 0$
for $i = 1$ **to** n **do**
  $V_i \leftarrow \infty$
end for
for $i = 1$ **to** n **do**
  $load \leftarrow 0$; $cost \leftarrow 0$; $j \leftarrow i$
  repeat
    $load \leftarrow load + q_{T_i}$
    if $i = j$ **then**
      $cost \leftarrow cost(0, T_j) + s_{T_j} + cost(T_j, 0)$
    else
      $cost \leftarrow cost - cost(T_{j-1}, 0) + cost(T_{j-1}, T_j) + s_{T_j} + cost(T_j, 0)$
    end if
    if $(load \leq Q)$ **and** $(cost \leq L)$ **then**
      {here subsequence $T_i \dots T_j$ corresponds to arc $(i-1, j)$ in H}
      if $V_{i-1} + cost < V_j$ **then**
        $V_j \leftarrow V_{i-1} + cost$
        $P_j \leftarrow i - 1$
      end if
      $j \leftarrow j + 1$
    end if
  until $(j > n)$ **or** $(load > Q)$ **or** $(cost > L)$
end for
Decode(P)
return set of routes

3.2.2 Resource-Constrained Elementary Shortest Path Problem

The *Resource-Constrained Elementary Shortest Path Problem* (RCESPP) is the problem of finding an elementary[1] shortest path from a source vertex v_s to a target vertex v_t in a network such that the overall resource usage does not exceed some given limits. Hence, record of the resources used by each path should be kept. Resources are consumed when visiting vertices or traversing arcs. Such problem is NP-hard (Dror (1994)). The standard approach to solve a RCESPP to optimality in pseudo-polynomial time is dynamic programming.

This approach relies upon the seminal work of Desrochers (1988) who proposed an

[1]synonym of Hamiltonian, vertices are visited only once

algorithm to solve a relaxed version of this problem, the *Resource Constrained Shortest Path Problem* (RCSPP), where the path needs not be elementary. Desrochers' procedure is a multi-label extension of the Bellman-Ford algorithm taking resource constraints into consideration. The dynamic programming-based Bellman-Ford algorithm is designed to find a shortest path in a graph. It is a label-correcting approach where a single label representing the cost of the path which gradually improves is assigned to each vertex of the given network. However, in the case of the RCSPP, the necessity of keeping record of used resources obliges to assign several labels to each vertex.

Feillet et al. (2004) explains that in label-correcting approaches, vertices are repeatedly treated and their labels extended. Within this approach, two strategies can be used: reaching algorithms and pulling algorithms. In reaching algorithms, labels on a vertex are extended to its successors, while in pulling algorithms, labels from its predecessors are pulled to the vertex currently treated. Aside from the label-correcting approach, there is the label-setting approach, an extension of Dijkstra's algorithm. It works by the permanent marking of the labels, which are treated in an order based on the resource consumption.

In this context, Desrochers' procedure is a label-correcting reaching algorithm. Its basic principle is to associate a label with each partial path that goes from the origin vertex v_s to a vertex v_i. The label represents the cost of the path and its consumption of resources. Unnecessary labels are eliminated as the search progresses. Vertices are iteratively treated until no new labels are created. When a vertex is treated, all its new labels are extended toward every possible successor vertex. Throughout the search, then, every vertex receives several labels. When a label is extended from vertex v_i to vertex v_{i+1} to generate another feasible label, the cost and resource consumption of the new label must be computed according to a recurrence formula.

The so-called label, which represents a feasible path, can be understood as a vector $V = [\zeta | r_1, r_2, \ldots, r_m]$ that memorizes the path cost ζ and the resource consumptions r_i along the corresponding path. These consumptions enable to know if a partial path can still be extended. The efficiency of the dynamic programming-based algorithm outlined in the former paragraph relies heavily upon the possibility of pruning labels that cannot lead to an optimal solution. For this purpose, suitable dominance tests are always performed when labels are extended, so that only non-dominated labels are stored. To find the optimal path only non-dominated paths need to be considered. For a survey on models and algorithms for the RCSPP and the RCESPP, the interested reader may consult Irnich and Desaulniers (2004). Feillet et al. (2004) presented an exact algorithm to solve the RCESPP applied to VRPs. Also, the papers of Righini and Salani (2006) and Righini and Salani (2008) which propose new ideas for RCESPP algorithms, and the publication of Boland et al. (2006) which explains an implementation of a label setting algorithm to solve the RCESPP.

3.3 *Selector* Operator

The explanation now proceeds to the clustering mechanism proposed in this thesis: the *Selector* operator. *Selector* is a label-correcting reaching algorithm that, given a sequence of customers, selects which ones to visit in order to obtain the best solution value for the given goal while keeping the original routing order and satisfying side constraints. Formally, the operator formulates the problem of selecting the customers to visit in some given giant tour as a RCESPP on an auxiliary directed acyclic graph H. Every traversed arc $(i, j) \in H \mid i < j$ indicates a resource consumption. The input of the algorithm is a permutation σ of a vertex set V which represents a set of customers. The depot is always the first vertex in this permutation. The output is a subset $V' \subset V$ to be visited in the same order given in σ such that the value of the objective function considered is optimal and the resource constraints are satisfied. It is important to note that such solution technique implies there is no need for a local search based on insertions and removals of customers as is the case in many heuristics, e.g., Campos et al. (2014), Vansteenwegen et al. (2009), Gendreau et al. (1997). Instead, our algorithm builds all the possibilities pruning those that cannot lead to the optimal solution.

3.3.1 Auxiliary Graph

The *Selector* operator can be described as searching for an optimal path in a directed acyclic graph that portrays how the vertices of the giant tour are connected to each other. It builds the set of vertices $S = \bigcup_{i \in [0,n-1]} \{\sigma_i\}$ of graph H from an initial permutation of the n vertices that can be visited. In other words, the topological order of the vertices contained in the auxiliary graph represent the position of a customer in the giant tour. In order to represent non-visited vertices, this set is replicated $|S| - 2$ times, eliminating the first element each time. As a result, there are now $n(n-1)/2$ vertices, plus the depot and a copy of it, σ_0^+, arranged in $n-1$ levels (the depot belongs to level 1, and the different levels are numbered from top to bottom). For instance, a directed graph obtained from a network of customers $G = (V, A)$ where $|V| = 5$ is illustrated in Figure 3.2 where only a subset of the possible arcs is represented. In this figure, each vertex is denoted σ_j^i, where i represents the level the vertex belongs to, while j indicates the vertex number.

The arc set is built as follows. Arcs whose endpoints lie within the same level i are firstly explained. Every arc of this kind represents a continuous path. Such arcs always start at the depot, (σ_0, σ_j^i), and exist if the subsequence $\{\sigma_0, \sigma_1^i, \sigma_2^i, \ldots, \sigma_j^i\}$ is feasible, for instance, the subpath does not exceed the maximal length allowed. Thus, for example, arc (σ_0, σ_3^1) in Figure 3.2 indicates the path $\{\sigma_0, \sigma_1^1, \sigma_2^1, \sigma_3^1\}$. In order to make it a tour, arcs whose head lies at σ_0^+ do not indicate a path but a finishing point, i.e., arc (σ_3^1, σ_0^+) is subsequence $\{\sigma_3^1, \sigma_0^+\}$ (dashed line). In addition, an arc between two vertices that lie in different levels $(\sigma_i^k, \sigma_j^m) \mid k \leq m - 2$ exists if it is feasible to skip the subsequence formed by the vertices located at the first position of the in-between skipped levels. For example, arc (σ_1^1, σ_4^4), in bold in Figure 3.2, indicates

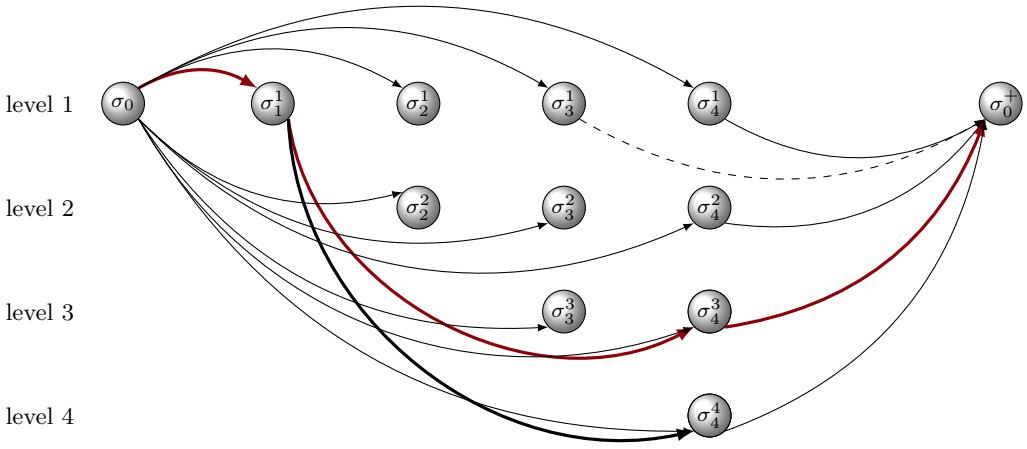

Figure 3.2: Auxiliary graph representing some of the possible arcs.

the subsequence $\{\sigma_2^2, \sigma_3^3\}$ has been skipped. Whereas, the coloured arcs exemplify the visited sequence $\{\sigma_0, \sigma_1^1, \sigma_3^3, \sigma_4^3, \sigma_0\}$.

A cost matrix specifying a non-negative integer cost between any two vertices σ_i is known, and these costs satisfy the triangle inequality. The cost of an arc is the sum of the weights of the arcs included in the subsequence. The problem is to search for an optimal path from σ_0 to σ_0^+ that meets a set of constraints that cannot be included in the definition of the arcs, for example, the need to cover a given set of vertices in a covering problem. The optimal path from vertex σ_0 to vertex σ_0^+ in H corresponds to an optimal splitting of the giant tour σ into visited and non-visited vertices. This search has pseudo-polynomial complexity of $O(Bn^2)$, where B represents the number of labels.

This auxiliary graph is described in order to clarify the algorithm. However, the vertices $\sigma^k \mid k > 1$ are not built explicitly. The algorithm maintains only the first level of vertices.

3.3.2 Labels

Dynamic programming is a technique that builds the sought solution in a bottom-up fashion by solving subproblems whose complexity gradually increases. In this case, the first subproblem solved is the one of finding the cost or profit and the resource consumption for the path $\{\sigma_0, \sigma_1\}$. With this information, it is possible to easily solve a similar subproblem for the path reaching vertex σ_2. Equally when reaching vertex σ_3, and so on. Therefore, the technique requires the storage of intermediate results to avoid recomputation. In practice, this is accomplished by using labels. In the case of *Selector*, a label represents a feasible elementary path that starts at σ_0 and has considered vertices up to σ_i. A label $\lambda = [z, i|r_1, r_2, \ldots, r_m]$ is defined as a triplet which stores: the value z of the objective function considered, the rank i reached in σ

(last vertex visited), and the resource consumption $r_1, r_2, \ldots, r_m$ that enables to know how a partial path can be extended. For example, in a covering problem, this resource consumption is interpreted as which vertices that must be covered are still uncovered, and this information enables to know if a vertex $v_i \in V$ can be skipped when extending a label. A vertex σ_i may be reached through different paths composed of visited and skipped predecessors, each with different cost and different use of the resources. As a result, a set of labels Λ^i is associated to each vertex σ_i, for $i \in \{0, \ldots, n-1\}$, starting with $\Lambda^0 = \{[0, 0, 0]\}$ for the vertex acting as the depot.

A label on a vertex is repeatedly extended to its successors until the considered restrictions prevent the creation of feasible labels. This operation is repeated until all labels have been extended in all feasible ways. When a label is extended, its objective function value (cost or profit) and its rank are computed using previous results, and the information linked to the side constraints (resources) is updated to ensure that a feasible solution is still possible. Then, for any i, a set of labels Λ^{i+1} is constructed by considering iteratively any arc $(j, i+1) \in H$ and extending all labels of j using a recurrence equation. Since every extension creates a new label, a way to control their proliferation is by applying dominance relations between pairs of labels. Suitable dominance criteria allow to identify labels whose extension cannot produce an optimal solution. Two labels can be compared only if both have reached the same rank i in σ. Also, the dominating label must be less or equally constrained by the resource information stored, and must offer a better or equal objective function value.

3.3.3 Algorithm

In an algorithm designed to visit all the vertices of the given network, a label can only be extended from vertex σ_i to successor vertex σ_{i+1} with no possibility of skipping vertices. In such case, the extension of a label corresponds to appending an additional arc $(i, i+1)$ to a path from σ_0 to σ_i, obtaining a feasible path from σ_0 to σ_{i+1}. The process also implies updating the objective function value, the rank reached $(i+1)$ and the resource consumption.

In *Selector*, a similar operation occurs when a label is extended, but in this algorithm vertices may be skipped, so we consider that a label is extended from vertex σ_i to vertex σ_{i+k}, where k may take any of the following values $\{1, 2, 3, \ldots, n-i-1\}$ with $n = |V|$. In addition, the extension of a label $\lambda = [z, i | r_1, r_2, \ldots, r_m]$ from vertex σ_i to successor vertex σ_{i+k}, implies that one of two possible operations is performed: visit vertex σ_{i+k}, or skip vertex σ_{i+k}. If when extending a label it is decided that the vertex σ_{i+k} is skipped, the extension proceeds to vertex σ_{i+k+1} and it is evaluated again if it is useful to visit it. Not visiting is possible, for example, when a vertex proves to be redundant or visiting it violates a restriction. However, no labels are created for skipped vertices. Only when the vertex is visited, a non-dominated label is stored.

When $k = 1$, the extension occurs to the immediate adjacent successor vertex. It is comparable to constructing an arc at some given level on the auxiliary graph H. On the other hand, when $2 \leq k \leq n - i - 1$, the extension occurs skipping a sequence of vertices. This is possible only if the side restrictions allow to skip the sequence

$\{\sigma_{i+1}, \ldots, \sigma_{i+k-1}\}$. This extension is equivalent to constructing an arc whose head lies at least two levels down in the auxiliary graph H.

Algorithm 5 illustrates the core procedure of *Selector*. The algorithm may be understood as first extending label $\lambda = [0, 0, 0]$ from the depot σ_0 to every level of graph H (lines 1-6). This is to say, extend the label λ to every vertex $\sigma_i \mid i \in \{1, 2, \ldots, n-1\}$ if possible, i.e., create the sequences of vertices $\{\sigma_0, \sigma_i\}$. If the extension is successful, it is followed by an extension that repeatedly attempts to visit the immediate adjacent successor vertex trying to go as deeply as it proves useful. In other words, it attempts to construct sequence $\{\sigma_0, \sigma_i, \sigma_{i+1}, \sigma_{i+2}, \ldots, \sigma_{i+j}\} \mid j = n - i - 1$. However, at each step of the construction of this sequence, i.e., at every visited vertex, a non-dominated label is stored. Later, if possible, the labels that were stored at each step are extended more deeply but now considering skipping defined sequences of vertices (lines 7-10). In other words, an initial set of labels is created for each visited vertex and the set is further extended later. The complexity of the first phase is $O(n^2)$. Figure 3.3 shows an example of the initial phase where only a small subset of V is shown for explanatory purposes.

Algorithm 5 : Selector

Input: giant tour σ, distance matrix D
Output: optimal tour of visited vertices T, cost value of tour $c(T)$

```
    {build an initial set of labels, Λ}
 1: for ( i = 1 to n − 1 ) do
 2:    if ( λ_i can be extended from σ_0 to σ_i ) then
 3:       Λ ← Λ ∪ {λ_i} {λ_i is the label being treated}
 4:       Extend(λ_i) {see Algorithm 6}
 5:    end if
 6: end for
    {extend labels created skipping a defined sequence of vertices}
 7: while ( Λ ≠ ∅ ) do
 8:    λ ← Extract Best(Λ) {find label with best objective function value}
 9:    Extend Skipping(λ) {see Algorithm 7}
10: end while
```

Algorithm 6 depicts the general process of repeatedly extending a label λ to the immediate adjacent successor. Its main tasks are the following: (i) evaluate if it is useful/possible to visit a vertex; in case it is, (ii) update the label data; (iii) update the best-known solution if applicable and (iv) determine if the label is non-dominated. In this algorithm, stopping criteria other than reaching σ_{n-1} can be defined. For example, finding a complete solution or finding a path whose objective function value is worse than the best incumbent solution. The algorithm is adapted to the problem treated to make it efficient, so it is further detailed in each of the chapters that explain the application of *Selector* to solve a specific problem.

Algorithm 7 shows the next phase. The initial labels are further extended skipping defined sequences of vertices. These sequences are located after the last rank visited.

Algorithm 6 : Extend(λ)

Input: label to be extended $\lambda = [\zeta, i \mid r_1, \ldots, r_m]$, z^*
Output: labels derived from λ
{only non-dominated labels that can be later extended skipping are kept}
{z^* represents the best known objective function value}

```
1: for ( j = i + 1 to n − 1 ) do
2:   if ( vertex σ_j is visited ) then
3:     update rank(λ_j) and resource consumption(λ_j)
4:     compute z(λ_j)
5:     compare z(λ_j) against z* and update z* if possible
6:     if ( λ_j is a complete solution ) then
7:       return
8:     end if
9:     if ( λ_j not dominated ) then
10:      Λ ← Λ ∪ {λ_j}
11:    end if
12:  end if
13: end for
```

Extending from vertex σ_i, the first successor that can be visited skipping intermediate vertices is σ_{i+2}, i.e., it skips one vertex. In addition, if the restrictions allow it, the maximum number of extensions ψ that can be done from σ_i to successors $\sigma_k \mid k \geq i+2$ is given by $\psi = n - i - 2$. Then, when a label λ is chosen for extension, it first attempts to reach vertex σ_{i+2}, this is, build sequence $\{\sigma_0, \ldots, \sigma_i, \sigma_{i+2}\}$, and from this point, it does an extension that iteratively attempts to visit the immediate adjacent successor vertex, i.e., it creates, if possible, sequence $\{\sigma_0, \ldots, \sigma_i, \sigma_{i+2}, \sigma_{i+3}, \ldots, \sigma_{i+j}\} \mid j = n - i - 1$. It continues with the next possible extension for λ which is to successor vertex σ_{i+3}, in this case it skips two vertices. It proceeds again with an iterative extension to the immediate adjacent successor vertex, i.e., it creates sequence $\{\sigma_0, \ldots, \sigma_i, \sigma_{i+3}, \sigma_{i+4}, \ldots, \sigma_{i+j}\} \mid j = n-i-1$. The process repeats, if possible, until ψ is reached. The number of extensions possible, ψ, may not be reached if, for example, it is not feasible to skip a certain vertex within the subsequence because vertices may be left uncovered. Then, the remaining possibilities that consider even larger subsequences of skipped vertices are not feasible and, therefore, not worth analysing.

At each iteration, the label chosen for extension is always the one that documents the best objective function value and the execution of this second step continues until there are no labels to extend. Recall that at every extension, the best solution found is updated so that at the end it can be retrieved from this variable. The complexity of this extension is $O(n^2)$. Figure 3.4 shows an example with the same assumptions explained for Figure 3.3.

A distinctive and important characteristic of *Selector* is that aside from the constraints mentioned in the definition of the problem, it does not impose any further restrictions on the selected vertices of V such as adjacency, for example. This operator can discard any vertex $v_i \in V$ at any point in the tour. Bouly et al. (2010) designed

Algorithm 7 : Extend Skipping(λ)

Input: label to be extended $\lambda = [\zeta, i \mid r_1, \ldots, r_m]$
Output: labels derived from λ

```
1: for ( k = 2 to n − i − 1) do
2:    if ( λ_i can be extended from σ_i to σ_{i+k} ) then
3:       Extend(λ_{i+k})
4:    end if
5: end for
```

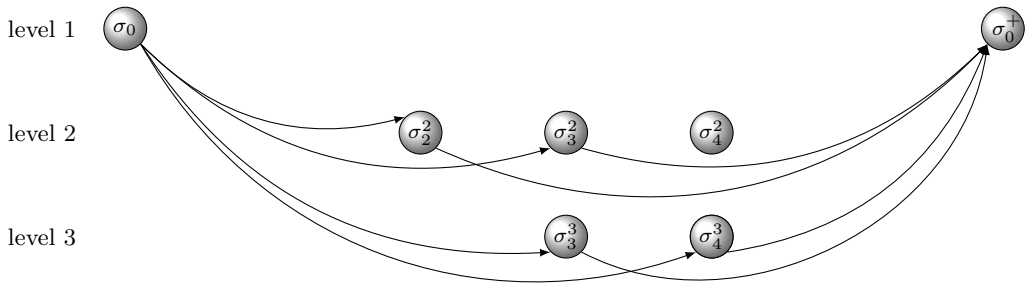

Figure 3.3: Phase 1: Create initial list. The labels created in level 2 document sequences: $\{\sigma_0, \sigma_2, \sigma_0\}$, $\{\sigma_0, \sigma_2, \sigma_3, \sigma_0\}$. Label $\{\sigma_0, \sigma_2, \sigma_3, \sigma_4, \sigma_0\}$ is not created because it is either infeasible or useless. The labels created in level 3 store the following sequences: $\{\sigma_0, \sigma_3, \sigma_0\}$, $\{\sigma_0, \sigma_3, \sigma_4, \sigma_0\}$. Thus, the set of labels associated with each vertex contains $\Lambda^2 = \{\{\sigma_0, \sigma_2, \sigma_0\}\}$, $\Lambda^3 = \{\{\sigma_0, \sigma_2, \sigma_3, \sigma_0\}, \{\sigma_0, \sigma_3, \sigma_0\}\}$, and $\Lambda^4 = \{\{\sigma_0, \sigma_3, \sigma_4, \sigma_0\}\}$.

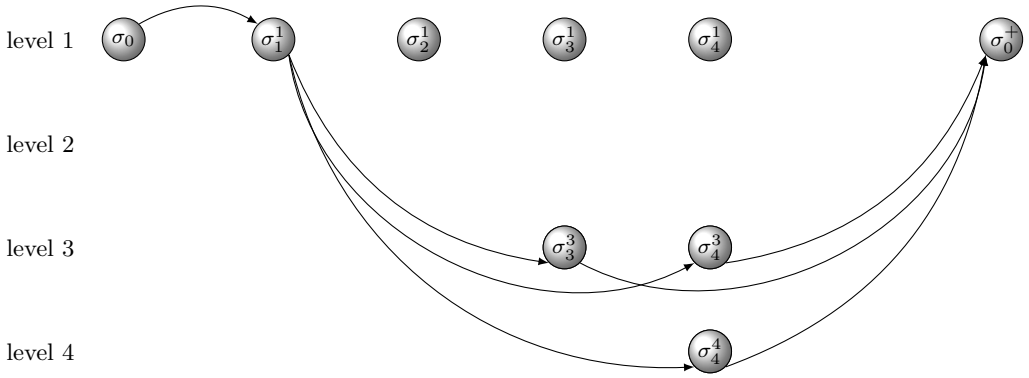

Figure 3.4: Phase 2: Extension of labels skipping defined subsequences of vertices. The label that stores the subsequence $\{\sigma_0, \sigma_1, \sigma_0\}$ is extended to levels 3 and 4. Labels which store the following subsequences are created: $\{\sigma_0, \sigma_1, \sigma_3, \sigma_0\}$, $\{\sigma_0, \sigma_1, \sigma_3, \sigma_4, \sigma_0\}$ and $\{\sigma_0, \sigma_1, \sigma_4, \sigma_0\}$ respectively.

an operator to solve the *Team Orienteering Problem* which selects m possible routes that satisfy the constraint on the route length. It is a polynomial algorithm based on a PERT/CPM technique. However, a route must be constituted of consecutive (adjacent) vertices.

3.4 Generating a Feasible Solution

The methodology proposed requires building quickly a feasible solution. This solution serves as a probe to determine the execution of *Selector* to optimality, and in some problems, it is used as a bound. There are several ways to construct it. One of them is using a greedy heuristic. For example, when solving the *Covering Tour Problem*, Gendreau et al. (1997) used the greedy heuristic designed by Balas and Ho (1980) to solve the *Set Covering Problem*, PRIMAL1. The heuristic gradually includes vertices into the solution according to a greedy criterion in order to minimise a given function. Three different functions are used in PRIMAL1. It builds two distinct solutions by applying these functions in a different order, and chooses the best one. Another possibility is a GRASP-based construction method as the one used by Campos et al. (2014) to solve the *Orienteering Problem*. Another option we considered was using a simplified version of *Selector*.

When deciding on the method to use to construct this solution, one has to find a good compromise between solution quality and computational effort to obtain it. We did some testing to observe the ratio $\frac{z}{z^*}$ (z represents a solution value and z^* the optimal value) obtained by our proposal. It was close enough to one many times and the computational effort was frugal. This is, then, the procedure implemented.

Algorithm 8 seeks a solution using the same graph and extension operations as the main body of *Selector*, howbeit, it only explores some of the possible paths to find the solution fast. Starting at σ_0 it tries to extend λ to $\sigma_i \ \ \forall i \in \{1, 2, \ldots, n-1\}$, and

from σ_i it will attempt to extend λ_i to σ_{i+k} $\forall k \in \{1, 2, \ldots, n-i-1\}$. Once sequence $\{\sigma_0, \sigma_i, \sigma_{i+k}\}$ is constructed, it continues, if possible, by iteratively visiting the adjacent successor vertex. For instance, it first attempts to construct sequence $\{\sigma_0, \sigma_1, \sigma_2\}$, and will attempt to continue by repeatedly extending to the immediate adjacent successor. Next, it constructs the sequence for the next value of k: $\{\sigma_0, \sigma_1, \sigma_3\}$, and again repeats the extension to the adjacent successors. Once all values of k are exhausted, it treats the next vertex σ_i in a similar way. The process finishes when all vertices σ_i have been treated. Every complete tour found is compared and the best one is kept, no labels are stored in order to execute it fast. The complexity of the search is $O(n^3)$.

Algorithm 8 : Search Feasible Solution

Input: giant tour σ, distance matrix D
Output: feasible tour of visited vertices T, cost value of tour $c(T)$

```
1: i ← 1
2: while ( λ_0 can be extended from σ_0 to σ_i ) do
3:    k ← i
4:    while ( λ_i can be extended from σ_i to σ_{k+1} ) do
5:       Extend(λ_{k+1}) {see Algorithm 6}
6:       k ← k + 1
7:    end while
8:    i ← i + 1
9: end while
```

3.5 Performance Improvements

To either maintain tractability or speed up the search, three mechanisms were introduced in *Selector*: (i) extending only a predefined number of labels; (ii) computing a lower/upper bound, which is compared to the incumbent best feasible solution; and (iii) performing bidirectional search in the graph.

3.5.1 Restricting the Number of Labels Extended

In computer science, one common way to attempt to maintain tractability is to use beam search so we adopted some of the techniques used by this type of process. The term *beam search* was coined by Raj Reddy, Carnegie Mellon University, 1976. Beam search is a restricted version of either a breadth-first search or a best-first search, and it is restricted in the sense that the amount of memory available for storing the set of states is limited, and in the sense that less-promising states can be pruned at any step in the search by problem-specific heuristics as explained by Zhang (1999). The set of most promising states is called the "beam". Beam search has the advantage of potentially reducing the time of a search. This potential advantage rests upon the accuracy and effectiveness of the heuristic rules used for pruning states, and having such rules can be somewhat difficult due to the expert knowledge required of the problem

domain. The main disadvantage of a beam search is that the search may result in a non-optimal solution. Therefore, this forward-pruning heuristic search sacrifices optimality for tractability. Despite this disadvantage, it is able to achieve a satisfactory level of solution quality, and has found success in areas such as machine learning and speech recognition (Zhang, 1999).

In *Selector*, we are not dealing with a shortage of memory, but we aim to restrict the number of labels extended in order to maintain tractability. The notion of *beam search* is applied in *Selector* in the sense that only promising labels are stored in the search queue (beam), and only a predefined number (beam width) of the labels stored is extended. A search process that extends the most promising label of a limited set. This way a reasonable search time is guaranteed since computations are restricted by a known value. Promising labels are identified by a bounding mechanism, as explained in section 3.5.2.

Therefore, this type of search requires that the algorithm knows the rules for pruning labels and the beam width. This technique was applied to the solution algorithm for the OP as presented in Chapter 6.

3.5.2 Computing a Lower/Upper Bound

Pruning non-promising states first requires identifying them. This is possible using a bound. In *Selector*, computing a bound serves two pruning purposes: avoid storing and avoid extending non-promising labels. The following function may be used to estimate the objective function value of a complete path

$$\mu(\lambda) = z(\lambda) + h(\lambda) \tag{3.1}$$

where

- $\lambda = [z, \sigma_i | r_1, r_2, \ldots, r_m]$ is a label that memorizes σ_i as the last visited vertex on the path represented.
- $z(\lambda)$ is the actual objective function value of the label (from the start vertex σ_0 to vertex σ_i).
- $h(\lambda)$ is the lower/upper bound computed on the objective function visiting only vertices in the subsequence $\varphi = \{\sigma_{i+1}, \ldots, \sigma_{n-1}\}$. This can be a lower bound of the cost to cover the remaining vertices or an upper bound of the profit collected from the vertices that lie ahead of σ_i.

Equation A.1 is inspired in the A* (pronounced “A star”) algorithm used in computer science for path finding and graph traversal. Noted for its performance and accuracy, it enjoys widespread use. Hart et al. (1968) of Stanford Research Institute (now SRI International) first described the algorithm, which was used by a prototype robot to improve its path planning. It is an extension of Dijkstra’s shortest path algorithm using heuristics to guide its search. At each iteration of its main loop, A* needs to determine which of its partial paths to expand into a longer path. It does so based

on an estimate of the cost (total weight) still to go to the goal vertex. Specifically, A* selects the path that minimises $f(n) = g(n) + h(n)$ where n is the last vertex on the path, $g(n)$ is the cost of the path from the start vertex to n, and $h(n)$ is the cost estimated by a problem-specific heuristic of the cheapest path from n to the goal.

Retaking the mechanics of *Selector*, to solve Equation A.1, we only need to find the value of bound $h(\lambda)$. Of course, the vertices we want to visit are those with a good $benefit/cost$ ratio. Then, the value of bound $h(\lambda)$ can be found by solving a *Fractional Knapsack Problem* (FKSP), i.e., the linear relaxation of a 0-1 Knapsack Problem. The FKSP can be solved in polynomial time, $O(n \log n)$ taking into account the sorting of the items considered. In order to solve this knapsack problem, it is necessary to first define the objects, their profit and their weight. An object is created for each vertex in φ. The profit p_i of an object is given by the number of vertices it covers or by the profit its visitation yields. The weight w_i of an object is computed as follows. Let $\Delta_i = \{w_{ji}\}_{j=0}^{i-1}$ be the set of weights of the in-going edges of σ_i, those that connect vertex σ_i with each of its predecessor vertices in the giant tour σ, and $\Delta'_i = \{w_{ik}\}_{k=i+1}^{n-1} \cup \{w_{i0}\}$ be the set of weights of the out-going edges of σ_i, those that connect vertex σ_i with each of its successor vertices in σ. Then, a lower bound of the travel cost (weight) of visiting vertex σ_i can be obtained by

$$w_i = \min \Delta_i + \min \Delta'_i \tag{3.2}$$

In case there is a tie between both sets—the predecessor vertex is the same as the successor—the global second-best edge weight is chosen.

The FKSP is solved with the classical greedy algorithm of Dantzig (1957). The ratio $profit/weight$, $\frac{p_i}{w_i}$, serves to determine the best vertices to visit. Order the vertices in non-increasing order according to their ratio $profit/weight$, and select them in sequence until the capacity of the knapsack is exceeded. In this case, the capacity of the knapsack may represent the number of clients still to cover or the length of the path that still remains out of the maximum allowed. Nonetheless, in the FKSP, we do not have to select all of the profit of a vertex, but rather can take any fraction of it. Thus, for the last vertex chosen, we may only include the fraction of profit that fits in the remaining capacity. This way the greedy algorithm never wastes any capacity, and as a result, it always yields an optimal solution for the FKSP as explained in Neapolitan and Naimipour (1998). Depending on the objective function considered, the value of bound $h(\lambda)$ is then the sum of the profit or weight values of the chosen vertices.

If the estimated objective function value (bound) of a given path is worse than the best-known solution, the search in that trajectory must be abandoned. Thus, function $\mu(\lambda)$ needs to be calculated at each step of the extension of a label in order to determine if the label is promising. It is worth noting that when solving the FKSP, computing the ratio $profit/weight$ at the level of the giant tour provides more accuracy than doing it at the level of the graph G that represents the given network of customers. However, it results in more computational effort since the metaheuristic builds a new giant tour at each iteration.

The value of function $\mu(\lambda)$ is stored in an added label field and it is applied again in the same manner when the label is retrieved for extension. This test is useful because the value of the best-known solution might have changed since the label was stored. This technique was applied to the solution algorithms for the CTP and the OP as explained in Chapters 4 and 6 respectively.

3.5.3 Bidirectional Search

Pohl (1971) was the first one to design and implement a bidirectional heuristic search algorithm. In this kind of algorithm, two search processes are performed simultaneously: one starts from σ_0 and considers σ in the order given, and the other one also starts at σ_0 but considers the sequence obtained by inverting σ. When the two search frontiers intersect, the algorithm can reconstruct a single path that extends from the start vertex through the frontier intersection to the goal vertex. In the implementation of this kind of search in *Selector*, the recurrence equation, dominance rule, label-extension procedures, feasibility rule and use of an upper bound are symmetrical to those previously presented.

The rationale behind using bidirectional search is the following. The dynamic programming algorithm presented generates a number of labels which rapidly increases with the size of the problem at hand. Every time a label λ is extended from σ_i, it generates as many other labels as the number of possible successors of σ_i. Therefore, in the worst case, the number of labels grows exponentially with the number of vertices in the path. Due to this exponential dependence on the number of steps, it is intuitive that generating shorter paths may yield a significant advantage in terms of number of labels considered. This is precisely the effect of bidirectional search with bounding, whose purpose is to limit the length of the paths considered to at most half of the length of the optimal path.

Henceforth, it is explained how key performance features of this type of search were implemented. The search process to perform at each iteration needs to be selected. Although iterating equally between both searches—forward and backward—would be the simplest method, it is not the most efficient. The best strategy is to identify the path (label) with the best objective function value so far. That is, during each iteration, concentrate the computational effort on the search having the best path cost.

A new problem arises during a bidirectional search, namely ensuring that the two search frontiers actually meet. For this reason, at each iteration, the occurrence of complete paths is monitored. Every time a non-dominated label is stored, the opposite search queue is revised to determine if there is a label that can be joined to it to form a complete solution. Thus, two labels of opposite direction meet when the last visited vertices match, and it is then tested if they can form a complete feasible solution. If the test yields positive, this solution is compared against the incumbent best known and the latter is updated if necessary. Both labels examined are kept regardless of the result of this test, since with another opposite half they may construct a better solution.

The termination policy is crucial to significantly reduce the computational time.

The monodirectional version proceeds until all labels are treated, but applying this criterion to the bidirectional version may produce a performance equal or worse to that of the monodirectional version for some instances.

The classical termination criterion of a bidirectional search on a minimisation problem compares the sum of the lowest actual forward cost plus the lowest actual backward cost with the cost of the best-known solution, and stops if the former is larger or equal to the latter. However, in our case, labels are arranged by estimated cost $\mu(\lambda)$, so the corresponding minimum actual cost is not available in $O(1)$ complexity. For this reason, we adapted the termination criterion as follows: if Equation A.3 is true and the two labels that comply with this equation constitute a feasible solution, then the algorithm stops the search.

$$\min_{\sigma_i \in S \setminus \{\sigma_0\}} \{\mu_i(\lambda)^{\text{forward}}\} + \min_{\sigma_i \in S \setminus \{\sigma_0\}} \{\mu_i(\lambda)^{\text{backward}}\} \geq bestKnownCost \tag{3.3}$$

If both labels form a feasible solution, the lower bound computed by Equation A.3 gives the minimum cost value that can be obtained for a complete feasible solution. If this minimum value is worse than the best-known solution, it is certain our solution value will not improve and we can halt the search. This technique was applied to the solution algorithm for the CTP as detailed in Chapter 4.

3.6 Multi-Vehicle *Selector*

The first idea that springs up when solving a multi-VRPOV with *Selector* is to pipe the output of *Selector* to the input of the *Split* operator, a select first–cluster second type of approach. It was certainly tried, but it only worked for very small values of the number of vertices that can be visited, $|V| \leq 25$. It was shown that an optimal selection of visits followed by an optimal partition does not necessarily lead to an optimal set of vehicle routes. Therefore, either the selection and the partition are done simultaneously, or a local search VRP heuristic first selects vertices and then partitions them into routes. *Split* had formerly been used inside such local search framework as a tool to evaluate the sequence of vertices selected by a previous stand-alone process, for instance, the heuristic method suggested by Hà et al. (2013). We chose to develop the novel idea of an *m-Selector* that chooses the best vertices to visit and at the same time solves how these vertices can be optimally arranged in routes.

Solving the partition problem requires inserting visits to the depot in order to evaluate the total cost of segmenting the given tour into feasible vehicle routes, as it is done by the *Split* algorithm. In the case of the multi-vehicle *Selector*, this would require the creation of even more labels than the ones produced by the single-vehicle version. Given the exponential dependence the growth in the number of labels has with the size of the path searched, the idea of adding more labels is not attractive. Instead, a way of doing these visits implicitly was sought, and it came in the form of using the *Split* algorithm as the cost function. Hence, instead of using the simple function of adding edge costs of the single-vehicle version, a modified version of *Split* was implemented

in order to compute the optimal cost of segmenting the selected vertices into feasible routes. As a matter of fact, the equation used by *Split* to compute the multi-route tour cost is very similar to the one used by the single-vehicle version of *Selector*. A modified version of *Split* is in the sense that the segmentation of the VRPOV tour has to occur step by step as vertices are added to it. This is to say, the trips represented in graph H of the *Split* algorithm are built as vertices are selected. In *Selector*, the cost of a label is the cost of assigning the selected vertices to a single route, while in *m-Selector* the cost is that of assigning them to several routes. In essence, the same notion. If when selecting a vertex, it is also evaluated how the selected set can be segmented into feasible routes, the resulting cost is truly considering both levels of decision-making at the same time. Since in *m-Selector* both decision processes—selecting and segmenting—are done by exact methods, the output is optimal. The assumption that all customers have the same demand made it easier to implement the idea. Embedding the *Split* algorithm in *Selector* in order to compute the cost of a label is more computationally demanding than calculating the simple addition of the cost of an edge, so ways to achieve good performance was also a matter that required careful consideration.

The structure of the *m-Selector* label remains basically the same, albeit the two vectors that the *Split* algorithm uses to keep track of the travel cost and of the predecessor vertex for each customer selected were incorporated. The dominance rule also adds a test in order to consider the vehicle capacity, and performance improved when more methods to build the initial giant tour were made available. All in all, merging both processes (*Selector* and *Split*) resulted in few changes to the original design of the algorithm, so the resulting implementation of *m-Selector* has a core which is very similar to the one of the original version. Chapter 5 explains the implementation in more detail.

3.7 Metaheuristic Framework

The second component of the solution approach presented in this study is the routing mechanism. This thesis work proposes to use the adaptive large neighborhood search (ALNS) algorithm. Chapter 2 presented some of the advantages of searching over a large neighborhood. ALNS is a local search framework which uses several competing destroy and repair sub-heuristics to modify the current solution, and it chooses amongst them using statistics gathered during the search. Even though it is a general heuristic, it can be as efficient as most specialized heuristics since its core may be formed by best-performing heuristics for the problem treated.

Figure A.1 shows how the proposed routing and clustering components interact when solving a minimisation problem. The algorithm starts with an initial giant tour σ^{init} which can be produced randomly or via a construction heuristic. Tour σ^{init} undergoes a *2-opt* local search process to improve its length rapidly and avoid a long random initial walk. Then, in each iteration, the algorithm considers a giant tour σ and has at its disposal several destruction and construction sub-heuristics to modify it. The algorithm first selects, with an adaptive probability, a destruction sub-heuristic which

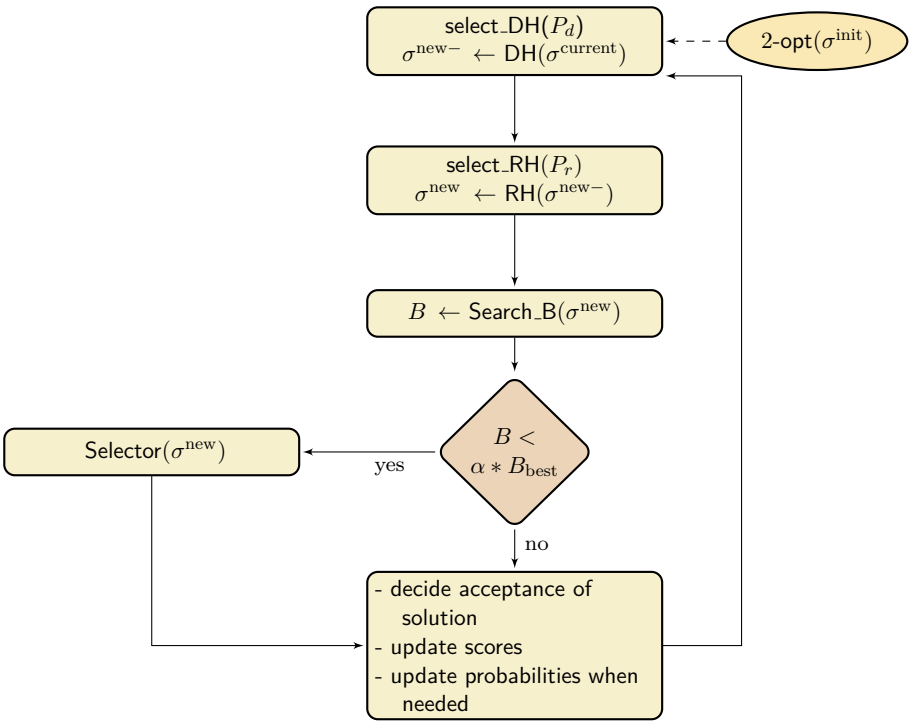

Figure 3.5: Routing and clustering components of the unified solution methodology.

removes some customers from the giant tour, and produces $\sigma^{\text{new}-}$. Next, again with an adaptive probability, it selects a repair sub-heuristic which inserts the customers back, but choosing better places so that the tour length is further improved. Over this new giant tour σ^{new}, it looks for a feasible solution (bound) using a reduced version of the *Selector* operator. To search for this solution efficiently, the operator analyses only a subset of the possibilities. Next, if the value of the feasible solution improves, *Selector* is executed, this time to optimality, with the expectation of obtaining a better VRPOV solution from σ^{new}. To avoid making this step very strict, the value of the best-known bound is multiplied by a convenient α value. The algorithm also contains a scheme to avoid stagnation of the search process. In addition, the adaptive layer requires updating the scores of the sub-heuristics used in the iteration, and, every update period, the selecting probability of every sub-heuristic. The process repeats until the termination criterion is met.

Improvements on the travel cost of the giant tour do not necessarily lead to a better objective function value on the tour computed by *Selector*. Furthermore, an optimal giant tour does not necessarily yield an optimal VRPOV solution value. However, in the long run, the VRPOV tour does benefit from improvements on the length of the giant tour, since the shorter the tour, the more likely additional vertices can be put into it. An important consequence of this independence is that execution of *Selector* to optimality at each iteration is of low benefit. The computed bound can serve as a probe to determine if the complete process is worth executing. This derives in important time savings.

Parameter	Meaning
γ	number of vertices removed in each ALNS iteration (instance size dependent)
ς	period size for updating probabilities in number of ALNS iterations
τ	reaction factor that controls the rate of change of the weight adjustment
δ	avoids determinism in the SRH
ρ	avoids determinism in the WRH
κ_1	score for finding a new global best solution
κ_2	score for finding a new solution that is better than the current one
κ_3	score for finding a new non-improving solution that is accepted
β	cooling factor used by simulated annealing
ϵ	fixes the upper limit of vertices removed at each iteration

Table 3.1: The ALNS parameters were tuned using the R-implemented **irace** package developed by López-Ibáñez et al. (2011).

The main elements of the ALNS implementation are explained following the three backbones explained by Pisinger and Ropke (2010): (i) the destroy heuristics (Section 3.7.1), (ii) the repair heuristics (Section 3.7.2), and (iii) the metaheuristic that defines the criteria to accept a new solution and guides the search (Section 3.7.3),. Three destroy and three repair heuristics were implemented. In the following, lower-case Greek letters indicate the user-controlled parameters documented in Table A.1.

The ALNS parameters were firstly tuned using a *ceteris paribus* approach based on sets of three or four values for each parameter. Later on, a software package that automatically configures optimisation algorithms was used and better results were obtained. It is the R-implemented **irace** package developed by López-Ibáñez et al. (2011), Iterated Racing for Automatic Algorithm Configuration. It implements the iterated racing procedure, an extension of Iterated F-race.

3.7.1 Destroy Sub-heuristics

Shaw Removal Heuristic (SRH). Originally proposed by Shaw (1997), its general idea is to remove vertices that exhibit similitude, characteristic computed by a *relatedness measure* $R(i, j)$. It is expected to be reasonably easy to shuffle similar vertices around and thereby create new, perhaps better, solutions. We measure the similarity between two vertices by $R(i, j) = d_{ij}$, where d_{ij} is the Euclidian distance between vertices σ_i and σ_j. The lower $R(i, j)$ is, the more related the two vertices are. This relatedness measure is used to remove vertices in the same way as described by Shaw (1998). Given a solution σ, the first vertex σ_i to be removed is selected randomly, and it is put into set F of removed vertices. Thereafter, in every iteration the algorithm randomly selects a vertex σ_i from F, computes $R(i, j)$ between it and all of the vertices $\sigma_j \in \sigma$, and then chooses a new vertex $\sigma_i \in \sigma$ to be inserted in F. This process repeats

while $|F| < \gamma$. In more detail, choosing the vertex to remove requires building a list L ordered by increasing relatedness values. A random number y is chosen in the interval [0, 1] and vertex $i = \lfloor y^{\delta}|L| \rfloor$ is removed from σ. Parameter $\delta \geq 1$ controls the amount of randomization. In order to avoid the sorting required at each iteration, a nearest vertex matrix is pre-computed and kept at hand. The complexity of the SRH is $O(n^2)$.

Worst Removal Heuristic (WRH). Ropke and Pisinger (2006) propose a heuristic that randomly removes vertices with a high cost in the current solution σ and tries to insert them in better positions. Let $cost(i, \sigma) = f(\sigma) - f_{-i}(\sigma)$ be the cost associated with vertex σ_i in the current solution σ, where $f_{-i}(\sigma)$ is the solution cost without vertex σ_i. vertices are first sorted according to $cost(i, \sigma)$ and then one is randomly chosen to be removed. The process iterates recalculating the costs, $cost(i, \sigma)$, until it has removed the indicated number of vertices. The removal, though random, is user-controlled by parameter ρ. The complexity of the WRH is $O(n^2)$.

Random Removal Heuristic (RRH). This procedure simply selects γ vertices at random and removes them from the current solution σ. Though it tends to generate a poor set of removed members, it is useful to diversify the search. The complexity of the RRH is $O(n)$.

How Many to Remove. The number of vertices removed, γ, from the current solution σ is key to the ALNS performance. When few elements are removed, the heuristic has a higher probability of being trapped in one suboptimal area of the search space. On the other hand, when too many are removed, it is almost like starting from scratch and the insertion heuristics cannot build a good solution from such situation. In addition, the larger the number removed, the larger the execution time of both insertion and removing heuristics. Parameter γ is chosen randomly between a lower and an upper limit. The lower limit is fixed at a value given according to the number of vertices in σ, while the upper limit is fine-tuned with parameter ϵ. This parameter indicates the maximum percentage of elements removed from the complete solution size. Hence, the algorithm works with a randomized degree of destruction in the interval $[0.3 \times |V|,\ \epsilon \times |V|]$.

3.7.2 Repair Sub-heuristics

Best Greedy Heuristic (BGH). This simple construction heuristic performs at most γ iterations as it inserts one vertex into solution σ in each iteration. The minimum cost position value is computed for all vertices waiting insertion—set F—and the one with the minimum global cost position is chosen. This process is repeated until $F = \emptyset$. The complexity of the BGH is $O(n^2)$.

First Greedy Heuristic (FGH). This heuristic works similarly to the previous one. However, instead of inserting the vertex having the minimum global cost position, it inserts the one sitting in the first position. That is to say, it respects the order of

the vertices in F. After the first vertex has been inserted, the minimum cost position for each is recalculated and the process repeats until all vertices in set F have been inserted.

Ropke and Pisinger (2006) add a noise term to the objective function during the insertion phase of the BGH and regret-k heuristics in order to randomize them and avoid always making the move that seems best locally. In our implementation, the FGH is used mainly to introduce this noise into the insertion process as done by Ribeiro and Laporte (2012). This heuristic obviously runs faster than the BGH.

Regret Heuristic (RKH). This heuristic tries to improve the myopic behaviour of the greedy heuristics by incorporating a kind of look ahead information when selecting the vertex to insert, as done by Ropke and Pisinger (2006) and Pisinger and Ropke (2007). Let Δf_i^1 denote the change in tour length incurred by inserting vertex σ_i at its minimum cost position, and Δf_i^2 denote the change by inserting it at its second best position. The regret value is defined as $c_i^* = \Delta f_i^2 - \Delta f_i^1$. In each iteration, the heuristic inserts the vertex σ_i that maximizes the regret value c_i^* at its minimum cost position. Ties are broken by selecting the vertex with lowest cost insertion. Informally speaking, it chooses the insertion that we will regret the most if it is not done now. This is a time-consuming operator but unnecessary computations were avoided when computing Δf_i^n. The complexity of the RKH is $O(n^3)$.

Choosing a Destroy-Repair Heuristic Pair. In order to select a heuristic, weights are assigned to them and a *roulette wheel selection principle* is applied by the adaptive layer. Let $D = \{d_i | i = 1, \ldots, 3\}$ be the set of destroy sub-heuristics, and $R = \{r_i | i = 1, \ldots, 3\}$ the set of repair sub-heuristics. The weights of the heuristics are denoted $w(d_i)$ and $w(r_i)$ respectively, so that the probabilities to select one are

$$p(d_i) = \frac{w(d_i)}{\sum_{j=1}^{3} w(d_j)}, \quad p(r_i) = \frac{w(r_i)}{\sum_{j=1}^{3} w(r_j)} \tag{3.4}$$

The removal heuristic is selected independently of the insertion heuristic and vice versa. Initially, all heuristics are equally likely to be chosen, e.g., $w(d_i) = 1 \quad \forall d_i \in D$.

Adaptive Weight Adjustment. Adjusting the weights of the heuristics enables to increase the probability that successful heuristics are more frequently used than less successful ones. The weights could be updated at every iteration or after some given number of iterations. In this version, the second one is used for efficiency reasons. Hence, the total number of ALNS iterations is divided into a number of update periods whose size, ς, is experimentally determined. These update periods are also named segments. Let h denote a destroy or repair heuristic. To allow the weight adjustment, a score $s(h)$ is memorized for every heuristic, and it is updated at each iteration by a quantity equal to parameters κ_k, where $k \in \{1, 2, 3\}$, when it identifies new solutions, refer to Table A.1. For reasonable adjustments the inequality $\kappa_1 > \kappa_2 > \kappa_3$ is ensured. Then, at the end of each update period, these recorded scores are used to calculate

new weights and probabilities thereof. In addition, all weights are reset to zero at the beginning of each segment. Since two heuristics are applied in each iteration, the scores for both are updated by the same amount.

New weights are computed as follows. Let $w(h_i)_j$ be the weight of heuristic i in update period j. After period j finishes, the new weight to be used in period $j+1$ for heuristic h_i is given by

$$w(h_i)_{j+1} = \begin{cases} w(h_i)_j(1-\tau) + \tau\dfrac{s(h_i)}{u(h_i)}, & \text{if } u(h_i) > 0 \\ w(h_i)_j(1-\tau), & \text{if } u(h_i) = 0 \end{cases} \tag{3.5}$$

where $s(h_i)$ is the score of heuristic h_i obtained during the last period and $u(h_i)$ is the number of times heuristic h_i was used during this same period. τ is known as the reaction factor, and it controls how quickly the weight adjustment mechanism reacts to changes in the effectiveness of the heuristics. If $\tau = 0$, the weight adjustment is ignored and the initial weights prevail. If $\tau = 1$, the score obtained in the last segment decides the weight to be used in the next one.

This implementation keeps track of visited solutions using a hash table. A hash key is assigned to each solution and this key is stored in the table.

3.7.3 Simulated Annealing Guides the Search

Simulated annealing (SA) is the outer metaheuristic that guides the search. SA applied to optimisation problems emerges from the work of Kirkpatrick et al. (1983). It had a major impact on the field of heuristic search for its simplicity and efficiency in solving combinatorial optimisation problems as presented in Talbi (2009). SA is a stochastic algorithm that enables, under some conditions, the degradation of a solution with the objective to escape from local optima and so to delay the convergence. Then, SA implies to not only accept solutions that are better than the current solution, but rather, on occasion, accept solutions that are worse than the current one.

SA is a memoryless algorithm in the sense that it does not use any information gathered during the search, and it is based on the principles of statistical mechanics. From an initial solution, a random neighbor is generated at each iteration. Moves that improve the objective function are always accepted. Otherwise, the neighbor is accepted with a calculated probability that depends on the control parameter called temperature, t, and on the amount of degradation of the objective function (energy), $\Delta E = f(\sigma') - f(\sigma)$. The computed value ΔE represents the difference in the objective value between the current solution σ, and the generated neighbouring solution σ' (minimisation process). As the algorithm progresses, the probability of accepting a non-improving solution decreases. This probability follows, in general, the Boltzmann distribution.

$$P(\Delta E, t) = e^{-\dfrac{f(\sigma') - f(\sigma)}{t}} \tag{3.6}$$

Hence, the probability of accepting a non-improving solution is proportional to

parameter t, and inversely proportional to the change of the objective function, ΔE. The temperature starts at a preset level, which Ropke and Pisinger (2006) explain is instance-dependent, and is decreased in every iteration according to a cooling schedule such that few solutions are accepted towards the end of the search, $t = t \cdot \beta$, where $0 < \beta < 1$ is the cooling factor. In this implementation, the initial value of t is set to the length of the improved initial giant tour divided by an instance-dependent factor, and t decreases in each iteration. This is, in this SA scheme, only one solution is explored at each value of temperature.

3.8 Conclusions

The solution methodology proposed emulates a route first–cluster second constructive heuristic, so it is composed of a clustering method and a routing method. The clustering method proposed is the *Selector* operator which is a dynamic programming-based algorithm aimed to solve VRPOV. The algorithm is an adaptation of the one developed by Desrochers (1988) in the context of the RCSPP. It is a label-correcting, reaching algorithm. *Selector* is integrated into an ALNS metaheuristic, the routing method, which fulfills the task of assigning the visitation order of the n given customers. From this constructed sequence, *Selector* optimally retrieves the ones to visit. The problem of selecting the visited customers is formulated as a RCESPP on an auxiliary directed acyclic graph where the side restrictions of the problem considered act as the constraining resource. This auxiliary graph represents the topological order of the n customers contained in the giant tour treated.

The basic principle of the algorithm is to associate with each partial path from the depot vertex σ_0 to a vertex $\sigma_i \in H$ a label representing the cost of the path and its consumption of resources, and to eliminate useless labels with the aid of dominance rules as the search progresses. Labels are iteratively extended in all feasible ways until no more labels can be created, while the incumbent best path is updated throughout the search process. The extension of a label to a visited vertex corresponds to (1) appending an additional arc $(i, i + k)$ to a path from σ_0 to σ_i, obtaining a feasible path from σ_0 to σ_{i+k}; and (2) updating the cost of the path and its consumptions of resources. Since the number of labels produced during the search has an exponential dependence on the size of the problem at hand, diverse mechanisms aimed to limit the proliferation of labels were added to the basic algorithm. Furthermore, a multi-vehicle version which co-works with the *Split* operator was also proposed.

Chapter 4

Solving the Covering Tour Problem with *Selector*

Contents

4.1 Introduction

Section 2.2.3 introduced the family of problems that include the notion of cover, provided an insight into their different applications, and described some of the most relevant approaches that have been used to solve problems of this subclass. This chapter focuses on one of them—the *Covering Tour Problem* (CTP)—and describes a solution procedure based on the *Selector* operator that allows to convert a giant tour into an optimal CTP solution. The method is competitive as shown by the quality of results evaluated using the output of a state-of-the-art exact algorithm, the one of Gendreau et al. (1997).

The CTP emerged as a problem where features other than merely minimising route length are considered. Current and Schilling (1994) presented practical situations where a vehicle travels through a network making service stops and natural aims are to:

(1) maximise cover, interpreted as the demand within a defined distance from a service stop.

(2) maximise access to the service by minimising the sum of the distances to the nearest service stop.

(3) minimise route length, might also be considered as cost, or travel time.

Considering items (1) and (3), a *Shortest Path Problem* that seeks maximal covering appears. Current and Schilling (1994) formulated it as the explicit multi-objective problem named *Maximal Covering Tour Problem*. Considering items (2) and (3), a *Median Shortest Path Problem* is defined which they formulated as the *Median Tour Problem*, also an explicit multi-objective problem. Boffey et al. (1995) argue that these problems may be regarded as an implicit multi-objective problem corresponding to the surrogate problem of minimising route length subject to a fixed percentage of the relevant demand being covered. Gendreau et al. (1997) introduced the CTP with this implicit multi-objective approach.

The CTP defines two types of vertices: the covering ones (visited service stops) and the ones to cover (implicitly visited demand points). The aim is to identify a lowest-cost elementary cycle over a subset of the covering vertices in such a way that every element not of this type is covered. In this case, a vertex is considered covered when it lies within a prespecified radius from at least one covering vertex. A CTP solution has two components. The first component corresponds to a *Set Covering Problem* (SCP) and selects the points that are visited. The second component corresponds to a TSP and provides the order in which the selected points are visited.

The remainder of the chapter is organised as follows. Section 4.2 makes a formal presentation of the problem, discusses practical applications of it, and provides a view of the methods previously proposed to solve the CTP. The solution approach used to assess the quality of our results is explained in more detail. Sections 4.3-4.5 present the solution method suggested in this thesis. Section 4.6 documents the results obtained. Finally, Section 4.7 discusses the contribution of this work.

4.2 The Covering Tour Problem

4.2.1 Formal Definition of the CTP

Although integer programming—field that studies ways to solve optimisation problems defined with integer variables—is used for solving exactly a variety of combinatorial optimisation problems, metaheuristic approaches tend to directly exploit the combinatorial nature of a problem rather than its integer programming formulation. For this reason, the mathematical models of the problems treated are not presented in this manuscript.

The CTP is a generalisation of the *Travelling Salesman Problem* (TSP) and it can be formally described as follows. Let $G = (N, E)$ be an undirected complete graph, where $N = V \cup W$ represents the vertex set and the edge set is given by

$E = \{(v_i, v_j) | v_i, v_j \in N, i < j\}$. Set $V = \{v_0, v_1, v_2, \ldots, v_{n-1}\}$ is the subset of n vertices that *can* be visited at most once, while W is the subset of vertices that must be *covered* but cannot be visited. Set V includes a subset T of vertices that *must* be visited, $|T| \geq 1$. Vertex v_0 represents the depot. Let d_{ij} be the distance associated with each edge $(i, j) \in E$, and $D = (d_{ij})$ the distance matrix defined on E that satisfies the triangle inequality. A vertex $w_i \in W$ is covered if there exists at least one vertex $v_j \in V$ in the cycle for which $d_{ij} \leqslant c$, where c is known as the *covering distance*. Each vertex in V covers a subset of W, so for each $w_i \in W$, a subset of vertices of V that can cover it is given. A solution is a minimum-length Hamiltonian cycle on a subset of V such that for all the vertices in W at least one vertex of its covering set is visited. Figure 4.1 shows a feasible CTP tour for an instance where $|V| = 11$, and $|W| = 16$, and exemplifies how the subset $\{v_A, v_B\} \in V$ covers vertices $\{w_1, w_2\}$. The square represents the depot.

In this study, the cardinality of set T was kept at one since higher values tend to produce easier problems because less labels are produced. The only member of T is the depot and this vertex covers no vertex $w_i \in W$. The CTP is NP-hard as it reduces to a TSP when $c = 0$ and every vertex of W coincides with a vertex of V, $V = W$.

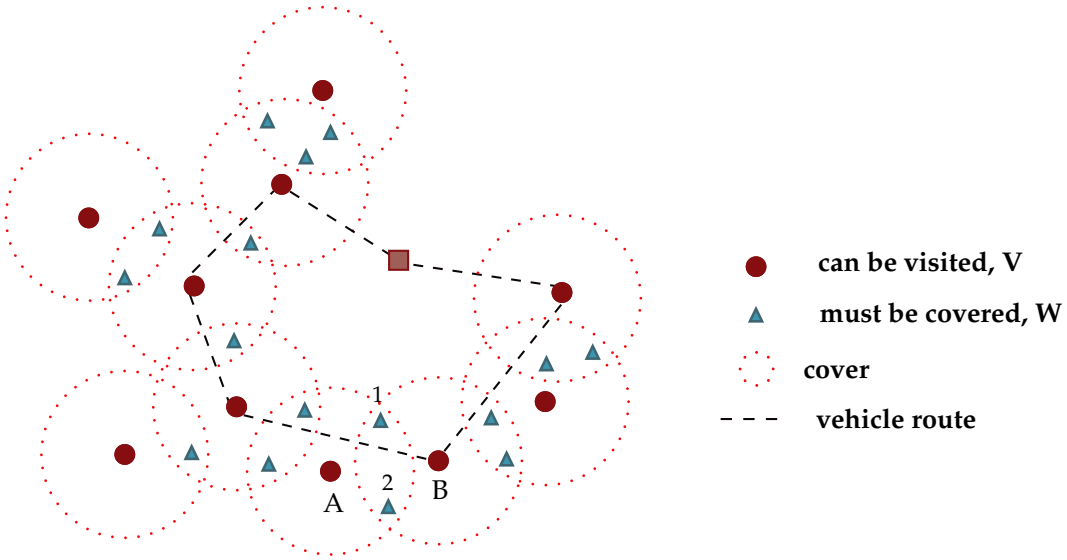

Figure 4.1: Example of CTP tour.

4.2.2 Applications Reported in the Literature

The CTP finds application in situations in which one needs to design a tour in a network where there are points that can be visited and points that cannot be visited explicitly, but can easily be reached from the points that can be in the tour. Such situation arises in distribution network design such as the bimodal distribution network explained by Current and Schilling (1989), where the customers (cities) on the tour are served by

air, and the smaller cities not in the tour are served (covered) by trucks which originate at their nearest city on the air route.

Furthermore, the notion of having points that cannot be visited explicitly, yet demand service, finds wide application in humanitarian logistics and health care delivery where service crews may face unavailable roads or scarce resources. For instance, in many developing countries, the ministries of health face the problem of providing a number of primary health care facilities sufficient enough to be geographically accessible, yet few enough to be properly stocked and staffed. Using the CTP, Hodgson et al. (1998) successfully modelled the use of mobile facilities to solve the challenge in Suhum District, Ghana. The model minimises a mobile facility's travel while serving all population centres within range of a feasible stop.

Doerner et al. (2007) published a multicriteria treatment of the CTP for planning mobile health care facilities. The goal of their problem is to find a single tour over a subset of vertices considering three objectives: (1) improve the economic efficiency of the tour (2) minimise the average distances that the unvisited people travel to reach their nearest tour stops and (3) minimise the percentage of population unable to reach a tour stop within a prespecified maximum travel time. They developed both a multi-objective ant colony optimisation method and multi-objective genetic algorithms to solve it, and applied their approach to the Thiès region in Senegal. Doerner and Hartl (2008) discussed the CTP in the context of its application to health care logistics and disaster relief with a focus on the Austrian situation.

Another application arises in the problem of determining the appropriate location of post boxes in urban areas such that the distance travelled by the collecting vehicle through all post boxes is minimal and every user is located within a reasonable travel distance from a post box (Labbé and Laporte, 1986). ReVelle and Laporte (1993) used the CTP model to plan the stops of a circus so that it is always accessible by unvisited populations. They presented it as the *Travelling Circus Problem* and any unvisited location may generate a penalty.

4.2.3 Solution Approaches Reported in the Literature

Despite its practical importance, few publications exist for the CTP. In the literature, only one exact method, a branch-and-cut algorithm by Gendreau et al. (1997), has been presented so far to solve the CTP. Jozefowiez et al. (2007) retook the implicit multi-objective nature of the CTP and proposed a bi-objective treatment of the problem: minimisation of the tour length and minimisation of the covering distance, and developed a two-phase cooperative strategy that combines a multi-objective evolutionary algorithm with the branch-and-cut algorithm of Gendreau et al. (1997). To assess the efficiency of their hybrid metaheuristic, they also developed an $\epsilon-$constraint approach to determine optimal Pareto sets. Furthermore, they applied their method to the real-world case of the Suhum district in Ghana.

Besides their exact method, Gendreau et al. (1997) also presented a heuristic solution approach. It is based upon the combination of approximate solution procedures for the *Set Covering Problem* (PRIMAL1 due to Balas and Ho (1980)) and the TSP

(GENIUS of Gendreau et al. (1992)). They applied their method to randomly-built instances where $|V| \leq 100$ and $|W| \leq 500$, and compared their results against the solutions computed by their exact method. Their heuristic solved most instances tried within 3% of optimality. Other heuristic algorithms have also been studied. Motta et al. (2001) proposed a GRASP metaheuristic to solve a generalized version of the CTP where the tour may also include vertices of set W. They also presented some reduction rules to meaningfully diminish the size of the generalized CTP instances treated. They studied randomly-generated data sets in which $|V \cup W| \leq 300$. Baldacci et al. (2005) treated the CTP with a two-commodity flow formulation and developed three scatter-search heuristic algorithms. They treat randomly-produced instances with $|V| \leq 100$ and $|W| \leq 500$. For $|V| > 75$ optimality gaps can be very large. Although the instances they solved are different from the ones used by Gendreau et al. (1997), they claim their heuristic is slower than the one of Gendreau et al. (1997).

4.2.4 Solution Approach Used as Reference

Gendreau et al. (1997) formulated the CTP as an integer linear program, investigated polyhedral properties of several classes of constraints and developed a branch-and-cut algorithm. In their algorithm, a linear program containing a subset of valid constraints is solved at a generic vertex of the enumeration tree. Violated constraints are searched and some of them are introduced into the current program which is then reoptimised. The process repeats until a feasible or dominated solution is found, or until it becomes more promising to branch on a fractional variable. In order to save time and memory, ineffective constraints are periodically deleted from the program. The method was applied to solve instances with $|V| \leq 100$ and $|W| \leq 500$. An instance was deemed successful if it could be solved within three hours. Hodgson et al. (1998) applied this exact method to a real problem in rural health care delivery: the routing of a mobile medical facility. This problem is described in Section 4.2.2.

4.3 Application of *Selector* to Solve the CTP

Chapter 3 provides an explanation of the principles of design and operation of *Selector*. However, any unified method must ultimately account for the objective and constraints of the specific problem to solve, so components of the method need to be tailored in order to fit the problem's needs. This section explains how the operator is adjusted and its principles are put into practice to solve the CTP. At a later section, the practical proposal to enhance its performance is presented.

The basic principle of the label-correcting reaching algorithm designed is to associate a label with each elementary partial path that goes from the depot vertex σ_0 to vertex σ_i. The label memorizes the value of the objective—length of the tour—and maintains accounting of the used resources, in this case, the vertices that need to be covered. Useless labels are eliminated as the search progresses with the aid of dominance rules. A label is extended toward every successor vertex as long as feasible labels can be generated. Therefore, in order to apply the explained algorithm, we first need

to define the structure of the label, the dominance criterion, and the considerations made when extending a label such as redundancy checking, feasibility testing and the updating of a label.

4.3.1 Label Definition

A label $\lambda = [\zeta, \nu, \omega]$ is defined by:

i. ζ, the travel cost of the partial elementary tour from vertex σ_0 to vertex σ_ν. This corresponds to the sum of the weights of the edges included in the label.

ii. ν, the last vertex visited in the partial tour.

iii. ω, a vector indexed by the vertices of W where $\omega[i]$ indicates either that $w_i \in W$ is already covered or the number of vertices of V that can still be visited to cover it.

Other information could be stored in the label to accelerate the search, but these three pieces of data are the only ones really needed. For example, it is useful to know in $O(1)$ complexity how many vertices are already covered by the label at hand or the value of a previously computed bound. However, one must keep in mind that the more complex the label is, the less efficient the algorithm becomes.

The labels generated could be stored in a single search queue. However, to attain search efficiency, every vertex $\sigma_i \in S \setminus \{\sigma_0\}$ maintains a separate label queue Λ^i. Each label is stored in the queue of its last visited vertex, and the queues are arranged by decreasing order of cost. The label chosen for extension is the one that documents the lowest path cost. It is a best-first search.

4.3.2 Dominance Test

This test is performed before storing a label in order to prune the redundant ones: at least one other label with a similar trajectory exists that offers a better or equal travel cost value and a less constrained or equal resource consumption. A label $\lambda_1 = [\zeta_1, \nu_1, \omega_1]$ dominates a label $\lambda_2 = [\zeta_2, \nu_2, \omega_2]$ with $\lambda_1 \neq \lambda_2$ if and only if:

i. $\nu_1 = \nu_2$

ii. $\zeta_1 \leq \zeta_2$

iii. $\{w_i : \omega_2[i] \text{ indicates } w_i \text{ is covered}\} \subseteq \{w_i : \omega_1[i] \text{ indicates } w_i \text{ is covered}\}$

Two labels are comparable only when arriving at the same vertex so that similar past and future trajectories are compared. A label dominates when its cost is lower or equal, and its set of covered vertices is a superset of the second one. To perform test (iii), the vector ω maintained in each label is used to compare the vertices covered by both labels tested. In $O(1)$ complexity the flag that indicates if any vertex $w_i \in W$ is covered can be known.

4.3.3 Extension of a Label

In the *Selector* algorithm, the extension of a label $\lambda = [\zeta, \nu, \omega]$ from vertex σ_ν to a successor vertex σ_k with $k > \nu$ implies that one of two possible operations is performed: visit vertex σ_k or skip vertex σ_k. However, at every extension, feasibility must be maintained and redundant vertices must not be considered. Therefore, when no vertex $w_i \in W$, is left uncovered the label λ is extended by skipping vertex σ_k. Whereas, when this restriction is not met, extension is done by visiting σ_k. The details of these operations are as follows.

Redundancy Checking. When extending the label by visiting σ_k, we need to ensure that σ_k is useful, i.e., that all the vertices w_i it covers are not already covered. For computational efficiency, a matrix relating the coverage of the vertices $w_i \in W$ by the vertices $v_j \in V$ is precomputed and kept at hand, so that information needed for decision-making is readily available.

Feasibility Checking. Let $\overline{\Omega}$ denote the subset of vertices of W that are not covered by the subpath represented by label λ. When extending the label by skipping vertex σ_k, we must be certain that each $w_i \in \overline{\Omega}$ is not affected by this decision. This is to say that, for each $w_i \in \overline{\Omega}$ there still remain vertices ahead of σ_k which can be visited to cover it. The number of such vertices is kept through field ω. Thus, for each $w_i \in \overline{\Omega}$, feasible labels yield $\omega[i] > 0$.

Look-Ahead Mechanism. This mechanism allows to further reduce the number of labels created. Let $\overline{\Omega^k}$ be the set of vertices w_i that remain to be covered in the set of σ_k. If when extending a label it is found that a vertex σ_k must be visited in the future because it is the only one remaining that can cover a set $\Gamma \subseteq \overline{\Omega^k}$, then all the vertices $w_i \in \overline{\Omega^k}$, but one, are marked as covered. One vertex $w_i \in \Gamma$ must remain uncovered, so that when vertex σ_k is reached, it proves useful. Looking ahead makes some label extensions unnecessary because redundant vertices are detected earlier and, therefore, are skipped. As a result, less labels are created. In addition, when the dominance test is applied, the sets compared are already taking into consideration vertices $w_i \in \overline{\Omega^k}$ as covered, and this also helps to delete useless labels earlier.

Figure 4.2 provides an example where $i < j < k$. The label is extended to vertex σ_i, and it is found that for set $\Gamma = \{w_5, w_7\}$ the only vertex that remains ahead which can cover it is vertex σ_k. Then, the vertices $\{w_2, w_4, w_7, w_9\} \in \overline{\Omega^k}$ are marked as covered. Vertex w_5 remains as uncovered, so that when vertex σ_k is reached, it is found useful. This is, vertex w_5 remains as a sentinel to ensure vertex σ_k is included. The former implies that set $\overline{\Omega^j}$ is now marked as covered, making vertex σ_j redundant. Therefore, when σ_j is reached no labels are produced.

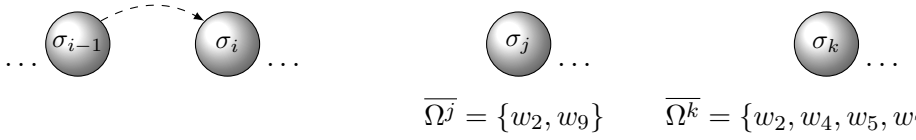

Figure 4.2: Portion of a giant tour used to exemplify the look-ahead mechanism.

Algorithm for Extending a Label. Algorithm 9 shows the specific implementation of Algorithm 6 for the CTP. It depicts how a label is iteratively extended to the adjacent successor vertex to maintain feasibility and efficiency. Label $\lambda_{current}$ stores the last vertex visited in the path, and label λ_j memorizes the vertex to which the label is extended, σ_j. Set Ω contains the vertices $w_i \in W$ that are already covered by the path, and set $\overline{\Omega^j}$ contains the vertices that remain to be covered in the set of vertex σ_j. Set Λ^j contains the non-dominated labels stored at σ_j. At every step of the label extension, the following conditions are verified:

i. vertex to be included is not redundant (line 3)

ii. $\zeta(\lambda_j) < UB_{\text{best}}$ (line 6)

iii. label is non-dominated (line 9)

First, it must be ensured that the vertex is useful. Any vertex that turns out to be redundant is simply skipped and the construction of the path continues, no labels are kept for skipped vertices. It is important to note that in this test the look-ahead mechanism is also applied and the resources (vector ω) are updated. If the vertex is worth visiting, the label cost and label index ν are updated. Next, test (ii) guarantees that the search in that trajectory is abandoned if the cost of the path turns out to be worse than the cost of the incumbent best-known solution. This acts as a bounding mechanism that enables to further control the proliferation of labels. If test (ii) is true, it is possible that the extension yields a complete solution. In such case, the incumbent best-known upper bound is updated in order to improve the limits for the creation of labels, and the extension in that direction stops. Otherwise (no complete solution exists), test (iii) makes sure only useful labels are kept.

4.4 Performance Improvements

A version of *Selector* which uses bidirectional search with bounding was implemented in order to study the performance improvement that could be attained. In this search, the recurrence equation, dominance rule, label-extension procedures, feasibility and redundancy checking and use of an upper bound are symmetrical to those previously presented, albeit the look-ahead mechanism is not used.

The bidirectional search, however, includes the computation of a lower bound on the cost of the tour, which aims at reducing label proliferation. Having a good lower bound allows to identify non-promising labels that can be pruned. Then, at each step

Algorithm 9 : Extend(λ) in the CTP

Input: label to be extended $\lambda = [\zeta, \nu, \omega]$, UB_{best}, $|\Omega|$
Output: labels derived from $\lambda = [\zeta, \nu, \omega]$
{only non-dominated labels that can later be extended skipping are kept}

```
1: λ_current ← λ
2: for ( j = ν + 1 to n − 1 ) do
3:   if ( vertex σ_j is not redundant ) then
4:     |Ω| ← |Ω ∪ Ω̄^j|
5:     ζ(λ_j) ← ζ(λ_current) + cost(σ_current, σ_j)
6:     if ( ζ(λ_j) < UB_best ) then
7:       if ( |Ω| ≠ |W| ) then
8:         λ_current ← λ_j
9:         if ( λ_j non-dominated ) then
10:          Λ^j ← Λ^j ∪ {λ_j}
11:        end if
12:      else
13:        UB_best ← ζ(λ_j)
14:        return {complete solution has been built}
15:      end if
16:    else
17:      return {cost of path being explored is worse than best-known cost}
18:    end if
19:  end if
20: end for
```

of the label extension, a lower bound on the cost of the complete tour represented by the label needs to be computed. The obtained bound can be compared against the incumbent best-known cost in order to determine if the label created by visiting that vertex is worth storing for further extension. One way to estimate the cost of the path that remains to be searched is by solving a *Fractional Knapsack Problem* (FKSP). Both mechanisms—computation of a bound and bidirectional search—are thoroughly explained in Chapter 3. Hereafter, the explanation indicates only the specifics of the computation of the lower bound. To estimate the cost of a complete path, Equation 4.1 can be used.

$$\mu(\lambda) = \zeta + h(\lambda) \tag{4.1}$$

where

- $\lambda = [\zeta, i, \omega]$ is a label that memorizes σ_i as the last visited vertex
- $\overline{\Omega}$ is a set of vertices that remain to be covered
- $h(\lambda)$ is a lower bound computed on the cost incurred to cover the vertices $w_j \in \overline{\Omega}$ visiting only vertices in the subsequence $\varphi = \{\sigma_{i+1}, \ldots, \sigma_{n-1}\}$. This is, cover the remaining vertices w_j using only vertices that lie ahead of σ_i.

In the FKSP solved in order to estimate the cost of the unexplored path, the profit of an object $\sigma_i \in \varphi$ is given by the number of vertices $w_j \in W$ the object covers, and the capacity of the knapsack is given by $|\overline{\Omega}|$. The value of bound $h(\lambda)$ is then the sum of the weight values of the vertices chosen from φ. Figure 4.3 shows an example. Let us consider that the figures in parenthesis indicate the vertices $w_i \in W$ covered by vertices $\sigma_i \mid i \in \{1, \ldots, 6\}$, and that the label has been extended up to vertex σ_3 and no vertex has been skipped. Also, $|W| = 28$ and 13 vertices $w_i \in W$ have been covered. First, we determine the capacity of the knapsack which is given by the vertices that remain to be covered: $C = |W| - |\Omega| = |\overline{\Omega}|$. Applying this equation to our example yields $C = 28 - 13 = 15$.

Next, we define the objects that will be put into the knapsack. These are vertices $\varphi = \{\sigma_4, \sigma_5, \sigma_6\}$. The problem is solved with a greedy approach, so we need to determine the ratio $profit/weight$ for each item in φ. The profit is a piece of data stored for each σ_i, and its weight d_i is computed in each iteration as follows. Let $\Delta_i = \{d_{ji}\}_{j=0}^{i-1}$ be the set of weights of the in-going edges of σ_i, those that connect vertex σ_i with each of its predecessor vertices in the giant tour σ, and $\Delta_i' = \{d_{ik}\}_{k=i+1}^{n-1} \cup \{d_{i0}\}$ be the set of weights of the out-going edges of σ_i, those that connect vertex σ_i with each of its successor vertices in σ. Then, an estimation of the least travel cost (weight) of visiting vertex σ_i can be obtained by

$$d_i = \min \Delta_i + \min \Delta_i' \tag{4.2}$$

Exemplifying this for vertex σ_4 we have $\Delta_4 = \{d(0,4), d(1,4), d(2,4), d(3,4)\}$, $\Delta_4' = \{d(4,5), d(4,6), d(4,0)\}$, and $d_4 = \min \Delta_4 + \min \Delta_4'$. The same applies for the rest of the vertices in φ. Since our greedy strategy is to choose the items with the largest profit per unit weight first, we order these items in decreasing order of their $profit/weight$ ratio. Let us suppose we get σ_6, σ_4, σ_5. Then, we choose the items in this order, so we take all of σ_6 and σ_4, $C = 15 - 7 - 6 = 2$. However, the greedy algorithm never wastes any capacity in the FKSP, so we can use the 2 units of the remaining capacity to take $\frac{2}{5}$ of σ_5. Then, our total estimated cost for the path that remains to be explored is $h(\lambda) = d_6 + d_4 + \frac{2}{5}d_5$.

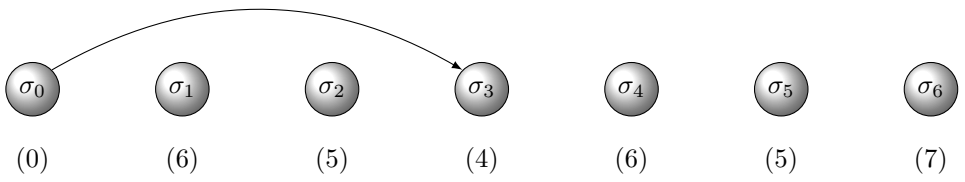

Figure 4.3: Giant tour used to exemplify the computation of a lower bound.

Applying this result to the original problem, if $\mu(\lambda) \geq UB_{\text{best}}$, the label can be pruned. Otherwise, the label is stored if it is non-dominated. In the bidirectional version

of *Selector*, the value of function $\mu(\lambda)$ is stored in a label field and it is applied again in the same manner when the label is retrieved for extension. This test is useful because the value of the best-known solution might have changed since the label was stored.

4.5 ALNS: Pseudocode and Parameters

The ALNS algorithm is used as described in Chapter 3, albeit its parameters need to be defined for the target problem. Algorithm 10 illustrates the ALNS process implemented to solve the CTP with simulated annealing as the outer metaheuristic that guides the search. This algorithm shows the specific implementation of the general process depicted in Figure A.1. The algorithm deals with two solution variables: the giant tour σ iteratively improved by the ALNS removal and insertion operators, and the CTP solution T produced by *Selector*. The ALNS-algorithm shown corresponds to the version of *Selector* that uses bidirectional search with bounding. The monodirectional version is very much alike. It only deletes line 16 which builds the structures needed to solve the FKSP in the bounding mechanism.

The process starts with a random giant tour σ^{init} which undergoes a local search procedure *2-opt* to improve its length. The rationale behind this optimisation is that a shorter tour allows *Selector* to work more efficiently. The algorithm then sets the initial temperature to the length of the initial giant tour and prepares the adaptive layer of the ALNS in order to start the loop of iterations. A destroy-repair pair of sub-heuristics tries to further improve the length of the giant tour. Next, *Selector* quickly finds a feasible solution (upper bound) over this enhanced tour. This bound serves as a probe to determine if *Selector* will be executed to optimality. Then, if this test is positive, the data needed to solve the FKSP which allows to determine a lower bound is computed. Next, if the solution obtained by *Selector* improves, it is accepted; otherwise, it may be accepted with a computed probability. Finally, the temperature and the adaptive layer are updated, and the process restarts by choosing again a pair of sub-heuristics.

To define an efficient parameter setting, we used the irace[1] package of López-Ibáñez et al. (2011). A separate set of 50 learning instances (some derived from TSPLIB and others randomly generated) was designed and both versions of *Selector*—unidirectional and bidirectional—were tuned independently. The learning instances used to calibrate the algorithm are, of course, different from the ones used in the benchmark. The two resulting sets of parameters are listed in Table 4.1, where M and B stand for monodirectional and bidirectional search with bound respectively. Only the first row of Table 4.1 indicates a value shared by both versions.

[1]The software and its documentation are available at http://iridia.ulb.ac.be/irace

Algorithm 10 : The General Framework of the ALNS with Simulated Annealing

Input: Giant tour σ^{init}, distance matrix D
Output: T_{best} and $c(T_{\text{best}})$ {Best CTP tour and its cost}

2-opt(σ^{init})
compute $l_0(\sigma^{\text{init}})$ {cost of initial giant tour}
initialize, to the same value, probability P_r^t for each removal operator $r \in R$, and likewise probability P_i^t for each insertion operator $i \in I$.
$t \leftarrow l_0$, {set initial temperature, variable used in probability function}
$l_{\text{current}} \leftarrow l_0$; $\sigma^{\text{current}} \leftarrow \sigma^{\text{init}}$
$UB_{\text{best}} \leftarrow$Search Upper Bound($\sigma^{\text{init}}$) {see Algorithm 8}
$c(T_{\text{best}}) \leftarrow c(T_{\text{current}}) \leftarrow$Selector($\sigma^{\text{init}}$) {see Algorithm 5}
$i \leftarrow 1$ {iteration counter}
repeat
  select a removal operator $r \in R$ with probability P_r^t {roulette wheel}
  obtain $\sigma^{\text{new}-}$ by applying r to σ^{current}
  select an insertion operator $i \in I$ with probability P_i^t
  obtain σ^{new} by applying i to $\sigma^{\text{new}-}$
  $UB_{\text{current}} \leftarrow$Search Upper Bound($\sigma^{\text{new}}$)
  if ($UB_{\text{current}} < \alpha \cdot UB_{\text{best}}$) **then**
    compute ratio *profit/weight* for each $\sigma_i \in \sigma^{\text{new}} \backslash \{\sigma_0\}$; sort ratios
    $c(T_{\text{new}}) \leftarrow$Selector($\sigma^{\text{new}}$)
  else
    $c(T_{\text{new}}) \leftarrow UB_{\text{current}}$
  end if
  if ($UB_{\text{current}} < UB_{\text{best}}$) **then**
    $UB_{\text{best}} \leftarrow UB_{\text{current}}$
  end if
  {decide acceptance of new solution}
  if ($c(T_{\text{new}}) < c(T_{\text{current}})$) **then**
    $c(T_{\text{current}}) \leftarrow c(T_{\text{new}})$; $\sigma^{\text{current}} \leftarrow \sigma^{\text{new}}$
  else
    $p \leftarrow e^{-\frac{c(T_{\text{new}}) - c(T_{\text{current}})}{t}}$
    generate a random number $n \in [0, 1]$
    {new solution might be accepted even if it is worse}
    if ($n < p$) **then**
      $c(T_{\text{current}}) \leftarrow c(T_{\text{new}})$; $\sigma^{\text{current}} \leftarrow \sigma^{\text{new}}$
    end if
  end if
  if ($c(T_{\text{new}}) < c(T_{\text{best}})$) **then**
    $c(T_{\text{best}}) \leftarrow c(T_{\text{new}})$; $T_{\text{best}} \leftarrow T_{\text{new}}$
  end if
  $t \leftarrow \beta \cdot t$ {cooling rate set to be very slow}
  if (segment size $= \varsigma$) **then**
    update probabilities using the adaptive weight adjustment procedure
  end if
  $i \leftarrow i + 1$
until (defined number of iterations is met)

Table 4.1: Values of the ALNS parameters after tuning with irace for both search algorithms.

Parameter	Meaning	M-Value	B-Value
γ	number of vertices removed in each ALNS iteration (instance size dependent)	$[0.3 \cdot \|V\|, \epsilon \cdot \|V\|]$	
ς	segment size for updating probabilities in number of ALNS iterations	100	75
τ	reaction factor that controls the rate of change of the weight adjustment	0.4	0.4
δ	avoids determinism in the SRH	7	6
ρ	avoids determinism in the WRH	2	3
κ_1	score for finding a new global best solution	60	40
κ_2	score for finding a new solution that is better than the current one	25	25
κ_3	score for finding a new non-improving solution that is accepted	15	15
β	cooling factor used by simulated annealing	0.99999	0.99999
ϵ	fixes the upper limit of vertices removed at each iteration	0.5	0.5

4.6 Computational Results

4.6.1 Benchmarking Conditions.

The majority of the studies done for the CTP have conducted their experimentation using test beds of randomly generated instances, and unfortunately no library of data sets exists in the literature. Therefore, in order to assess the effectiveness of our method, we carried out two benchmarks.

(1) Comparison of the results obtained by the monodirectional version of our metaheuristic against the optimal solutions computed by the branch-and-cut algorithm of Gendreau et al. (1997). For this purpose, small and medium-size instances were used.

(2) Comparison of the results obtained by both the monodirectional and bidirectional versions against each other. In this benchmark medium and large-sized instances were utilized. The unidirectional version consists of the basic algorithm with no performance enhancements, while the bidirectional version integrates bounding.

We created a test set based on 18 TSPLIB[2] instances (Reinelt (1991)) whose sizes range from 100 to 575 vertices. Set V is defined by taking the first $|V|$ points, while W is defined by the remaining points. The value $|V|$ was chosen to be around 15%, 25%, 35% and 50% of the instance size. However, it should not be understood that all instances were tested for these four different values of $|V|$, small or medium-sized ones use only two or three values. The end result is a set of 60 different test instances. In benchmark one, 24 instances were used, and in benchmark two 44. These last two numbers do not add up to 60 because eight instances were used in both benchmarks. The number contained in the instance name indicates its size.

[2]Files found at http://www.iwr.uni-heidelberg.de/groups/comopt/software/TSPLIB95/

The costs $\{c_{i,j}\}$ are treated as integer values equal to $\lfloor d_{ij} + .5 \rfloor$, where d_{ij} is the Euclidean distance between points i and j (Reinelt (1991)). The value of c is computed using

$$c = \max \left(\max_{v_k \in V \setminus \{v_0\}} \min_{w_l \in W} \{c_{l,k}\}, \max_{w_l \in W} \{c_{l,k(l)}\} \right) \tag{4.3}$$

where $k(l)$ indicates the vertex $v_k \in V \setminus \{v_0\}$ that is the second nearest to w_l. Computing this value in such way ensures that each vertex $v_i \in V \setminus \{v_0\}$ covers at least one vertex $w_i \in W$, and each vertex $w_i \in W$ is covered by at least two different vertices $v_i \in V \setminus \{v_0\}$ as explained in Gendreau et al. (1997).

Several independent executions were done to test our randomized heuristic. Each instance is solved 30 times with a different seed each time, and each execution lasts 30,000 iterations. The value of α, parameter used to make the test that determines the execution of *Selector* to optimality less strict (line 15 of Algorithm 10), was kept at one in both benchmarks. The heuristic algorithms are coded in C++ (gcc Ubuntu/Linaro 4.6.3-1ubuntu5) and the exact algorithm of Gendreau et al. (1997) is written in Python 2.7 and uses 5.6 Gurobi callbacks. Library Python-Igraph 0.7.0 helps to solve graph problems occurring in the valid cut separation. The benchmarks were done under Linux OS type 64 bits on an Intel Core i7-4770 CPU@3.40GHz machine with 8 GB of memory.

4.6.2 Discussion of Tables of Results

Table 4.2 shows the results of the first benchmark where 24 small and medium-size instances were used. As mentioned before, the quality of these results is evaluated using the output of the state-of-the-art exact algorithm of Gendreau et al. (1997). In this table, the first three columns document the instance information, the next three report the findings of the exact method, and the last five those of the heuristic approach. The solution quality is measured for each instance as a percentage of deviation from the best cost value found by the branch-and-cut algorithm, z_{BKS}. Then, $\overline{\theta} = 100 * (\overline{z} - z_{\text{BKS}})/z_{\text{BKS}}$, where $\overline{z}$ documents the average tour cost value computed over the 30 independent executions. Columns UB and Opt show the time in seconds needed to reach an upper bound and the optimal value respectively. Column $\overline{\theta}$ indicates the deviation of the heuristic solution from the best value found in percentage, and $\overline{t}$ corresponds to the total run time in seconds. These two figures are average values over the 30 executions. Column Found indicates how many times the heuristic found the optimum value out of the set of 30 executions. Column Best Gap shows how close (in percentage) the heuristic came to the optimum value, and the last one, labeled $S_{\text{N}-1}$, exhibits the corrected sample standard deviation.

The results shown in Table 4.2 allow us to state that it is a heuristic capable of identifying very good quality solutions quite quickly, since for 96% of the instances it was capable of finding the optimum value rapidly. In the few cases where the optimum was not reached, the minimum value computed was less than 1% away from the optimal solution value. In addition, the average deviation is typically within 1% of optimality, and it repeatedly found the optimum value for 63% of the instances. Furthermore, in

general, the spread around the optimum of the values computed is very moderate.

The second benchmark was done using a set of 44 medium and large-sized instances, and each was tested with both the monodirectional and the bidirectional with bounding versions. Thus, there are 88 possible results. A comparison of these results can be observed in Tables 4.3 and 4.4 which document the evolution of the search. In these tables, column $\overline{z}$ documents the average tour cost value computed over the 30 independent executions, while column $\overline{t}$ shows the average execution time (in seconds) obtained by the heuristic, both values are documented at the number of iterations indicated in the corresponding box. For each instance, the first line exhibits the evolution of the search for the unidirectional version, whereas the line below it for the bidirectional one. The abbreviation TO stands for time-out. To execute the 30,000 iterations, a time limit was set. If a given instance exceeds this limit at least once in the 30 times it is solved, then a TO mark is given to it. Hence, results not listed or labeled as TO in these two tables obtained a time-out mark. This occurred in 15 out of the 88 possible results.

The behaviour exhibited by the heuristic in the first benchmark leads us to believe that the solution values reported in Tables 4.3 and 4.4 are of high quality. However, since the results of both heuristics are compared against each other, no specification can be made regarding their optimality gap. In addition, the average gap obtained by our algorithm is better than the average gap reached by both the scatter search processes published in Baldacci et al. (2005) and the heuristic algorithm of Gendreau et al. (1997). Nonetheless, it was not possible to obtain the exact same instances to test. Both studies were conducted using randomly-generated instances no longer available.

Tables 4.3 and 4.4 demonstrate that, for 54 (74%) out of the 73 results obtained, there is no improvement on the solution value as the number of iterations increases. Faster execution times could be achieved by adding an additional criterion to stop the search when a certain number of iterations have taken place with no change. One also observes that the problem difficulty increases with $|V|$, but is fairly insensitive to $|W|$.

Table 4.5 lists the best values found by the two search processes—mono and bidirectional. Column $\overline{\theta}$ indicates the deviation (in percentage) of the final average value—$\overline{z}$ obtained at the end of the 30,000 iterations—from the best value found during the whole search process. The results show that for nearly 92% of the results, the average deviation lies within 1% of the best value found. They also demonstrate that both versions find the same best values but in one instance (pr_299, $|V| = 45$), and the deviation of the average value from the best value found is only slightly smaller for nearly all instances in the bidirectional search. However, it is important to note that the bidirectional version was capable of solving instance rd_400, $|V| = 100$, while the unidirectional was not.

Table 4.6 compares only one run (*Selector* is executed to optimality only once) of each type of search under exact circumstances in order to get an insight of the number of labels each version deals with during the search. It can be observed that the bidirectional search benefits from the bounding mechanism that prevents the creation of labels, since in 44% of the instances the number of labels stored is smaller than the one of the monodirectional version.

To compare the performance of both search versions, we applied the Mann-Whitney test with a significance value $\alpha = 0.05$. This is a well-known nonparametric statistical hypothesis test which performs a pairwise comparison of the heuristics. In each comparison, the null hypothesis is that the two heuristics exhibit equivalent performance while the alternative hypothesis is that one of them is better. This performance assessment was done using the tool suite provided in PISA[3]. Table 4.7 shows the results. The metrics considered are execution time, t, and objective function value, z. For the metric under examination, either the results of the bidirectional search are better ($\succ$) than those of the monodirectional version, either they are not better ($\prec$) or there is no statistically significant difference between both versions ($\equiv$). In each cell of Table 4.7, read row (bidirectional) against column (monodirectional). In this table, one observes that regarding the execution time, the unidirectional search yields better results in nearly 60% of the instances. This is not surprising because the bidirectional algorithm demands higher computational effort since it is calculating and sorting the items needed to solve the FKSP every time *Selector* is executed; for every label generated, it computes the lower bound useful to evaluate it; this heuristic manages twice the number of search queues; and it verifies the existence of complete solutions every time a label is stored. Regarding the second metric, both versions are equivalent most of the times.

4.7 Conclusions

Our contribution is the development of a novel solution method for a difficult combinatorial optimisation problem which finds application in network design and vehicle routing. Its key feature is the *Selector* operator which optimally splits an initial sequence of vertices into subsequences of visited and non-visited ones. The operator works within an adaptive large neighborhood search. It is a simple, easy to implement heuristic and its core, the *Selector* operator, is new and creative in its own right. We have proposed both a monodirectional and a bidirectional with bounding version of a heuristic method capable of obtaining very high quality solutions in short periods of time. For small and medium-sized instances, the gap to optimality is within 1%. The bidirectional version improves the performance of the original version from the point of view that it allowed to find solution values in instances where the latter could not. We have reported computational results for a set of instances whose size ranges from a 100 to 575 vertices and the tour may contain from 25 up to 268 vertices.

[3]Software available at http://www.tik.ee.ethz.ch/sop/pisa/?page=pisa.php

Table 4.2: Comparison of the results obtained by the heuristic and the branch-and-cut algorithm of Gendreau et al. (1997) using small and medium-sized instances.

			Exact Method (B&C)			Selector-ALNS				
Instance Based on	$\|V\|$	$\|W\|$	Optimum	UB(s)	Opt(s)	$\overline{\theta}$(%)	$\bar{t}$(s)	Found	Best Gap(%)	$S_{\mathrm{N}-1}$
kroA100	25	75	7985	0.17	0.17	2.36	0.42	14	0	272.51
kroA100	50	50	8608	22.20	44.95	0.21	0.95	11	0	23.06
kroB100	25	75	6449	0.21	0.27	0.16	0.50	24	0	22.79
kroB100	50	50	8043	1.18	21.54	0.70	1.25	1	0	60.20
kroC100	25	75	6161	0.01	0.01	0	0.81	30	0	0
kroC100	50	50	7942	0.81	0.81	0	2.27	30	0	0
kroD100	25	75	6651	0.24	0.38	0	0.31	30	0	0
kroD100	50	50	8411	3.75	4.33	0.02	1.13	27	0	4.64
kroE100	25	75	7417	0.26	0.27	0.02	0.42	29	0	8.71
kroE100	50	50	8493	1.10	1.11	0	1.00	30	0	0
kroA150	25	125	8050	0.13	0.13	1.43	0.54	3	0	131.72
kroA150	50	100	9623	118.80	121.58	0.37	1.16	2	0	38.56
kroA150	75	75	9971	1569.38	2884.34	0.60	2.93	0	0.59	59.81
kroB150	25	125	6165	0.01	0.01	0	1.50	30	0	0
kroB150	50	100	7818	1.16	1.16	0.02	2.32	29	0	7.23
kroB150	75	75	7434	13.34	38.24	0.01	4.32	26	0	2.16
kroA200	25	175	6165	0.01	0.01	0	1.63	30	0	0
kroA200	50	150	8273	0.46	0.49	0	5.09	30	0	0
kroA200	75	125	8499	141.97	266.19	0	6.31	30	0	0
kroA200	100	100	8355	4110.87	4789.22	0	14.21	30	0	0
kroB200	25	175	6450	0.15	0.15	0.18	1.17	23	0	24.62
kroB200	50	150	8171	2.69	3.46	0.78	2.6	3	0	77.29
kroB200	75	125	10007	0.65	0.65	1.42	4.5	5	0	166.52
kroB200	100	100	9988	17.20	17.68	1.73	11.60	5	0	202.63

Table 4.3: Comparison of the results obtained by the monodirectional and the bidirectional algorithms using large-sized instances (Part 1/2).

ALNS iterations			1,000		5,000		10,000		15,000		20,000		25,000		30,000	
Instance Based on	$\|V\|$	$\|W\|$	$\overline{z}$	$\overline{t}$	$\overline{z}$	$\overline{t}$	$\overline{z}$	$\overline{t}$	$\overline{z}$	$\overline{t}$	$\overline{z}$	$\overline{t}$	$\overline{z}$	$\overline{t}$	$\overline{z}$	$\overline{t}$
kroA200	25	175	6165	0.06	6165	0.28	6165	0.54	6165	0.80	6165	1.1	6165	1.3	6165	1.6
			6165	0.07	6165	0.31	6165	0.59	6165	0.88	6165	1.2	6165	1.5	6165	1.7
kroA200	50	150	8275	0.22	8275	0.85	8275	1.5	8275	2.1	8273	2.7	8273	3.3	8273	4.0
			8273	0.32	8273	1.1	8273	1.9	8273	2.7	8273	3.5	8273	4.3	8273	5.0
kroA200	75	125	8499	0.61	8499	1.8	8499	3.1	8499	4.1	8499	5.1	8499	5.9	8499	6.9
			8499	0.82	8499	2.1	8499	3.3	8499	4.5	8499	5.5	8499	6.5	8499	7.5
kroA200	100	100	8356	1.2	8356	4.3	8356	7.7	8356	10.3	8356	12.4	8356	14.4	8356	16.4
			8356	1.7	8356	5.8	8356	9.4	8356	12	8356	14.2	8356	16.4	8356	18.9
kroB200	25	175	6450	0.04	6450	0.20	6450	0.38	6450	0.57	6450	0.74	6450	0.94	6450	1.1
			6450	0.05	6450	0.22	6450	0.42	6450	0.62	6450	0.82	6450	1.0	6450	1.2
kroB200	50	150	8235	0.32	8230	0.78	8228	1.3	8225	1.7	8225	2.1	8223	2.6	8221	3
			8234	0.42	8233	0.91	8231	1.4	8224	1.9	8223	2.4	8219	2.8	8219	3.2
kroB200	75	125	10196	0.59	10196	1.8	10193	3.3	10185	4.5	10185	5.6	10185	6.8	10179	7.8
			10184	1.1	10160	2.8	10150	4.4	10150	5.8	10150	6.8	10144	7.8	10144	8.8
kroB200	100	100	10183	3.1	10143	6.9	10130	10.3	10120	14.6	10111	16.8	10103	19.1	10100	21.2
			10256	32.1	10238	61.7	10168	110	10141	121	10107	143	10094	155	10093	157
pr_264	40	224	5280	0.96	5280	4.2	5280	8.4	5280	12.8	5280	17.1	5280	21	5280	24.8
			5280	1	5280	5	5280	9.6	5280	14	5280	18.4	5280	22.6	5280	27
pr_264	66	198	5280	5.4	5280	26.9	5280	52.9	5280	79.7	5280	106	5280	132	5280	157
			5280	6.4	5280	31.2	5280	61.2	5280	90	5280	118	5280	147	5280	176
pr_264	92	172	5553	7.8	5553	33.1	5553	64.1	5553	92.1	5553	118	5553	147	5553	177
			5553	8.2	5553	33.6	5553	60.2	5553	88.1	5553	115	5553	141	5553	165
pr_264	132	132	6401	3.5	6401	21.7	6401	40.6	6401	56.3	6401	79	6401	105	6401	126
			6401	3.6	6401	28.6	6401	60.4	6401	89.6	6401	127	6401	170	6401	225
pr_299	45	254	10942	2.1	10942	9.2	10942	17.8	10942	26.1	10942	35.1	10942	44.5	10942	53.5
			10941	2.8	10941	11.6	10941	22.6	10941	34.9	10941	46.8	10941	57.9	10941	68.9
pr_299	75	224	10874	5.5	10874	27.3	10874	54.1	10874	79.7	10874	104	10874	127	10874	147
			10874	6.5	10874	34	10874	65.3	10874	96.8	10874	128	10874	156	10874	184
pr_299	105	194	13369	10.2	13369	40.3	13369	74.3	13369	105	13369	136	13369	166	13369	192
			13369	11.5	13369	50.3	13369	91.5	13369	129	13369	166	13369	201	13369	237
pr_299	150	149	11621	12.5	11621	48.7	11621	91.1	11621	132	11621	171	11621	209	11621	250
			11622	15.2	11622	60	11622	111	11622	157	11622	205	11622	250	11622	294
lin_318	48	270	3262	3.9	3262	15.7	3262	29.5	3262	42.3	3262	54.7	3262	65.5	3262	76.9
			3262	4.2	3262	15.7	3262	29.4	3262	43.8	3262	57.1	3262	70.9	3262	84.5
lin_318	80	238	4958	20	4958	98.9	4957	195	4957	292	4957	381	4957	470	4957	551
			4957	20.6	4957	95	4957	172	4957	236	4957	301	4957	374	4957	450
lin_318	112	206	8419	42.9	8419	207	8419	402	8419	589	8419	778	8419	962	8419	1,150
			8419	44.8	8419	220	8419	425	8419	619	8419	809	8419	992	8419	1,170
lin_318	159	159	5467	34.8	5467	180	5467	356	5467	537	5467	720	5467	905	5467	1,090
			5467	35.8	5467	187	5467	374	5467	557	5467	740	5467	926	5467	1,110
rd_400	60	340	2642	2.2	2631	4.1	2630	6.4	2630	8.6	2630	10.6	2630	12.6	2630	14.8
			2636	2.6	2626	5	2624	7.8	2624	10.3	2624	12.7	2624	15	2624	17.5
rd_400	100	300	TO													
			3881	127	3854	198	3852	636	3848	650	3846	657	3845	667	3845	753

Table 4.4: Comparison of the results obtained by the monodirectional and the bidirectional algorithms using large-sized instances (Part 2/2).

ALNS iterations			1,000		5,000		10,000		15,000		20,000		25,000		30,000	
Instance Based on	$\|V\|$	$\|W\|$	$\overline{z}$	$\overline{t}$	$\overline{z}$	$\overline{t}$	$\overline{z}$	$\overline{t}$	$\overline{z}$	$\overline{t}$	$\overline{z}$	$\overline{t}$	$\overline{z}$	$\overline{t}$	$\overline{z}$	$\overline{t}$
pcb_442	67	375	6009	5.9	6009	28.2	6009	53.1	6009	76.1	6009	97.5	6009	117	6009	135
			6009	5.9	6009	29.2	6009	55.6	6009	78.4	6009	101	6009	122	6009	144
pcb_442	111	331	4748	69.8	4743	273	4742	522	4742	782	4742	1,020	4742	1,280	4742	1,520
			4749	56.4	4744	227	4742	425	4742	615	4742	826	4742	1,030	4742	1,220
pcb_442	155	287	6282	99.8	6277	461	6271	914	6271	1,370	6271	1,810	6271	2,250	6271	2,700
			6279	73.2	6271	354	6271	697	6271	1,030	6271	1,380	6271	1,750	6271	2,100
pcb_442	221	221	6985	86.6	6985	400	6985	741	6985	1,090	6985	1,440	6985	1,770	6985	2,120
			6985	95.7	6985	443	6985	850	6985	1,240	6985	1,660	6985	2,070	6985	2,490
d_493	75	418	6840	7.6	6840	38.6	6840	77.4	6840	117	6840	157	6840	197	6840	235
			6840	4.8	6840	23.4	6840	46.6	6840	69.4	6840	91.5	6840	114	6840	136
d_493	124	369	6475	3.1	6475	15.1	6475	30.1	6475	45.4	6475	60.9	6475	76.3	6475	91.8
			6475	2.6	6475	13.2	6475	26.4	6475	39.5	6475	52.8	6475	65.7	6475	78.4
d_493	173	320	5682	106	5682	395	5682	706	5682	1,060	5682	1,450	5682	1,850	5682	2,270
			5682	116	5682	455	5682	881	5682	1,320	5682	1,750	5682	2,160	5682	2,650
att_532	81	451	2262	66.7	2262	330	2262	631	2262	841	2262	1,010	2262	1,160	2262	1,340
			2262	62.1	2262	288	2262	501	2262	673	2262	865	2262	1,060	2262	1,220
att_532	134	398	2715	139	2715	732	2715	1,390	2715	1,920	2715	2,470	2715	3,080	2715	3,630
			2715	143	2715	782	2715	1,550	2715	2,240	2715	2,940	2715	3,670	2715	4,400
ali_535	81	454	43537	5.9	42888	32.8	42787	62.5	42706	93.2	42705	127	42697	159	42697	191
			43340	9.1	42837	43.1	42820	85.1	42742	122	42721	158	42706	193	42706	231
ali_535	135	400	46089	15.2	44788	85.2	44119	195	43760	318	43634	431	43454	548	43454	662
			46213	13.6	44393	84.5	43953	170	43798	254	43540	336	43427	413	43427	492
ali_535	188	347	49735	37	49307	188	48432	394	48141	603	47563	782	47145	948	46842	1,140
			49874	39	49111	200	48254	383	47522	604	47249	816	46645	1,020	46493	1,190
ali_535	268	267	50000	20.6	50000	131	50000	305	50000	488	50000	689	50000	909	50000	1,150
			50000	23.8	50000	247	50000	553	50000	917	50000	1,250	50000	1,600	50000	1,920
rat_575	87	488	262	52.8	262	256	262	503	262	742	262	970	262	1,200	262	1,430
			262	68	262	339	262	675	262	969	262	1,250	262	1,510	262	1,790
rat_575	145	430	375	110	375	468	375	915	375	1,350	375	1,790	375	2,230	375	2,650
			375	123	375	584	375	1,100	375	1,590	375	2,070	375	2,540	375	3,010

Table 4.5: Best values found by the monodirectional and bidirectional searches.

Instance	$\|V\|$	$\|W\|$	MONODIRECTIONAL		BIDIRECTIONAL w/BND	
Based on			best found	$\overline{\theta}$ (%)	best found	$\overline{\theta}$ (%)
kroA200	25	175	6165	0	6165	0
	50	150	8273	0	8273	0
	75	125	8499	0	8499	0
	100	100	8356	0	8356	0
kroB200	25	175	6450	0	6450	0
	50	150	8171	0.61	8171	0.59
	75	125	10008	1.71	10008	1.36
	100	100	9988	1.12	9988	1.05
pr_264	40	224	5280	0	5280	0
	66	198	5280	0	5280	0
	92	172	5553	0	5553	0
	132	132	6401	0	6401	0
pr_299	45	254	10942	0	10941	0
	75	224	10874	0	10874	0
	105	194	13369	0	13369	0
	150	149	11621	0	11621	0
lin_318	48	270	3262	0	3262	0
	80	238	4957	0	4957	0
	112	206	8419	0	8419	0
	159	159	5467	0	5467	0
rd_400	60	340	2621	0.34	2621	0.11
	100	300	NR		3820	0.65
pcb_442	67	375	6009	0	6009	0
	111	331	4742	0	4742	0
	155	287	6271	0	6271	0
	221	221	6985	0	6985	0
d_493	75	418	6840	0	6840	0
	124	369	6475	0	6475	0
	173	320	5682	0	5682	0
att_532	81	451	2262	0	2262	0
	134	398	2715	0	2715	0
ali_535	81	454	42643	0.13	42643	0.15
	135	400	43150	0.70	43150	0.64
	188	347	45450	3.06	45450	2.29
	268	267	50000	0	50000	0
rat_575	87	488	262	0	262	0
	145	430	375	0	375	0

Table 4.6: Comparison of the number of labels produced in one iteration of Selector and CPU time used.

Instance	$\|V\|$	$\|W\|$	MONODIRECTIONAL			BIDIRECTIONAL w/BND		
Based on			created	stored	t (s)	created	stored	t (s)
kroA200	25	175	564	171	0	1,073	303	0
	50	150	19,010	1,801	0	21,891	2,173	0.01
	75	125	68,408	5,121	0.02	107,849	5,381	0.03
	100	100	287,689	11,356	0.06	371,212	9,947	0.09
kroB200	25	175	2,525	500	0	1,259	508	0
	50	150	72,764	8,262	0.03	32,206	4,771	0.02
	75	125	102,657	10,607	0.03	101,326	15,579	0.07
	100	100	337,443	26,695	0.11	329,576	24,564	0.14
pr_264	40	224	285	169	0	217	87	0
	66	198	1,365	281	0	1,417	296	0.01
	92	172	1,957	383	0	2,653	594	0.01
	132	132	5,833	709	0	186,331	8,706	0.15
pr_299	45	254	5,718	2,246	0	5,856	2,236	0.01
	75	224	11,728	4,559	0.02	8,898	3,051	0.01
	105	194	337,600	37,879	0.71	213,819	32,757	0.76
	150	149	825,417	56,829	1.11	473,044	37,891	0.81
lin_318	48	270	2,473	353	0	2,526	368	0.01
	80	238	19,461	2,018	0.02	16,778	2,075	0.02
	112	206	509,751	9,320	0.22	570,819	9,654	0.26
	159	159	60,046	5,278	0.05	55,965	4,423	0.04
rd_400	60	340	247,176	15,257	0.25	243,003	16,594	0.30
	100	300	13,257,328	316,542	28.10	7,278,543	188,562	12.94
	140	260	17,228,926	529,899	68.32	8,691,701	291,064	19.31
pcb_442	67	375	33,514	24,566	0.02	7,251	3,814	0
	111	331	79,665	10,279	0.09	63,120	8,540	0.08
	155	287	375,959	36,638	0.36	242,391	18,208	0.20
	221	221	1,782,107	97,848	0.72	582,023	34,988	0.38
d_493	75	418	11,242	969	0.01	8,066	1,217	0.01
	124	369	730	158	0.01	1,077	432	0
	173	320	765	289	0	1,359	528	0.01
	247	246	285	246	0.17	532	477	0.01
att_532	81	451	15,800	622	0.01	19,398	835	0.02
	134	398	56,708	1,525	0.05	100,868	3,055	0.07
	187	345	430,068	5,140	0.35	602,604	8,729	0.58
	266	266	1,348,446	11,705	0.86	2,220,767	27,703	1.86
ali_535	81	454	51,207	3,603	0.05	155,966	7,859	0.15
	135	400	1,543,983	28,002	1.12	948,595	17,205	0.93
	188	347	790,841	19,151	0.86	629,960	9,008	0.60
	268	267	468,916	21,781	0.33	6,816,306	92,834	9.17
rat_575	87	488	3,749	785	0.01	2,823	600	0.01
	145	430	16,589	2,846	0.12	24,246	2,448	0.05
	202	373	45,693	6,848	0.11	101,751	7,025	0.15
	288	287	664,958	22,559	0.73	246,734	9,661	0.22

Table 4.7: Results of the Mann-Whitney statistical test for the monodirectional and bidirectional algorithms.

Instance				MONO	
Based on	$\|V\|$	$\|W\|$		t	z
kroA200	25	175	BI	$\prec$	$\equiv$
	50	150	BI	$\prec$	$\equiv$
	75	125	BI	$\prec$	$\equiv$
	100	100	BI	$\prec$	$\equiv$
kroB200	25	175	BI	$\prec$	$\equiv$
	50	150	BI	$\prec$	$\equiv$
	75	125	BI	$\prec$	$\equiv$
	100	100	BI	$\prec$	$\equiv$
pr_264	40	224	BI	$\prec$	$\equiv$
	66	198	BI	$\prec$	$\equiv$
	92	172	BI	$\equiv$	$\equiv$
	132	132	BI	$\prec$	$\equiv$
pr_299	45	254	BI	$\prec$	$\equiv$
	75	224	BI	$\prec$	$\equiv$
	105	194	BI	$\prec$	$\equiv$
	150	149	BI	$\prec$	$\prec$
lin_318	48	270	BI	$\equiv$	$\equiv$
	80	238	BI	$\equiv$	$\equiv$
	112	206	BI	$\equiv$	$\equiv$
	159	159	BI	$\equiv$	$\equiv$
rd_400	60	340	BI	$\prec$	$\equiv$
	100	300	BI	$\succ$	$\succ$
pcb_442	67	375	BI	$\equiv$	$\equiv$
	111	331	BI	$\equiv$	$\equiv$
	155	287	BI	$\equiv$	$\equiv$
	221	221	BI	$\prec$	$\equiv$
d_493	75	418	BI	$\equiv$	$\equiv$
	124	369	BI	$\equiv$	$\equiv$
	173	320	BI	$\prec$	$\equiv$
att_532	81	451	BI	$\equiv$	$\equiv$
	134	398	BI	$\prec$	$\equiv$
ali_535	81	454	BI	$\prec$	$\equiv$
	135	400	BI	$\equiv$	$\equiv$
	188	347	BI	$\equiv$	$\equiv$
	268	267	BI	$\prec$	$\equiv$
rat_575	87	488	BI	$\prec$	$\equiv$
	145	430	BI	$\equiv$	$\equiv$

Chapter 5

Solving the Multi-Vehicle Covering Tour Problem with *m*-*Selector*

Contents

5.1 Introduction

In this fifth chapter, the solution procedure proposed to solve the multi-vehicle version of the *Covering Tour Problem* (*m*-CTP) is discussed. The *m*-CTP is a generalisation of the CTP. Hence, like the CTP, it considers two kinds of geographically scattered locations. The first kind are the covering ones, potential locations at which vehicles may stop, and the second kind are the locations to cover. The latter cannot be visited, but they must lie within the required distance of a vehicle route. The problem consists in identifying a minimum-length set of routes over the covering locations such that those that cannot be visited by the routes are close enough to a covering location.

To solve the problem, a version of *Selector* that manages the definition of routes was implemented. This version is named *m-Selector*. The method is competitive as shown by the quality of results evaluated using the output of exact (Hà et al. (2013) and Jozefowiez (2014)) and heuristic methods (Hà et al. (2013) and Kammoun et al. (2015)).

The remainder of the chapter is organised as follows. Section 5.2 formally presents the problem, discusses applications of it, and provides a view of some of the solution approaches that have been presented in the literature. The methods used as reference to compare our results are explained more precisely. Sections 5.3–5.4 explain the solution method proposed in this thesis. Section 5.5 documents the results obtained. Finally, Section 5.6 discusses the contribution of this work.

5.2 The Multi-Vehicle Covering Tour Problem

5.2.1 Formal Definition of the m-CTP

The m-CTP can formally be described by considering a complete undirected graph $G = (N, E)$ where the vertex set is represented by $N = V \cup W$ and the edge set by $E = \{(v_i, v_j) | v_i, v_j \in N, i < j\}$. The subset of n vertices that *can* be visited at most once is given by $V = \{v_0, v_1, v_2, \ldots, v_{n-1}\}$, while the subset of vertices that must be *covered* but cannot be visited is given by W. Set V includes a subset T of vertices that *must* be visited, $|T| \geq 1$. Vertex v_0 represents the depot where m identical vehicles are available. Let d_{ij} be the distance associated with each edge $(i, j) \in E$, and $D = (d_{ij})$ the distance matrix defined on E that satisfies the triangle inequality. A vertex $w_i \in W$ is covered if there exists at least one vertex $v_j \in V$ in the set of routes for which $d_{ij} \leqslant c$, where c is known as the *covering distance*. Each vertex in V covers a subset of W, so for each $w_i \in W$, a subset of vertices of V that can cover it is given. The m-CTP calls to find a set of m vehicle routes such that the total travel cost is minimised and the following constraints are satisfied:

i. There are at most m vehicle routes, and each of them starts and ends at the depot vertex v_0.

ii. Each vertex $v_i \in V$ belongs to at most one route, while each vertex $v_i \in T$ belongs to exactly one route.

iii. Each vertex $w_i \in W$ is covered by only one route.

iv. The number of vertices on any route cannot exceed a preset value p (depot not included).

v. The length of each route cannot exceed a prefixed value q.

The m-CTP is NP-hard as it reduces to a CVRP with unit demand when $T = V$ and $W = \emptyset$, or simply to a CTP when there are no capacity constraints. Figure 5.1 shows a feasible m-CTP tour for an instance where $|V| = 13$, $|W| = 20$, $m = 4$, and $p = 3$. The square represents the depot.

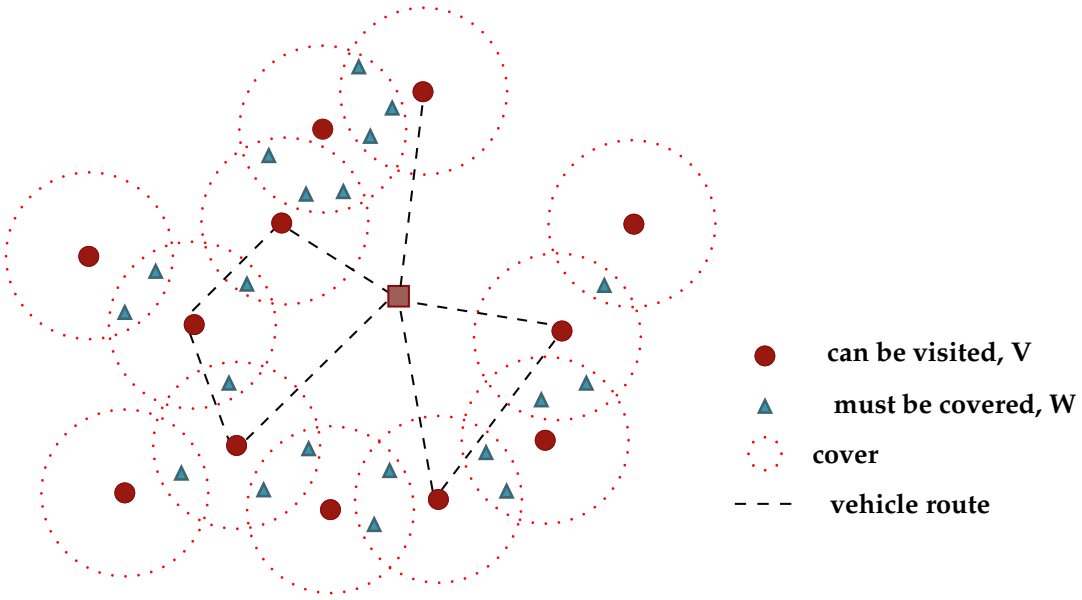

Figure 5.1: Example of m-CTP tour.

5.2.2 Applications Reported in the Literature

Likewise the CTP, the problem finds application in the design of bilevel transportation networks, for example, networks for humanitarian aid distribution. In an emergency situation, help crews cannot visit every single dwelling. Rather, a set of help-dispatch centres is established, and in order to obtain survival goods, inhabitants are expected to reach these centres which are located within reasonable distance. The centres must be supplied with goods from a central depot using a capacitated vehicle fleet.

A scenario of disaster relief distribution based on a covering problem is studied by Doerner and Hartl (2008) in the Austrian context. Nolz et al. (2010) defined a multi-objective covering problem to plan routes for the delivery of drinking water to the population affected by a disaster. The objectives they considered were minimise (1) the total distance travelled by the covered customers in order to reach their nearest visited stop; (2) the number of customers unable to reach a visited stop within a prespecified maximum distance; (3) the tour length; and (4) the latest arrival time at a customer. They solved bi-objective problems by considering simultaneously only two of the objectives formerly presented. They developed a hybrid metaheuristic encompassing genetic algorithms, variable neighborhood search and path relinking in order to solve these bi-objective problems. In addition, the algorithm is tested on real-world data from the province of Manabí in Ecuador.

Naji-Azimi et al. (2012) use the m-CTP to model the location and supply of distribution centers which aim to provide humanitarian help to the people affected by a disaster. They present a mathematical model, and a heuristic solution approach: a multi-start local search algorithm. Their local search contains four procedures designed

to tackle specific characteristics of a solution.

In the context of health care distribution, the dairy industry presents the location and routing of milk collection points and animal health care facilities which farmers can reach, Simms (1989). In Western countries, a problem, similar to the one presented by Hodgson et al. (1998) for the CTP, is encountered in the design of routes for mobile health care prevention teams (Brown and Fintor, 1995). Another application arises in the study of Oliveira et al. (2013) in which they adapt the model of the m-CTP in order to design routes for patrolling urban geographical areas in the city of Vinhedo, São Paulo, Brazil. They solve their model with a heuristic approach based on four heuristic algorithms that are an extension of the heuristics presented by Hachicha et al. (2000) and classic VRP heuristics.

Flores-Garza et al. (2015) presented a variant of the m-CTP where the objective function is modified to better suit the considerations made by humanitarian logistics. Classic logistics problems aim at maximising profit, while humanitarian logistics aims at minimising losses by providing relief to the victims as soon as possible (Galindo and Batta, 2013). Therefore, length minimisation may not properly reflect the need for quick service, equity and fairness which characterise the transport of survival goods. Instead, fairer objective functions such as the minimisation of the sum of arrival times at delivery points have been proposed (Ngueveu et al., 2010). Flores-Garza et al. (2015) consider such function and propose a mixed integer linear formulation and a GRASP-based solution procedure.

5.2.3 Solution Approaches Reported in the Literature

The branch-and-cut method developed by Gendreau et al. (1997) for the CTP does not extend easily to the m-CTP, as this problem turns out to be more difficult to solve than the standard VRP, which itself can rarely be solved exactly when $|V| > 100$. Heuristic approaches were the first attempts to solve the problem. Hachicha et al. (2000) introduced the m-CTP. They used a three-index vehicle flow formulation, and three heuristics based on classic VRP algorithms: savings, sweep and route first–cluster second to solve randomly-generated instances with $|V| \leq 200$. They compared the results of the three heuristics to each other. The optimality gap is, therefore, unknown. Not many publications can be identified after this introductory article until Tricoire et al. (2012) presented a bi-objective variant with stochastic demand. Cost and expected uncovered demand define the two objectives. The problem was solved with a branch-and-cut technique within an ϵ-constraint algorithm. Furthermore, the approach was applied to solve the organisation of a disaster relief operation for rural communities in Senegal.

Later on, Hà et al. (2013) proposed a new formulation for a variant of the m-CTP—named the m-CTP-p—where the route length constraint is relaxed and the m number of vehicles used is a decision variable. They used an exact as well as a heuristic method to solve the problem. Murakami (2014) formulated the m-CTP as a *Set Covering Problem* (SCP). To solve it, he suggested a column generation approach where the route generation subproblem is solved by the cheapest insertion heuristic and

the heuristic of Gendreau et al. (1997) for the CTP. The testing considers randomly-generated instances with $|V| \leq 500$. He compares his results against the ones of the three heuristics of Hachicha et al. (2000), and obtains better quality solutions, but worse execution times. Kammoun et al. (2015) solved the same variant of the m-CTP as Hà et al. (2013) using an algorithm based on the variable neighborhood search (VNS) proposed by Mladenović and Hansen (1997). Their results outperform those of Hà et al. (2013).

Recently, Allahyari et al. (2015) published a multi-depot version of a variant of the m-CTP, the *Multi-Depot Covering Tour Vehicle Routing Problem*. They developed two mixed integer programming formulations and implemented a hybrid metaheuristic combining GRASP, iterated local search and simulated annealing. They treat instances with $|V| \leq 90$. For values of $|V| \leq 30$ they solve their mathematical model with a commercial solver and compare the results of their heuristic against this output. They obtain a very competitive comparison.

Exact methods to solve the problem have also been proposed. Lopes et al. (2013) proposed a branch-and-price algorithm which enforces all the constraints defined in the problem. The pricing subproblems reduce to the RCESPP and are solved with dynamic programming. However, to find routes with negative reduced cost faster, they also implement a GRASP algorithm which embeds a VNS metaheuristic in the local search phase. Their test bed is composed of randomly-generated instances with $|V| \leq 200$, but the algorithm only managed to solve to optimality 33% of it within a four-hour execution time limit. Jozefowiez (2014) suggested a path-based model for the m-CTP and a branch-and-price algorithm to solve it.

5.2.4 Solution Approaches Used as Reference

Hà et al. (2013) contributed by providing a model, a-branch-and-cut procedure and a two-phase hybrid metaheuristic. Their formulation is based on the one done for the CTP by Baldacci et al. (2005), and considers that each vertex of $V \setminus \{v_0\}$ has a unit demand. They relaxed the constraint on the length of each route of the original problem because their formulation cannot express it. However, its advantage is that the number of variables and constraints increases polynomially with the instance size. Their branch-and-cut method uses the valid inequalities proposed by Baldacci et al. (2005) for their CTP solution method. They allotted a running time of two hours to their exact algorithm.

In their two-phase heuristic, the first phase solves exactly a SCP in order to generate a subset of V that can cover all the vertices of W. That is to say, the generated set are the vertices that must be visited, a CTP-tour. The problem then becomes a VRP which is solved by an algorithm based on the evolutionary local search (ELS) method of Prins (2009). The backbone of their ELS implementation is the *Split* operator. Using one of the sets generated in the first phase, their algorithm produces a giant tour over which the *Split* operator is applied. Next, the set of routes obtained undergoes local search to improve its quality. They use two classic local search neighborhoods: relocation of a vertex and swapping the position of two vertices. They introduce: combining two

unsaturated routes and replacing a vertex in the solution by a new vertex. For the next iteration, they concatenate the set of routes in order to obtain the original giant tour and apply a mutation operator over it to obtain a new one. The algorithm iterates a fixed number of times. The results obtained by the metaheuristic, compared against the output of their branch-and-cut method when possible, show an optimality gap of up to 1.5%. Their study considered TSPLIB-derived instances with $|V| \leq 100$.

In the branch-and-price approach of Jozefowiez (2014), the subproblem is a variant of the *Profitable Tour Problem*, which is modeled as a *Ring Star Problem* and solved with a branch-and-cut procedure. The algorithm was applied to randomly-generated instances and to TSPLIB-derived instances with $|V| \leq 60$. Some of the tests were done relaxing the route length constraint, while others relaxing the restriction on the number of visited vertices on a route. His experimentation showed that the constraint on the number of vertices is easier to manage than the one on the route length in terms of execution times as the subproblem is easier to solve.

Kammoun et al. (2015) proposed a simple deterministic case of VNS, known as variable neighborhood descent (VND), which is based on finding the best neighbor. The basic idea of VNS and its variants is the systematic change of neighborhoods within a local search procedure when the search becomes trapped at local optima (Hansen et al., 2010). More precisely, VNS explores larger and larger neighborhoods of the incumbent solution. The search jumps from its current point in the solution space to a new one if an improvement has been made or some acceptance criterion is met. Kammoun et al. (2015) used two neighborhood structures within their VND algorithm: inserting the vertex that maximises the number of covered vertices, and swapping a vertex that belongs to the solution by one not on it.

5.3 Application of *m*-*Selector* to Solve the *m*-CTP

The considerations done in this study are the following.

i. The m number of vehicles used is a decision variable.

ii. Firstly, the restriction on the number of vertices allowed on a route is considered, $p \leq k$, and the restriction on the route length is relaxed, $q = +\infty$. Secondly, $p = +\infty$ and $q \leq 2\varphi + \varrho$, where

$$\varphi = \max_{v_j \in V \setminus \{v_0\}} \{c_{0,j}\} \tag{5.1}$$

$c_{0,j}$ is the cost of reaching vertex v_j from the depot (weight of edge (v_0, v_j)), and ϱ is a given constant. The rationale behind the value of q is as follows. If a vehicle is needed to serve every vertex $v_j \in V \setminus \{v_0\}$, then for the problem to be feasible, the minimum route length needs to be $q = 2\varphi$.

iii. Every vertex $v_i \in V \setminus \{v_0\}$ has a demand of one, therefore, every vehicle has a capacity of p.

The principles of operation of *m-Selector* are very similar to those of the single-vehicle version. The algorithm acts on a sequence of vertices that represent customers, and simultaneously selects the best ones to visit and assigns them to optimal vehicle routes, while keeping the original routing order and satisfying side constraints. The selection problem is modelled as a RCESPP in an auxiliary acyclic directed graph, while the problem of segmenting the selected vertices into feasible routes is solved via a modified version of the *Split* operator embedded into the RCESPP algorithm. Formally, the input of the algorithm is a permutation σ of set V, where the depot is always the first vertex. The output is a subset $V' \subset V$ segmented into routes and to be visited in the same order given in σ such that the value of the objective function considered is optimal and the side constraints are satisfied.

As explained in Chapter 3, the core of the algorithm remained the same, however, changes were indeed necessary to enable the construction of more than one route. The next sections explain the changes introduced in order to have a multi-vehicle version.

5.3.1 Label Definition

In general terms, the definition of a label is the same as the former one. A label $\lambda = [\zeta, \nu, \omega]$ stores

i. ζ, the optimal cost of the partial multi-route tour.

ii. ν, the last vertex visited in the elementary partial tour.

iii. ω, a vector indexed by the vertices of W where $\omega[i]$ indicates either $w_i \in W$ is already covered or the number of vertices of V that can still be visited to cover it.

However, auxiliary vectors, used to compute the split cost and keep the multi-route tour, are also memorized. No field is maintained for the vehicle capacity Q, since this information can be inferred from the number of vertices already visited. Recall that all vertices $v_i \in V \setminus \{v_0\}$ have a unit demand.

5.3.2 Dominance Rule

Dominance tests are always performed when labels are extended, so that the algorithm records only non-dominated labels. A label $\lambda_1 = [\zeta_1, \nu_1, \omega_1]$ dominates a label $\lambda_2 = [\zeta_2, \nu_2, \omega_2]$ with $\lambda_1 \neq \lambda_2$ if and only if:

i $\nu_1 = \nu_2$

ii $\zeta_1 \leq \zeta_2$

iii $\{w_i : \omega_2[i]$ indicates w_i is covered$\} \subseteq \{w_i : \omega_1[i]$ indicates w_i is covered$\}$

iv $Q_1 \geq Q_2$

This is to say, two labels are comparable only when arriving at the same vertex. In this way, equivalent past and future trajectories are compared. One label dominates when its cost is lower or equal, its set of covered vertices is a superset of the second one, and its remaining vehicle capacity is larger or equal. It is important to consider the vehicle capacity because the value of this variable determines how soon the vehicle must return to the depot, so a label with larger capacity has the possibility of attaining a lower tour cost.

5.3.3 Cost Function

When extending a label from vertex σ_i to a successor vertex σ_k with $k > i$, the single-vehicle version computes the cost of label λ_k using a very simple recurrence equation which simply adds the weight of the arc added to the path to the known cost:

$$\zeta(\lambda_k) = \zeta(\lambda_i) + cost(i,k) \ \mid 1 \leq k \leq n - i - 1 \tag{5.2}$$

Instead, the *m-Selector* operator computes the cost of segmenting the path represented by the label into feasible routes. The cost of a label is, therefore, the cost of the best routes possible. This computation is done with a modified version of the *Split* operator. It is modified in the sense that it computes the arc weights of graph H of the *Split* algorithm as vertices are added to the path represented by the label. The fact that the demand is equal for all customers allows to implement a very simple algorithm. Then, to determine the best split for the set of vertices included in the label, one needs to figure out only the lowest travel cost possible to reach the last added vertex. This means one has to find the weight of the in-going arcs that exist in graph H for the added vertex and choose the best one. The cost function varies slightly depending on the restriction relaxed, $q = +\infty$ (route length) or $p = +\infty$ (number of vertices visited on a route), so the two versions will be explained separately. However, these algorithms can be merged into one (see Vargas et al. (2016)). For clarity reasons, they will be explained separately.

Considering p constant and $q = +\infty$. Algorithm 11 shows how it is computed. Vectors $V[\,]$ and $P[\,]$ have the same meaning as in the *Split* algorithm. They store the best travel cost to the corresponding vertex and the index of the predecessor vertex respectively. Since demand is equal for all customers, the route capacity is given by p. Hence, at most p arcs must be constructed in graph H. Firstly, the algorithm determines the vertex where the construction of the arcs in graph H will stop, variable $lastDepot$. Secondly, it iteratively computes the weight of at most p arcs, according to the definition of arcs given by the *Split* operator. The one of lowest weight is the one stored.

Figure 5.2 illustrates an example in which $p = 4$ and $\nu = 6$. Graph H represents the vertices stored so far in the label. First, the stopping point is determined. In this case, it is vertex σ_2, since every route can have at most four vertices. Recall that in the *Split* algorithm the tail of an arc rests at the depot, so the first iteration computes the

Algorithm 11 : Compute $\zeta(\lambda)$ in *m-Selector* when $q = +\infty$

Input: extended label $\lambda = [\ \zeta, \nu, \omega \mid V[\ ], P[\ ]\ ]$, p
Output: best total route cost stored in $V[\nu]$

```
1: i ← ν
2: lastDepot ← ν − p
3: if ( lastDepot < 0 ) then
4:    lastDepot ← 0
5: end if
6: path ← cost(σ_i, σ_0)
7: while ( i > lastDepot ) do
8:    arcH = V[i − 1] + cost(σ_0, σ_i) + path
9:    if ( arcH < V[ν] ) then
10:      V[ν] ← arcH
11:      P[ν] ← i − 1
12:   end if
13:   path ← path + cost(σ_{i−1}, σ_i)
14:   i ← i − 1
15: end while
```

arc cost for trip $\{\sigma_0, \sigma_6, \sigma_0\}$ (shown in green). The cost to get to vertex σ_5 is known from the previous computation. In the second iteration, the depot is now at σ_4, so it computes the cost of trip $\{\sigma_0, \sigma_5, \sigma_6, \sigma_0\}$ and the cost to get to σ_4 is again known. It follows in the same fashion until it reaches vertex σ_2 and constructs the arc that represents trip $\{\sigma_0, \sigma_3, \sigma_4, \sigma_5, \sigma_6, \sigma_0\}$. No more customers can be put into the route, so it stops.

Considering $q = 2\varphi + \varrho$ and $p = +\infty$. Algorithm 12 shows how it is computed. This algorithm shares many similarities with Algorithm 11. Vectors $V[\ ]$ and $P[\ ]$ have the same meaning. They store the best travel cost to the corresponding vertex and the index of the predecessor vertex respectively. Variable i stores the position of the depot. The algorithm constructs arcs in graph H as long as the route length is within the limit, and there still are vertices where to place the depot. Since the cost matrix satisfies the triangle inequality, it is safe to stop once the maximum route length is exceeded. Then, the algorithm constructs at most ν arcs.

5.3.4 Sets of Elite Vertices

As the size of V increases, the problem becomes more challenging. For this reason, we established a strategy for working with a smaller set V, but composed of elite vertices that might still allow to find an optimal solution or at least a solution of high quality in low execution times. In the test bed considered in the benchmark, the only instance that showed very long execution times, and which motivated the design of this strategy, was the one derived from kroB200 with $|V| = 100$.

The strategy is the following. We use different vertex-selection criteria to generate

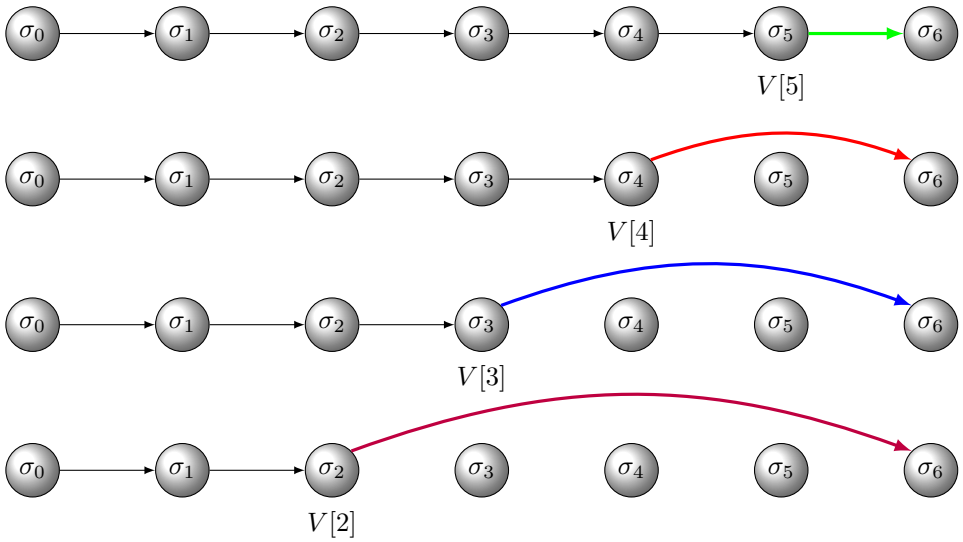

Figure 5.2: Example of the computation of the cost of a label for $p = 4$ and $q = +\infty$. The first row, from top to bottom, shows the first iteration, the second row depicts the second one, and so forth.

different subsets of V, $\Gamma = \{\gamma_i \mid \gamma_i \subset V\}$ for $i \in \{1, \ldots, s\}$. Next, we merge the sets obtained into a single set of elite vertices, γ'. Finally, the *m-Selector*-based algorithm previously presented is executed to obtain the final solution.

It is important to say that, in order to obtain a feasible solution, any subset $\gamma_i \subset V$ created must be capable of covering all the vertices in W. When a set of elite vertices is used, the algorithm works in exactly the same way as explained, but it does so over a sub-graph of G whose set V is built with the elite vertices. It is worth noting that the process of reducing set V still requires computing the covering distance with the original set, so that we insure the problem solved is the original problem.

Let Ω^i be the set of vertices $w_i \in W$ covered by vertex $v_i \in V \setminus \{v_0\}$, and χ_i be the sum of two edges connecting vertex $v_i \in V \setminus \{v_0\}$ to any other vertex $v_j \in V$. Then, the following criteria for creating subsets of V were applied:

i. Compute the ratio $\dfrac{|\Omega^i|}{\log_2 \chi_i}$ for each vertex $v_i \in V \setminus \{v_0\}$, where χ_i considers the two smallest edges (on the original graph G) of vertex σ_i. Order the vertices in decreasing order of this ratio. To build the subset, select the vertices, starting by best, until every vertex $w_i \in W$ is covered by at least two vertices $v_i \in V \setminus \{v_0\}$. While selecting, discard redundant vertices.

ii. Same as above but the edges considered are the largest ones.

iii. Compute the ratio $\dfrac{|\Omega^i|}{\log_2 \chi_i}$ for each vertex, where χ_i considers the two largest edges of vertex σ_i. Order the vertices in decreasing order of this ratio. Starting by the best vertex, determine the vertices it dominates below it in the ordering

Algorithm 12 : Compute $\zeta(\lambda)$ in *m-Selector* when $p = +\infty$

Input: extended label $\lambda = [\ \zeta, \nu, \omega \mid V[\ ], P[\ ]\ ]$, q
Output: best total route cost stored in $V[\nu]$

```
1: i ← ν
2: fits ← true
3: path ← cost(σ_i, σ_0)
4: while ( fits ) and ( i > 0 ) do
5:    route ← cost(σ_0, σ_i) + path
6:    if ( route ≤ q ) then
7:       arcH = V[i − 1] + route
8:       if ( arcH < V[ν] ) then
9:          V[ν] ← arcH
10:         P[ν] ← i − 1
11:      end if
12:      path ← path + cost(σ_{i−1}, σ_i)
13:      i ← i − 1
14:   else
15:      fits ← false
16:   end if
17: end while
```

made and eliminate them. Continue until all vertices $v_i \in V \setminus \{v_0\}$ are treated. A vertex v_i dominates a vertex v_j if and only if $\Omega^j \subseteq \Omega^i$. The vertices that are not eliminated compose the subset.

Using the above three criteria, we obtained the sets of elite vertices Γ for all the 200-vertex instances of the test bed. Very good results were obtained as can be observed in the corresponding table of results. One of the criteria tested to construct subsets $\gamma_i \subset V$ was solving a SCP using the Gurobi solver. However, the sets obtained were of poor quality and finally rendered useless. In the work of Hà et al. (2013), this strategy works because, over the generated subset, they apply local search procedures that allow to introduce new vertices to it. In that way, vertices that enable to obtain high-quality or optimal solutions and were not originally included in the subset can become part of it, but in our case such is not possible. Our algorithm must start with a good-quality set.

5.4 ALNS: Pseudocode and Parameters

Unlike *Selector*, we observed that *m-Selector* showed to be sensitive to the starting giant tour given to the algorithm. The single-vehicle version uses a quickly constructed random initial giant tour, σ^{init}, which undergoes a *2-opt* procedure to improve its total length. However, in the multi-vehicle version, results improved when a better starting point was used. Five methods were tested on the twelve benchmark instances to construct σ^{init}: (a) Nearest neighbour; (b) Nearest insertion; (c) Farthest insertion;

(d) Cheapest insertion; and (e) Random. For eleven instances, nearest neighbour or farthest insertion were either the best or second best methods. Best means the method obtained the shortest σ^{init} after undergoing the *2-opt* procedure. As a result of this test, the ALNS version of *m-Selector* incorporates these two construction methods and allows the instance to work with the best choice (second best choice in only one case). For the remaining instance (the one derived from kroA100 where $|V| = 50$), nearest insertion was the best, random was the second best choice, and farthest insertion came in third place. Thus, this one works with its third best. It is worth mentioning that random was the best or second best construction method in four instances.

Algorithm 13 illustrates the ALNS process implemented to solve the m-CTP with simulated annealing as the outer metaheuristic that guides the search. This algorithm is quite similar to the one explained in Chapter 4. The main difference is that it incorporates methods to construct the initial giant tour σ^{init}.

To define an efficient parameter setting, we again used the irace package of López-Ibáñez et al. (2011). A learning set of 50 instances (some derived from TSPLIB and others randomly generated) was designed and *m-Selector* was executed over them. The learning instances used to calibrate the algorithm are, of course, different from the ones used in the benchmark. The resulting set is shown in Table 5.1.

Table 5.1: Values of the ALNS parameters after tuning with irace.

Parameter	Meaning	Value
γ	number of vertices removed in each ALNS iteration (instance size dependent)	$[0.3 \cdot \|V\|, \epsilon \cdot \|V\|]$
ς	segment size for updating probabilities in number of ALNS iterations	75
τ	reaction factor that controls the rate of change of the weight adjustment	0.36
δ	avoids determinism in the SRH	7
ρ	avoids determinism in the WRH	2
κ_1	score for finding a new global best solution	50
κ_2	score for finding a new solution that is better than the current one	20
κ_3	score for finding a new non-improving solution that is accepted	5
β	cooling factor used by simulated annealing	0.99975
ϵ	fixes the upper limit of vertices removed at each iteration	0.50

5.5 Computational Results

5.5.1 Benchmarking Conditions

The testing conditions were kept as the former ones: 30 independent executions of the algorithm per instance, 30,000 ALNS-iterations per execution. The hardware and software platforms are also the same, and so are the definitions of cost values and cover.

There are neither libraries of instances nor reference benchmarks for the m-CTP. Hence, to be able to compare the quality of the results yielded by our method, we

Algorithm 13 : The General Framework of the ALNS with Simulated Annealing

Input: Set V, distance matrix D
Output: T_{best} and $c(T_{\text{best}})$ {Best m-CTP tour and its cost}

$\sigma^{\text{init}} \leftarrow$ Construct Tour(V)
2-opt(σ^{init})
compute $l_0(\sigma^{\text{init}})$ {cost of initial giant tour}
initialize, to the same value, probability P_r^t for each removal operator $r \in R$, and likewise probability P_i^t for each insertion operator $i \in I$.
$t \leftarrow l_0$, {set initial temperature, variable used in probability function}
$l_{\text{current}} \leftarrow l_0$; $\sigma^{\text{current}} \leftarrow \sigma^{\text{init}}$
$UB_{\text{best}} \leftarrow$Search Upper Bound($\sigma^{\text{init}}$) {see Algorithm 8}
$c(T_{\text{best}}) \leftarrow c(T_{\text{current}}) \leftarrow m$-Selector($\sigma^{\text{init}}$) {see Algorithm 5}
$i \leftarrow 1$ {iteration counter}
repeat
  select a removal operator $r \in R$ with probability P_r^t {roulette wheel}
  obtain $\sigma^{\text{new}-}$ by applying r to σ^{current}
  select an insertion operator $i \in I$ with probability P_i^t
  obtain σ^{new} by applying i to $\sigma^{\text{new}-}$
  $UB_{\text{current}} \leftarrow$Search Upper Bound($\sigma^{\text{new}}$)
  if ($UB_{\text{current}} < \alpha \cdot UB_{\text{best}}$) **then**
    $c(T_{\text{new}}) \leftarrow m$-Selector($\sigma^{\text{new}}$)
  else
    $c(T_{\text{new}}) \leftarrow UB_{\text{current}}$
  end if
  if ($UB_{\text{current}} < UB_{\text{best}}$) **then**
    $UB_{\text{best}} \leftarrow UB_{\text{current}}$
  end if
  {decide acceptance of new solution}
  if ($c(T_{\text{new}}) < c(T_{\text{current}})$) **then**
    $c(T_{\text{current}}) \leftarrow c(T_{\text{new}})$; $\sigma^{\text{current}} \leftarrow \sigma^{\text{new}}$
  else
    $p \leftarrow e^{-\dfrac{c(T_{\text{new}}) - c(T_{\text{current}})}{t}}$
    generate a random number $n \in [0, 1]$
    {new solution might be accepted even if it is worse}
    if ($n < p$) **then**
      $c(T_{\text{current}}) \leftarrow c(T_{\text{new}})$; $\sigma^{\text{current}} \leftarrow \sigma^{\text{new}}$
    end if
  end if
  if ($c(T_{\text{new}}) < c(T_{\text{best}})$) **then**
    $c(T_{\text{best}}) \leftarrow c(T_{\text{new}})$; $T_{\text{best}} \leftarrow T_{\text{new}}$
  end if
  $t \leftarrow \beta \cdot t$ {cooling rate set to be very slow}
  if (segment size $= \varsigma$) **then**
    update probabilities using the adaptive weight adjustment procedure
  end if
  $i \leftarrow i + 1$
until (defined number of iterations is met)

performed two different sets of tests. In each set, one of the side restrictions related to bounds p or q is relaxed. The first set considers p constant and $q = +\infty$, whereas the second one establishes $q = 2\varphi + \varrho$ and $p = +\infty$. The considerations for each case are as follows.

Considering p constant and $q = +\infty$. In this set of tests, the quality of the results obtained by our method are compared against the output of both the branch-and-cut method and the meta-heuristic of Hà et al. (2013). In addition, we also compare against the results of the VNS heuristic of Kammoun et al. (2015). Therefore, the instances used are derived from TSPLIB-instances kroX100 | X $\in \{A - D\}$, and kroX200 | X $\in \{A - B\}$ (Reinelt, 1991), and $|V| \in \{25, 50, 100\}$. In total, 12 instances are obtained and each is tested with four different values of $p \in \{4, 5, 6, 8\}$. Hence, there are 48 possible results, with α fixed at some given value.

It is also worth noting that for some instances, tests were conducted using an initial giant tour built with a strategy that is different to the one observed to be the best. This is the reason why some tables of results document tests where the strategy used is documented. This testing was done to learn more about the behaviour of our algorithm.

Considering $q = 2\varphi + \varrho$ and $p = +\infty$. In this set of tests, the quality of the results obtained by our method are compared against the output of the branch-and-price method of Jozefowiez (2014). Therefore, the instances used are derived from TSPLIB-instances kroX150 and kroX200 | X $\in \{A - B\}$ (Reinelt, 1991), and $|V| = 50$. In total, 4 instances are obtained and each is tested with two different values of $\varrho \in \{250, 500\}$. Hence, there are 8 possible results, with α fixed at some given value.

5.5.2 Discussion of Tables of Results

Considering p constant and $q = +\infty$. Tables 5.2, 5.3, and 5.4 offer a comparison of the solution quality and the execution time amongst (1) *m-Selector*; (2) the branch-and-cut method as well as the two-phase hybrid heuristic of Hà et al. (2013); and (3) the VNS of Kammoun et al. (2015). In addition to instance-related information ($|V|, |W|$ and p), the columns of these tables document the following.

- α, the constant by which the best known solution is multiplied when compared against the current feasible solution in order to determine the execution of *m-Selector* to optimality. If the value does not appear, it is assumed equal to the nearest one shown above.
- z, the solution value obtained by the corresponding heuristic. In the case of *m-Selector*, two values are shown. The upper value corresponds to the minimum value obtained in the 30 executions done, while the lower one corresponds to the maximum value. If only one value appears, it means both are equal.
- t, the time (in seconds) needed to find the solution shown on the immediate left column. In the case of *m-Selector*, it is an average time.

- θ, the percentage of deviation from the best-known solution (z_{BKS}), $\theta = 100(z - z_{\text{BKS}})/z_{\text{BKS}}$.
- Opt, the optimal value found by the branch-and-cut method.
- $|\gamma'|$, the size of the set of elite vertices (applies only to Table 5.9).

Tables 5.6 and 5.7 document the benchmark results, using the complete original vertex set. The results obtained using elite sets are reported in Table 5.8. The columns of these tables document the following data (average values are over the 30 runs).

- α, the same meaning defined above.
- $\overline{iter}$, the average number of iterations needed to find the best value.
- $\overline{z}$, the average best value found by the algorithm.
- $\overline{t}$, the average time (in seconds) taken to find the best value.
- $\overline{\theta}$, the average percentage of deviation from the best-known solution $\theta = 100 * (\overline{z} - z_{\text{BKS}})/z_{\text{BKS}}$.
- Found, the number of times, out of 30, the optimal or best-known value was found by the algorithm.
- Percent, the above number expressed in percentage.
- Best Gap, the gap (in percentage) between the optimal or best-known value and the best value found by the algorithm in the 30 runs.

In case the construction method used to build σ^{init} is not the best suited for the given instance, its name is indicated below the name of the instance. Recall that for the instance derived from kroB200 and $|V| = 100$ these results are for the original data set.

Considering $q = 2\varphi + \varrho$ and $p = +\infty$. Table 5.5 offers a comparison of the solution quality and the execution time between *m-Selector* and the branch-and-price method of Jozefowiez (2014). The columns of this table document the same information as explained for Table 5.2, except for ϱ. The benchmark results are documented in Table 5.9. The columns of this table document the same information as explained for Table 5.6, except for the one labelled ϱ.

5.5.3 Comments

The following comments are based on the comparisons presented in Tables 5.2, 5.3, 5.4 and 5.5.

- In Table 5.2, one observes that *m-Selector* finds the optimal value for all the tested instances, and execution times are comparable to the ones of the state-of the-art heuristic most of the times. In Table 5.3, the same can be stated but for the instance derived from kroB200 with $|V| = 100$. Even though it is difficult to perform comparisons of execution times taken from different hardware platforms, some insight can be given by standard benchmarks. Our processor has a SiSoft Sandra Whetstone benchmark score 6.6 times higher than the one used by Hà et al. (2013), and no comparison can be stated for the hardware used by Kammoun et al. (2015), since their paper does not specify it. Results show that the problem difficulty increases with $|V|$, but is fairly insensitive to $|W|$.

  It is worth noting that *m-Selector* is solving two combinatorial optimization problems simultaneously: the optimal selection of the covering vertices and its segmentation into optimal vehicle routes. Furthermore, there is an important difference amongst these three heuristics. The *m-Selector* operator and the heuristic of Hà et al. (2013) are more general procedures since they can introduce the constraint on the route length rather easily. The use they make of the *Split* operator enables this feature. The former is not true for the VNS procedure which would require specific changes in the different operators used in the local search process, and also in the procedure that constructs the initial solution.

  Table 5.4 offers the same type of comparison as Table 5.3, but using sets of elite vertices. In the cases where it was possible to get results using the complete set V, we can compare and state that the algorithm obtains the same solution quality for all the instances considered, and in some cases lower execution times than those obtained using the complete set. For the instance derived from kroB200 with $|V| = 100$, solution quality is slightly inferior to the one of the reference heuristic and execution times notably higher. The longer execution times are a direct consequence of the need to use larger α values. For the instances in which $|V| = 100$, the set reduces roughly to half its size, while for those in which $|V| = 50$ the set size reduces around 20%.

  All in all, in 46 out of the 48 possible results, the best gap obtained is zero. This is to say, the algorithm finds the optimal or best-known value in at least one of the 30 executions tried. However, for the results not included in this remark (kroB200 with $|V| = 100$ and $p \in \{4, 8\}$, Table 5.4), the deviation from the optimum is below 1%.

- Table 5.5 shows that the optimal value is found in all tested instances very quickly.

The following comments are based on the benchmark results presented in Tables 5.6, 5.7, 5.8. Only the last comment pertains to Table 5.9. Within a given instance, results are grouped by α value, so the remarks presented are done considering the group of best results.

- For instances where $|N| = 100$ and $|V| = 25$ the performance is outstanding

since only one iteration is needed to find the optimum value and this occurs in each instance. However, these problems are fairly easy to solve.

- For instances where $|N| = 100$ and $|V| = 50$ the performance is very good. The algorithm finds the optimal value for all the instances in at least three out of the 30 executions. In roughly 40% of the cases, it finds it in all the executions. For the instance derived from kroD100 | $p = 6$, the metaheuristic of Hà et al. (2013) fails to find the optimum. Nonetheless, ours does. The average gap remains relatively low, below 1.7%, but for one instance (kroD100| $p = 4$).

- For instances where $|N| = 200$ and $|V| = 50$ the performance is also very good. A larger alpha value yields solutions of better quality, but, at the same time, it deteriorates the execution time since m-Selector is executed a greater number of times. Also, it shows that the quality of results is influenced by the construction method used to build the initial giant tour. For instance, see the last two blocks of results of the instance obtained from kroB200. The results are of higher quality, and, for some cases, also better execution times are achieved when constructing the σ^{init} with random, which, as a matter of fact, is not the method that yields the best initial sequence.

  Except for one case (instance kroB200| $p = 4$), the average gap remains below 1.5%. Unfortunately, our times are frequently quite larger than the ones of the metaheuristics we are comparing against.

- For instances where $|N| = 200$ and $|V| = 100$ the performance is acceptable, but the remarks regarding the output obtained are less generalisable. For the instance obtained from kroA200, it was possible to explore the behaviour using values of α which are equal to the ones used for smaller instances. However, the instance derived from kroB200 exhibited inoperable times for $\alpha = 1.05$, though the testing was patiently done. Again, we can observe that results improve as α grows, but execution times worsen. For the instance obtained from kroA200, the optimal or best upper bound value is found in at least one execution out of the 30. In this instance, the optimal value is unknown for the last two cases, and our algorithm does not improve the upper bound values known. Furthermore, comparing the two blocks of results for $\alpha = 1.05$, one observes that results are of similar quality, but execution times are worse for the random construction method, which came out as the best choice for this instance. An opposite situation to the one observed in the instance derived from kroB200 with $|V| = 50$, however, in the latter random is not the best choice.

  For the instance derived from kroB200, our algorithm neither finds the optimal values nor does it improve the best-known upper bounds. Average gap values remain between 2 and 5 percent, and execution times are, in general, higher than the ones of the reference metaheuristics. However, these results quite improve when we test this instance with a set of elite vertices. In these tests, a larger α value was used so execution times are higher than with the original set.

- In eleven out of the twelve instances studied, the best gap obtained is zero for all values of p. This is to say, the algorithm finds the optimal value in at least one of the 30 executions tried. However, for the instance not included in this remark (kroB200 with $|V| = 100$), the best gap becomes zero for $p = 5$ and $p = 6$ when a reduced set of vertices is used. For the other two values of p treated, the deviation from the optimum is below 1%.

- It is worth noting that *m*-Selector is solving two NP-hard, combinatorial optimisation problems simultaneously: the selection of the covering vertices and its arrangement in optimal vehicle routes. The execution times reported by Hà et al. (2013) refer only to the solution of the VRP. The covering part is solved via integer programming, and they only mention that the solution of the mathematical model is rapid even for the large instances tested, however, no quantitative measure is given. Regarding the computers used, our machine has a SiSoft Sandra Whetstone benchmark score 6.6 times higher than theirs, and no comparison can be stated for the hardware used by Kammoun et al. (2015) since their paper does not specify it.

- In Table 5.9, one observes that for small values of α the algorithm is able to find the optimal value in a very reasonable amount of time in all tested instances. No general statement can be made as to which problem ($p = +\infty$ or $q = +\infty$) is easier to solve.

5.6 Conclusions

This study proposed a novel solution method to solve the m-CTP. Results, as good as the ones of the state-of-the-art algorithms, are presented. In addition, we introduced a novel way to use the *Split* operator which allows to solve the target problem with a dynamic programming-based algorithm without resorting to the use of additional labels to define and evaluate routes. The considerations made by the the problem allow a quite simple implementation of a modified version of the *Split* algorithm. An additional advantage of using the *Split* algorithm is that the segmentation of the selected vertices into vehicle routes is optimal for the original sequence given.

Computational results were obtained for instances with up to 200 vertices, where $|V| \leq 100$. On the one hand, when $q = +\infty$, for almost all of the instances tested, the solution quality of our results is equal to that of the state-of-the-art heuristic. The *m-Selector* operator missed to find the same value in only two of the results possible, and in these cases the deviation is below 1%. However, the execution time of some results is quite higher. This outcome is very encouraging, and our future research path is to find further means to improve the performance of our algorithm. On the other hand, when $p = +\infty$, in a very short execution time the solution quality of our results equals that of the state-of-the-art exact algorithm.

When solving the CTP, it was possible to obtain good quality solutions using α values of one. Such was not possible when solving the m-CTP, a more difficult problem. Values larger than one were necessary. This has a direct effect on the execution time.

Table 5.2: Comparison of the best values found by *m-Selector*, the branch-and-cut method as well as the hybrid heuristic of Hà et al. (2013), and the VNS procedure of Kammoun et al. (2015) for 100-vertex instances (p constant and $q = +\infty$).

Instance					*m*-Selector-ALNS				Hà et al.				VNS	
Based on	$\|V\|$	$\|W\|$	p	α	z	$\bar{t}$ (s)	θ	Opt	t (s)	z	t (s)	z	t (s)
kroA100	25	75	4	1.03	8479	0.00	0	8479	1.13	8479	0.16	8479	0.016
			5		8479	0.00	0	8479	3.27	8479	0.17	8479	0.016
			6		8479	0.00	0	8479	6.87	8479	0.16	8479	0.013
			8		7985	0.00	0	7985	20.10	7985	0.16	7985	0.014
kroB100	25	75	4	1.03	7146	0.00	0	7146	1.81	7146	0.22	7146	0.004
			5		6901	0.00	0	6901	3.23	6901	0.18	6901	0.005
			6		6450	0.00	0	6450	4.33	6450	0.23	6450	0.004
			8		6450	0.00	0	6450	10.88	6450	0.20	6450	0.004
kroC100	25	75	4	1.03	6161	0.00	0	6161	2.82	6161	0.16	6161	0.004
			5		6161	0.00	0	6161	5.81	6161	0.16	6161	0.004
			6		6161	0.00	0	6161	7.73	6161	0.15	6161	0.004
			8		6161	0.00	0	6161	9.42	6161	0.17	6161	0.004
kroD100	25	75	4	1.03	7671	0.00	0	7671	1.04	7671	0.16	7671	0.020
			5		7465	0.00	0	7465	5.38	7465	0.16	7465	0.022
			6		6651	0.00	0	6651	3.80	6651	0.15	6651	0.015
			8		6651	0.00	0	6651	12.85	6651	0.16	6651	0.014
kroA100	50	50	4	1.03	10271	0.00	0	10271	9.91	10271	0.80	10271	0.022
			5		9220/9742	0.54	0	9220	12.36	9220	0.78	9220	0.017
			6		9130	0.00	0	9130	24.79	9130	0.81	9130	0.023
			8		9130	0.00	0	9130	203.93	9130	0.81	9130	0.018
kroB100	50	50	4	1.03	10107/10754	4.14	0	10107	16.63	10107	0.62	10107	0.012
			5		9723/9976	2.94	0	9723	84.08	9723	0.64	9723	0.009
			6		9382/9529	4.03	0	9382	162.24	9382	0.58	9382	0.016
			8		8348/8379	0.74	0	8348	76.06	8348	0.58	8348	0.016
kroC100	50	50	4	1.03	11372	0.00	0	11372	8.12	11372	0.64	11372	0.028
			5		9900/9976	1.37	0	9900	13.28	9900	0.67	9900	0.013
			6		9895	0.00	0	9895	56.91	9895	0.67	9895	0.017
			8		8699	0.00	0	8699	8.47	8699	0.65	8699	0.007
kroD100	50	50	4	1.03	11606/12161	0.45	0	11606	9.34	11606	0.93	11606	0.021
			5		10770/11357	0.47	0	10770	29.32	10770	0.85	10770	0.263
			6		10525/10764	0.33	0	10525	281.28	10680	0.82	10525	0.026
			8		9361/9566	0.00	0	9361	110.62	9361	0.93	9361	0.028

Table 5.3: Comparison of the best values found by *m-Selector*, the branch-and-cut method as well as the hybrid heuristic of Hà et al. (2013), and the VNS procedure of Kammoun et al. (2015) for 200-vertex instances (p constant and $q = +\infty$).

Instance					*m*-Selector-ALNS			Hà et al.				VNS	
Based on	$\|V\|$	$\|W\|$	p	α	z	$\bar{t}$ (s)	θ	Opt	t (s)	z	t (s)	z	t (s)
kroA200	50	150	4	1.003	11550/11612	0.14	0	11550	82.21	11550	0.89	11550	0.024
			5		10407/10769	2.81	0	10407	340.58	10407	0.87	10407	0.025
			6		10068/10362	3.76	0	10068	1075.80	10068	0.89	10068	0.023
			8		8896	0.00	0	8896	153.40	8896	0.94	8896	0.063
kroB200	50	150	4	1.05	11175/11833	3.45	0	11175	166.00	11175	0.91	11175	0.177
			5	1.003	10502/10533	0.14	0	10502	1114.67	10502	0.90	10502	0.020
			6		9799	0.00	0	9799	1273.97	9799	0.91	9799	0.018
			8		8846/9124	0.78	0	8846	629.77	8846	0.87	8846	0.072
kroA200	100	100	4	1.05	11885	0.00	0	11885	4593.91	11885	2.89	11885	0.154
			5		10234/10400	0.23	0	10234	1440.13	10234	2.82	10234	0.058
			6		10020/10075	4.78	0	-	7200.13	10020	2.92	10020	0.026
			8		9093	0.00	0	-	7200.12	9093	2.92	9093	0.270
kroB200	100	100	4	1.2	-	600.00	-	18370	6614.98	18370	15.03	18370	0.057
			5		-	600.00	-	15876	1471.99	15876	15.61	15876	0.076
			6		-	600.00	-	-	7200.08	14926	14.83	14867	0.05
			8		-	600.00	-	-	7200.09	13137	15.68	13137	0.026

Table 5.4: Comparison of the best values found by *m-Selector*, when using sets of elite vertices, against the branch-and-cut method as well as the hybrid heuristic of Hà et al. (2013), and the VNS procedure of Kammoun et al. (2015) for 200-vertex instances (p constant and $q = +\infty$).

Instance						*m*-Selector-ALNS			Hà et al.				VNS	
Based on	$\|V\|$	$\|\gamma'\|$	$\|W\|$	p	α	z	$\bar{t}$ (s)	θ	Opt	t (s)	z	t (s)	z	t (s)
kroA200	50	40	150	4	1.003	11550/11612	0.11	0	11550	82.21	11550	0.89	11550	0.024
				5		10407/10880	0.92	0	10407	340.58	10407	0.87	10407	0.025
				6		10068/10362	1.55	0	10068	1075.80	10068	0.89	10068	0.023
				8		8896	0.00	0	8896	153.40	8896	0.94	8896	0.063
kroB200	50	42	150	4	1.05	11175/11646	3.59	0	11175	166.00	11175	0.91	11175	0.177
				5	1.003	10502/10533	1.57	0	10502	1114.67	10502	0.90	10502	0.020
				6		9799/10057	0.37	0	9799	1273.97	9799	0.91	9799	0.018
				8		8846/9124	0.68	0	8846	629.77	8846	0.87	8846	0.072
kroA200	100	51	100	4	1.05	11885	0.00	0	11885	4593.91	11885	2.89	11885	0.154
				5		10234/10655	1.44	0	10234	1440.13	10234	2.82	10234	0.058
				6		10020/10075	5.34	0	-	7200.13	10020	2.92	10020	0.026
				8		9093	0.00	0	-	7200.12	9093	2.92	9093	0.270
kroB200	100	49	100	4	1.2	18424/18921	23.33	0.29	18370	6614.98	18370	15.03	18370	0.057
				5		15876/16408	45.83	0	15876	1471.99	15876	15.61	15876	0.076
				6		14867/15422	35.54	0	-	7200.08	14926	14.83	14867	0.05
				8		13232/13444	6.11	0.72	-	7200.09	13137	15.68	13137	0.026

Table 5.5: Comparison of the best values found by *m-Selector*, and the branch-and-price method of Jozefowiez (2014) ($q = 2\varphi + \varrho$ and $p = +\infty$).

Instance					*m*-Selector-ALNS			Jozefowiez	
Based on	$\|V\|$	$\|W\|$	ϱ	α	z	$\bar{t}$ (s)	θ	Opt	t (s)
kroA150	50	100	250	1.003	13654/13680	0.37	0	13654	3072
kroA150			500	1.05	13654/13680	1.69	0	13654	5598
kroB150	50	100	250	1.003	9479/9901	1.64	0	9479	1673
kroB150			500		9479/9731	1.53	0	9479	4126
kroA200	50	150	250	1.05	11022/11024	0.13	0	11022	12919
kroA200			500		11022/11024	0.11	0	11022	13812
kroB200	50	150	250	1.003	9362/9630	3.63	0	9362	5403
kroB200			500		9360/9570	6.93	0	9360	11308

Table 5.6: Results obtained by *m-Selector* for 100-vertex instances when $q = +\infty$.

Instance Based on	$\|V\|$	$\|W\|$	p	α	*m*-Selector-ALNS $\overline{iter}$	$\overline{z}$	$\overline{t}$ (s)	$\overline{\theta}$	Found	Percent	Best Gap
kroA100	25	75	4	1.03	1	8479	0.00	0	30	100	0
			5		1	8,479	0.00	0	30	100	0
			6		1	8,479	0.00	0	30	100	0
			8		1	7,985	0.00	0	30	100	0
kroB100	25	75	4	1.03	1	7,146	0.00	0	30	100	0
			5		1	6,901	0.00	0	30	100	0
			6		1	6,450	0.00	0	30	100	0
			8		1	6,450	0.00	0	30	100	0
kroC100	25	75	4	1.03	1	6,161	0.00	0	30	100	0
			5		1	6,161	0.00	0	30	100	0
			6		1	6,161	0.00	0	30	100	0
			8		1	6,161	0.00	0	30	100	0
kroD100	25	75	4	1.03	1	7,671	0.00	0	30	100	0
			5		1	7,465	0.00	0	30	100	0
			6		1	6,651	0.00	0	30	100	0
			8		1	6,651	0.00	0	30	100	0
kroA100	50	50	4	1.03	1	10,271	0.00	0	30	100	0
			5		8,141	9,294	0.54	0.80	24	80	0
			6		99	9,130	0.00	0	30	100	0
			8		29	9,130	0.00	0	30	100	0
kroB100	50	50	4	1.03	14,635	10,264	4.14	1.55	12	40	0
			5		13,121	9,776	2.94	0.55	11	36	0
			6		14,224	9,419	4.03	0.39	4	13	0
			8		4,383	8,349	0.74	0.02	28	93	0
kroC100	50	50	4	1.03	1	11,372	0.00	0	30	100	0
			5		9,846	9,916	1.37	0.16	10	33	0
			6		1	9,895	0.00	0	30	100	0
			8		40	8,699	0.00	0	30	100	0
kroD100	50	50	4	1.03	3,227	12,070	0.45	4.00	4	13	0
			5		3,779	10,948	0.47	1.65	5	16	0
			6		2,672	10,688	0.33	1.55	3	10	0
			8		59	9,368	0.00	0.07	29	96	0
kroD100	50	50	4	1.1	8,163	11,868	3.23	2.26	13	43	0
			5		3,674	10,905	0.95	1.25	10	33	0
			6		6,521	10,622	2.09	0.92	12	40	0
			8		56	9,361	0.00	0	30	100	0

Table 5.7: Results obtained by *m-Selector* for 200-vertex instances when $q = +\infty$.

Instance Based on	$\|V\|$	$\|W\|$	p	α	*m*-Selector-ALNS $\overline{iter}$	$\overline{z}$	$\overline{t}$ (s)	$\overline{\theta}$	Found	Percent	Best Gap
kroA200	50	150	4	1.05	1,482	11,598	0.38	0.42	7	23	0
			5		11,775	10,444	5.69	0.36	14	46	0
			6		12,486	10,095	5.79	0.27	8	26	0
			8		1	8,896	0.00	0	30	100	0
kroA200	50	150	4	1.003	712	11,606	0.14	0.49	3	10	0
			5		10,658	10,530	2.81	1.18	2	6	0
			6		13,403	10,146	3.76	0.77	3	10	0
			8		1	8,896	0.00	0	30	100	0
kroB200	50	150	4	1.05	3,154	11,531	3.45	3.19	6	20	0
			5		181	10,506	0.29	0.04	26	86	0
			6		6,876	9,936	9.85	1.40	8	26	0
			8		4,010	8,944	7.46	1.11	18	60	0
kroB200	50	150	4	1.003	3,860	11,647	1.82	4.22	0	0	3.95
			5		499	10,516	0.14	0.13	16	53	0
			6		2,398	10,029	0.72	2.35	1	3	0
			8		3,236	9,030	0.78	2.08	9	30	0
kroB200	50	150	4	1.003	2,958	11,631	0.58	4.08	2	6	0
Random			5		796	10,510	0.24	0.08	22	73	0
			6		1	9,799	0.00	0	30	100	0
			8		2,753	8,923	0.81	0.87	20	66	0
kroA200	100	100	4	1.05	1	11,885	0.00	0	30	100	0
			5		93	10,394	0.23	1.56	1	3	0
			6		353	10,073	4.78	0.53	1	3	0
			8		1	9,093	0.00	0	30	100	0
kroA200	100	100	4	1.003	1	11,885	0.00	0	30	100	0
			5		100	10,400	0.09	1.62	0	0	1.62
			6		1	10,075	0.00	0.55	0	0	0.55
			8		1	9,093	0.00	0	30	100	0
kroA200	100	100	4	1.05	1	11,885	0.00	0	30	100	0
Random			5		1,127	10,394	188.41	1.56	1	3	0
			6		1,538	10,069	30.38	0.49	2	6	0
			8		0	9,093	0.00	0	30	100	0
kroB200	100	100	4	1.05	6,927	18,789	94.64	2.28	0	0	2.15
			5		4,779	16,601	133.48	4.57	0	0	2.16
			6		8,533	15,415	304.78	3.28	0	0	2.39
			8		2,316	13,391	5.34	1.93	0	0	1.66
kroB200	100	100	4	1.003	9,181	19,062	26.46	3.77	0	0	2.00
			5		1	16,682	0.00	5.08	0	0	5.08
			6		9,704	15,585	35.33	4.42	0	0	3.32
			8		2,394	13,400	4.08	2.00	0	0	1.60
kroB200	100	100	4	1.003	6,713	19,007	19.46	3.47	0	0	2.47
Random			5		1,645	17,056	1.21	7.43	0	0	4.52
			6		9,096	15,615	24.75	4.62	0	0	2.60
			8		12,055	13,491	98.64	2.69	0	0	1.66

Table 5.8: Results obtained by *m-Selector* for sets of elite vertices when $q = +\infty$.

Set	$\|V\|$	$\|W\|$	p	α	*m*-Selector-ALNS						Best Gap
					$\overline{iter}$	$\overline{z}$	$\overline{t}$ (s)	$\overline{\theta}$	Found	Percent	
kroB200_1	49	100	4	1.2	13,885	18,592	23.33	1.21	0	0	0.29
Random			5		14,276	16,129	26.42	1.59	0	0	1.13
			6		13,350	15,204	35.54	2.27	3	10	0
			8		2,872	13,438	6.11	2.29	0	0	0.72
kroB200_2	49	100	4	1.2	15,164	18,591	43.94	1.20	0	0	0.29
NN			5		16,254	16,124	45.83	1.56	1	3	0
			6		16,573	15,151	52.60	1.91	6	20	0
			8		954	1,3444	2.82	2.34	0	0	2.34
kroB200_3	48	100	4	1.2	10,035	18,682	173.10	1.70	0	0	1.3
Random			5		14,146	16,280	278.53	2.54	0	0	0.25
			6		13,183	15,447	306.34	3.49	0	0	2.06
			8		4,153	13,466	124.63	2.50	0	0	0.81

Table 5.9: Results obtained by *m-Selector* when $p = +\infty$.

Instance Based on	$\|V\|$	$\|W\|$	ϱ	α	*m*-Selector-ALNS						Best Gap
					$\overline{iter}$	$\overline{z}$	$\overline{t}$ (s)	$\overline{\theta}$	Found	Percent	
kroA150	50	100	250	1.003	5055	13677	0.37	0.17	2	6	0
kroA150			250	1.05	3770	13674	0.93	0.15	5	16	0
kroA150			500	1.003	2577	13677	0.19	0.37	0	0	0.01
kroA150			500	1.05	4684	13675	1.69	0.15	4	13	0
kroB150	50	100	250	1.003	10619	9544	1.64	0.69	14	46	0
kroB150			250	1.05	10886	9479	3.09	0.00	29	96	0
kroB150			500	1.003	10302	9524	1.53	0.47	14	46	0
kroB150			500	1.05	7988	9479	1.86	0.00	29	96	0
kroA200	50	150	250	1.003	0	11024	0.00	0.02	0	0	0.02
kroA200			250	1.05	445	11024	0.13	0.02	1	3	0
kroA200			250	1.1	0	11024	0.00	0.02	0	0	0.02
kroA200			500	1.003	0	11024	0.00	0.02	0	0	0.02
kroA200			500	1.05	381	11024	0.11	0.02	1	3	0
kroA200			500	1.1	0	11024	0.00	0.02	0	0	0.02
kroB200	50	150	250	1.003	9477	9428	3.63	0.70	13	43	0
kroB200			250	1.05	12062	9407	28.01	0.48	13	43	0
kroB200			250	1.1	5237	9380	49.62	0.19	19	63	0
kroB200			500	1.003	9531	9406	6.93	0.49	3	10	0
kroB200			500	1.05	12992	9366	21.56	0.06	18	60	0
kroB200			500	1.1	8365	9361	45.66	0.01	19	63	0

Chapter 6

Solving the Orienteering Problem with *Selector*

Contents

6.1 Introduction

This chapter discusses the specific practical details of the approach developed to solve the *Orienteering Problem* (OP). Feillet et al. (2005) situate this problem, together with other variants of the TSP, under the family they named Travelling Salesman Problems with Profits. The unifying concept in this family is the notion of collecting a profit when visiting a vertex and the use of only one vehicle. In the OP, a set of locations (vertices in a graph when modelled) is given, each with an associated profit, and the travel between locations involves a known cost. The problem calls to maximise the collected profit subject to a constraint on the total travel cost allowed. Then, not all the given locations can be visited since the available time (or distance) is limited. Unlike the

covering problems treated in previous chapters, the OP and its multi-vehicle version, the *Team Orienteering Problem* (TOP), have been thoroughly studied, and extensive literature has been published together with standard libraries of data sets. The survey of Vansteenwegen et al. (2011a) and the chapter written by Archetti et al. (2014) illustrate this fact.

The OP, whose name originates from a treasure-hunt game in which players must collect scores in a preset time frame by visiting control points, may be seen as a bicriteria problem with two opposite objectives: one pushing the vehicle to travel in order to collect profit, and the other one encouraging it to minimise the travel cost with the option of skipping vertices. It is a combination of vertex selection and determination of the shortest elementary cycle among the selected vertices. In other words, a combination of the *Knapsack Problem* and the TSP. Nevertheless, most researchers address it as a single-criterion version where one of the objectives is constrained with a specified bound value. In the case of the OP, the travel cost objective is treated as a constraint, hence, the goal is to find a circuit that maximises the collected profit but whose length does not exceed a given bound.

Golden et al. (1987) and Laporte and Martello (1990) demonstrated that it is an NP-hard combinatorial optimisation problem. In the literature, the OP has been given several names: the *Selective Traveling Salesman Problem* (STSP) as in Laporte and Martello (1990), Gendreau et al. (1998a), and Thomadsen and Stidsen (2003); the *Maximum Collection Problem* as in Kataoka and Morito (1988); and the *Bank Robber Problem* as in Awerbuch et al. (1998).

The remainder of the chapter is organised as follows. Section 6.2 formally presents the problem, provides an insight of the numerous applications reported for the OP, and explains different solution approaches that have been presented in the literature. The solution approaches that served as a reference against which we compare the quality of our results are explained in more detail. Sections 6.3-6.5 explain the solution method proposed in this thesis to solve the OP. Section 6.6 documents and discusses the results obtained. Finally, Section 6.7 reports the contribution of this work and provides conclusions.

6.2 The Orienteering Problem

6.2.1 Formal Definition

The OP can be formally defined by considering a complete, undirected graph $G = (V, E)$ where $V = \{v_0, v_1, \ldots, v_{n-1}\}$ defines its vertex set which represents the n profit-collecting points. Vertex v_0 designates the depot, and set $E = \{(v_i, v_j) | v_i, v_j \in V, i < j\}$ defines the edge set. Let p_i denote the non-negative profit associated with each vertex $v_i \in V$ ($p_0 = 0$). Each vertex v_i is defined by its Euclidean coordinates, so let d_{ij} be the distance associated with each edge $(i, j) \in E$, and $D = (d_{ij})$ the distance matrix that satisfies the triangle inequality. The OP consists of determining a maximal profit Hamiltonian tour over a subset of V, which includes v_0 as the starting point, while satisfying the given travel cost limit $L_{\max}$. We assume all

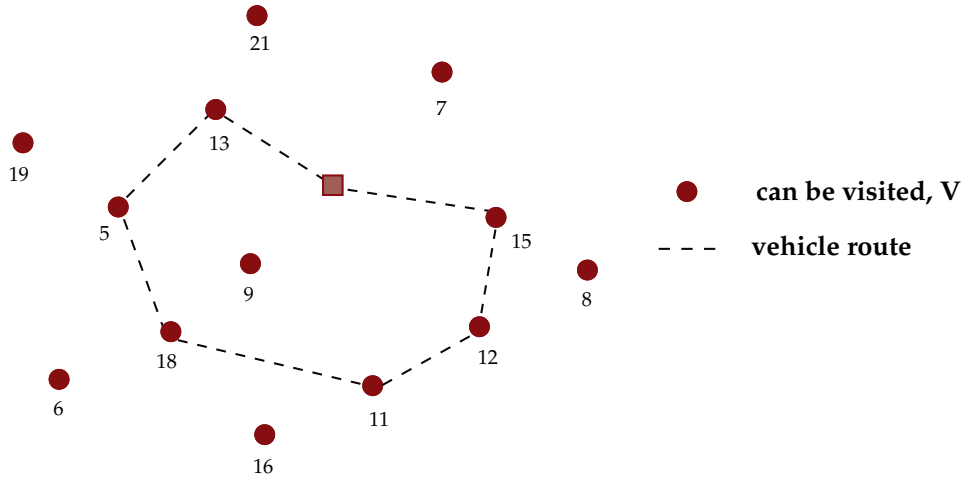

Figure 6.1: Example of OP tour.

profits are integer, and interpret distances as travel costs. Figure 6.1 depicts a feasible OP tour. The square represents the depot. The number next to the vertices that can be visited indicates the profit gained upon visitation of the vertex.

In many studies, the OP is defined in terms of a path to be found between a start and a goal point rather than a circuit. The difference is innocuous. The two endpoints of the sought path may coincide or adding a dummy arc from the destination to the origin of the path makes the two problems equivalent.

6.2.2 Applications Reported in the Literature

A number of applications in logistics, tourism, vehicle routing, production scheduling and other fields have been modelled as orienteering problems, which explains its wide study. Tsiligirides (1984) and Ramesh and Brown (1991) proposed the OP to model the situation of a salesperson who wants to keep the total travelled time within a limit, but wants to maximise total sales. The salesperson knows the amount of sales to expect from each customer. A related but much more complex problem has recently been presented by Tricoire et al. (2010): the *Multi-Period Orienteering Problem with Multiple Time Windows* (MuPOPTW), a new routing problem combining objective and constraints from the OP and TOP, constraints from standard vehicle routing problems, and original constraints from a real-world application. Field workers and sales representatives can use a tool based on this model to schedule their customer visits for a certain planning horizon. They apply a variable neighborhood search algorithm to solve it.

In logistics, Golden et al. (1987) presented the fuel delivery problem, and used the OP to represent the first phase of the problem. A company has to provide heating fuel to a large number of customers on a daily basis, and must maintain the customers' fuel supply at a certain level at all times. This forecasted supply level can set a measure of

urgency of service which translates into a score. A primary goal is to select a subset of customers who urgently require a delivery and are clustered in such a way that an efficient truck path can be constructed. A similar situation is solved by Hemmelmayr et al. (2009) for the Austrian Red Cross, entity that needs to deliver blood supply to hospitals. Their solution approach uses a basic heuristic that is driven by blood inventory levels at the hospitals. Each day, those hospitals that need to receive a delivery on that day, because they would otherwise experience a stockout the next day, will be visited.

Tourism is an area where the OP has found wide applicability. Vansteenwegen and Van Oudheusden (2007) presented the *Tourist Trip Design Problems* (TTDP) in which the OP is the simplest form of the TTDP. Due to time and budget restrictions, tourists must select what they believe to be the most interesting sites to visit. The TTDP solve the task of designing a feasible plan in order to visit the desired attractions in the available time span. Though the OP makes no provision for the modelling of the time spent visiting the selected site, this time can easily be modelled as part of the travel time to reach (or depart from) the site. Typically, half of the visiting time is added to the travel time of all incoming arcs and the other half is added to the outgoing arcs. Nowadays, mobile tourist guide applications are available, refer to Souffriau et al. (2008) and Vansteenwegen et al. (2011b). Gavalas et al. (2014) published a survey in which models, algorithmic approaches and methodologies concerning tourist trip design problems are discussed.

Wang et al. (2008) considered the *Generalised Orienteering Problem* (GOP) to model tourist plans that suggest visiting combinations of tourist attractions. This problem differs from the OP in the objective function considered. While in the OP the profit values associated with each vertex are added to obtain the total collected profit, in the GOP, this total is a nonlinear function of the visited vertices. This function is applied in the sense that attractions are variations on a certain theme and in order to really appreciate the series, it is preferable to visit all, or low valued attractions become more appealing when visited in combination with others.

Schilde et al. (2009) modelled a tourist's different interests when selecting attraction sites with a multi-objective OP. Their solution method is an adaptation of the Pareto ant colony optimisation (ACO) algorithm, and it also includes variable neighborhood search (VNS), which is extended to the multi-objective case. Both methods are hybridized with path relinking procedures. They solve real-world instances from different Austrian regions and from the cities of Vienna and Padua together with library instances.

Ilhan et al. (2008) reported an application of inventory management using the OP. A US car manufacturer must reimburse its suppliers for the inventory at hand when design changes lead to inventory becoming obsolete. It is in the manufacturer's best interest to audit the inventory claims of the suppliers so that they only pay for actual inventory. With a limited number of auditors, the car manufacturer must determine which suppliers to visit to maximise the recovered claims. A recovered claim is the difference between the value of the inventory claimed by the supplier and the audited inventory value. To model the situation they defined the *Orienteering Problem with*

Stochastic Profits (OPSP) in which normally distributed random profits are associated with the vertices. The goal is to maximise the probability of collecting more than a prespecified target profit level within a prespecified time limit. They developed an exact solution approach and a bi-objective genetic algorithm to solve the OPSP.

Wang et al. (2008) suggested a military application of the OP. The expedition of an unmanned aircraft or submarine involved in surveillance activities is constrained by its fuel supply, so it needs to select the best sites to visit or photograph. Thomadsen and Stidsen (2003) studied a variant of the STSP, the *Quadratic STSP* (QSTSP) where each pair of vertices has an associated profit which can be gained only if both vertices are visited. The QSTSP emerges as a subproblem when constructing hierarchical ring networks.

6.2.3 Solution Approaches Reported in the Literature

Since the 1980s, diverse solution methods have been developed for the OP. Amongst the exact solution approaches that have been suggested one finds the following. Laporte and Martello (1990) solved small, randomly-generated problems containing up to 20 points using a branch-and-bound method while Ramesh et al. (1992) used Lagrangian relaxation within a branch-and-bound procedure to solve also randomly-generated problems that contain as many as 150 points. On the other hand, more efficient solution procedures use a branch-and-cut scheme aided by heuristics. Gendreau et al. (1998a) proposed a branch-and-cut algorithm to solve randomly-generated instances of up to 300 locations. In order to provide an initial feasible solution, this procedure uses two new heuristics, which make use of the already known GENIUS composite heuristic for the TSP (Gendreau et al., 1992). Both heuristics perform very well, but they do not introduce any major new concept. Fischetti et al. (1998) solved TSPLIB-derived instances involving up to 500 locations using also a branch-and-cut procedure.

A number of approximate methods have also been proposed. The following are some of the most relevant studies published. Studies that solved only small instances ($n \leqslant 66$) are firstly presented. Tsiligirides (1984) solved the OP by using both a stochastic heuristic based on Monte Carlo techniques and a deterministic heuristic based on the Wren and Holliday (1972) vehicle-routing procedure. He contributed a set of instances ($21 \leq n \leq 33$) that are still used today to compare the performance of newly proposed methods. Optimal values for these instances are known and most methods presented nowadays are able to find them. Golden et al. (1987) presented a center-of-gravity heuristic based on the Knapsack Problem which Golden et al. (1988) later improved by incorporating Tsiligirides's randomization concept along with learning capabilities. Keller (1989) modified his multi-objective vending problem heuristic to solve the OP. Ramesh and Brown (1991) developed a four-phase heuristic where they consider a general cost function rather than the Euclidean cost function assumed in the former approaches. They also use a tabu list to store the computed paths in order to avoid them, but make no reference of their method as a tabu search. Wang et al. (1995) applied an artificial neural network method to solve the OP, whereas Chao et al. (1996b) exploited the underlying geometry of the problem and proposed

a five-step heuristic which considers only vertices that can be reached. These vertices lie within an ellipse whose foci are the start and end vertices, and $L_{\max}$ is the length of the major axis. The algorithm implements an efficient initialisation step, three improvement steps and one diversification step. This heuristic clearly outperforms the algorithms of Tsiligirides (1984), Golden et al. (1987), Golden et al. (1988), Keller (1989) and Ramesh and Brown (1991). Chao et al. (1996b) proposed additional sets of testing instances ($n = 64$ or $n = 66$) which have been used to test the procedures recently published as the one of Schilde et al. (2009) (explained in Section 6.2.2). The instances proposed by Tsiligirides and Chao are distributed in sets. All the instances in a set are based on the same graph, and they only differ in the value given to the tour limit, $L_{\max}$.

To solve larger instances three solution methods have been published in the literature: Gendreau et al. (1998b), Vansteenwegen et al. (2009), and Campos et al. (2014). The first paper describes a tabu search heuristic that yields near-optimal solutions for randomly-generated instances with up to 300 vertices. This heuristic iteratively inserts clusters of locations in the tour or removes a chain of locations. Compared to previous methods, this algorithm reduces both the probability of getting trapped in a local optimum, and the probability of including locations with a high score that are far from the current tour. The second paper explains a guided local search (GLS) metaheuristic that successfully solves a subset of the TSPLIB-derived instances (up to 400 points) studied by Fischetti et al. (1998). This algorithm is the one used to solve the TTDP explained in Section 6.2.2. The state-of the-art metaheuristic is the GRASP with path relinking of Campos et al. (2014), which also solves the same instances proposed by Fischetti et al. (1998), but obtains better quality solutions than those of the GLS of Vansteenwegen et al. (2009). The procedures of Vansteenwegen et al. (2009), and Campos et al. (2014) were also tested on the small instances of Tsiligirides (1984) and Chao et al. (1996b).

Considering the methods explained, the Pareto-ACO and Pareto-VNS heuristics of Schilde et al. (2009), and the GRASP of Campos et al. (2014) outperform both the heuristics of Chao et al. (1996b) and the GLS of Vansteenwegen et al. (2009). The solution quality of the Pareto-ACO and Pareto-VNS heuristics is very similar to the one obtained by the GRASP with path-relinking, though the Pareto heuristics were not tested with standard large-sized instances.

Feillet et al. (2005) presented an in-depth explanation of some of the works previously mentioned. Their focus is on single-vehicle problems with profits, whereas Vansteenwegen et al. (2011a) published a very detailed survey for both the OP and the TOP, its variants, and its applications. The chapter of Archetti et al. (2014) on vehicle routing with profits presents the complete family of problems the OP belongs to.

6.2.4 Solution Approaches Used as Reference

The quality of our results was compared against the output of the branch-and-cut method of Fischetti et al. (1998), and against the results of both the GLS-based heuris-

tic of Vansteenwegen et al. (2009) and the GRASP heuristic with path-relinking of Campos et al. (2014).

The algorithm of Fischetti et al. (1998) is based on several families of valid inequalities. They introduce a family of cuts, called conditional cuts, which can cut off the optimal OP solution, and propose an effective way to use them within the overall branch-and-cut framework. They also adapt and simplify the heuristic of Ramesh and Brown (1991) in order to compute a lower bound in the root node of the branch-and-cut search tree, and quickly tighten the bounds with valid inequalities all along the search tree. Their method requires about five hours of CPU time to obtain the optimum in some of their largest instances. Fischetti et al. (1998) also contributed by providing a library of benchmark instances derived from the OP, VRP and TSP literature. They defined three different ways to generate the profit of a vertex[1].

GLS, developed by Voudouris and Tsang (2003), is a penalty-based metaheuristic that sits on top of other local search algorithms, with the aim of improving their efficiency and robustness. GLS penalises, based on a utility function, unwanted solution features during each local search iteration. The penalty augments the objective function during every iteration. In this way the solution procedure may escape from local optima and allow the search to continue. The version of Vansteenwegen et al. (2009) is a composite heuristic which firstly constructs an initial solution using the greedy construction heuristic of Chao et al. (1996b). Afterwards, two local search heuristics try to increase the total score of the solution. One of this pair of heuristics seeks to insert new locations into a tour in the position that yields the least travel cost. The order in which the locations are considered for insertion is based on an "appropriateness" measure computed for the tour. The second heuristic seeks to replace an included location by a non-included one with a higher score. Again, the locations not yet included are ranked by their appropriateness. This replacing heuristic works under a guided local search mechanism in order to improve its performance, and it is the one that contributes the most to the enhancement of the solution quality. The heuristics perform a maximum of 1,000 iterations. Finally, a third heuristic tries to reduce the travel cost between the previously selected locations. In this heuristic, 2-*opt* is used as the local search mechanism, and it also uses guided local search to improve its performance. No randomness appears in the GLS algorithm.

GRASP, developed by Feo and Resende (1995), is a multi-start metaheuristic for combinatorial optimisation problems, in which each iteration consists of two phases: construction and local search (improvement). The construction phase builds a feasible solution, whose neighborhood is investigated until a local optimum is found during the local search phase. The best overall solution is kept as the result. In the context of GRASP, path-relinking (PR) is a form of intensification (Laguna and Marti, 1999). Starting from a GRASP solution, it tries to find a path between this solution and a chosen elite solution which serves as a guide. Campos et al. (2014) explore four construction methods and combine two neighborhoods in the local search phase. One is based on the exchange of a vertex in the tour for another one not in the tour, while

[1]All the benchmark instances mentioned are available at www.mech.kuleuven.be/cib/op

meeting the constraint on the tour length. After a one-to-one exchange is performed, the algorithm tries the second neighborhood: an insertion move in which a vertex not currently in the tour is considered to be added to it. After each exchange, as many insertions as $L_{\max}$ allows are considered. Insertions occur at the best place possible. Both neighborhoods are applied to all the vertices in the tour. A *2-opt* neighborhood, both at the start of the improvement phase and at the end, is used to further improve the solution. All the different solutions obtained undergo a relinking post-processing.

6.3 Application of *Selector* to Solve the OP

Gendreau et al. (1998b) explain why, despite its apparent simplicity, it is rather difficult to design good heuristics for the OP. Profits and distances are independent and a good solution with respect to one criterion is often unsatisfactory with respect to the other, so selecting the vertices that are part of the optimal solution is difficult. Simple construction and improvement heuristics, though rather quick, may direct the algorithm in undesirable directions. As a consequence, it might happen that large portions of the search space remain unexplored, and also previous wrong decisions are not corrected.

The main challenge we faced when trying to solve the OP using the *Selector* operator was how to control the enormous proliferation of labels. In the CTP, it is desirable to skip vertices in order to reduce the travel cost, whereas in the OP, it is desirable to visit vertices to increase the profit. The objective, then, favors the label extension, and every visited vertex generates a label. The search space is much larger in the OP. This high proliferation of labels obliged to augment the standard mean—dominance rules—this kind of algorithm provides for label growth control. However, when adding the measures to control label growth, it was necessary to reach a compromise between optimality of the decoding and search time efficiency. In spite of these additions, the core of the *Selector* operator and the solution method applied remain the same.

In order to apply the designed algorithm to the solution of the OP, one needs to define: the structure of the label, the dominance rule, and the steps to be taken when a label is iteratively extended trying to reach the immediate successor vertex. Hereupon the features of the basic *Selector* algorithm are described, and at a later section, the performance enhancements aimed at controlling the combinatorial explosion of labels are presented.

6.3.1 Label Definition

A label represents an elementary path that starts at the origin vertex σ_0 and has considered vertices up to the one where the label is stored. In this algorithm, a label is a quadruplet $\lambda = [\pi, \nu, \zeta, \mu]$ that stores data which is useful for the decision-making at different stages of the algorithm.

i. π, the profit collected from the depot σ_0 to vertex σ_ν, which corresponds to the sum of the profit values of the vertices included in the label.

ii. ν, the last vertex visited in the partial tour (site where the label is stored).

iii. ζ, the travel cost of the partial elementary tour from vertex σ_0 to vertex σ_ν. This corresponds to the sum of the weights of the edges included in the label.

iv. μ, an upper bound on the total profit of the tour. This value corresponds to the sum of the profit already collected from the origin vertex to vertex σ_ν (value stored in π), plus an upper bound of the profit that can be collected from the adjacent successor of σ_ν, $\sigma_{\nu+1}$, to the goal vertex. In a later section, an explanation on how this upper bound is computed is provided.

6.3.2 Dominance Test

Such test is performed before storing a label in order to prune those that are not useful, since there is at least one other label with a similar trajectory, but which offers a better or equal profit value and a smaller or equal resource consumption. A label $\lambda_1 = [\pi_1, \nu_1, \zeta_1, \mu_1]$ dominates a label $\lambda_2 = [\pi_2, \nu_2, \zeta_2, \mu_2]$ with $\lambda_1 \neq \lambda_2$ if and only if:

i $\nu_1 = \nu_2$

ii $\pi_1 \geq \pi_2$

iii $\zeta_1 \leq \zeta_2$

Two labels are comparable only when arriving at the same vertex, so that they share a similar past trajectory and similar possibilities for their future paths. A label eliminates another one when its collected profit value is greater or equal to the value of the second one and its travel cost is lower or equal.

6.3.3 Extension of a Label

In the *Selector* algorithm, the extension of a label $\lambda = [\pi, \nu, \zeta, \mu]$ from vertex σ_ν to a successor vertex σ_k with $k > \nu$ implies that one of two possible operations is performed: visit vertex σ_k or skip vertex σ_k. At every extension, feasibility must be maintained so we need to ensure that the constraint on the tour length is met. Therefore, when the total tour length is within the limit, the label λ is extended by visiting vertex σ_k. Whereas, when this restriction is not met, extension is done by skipping σ_k.

The algorithm to extend a label will now be explained. However, one of its steps is the computation and application of an upper bound. The procedure to compute this bound is discussed in the next section. Algorithm 14 shows the specific implementation of Algorithm 6 for the OP. It depicts how a label is iteratively extended to the adjacent successor vertex to maintain feasibility and efficiency. Label $\lambda_{current}$ stores the last vertex visited in the path, and label λ_j memorizes the vertex to which the label is extended, σ_j. At every step of the label extension the following four conditions are tested:

Algorithm 14 : Extend(λ) in the OP

Input: label to be extended $\lambda = [\pi, \nu, \zeta, \mu]$, LB_{best}, $L_{\max}$
Output: labels derived from $\lambda = [\pi, \nu, \zeta, \mu]$
{only non-dominated labels that can later be extended skipping, are kept}

```
1: λ_current ← λ
2: for ( j = ν + 1, n − 1 ) do
3:   if ( ζ(λ_current) + cost(σ_current, σ_j) ≤ L_max ) then
4:     π(λ_j) ← π(λ_currrent) + profit(σ_j)
5:     λ_current ← λ_j
6:     if ( π(λ_j) > LB_best ) then
7:       LB_best ← π(λ_j)
8:       compute upper bound μ(λ_j) {solve FKSP to determine if the label is promising}
9:       if ( μ(λ_j) > LB_best ) then
10:        if ( λ_j not dominated ) then
11:          Λ^j ← Λ^j ∪ {λ_j}
12:        end if
13:      else
14:        return
15:      end if
16:    end if
17:  end if
18: end for
```

i. $\zeta_{\text{path}} \leq L_{\max}$ (line 3)

ii. $LB_{\text{current}} > LB_{\text{best}}$ (line 6)

iii. $\mu(\lambda) > LB_{\text{best}}$ (line 9)

iv. label is not dominated (line 10)

Test (i) enforces the restriction on the maximum tour length. Any vertex whose visit causes the travel cost to exceed the preset bound is simply skipped and the construction of the path continues in the same trajectory. No labels are kept for skipped vertices. If it is possible to reach a vertex, the collected profit is updated, and test (ii) guarantees that the best incumbent always stores the best solution found so far. The upper bound $\mu(\lambda)$ is calculated at each step of the extension of a label by solving a FKSP, and its value is stored in label field μ for later use. Test (iii) allows to determine if a label can be pruned. When $\mu(\lambda) \leq$ *bestKnownProfit*, the search in that direction will not find anything better than the current best solution and it can be abandoned. Note that *bestKnownProfit* $= LB_{\text{best}}$, since we are solving a maximisation problem.

The value of function $\mu(\lambda)$ is used again when the label is retrieved for extension skipping a defined sequence of vertices in the following way: if $\mu(\lambda) >$ *bestKnownProfit*, the label is extended. This test is useful because the value of the best-known solution might have changed since the label was stored.

For each vertex, its queue of labels is ordered by decreasing value of function $\mu(\lambda)$, and in each iteration, the one chosen for extension is the most promising one, i.e., that with the global highest value for this function.

6.4 Performance Improvements

To meet the challenge of controlling the large number of labels created by *Selector* when used to solve the OP, two mechanisms that aim at reducing label proliferation were implemented. Their principles are explained in Chapter 3, so in this section only the specifics of the implementation are discussed.

6.4.1 Computation of an Upper Bound

The knowledge of a well computed upper bound allows to identify non-promising labels that can be pruned. Then, at each step of the label extension, an upper bound on the profit of the complete tour represented by the label needs to be computed. The obtained bound can be compared against the incumbent best-known profit in order to determine if the label created by visiting that vertex is worth storing for further extension. One way to estimate the profit of the path that remains to be searched is by solving a *Fractional Knapsack Problem* (FKSP). Equation 6.1 can be used to estimate the profit of a complete path.

$$\mu(\lambda) = \pi + h(\lambda) \tag{6.1}$$

where

- $\lambda = [\zeta, \sigma_i, \pi, \mu]$ is a label that memorizes σ_i as the last visited vertex.
- π is the sum of the profit values of the already visited vertices.
- $h(\lambda)$ is the upper bound computed on the profit that can be collected visiting only vertices in the subsequence $\varphi = \{\sigma_{i+1}, \ldots, \sigma_{n-1}\}$. This is to say, the profit obtained by visiting only vertices that lie ahead of σ_i.

When solving the FKSP for a given label λ, the capacity of the knapsack is computed as $C = L_{\max} - \zeta$. The objects are the vertices in the subsequence φ. Their profit is self-explanatory and their weight is computed as the sum of the minimum in-going edge weight plus the minimum out-going edge weight. The edges considered are the ones that connect any $\sigma_i \in \varphi$ with its predecessor and successor vertices respectively in the giant tour σ considered in that iteration. The FKSP is solved with a greedy approach. The value of $h(\lambda)$ is the sum of the profit values of the vertices chosen from sequence φ. For a detailed example on the solution of the FKSP, the reader may refer to Section 4.4.

As formerly mentioned, if the upper bound computed on the profit of a given path is worse than the best-known solution, the search in that trajectory must be abandoned. Consequently, if $\mu(\lambda) \leq bestKnownProfit$, the label can be pruned. Otherwise, the label is stored if it is non-dominated.

6.4.2 Restrict the Number of Labels Extended

In order to guarantee a reasonable search time, a limit can be set on the number of labels that are extended when a given instance is solved. In this way, computations are restricted by a known value. However, if only some labels will be extended, we want those to be very good labels, so that, hopefully, we can find a good-quality solution. Hence, we need a mechanism to identify promising labels. This is done in *Selector* with the bounding mechanism explained in the former section. Then, in this implementation of *Selector*, the algorithm extends the most promising label of a limited set. We name the cardinality of the limited set *beam width*. This number was experimentally determined for the instances tested and it is one of the inputs of the algorithm. During the search process, accounting is kept on the number of labels extended, and when this number reaches the beam width, the search stops. However, there is an important consequence in this scheme: it is no longer guaranteed that the solution found is optimal for the given giant tour. This feature makes it an approximate search. All in all, in this implementation, there are three ways for reducing label growth: dominance rules, bounding and limitation on the number of labels extended.

6.5 ALNS: Pseudocode and Parameters

To improve our results, we explored four different construction methods to build the initial giant tour. The aim of a constructive method that is part of a larger solving method is to produce good starting points for the master method. In particular, we want a method that builds an initial giant tour σ^{init} considering both the goal of the problem—profit maximisation—and the total tour length restriction. We believed this process would generate a good σ^{init} for *Selector* to operate on. However, we also included construction methods with different considerations in order to compare. The following explanations mention an identifier for the tours constructed. These identifiers are useful when reading the tables of results, and the same remark applies for the name given to the construction method.

a. Constructive method C1. Each candidate vertex σ_i is evaluated by a greedy function. The one with the best function value is chosen and inserted at the best position (this is, the one that produces the least increase in the tour length). The function used is a ratio between the profit p_i associated with σ_i and its smallest insertion cost Δl_i in σ^{init} as shown in equation (6.2). The vertex with the maximum quotient is the best choice. In other words, vertices with high scores that do not add too much length to the tour are ideal candidates. Afterwards, the produced sequence undergoes a *2-opt* procedure. Tour σ^{init1} is built using this method.

$$f(\sigma_i) = \frac{p_i}{\min \Delta l_i} \tag{6.2}$$

b. Constructive method C2. Same principle as method C1, but randomization is introduced. In each selection of σ_i, the best or second best candidate is chosen according to a generated random number. We seek to enable one second best candidate in one out of ten choices. Tours $\sigma^{\text{initk}} \mid k \in \{2, \ldots, 10\}$ are built using this method.

c. Constructive method C3. Each candidate element σ_i is randomly selected, and the produced sequence undergoes a *2-opt* procedure. Tour σ^{init11} is built using this method.

d. Constructive method C4. The algorithms of nearest neighbour and farthest insertion are used to build σ^{init}. Both sequences obtained undergo a local search procedure *2-opt* to optimize their length and the shorter one is kept. Tour σ^{init12} is built using this method.

Algorithm 15 illustrates the ALNS process implemented to solve the OP with simulated annealing as the outer metaheuristic that guides the search. This algorithm is quite similar to the one explained in Chapter 4. It starts with the construction of the initial giant tour using one of the four construction methods presented, and this tour undergoes a local search procedure *2-opt* to improve its length. The rationale behind this optimisation is that the shorter the tour, the more "room" we have to fit in vertices in the tour extracted by *Selector*. The algorithm then prepares the adaptive layer of the ALNS. Next, it finds a lower bound to start the loop of iterations. The process continues as formerly explained in Chapter 4.

To define an efficient parameter setting, we again used the irace package of López-Ibáñez et al. (2011). A learning set of 50 instances (some derived from TSPLIB and others randomly generated) was designed and *Selector* was run over them. The learning instances used to calibrate the algorithm are, of course, different from the ones used in the benchmark. The resulting set is shown in Table 6.1.

Table 6.1: Values of the ALNS parameters after tuning with irace.

Parameter	Meaning	Value
γ	number of vertices removed in each ALNS iteration (instance size dependent)	$[0.3 \cdot \lvert V\rvert, \epsilon \cdot \lvert V\rvert]$
ς	segment size for updating probabilities in number of ALNS iterations	50
τ	reaction factor that controls the rate of change of the weight adjustment	0.35
δ	avoids determinism in the SRH	7
ρ	avoids determinism in the WRH	3
κ_1	score for finding a new global best solution	40
κ_2	score for finding a new solution that is better than the current one	25
κ_3	score for finding a new non-improving solution that is accepted	10
β	cooling factor used by simulated annealing	0.99999
ϵ	fixes the upper limit of vertices removed at each iteration	0.50

Algorithm 15 : The General Framework of the ALNS with Simulated Annealing

Input: Set V, distance matrix D
Output: T_{best} and $c(T_{\text{best}})$ {Best OP tour and its profit}

$\sigma^{\text{init}} \leftarrow$ Construct Tour(V) ; 2-opt(σ^{init})
compute $l_0(\sigma^{\text{init}})$ {cost of initial giant tour}
initialize, to the same value, probability P_r^t for each removal operator $r \in R$, and likewise probability P_i^t for each insertion operator $i \in I$.
$t \leftarrow l_0$, {set initial temperature, variable used in probability function}
$l_{\text{current}} \leftarrow l_0$; $\sigma^{\text{current}} \leftarrow \sigma^{\text{init}}$
$LB_{\text{best}} \leftarrow$Search Lower Bound($\sigma^{\text{init}}$) ; $c(T_{\text{best}}) \leftarrow c(T_{\text{current}}) \leftarrow$Selector($\sigma^{\text{init}}$)
$i \leftarrow 1$ {iteration counter}
repeat
  select a removal operator $r \in R$ with probability P_r^t {roulette wheel}
  obtain $\sigma^{\text{new}-}$ by applying r to σ^{current}
  select an insertion operator $i \in I$ with probability P_i^t
  obtain σ^{new} by applying i to $\sigma^{\text{new}-}$
  $LB_{\text{current}} \leftarrow$Search Lower Bound($\sigma^{\text{new}}$)
  if ($LB_{\text{current}} > \alpha \cdot LB_{\text{best}}$) **then**
    compute ratio *profit/weight* for each $\sigma_i \in \sigma^{\text{new}} \backslash \{\sigma_0\}$; sort ratios
    $c(T_{\text{new}}) \leftarrow$Selector($\sigma^{\text{new}}$)
  else
    $c(T_{\text{new}}) \leftarrow LB_{\text{current}}$
  end if
  if ($LB_{\text{current}} > LB_{\text{best}}$) **then**
    $LB_{\text{best}} \leftarrow LB_{\text{current}}$
  end if
  {decide acceptance of new solution}
  if ($c(T_{\text{new}}) > c(T_{\text{current}})$) **then**
    $c(T_{\text{current}}) \leftarrow c(T_{\text{new}})$; $\sigma^{\text{current}} \leftarrow \sigma^{\text{new}}$
  else
    $p \leftarrow e^{-\frac{c(T_{\text{current}}) - c(T_{\text{new}})}{t}}$
    generate a random number $n \in [0, 1]$
    {new solution might be accepted even if it is worse}
    if ($n < p$) **then**
      $c(T_{\text{current}}) \leftarrow c(T_{\text{new}})$; $\sigma^{\text{current}} \leftarrow \sigma^{\text{new}}$
    end if
  end if
  if ($c(T_{\text{new}}) > c(T_{\text{best}})$) **then**
    $c(T_{\text{best}}) \leftarrow c(T_{\text{new}})$; $T_{\text{best}} \leftarrow T_{\text{new}}$
  end if
  $t \leftarrow \beta \cdot t$ {cooling rate set to be very slow}
  if (segment size $= \varsigma$) **then**
    update probabilities using the adaptive weight adjustment procedure
  end if
  $i \leftarrow i + 1$
until (defined number of iterations is met)

6.6 Computational Results

6.6.1 Benchmarking Conditions

In order to assess the effectiveness of our method, the results obtained by the metaheuristic are compared against the optimal solution values computed by the branch-and-cut algorithm of Fischetti et al. (1998). Moreover, they are also compared against the solution values obtained by the guided local search (GLS) metaheuristic of Vansteenwegen et al. (2009) and against the state-of-the-art GRASP with path-relinking heuristic of Campos et al. (2014). Vansteenwegen et al. (2009) and Campos et al. (2014) also compare their results against the ones of Fischetti et al. (1998).

However, there exist discrepancies in the results of Fischetti et al. (1998). Vansteenwegen et al. (2009) compared their results against the ones of the original publication of Fischetti et al. (1998), and for some instances, reported higher solution values than the optimal values published by Fischetti et al. (1998). The differences were not very large and they explained them to be the product of rounding-off errors. Later, Campos et al. (2014) experienced the same but finding much larger differences. Due to this situation, they asked for the original code and conducted their own benchmark for the branch-and-cut algorithm, using the same hardware platform and linear programming solver as Fischetti et al. (1998). Campos et al. (2014) publish all the results of their benchmark, but provide no explanation for the differences. For this reason, our tables of results list both values for the branch-and-cut algorithm.

To conduct our experimentation, we used the data set proposed by Vansteenwegen et al. (2009), which itself is a subset of the instances proposed by Fischetti et al. (1998). This test bed of eleven instances is taken from the OP, VRP and TSP literature. Problems whose name starts with the particle *tsi* are OP instances introduced by Tsiligirides (1984), but their travel times were multiplied by 100. The ones identified by the particle *eil* are VRP instances taken from the TSPLIB of Reinelt (1991). The rest are TSP instances also contained in TSPLIB. Instance size ranges from 21 to 400 vertices. The number contained in the instance name indicates its size.

Vansteenwegen et al. (2009) demonstrate that problems with a maximum total travel cost, $L_{\max}$, that allows to select almost all locations or almost no locations are easier to solve than problems where around half of the locations can be selected. The eleven test problems have an $L_{\max}$ value that allows to select half of the points. Fischetti et al. (1998) define the maximum total travel cost as

$$L_{\max} = \lceil \alpha \cdot v(\text{TSP}) \rceil \tag{6.3}$$

where $v(\text{TSP})$ is the length of the shortest Hamiltonian tour, and $\alpha = 0.5$. For all instances taken from TSPLIB, the value $v(\text{TSP})$ is provided within the library. For problems *tsi*21, *tsi*32, and *tsi*33 the values 4598, 8254, and 9755 were used respectively.

The costs $\{c_{ij}\}$ are treated as integer values equal to $\lfloor d_{ij} + .5 \rfloor$, where d_{ij} is the Euclidean distance between points i and j, Reinelt (1991). The profit values are obtained in the following way. For VRP instance *eil*30, the client demands are inter-

preted as profits, while for the rest of the non-OP instances, the vertex profits p_i for $i \in V \backslash \{0\}$ are pseudorandom values in the range $[1, 100]$ and are generated as suggested by Fischetti et al. (1998):

$$p_i = 1 + (7141 \cdot (i + 1) + 73) \text{mod}(100) \tag{6.4}$$

Using the described instances, several independent executions of the heuristic were performed, as is customary in testing the performance of randomized heuristic algorithms. For a given giant tour, an instance is solved ten times with a different seed each time, and each execution lasts 20,000 iterations. The hardware and software platforms are as reported in Chapter 4.

Four construction methods were used to build the initial giant tour σ^{init}. For methods C1, C3 and C4 only one tour is tested, but for method C2 nine different tours are explored. Instances tsi21 and tsi32 were tested with method C1 only because they are very easy to solve problems, and we observed very small variation in the results. We consistently obtained the optimal value in the majority of the executions. For tsi21 the minimum obtained was 200 and for tsi32 it was 150. Furthermore, these two instances were not included in the benchmark of Campos et al. (2014), only in the one of Vansteenwegen et al. (2009). This is another reason why we did not test them with all the construction methods proposed. For instances eil30 and tsi33, we only tested four tours, instead of nine, in construction method C2. The reasons are again easiness of solution and no variation in results.

6.6.2 Discussion of Tables of Results

Table 6.2 documents the comparison of the best values found by the three heuristics (GLS, GRASP and *Selector*) against the optimum value found by the branch-and-cut algorithm. Due to the discrepancies in the results of the branch-and-cut method, we list the results obtained by both research groups, and the gap values reported are against the exact method documented. Letter C indicates that the results are obtained by Campos et al. (2014) or that the gap reported is against these results, while letter F corresponds to Fischetti et al. (1998). Time is given in seconds and the gap in percentage. In the case of *Selector*, the time shown is the one needed to find the best value reported. *Selector* clearly outperforms the GLS method, since the quality of the solutions obtained is always notably higher than the ones of the GLS heuristic, though for some instances execution time is worse. However, when compared to the GRASP method, the quality of the solution obtained is moderately inferior for most instances. It is better only in one instance, surprisingly, the largest one, but the execution time in this case is far worse for *Selector*.

However, it is difficult to perform indirect comparisons of execution times taken from different hardware platforms. Our processor has a SiSoft Sandra Whetstone benchmark score that is roughly 3 times the value of the processor used by the GRASP algorithm. Nonetheless, our average execution time is almost twice the value of the GRASP method. Specifically, in some instances our time is considerably smaller, but

Table 6.2: Comparison of the best values found by the three heuristics against the optimal values obtained by the branch-and-cut procedure.

instance	optimum Fischetti	optimum Campos	time	GLS	gap F	time	GRASP	gap C	time	Selector	gap F	gap C	time
tsi21	205	-	0.7	210	-2.44	0.3	-	-	-	205	0.0	-	0.0
tsi32	160	-	1.3	160	0.0	0.7	-	-	-	160	0.0	-	0.0
eil30	7600	7600	5.2	7600	0.0	0.7	7600	0.0	0.0	7600	0.0	0.0	0.0
tsi33	500	500	1.8	510	-2.0	0.7	500	0.0	0.6	500	0.0	0.0	0.0
eil51	1674	1778	30.7	1707	-2.0	3.2	1778	0.0	3.8	1731	-3.41	2.64	5.6
rd100	3359	3470	27.8	3265	2.8	8.3	3453	0.49	33.7	3442	-2.47	0.81	166.5
kroA100	3212	3181	67.8	3165	1.5	11.6	3181	0.0	29.1	3165	1.46	0.50	0.0
kroA200	6547	6616	805.1	5428	17	29.4	6551	0.98	111.5	6429	1.80	2.83	0.0
pr299	9161	9107	18000	8088	12	64.9	8689	4.59	400.9	8681	5.24	4.68	0.0
lin318	10900	10962	18000	9145	16	101.2	10339	5.68	339.7	10121	7.15	7.67	0.0
rd400	13648	13555	18000	11362	17	215.5	12365	8.78	229.2	12916	5.36	4.71	1780.5
Average			4994.6		5.4	39.7		2.28	127.6		1.38	2.65	177.5

for two instances it is remarkably higher. Specially when one considers that ours is a much faster hardware platform.

Table 6.3 and Table 6.4 document the maximum and minimum values obtained by the different construction methods tested. Values in bold indicate the best value found for the given instance. In general, the difference between the best and worst values is roughly 5%. The solution procedure does benefit from considering both the profit and the length in the construction of σ^{init} since methods C1 or C2 find the best values. However, it is not sufficient. The way the giant tour is improved throughout the search needs to be revised. This means the sub-heuristics of the ALNS need to consider both profit and length and not only length as presently done.

Table 6.5, Table 6.6 and Table 6.7 report the average best results obtained by each $\sigma^{\text{initk}} \mid k \in \{1, \ldots, 12\}$ over the ten runs. The columns of these tables document the following.

- $\overline{iter}$, average number of iterations needed to find the best profit value.
- $\overline{z}$, average best profit value found for the given σ^{init}.
- $\overline{t}$, average time taken to find the best profit value in seconds.
- $\overline{\theta}$, average percentage of deviation from the best-known solution (z_{BKS}), $\theta = 100 * (z_{\text{BKS}} - \overline{z})/z_{\text{BKS}}$.

One observes in these tables that the larger the instance, the larger the average deviation from the optimum. No general statement can be made regarding which

construction method obtains the lowest average deviation. The average number of iterations needed to find the best value is usually below 5,000. One also observes great variability in the average time needed to find the best solution.

Table 6.8, Table 6.9, Table 6.10 and Table 6.11 demonstrate the evolution of the search process, and for this purpose, results have been grouped every 5,000 iterations. In these tables, the first row shows the output for σ^{init1} (C1), the next nine rows (four for the first two instances) for σ^{initk}, $k \in \{2 \dots 10\}$ (C2), the next for σ^{init11} (C3), and the last one for σ^{init12} (C4). The columns document the following (average values are over the 10 runs).

- BW, size of the beam width used.
- α, constant by which the best-known solution is multiplied when compared against the current feasible solution in order to determine the execution of *Selector*.
- OPT, optimal value found by Campos et al. (2014).
- $\overline{SV}$, average profit value.
- $\overline{time}$, average total run time in seconds.
- $\overline{\theta}$, average percentage of deviation from the best-known solution.

These tables show that very frequently the deviation highly improves as more iterations are done. The gap reached varies greatly among the different initial giant tours used to solve the instance.

6.7 Conclusions

We successfully applied the *Selector* operator to the solution of a difficult single-vehicle problem that involves gaining profit upon visitation. Having been able to solve both types of problems considered in this work: covering and profit, we demonstrate that the operator provides a unified methodology for vehicle routing problems with optional visits. The TSP with profits considered proved to be much more difficult to solve than the covering problems studied. Further ideas and concepts need to be explored in order to improve the execution times.

In addition, even though the quality of our results is quite high, they are not better yet than the ones of the state-of-the-art-heuristic. The average deviation for *Selector* in the test bed used is 0.4% larger than the average deviation of the best reference heuristic. Nonetheless, the following considerations must be made. The ALNS method implemented is very simple. More sophisticated destroy-repair sub-heuristics that take into consideration, not only the length of the tour, but also the gained profit can easily enhance it and lead to better results. The other consideration to be made is that Campos et al. (2014) used the instances tested to calibrate their algorithm. This is, the test bed was used to find the best values for key search parameters and also to

compare different search strategies. In other words, the algorithm is prepared to solve those instances. This is in great contrast with our heuristic in which the tuning of the algorithmic parameters is made using learning instances which are different from the ones tested. The test bed included instances whose size ranges from 21 to 400 vertices.

Table 6.3: Maximum and minimum values found by the different construction methods for instances in which $n \leq 50$.

instance	C1	C2				C3	C4
	σ^{init1}	σ^{init2}	σ^{init3}	σ^{init4}	σ^{init5}	σ^{init11}	σ^{init12}
eil30	**7600**	**7600**	**7600**	**7600**	**7600**	**7600**	7275
	7600	7250	7600	7600	7600	7600	7275
tsi33	**500**	**500**	**500**	**500**	**500**	**500**	**500**
	490	490	490	490	490	500	490

Table 6.4: Maximum and minimum values found by the different construction methods for instances in which $n > 50$.

instance	C1	C2									C3	C4
	σ^{init1}	σ^{init2}	σ^{init3}	σ^{init4}	σ^{init5}	σ^{init6}	σ^{init7}	σ^{init8}	σ^{init9}	σ^{init10}	σ^{init11}	σ^{init12}
eil51	1727	**1731**	1696	1720	**1731**	1707	1707	1707	1707	1707	1707	1720
	1698	1680	1665	1662	1665	1656	1651	1652	1660	1683	1652	1680
rd100	3403	3423	3403	3403	3403	3410	3423	**3442**	3427	3403	3427	3423
	3376	3401	3086	3379	3376	3355	3363	3094	3386	3363	3394	3373
kroA100	3122	**3165**	3129	**3165**	3087	3102	3134	3107	**3165**	3125	3129	3131
	3019	3033	2818	3037	2935	2920	3063	3051	2984	3074	3087	2835
kroA200	6374	6251	6396	6134	6329	6344	6382	6254	6153	**6429**	6169	6167
	6235	6004	6165	5805	6085	6210	5985	5988	6059	6152	5860	5973
pr299	8505	8576	8550	8474	**8681**	8575	8594	8508	8607	8600	8433	8513
	8177	8211	7839	7773	8411	7747	7965	8358	8356	8032	8113	8301
lin318	**10121**	9613	9613	9829	9545	9529	9713	9769	9545	9735	9878	9572
	9250	9043	9065	9144	9151	9062	9074	9281	8879	9123	9336	9113
rd400	12106	12408	12460	12176	12297	12604	12311	12240	12041	**12916**	12085	12465
	11581	11672	11774	11718	11592	12050	11753	11668	11708	12565	11439	12152

Table 6.5: Best average values over the ten executions (Part 1/3).

Instance		$\overline{iter}$	$\overline{z}$	$\overline{t}$ (s)	$\overline{\theta}$	Campos et al. *optimum*	Campos et al. z	Campos et al. $t(s)$
eil30	σ^{init1}	0	7600	0.00	0	7600	7600	0
	σ^{init2}	1998	7300	0.05	3.95			
	σ^{init3}	0	7600	0.00	0			
	σ^{init4}	0	7600	0.00	0			
	σ^{init5}	0	7600	0.00	0			
	σ^{init11}	0	7600	0.00	0			
	σ^{init12}	0	7275	0.00	4.28			
tsi33	σ^{init1}	1	497	0.00	0.6	500	500	0.6
	σ^{init2}	2743	494	0.07	1.2			
	σ^{init3}	2748	492	0.08	1.6			
	σ^{init4}	3889	492	0.16	1.6			
	σ^{init5}	2837	494	0.12	1.2			
	σ^{init11}	41	500	0.00	0			
	σ^{init12}	13	495	0.00	1.0			
eil51	σ^{init1}	932	1713	0.61	3.7	1778	1778	3.8
	σ^{init2}	386	1707	0.49	4.0			
	σ^{init3}	6500	1674	4.99	5.8			
	σ^{init4}	2540	1687	2.30	5.1			
	σ^{init5}	3751	1689	3.48	5.0			
	σ^{init6}	6302	1686	4.35	5.2			
	σ^{init7}	7282	1677	4.74	5.7			
	σ^{init8}	5730	1691	5.00	4.9			
	σ^{init9}	2817	1695	3.02	4.7			
	σ^{init10}	3777	1696	3.23	4.6			
	σ^{init11}	4130	1690	3.97	4.9			
	σ^{init12}	1964	1706	1.96	4.0			

Table 6.6: Best average values over the ten executions (Part 2/3).

Instance		$\overline{iter}$	$\overline{z}$	$\overline{t}$ (s)	$\overline{\theta}$	Campos et al. *optimum*	Campos et al. z	Campos et al. $t(s)$
rd100	σ^{init1}	2374	3390	25.44	2.3	3470	3453	33.7
	σ^{init2}	3744	3411	28.57	1.7			
	σ^{init3}	4440	3354	31.27	3.3			
	σ^{init4}	3489	3389	21.12	2.3			
	σ^{init5}	1945	3396	37.49	2.1			
	σ^{init6}	3268	3384	20.36	2.5			
	σ^{init7}	3070	3388	30.22	2.4			
	σ^{init8}	7166	3294	31.31	5.1			
	σ^{init9}	4259	3412	28.93	1.7			
	σ^{init10}	2123	3387	5.47	2.4			
	σ^{init11}	5231	3411	51.21	1.7			
	σ^{init12}	2230	3405	19.84	1.9			
kroA100	σ^{init1}	1382	3058	5.20	3.9	3181	3181	29.1
	σ^{init2}	4271	3082	16.67	3.1			
	σ^{init3}	2726	2988	6.66	6.1			
	σ^{init4}	281	3097	0.00	2.6			
	σ^{init5}	4878	3010	8.05	5.4			
	σ^{init6}	868	3033	3.67	4.7			
	σ^{init7}	1349	3100	12.50	2.5			
	σ^{init8}	3231	3083	12.45	3.1			
	σ^{init9}	2327	3083	7.40	3.1			
	σ^{init10}	2441	3093	12.87	2.8			
	σ^{init11}	2767	3103	5.95	2.5			
	σ^{init12}	6248	3014	16.84	5.2			
kroA200	σ^{init1}	1235	6305	19.53	4.7	6616	6551	111.5
	σ^{init2}	1715	6115	16.98	7.6			
	σ^{init3}	9619	6279	91.05	5.1			
	σ^{init4}	295	5934	3.84	10.3			
	σ^{init5}	2647	6253	32.79	5.5			
	σ^{init6}	1464	6285	11.07	5.0			
	σ^{init7}	1660	6212	29.78	6.1			
	σ^{init8}	811	6159	11.24	6.9			
	σ^{init9}	978	6112	7.89	7.6			
	σ^{init10}	404	6299	2.76	4.8			
	σ^{init11}	2186	6045	22.63	8.6			
	σ^{init12}	921	6045	5.29	8.6			

Table 6.7: Best average values over the ten executions (Part 3/3).

Instance		$\overline{iter}$	$\overline{z}$	$\overline{t}$ (s)	$\overline{\theta}$	Campos et al. *optimum*	Campos et al. z	Campos et al. $t(s)$
pr299	σ^{init1}	7589	8357	180.98	8.2	9107	8689	400.9
	σ^{init2}	3770	8393	96.79	7.8			
	σ^{init3}	1900	8072	68.52	11.4			
	σ^{init4}	6232	8185	155.88	10.1			
	σ^{init5}	836	8484	15.07	6.8			
	σ^{init6}	4625	8330	117.72	8.5			
	σ^{init7}	2394	8308	76.24	8.8			
	σ^{init8}	2539	8449	73.05	7.2			
	σ^{init9}	4965	8495	107.15	6.7			
	σ^{init10}	3879	8279	89.63	9.1			
	σ^{init11}	1023	8243	28.24	9.5			
	σ^{init12}	3046	8409	97.25	7.7			
lin318	σ^{init1}	3208	9473	58.82	13.6	10962	10339	339.7
	σ^{init2}	2279	9316	105.21	15.0			
	σ^{init3}	2168	9312	38.90	15.1			
	σ^{init4}	3576	9445	130.64	13.8			
	σ^{init5}	2153	9330	39.86	14.9			
	σ^{init6}	3476	9305	104.08	15.1			
	σ^{init7}	2702	9384	117.49	14.4			
	σ^{init8}	2283	9524	63.16	13.1			
	σ^{init9}	605	9105	12.24	16.9			
	σ^{init10}	2277	9400	82.24	14.2			
	σ^{init11}	1626	9611	31.51	12.3			
	σ^{init12}	2933	9348	76.41	14.7			
rd400	σ^{init1}	4232	11704	401.05	13.7	13555	12365	229.2
	σ^{init2}	7917	11993	723.58	11.5			
	σ^{init3}	5010	12131	448.76	10.5			
	σ^{init4}	5582	11856	362.40	12.5			
	σ^{init5}	5040	11980	321.10	11.6			
	σ^{init6}	7060	12312	585.27	9.2			
	σ^{init7}	5489	11961	409.20	11.8			
	σ^{init8}	9730	11928	998.68	12.0			
	σ^{init9}	6783	11871	501.93	12.4			
	σ^{init10}	5127	12774	430.72	5.8			
	σ^{init11}	5423	11891	403.66	12.3			
	σ^{init12}	1519	12343	129.44	8.9			

Table 6.8: Comparison of the results obtained by the different construction methods on the test problems of Fischetti (Part 1/4).

iterations					1,000			5,000			10,000			15,000			20,000		
instance	BW	α	OPT	σ^{init}	$\overline{SV}$	$\overline{time}$	$\overline{\theta}$	$\overline{SV}$	$\overline{time}$	$\overline{\theta}$	$\overline{SV}$	$\overline{time}$	$\overline{\theta}$	$\overline{SV}$	$\overline{time}$	$\overline{\theta}$	$\overline{SV}$	$\overline{time}$	$\overline{\theta}$
eil30	10	1.00	7600	σ^{init1}	7600	0.03	0.00	7600	0.16	0.00	7600	0.32	0.00	7600	0.46	0.00	7600	0.60	0.00
				σ^{init2}	7263	0.03	4.43	7265	0.14	4.41	7265	0.30	4.41	7265	0.47	4.41	7300	0.64	3.95
				σ^{init3}	7600	0.04	0.00	7600	0.18	0.00	7600	0.35	0.00	7600	0.51	0.00	7600	0.66	0.00
				σ^{init4}	7600	0.03	0.00	7600	0.14	0.00	7600	0.26	0.00	7600	0.39	0.00	7600	0.53	0.00
				σ^{init5}	7600	0.03	0.00	7600	0.17	0.00	7600	0.33	0.00	7600	0.48	0.00	7600	0.64	0.00
				σ^{init11}	7600	0.03	0.00	7600	0.16	0.00	7600	0.32	0.00	7600	0.48	0.00	7600	0.64	0.00
				σ^{init12}	7275	0.04	4.28	7275	0.19	4.28	7275	0.33	4.28	7275	0.47	4.28	7275	0.62	4.28
tsi33	10	1.00	500	σ^{init1}	497	0.04	0.60	497	0.18	0.60	497	0.38	0.60	497	0.58	0.60	497	0.78	0.60
				σ^{init2}	490	0.04	2.00	492	0.22	1.60	493	0.46	1.40	494	0.69	1.20	494	0.91	1.20
				σ^{init3}	486	0.05	2.80	489	0.24	2.20	492	0.46	1.60	492	0.66	1.60	492	0.87	1.60
				σ^{init4}	485	0.05	3.00	489	0.25	2.20	491	0.48	1.80	491	0.70	1.80	492	0.91	1.60
				σ^{init5}	493	0.05	1.40	493	0.23	1.40	493	0.44	1.40	493	0.65	1.40	493	0.86	1.40
				σ^{init11}	500	0.04	0.00	500	0.20	0.00	500	0.41	0.00	500	0.61	0.00	500	0.81	0.00
				σ^{init12}	495	0.05	1.00	495	0.22	1.00	495	0.44	1.00	495	0.69	1.00	495	0.92	1.00
eil51	100	0.90	1778	σ^{init1}	1712	3.40	3.71	1713	7.97	3.66	1713	10.82	3.66	1713	13.64	3.66	1713	15.88	3.66
				σ^{init2}	1705	4.75	4.11	1707	10.06	3.99	1707	14.59	3.99	1707	17.31	3.99	1707	20.38	3.99
				σ^{init3}	1643	2.89	7.59	1667	6.95	6.24	1670	9.60	6.07	1670	11.79	6.07	1674	13.73	5.85
				σ^{init4}	1667	3.52	6.24	1685	8.00	5.23	1687	13.28	5.12	1687	17.62	5.12	1687	19.80	5.12
				σ^{init5}	1679	3.07	5.57	1687	7.61	5.12	1689	10.73	5.01	1689	13.30	5.01	1689	16.36	5.01
				σ^{init6}	1637	2.56	7.93	1670	7.13	6.07	1675	10.82	5.79	1686	13.61	5.17	1686	14.99	5.17
				σ^{init7}	1644	2.42	7.54	1661	6.28	6.58	1667	8.41	6.24	1672	10.53	5.96	1677	12.90	5.68
				σ^{init8}	1650	2.86	7.20	1683	6.53	5.34	1688	10.41	5.06	1691	12.72	4.89	1691	15.41	4.89
				σ^{init9}	1661	4.95	6.58	1692	8.54	4.84	1693	12.78	4.78	1695	15.76	4.67	1695	18.04	4.67
				σ^{init10}	1687	3.99	5.12	1692	8.21	4.84	1692	11.81	4.84	1696	14.44	4.61	1696	16.78	4.61
				σ^{init11}	1670	3.57	6.07	1687	7.44	5.12	1688	10.36	5.06	1690	14.04	4.95	1690	16.29	4.95
				σ^{init12}	1701	4.29	4.33	1704	9.86	4.16	1706	15.59	4.05	1706	19.51	4.05	1706	22.54	4.05

Table 6.9: Comparison of the results obtained by the different construction methods on the test problems of Fischetti (Part 2/4).

iterations					1,000			5,000			10,000			15,000			20,000		
instance	BW	α	OPT	σ^{init}	$\overline{SV}$	$\overline{time}$	$\overline{\theta}$	$\overline{SV}$	$\overline{time}$	$\overline{\theta}$	$\overline{SV}$	$\overline{time}$	$\overline{\theta}$	$\overline{SV}$	$\overline{time}$	$\overline{\theta}$	$\overline{SV}$	$\overline{time}$	$\overline{\theta}$
rd100	50	0.99	3470	σ^{init1}	3380	35.47	2.59	3386	67.12	2.42	3388	72.91	2.36	3390	78.63	2.31	3390	82.48	2.31
				σ^{init2}	3372	19.47	2.82	3409	65.05	1.76	3411	97.56	1.70	3411	121.02	1.70	3411	172.73	1.70
				σ^{init3}	3314	30.63	4.50	3344	50.14	3.63	3344	63.47	3.63	3354	77.18	3.34	3354	97.21	3.34
				σ^{init4}	3381	25.55	2.56	3387	65.29	2.39	3387	71.11	2.39	3388	77.05	2.36	3389	80.99	2.33
				σ^{init5}	3388	48.92	2.36	3394	107.73	2.19	3396	135.30	2.13	3396	142.05	2.13	3396	147.06	2.13
				σ^{init6}	3370	45.14	2.88	3375	86.64	2.74	3377	92.34	2.68	3384	95.50	2.48	3384	99.72	2.48
				σ^{init7}	3375	34.84	2.74	3385	75.57	2.45	3385	83.64	2.45	3388	91.99	2.36	3388	104.47	2.36
				σ^{init8}	3165	13.65	8.79	3238	30.32	6.69	3278	41.19	5.53	3283	54.87	5.39	3294	65.30	5.07
				σ^{init9}	3395	26.04	2.16	3404	49.51	1.90	3410	67.38	1.73	3412	87.30	1.67	3412	108.10	1.67
				σ^{init10}	3377	27.73	2.68	3381	58.96	2.56	3381	63.96	2.56	3387	74.90	2.39	3387	79.42	2.39
				σ^{init11}	3390	25.93	2.31	3404	65.82	1.90	3407	89.08	1.82	3409	112.44	1.76	3411	132.63	1.70
				σ^{init12}	3399	24.56	2.05	3403	56.23	1.93	3403	75.56	1.93	3405	99.66	1.87	3405	120.22	1.87
kroA100	20	0.99	3181	σ^{init1}	3033	12.20	4.65	3053	23.14	4.02	3058	28.67	3.87	3058	32.99	3.87	3058	38.19	3.87
				σ^{init2}	3071	29.29	3.46	3075	38.24	3.33	3080	42.09	3.18	3081	47.74	3.14	3082	53.60	3.11
				σ^{init3}	2946	18.86	7.39	2988	34.69	6.07	2988	41.75	6.07	2988	52.71	6.07	2988	56.76	6.07
				σ^{init4}	3097	23.23	2.64	3097	35.44	2.64	3097	40.26	2.64	3097	44.12	2.64	3097	47.30	2.64
				σ^{init5}	2966	12.24	6.76	2983	18.37	6.22	2983	22.81	6.22	3010	31.99	5.38	3010	37.79	5.38
				σ^{init6}	3016	24.35	5.19	3024	29.36	4.94	3033	37.66	4.65	3033	42.12	4.65	3033	47.50	4.65
				σ^{init7}	3074	18.56	3.36	3093	34.42	2.77	3100	39.16	2.55	3100	44.55	2.55	3100	48.41	2.55
				σ^{init8}	3067	24.17	3.58	3071	33.02	3.46	3071	37.21	3.46	3077	41.55	3.27	3083	46.76	3.08
				σ^{init9}	3072	14.51	3.43	3078	25.52	3.24	3083	29.96	3.08	3083	33.59	3.08	3083	37.78	3.08
				σ^{init10}	3083	26.07	3.08	3089	42.22	2.89	3092	47.46	2.80	3093	51.54	2.77	3093	56.36	2.77
				σ^{init11}	3095	17.14	2.70	3101	27.89	2.51	3101	31.85	2.51	3101	35.29	2.51	3103	40.07	2.45
				σ^{init12}	2828	11.45	11.10	2949	26.89	7.29	2979	32.08	6.35	3013	37.71	5.28	3014	42.64	5.25

Table 6.10: Comparison of the results obtained by the different construction methods on the test problems of Fischetti (Part 3/4).

iterations					1,000			5,000			10,000			15,000			20,000		
instance	BW	α	OPT	σ^{init}	$\overline{SV}$	$\overline{time}$	$\overline{\theta}$	$\overline{SV}$	$\overline{time}$	$\overline{\theta}$	$\overline{SV}$	$\overline{time}$	$\overline{\theta}$	$\overline{SV}$	$\overline{time}$	$\overline{\theta}$	$\overline{SV}$	$\overline{time}$	$\overline{\theta}$
kroA200	8	1.00	6616	σ^{init1}	6266	60.38	5.29	6288	102.22	4.96	6305	137.39	4.70	6305	157.67	4.70	6305	182.02	4.70
				σ^{init2}	6078	25.52	8.13	6098	65.69	7.83	6115	100.10	7.57	6115	133.97	7.57	6115	158.23	7.57
				σ^{init3}	5845	38.05	11.65	6055	87.00	8.48	6126	123.99	7.41	6146	162.41	7.10	6279	211.54	5.09
				σ^{init4}	5929	38.34	10.38	5934	74.71	10.31	5934	110.74	10.31	5934	140.90	10.31	5934	172.85	10.31
				σ^{init5}	6200	51.48	6.29	6238	106.92	5.71	6246	139.48	5.59	6253	159.33	5.49	6253	176.92	5.49
				σ^{init6}	6273	49.25	5.18	6282	86.19	5.05	6285	116.13	5.00	6285	141.72	5.00	6285	172.21	5.00
				σ^{init7}	6133	53.78	7.30	6177	87.95	6.64	6212	109.51	6.11	6212	138.46	6.11	6212	163.34	6.11
				σ^{init8}	6140	26.47	7.19	6159	65.53	6.91	6159	103.49	6.91	6159	140.22	6.91	6159	172.47	6.91
				σ^{init9}	6099	27.91	7.81	6112	60.22	7.62	6112	100.12	7.62	6112	127.18	7.62	6112	152.72	7.62
				σ^{init10}	6296	28.11	4.84	6299	59.94	4.79	6299	85.76	4.79	6299	118.08	4.79	6299	146.86	4.79
				σ^{init11}	5989	29.21	9.48	6028	67.79	8.89	6028	114.28	8.89	6028	148.78	8.89	6045	180.65	8.63
				σ^{init12}	6038	27.63	8.74	6038	61.83	8.74	6045	92.26	8.63	6045	136.69	8.63	6045	167.21	8.63
pr299	2	1.00	9107	σ^{init1}	7819	59.85	14.14	8012	180.76	12.02	8270	346.75	9.19	8357	477.11	8.24	8357	609.89	8.24
				σ^{init2}	8285	83.73	9.03	8379	231.44	7.99	8379	373.50	7.99	8393	539.07	7.84	8393	685.77	7.84
				σ^{init3}	7955	92.86	12.65	8022	265.18	11.91	8072	423.80	11.36	8072	565.91	11.36	8072	714.70	11.36
				σ^{init4}	7825	69.13	14.08	7973	188.34	12.45	8137	345.85	10.65	8185	477.60	10.12	8185	627.84	10.12
				σ^{init5}	8476	79.63	6.93	8484	188.38	6.84	8484	308.05	6.84	8484	417.60	6.84	8484	543.20	6.84
				σ^{init6}	7497	33.88	17.68	8259	150.93	9.31	8300	271.63	8.86	8315	389.44	8.70	8330	530.69	8.53
				σ^{init7}	8165	78.08	10.34	8253	194.14	9.38	8301	313.03	8.85	8308	417.02	8.77	8308	528.64	8.77
				σ^{init8}	8385	93.68	7.93	8446	211.01	7.26	8449	337.59	7.23	8449	440.59	7.23	8449	523.54	7.23
				σ^{init9}	8413	59.61	7.62	8485	177.18	6.83	8492	296.18	6.75	8492	410.45	6.75	8495	503.70	6.72
				σ^{init10}	8087	78.36	11.20	8145	182.25	10.56	8238	275.50	9.54	8279	439.38	9.09	8279	608.67	9.09
				σ^{init11}	8217	77.45	9.77	8230	189.21	9.63	8243	276.67	9.49	8243	410.84	9.49	8243	546.21	9.49
				σ^{init12}	8344	80.42	8.38	8394	202.71	7.83	8394	312.17	7.83	8409	422.66	7.66	8409	495.57	7.66

Table 6.11: Comparison of the results obtained by the different construction methods on the test problems of Fischetti (Part 4/4).

iterations					1,000			5,000			10,000			15,000			20,000		
instance	BW	α	OPT	σ^{init}	$\overline{SV}$	$\overline{time}$	$\overline{\theta}$	$\overline{SV}$	$\overline{time}$	$\overline{\theta}$	$\overline{SV}$	$\overline{time}$	$\overline{\theta}$	$\overline{SV}$	$\overline{time}$	$\overline{\theta}$	$\overline{SV}$	$\overline{time}$	$\overline{\theta}$
lin318	2	1.00	10962	σ^{init1}	9445	97.29	13.84	9461	283.28	13.69	9461	463.65	13.69	9461	590.83	13.69	9473	732.80	13.58
				σ^{init2}	9208	110.10	16.00	9313	274.07	15.04	9313	418.10	15.04	9316	558.56	15.02	9316	734.62	15.02
				σ^{init3}	9169	67.03	16.36	9312	227.36	15.05	9312	375.57	15.05	9312	531.40	15.05	9312	662.28	15.05
				σ^{init4}	9240	116.69	15.71	9437	327.44	13.91	9438	527.79	13.90	9445	687.76	13.84	9445	808.03	13.84
				σ^{init5}	9236	72.77	15.75	9330	265.75	14.89	9330	460.64	14.89	9330	668.40	14.89	9330	837.44	14.89
				σ^{init6}	9141	75.08	16.61	9280	235.79	15.34	9305	363.11	15.12	9305	477.21	15.12	9305	598.74	15.12
				σ^{init7}	9140	104.20	16.62	9363	273.37	14.59	9363	437.98	14.59	9384	650.99	14.40	9384	793.94	14.40
				σ^{init8}	9411	77.82	14.15	9514	249.67	13.21	9514	430.51	13.21	9524	609.17	13.12	9524	777.47	13.12
				σ^{init9}	9065	66.65	17.31	9105	203.17	16.94	9105	340.92	16.94	9105	495.82	16.94	9105	627.57	16.94
				σ^{init10}	9200	92.67	16.07	9390	250.18	14.34	9400	398.87	14.25	9400	551.86	14.25	9400	681.40	14.25
				σ^{init11}	9533	80.67	13.04	9595	216.30	12.47	9611	368.21	12.32	9611	509.45	12.32	9611	639.14	12.32
				σ^{init12}	9299	76.28	15.17	9317	208.15	15.01	9324	343.26	14.94	9348	469.40	14.72	9348	614.97	14.72
rd400	1	1.00	13555	σ^{init1}	11620	157.08	14.28	11688	528.08	13.77	11693	953.86	13.74	11704	1383.94	13.66	11704	1829.37	13.66
				σ^{init2}	11787	164.03	13.04	11907	652.45	12.16	11928	1239.12	12.00	11950	1816.12	11.84	11993	2293.29	11.52
				σ^{init3}	11944	197.48	11.88	12054	640.69	11.07	12089	1101.90	10.82	12089	1482.90	10.82	12131	1868.02	10.51
				σ^{init4}	11830	169.06	12.73	11832	533.54	12.71	11848	984.05	12.59	11851	1472.14	12.57	11856	2011.19	12.53
				σ^{init5}	11884	168.87	12.33	11933	563.27	11.97	11979	977.70	11.63	11979	1294.78	11.63	11980	1608.13	11.62
				σ^{init6}	12129	180.69	10.52	12234	665.43	9.75	12255	1274.58	9.59	12311	1845.82	9.18	12312	2399.21	9.17
				σ^{init7}	11862	173.36	12.49	11893	564.16	12.26	11950	1109.30	11.84	11953	1648.99	11.82	11961	2100.94	11.76
				σ^{init8}	11557	157.57	14.74	11650	629.56	14.05	11770	1217.94	13.17	11915	1666.36	12.10	11928	2179.07	12.00
				σ^{init9}	11749	132.41	13.32	11793	504.27	13.00	11859	982.91	12.51	11867	1369.08	12.45	11871	1735.58	12.42
				σ^{init10}	12720	151.37	6.16	12736	516.88	6.04	12736	967.39	6.04	12753	1454.64	5.92	12774	1958.90	5.76
				σ^{init11}	11727	156.24	13.49	11819	512.09	12.81	11862	969.27	12.49	11891	1469.45	12.28	11891	1949.60	12.28
				σ^{init12}	12334	167.45	9.01	12334	512.36	9.01	12334	849.91	9.01	12343	1185.91	8.94	12343	1563.64	8.94

Chapter 7

Conclusions and Perspectives

Contents

7.1 Conclusions

The main focus of this thesis is the study of a heuristic methodology general enough to be applied to a class of routing problems in which one does not know, a priori, which customers will be on the tour. The key element of this study is an operator which formulates the problem of selecting the best customers to visit for the given objective as a *Resource Constrained Elementary Shortest Path Problem* on an auxiliary directed acyclic graph where the side restrictions of the problem considered act as the constraining resource. This auxiliary graph represents the topological order of the n customers treated. The operator solves the problem with a dynamic programming approach, and the algorithm is an adaptation of the one developed by Desrochers (1988) in the context of the RCSPP. It is a label-correcting reaching algorithm Feillet et al. (2004). The solution methodology proposed emulates a route first–cluster second constructive heuristic, so the operator works embedded into a generic metaheuristic in charge of providing permutations of customers (routing) from which the operator retrieves a VRPOV solution (clustering) which fulfills the resource constraints, while optimising the given objective. The operator can be understood as a splitting operator that separates the customers into visited and non-visited sequences.

In order to enhance the performance of our operator, several strategies were explored such as computing a lower/upper bound that enables to identify promising labels and prune those that are not. This mechanism greatly contributed to control the proliferation of labels and it proved to be crucial in the solution of the *Orienteering Problem* (OP). Bidirectional search was also explored when solving the *Covering Tour Problem* (CTP), although we were not able to solve a considerably larger set of instances with this version than with the monodirectional one. Other resource explored was limiting the number of labels extended in a given search. Even though this transformed the search into a non-optimal one, this technique allowed to attain tractability when solving the OP. We also looked into reducing the size of the given instance by defining criteria to select sets of elite vertices that still allowed to obtain good-quality solutions. This strategy proved successful and allowed to solve a 200-vertex instance

that exhibited quite high execution times. Providing better starting points for the algorithm was also analysed, and better suited construction methods for the problem at hand allowed to attain better results.

The metaheuristic chosen does not use small conventional moves to perform local search. Instead, it explores very large moves that can rearrange a high percentage of the solution in the hope of avoiding difficulties in moving from one promising area of the search space to another. In reality, such a large neighborhood is merely sampled, but it is still possible to obtain good performance as shown by the results obtained.

When designing the heuristic, the characteristics sought by users and explained in Chapter 1 such as simplicity, precision, easiness in adapting to a variety of problems, etc. were taken into account, though we might not have been successful in integrating them wholly in all our implementations.

The heuristic was applied to diverse VRPOV: the single-vehicle CTP, the multi-vehicle CTP (m-CTP), and the OP. The customization required by the operator in order to adapt it to the solution of a particular problem was fairly simple. Very few and specific modules of the code needed tailoring. The CTP and m-CTP are much easier to solve problems compared to the OP which required more mechanisms to control the proliferation of labels. When solving the CTP, the quality of the results obtained for small and medium-sized instances, compared against the output of an exact method, is very good. For most of the tested instances, the solution obtained is within 1% of optimality. There are neither standard libraries of instances nor reference benchmarks for the CTP, so for larger instances, the results of the monodirectional and bidirectional searches were compared. Therefore, the gap to optimality is unknown. Computational results were reported for a set of instances whose size ranges from 100 to 575 vertices and the tour may contain from 25 up to 268 vertices.

Regarding the m-CTP, the implementation of *Selector* introduces a novel way to use the *Split* operator which allowed to solve the problem without resorting to the use of labels to define and evaluate routes. The considerations made by the problem eased the task of adapting this operator. The results obtained are of similar quality to the ones of the state-of-the-art exact and heuristic algorithms. However, the execution time for the case when the route length constraint is relaxed needs to be improved. Instances with up to 200 vertices where the tour may contain up to 100 of them were solved.

The quality of the results obtained for the OP is very close to the one of the state-of-the-art heuristic. However, in this heuristic the test bed is used to calibrate the algorithm. This is, the tested instances were used to find the best values for key search parameters and also to compare different search strategies. In other words, the algorithm is prepared to solve those instances. This is in great contrast with our heuristic in which the tuning of the algorithmic parameters is made using learning instances which are different from the ones tested. Notwithstanding, further ideas and concepts need to be explored in order to improve the execution times. Instances as large as 400 vertices were solved.

All in all, the contributions of this thesis work are (1) the development of the *m-Selector* operator and its performance improvement features; (2) the unified heuris-

tic methodology that incorporates this operator to solve VRPOV; and (3) the novel ways proposed to solve the different VRPOV in which the developed methodology was applied.

7.2 Perspectives

Heuristic search for solving VRP is a widely spread topic in the operational research community. Some studies have reached quite a high level, yet researchers will continue to be challenged to develop even better, more general and robust heuristics as more complex vehicle routing problems keep coming up in real life together with the need to solve larger instances. In my opinion, it is very likely that the trend will continue to be towards hybrid heuristics that take advantage of the benefits of diverse solution methods. Also, parallel implementations will continue to be an important topic of research.

In our specific case, further research can be done by applying the heuristic to the solution of other VRPOV. In fact, we have already started this activity. We have adapted the *Selector* operator to solve the *Cumulative Covering Tour Problem* where the objective is not total route cost minimisation, but the minimisation of the sum of arrival times at delivery points. We have concluded the one-vehicle version, and have compared the results yielded by our implementation to the ones of the GRASP-based heuristic of Flores-Garza et al. (2015). *Selector* has clearly outperformed these results. We will continue to implement the multi-vehicle version. Another problem that is clearly of interest because of its wide applicability is the *Team Orienteering Problem*.

Another research path is the metaheuristic used. The ALNS is very modular so it is not difficult to further explore its different components. The implementation done in this thesis work is quite simple, and it can be enriched by developing more sophisticated sub-heuristics that consider other measures rather than only the length of the giant tour constructed. For example, consider profit as well as length in the routing problems with profits. In addition, the method used to accept a non-improving solution can be further explored. The scheme used, simulated annealing, is simple to implement and integrates well with the ALNS, but Table 2.6 refers some other possibilities. Other element that could be analysed is to add the noise term, suggested by Ropke and Pisinger (2006) to favour diversification, in a different manner to the one used. A more radical move is to replace the metaheuristic for other one, such as a population-based method like genetic algorithms or ant colony optimisation.

However, for all its strengths, the greatest shortcoming of *Selector* is that a dynamic programming formulation of a problem always yields an algorithm whose efficiency is determined by the number of states, labels in our case, created. Thus, a future research path for *Selector* is finding additional ways to reduce the number of labels created. One possibility is the idea introduced by Toth and Vigo (2003) in their granular tabu search algorithm: graph sparsification. This is to say, approximate a given graph by a graph with fewer edges or vertices. The main idea behind this pre-processing stems from the observation that the longer edges of a graph have only a small possibility of belonging

to an optimal solution. Hence, by eliminating all edges whose length exceeds a value, which Toth and Vigo named *granularity threshold*, several non-promising solutions will never be considered by the search process. In order to set the granularity threshold, Toth and Vigo proposed using $\nu = \beta\bar{c}$, where β is referred as a *sparsification parameter* and $\bar{c}$ is the average edge length of a solution obtained by a fast heuristic. These authors explain that if $\beta \in [1.0, 2.0]$, then the percentage of remaining edges in the graph tends to be in the 10% to 20% range. If graph sparsification is applied to the auxiliary graph H over which *Selector* operates, less labels will be created and a more efficient algorithm is possible. However, adequate values of β need to be investigated together with an appropriate granularity threshold. In addition, considerations such as maintaining a set of important edges such as those incident to the depot or belonging to high-quality solutions need to be examined.

Solving the RCSPP with a dynamic programming-based algorithm is closely related to solving multi-objective shortest path problems. The aim in these problems is also to generate non-dominated paths (i.e. Pareto optimal paths). Then, another possible research path could be to develop a multi-objective *Selector* operator. Many of the VRPOV are multi-objective in nature and could be treated as explicit bi-objective problems, so an operator with this approach is not out of place.

Appendix A

Un opératur de programmation dynamique pour les méta-heuristiques pour résoudre les problèmes de tournées de véhicules avec des visites optionnelles

Contents

A.1 Introduction

Cette thèse vise à contribuer à l'étude des problèmes de tournées des véhicules avec des visites optionnelles (PTVVO) en proposant une approche heuristique unifiée pour

résoudre ce genre de problèmes. La méthodologie développée a été appliquée pour résoudre le Covering Tour Problem (CTP) Gendreau et al. (1997), le Multi-Vehicle Covering Tour Problem (m-CTP) (Hachicha et al., 2000), et le Orienteering Problem(OP) (Tsiligirides, 1984).

Nous avons développé un méta-heuristique hybride basé sur l'approche séminale de Beasley (1983) connu comme *route first–cluster second*. Sa principale caractéristique est un opérateur basé sur la programmation dynamique, nommé *Selector*, visant à extraire des solutions de PTVVO. La phase de routage de l'approche de solution unifiée proposée est géré par un méta-heuristique générique qui produit des tours géantes (permutation de tous les n clients pour visiter) de haute qualité en utilisant des heuristiques subordonnés. Le méta-heuristique appliquée pour ce travail est la recherche adaptatif à grand voisinage introduit par Ropke and Pisinger (2006). La phase de cluster est résolu par l'opérateur *Selector* incorporé dans le méta-heuristique. L'opérateur accomplit les tâches suivantes: (i) sélectionne de façon optimale les sommets à visiter dans le tour géant formée par les sous-heuristiques (groupes les sommets dans sous-séquences visités et sous-séquences non visités); (ii) évalue les coûts de l'itinéraire et (iii) des problèmes multi-véhicules, co-travaux avec le opérateur *Split* de façon optimale le segment un ensemble de sommets sélectionnés dans les itinéraires faisables de véhicules possibles. La tâche globale du méta-heuristique est de construire de tours géantes de bons qualité à partir de laquelle l'opérateur *Selector* récupère des solutions PTVVO efficaces.

L'opérateur *Selector* est un nouvel algorithme introduit dans ce travail de thèse. Les travaux préliminaires sur elle a été présentée dans le XVII Conferencia Latino Americana en Investigación de Operaciones (CLAIO 2014), Monterrey, Mexique, Octobre 6-10 2014. D'autres travaux a ensuite été présenté dans le 9^{e} Learning and Intelligent OptimizatioN Conference (LION9), à Lille, en France, janvier 12-15, 2015, et un document a été publié dans les actes du colloque, (Vargas et al., 2015a). *Selector* a d'abord été appliquée à la solution du CTP, et plus tard, la version mis en œuvre pour résoudre le OP a été présenté dans la 4^{e} réunion du EURO Working Group on Vehicle Routing and Logistics Optimization (VeRoLog 2015) à Vienne, en Autriche, juin 8-10, 2015. La version de *Selector* visant à résoudre plusieurs véhicules PTVVO a été appliqué à la solution du m-CTP, et les résultats ont été présentés à la 17ème Conférence de la Société Française de Recherche Opérationnelle et d'Aide à la Décision (ROADEF 2016) à Compiègne, en France, février 10-12, 2016.

Vidal et al. (2015) a récemment publié un recherche à grand voisinage pour problèmes de tournée de plusieurs véhicules avec des profits basés sur un algorithme qui suit les mêmes principes que *Selector*. Ils produisent d'une manière heuristique une solution conventionnelle de tournée de véhicule, qu'ils nomment une représentation exhaustive de la solution car elle rend visite à tous les clients, puis à plusieurs reprises appliquent leur sélection algorithme sur chaque nouvel itinéraire afin de récupérer les séquences optimales de visites aux clients. Ils testent leur structure de voisinage dans les trois cadres heuristiques: un multi-démarrage recherche locale, un multi-démarrage itéré recherche locale basée sur la méthode de Prins (2009), et une recherche génétique hybride (UHGS) découle directement du cadre général de la Vidal et al. (2014). *Selector*

est envisagé comme un opérateur unifié pour résoudre PTVVO, alors que leur select algorithme est conçu uniquement dans le cadre de problèmes de tournée de plusieurs véhicules avec les bénéfices.

Le résumé est organisé comme suit. Le point de départ de l'explication est une brève présentation, dans la section A.2, des fondements de la méthode. Ensuite, une explication de l'algorithme de *Selector* est fourni dans la section A.3. Sections A.4 à A.6 présentent des modifications effectuées à l'algorithme de base de *Selector* pour remplir des objectifs spécifiques. La mise en œuvre faite de le méta-heuristique sélectionnée est ensuite détaillé dans la section A.7. Les observations finales sont présentées dans la section A.8.

A.2 Le Bases du *Selector*

Quand la résolution d'un PTVVO, l'opérateur *Selector* divise de manière optimale une tour géant en séquences de sommets visités et non visités d'une manière semblable comme le *Split* opérateur segmente de façon optimale une tour géant dans les itinéraires faisables de véhicules lorsqu'il est appliqué pour résoudre le problème de tournée de véhicule capacitation tel que proposé par Prins (2004). La division de la tour géant dans des itinéraires de véhicule nécessite de résoudre un problème de plus court chemin. Cependant, dans le cas d'un PTVVO, les restrictions latérales ont considéré des actes comme ressources de contrainte et le problème à résoudre alors devient un problème de plus court chemin élémentaire avec ressources limitées (PPCCERL).

L'opérateur *Selector* partage des similitudes avec le opérateur *Split*. Cependant, le fait de ne pas savoir a priori que les sommets constituent la tournée laisse un problème un plus difficile à résoudre. Néanmoins, le PPCCERL pour un PTVVO peut être résolu assez rapidement dans la pratique en adaptant l'algorithme conçu par Desrochers (1988) pour le problème de plus court chemin avec ressources limitées (PPCCRL). L'exigence élémentaire est garantie par le fait que l'ordre donné par la tour géant est respectée.

Problème de Plus Court Chemin Élémentaire avec Ressources Limitées (PPCCERL). Le PPCCERL est le problème de trouver un plus court chemin élémentaire d'un sommet de source v_s à un sommet de cible v_t dans un réseau tels que l'utilisation de ressource globale ne dépasse pas quelques limites données. Par conséquent, le disque des ressources employées par chaque chemin devrait être gardé. Des ressources sont consommées quand la visite des sommets ou la traversée des arches. Un tel problème est NP-difficile (Dror (1994)). L'approche standard pour résoudre un PPCCERL à l'optimalité est programmation dynamique, et d'avoir la complexité pseudo-polynomiale.

Cette approche compte sur le travail séminal de Desrochers (1988) qui a proposé un algorithme pour résoudre une version décontractée de ce problème, le PPCCRL, où les besoins de chemin ne pas être élémentaire. La procédure de Desrochers est une extension multi-label de l'algorithme de Bellman-Ford prenant en compte des

contraintes de ressource. La nécessité de garder le disque des ressources utilisées oblige d'assigner plusieurs labels à chaque sommet.

La procédure de Desrochers est un algorithme de atteignant de correction de label. Son principe de base est d'associer un label à chaque chemin partiel qui va du sommet d'origine v_s à un sommet v_i. Le label représente le coût du chemin et de sa consommation des ressources. On élimine des labels inutiles pendant que la recherche progresse. Des sommets sont itérativement traités jusqu'à ce qu'aucun nouveau label ne soit créé. Quand un sommet est traité, tous ses nouveaux labels sont prolongés vers chaque sommet de successeur possible. Dans toute la recherche, puis, chaque sommet reçoit plusieurs labels. Quand un label est prolongé du sommet v_i au sommet v_{i+1} pour produire d'un autre label faisable, le coûtent et la consommation de ressource du nouveau label doit être calculée selon une formule de répétition.

Le soi-disant label, qui représente un chemin faisable, peut être compris comme un vecteur $V = [\zeta | r_1, r_2, \ldots, ler_m]$ qui mémorise le coût du chemin ζ et les consommations de ressource r_i le long du chemin correspondant. Ces consommations permettent de savoir si un chemin partiel peut encore être prolongé. L'efficacité de l'algorithme basé sur la programmation dynamique décrit dans l'ancien paragraphe compte fortement sur la possibilité d'élagage les labels qui ne peuvent pas mener à une solution optimale. À cet effet, des essais appropriés de dominance sont toujours réalisés quand les labels sont prolongés, de sorte que seulement des labels dominés par non soient stockés. Pour trouver le chemin optimal seulement les chemins dominés par non doivent être considérés comme. Pour une enquête sur des modèles et des algorithmes pour le PPCCRL et le PPCCERL, le lecteur intéressé peut consulter Irnich and Desaulniers (2004). Feillet et al. (2004) a présenté un algorithme exact pour résoudre le PPCCERL appliqué à Problèmes de Tournées de Véhicules.

A.3 L'Opérateur *Selector*

L'explication procède maintenant au mécanisme de groupement proposé dans cette thèse : l'opérateur *Selector*. L'opérateur est un algorithme de atteignant de correction de label et un algorithme basé sur la programmation dynamique qui, donné un ordre des clients, sélectionne lesquels pour visiter afin d'obtenir la meilleure solution évaluent pour le but donné tout en gardant l'ordre de acheminement original et les contraintes latérales satisfaisantes. Formellement, l'opérateur formule le problème de sélectionner les clients pour visiter dans une certaine tour géant donnée comme un PPCCERL sur un graphe auxiliaire H qui est dirigé et acyclique. Ce graphe auxiliaire représente l'ordre topologique des les n clients contenus dans le tour géant trait. Chaque arc traversé $(i, j) \in H \mid i < j$ indique une consommation de ressource. L'entrée de l'algorithme est une permutation σ d'un ensemble de sommet V qui représente un ensemble de clients. Le dépôt est toujours le premier sommet dans cette permutation. La sortie de l'algorithme est un sous-ensemble $V' \subset V$ à visiter dans le même ordre donné dans σ tels que la valeur de la fonction objectif considérée est optimale et les contraintes de ressource sont satisfaisantes. Il est important de noter qu'une

telle technique de solution implique là n'est aucun besoin de recherche locale basée sur des insertions et des retraits des clients de même que le cas dans les beaucoup heuristique, par exemple, Campos et al. (2014), Vansteenwegen et al. (2009), Gendreau et al. (1997). Au lieu de cela, notre algorithme construit toutes les possibilités d'élagage ceux qui ne peuvent pas conduire à la solution optimale.

A.3.1 Labels

La programmation dynamique est une technique qui établit la solution cherchée d'une mode de bas en haut par la solution des sous-problèmes dont la complexité augmente graduellement. Dans ce cas, le premier sous-problème résolu est celui de trouver le coût ou le bénéfice et la consommation des ressources pour le chemin $\{\sigma_0, \sigma_1\}$. Avec cette information, il est possible de résoudre facilement un sous-problème semblable pour le chemin pour atteindre le sommet σ_2. Également en atteignant le sommet σ_3, et ainsi de suite. Par conséquent, la technique exige du stockage des résultats intermédiaires d'éviter le recomputation. Dans la pratique, ce fait à l'aide des labels. Dans le cas de *Selector*, un label représente un chemin élémentaire faisable que les débuts à σ_0 et a considéré des sommets jusqu'à σ_i. Un label $\lambda = [z, i|r_1, r_2, \ldots, r_m]$ est défini comme triplet qui stocke: la valeur z de la fonction objectif considérée, le grade i a atteint dans σ (dernier sommet visité), et la consommation de ressource $r_1, r_2, \ldots, r_m$ qui permet de savoir un chemin partiel peut être prolongé. Par exemple, dans un problème de couvrant, cette consommation de ressource est interprétée comme que des sommets qui doivent être couverts sont encore découvert, et cette information permet de savoir si un sommet $v_i \in V$ peut être sauté en prolongeant un label. Un sommet σ_i peut être accédé par différents chemins composés de prédécesseurs visites et sautés, de chacun avec le coût différent et d'utilisation différente des ressources. En conséquence, un ensemble de labels Λ^i est associé à chaque sommet σ_i, pour $i \in \{0, \ldots, n-1\}$, commençant par $\Lambda^0 = \{[0, 0, 0]\}$ pour le sommet agissant en tant que dépôt. Une label sur un sommet est prolongée à plusieurs reprises à ses successeurs jusqu'à ce que les restrictions considérées empêchent la création d'labels faisables. Cette opération se répète jusqu'à ce que toutes les labels ont été prolongées dans toutes les manières faisables.

Des répétitions de cette opération jusqu'à tous les labels ont été de toutes les manières. Quand un label est prolongé, sa valeur de la fonction objectif (coût ou bénéfice) et son rang sont calculés utilisant des résultats précédents, et l'information liée aux contraintes latérales (ressources) est mise à jour pour s'assurer qu'une solution faisable est encore possible. Dans chaque itération, le label prolongé est toujours celui qui documente la meilleure valeur de la fonction objectif. Puis, pour n'importe quel i, un ensemble de labels Λ^{i+1} est construit en considérant itérativement n'importe quel arc $(j, i+1) \in H$ et en prolongeant tous les labels de j utilisant une équation de répétition. Puisque chaque extension crée un nouveau label, une manière de commander leur prolifération est en appliquant des relations de dominance entre les paires de labels. Les critères appropriés de dominance laissent identifier les labels dont l'extension ne peut pas produire une solution optimale. Deux labels peuvent être comparés seulement

si chacun des deux ont atteint le même grade i dans σ. En outre, le label de domination doit être moins ou également contraint par l'information de ressource stockée, et doit offrir une meilleure ou égale valeur de la fonction objectif.

A.3.2 Bases de l'Algorithme

Dans un algorithme conçu pour visiter tous les sommets du réseau donné, un label peut seulement être prolongé du sommet σ_i au sommet de successeur σ_{i+1} sans la possibilité de sauter des sommets. Dans un tel cas, l'extension d'un label correspond à apposer un arc supplémentaire $(i, i+1)$ à un chemin de σ_0 à σ_i, obtenant un chemin faisable de σ_0 à σ_{i+1}. Le processus implique également mettre à jour la valeur de la fonction objectif, le dernier sommet atteint $(i+1)$ et la consommation des ressources.

Dans (m)-*Selector*, une opération semblable se produit quand un label est prolongé, mais dans cet algorithme des sommets peuvent être sautés, ainsi nous considérons qu'un label est prolongé du sommet σ_i au sommet σ_{i+k}, où k peut prendre l'un des après les valeurs $\{1, 2, 3, \ldots, n-i-1\}$ avec $n = |V|$. En outre, l'extension d'un label $\lambda = [\zeta, \nu | r_1, r_2, \ldots, r_m]$ du sommet σ_i au sommet de successeur σ_{i+k}, implique qu'une de deux opérations possibles est effectuée: visitez le sommet σ_{i+k}, ou sautez le sommet σ_{i+k}. Si en prolongeant un label il est décidé que le sommet σ_{i+k} est sauté, l'extension procède au sommet σ_{i+k+1} et il est évalué à nouveau s'il est utile de visiter il. Non visite est possible, par exemple, quand un sommet se révèle être redondant ou en visitant il viole une restriction. Cependant, pas des labels sont créées pour les sommets sautés. Seulement quand le sommet est visité, un label dominé par non est stocké.

Quand $k = 1$, l'extension se produit au sommet successeur adjacent immédiat. D'autre part, quand $2 \leq k \leq n-i-1$, l'extension se produit sautant un séquence des sommets. Ceci est possible seulement si les restrictions latérales laissent sauter la séquence $\{\sigma_{i+1}, \ldots, \sigma_{i+k-1}\}$.

Une caractéristique distinctive et importante de (m)-*Selector* est c'hormis les contraintes mentionnées dans la définition du problème, il n'impose pas toute autre restriction aux sommets sélectionnés de V tels que la contiguïté, par exemple. Cet opérateur peut rejeter tout sommet $v_i \in V$ à tout moment de la tournée.

A.4 Générer une Solution Faisable

La méthodologie de solution proposée nécessite établir rapidement une solution faisable. Cette solution est utilisée comme une sonde pour déterminer l'exécution de *Selector* à l'optimalité, et dans quelques problèmes, el est utilisée comme une limite. Il y a plusieurs manières de construire cette solution. Nous avons considéré utilisant une version simplifiée de *Selector*.

L'algorithme simplifié cherche une solution utilisant le même graphe et les même opérations d'extension que le noyau de l'algorithme *Selector*, cependant, il explore seulement certains des chemins possibles pour trouver la solution rapide. Commençant à σ_0 il essaye pour se prolonger λ à $\sigma_i \ \forall i \in \{1, 2, \ldots, n-1\}$, et de σ_i il essayer pour se prolonger λ_i à $\sigma_{i+k} \ \forall k \in \{1, 2, \ldots, n-i-1\}$. Une fois la séquence $\{\sigma_0, \sigma_i, \sigma_{i+k}\}$ est

construit, il continue, si possible, en visitant itérativement le sommet successeur adjacent. Par exemple, il essaye d'abord de construire la séquence $\{\sigma_0, \sigma_1, \sigma_2\}$, et essayera de continuer en se prolongeant à plusieurs reprises au successeur adjacent immédiat. Après, il construit la séquence pour la prochaine valeur de k : $\{\sigma_0, \sigma_1, \sigma_3\}$, et répète encore l'extension aux successeurs adjacents. Une fois que toutes les valeurs de k sont épuisées, il traite le prochain sommet σ_i d'une manière semblable. Le processus finit quand tous les sommets σ_i ont été traités. Chaque tour complèt trouvé est comparé et le meilleur est gardé, aucun labels sont stockées afin de l'exécuter rapidement.

A.5 Amélioration de Performance

Pour maintenir la tractabilité ou accélérer la recherche, trois mécanismes étaient présenté dans *Selector*: (i) prolonger seulement un nombre prédéfini de labels; (ii) calculer une limite inférieure/supérieure, qui est comparée à la meilleure solution faisable en exercice; et (iii) exécuter la recherche bidirectionnelle dans le graphe.

A.5.1 Limiter le Nombre de Labels Prolongés

Dans de l'informatique, une manière commune d'essayer de maintenir la tractabilité est d'employer la *beam search*. Le terme *beam search* a été inventé par Raj Reddy, Carnegie Mellon University, 1976. La *beam search* est une version restreinte d'une recherche en largeur ou d'une meilleur-première recherche, et il est limité dans le sens que la quantité de mémoire disponible pour stocker l'ensemble d'états est limitée, et dans le sens que des états moins-prometteurs peuvent être taillés à n'importe quelle étape dans la recherche par l'heuristique problème-spécifique comme expliqué par Zhang (1999). L'ensemble de la plupart des états prometteurs s'appelle le *beam*. La *beam search* a l'avantage de réduire potentiellement la période d'une recherche. L'inconvénient principal de la *beam search* est que la recherche peut avoir comme conséquence une solution non-optimale. Par conséquent, cette recherche heuristique sacrifie l'optimalité pour la tractabilité. En dépit de cet inconvénient, il peut réaliser un niveau satisfaisant de qualité de solution, et a trouvé le succès dans les secteurs tels que l'apprentissage automatique et la reconnaissance de la parole (Zhang, 1999).

Dans *Selector*, nous ne traitons pas une pénurie de mémoire, mais nous visons à limiter le nombre de labels prolongés afin de maintenir la tractabilité. Les notions de le *beam* et la *beam width* sont appliqué dans *Selector* dans le sens que seulement un nombre prédéfini (le beam width) des labels stockés les plus prometteurs (le beam) sont prolongé. En d'autres termes, le beam est la queue d'attente de recherche parce que seulement des labels de promesse sont stockés, et seulement un nombre prédéfini de labels stockés dans cette queue d'attente sont prolongé. Des labels de promesse sont identifiés par un mécanisme de bondissement, comme expliqué dans la section 3.5.2. En d'autres termes, c'est un processus de recherche qui prolonge le label le plus prometteur dans un ensemble limité. De cette façon un temps de recherche raisonnable est garantie puisque des calculs sont limités par une valeur connue.

Ce type de recherche exige alors que l'algorithme connaît les règles pour l'élagage labels et la beam width. Cette technique a été appliquée à l'algorithme de solution pour l'OP.

A.5.2 Calculer une Limite Inférieure/Supérieure

Pour élaguer les états non-prometteurs exige d'abord les identifier. C'est possible utilisant une limite. Dans *Selector*, calculant une limite atteint deux objectifs d'élagage : éviter de stocker et éviter de prolonger les labels non-prometteurs. La fonction suivante peut être employée pour estimer la valeur de la fonction objectif d'un chemin complet

$$\mu(\lambda) = z(\lambda) + h(\lambda) \tag{A.1}$$

là où

- $\lambda = [z, \sigma_i | r_1, r_2, \ldots, r_m]$ représente le label qui mémorise σ_i comme le dernier sommet visité sur le chemin a représenté.
- $z(\lambda)$ représente la valeur réelle de la fonction objectif du label (dès le début sommet σ_0 à sommet σ_i).
- $h(\lambda)$ représente la limite inférieure/supérieure calculée sur la fonction objectif visitant seulement des sommets dans le subsequence $\varphi = \{\sigma_{i+1}, \ldots, \sigma_{n-1}\}$. Ceci peut être une limite inférieure du coût pour couvrir les sommets restants ou une limite supérieure du bénéfice recueilli à partir des sommets qui se trouvent devant σ_i.

Équation A.1 est inspiré dans l'algorithme d'A* utilisé dans de l'informatique pour le recherche de un chemin et le traversal de un graphe. Hart et al. (1968) de Stanford Research Institute (maintenant SRI International) a décrit la première fois l'algorithme. C'est une extension de l'algorithme du plus court chemin de Dijkstra utilisant l'heuristique pour guider sa recherche. À chaque itération de sa boucle principale, A* doit déterminer lesquels de ses chemins partiels à augmenter dans un plus long chemin. Il fait ainsi basé sur une estimation du coût (poids total) reste à parcourir pour le sommet de but. Spécifiquement, A* sélectionne le chemin qui réduit au minimum l'équation $f(n) = g(n) + h(n)$ où n est le dernier sommet sur le chemin, $g(n)$ est le coût du chemin à partir du sommet de début à n, et $h(n)$ est le coût estimé par une heuristique spécifique au problème du chemin le moins cher à partir de n au but.

Reprenant la mécanique de *Selector*, pour résoudre l'équation A.1, nous devons seulement trouver la valeur de la limite $h(\lambda)$. Naturellement, les sommets que nous voulons visiter sont ceux avec un bon rapport de $\frac{avantage}{coût}$.

Ensuite, la valeur de la limite $h(\lambda)$ peut être trouvée par la solution de un *Problème de Sac Partiel* (PSP), i.e. la relaxation linéaire d'un Problème du Sac 0-1, quel complexité est polynôme, $O(n \log n)$, prenant en considération le tri des articles considérés. Afin de résoudre ce problème de sac, il est nécessaire de définir d'abord tout les objets, leur bénéfice et leur poids. Un objet est créé pour chaque sommet dans φ. Le bénéfice

p_i d'un objet est donné par le nombre de sommets qu'il couvre ou par le bénéfice sa visite rapporte. Le poids w_i d'un objet est calculé comme suit. Soit $\Delta = \{w_{ji}\}_{j=0}^{i-1}$ l'ensemble des poids des arêtes d'entrée de σ_i, ceux qui se connectent le sommet σ_i avec chacun de ses sommets prédécesseurs dans σ, et soit $\Delta' = \{w_{ik}\}_{k=i+1}^{n-1} \cup \{w_{i0}\}$ l'ensemble des poids des arêtes sortantes de σ_i, ceux qui se connectent le sommet σ_i avec chacun de ses sommets successeurs dans σ. Ensuite, une limite inférieure du coût de voyage (poids) de visiter le sommet σ_i peut être obtenue par

$$w_i = \min \Delta + \min \Delta' \tag{A.2}$$

Dans le cas où il y a égalité entre les deux ensembles — le sommet prédécesseur est le même que le successeur — le deuxième meilleur poids global de arête est choisi. Le PSP est résolu avec l'algorithme avide classique de Dantzig (1957). Le rapport $bénéfice/poids$, $\frac{p_i}{w_i}$, services pour déterminer les meilleurs sommets pour visiter. Commandez les sommets dans l'ordre non-croissant en fonction de leur rapport $bénéfice/poids$, et sélectionnez-les dans l'ordre jusqu'à ce que la capacité du sac est dépassée. Dans ce cas, la capacité du sac peut représenter le nombre de clients encore pour couvrir ou la longueur du chemin qui reste hors du maximum permis. Néanmoins, dans le PSP, nous ne devons pas sélectionner tout le bénéfice d'un sommet, mais plutôt peut prendre n'importe quelle fraction de celle-ci. Ainsi, pour le dernier sommet choisi, nous pouvons seulement inclure la fraction du bénéfice qui s'adapte dans la capacité restante. De cette façon l'algorithme avide ne gaspille jamais n'importe quelle capacité, et en conséquence, il rapporte toujours une solution optimale pour le PSP comme dans expliqué Neapolitan and Naimipour (1998). Selon la fonction objectif considérée, la valeur de la limite $h(\lambda)$ est alors la somme des valeurs de bénéfice ou de poids des sommets choisis. Si la valeur estimée de la fonction objectif (limite) d'un chemin indiqué est plus mauvaise que la solution la plus connue, la recherche dans cette trajectoire doit être abandonnée. Ainsi, la fonction $\mu(\lambda)$ doit être calculée à chaque étape de l'extension d'un label afin de déterminer si le label est prometteur. La valeur de la fonction $\mu(\lambda)$ est stockée dans un champ supplémentaire de le label et il est appliqué encore de la même manière quand le label est récupéré pour l'extension. Ce test est utile parce que la valeur de la solution la plus connue pourrait avoir changé depuis que le label a été stocké. Cette technique a été appliquée aux algorithmes de solution pour le CTP et l'OP.

A.5.3 Recherche Bidirectionnelle

Pohl (1971) était le premier pour concevoir et mettre en application un algorithme de recherche heuristique bidirectionnel. Dans ce genre d'algorithme, deux processus de recherche sont effectués simultanément: on commence à partir de σ_0 et considère σ dans l'ordre donné, et l'autre également commence à σ_0 mais considère que la séquence obtenu en inversant σ. Quand les deux frontières de recherche intersectent, l'algorithme peut reconstruire un chemin unique qui prolonge dès le sommet de départ par l'intersection de la frontière au le sommet de but.

Le raisonnement derrière employer la recherche bidirectionnelle est le suivant. L'algorithme de programmation dynamique présenté produit d'un certain nombre de labels qui augmente rapidement avec la taille du problème actuel. Chaque fois un label λ est prolongé de σ_i, il produit d'autant d'autres labels comme nombre de successeurs possibles de σ_i. Par conséquent, dans le pire des cas, le nombre de labels se développe exponentiellement avec le nombre de sommets dans le chemin. En raison de cette dépendance exponentielle à l'égard le nombre d'étapes, il est intuitif que produire des chemins plus courts puisse rapporter un avantage significatif en termes de nombre de labels considérés. Ceci est précisément l'effet de la recherche bidirectionnelle avec le bondissement, quel but est de limiter la longueur des chemins a considéré tout au plus à la moitié de la longueur du chemin optimal.

Désormais, il est expliqué comment les principales caractéristiques de performance de ce type de recherche ont été mises en application. Le processus de recherche à exécuter à chaque itération doit être sélectionné. Bien que itérer à parts égales entre les deux recherches — vers l'avant et vers l'arrière — serait la méthode la plus simple, il n'est pas le plus efficace. La meilleure stratégie est d'identifier le chemin (label) avec la meilleure valeur de la fonction objectif jusqu'à présent. C'est-à-dire, pendant chaque itération, concentrer l'effort informatique sur la recherche ayant le meilleur chemin.

Un nouveau problème surgit pendant une recherche bidirectionnelle, à savoir s'assurant que les deux frontières de recherche se réunissent réellement. Pour cette raison, à chaque itération, l'occurrence de chemins complets est surveillée. Chaque fois un label dominé par non est stocké, la queue de recherche opposée est vérifiée pour déterminer s'il y a un label qui peut être joint à lui pour former une solution complète. Ainsi, deux labels de la direction opposée se réunissent quand les derniers sommets visités s'assortissent, et il est alors examiné s'ils peuvent former une solution faisable complète. Si le test est positif, cette solution est comparée contre le titulaire le plus connu et ce dernier est mis à jour si nécessaire. Les deux labels examinés sont gardés quel que soit le résultat de ce test, puisqu'avec une autre moitié opposée ils peuvent construire une meilleure solution.

La politique de terminaison est cruciale pour réduire de manière significative le temps informatique. La version de monodirectional procède jusqu'à ce que tous les labels sont traités, mais l'application de ce critère à la version bi-directionnel peut produire une performance égale ou pire à celle de la version unidirectionnelle pour certains cas.

Le critère classique de terminaison d'une recherche bidirectionnelle sur un problème de minimisation compare la somme des plus bas coût réel de l'avant, plus le coût le plus bas en arrière réelle avec le coût de la solution la plus connue, et arrêts si la première est supérieure ou égale à ce dernier.

Cependant, dans notre cas, des labels sont organisés par le coût estimatif $\mu(\lambda)$, de sorte que le coût réel minimum correspondant n'est pas disponible dans complexité $O(1)$. Pour cette raison, nous avons adapté le critère de terminaison comme suit: si l'équation A.3 est vraie et les deux labels qui sont conformes à cette équation constituent une solution faisable, alors l'algorithme arrête la recherche.

$$\min_{\sigma_i \in S \setminus \{\sigma_0\}} \{\mu_i(\lambda)^{enavant}\} + \min_{\sigma_i \in S \setminus \{\sigma_0\}} \{\mu_i(\lambda)^{versl'arrière}\} \geq LeCoûtLePlusConnu \quad \text{(A.3)}$$

Si les deux labels forment une solution faisable, la limite inférieure calculée par l'équation A.3 donne la valeur de coût minimum qui peut être obtenu pour une solution faisable complète. Si cette valeur minimum est pire que la solution la plus connue, il est certain que notre valeur de solution ne s'améliorera pas et nous pouvons arrêter la recherche. Cette technique a été appliquée à l'algorithme de solution pour le CTP.

A.6 *Selector* de Multi-Véhicule

La première idée qui surgit lors de la résolution d'un multi-PTVVO avec *Selector* est de canaliser la sortie de *Selector* à l'entrée de la opérateur *Split*, une approche de type sélectionner première–regrouper deuxième. Il a été certainement essayé, mais cela a seulement fonctionné pour des valeurs très petites du nombre de sommets qui peuvent être visités, $|V| \leq 25$. Il a été montré que une sélection optimale des visites suivies d'une partition optimale ne conduit pas nécessairement à un ensemble optimal de routes de véhicules. Par conséquent, l'un ou l'autre la sélection et la partition sont effectuées simultanément, ou une heuristique de recherche locale sélectionne d'abord les sommets, et puis les divise dans les routes. Nous avons choisi de développer l'idée nouvelle d'un *m-Selector* qui choisit les meilleurs sommets à visiter et résout en même temps comment ces sommets peuvent être arrangés de façon optimale dans des routes.

La solution du problème de partition exige insérer des visites au dépôt afin d'évaluer le coût total de segmenter la tournée donnée dans les routes de véhicule faisables, comme il est fait par l'algorithme *Split*. Dans le cas du l'opérateur *Selector* de multi-véhicule, ceci exigerait la création encore plus des labels que ceux produits par la version à un seul véhicule. Etant donné la dépendance exponentielle la croissance du nombre de labels a avec la taille du chemin recherché, l'idée d'ajouter plus de labels n'est pas attrayante. Au lieu de cela, une manière de faire ces visites implicitement a été cherchée, et il est venu sous la forme d'utiliser l'algorithme *Split* comme la fonction de coût. Par conséquent, au lieu d'utiliser la fonction simple d'ajouter des coûts de arêtes de la version à un seul véhicule, une version modifiée de l'algorithme *Split* a été mis en œuvre afin de calculer le coût optimal de segmenter les sommets sélectionnés dans les routes faisables. Une version modifiée de *Split* est dans le sens que la segmentation de la tournée de PTVVO doit se produire point par point pendant que des sommets sont ajoutés à elle. C'est de dire, les voyages représentés dans le graphique H de l'algorithme *Split* sont construits comme les sommets sont sélectionnés. Dans *Selector*, le coût d'un label est le coût d'assigner les sommets sélectionnés à un seul route, tandis que dans *m-Selector* le coût est celui de les assigner à plusieurs itinéraires. Essentiellement, la même notion. Si lors de la sélection d'un sommet, il est également évalué comment l'ensemble sélectionné peut être segmenté en routes faisables, le coût en résultant considère vraiment les deux niveaux de prise de décision en même temps.

Puisque dans *m-Selector* les deux processus de décision—sélectionner et segmenter—sont faits par des méthodes précises, la sortie est optimale. L'hypothèse que tous les clients ont la même demande l'a facilité pour mettre en application l'idée. L'intégration de l'algorithme *Split* dans le *Selector* afin de calculer le coût d'un label est plus informatiquement exigeant que le calcul de l'addition simple du coût d'une arête, ainsi les manières de réaliser la bonne performance étaient également une question qui a exigé un examen consciencieux.

La structure du label de *m-Selector* demeure fondamentalement la même, cependant les deux vecteurs que l'algorithme *Split* utilise pour maintenir le coût de voyage et du sommet prédécesseur pour chaque client sélectionné ont été incorporés. La règle de dominance ajoute également un test afin de considérer la capacité de véhicule, et la performance améliore quand plusieurs méthodes pour construire le premièr tour géant ont été rendues disponibles. Par conséquent, le fusionnement de les deux processus (*Selector* et *Split*) a eu comme conséquence peu de changements à la conception originale de l'algorithme, de sorte que la mise en oeuvre résultant de *m-Selector* a un noyau qui est très semblable à celui de la version originale.

A.7 Le Cadre du Méta-heuristic

Le deuxième composant de l'approche de solution présentée dans cette étude est le mécanisme de routage. Ce travail de thèse propose d'utiliser l'algorithme de recherche adaptatif à grand voisinage (RAGV): un cadre de recherche locale qui utilise plusieurs sous-heuristiques de destruction et de construction qui concurrencent et il choisit parmi eux utilisant des statistiques recueillies pendant la recherche.

La figure A.1 montre comment les composants de routage et de groupement proposés interagissent pour résoudre un problème de minimisation. L'algorithme commence par une tour géant initial σ^{init} qui peut être produite de manière aléatoire ou par l'intermédiaire d'une heuristique de construction. Le tour σ^{init} subit un processus de recherche locale *2-opt* pour améliorer sa longueur rapidement. Puis, dans chaque itération, l'algorithme considéré un tour géant σ et a à sa disposition plusieurs sous-heuristiques de destruction et de construction pour le modifier. L'algorithme sélectionne d'abord, avec une probabilité adaptatif, une sous-heuristique de destruction qui élimine quelques clients du tour géant, et produit σ^{nouveau}. Puis, encore avec une probabilité adaptatif, il sélectionne un sous-heuristique de réparation qui insère les clients de retour, mais choisissant de meilleurs endroits de telle sorte que la longueur du tour geant est plus loin améliorée. Au-dessus de cette nouveau tour géant σ^{nouveau}, il cherche une solution faisable (une limite) utilisant une version réduite de l'opérateur *Selector*. Pour rechercher cette solution efficacement, l'opérateur analyse seulement un sous-ensemble des possibilités. Puis, si la valeur de la solution réalisable est améliorée, le *Selector* est exécuté, cette fois à l'optimalité, avec l'espoir d'obtenir une meilleure solution de PTVVO à partir de σ^{nouveau}. Pour éviter de faire cette étape très stricte, la valeur de la limite la plus connue est multipliée par une valeur pratique de α. L'algorithme contient également un système pour éviter la stagnation du processus

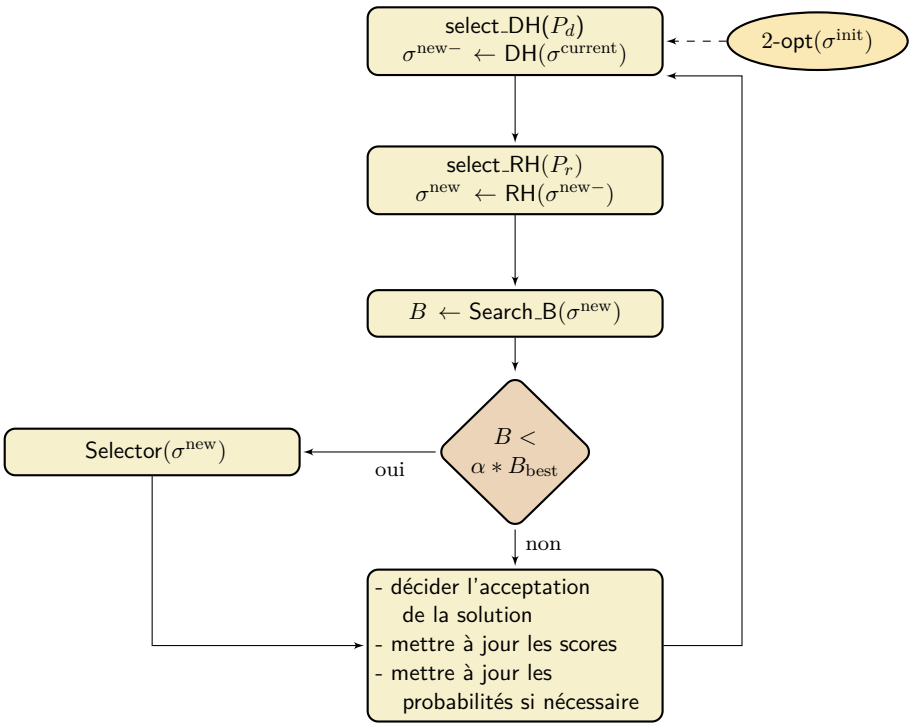

Figure A.1: Composants de routage et de groupement de la méthodologie unifié de solution.

de recherche: recuit simulé (RS). En plus de cette, la couche adaptatif exige mettre à jour les scores des sous-heuristique utilisées dans l'itération, et, chaque période de mise à jour, la probabilité de sélection de chaque sous-heuristique. Le processus se répète jusqu'à la critère de rupture est atteint.

Les améliorations sur les coût de voyage du tour géant ne mènent pas nécessairement à une meilleure valeur de la fonction objective en le tournée calculée par le *Selector*. En outre, un tour géant optimal ne rapporte pas nécessairement une valeur de solution de PTVVO optimale. Cependant, à longue échéance, le tournée de PTVVO bénéficie d'améliorations sur la longueur du tour géant, puisque plus le tour est court, plus les sommets supplémentaires peuvent être mis dans il. Une conséquence importante de cette indépendance est que l'exécution du *Selector* à l'optimalité à chaque itération est de bas avantage. La limite calculée peut servir de sonde pour déterminer si le processus complet vaut l'exécution. Ceci dérive dans l'épargne importante de temps. Nous suivons les trois bases expliquées par Pis pour expliquer les éléments principaux de la mise en œuvre de RAGV: (i) les sous-heuristiques de destruction, (ii) les sous-heuristiques de construction, et (iii) le metaheuristic qui définit les critères pour accepter une nouvelle solution. Trois sous-heuristiques de destruction et trois sous-heuristiques de construction ont été mises en œuvre. Les paramètres du méta-heuristique ont été accordés utilisant le paquet **irace** mis en œuvre en R et développé par López-Ibáñez et al. (2011), Iterated Racing for Automatic Algorithm Configuration. Il met en œuvre la procédure de course itérative, une extension de F-course

Paramètre	Signification
γ	le nombre de sommets enlevés à chaque itération de RAGV (dépend de la taille de l'instance)
ς	la taille de la période pour mettre à jour des probabilités en terme des itérations de RAGV
τ	le facteur de réaction qui commande le taux de changement de l'ajustement de poids
δ	évite le déterminisme dans le HSS
ρ	évite le déterminisme dans le HSP
κ_1	le score donné pour trouver une nouvelle meilleure solution globale
κ_2	le score donné pour trouver une nouvelle solution qui est meilleure que l'actuelle
κ_3	le score donné pour trouver une nouvelle solution de non-amélioration qui est acceptée
β	facteur de refroidissement employé par le recuit simulé
ϵ	fixe la limite supérieure des sommets enlevés à chaque itération

Table A.1: Les paramètres de RAGV sont réglés avec le software **irace** mis en œuvre en R par López-Ibáñez et al. (2011).

Itérative. Dans le suivant, les lettres grecques en bas de casse indiquent les paramètres dépendants de l'utilisateur documentés dans le Tableau A.1.

A.7.1 Les Sous-heuristiques de Destruction

L'Heuristique de Suppression de Shaw (HSS). Initialement proposé par Shaw (1997), son idée générale est de supprimer les sommets qui montrent la similitude, caractéristique calculée par une *mesure de connexité* $R(i, j)$. Nous mesurons la similitude entre deux sommets par $R(i, j) = d_{ij}$, où d_{ij} est la distance euclidienne entre les sommets σi et σ_j. Le $R(i, j)$ inférieur est, plus les deux sommets sont plus connexes. Cette mesure de connexité est utilisée pour supprimer les sommets de la même manière que celle décrite par Shaw (1998).

L'Heuristique de Suppression des Pires (HSP). Ropke and Pisinger (2006) proposer une heuristique qui retire de façon aléatoire les sommets avec un coût élevé dans la solution actuelle σ et il essaie de les insérer dans de meilleures positions. Soit $coût(i, \sigma) = f(\sigma) - f_{-i}(\sigma)$ le coût associé au sommet σ_i dans la solution actuelle σ, où $f_{-i}(\sigma)$ est le coût de la solution sans le sommet σ_i. Des sommets sont d'abord assortis selon $coût(i, \sigma)$ et ensuite un est aléatoirement choisi pour être retiré. Le processus réitère recalculer les coûts, $coût(i, \sigma)$, jusqu'à ce qu'il ait retiré le nombre de sommets indiqués.

L'Heuristique de Suppression Aléatoire (HSA). Cette procédure sélectionne simplement γ sommets au hasard et les retire de la solution actuelle σ. Bien qu'il tende à produire d'un ensemble pauvre de membres enlevés, il est utile de diversifier la recherche.

Combien à Supprimer. Le paramètre γ indique le nombre d'éléments supprimés à partir de la taille complète de la solution. Ce paramètre γ est choisi de manière aléatoire entre une limite inférieure et une limite supérieure. La limite inférieure est fixée à une valeur donnée en fonction du nombre de sommets dans la solution *sigma*, tandis que la limite supérieure est affiné avec le paramètre ϵ. Par conséquent, l'algorithme fonctionne avec un degré randomisé de destruction dans l'intervalle $[0,3 \times |V|,\ \epsilon \times |V|]$.

A.7.2 Les Sous-heuristiques de Réparation

Le Meilleur Heuristique Glouton (MHG). Cet heuristique de construction simple effectue au plus γ itérations comme il insère un sommet dans la solution σ à chaque itération. La valeur de la position de coût minimum est calculé pour tous les sommets attendant l'insertion —set$F$— et la position de coût minimum global est choisie. Ce processus est répété jusqu'à ce que $F = \emptyset$.

Le Premier Heuristique Glouton (PHG). Cet heuristique fonctionne de façon similaire à la précédente. Cependant, au lieu d'insérer le sommet ayant la position du coût global minimum, il insère le sommet situé dans la première position. Autrement dit, il respecte l'ordre des sommets dans F. Après le premier sommet a-été insérée, la position de coût minimum pour chaque sommet est recalculée et le processus se répète jusqu'à tous les sommets dans l'ensemble F ont été insérées.

Ropke and Pisinger (2006) ajoutent un terme de bruit à la fonction objective pendant la phase d'insertion du MHG et du heuristique regret-k afin de les randomiser et évitez de faire toujours le mouvement qui semble le mieux localement. Dans notre implémentation, le PHG est principalement utilisé pour introduire ce bruit dans le processus d'insertion comme fait par Ribeiro and Laporte (2012). Cet heuristique fonctionne évidemment plus rapidement que le MHG.

L'Heuristique Regret (HRK). Cet heuristique tente d'améliorer le comportement myope des heuristiques gloutonnes en incorporant une sorte de regarder en avant l'information lors de la sélection du sommet à insérer, comme fait par Ropke and Pisinger (2006) and Pisinger and Ropke (2007). Soit Δf_i^1 représentent la variation de la longueur du tour encourue par l'insertion du sommet σ_i à sa position de coût minimum, et Δf_i^2 représentent le changement en l'insérant à sa deuxième meilleure position. La valeur de regret est définie comme $c_i^* = \Delta f_i^2 - \Delta f_i^1$. À chaque itération, l'heuristique insère le sommet σ_i qui maximise la valeur de regret c_i^* à sa position de coût minimum. Les égalités sont en sélectionnant le sommet avec le insertion de la plus faible coût. Parlant officieusement, il choisit l'insertion que nous regretterons les

la plupart si elle n'est pas faite maintenant. C'est un opérateur qui prend du temps, mais les calculs inutiles ont été évités lors du calcul de Δf_i^n.

Choix d'Une Paire Heuristique de Destruction-Réparation. Afin de sélectionner un heuristique, des poids leur sont assignés et un principe de *sélection de roue de roulette* est appliqué par la couche adaptative. Soit $D = \{d_i | i = 1, \ldots, 3\}$ l'ensemble des sous-heuristiques de destruction et $R = \{r_i | i = 1, \ldots, 3\}$ l'ensemble des sous-heuristiques de réparation. Les poids des heuristiques sont dénotés $w(d_i)$ et $w(r_i)$ respectivement, de sorte que les probabilités pour sélectionner l'un sont

$$p(d_i) = \frac{w(d_i)}{\sum_{j=1}^{3} w(d_j)}, \quad p(r_i) = \frac{w(r_i)}{\sum_{j=1}^{3} w(r_j)} \tag{A.4}$$

L'heuristique de destruction est sélectionné indépendamment de l'heuristique de réeparation et vice versa. Initialement, tous les heuristiques sont également susceptibles d'être choisis, par exemple, $w(d_i) = 1 \quad \forall d_i \in D$.

Ajustement Adaptatif de Poids Soit ς (déterminée expérimentalement) désignent les périodes de mise à jour dans laquelle le nombre total d'itérations ALNS est divisé, et h dénotent un heuristique de réparation ou de destruction. Pour permettre l'ajustement de poids, un score $s(h)$ est mémorisé pour chaque heuristique, et il est mis à jour à chaque itération par une quantité égale aux paramètres kappa où k $\in \{1, 2, 3\}$, quand il identifie de nouvelles solutions, voir le Tableau A.1. Pour des ajustements raisonnables l'inégalité $\kappa_1 > \kappa_2 > \kappa_3$ est assurée. Puis, à la fin de chaque période de mise à jour, ces scores enregistrés sont employés pour calculer de nouveaux poids et probabilités s'y rapportant. En outre, tous les poids sont remis à zéro au début de chaque période. Depuis deux heuristiques sont appliqués à chaque itération, les scores pour les deux sont mis à jour par le même montant.

Les nouveaux poids sont calculés comme suit. Soit $w(h_i)_j$ le poids de l'heuristique i à la période de mise à jour j. Après la période j finitions, le nouveau poids à utiliser dans la période $j + 1$ pour l'heuristique h_i est donnée par

$$w(h_i)_{j+1} = \begin{cases} w(h_i)_j(1-\tau) + \tau\dfrac{s(h_i)}{u(h_i)}, & \text{if } u(h_i) > 0 \\ w(h_i)_j(1-\tau), & \text{if } u(h_i) = 0 \end{cases} \tag{A.5}$$

où $s(h_i)$ est le score de l'heuristique h_i obtenu au cours de la dernière période et $u(h_i)$ est le nombre de fois heuristique h_i a été utilisé au cours de cette même période. τ est connu comme le facteur de réaction, et il commande à quelle rapidité le mécanisme d'ajustement de poids réagit aux changements de l'efficacité de l'heuristique. Cette mise en oeuvre assure le suivi des solutions visitées en utilisant une table de hachage. Une clé de hachage est associée à chaque solution, et cette clé est stockée dans la table.

A.7.3 Le Recuit Simulé Guide la Recherche

Le recuit simulé (RS) est le metaheuristic externe qui guide la recherche. RS appliquée aux problèmes d'optimisation émerge du travail de Kirkpatrick et al. (1983). Il a eu un impact majeur sur le champ de la recherche heuristique pour sa simplicité et son efficacité en résolvant ded problèmes d'optimisation combinatoire comme présenté dans Talbi. RS est un algorithme stochastique qui permet, dans certaines conditions, la dégradation d'une solution avec l'objectif à échapper des optimums locaux et pour retarder ainsi la convergence. Puis, RS implique d'accepter non seulement les solutions qui sont meilleures que la solution actuelle, mais plutôt, occasionnellement, accepte les solutions qui sont plus mauvaises que le actuelle.

A.8 Conclusions

La méthodologie de solution proposée émule un heuristique constructive connu comme route first–cluster second, ainsi il se compose d'un procédé de séparation (groupement) et d'une méthode qui construit un tour. La méthode de séparation (groupement) proposée est l'opérateur *Selector* qui est un algorithme basé sur la programmation dynamique visant pour résoudre PTVVO. L'algorithme est une adaptation de celui développée par Desrochers (1988) dans le contexte du PPCCRL. Il est un algorithme atteignant de correction de label. L'opérateur *Selector* est intégré dans un métaheuristique RAGV, qui est la méthode qui construit un tour, cette méthode accomplit la tâche d'assigner l'ordre de visite des n clients donnés. De cette séquence construite, *Selector* recherche de façon optimale ceux à visiter. Le problème de sélectionner les clients visités est formulé comme un PPCCERL sur un graphe auxiliaire acyclique et dirigé où les restrictions du problème considéré agissent en tant que les ressources de contrainte. Ce graphe auxiliaire représente l'ordre topologique des n clients contenues dans la tour géant traité.

Le principe de base de l'algorithme est d'associer à chaque chemin partiel du sommet de dépôt σ_0 à un sommet $\sigma_i \in H$ un label représentant le coût du chemin et sa consommation des ressources et éliminer les labels inutiles à l'aide des règles de dominance comme la recherche progresse. Des labels sont itérativement prolongés de toutes les manières faisables jusqu'à ce que plus de labels ne puissent être créés, alors que le meilleur chemin en exercice est mis à jour dans tout le processus de recherche. L'extension d'un label à un sommet visité correspond à (1) ajouter un arc supplémentaire $(i, i+k)$ à un chemin de σ_0 à σ_i, obtenant un chemin faisable de σ_0 à σ_{i+k}; et (2) mise à jour le coût du chemin et de ses consommations des ressources. Puisque le nombre de labels produits au cours de la recherche a une dépendance exponentielle de la taille du problème actuel, divers mécanismes visant à limiter la prolifération des labels ont été ajoutés à l'algorithme de base. En outre, une version multi-véhicule qui co-travaux avec le *Split* opérateur a également été proposé.

Bibliography

R.K. Ahuja, Ö. Ergun, J.B. Orlin, and A.P. Punnen. A survey of very large-scale neighborhood search techniques. *Discrete Applied Mathematics*, 123:75–102, 2002.

D. Aksen, O. Kaya, F.S. Salman, and Ö. Tüncel. An adaptive large neighborhood search algorithm for a selective and periodic inventory routing problem. *European Journal of Operational Research*, 239:413–426, 2014.

S. Allahyari, M. Salari, and D. Vigo. A hybrid metaheuristic algorithm for the multi-depot covering tour vehicle routing problem. *European Journal of Operational Research*, 242:756–768, 2015.

D.L. Applegate, R.E. Bixby, V. Chvátal, and W.J. Cook. *The Traveling Salesman Problem: A Computational Study*. Princeton University Press, 2007.

C. Archetti and M.G. Speranza. A survey on matheuristics for routing problems. *EURO Journal on Computational Optimization*, 2:223–246, 2014.

C. Archetti, D. Feillet, A. Hertz, and M.G. Speranza. The capacitated team orienteering and profitable tour problems. *Journal of the Operational Research Society*, 60: 831–842, 2009.

C. Archetti, M.G. Speranza, and D. Vigo. *Vehicle Routing: Problems, Methods and Applications*, chapter Vehicle routing problems with profits, pages 273–298. SIAM, Philadelphia, USA, 2014.

B. Awerbuch, Y. Azar, A. Blum, and S. Vempala. New approximation guarantees for minimum-weight k-trees and prize-collecting salesmen. *SIAM Journal on Computing*, 28:254–262, 1998.

N. Azi, M. Gendreau, and J-Y. Potvin. An adaptive large neighborhood search for a vehicle routing problem with multiple routes. *Computers & Operations Research*, 41:167–173, 2014.

E. Balas. The prize-collecting travelling salesman problem. *Networks*, 19:621–636, 1989.

E. Balas and A. Ho. Set covering algorithms using cutting planes, heuristics, and subgradient optimization: A computational study. *Mathematical Programming*, 12: 37–60, 1980.

R. Baldacci, M.A. Boschetti, V. Maniezzo, and M. Zamboni. *Metaheuristic Optimization Via Memory and Evolution*, chapter Scatter search methods for the covering tour problem, pages 59–91. Kluwer, Boston, USA, 2005.

J. Beasley. Route first-cluster second methods for vehicle routing. *OMEGA The International Journal of Management Science*, 11:403–408, 1983.

J. Beasley and E.M. Nascimento. The vehicle routing-allocation problem: a unifying network. *Top*, 4:65–86, 1996.

R. Bellman. *Dynamic programming*. Princeton University Press, New Jersey, USA, 1957.

C. Blum and A. Roli. Metaheuristics in combinatorial optimization: Overview and conceptual comparison. *ACM Computing Surveys*, 35:268–308, 2003.

C. Blum, M.J.B. Aguilera, A. Roli, and M. Samples, editors. *Hybrid Metaheuristics An Emerging Approach to Optimization*. Springer-Verlag, Berlin, Germany, 2008.

C. Blum, J. Puchinger, G.R. Raidl, and A. Roli. Hybrid metaheuristics in combinatorial optimization: A survey. *Applied Soft Computing*, 11:4135–4151, 2011.

B Boffey, F.R. Fernández-García, G. Laporte, J.A. Mesa, and Pelegrín-Pelegrín B. Multiobjective routing problems. *Top*, 3:167–220, 1995.

N. Boland, J. Dethridge, and I. Dumitrescu. Accelerated label setting algorithms for the elementary resource constrained shortest path problem. *Operations Research Letters*, 34:58–68, 2006.

M.-C. Bolduc, J. Renaud, F. Boctor, and G. Laporte. A perturbation metaheuristic for the vehicle routing problem with private fleet and common carriers. *Journal of the Operational Research Society*, 59:776–787, 2008.

H. Bouly, D.-C. Dang, and A. Moukrim. A memetic algorithm for the team orienteering problem. *4OR*, 8:49–70, 2010.

M. L Brown and L. Fintor. US screening mammography services with mobile units: results from the national survey of mammography facilities. *Radiology*, 195:529–532, 1995.

S.E. Butt and T.M. Cavalier. A heuristic for the multiple tour maximum collection problem. *Computers & Operations Research*, 21:101–111, 1994.

V. Campos, R. Martí, J. Sánchez-Oro, and A. Duarte. GRASP with path relinking for the orienteering problem. *Journal of the Operational Research Society*, 65:1800–1813, 2014.

I-M. Chao, B.L. Golden, and E.A. Wasil. The team orienteering problem. *European Journal of Operational Research*, 88:464–474, 1996a.

I-M. Chao, B.L. Golden, and E.A. Wasil. A fast and effective heuristic for the orienteering problem. *European Journal of Operational Research*, 88:475–489, 1996b.

M. Chiarandini, I. Dumitrescu, and T. Stützle. *Hybrid Metaheuristics An Emerging Approach to Optimization*, chapter Very large-scale neighborhood search: overview and case studies on coloring problems, pages 117–150. Springer-Verlag, Berlin, Germany, 2008.

C.-W. Chu. A heuristic algorithm for the truckload and less-than-truckload problem. *European Journal of Operational Research*, 165:657–667, 2005.

G. Clarke and J.W. Wright. Scheduling of vehicles from a central depot to a number of delivery points. *Operations Research*, 12:568–581, 1964.

J.R. Current and D.A. Schilling. The covering salesman problem. *Transportation Science*, 23:208–213, 1989.

J.R. Current and D.A. Schilling. The median tour and maximal covering problems. *European Journal of Operational Research*, 73:114–126, 1994.

G. Dantzig, R. Fulkerson, and S. Johnson. Solutions of a large-scale traveling-salesman problem. *J. Opns. Res. Soc.*, 2:393–410, 1954.

G.B. Dantzig. Discrete-variable extremum problems. *Operations Research*, 5:266–288, 1957.

G.B. Dantzig and J.H. Ramser. The truck dispatching problem. *Management Science*, 6:80–91, 1959.

B. De Backer, V. Furnon, P. Shaw, P. Kilby, and P. Prosser. Solving vehicle routing problems using constraint programming and metaheuristics. *Journal of Heuristics*, 6:501–523, 2000.

W.A. Dees Jr and P.G. Karger. Automated rip-up and reroute techniques. In *Proceedings of the 19th Design Automation Conference*, pages 432–439. IEEE Press, 1982.

M. Dell'Amico, F. Maffioli, and P. Värbrand. On prize-collecting tours and the asymmetric travelling salesman problem. *International Transactions in Operational Research*, 2:297–308, 1995.

E. Demir, T. Bektaş, and G. Laporte. An adaptive large neighborhood search heuristic for the pollution-routing problem. *European Journal of Operational Research*, 223: 346–359, 2012.

M. Desrochers. An algorithm for the shortest path problem with resource constraints. *Technical Report G-88-27, GERAD*, 1988.

A. Divsalar, P. Vansteenwegen, K. Sörensen, and D. Cattrysse. A memetic algorithm for the orienteering problem with hotel selection. *European Journal of Operational Research*, 237:29–49, 2014.

K. Doerner, A. Focke, and W.J. Gutjahr. Multicriteria tour planning for mobile healthcare facilities in a developing country. *European Journal of Operational Research*, 179:1078–1096, 2007.

K.F. Doerner and R.F. Hartl. *The Vehicle Routing Problem: Latest Advances and New Challenges*, chapter Health care logistics, emergency, preparedness and disaster relief: New challenges for routing problems with a focus on the Austrian situation, pages 527–550. Springer, New York, USA, 2008.

M. Dror. Note on the complexity of the shortest path models for column generation in VRPTW. *Operational Research*, 42:977–978, 1994.

Gunter Dueck. New optimization heuristics: the great deluge algorithm and the record-to-record travel. *Journal of Computational Physics*, 104:86–92, 1993.

G.F.D. Duff. Differences, derivatives and decreasing rearrangements. *Canad. J. Math*, 19:1153–1178, 1967.

B. Eksioglu, A.V. Vural, and A. Reisman. The vehicle routing problem: A taxonomic review. *Computers & Industrial Engineering*, 57:1472–1483, 2009.

D. Feillet, P. Dejax, M. Gendreau, and C. Gueguen. An exact algorithm for the elementary shortest path problem with resource constraints: application to some vehicle routing problems. *Networks*, 44:216–229, 2004.

D. Feillet, P. Dejax, and M. Gendreau. Traveling salesman problems with profits. *Transportation Science*, 39(2):188–205, 2005.

T.A. Feo and M.G.C. Resende. Greedy randomized adaptive search procedures. *Journal of global optimization*, 6:109–133, 1995.

M. Fischetti, J.J. Salazar, and P. Toth. A branch-and-cut algorithm for the symmetric generalized traveling salesman problem. *Operations Research*, 45:378–394, 1997.

M. Fischetti, J.J. Salazar, and P. Toth. Solving the orienteering problem through branch-and-cut. *INFORMS Journal on Computing*, 10(2):133–148, 1998.

M.L. Fisher and R. Jaikumar. A generalized assignment heuristic for vehicle routing. *Networks*, 11:109–124, 1981.

D.A. Flores-Garza, M.A. Salazar-Aguilar, S.U. Ngueveu, and G. Laporte. The multi-vehicle cumulative covering tour problem. *Annals of Operations Research*, pages 1–20, 2015.

B. Funke, T. Grünert, and S. Irnich. Local search for vehicle routing and scheduling problems: Review and conceptual integration. *Journal of Heuristics*, 11:267–306, 2005.

G. Galindo and R. Batta. Review of recent developments in OR/MS research in disaster operations management. *European Journal of Operational Research*, 230:201–211, 2013.

D. Gavalas, C. Konstantopoulos, K. Mastakas, and G. Pantziou. A survey on algorithmic approaches for solving tourist trip design problems. *Journal of Heuristics*, 20: 291–328, 2014.

M. Gendreau and J.-Y. Potvin. Metaheuristics in combinatorial optimization. *Annals of Operations Research*, 140:189–213, 2005.

M. Gendreau, A. Hertz, and G. Laporte. New insertion and postoptimization procedures for the travelling salesman problem. *Operations Research*, 40:1086–1094, 1992.

M. Gendreau, G. Laporte, and F. Semet. The covering tour problem. *Operations Research*, 45:568–576, 1997.

M. Gendreau, G. Laporte, and F. Semet. A branch-and-cut algorithm for the undirected selective travelling salesman problem. *Networks*, 32:263–273, 1998a.

M. Gendreau, G. Laporte, and F. Semet. A tabu search heuristic for the undirected selective travelling salesman problem. *European Journal of Operational Research*, 106:539–545, 1998b.

B.E. Gillett and L.R. Miller. A heuristic algorithm for the vehicle-dispatch problem. *Operations Research*, 22:340–349, 1974.

B. Golden, L. Levy, and R. Vohra. The orienteering problem. *Naval Research Logistics*, 34:307–318, 1987.

B. Golden, Q. Wang, and L. Liu. A multifaceted heuristic for the orienteering problem. *Naval Research Logistics*, 35:359–366, 1988.

B. Golden, A. Assad, and E. Wasil, editors. *The Vehicle Routing Problem: Latest Advances and New Challenges*. Springer, New York, USA, 2008.

B. Golden, Z. Naji-Azimi, S. Raghavan, M. Salari, and P. Toth. The generalized covering salesman problem. *INFORMS Journal on Computing*, 24:534–553, 2012.

G. Gutin and A.P. Punnen, editors. *The Traveling Salesman Problem and Its Variants*. Kluwer Academic Publishers, Dordrecht, The Netherlands, 2002.

Minh Hoàng Hà, Nathalie Bostel, André Langevin, and Louis-Martin Rousseau. An exact algorithm and a metaheuristic for the multi-vehicle covering tour problem with a constraint on the number of vertices. *European Journal of Operational Research*, 226:211–220, 2013.

M. Hachicha, M.J. Hodgson, G. Laporte, and F. Semet. Heuristics for the multi-vehicle covering tour problem. *Computers & Operations Research*, 27:29–42, 2000.

R. Hall and J. Partyka. Vehicle routing software survey: Higher expectations drive transformation. *OR/MS Today*, 43, 2016.

P. Hansen, N. Mladenović, and J.A. Pérez-Moreno. Variable neighbourhood search: methods and applications. *Annals of Operations Research*, 175:367–407, 2010.

P.E. Hart, N.J. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *Systems Science and Cybernetics, IEEE Transactions on*, 4:100–107, 1968.

V. Hemmelmayr, K.F. Doerner, R.F. Hartl, and M.W.P. Savelsbergh. Delivery strategies for blood products supplies. *OR spectrum*, 31:707–725, 2009.

J. Hodgson, G. Laporte, and F. Semet. A covering tour model for planning mobile health care facilities in Suhum District, Ghana. *Journal of Regional Science*, 38: 621–638, 1998.

T.C. Hu, A.B. Kahng, and C.A. Tsao. Old bachelor acceptance: A new class of non-monotone threshold accepting methods. *ORSA Journal on Computing*, 7:417–425, 1995.

T. Ilhan, S.M.R. Iravani, and M.S. Daskin. The orienteering problem with stochastic profits. *IIE Transactions*, 40:406–421, 2008.

S. Irnich and G. Desaulniers. Shortest path problems with resource constraints. *Les Cahiers du GERAD*, G-2004-11:1–27, 2004.

L. Jourdan, M. Basseur, and E.-G. Talbi. Hybridizing exact methods and metaheuristics: A taxonomy. *European Journal of Operational Research*, 199:620–629, 2009.

N. Jozefowiez. A branch-and-price algorithm for the multivehicle covering tour problem. *Networks*, 64:160–168, 2014.

N. Jozefowiez, F. Semet, and E.G. Talbi. The bi-objective covering tour problem. *Computers & Operations Research*, 34:1929–1942, 2007.

M. Kammoun, H. Derbel, M. Ratli, and B. Jarboui. A variable neighborhood search for solving the multi-vehicle covering tour problem. *Electronic Notes in Discrete Mathematics*, 47:285–292, 2015.

D. Karapetyan and M. Reihaneh. An efficient hybrid ant colony system for the generalized traveling salesman problem. *Algorithmic Operations Research*, 7:21–28, 2012.

S. Kataoka and S. Morito. Single constraint maximum collection problem. *Journal of the Operations Research Society of Japan*, 31:515–530, 1988.

S. Kedad-Sidhoum and V.H. Nguyen. An exact algorithm for solving the ring star problem. *Optimization: A Journal of Mathematical Programming and Operations Research*, pages 125–140, 2010.

C. Keller. Algorithms to solve the orienteering problem: A comparison. *European Journal of Operational Research*, 41:224–231, 1989.

G.A.P. Kindervater and M.W.P. Savelsbergh. *Local Search in Combinatorial Optimization*, chapter Vehicle routing: Handling edge exchanges, pages 337–360. John Wiley & Son, 1997.

S. Kirkpatrick, C.D. Gelatt, and M.P. Vecchi. Optimization by simulated annealing. *Science*, 220:671–680, 1983.

A.A. Kovacs, S.N. Parragh, K.F. Doerner, and R.F. Hartl. Adaptive large neighborhood search for service technician routing and scheduling problems. *Journal of Scheduling*, 15:579–600, 2012.

N. Labadie, C. Prins, and C. Prodhon. *Metaheuristics for Vehicle Routing Problems*. John Wiley & Sons, 2016.

M. Labbé and G. Laporte. Maximizing user convenience and postal service efficiency in post box location. *Belgian J. Opns. Res. Statist. and Computer Sci.*, 26:21–35, 1986.

M. Labbé, G. Laporte, I. Rodríguez Martín, and J.J. Salazar González. The ring star problem: Polyhedral analysis and exact algorithm. *Networks*, 43:177–189, 2004.

M. Labbé, G. Laporte, I. Rodríguez Martín, and J.J. Salazar González. Locating median cycles in networks. *European Journal of Operational Research*, 160:457–470, 2005.

M. Laguna and R. Marti. Grasp and path relinking for 2-layer straight line crossing minimization. *INFORMS Journal on Computing*, 11:44–52, 1999.

G. Laporte. *Vehicle Routing: Methods and Studies*, chapter Location-routing problems, pages 193–207. North-Holland, Amsterdam, 1988.

G. Laporte. Tour location problems. *Studies in Locational Analysis*, 11:13–26, 1997.

G. Laporte and S. Martello. The selective traveling salesman problem. *Discrete Applied Mathematics*, 26:193–207, 1990.

G. Laporte and F. Semet. *The Vehicle Routing Problem*, chapter Classical and new heuristics for the vehicle routing problem, pages 109–128. SIAM, Philadelphia, USA, 2002.

G. Laporte, S. Chapleau, P.-E. Landry, and H. Mercure. An algorithm for the design of mailbox collection routes in urban areas. *Transportation Research Part B: Methodological*, 23:271–280, 1989.

E.L. Lawler, J.K. Lenstra, A.H.G. Rinnooy Kan, and D.B. Shmoys. *The Traveling Salesman Problem: A Guided Tour of Combinatorial Optimization*. John Wiley & Sons, New York, USA, 1985.

K. Li and H. Tian. A two-level self-adaptive variable neighborhood search algorithm for the prize-collecting vehicle routing problem. *Applied Soft Computing Journal*, page http://dx.doi.org/10.1016/j.asoc.2016.02.040, 2016.

S. Lin and B.W. Kernighan. An effective heuristic algorithm for the traveling-salesman problem. *Operations Research*, 21:498–516, 1973.

R. Lopes, V.A. Souza, and A.S. da Cunha. A branch-and-price algorithm for the multi-vehicle covering tour problem. *Electronic Notes in Discrete Mathematics*, 44:61–66, 2013.

M. López-Ibáñez, J. Dubois-Lacoste, T. Stützle, and M. Birattari. The irace package, Iterated Race for Automatic Algorithm Configuration. Technical Report TR/IRIDIA/2011-004, IRIDIA, Université Libre de Bruxelles, Belgium, 2011.

M. Milano and P. Van Hentenryck, editors. *Hybrid Optimization*. Springer, New York, USA, 2011.

N. Mladenović and P. Hansen. Variable neighborhood search. *Computers & Operations Research*, 24:1097–1100, 1997.

J.A. Moreno Pérez, M. Moreno-Vega, and I. Rodríguez Martín. Variable neighborhood tabu search and its application to the median cycle problem. *European Journal of Operational Research*, 151:365–378, 2003.

L. Motta, L.S. Ochi, and C. Martinhon. GRASP metaheuristics to the generalized covering tour problem. In *MIC'2001–4th Metaheuristics International Conference*, pages 387–391, Porto, Portugal, 2001.

K. Murakami. A column generation approach for the multi-vehicle covering tour problem. In *Automation Science and Engineering (CASE), 2014 IEEE International Conference on*, pages 1063–1068, 2014.

Z. Naji-Azimi, J. Renaud, A. Ruiz, and M. Salari. A covering tour approach to the location of satellite distribution centers to supply humanitarian aid. *European Journal of Operational Research*, 222:596–605, 2012.

R. Neapolitan and K. Naimipour. *Foundations of Algorithms: with C++ Pseudocode*. Jones and Bartlett Publishers, Inc., Boston, USA, 1998.

G.L. Nemhauser and L.A. Wolsey. *Integer and combinatorial optimization*. John Wiley & Sons, 1999.

S.U. Ngueveu, C. Prins, and R.W. Calvo. An effective memetic algorithm for the cumulative capacitated vehicle routing problem. *Computers & Operations Research*, 37:1877–1885, 2010.

P.C. Nolz, K.F. Doerner, W.J. Gutjahr, and R.F. Hartl. *Advances in Multi-Objective Nature Inspired Computing*, chapter A bi-objective metaheuristic for disaster relief operation planning, pages 167–187. Springer, 2010.

W.A. Oliveira, M.P. Mello, A.C. Moretti, and E.F. Reis. The multi-vehicle covering tour problem: building routes for urban patrolling. *arXiv:1309.5502*, 2013. 19 pages.

C.H. Papadimitriou and K. Steiglitz. *Combinatorial Optimization–Algorithms and Complexity*. Dover Publications, Inc., New York, USA, 1982.

M.B. Pedersen. *Optimization models and solution methods for intermodal transportation*. PhD thesis, Centre for Traffic and Transport, Technical University of Denmark, 2005.

D. Pisinger and S. Ropke. A general heuristic for vehicle routing problems. *Computers & Operations Research*, 34:2403–2435, 2007.

D. Pisinger and S. Ropke. *Handbook of Metaheuristics*, chapter Large neighborhood search, pages 399–419. Springer, New York, USA, 2010.

I. Pohl. *Machine Intelligence*, volume 6, chapter Bi-directional search, pages 127–140. Edinburgh University Press, 1971.

G. Polya. *How to solve it*. Princeton University Press, Princeton, NJ, 1945.

C. Prins. A simple and effective evolutionary algorithm for the vehicle routing problem. *Computers & Operations Research*, 31:1985–2002, 2004.

C. Prins. *Bio-Inspired Algorithms for the Vehicle Routing Problem*, chapter A GRASP× evolutionary local search hybrid for the vehicle routing problem, pages 35–53. Springer, 2009.

C. Prins, N. Labadi, and M. Reghioui. Tour splitting algorithms for vehicle routing problems. *International Journal of Production Research*, 47:507–535, 2009.

C. Prins, P. Lacomme, and C. Prodhon. Order-first split-second methods for vehicle routing problems: A review. *Transportation Research Part C: Emerging Technologies*, 40:179–200, 2014.

R. Ramesh and K. Brown. An efficient four-phase heuristic for the generalized orienteering problem. *Computers & Operations Research*, 18:151–165, 1991.

R. Ramesh, Y. Yoon, and M. Karwan. An optimal algorithm for the orienteering tour problem. *ORSA Journal on Computing*, 4:155–165, 1992.

G. Reinelt. TSPLIB – A traveling salesman problem library. *ORSA Journal on Computing*, 3:376–384, 1991.

J. Renaud, F.F. Boctor, and G. Laporte. Efficient heuristics for median cycle problems. *Journal of the Operational Research Society*, 55:179–186, 2004.

C.S. ReVelle and G. Laporte. New directions in plant location. *Studies in Locational Analysis*, 5:31–58, 1993.

G. Ribeiro and G. Laporte. An adaptive large neighborhood search heuristic for the cumulative capacitated vehicle routing problem. *Computers & Operations Research*, 39:728–735, 2012.

G. Righini and M. Salani. Summetry helps: Bounded bi-directional dynamic programming for the elementary shortest path problem with resource constraints. *Discrete Optimization*, 3:255–273, 2006.

G. Righini and M. Salani. New dynamic programming algorithms for the resource constrained shortest path problem. *Networks*, pages 155–170, 2008.

S. Ropke and D. Pisinger. An adaptive large neighborhood search heuristic for the pickup and delivery problem with time windows. *Transportation Science*, 40:455–472, 2006.

F. Rossi, P. Van Beek, and T. Walsh. *Handbook of constraint programming*. Elsevier, 2006.

M. Salari and Z. Naji-Azimi. An integer-programming-based local search for the covering salesman problem. *Computers & Operations Research*, 39:2594–2602, 2012.

M. Salari, M. Reihaneh, and M. Sabbagh. Combining ant colony optimization algorithm and dynamic programming technique for solving the covering salesman problem. *Computers & Industrial Engineering*, 83:244–251, 2015.

M. Schilde, K.F. Doerner, R.F. Hartl, and G. Kiechle. Metaheuristics for the bi-objective orienteering problem. *Swarm Intelligence*, 3:179–201, 2009.

G. Schrimpf, J. Schneider, H. Stamm-Wilbrandt, and G. Dueck. Record breaking optimization results using the ruin and recreate principle. *Journal of Computational Physics*, 159:139–171, 2000.

P. Shaw. A new local search algorithm providing high quality solutions to vehicle routing problems. Technical report, University of Strathclyde, Scotland, 1997.

P. Shaw. Using constraint programming and local search methods to solve vehicle routing problems. In *Proceedings of the 4th International Conference on Principles and Practice of Constraint Programming*, pages 417–431, New York, 1998. Springer.

J.C. Simms. Fixed and mobile facilities in dairy practice. *The Veterinary Clinics of North America - Food Animal Practice*, 5:591–601, 1989.

W. Souffriau, P. Vansteenwegen, J. Vertommen, G. Vanden Berghe, and D. Van Oudheusden. A personalized tourist trip design algorithm for mobile tourist guides. *Applied Artificial Intelligence*, 22:964–985, 2008.

A. Subramanian, E. Uchoa, and L.S. Ochi. A hybrid algorithm for a class of vehicle routing problems. *Computers & Operations Research*, 40:2519–2531, 2013.

E.G. Talbi. *Metaheuristics from design to implementation.* Wiley & Sons, USA, 2009.

L. Tang and X. Wang. Iterated local search algorithm based on very large-scale neighborhood for prize-collecting vehicle routing problem. *The International Journal of Advanced Manufacturing Technology*, 29:1246–1258, 2006.

T. Thomadsen and T. Stidsen. The quadratic selective travelling salesman problem. Technical report, Technical University of Denmark, 2003.

P.M. Thompson and H.N. Psaraftis. Cyclic transfer algorithm for multivehicle routing and scheduling problems. *Operations Research*, 41:935–946, 1993.

P. Toth and D. Vigo. *The Vehicle Routing Problem*, chapter An overview of vehicle routing problems, pages 1–26. Society for Industrial and Applied Mathematics, Philadelphia, USA, 2002.

P. Toth and D. Vigo. The granular tabu search and its application to the vehicle-routing problem. *Informs Journal on Computing*, 15:333–346, 2003.

P. Toth and D. Vigo, editors. *Vehicle Routing: Problems, Methods, and Applications.* Society for Industrial and Applied Mathematics, 2014.

F. Tricoire, M. Romauch, K.F. Doerner, and R.F. Hartl. Heuristics for the multi-period orienteering problem with multiple time windows. *Computers & Operations Research*, 37:351–367, 2010.

F. Tricoire, A. Graf, and W.J. Gutjahr. The bi-objective stochastic covering tour problem. *Computers & Operations Research*, 39:1582–1592, 2012.

T. Tsiligirides. Heuristic methods applied to orienteering. *Journal of the Operational Research Society*, 35(9):797–809, 1984.

R. Tunalioglu, C. Koc, and T. Bektas. A multiperiod location-routing problem. *Journal of the Operational Research Society*, 2016. ISSN 0160-5682. URL `http://dx.doi.org/10.1057/jors.2015.121`.

T. Urli. *Hybrid metaheuristics for combinatorial optimization.* PhD thesis, Dipartimento di Ingegneria Elettrica, Gestionale e Meccanica, Università degli Studi di Udine, 2014.

A. Van Breedam. *An analysis of the behavior of heuristics for the vehicle routing problem for a selection of problems with vehicle-related, customer-related, and time-related constraints.* PhD thesis, University of Antwerp, 1994.

P. Vansteenwegen and D. Van Oudheusden. The mobile tourist guide: An OR opportunity. *OR Insight*, 20:21–27, 2007.

P. Vansteenwegen, W. Souffriau, G. Vanden Berghe, and D. Van Oudheusden. A guided local search metaheuristic for the team orienteering problem. *European Journal of Operational Research*, 196:118–127, 2009.

P. Vansteenwegen, W. Souffriau, and D. Van Oudheusden. The orienteering problem: A survey. *European Journal of Operational Research*, 209:1–10, 2011a.

P. Vansteenwegen, W. Souffriau, G. Vanden Berghe, and D. Van Oudheusden. The City Trip Planner: An expert system for tourists. *Expert Systems with Applications*, 38:6540–6546, 2011b.

L. Vargas, N. Jozefowiez, and S.U. Ngueveu. A Selector operator-based adaptive large neighborhood search for the covering tour problem. In *Proceedings of the 9th Learning and Intelligent OptimizatioN Conference, LION9*, pages 170–185, New York, 2015a. Springer.

L. Vargas, N. Jozefowiez, and S.U. Ngueveu. A dynamic programming operator for tour location problems applied to the covering tour problem. *submitted to Journal of Heuristics*, 2015b.

L. Vargas, N. Jozefowiez, and S.U. Ngueveu. Solving the orienteering problem using a dynamic programming operator for tour location problems. Technical report, LAAS-CNRS, Toulouse, 2015c.

L. Vargas, N. Jozefowiez, and S.U. Ngueveu. A dynamic programming-based metaheuristic for the m-covering tour problem. *submitted to Computers & Operations Research*, 2016.

V.V. Vazirani. *Approximation Algorithms*. Springer, New York, USA, 2003.

T. Vidal, T.G. Crainic, M. Gendreau, and C. Prins. A unified solution framework for multi-attribute vehicle routing problems. *European Journal of Operational Research*, 234:658–673, 2014.

T. Vidal, N. Maculan, L.S. Ochi, and P.H. Vaz Penna. Large neighborhoods with implicit customer selection for vehicle routing problems with profits. *Transportation Science, Articles in Advance*, pages 1–15, 2015.

L. Vogt, C.A. Poojari, and J.E. Beasley. A tabu search algorithm for the single vehicle routing allocation problem. *Journal of the Operational Research Society*, 58:467–480, 2007.

C. Voudouris and E.P.K. Tsang. *Handbook of metaheuristics*, volume 57, chapter Guided local search, pages 185–218. Springer Science & Business Media, 2003.

S. Voß. *Local Search for Planning and Scheduling, LNCS 2148*, chapter Meta-heuristics: The state of the art, pages 1–23. Springer-Verlag, Berlin Heidelberg, 2001.

Q. Wang, X. Sun, B. Golden, and J. Jia. Using artificial neural networks to solve the orienteering problem. *Annals of Operations Research*, 61:111–120, 1995.

X. Wang, B. Golden, and E. Wasil. *The vehicle routing problem: latest advances and new challenges*, chapter Using a genetic algorithm to solve the generalized orienteering problem, pages 263–274. 2008.

A. Wren and A. Holliday. Computer scheduling of vehicles from one or more depots to a number of delivery points. *Operational Research Quarterly*, 23:333–344, 1972.

T. Zhang, W.A. Chaovalitwongse, Y.-J. Zhang, and P.M. Pardalos. The hot-rolling batch scheduling method based on the prize collecting vehicle routing problem. *Journal of Industrial and Management Optimization*, 5:749–765, 2009.

W. Zhang. *State-space search: Algorithms, complexity, extensions, and applications*. Springer, New York, USA, 1999.