UCL Université catholique de Louvain

École polytechnique de Louvain (EPL)

epl

Local Search for the Vehicle Routing Problem

Dissertation presented by
Laurent SMET , Charles THOMAS

for obtaining the Master's degree in
Computer Science

Supervisor(s)
Yves DEVILLE

Reader(s)
Pierre SCHAUS, Michael SAINT-GUILLAIN

Academic year 2015-2016

Acknowledgements

First, we would like to sincerely thank our supervisor Yves Deville for his guidance, his valuable remarks and comments and his support throughout this year.

We express our gratitude to Sascha Van Cauwelaert for his useful explanations on the performance profiles and letting us use his performance profiles builder, Cyrille Djemeppe for his valuable advice concerning our experimental plan and Pierre Schaus for his interesting remarks.

We also thank Quentin Cappart and Noémie Thomas that took time to read this dissertation and make it better as well as all the others that contributed to improve this document through discussions and comments.

Finally, we thank our families and relatives for helping us indirectly during this whole year with their encouragements and support, especially during the hard times.

Charles and Laurent

Contents

Chapter 1

Introduction

The Vehicle Routing Problem (VRP) is one of the most studied optimisation problems. It was defined in 1959 by George Dantzig and John Ramser [1] and consists in optimizing the delivery of goods to customers. This problem holds a central place in the fields of distribution management, logistics and transportation. The interest of the VRP is motivated by its economical importance as well as its challenging difficulty.

Since its definition, the VRP has led to many variants with additional constraints or refinements over the original problem. To deal with these, many different approaches have been developed [2–6]. However, despite the high number of algorithms that have been proposed to solve it, the VRP stays a real challenge to tackle. This is particularly the case when the size of the problem grows due to its NP-hard nature [7].

One of the approaches that can be used to solve the VRP is the local search [8]. The local search is an incomplete disruptive method that consists in exploring the search space locally by starting from an initial solution and iteratively improving it. To do so, the algorithm moves from solution to solution by applying local changes. The advantage of this technique is that a reasonable solution can be found in an enormous search space while using a small amount of resources. However, the solution found is not guaranteed to be optimal.

The aim of this thesis was, first, to study the vehicle routing problem and analyse the existing resolution methods in local search. Then, to extend a specific VRP module in the Open LS system to tackle new variants with efficiency and flexibility. Open LS is a new open source local search library written in Java and developed at the Hanoi University of Science and Technology [9].

This thesis is organised as follows:

Chapter 2 gives a formal definition of the VRP and present some of it's variants.

Chapter 3 addresses the local search approach, it's parameters and some existing techniques.

Chapter 4 extends the previous chapter and presents the state-of-the-art local search methods to solve the VRP.

Chapter 5 describes the VRPLS module, its design and its implementation.

Chapter 6 presents the experiments that we carried, their results and our analysis of these results.

Chapter 7 concludes this master thesis by highlighting some keys points of the VRPLS module and suggests some extensions and improvements.

Chapter 2

The vehicle routing problem

The Vehicle Routing Problem (VRP) is a generic name for a whole family of problems that extends the well known Travelling Salesman Problem (TSP) [10]. Its computational complexity is thus NP-hard [7, 11]. The general principle can be defined as follows: given a set of customers with known demands and one or more depots, the aim of the problem is to determine routes starting from and ending to a depot so that customers are served while minimising the total route cost and satisfying various constraints.

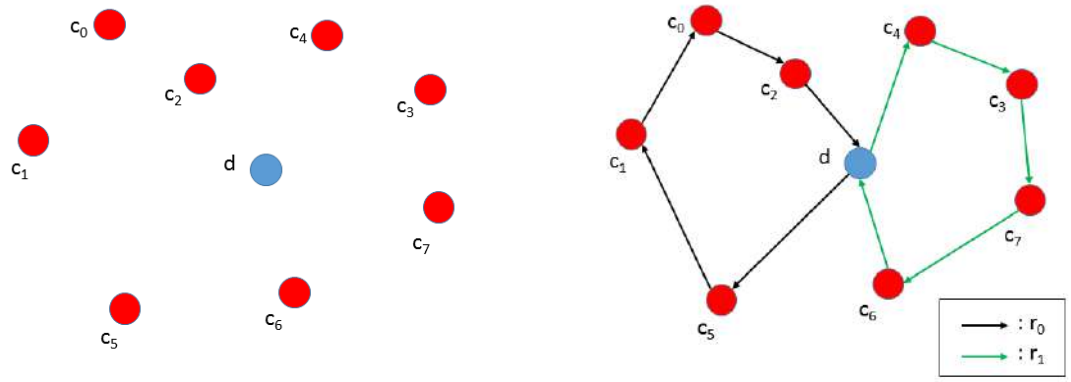

Figure 2.1: A VRP instance (left) and a possible solution (right).

The cost of a route can be based on the length, duration or other attributes such as the fuel consumption. Note that the number of routes used may have to be minimised as part of the problem. In other variants, this number is fixed or has a maximum value. In any case, a solution with a smaller number of routes will most likely be better in terms of cost as travels from and to the depot(s) are avoided.

Different constraints and variations can be added, which leads to many variants of the problem [12]. In this thesis, we tackled six of these variants: The Capacitated VRP (CVRP), the Distance constrained VRP (DVRP), the VRP with Time Windows (VRPTW), the Multiple Depots VRP (MDVRP), the VRP with Pick-ups and Deliveries (VRPPD) and the Heterogeneous vehicles VRP (HVRP). These variants are described in the following sections.

2.1 Capacitated VRP

The Capacitated VRP [13] is the basic version of this problem in which we have a single delivery vehicle profile with a fixed capacity for a single commodity. Each route must start and finish at a common depot. In this variant, we must deliver every customer with a minimum cost which corresponds to the total distance travelled by the vehicles.

The Capacitated VRP has only one possible depot d, a set of customers $C =< c_0, c_1, ..., c_{|C|-1} >$ with known demands $demand(c_i)$ and a same capacity κ for all the vehicles (maximum load per route). The cost (distance) between two nodes (a customer or a depot) is expressed by a cost function $dist(i, j)$.

Let $R =< r_0, r_1, ..., r_{|R|-1} >$ be the set of routes. Each route r_j is composed by an ordered sequence of customers such as $r_j =< rc_{j,0}, rc_{j,1}, ..., rc_{j,|r_j|-1} >$. Each customer $rc_{j,k}$ of the route corresponds to a customer c_i of the problem: $\forall rc_{j,k} \in r_j, \exists c_i \in C : c_i = rc_{j,k}$.

The problem consists in finding an assignment of routes so that the total cost of the routes is minimised (2.1), each customer c_i is delivered exactly once (2.2 & 2.3) and the total demand of each route is smaller than the capacity κ (2.4).

$$\min \sum_{r_j \in R} dist(d, rc_{j,0}) + \left(\sum_{k=1}^{|r_j|-1} dist(rc_{j,k-1}, rc_{j,k}) \right) + dist(rc_{j,|r_j|-1}, d) \tag{2.1}$$

$$\forall c_i \in C, \exists r_j \in R : \exists rc_{j,k} \in r_j : rc_{j,k} = c_i \tag{2.2}$$

$$\sum_{r_j \in R} |r_j| = |C| \tag{2.3}$$

$$\forall r_j \in R, \left(\sum_{k=0}^{|r_j|-1} demand(rc_{j,k}) \right) \leq \kappa \tag{2.4}$$

2.2 Distance constrained VRP

This variant [14] extends the capacitated VRP with an additional constraint: the total length of each route r_j cannot be higher than a given bound $maxLength$ (2.5).

$$\forall r_j \in R : \left(dist(d, rc_{j,0}) + \left(\sum_{k=1}^{|r_j|-1} dist(rc_{j,k-1}, rc_{j,k}) \right) + dist(rc_{j,|r_j|-1}, d) \right) \leq maxLength \tag{2.5}$$

This constraint can be used to represent situations where the vehicle used has a limited autonomy or where the driver can work for a limited time.

Additionally, a service time can be added when delivering a customer: $stime(rc_k)$. In this case, this might impact the cost function, distance constraint, or both. For example, we could imagine that the delivery vehicle has to be back to the depot by a given time but that the cost is calculated based on the distance travelled. In this case, the service time impacts the distance constraint (2.6) but not the cost calculation.

$$\forall r_j \in R : \left(dist(d, rc_{j,0}) + \left(\sum_{k=1}^{|r_j|-1} dist(rc_{j,k-1}, rc_{j,k}) + stime(rc_{j,k}) \right) + dist(rc_{j,|r_j|-1}, d) \right) \leq maxLength \tag{2.6}$$

2.3 Time windows

In this variant [15], each customer has an associated time window during which the delivery has to take place. The delivery vehicle might arrive before the beginning of the customer's time window but then, has to wait until the customer is available to be served. In some variants, this *waiting time* (2.10) can be taken into account in the cost calculation.

If the arrival time at a customer is after the end of the customer's time window, the constraint is considered as violated. Again, a service time $stime(rc_{j,k})$ can be associated to each delivery. In this case, two variants are possible: either the delivery has to end before the end of the customer's time window or the constraint is considered as respected as long as the arrival time is in the time window. In this last variant, note that the delivery could end after the time window due to the service time.

Let $stw(rc_{j,k})$ be the starting time window and $etw(rc_{j,k})$ the ending time window of the customer c_i corresponding to $rc_{j,k}$. We define the *arrival time* as the time at which the delivery vehicle arrives at a customer (2.8). The *end delivery time* is the time at which the delivery is finished and the vehicle leaves for the next customer (2.9). The constraint is expressed in equation 2.7. If there is no service time, it is replaced by 0 in equation 2.9.

$$\forall r_j \in R, \forall rc_{j,k} \in r_j, arrivalTime(rc_{j,k}) \leq etw(rc_{j,k}) \tag{2.7}$$

$$arrivalTime(rc_{j,k}) = \begin{cases} dist(d, rc_{j,0}) & \text{if } k = 0 \\ endDeliveryTime(rc_{j,k-1}) + dist(rc_{j,k-1}, rc_{j,k}) & \text{otherwise} \end{cases} \tag{2.8}$$

$$endDeliveryTime(rc_{j,k}) = max(arrivalTime(rc_{j,k}), stw(rc_{j,k})) + stime(rc_{j,k}) \tag{2.9}$$

$$waitingTime(rc_{j,k}) = \sum_{k=0}^{|r_j|-1} max(0, (stw(rc_{j,k}) - arrivalTime(rc_{j,k}))) \tag{2.10}$$

2.4 Multiple Depots

The multiple depot VRP [16] adds the possibility to have more than one depot. Two sub variants can be defined:

The first one allows a route to end at a different depot than the departing one. For example, in Figure 2.2, a solution has been found where the route r_1 (in green) starts from the depot d_1 and ends at the depot d_0.

In the other variant, each route has to start from and end at the same depot. A possible way to ease the computation is, in the first place, to associate each customer to a depot. Then, the VRP for each depot is solved independently as the vehicles of a given depot only serve the customers associated to this depot.

It is important to note that if the customers are not intermingled but rather clustered around depots, it might be better to split the problem into a series of independent VRP problems.

2.5 Pick-ups and Deliveries

For the VRP with Pick-ups and Deliveries [17], some customers can return commodities instead of being delivered. In this case, the capacity constraint must not be checked against the total demand of a route but at each stop. Indeed, we need to ensure that it is not violated due to a pick-up as it could increase the load of the vehicle above it's capacity.

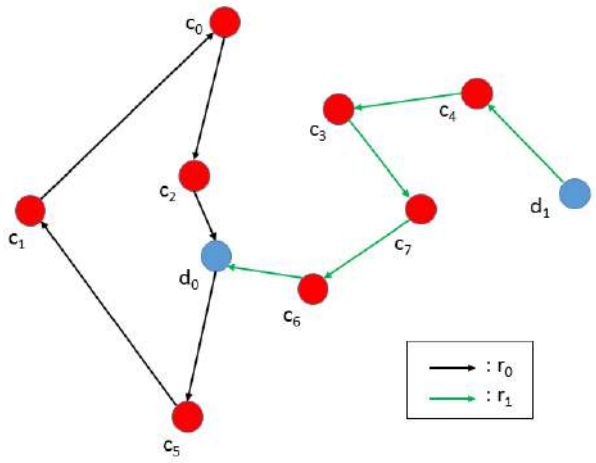

Figure 2.2: A MDVRP instance with 2 depots (blue) and 8 customers (red).

To ease the resolution, some restrictions can be added, such as not mixing deliveries and pick-ups by putting them on separated routes, or having all the deliveries of a route done before any pick-up.

2.6 Heterogeneous Vehicles

This variant [18] allows a company to have multiple delivery vehicles profiles, each one with various features (autonomy, capacity, consumption...). In that case, the choice of the vehicle has a direct impact on the cost calculation.

2.7 Other Variants

In practice, these variants are not exclusive and can be mixed with each other.

The vehicle routing problem contains many other variants such as the stochastic VRP [19], where some of the components of the problem are random, the multiple commodity VRP, in which more than one commodity is considered or the Periodic VRP [20], where the planning period is extended to several days.

These other variants are not explored in this thesis and thus are not further discussed.

Chapter 3

Local Search

Local search (LS) is a well-known technique to solve combinatorial optimisation problems. The main idea is, rather than exploring systematically the search space, to iteratively move from a solution to a neighbouring solution by applying local perturbations until an optimum has been reached.

At each step of the search, the algorithm considers a series of perturbations called *moves*. Each move consists in changing a small part of the current solution to obtain an alternative solution that is close to the previous one. The next solution is selected among this set of alternative solutions, which is called the *neighbourhood*. Typically, the neighbourhood is generated through *operators*, each one defining a type of perturbation to apply. The operators for the vehicle routing problem are explained in chapter 4.

Two other important notions in local search are the *intensification* and the *diversification*. The intensification is relative to the improvement of the current solution towards a local optimum whereas the diversification aims at the exploration of diverse parts of the search space to avoid being trapped in a local optimum.

Typically, these two aspects are implemented through a *heuristic* and a *meta-heuristic*. The heuristic deals with intensification by selecting the next improving move whereas the meta-heuristic introduces diversification by implementing strategies to explore varied parts of the search space and get out of local optima.

The local search approach sacrifices solution quality and optimality guarantee but greatly increases performance. In many cases, the local search gives a near-optimal or even an optimal solution within a small amount of time. It is particularly well suited for big instances where the search space is too large to be fully explored in a reasonable time. Another advantage is its low memory consumption as the neighbourhood considered at each step is relatively small in comparison to the whole search space.

3.1 Neighbourhood

The success of the local search approach greatly depends on the neighbourhood generated and on the efficiency to explore it.

If we consider the search space (the set of all possible solutions) as an undirected graph, each node is a possible solution and any move from one solution to another is represented as an edge. So, the neighbourhood of a solution can be seen as the set of nodes that are connected to the solution. A few properties [21] of the neighbourhood are worth discussing:

Let s be the current solution and $N(s)$ its neighbourhood.

Neighbourhood size The size of N(s) is a critical aspect of local search. Indeed, a larger neighbourhood takes more time to explore but, on the other hand, has more chances to contain a better solution. The size directly depends on the perturbations used to generate the neighbourhood.

Neighbourhood connectivity N(s) is said *weakly connected* if there exists a path in the search space graph from any solution s to an optimal solution s^*. In other words, among the solutions in the neighbourhood, at least one is part of a path that eventually leads to an optimal solution. This property ensures the reachability of optimal solutions. If such a path exists between every pair of solution s_i, s_j, we say that the neighbourhood is *optimally connected*.

In practice, considering only the connectivity of the neighbourhood is not enough to guarantee that the optimal solution will be found during the search. Indeed, the solution found depends on other aspects such as the heuristic or the search technique used. However, this is an important property to know when considering the exploration of the search space. If the neighbourhood is not weakly connected, a restart or jump strategy has to be implemented.

3.2 Solution Evaluation

In the most basic case, the evaluation of a solution consists in the value of the objective function. However, another factor to take into account is the respect of the constraints. Two approaches can be used to deal with constraints in the local search:

The first one is to enforce the respect of the constraints when generating the neighbourhood by ensuring that the moves generated by the operators do not violate any constraint. This approach guarantees that any solution explored during the search is feasible. However, it might lack flexibility and could break the connectivity of the neighbourhood. It can also be difficult to enforce at the level of the operators that no violating move is generated.

The other approach consists in evaluating the violation of a potential solution when exploring the neighbourhood and using it in the selection of the next solution. The problem thus becomes a multi-objective optimisation problem. The violation can either be introduced in the objective function through the means of a weighted sum or be considered as more important than the cost minimisation and used in a lexicographic optimisation.

The violation of a solution is computed by a constraint function. It generally consists in a sum of the violation of each constraint which is dependant on how much the constraint is violated. For example, for the case of the CVRP problem (see Section 2.1), the violation of the capacity constraint is be the sum of the excess demands for each route.

Evaluating the potential solutions in the neighbourhood is a key part of the search. The efficiency of the evaluation process has an important impact on the performances of the solver. Indeed, at each iteration, we potentially have to evaluate a large number of possible solutions in the neighbourhood in order to select the next move to apply. It is thus important to use an efficient evaluation strategy.

One way to do that is to evaluate incrementally the cost and violation of the solutions. This idea comes from the observation that a solution of the neighbourhood has features in common with the actual solution. It is therefore more efficient to maintain some information about the current solution and use it in the evaluation process rather than entirely recomputing it. This is done by using *Invariants* and *Differentiable Objects*.

3.2.1 Invariants

The *invariants* are functions that precise the values to maintain during the search. They define the way that we compute the cost and the violation of a solution. More precisely, they specify the computation procedure but do not manage the incremental changes which is the role of the differentiable objects. This allows the invariants to be simpler as they do not need to care about the evolution of the solution.

3.2.2 Differentiable Objects

The *differentiable objects* maintain the incremental values used and specified in the invariants. When a move is applied, the differentiable object updates its state to reflect the actual state of the solution. This allows us to avoid a lot of redundant computations on the whole solution and can drastically improve the performances of the evaluation.

3.3 Heuristics

The *heuristic* [22] chooses the next neighbour. It drives the search towards a local optimum. Several different heuristics can be used in a local search, each with their own advantages in terms of intensification or diversification. In this section, we discuss about the most common ones.

3.3.1 Best improvement

This heuristic (also called Hill-climbing in [22]) selects the neighbour that improves the most the current solution. In others words, we want to select the neighbour that minimises the most the evaluation function *f(x)*. This is expressed in Equation 3.1 where $H(N(s), s)$ is the heuristic function applied on the neighbourhood $N(s)$ and the current state s.

$$H(N(s), s) := \{n \in N(s) \mid f(n) = min_{k \in N(s)}(f(k))\} \tag{3.1}$$

3.3.2 Random improvement

The random improvement heuristic selects randomly a neighbour among the ones that are better than the current solution. This means that the best move might not be selected. This heuristic is thus worse in terms of intensification but better in terms of diversification.

$$H(N(s), s) := random\{n \in N(s) \,|\, f(n) < f(s)\} \tag{3.2}$$

3.3.3 First improvement

This heuristic is a variation of the previous one (3.3.2). In this case, we select the first neighbour that improves the current state without pursuing the exploration of the neighbourhood. Depending on the implementation, this approach can lead to a huge performance boost.

$$H(N(s), s) := \{n_0 \in N(s) \,|\, f(n) < f(s)\} \tag{3.3}$$

3.3.4 Random walk

The random walk heuristic is another variation of the random improvement: Here, we randomly select a neighbour among the whole neighbourhood. If this neighbour does not improve the current solution, we stay on the current state. This heuristic is best used in combination with a meta-heuristic that keeps track of the stagnation (to stay on the same solution for some time) and might restart the search if necessary.

$$\begin{aligned} &\text{select } n \in N(s) \text{ with probability } \frac{1}{|N(s)|} \\ &H(N(s), s) := \begin{cases} n & \text{if } f(n) < f(s) \\ s, & \text{otherwise} \end{cases} \end{aligned} \tag{3.4}$$

The main difference between this heuristic and random improvement is that with random walk we can stay on the current solution whereas, with random improvement, we always select a better solution (if there is one).

3.3.5 Metropolis

The metropolis heuristic is an extension of the random walk heuristic. We introduce a probability to select a degrading move to improve the diversification. It is used as part of the simulated annealing meta-heuristic (3.4.2). The parameter t determines the probability to select a degrading move and corresponds to the current temperature which varies during the simulated annealing search.

$$\begin{aligned} &\text{select } n \in N(s) \text{ with probability } \frac{1}{|N(s)|} \\ &H(N(s), s) := \begin{cases} n & \text{if } f(n) \leq f(s) \\ n \text{ with probability } \mathrm{e}^{\frac{f(s)-f(n)}{t}} & \text{if } f(n) > f(s) \\ s & \text{otherwise} \end{cases} \end{aligned} \tag{3.5}$$

3.4 Meta-heuristics

A meta-heuristic drives the search towards a global optimum. Rather than selecting the next move inside the neighbourhood based on local information as the heuristics, the meta-heuristics introduces diversity in the search. In opposition to the heuristic which can be described as a clear part of the local search, the meta-heuristic can take many forms. For example, it can be implemented as some kind of meta-search including the local search from multiple starting points inside the search space or change some parameters as the definition of the neighbourhood, the objective function or some others criteria.

3.4.1 Restarts

One way to implement a meta-heuristic is to introduce diversity by restarting the search several times. This restart can either be done from the same solution or from different solutions.

In this last case we call that an intensified search. To bring diversity, the search has to implement some kind of randomness or has to be combined with a tabu strategy (see 3.4.3).

In the other case, having multiple starting solutions allows the search to explore multiple parts of the search space. These solutions can be generated randomly, based on the current solution or based on the best solution found so far with some degrees of relaxation.

3.4.2 Simulated annealing

The simulated annealing meta-heuristic [23] is used in pair with the metropolis heuristic. As described in 3.3.5, this heuristic introduces a new parameter called the temperature. The idea behind the simulated annealing is to update this parameter at each iteration to slowly decrease the diversification as the search progresses.

The temperature parameter t fixes the trade-off between the intensification and the diversification. The search starts with a high temperature and thus a good chance to accept a degrading move. As the search continues, the temperature is slowly decreased in order to focus on intensification and eventually ends the search with a good solution.

An important part of this meta-heuristic is the temperature function that makes the parameter t evolve during the search. A good temperature function is the key to a successful simulated annealing search.

3.4.3 Tabu

The key idea behind the tabu search [24] is to keep in memory the solutions that have already been visited in order to avoid the exploration of the same part of the search space several times. Despite its greater memory requirement, this technique can drastically improve the efficiency of a search.

The main difficulty is to manage the storage and lookup of the visited solutions. The structure that stores the representations of explored states is called the tabu store. As it would be very costly to store the states in full, some way has to be find to represent the path explored. Generally, we use distinctive features of either the solutions or the moves.

Another aspect to deal with is that, as the available memory is finite, it is highly probable that at some point the tabu store will be full. In this case some strategy has to be implemented in order to clear a part of the store . The most common strategy is to remove the solutions that have been in the tabu store the longest.

3.5 Other techniques

Other local search techniques exist but are not in the scope of this thesis or are not adapted to the VRP. Among these techniques we can find the genetic algorithms [25] and the ant colony strategy [26].

The genetic algorithm mimics the natural selection process. It works by creating a pool of possible solutions then using parts of the best ones to create new solutions while introducing some randomness. The main advantage of this technique is that one does not need to understand how to solve the problem to implement a genetic search. However, this approach adds a layer of complexity to the problem as the whole replication and selection process has to be managed.

The ant colony technique, as suggested by its name, has been inspired by the behaviour of ants. It has originally been developed to solve graph based problems such as the TSP. The principle is to have multiple agents exploring pseudo-randomly the edges of the graph and marking them according to their quality. The agents are driven towards marked paths and thus, as the search progresses, the better solutions are slowly chosen over the others.

Finally, many VRP specific local search components have been developed. We present them in the next chapter.

Chapter 4

Local search for the VRP

In the previous chapter, we discuss the general principles of the local search and the techniques not specific to the vehicle routing problem. In this chapter, we focus on the different aspects of the local search that have to be adapted to the VRP and on the specific methods for the VRP.

We first present how the initial solution can be generated. Then, we explain how the neighbourhood is generated through the use of operators. Finally, we discuss some search methods specific to the vehicle routing problem.

4.1 Initial solution

As the local search is a perturbative method, it needs to start from an initial solution. This solution can be generated using different approaches:

Random generation: This approach consists in generating the initial solution randomly. The number of routes can be either chosen randomly or based on the total demand of the customers and the capacity constraints of the vehicles. The customers are assigned at random positions in random routes. While this solution most likely violate some of the constraints and can be far from the optimum, it is easily computable and presents the interest to offer a good diversification when coupled with a restart strategy.

Construction method: Another approach is to use a construction method such as the ones presented in Section 4.3.2. These methods can introduce random decisions to offer a better diversification. This approach has the advantage to provide a better initial solution than the random generation. However, it has a higher computational cost.

Mixed techniques: The idea here is, rather than using only local search, to use it to improve a solution found by another technique. Typically, a relaxed exact technique is used to find a first solution in an acceptable time and then this solution is improved through local search.

An important factor to take into account is whether the initial solution is feasible or violates some constraints. In the second case, this has to be taken into account during the search by selecting moves that lower the violation before minimising the cost to obtain a feasible solution.

Finally, another objective can be added in some variants of the VRP: depending on the specifications of the problem, the number of routes can either be constrained with a maximum or represented as an additional value to minimise. The difficulty here is to weight this new constraint or this new cost accordingly to the cost or constraint function. For this reason, some methods consider the number of vehicles as an additional objective which can then be used in a lexicographic optimisation. For example, the iterated minimum routes method described in Section 4.3.1 tries to minimise the number of routes before the cost.

4.2 Neighbourhood generation

As explained in Chapter 3, at each iteration, the neighbourhood is generated by the operators. Each operator generates a type of move to apply to the current state to obtain a neighbour.

The success of local search is highly dependant on the neighbourhood and thus on the operators used. Ideally, more than one operator is used to generate the neighbourhood and offer different kind of moves thus improving the connectivity and the diversification of the neighbourhood. However, having more operators leads to a higher use of resources to both generate the neighbourhood and explore it. It is therefore critical to choose which operators are used inside the search with great care.

In this section we present some of the most common operators [3, 4, 27] used in local search approaches for the VRP. Note that despite being presented in terms of vehicle routing problem, some of these operators are not specific to the VRP and can be used for other problems such as the Travelling Salesman Problem.

4.2.1 Relocate operator

This operator consists in relocating one customer from one route to another. Formally, the customer i from route r_0 is removed from r_0 and inserted in route r_1 at a position p where $r_0 \neq r_1$ and $0 \leq p \leq |r_1|$.

For example, in Figure 4.1, the customer c_4 is relocated from route r_1 to route r_0.

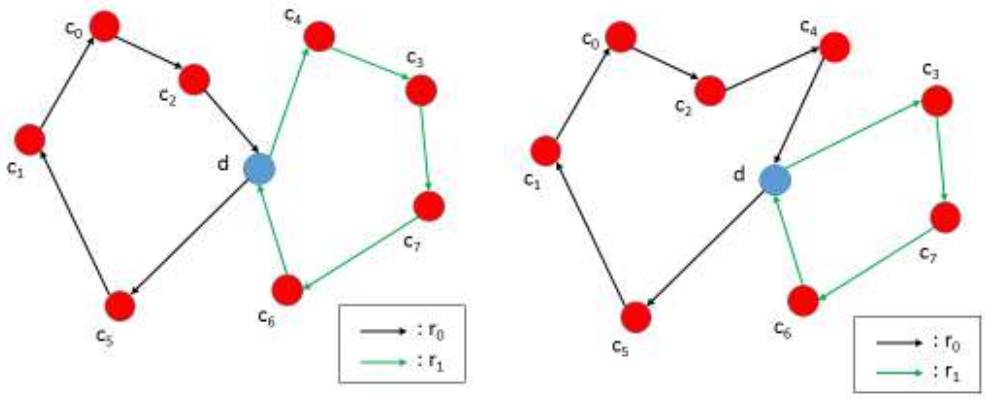

Figure 4.1: Example of relocate operator

4.2.2 Swap operator

This operator consists in swapping two customers from different routes. It can be seen as a double relocation in which the customers are inserted at their counterpart's position in the route.

More formally, the customer c_i at the position p_i in route r_0 is relocated at position p_j in route r_1 while the customer c_j at the position p_j in route r_1 is relocated at position p_i in route r_0.

In Figure 4.2, the customer c_6 from the route r_1 and the customer c_5 from the route r_0 are swapped according to their respective positions.

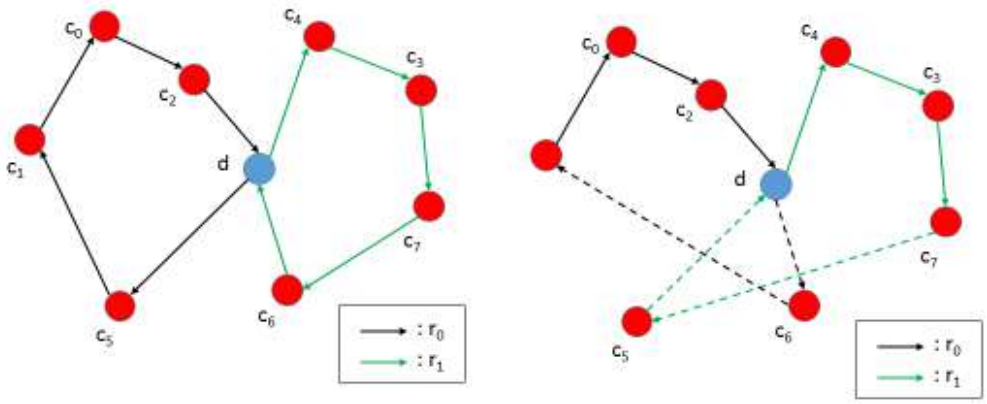

Figure 4.2: Example of swap operator

4.2.3 K-Exchange operator

This operator, which is also called k-opt, consists in dropping k edges in the same route and then reconnecting the resulting segments by other edges. As a result, some segments might be reversed. Formally, it consists of removing k edges in which the endpoints are at positions $< p_0, p_1 ... p_k >$ in the route. The goal is to reconnect the segments in a different order.

This operator might be defined for any number of edges starting from two. However, computing all the possible exchanges would be highly demanding in terms of resources. The operator is thus usually restricted to the 2-exchange or 3-exchange.

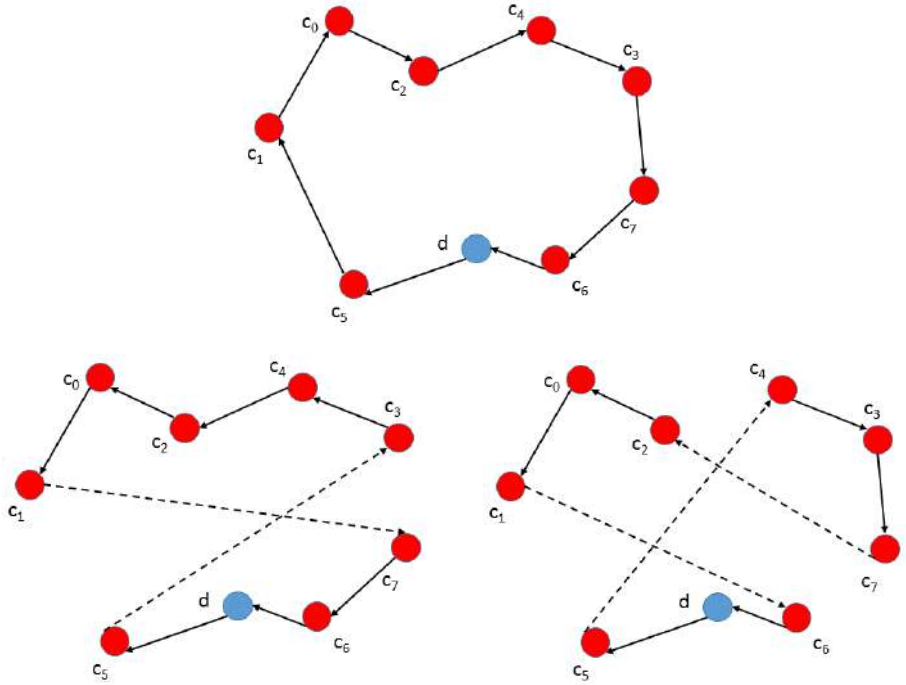

Figure 4.3: Examples of the 2-exchange (left) and the 3-exchange (right) operators

Examples of the 2-exchange and the 3-exchange operators are depicted in Figure 4.3. In the first case, two edges are removed with c_1 and c_7 as endpoints. The resulting route is a new route containing the segment starting at c_1 and ending at c_3 in its reversed form.

In the second case, these edges have c_1, c_4 and c_6 as endpoints. We obtain thus two segments starting at c_1 and c_4 and ending at c_2 and c_7 respectively. The second one is followed first in the same order. Then, the first one is followed in the reverse order.

4.2.4 Cross operator

This operator cuts two different routes in two parts and recompose them with crossing edges. In formal terms, it exchanges the segments from route r_1 and r_2 starting at customer c_i and c_j respectively and ending at the depot where $r_1 \neq r_2$.

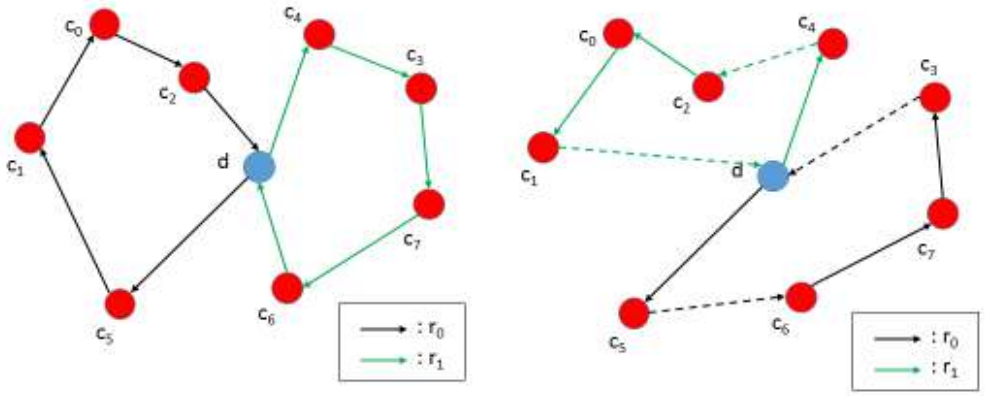

Figure 4.4: Example of cross operator

4.3 VRP search methods

Since its proposition in 1959, many specific search methods have been developed for the vehicle routing problem. In this section we review a few of them. First, we present the iterated minimum routes strategy. Then, we review a few construction methods. While these are not local search methods, they can be used to generate the initial solution and some of them can be adapted into operators such as the Clarke and Wright method (see Section 4.3.2).

4.3.1 Iterated minimum routes

A difficulty specific to the vehicle routing problem is to minimise at the same time the number of routes, the violation and the total cost of the routes. To avoid the usage of a complex weighted sum technique, one approach is to use a lexicographic optimisation by minimising the number of routes in the first place. This technique that uses this approach is called *iterated minimum routes*.

The idea behind the iterated minimum routes is to start the search with the minimum number of routes required to deliver all the customers. In the basic variant, this number can be easily computed as the total demand D divided by the capacity of the vehicles κ.

An initial solution with this number of routes is then generated and a local search is performed until a local optimum is reached or until a given limit of iterations or time. If the solution found is feasible, the process is repeated with the same number of routes. If not, the number of routes is incremented by 1 and a new search is performed. This whole meta-search is done until a fixed limit in terms of iterations, time or objective is reached.

4.3.2 Construction methods

Construction methods consist in building a solution by adding new elements iteratively to an initial empty solution. In the case of the VRP, it typically consists in starting from a solution with empty routes and adding customers to routes one by one. This kind of method is also incomplete and aims to build directly a good solution rather than looking for an optimal solution. It is thus generally less accurate at the benefice of a small computation time.

This kind of method is therefore well suited to create an initial solution for a local search. Note that the initial solution obtained might violate some constraints or have a higher number of routes than the optimal solution. Some of these methods can also be adapted as operators to be used among other classical operators in a local search.

In this section, we see two different construction methods for the vehicle routing problem.

Clarke and Wright

The *Clarke and Wright* construction method [28] consists in starting from a solution with as many routes as there are customers and one customer per route. Then, at each iteration, two routes are merged according to a savings heuristic. This savings heuristic chooses the two routes to merge that induce the most gain in terms of cost. For example in Figure 4.5 the two routes are merged into one, which induces a gain in terms of distance. This process is repeated until no routes can be merged without violation.

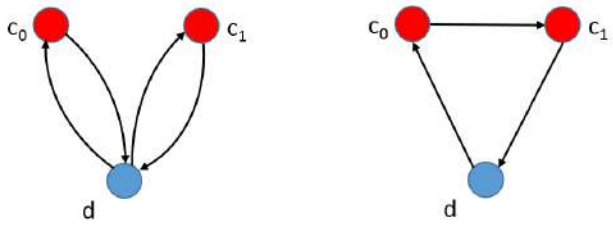

Figure 4.5: The Clarke and Wright method

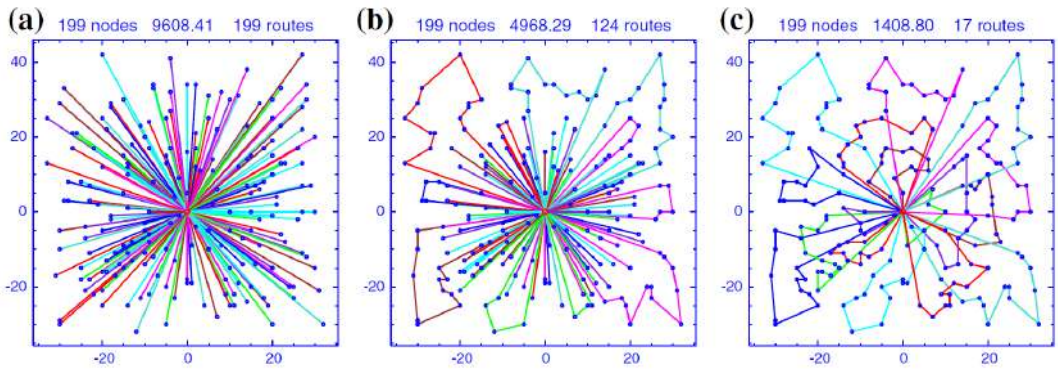

Figure 4.6: Progress of the Clarke and Wright method. **(a)** initial solution; **(b)** after 100 merges; **(c)** final solution.

This method can be adapted into a local search operator that merges two routes to form a single route. This operator can be used in a hybrid local search that starts from a solution with one route per customer and gradually merges these routes while performing other refinements using classical operators.

Note that additional constraints might impact the performances of this kind of method depending on the merge process used. For example, let us consider the case of the VRPTW (see

Section 2.3). In this case, merging two routes by appending them one after the other could induce some violation as the time windows of all the customers have to be respected. So, the merge operation would also need to reorder the customers inside the new route.

Sweep Line

The *sweep line* construction method's principle is to rotate a ray centred at the depot and gradually include customers in a route until the capacity or route length constraint is attained. When one of these conditions is raised, a new route is created and the process is continued until each customer is part of a route. Then, each route is solved independently as a Travelling Salesman Problem.

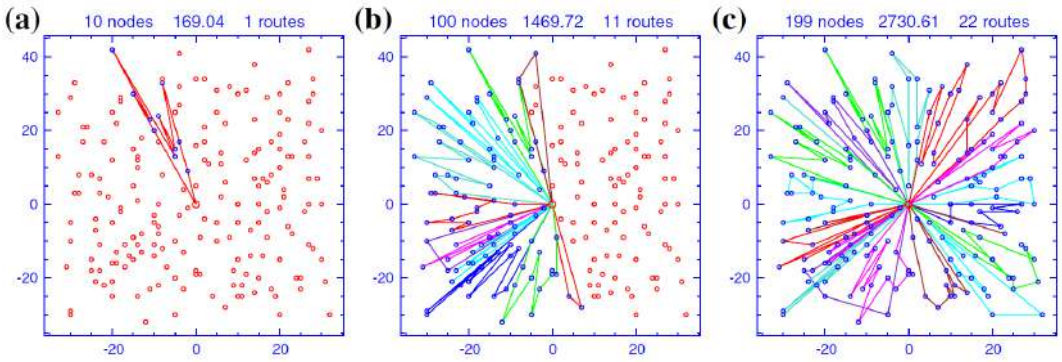

Figure 4.7: Progress of the sweep method. **(a)** after 10 nodes added; **(b)** after 100 nodes added; **(c)** final solution.

This method is efficient for the basic CVRP variants but could not work on other variants with additional constraints. Indeed, for the VRPTW, we have the same problem as with the Clarke and Wright method. Besides on the MDVRP variant (see Section 2.4), this system is inefficient as we do not have a single central depot but several ones dispersed among the customers.

Chapter 5

The VRPLS library

In this chapter, we present the VRP module of the Open LS library [29] that we refer to with the name VRPLS. First, we briefly discuss about the original version of this module that we were provided at the start of this thesis. Then, we present our own implementation with more details by explaining the general architecture and describing each part.

In order to meet our goal to tackle different variants of the vehicle routing problem and to provide a modular and extensible tool, we had to make many changes in regards to the original library. Our implementation is thus based on the original VRPLS module but is independent and can be used separately. These changes are explained and motivated in the following sections.

We implemented our solver in Java version 8 [30]. This version provides new features such as streams and lambdas expressions that we used in some parts of the program.

5.1 Original library

The OpenLS library [29] is an open source Java library which is developed by a team directed by *Pham Quang Dung* from Hanoi University of Science and Technology (HUST) [9]. It uses a local search approach to solve various optimisation problems.

The VRP module of this library can be used independently and allows to tackle the capacitated vehicle routing problem. It uses an iterated minimum routes approach (see 4.3.1).

This approach performs well on the CVRP problems. However, the implementation is not really modular and is difficult to extend to other variants or techniques. Indeed, there is a strong dependency between the different components and thus to change or extend one of these, many parts have to be adapted. Furthermore, some parameters are hard-coded which forces the user to modify the code to specialise his search.

For these reasons, we decided rather than extending this module to recreate an independent module which is inspired by the original library but has a modular architecture. This is this new module that we describe in this chapter.

5.2 General architecture

To create a generic implementation of the library, we needed a high level of modularity in order to allow multiple VRP variants, operators, heuristics and meta-heuristics. To do so, the different parts of the solver are implemented independently as separated modules. Each of them is easily extendable and can be can changed with a minimal impact on the rest of the solver.

In this section, we describe the general architecture of the VRPLS module. In the next sections, we further detail each part and explain how to use or extend them.

As you can see in Figure 5.1, our solver is divided into different components. This allows us to provide a modular and extensible implementation. Furthermore, we separated the generic local search components from the VRP specific components.

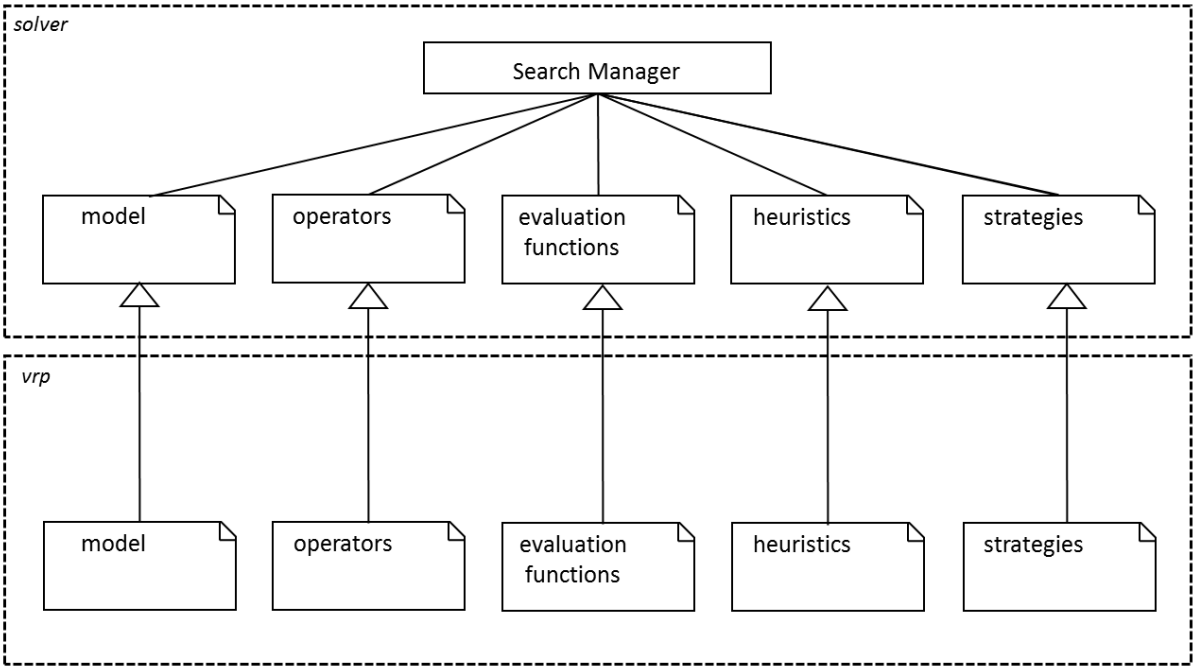

Figure 5.1: General architecture

The `solver` package contains all the classes that are related to the local search dynamic and can be used with different problems. It provides a local search framework through the use of abstractions. This means that our solver can be extended to others problems while keeping the same local search engine.

The VRP specific part of the solver is thus located in another package named `vrp`. The elements of this package are used by the solver to represent and solve VRP problems. The local search engine uses objects defined by interfaces [31], abstract classes [32] or generic types [33]. The VRP specific objects implement or extend these elements allowing them to be used in the search engine.

Both these packages follows the same division into different components. As you can see in Figure 5.1, we can identify 6 different components:

Search Manager: This class serves as controller of the search. It creates and maintains the objects used in the search and drives the local search using the other components. It is generic and does not need to be extended or implemented for a specific problem.

Model: The model is the set of objects used to represent the elements of the problem, the representation of its solutions and the way in which the moves are defined.

Operators: The operators are the classes that generates the neighbourhood by creating moves based on the current instance.

Evaluation functions: These are used to evaluate the cost and the violation of the current solution and the possible moves.

Heuristics: The heuristic chooses the next move from the neighbourhood based on the values returned by the evaluation functions.

Strategies: The search strategies define a type of search and its parameters. It is here that the various meta-heuristics are implemented.

Note that we did not show all the interactions between these components in Figure 5.1 to keep it simple. All these parts are further explained in their respective sections.

5.3 Search manager

The *search manager* is the controller of the solver. It instantiates the objects from the model that represent the problem and interacts with each part to drive the search.

Following the methods of the search strategy, the manager creates and assign an initial solution to the instance of the problem. Then at each iteration, it calls the operators that generate a neighbourhood which is passed to the heuristic. The heuristic then returns a move of the neighbourhood based on the result of the evaluation functions.

These evaluation functions are used to compute the cost and the violation of the potential solution evaluated. They use an incremental evaluation approach based on invariants and differentiable objects (see Sections 3.2.1 and 3.2.2). We consider the constraints before the objectives by using a lexicographical approach in our selection process: a solution is considered better than another if it has a smaller violation value. If the violations are equal, then we consider the cost.

The move returned is finally applied by the manager to the model and the best solution found so far is updated if needed. A functional representation of the iteration process can be seen in Figure 5.2. Note that Instance, Neighbourhood and Move are parts of the model that are described in Section 5.4.

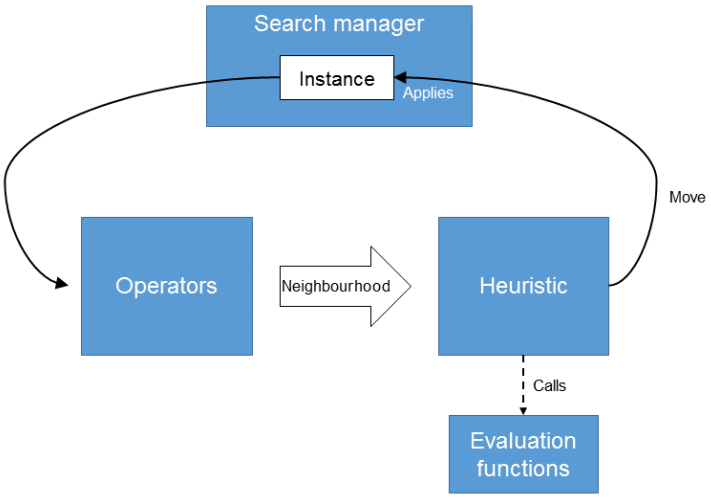

Figure 5.2: Functional representation of an iteration.

When a stopping condition is met, the search stops and the best solution can be retrieved. Eventually, this process is repeated several times with restarts depending on the search strategy used.

5.4 Model

The *model* is the part of the solver that represents the problem and the objects used to solve it. It contains all the classes that are manipulated by the other elements of the solver. For this reason, this part is the most difficult to change without impacting the rest of the library. You can see a representation of the architecture of the model part in Figure 5.3.

In the `solver` package, the search manager uses three different components: the Instance, the State and the Neighbourhood. These components are generic and need at some level to be implemented or extended for a given type of problem. In the `vrp` package, they are completed by various objects that are specific to the vehicle routing problem.

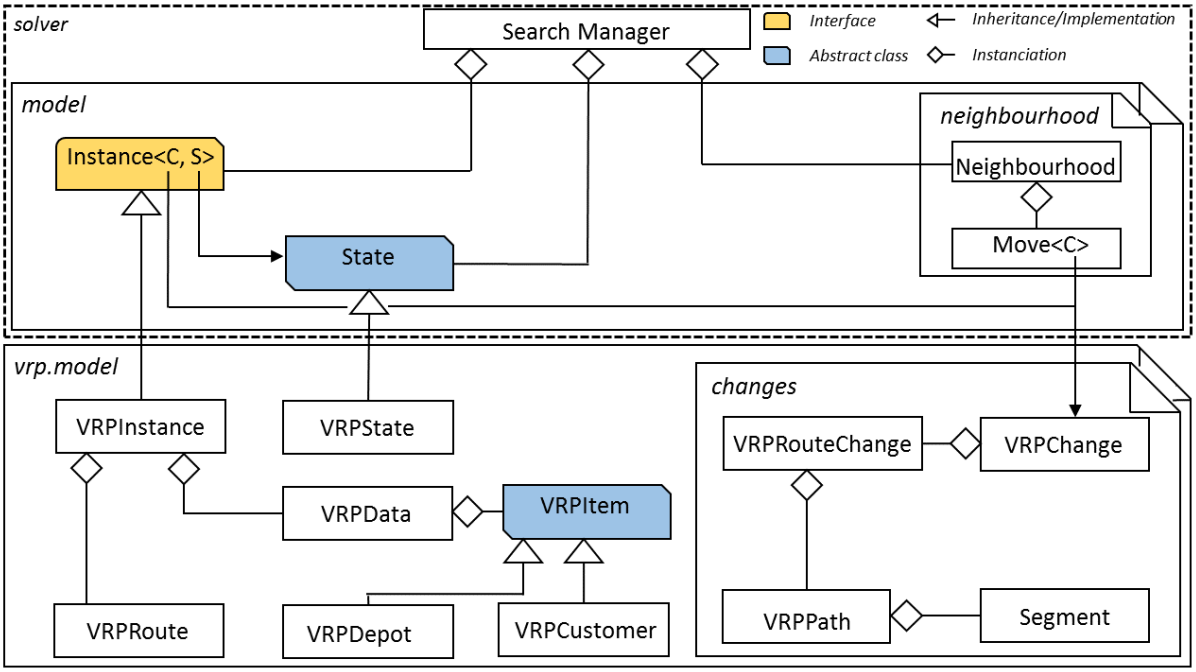

Figure 5.3: Model's Architecture

5.4.1 Instance

The `Instance` is the main component of the model. It is used to represent the current solution of the problem during the search. In the solver, the `Instance` is implemented as an interface which defines the main methods through which the search manager interacts.

In the specific case of the vehicle routing problem, we created a `VRPInstance` class that implements `Instance`. Basically, `VRPInstance` is composed by a set of route objects represented by the `VRPRoute` class, and the immutable data, `VRPData` which contains all the information about customers (`VRPCustomer`) and depots (`VRPDepot`).

While the `VRPRoute` objects change during the search, the `VRPData` and its sub-objects remain unchanged. This class serves to keep all the problem data such as the demand of the customers, the travel costs or other variant-relative data. Beside the cost matrix, it contains two `HashTables` [34] with respectively the customers and the depots. These two classes inherit from the same abstract parent class that defines a `VRPItem`. Each of these items is referenced by an unique identifier in the rest of the application which allows an easy access without data duplication.

A hard implementation choice that we had to make when creating the model was whether to try to make classes that support different variants of the VRP or to create specific objects for each variant that would inherit from the same abstract class. We choose the first option. Indeed, while this option complicates a bit some classes and might cause problems when adding more variants to the library, it allows to mix different variants. As a result, we can use the program to solve problems that have many constraints such as time windows with multiple depots and pick ups.

Routes representation

Each route is implemented as an `ArrayList` [35] of customers ids with additional variables like the starting depot, the ending depot and the vehicle used. While it is more common in VRP solvers to represent the routes with chained structures that form an hamiltonian circle, we choose to use an array list representation to benefit from an access by index to each customer in the route. This implies a higher cost when applying changes to the routes. However, it can speedup the

evaluation of some constraints which are performed more often. Indeed, we need to potentially explore the whole neighbourhood to choose the next move to apply.

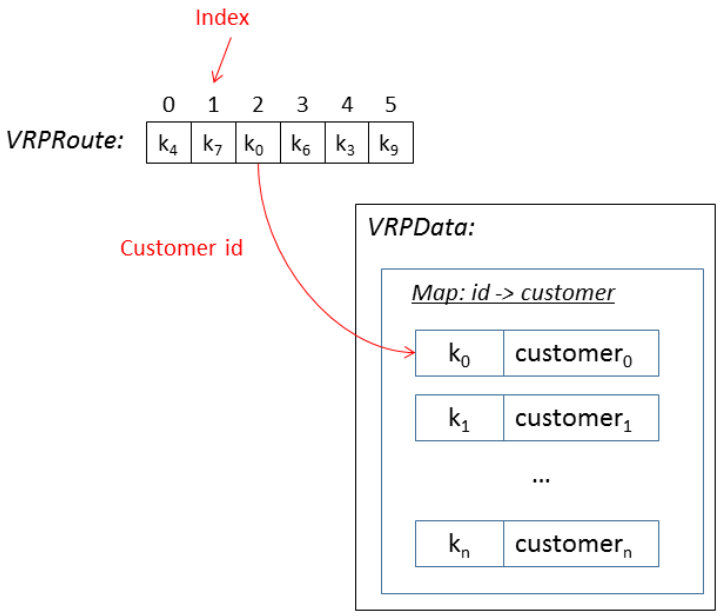

Figure 5.4: VRPRoute architecture using indexes and customer ids.

As you can see in Figure 5.4, the routes consists in arrays of customer ids. Each id refers to a customer in the hashtable that hosts all the customers in `VRPData`. Beside this array, the routes also maintain two fields corresponding to their starting and ending depot and one field which indicates the vehicle assigned to the route. These fields serve only for the Multiple Depot and Heterogeneous Vehicles VRP variants.

The `VRPRoute` objects are stocked in an `ArrayList`. This allows to retrieve a specific route by index.

5.4.2 State

The `State` represents a fixed solution of the problem. This class is used to keep track of the best solution or any relevant solutions during the search. The main difference with the instance is that once created, a state is never modified. The best-known state so far is kept by the `Search Manager` and is returned at the end of the search.

The `State` is implemented as an abstract class. It stocks the cost and the violation of the solution that it represents. It is extended by the `VRPState` class that represent the solution as a two dimensional array of customers ids, each line corresponding to a route.

5.4.3 Neighbourhood

At each iteration of the search, the neighbourhood is generated by the operators before being passed to the heuristic that returns a specific move. The neighbourhood consists thus in a collection of candidate moves.

In the original VRP module, the moves were evaluated as they were generated and only the best(s) one(s) were kept in a collection. Then, a random move was selected among this collection. While this approach is efficient, it lacks flexibility. Indeed, we might want to consider all the moves that improve the current solution and not the best one or use more advanced heuristics such as *Metropolis* (see 3.3.5).

In our implementation, to allow a modular system, we decided to separate the neighbourhood evaluation from the neighbourhood generation. To keep good performances, we decided to repre-

sent the neighbourhood as a stream. The streams are a new functionality of Java 8 [36]. Basically it consists in a lazy collection. In practice, this means that even if the generation and the evaluation of the neighbourhood are separated, they happen in parallel. When the heuristic requires a move of the neighbourhood to evaluate it, it is generated by the corresponding operator. We thus transformed our neighbourhood exploration into a producer-consumer system.

This approach allows us to keep good performances in terms of time and memory. However, due to the fairly new implementation of the streams in Java and some limitations, the streams have weaker performances in some cases. But beside its modularity, this implementation has the advantage of being easily parallelised. Indeed, in our implementation we used sequential streams. However, parallel streams do exist and could improve the performances of the system.

Moves

In the solver part the moves are represented by a class that contains a generic type which represents the changes to the instance. This generic type has to be implemented for any specific optimisation problem. Since it is entirely dependant on the problem, we did not provide any interface or abstract implementation for the changes.

The `Move` class is thus a wrapper for the changes. It also provides additional fields such as the violation and the cost of the move. These fields are initialised as null values and updated during the first evaluation of a move. They can then be used to query the cost or violation of a move without re-evaluating it.

In the case of the vehicle routing problem, the changes are represented by four different classes that instantiates one another. The main idea is to represent each specific change to a route as an alternate path that points to parts of either the same route or other routes.

At the top level the `VRPChange` class represents the changes of the whole current solution to obtain another one. It consists in a `HashTable` of `VRPRouteChange` objects mapped to the index of the concerned route in the `VRPInstance`'s `ArrayList`.

A `VRPRouteChange` instance represents the changes to a specific route. It consists in a map of `ChangePath` objects to the index at which they start in the route.

Each `ChangePath` is basically an alternate path for a part a route. It has starting and ending indexes and contains a collection of `Segment` objects.

These segments refers to parts of a route (which can be the modified route) with three fields: the index of the route, the index of the first element of the segment (inclusive) and the index at which the segment ends (exclusive).

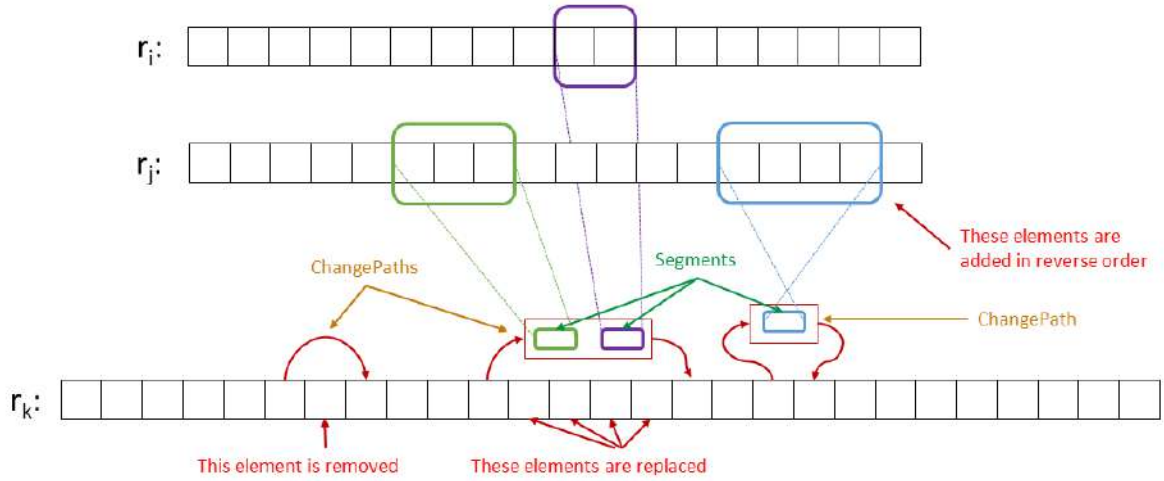

Figure 5.5: Change representation on one route using `ChangePaths` and `Segments`.

Figure 5.5 gives an overview of these features. You can see three examples of possible changes: removing customers by skipping a part of the route with an empty change path, replacing customers with segments from other routes or adding a reversed segment to the route.

With this system, we can represent complex changes while having a minimum memory cost as we point to parts of the routes currently existing in the instance. Note that this representation requires that each customer is part of one and only one route as otherwise we could not represent changes with these customers. In the case of a VRP problem where some customers would not be delivered (and thus not being part of a route), a dummy route would be needed to host these customers.

For the Multiple Depots and the Heterogeneous Vehicles VRP variants, we added extra fields in the `VRPRouteChange` to reflect a change of the depot of the vehicle of the route.

When a move is applied to the instance, the routes that are changed are copied and modified based on the changes and then replace the original routes. We needed to make this a two step process since we do not want to directly modify a route as a segment could refer to it. This point is a weakness of our implementation.

5.5 Operators

As explained in Chapter 3, the operators generate the neighbourhood at each iteration. The interface `Operator` defines two methods that have to be implemented by a class in order to be used as an operator:

generate which receives the manager in parameter and returns a `Neighbourhood` object that contains a stream of all the possible moves based on the current state of the instance.

size which computes the number of moves in the neighbourhood. This information can then be used in the heuristic to easily select random elements in the neighbourhood.

The operator system can be used in a hierarchical way by using an aggregation operator that aggregates the neighbourhoods produced by a set of sub-operators. This architecture allows to have multiple specialised operators, each one of them producing a specific type of move, which ease their implementation. Figure 5.6 shows a functional view of this system.

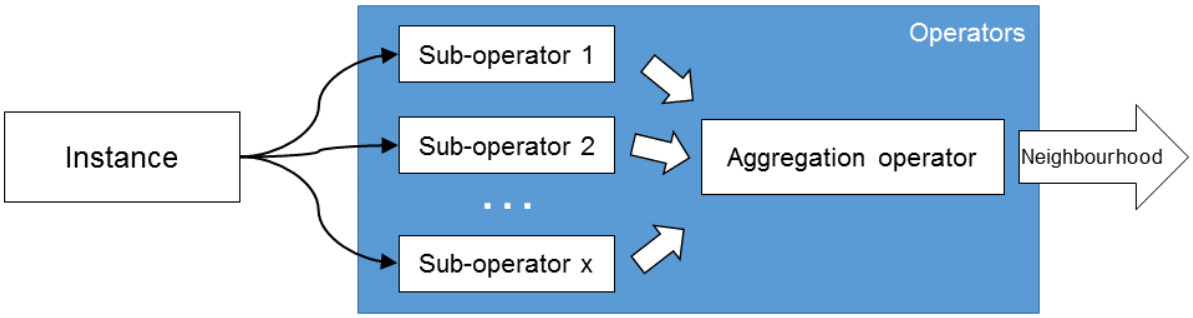

Figure 5.6: Functional representation of the operators.

5.5.1 Available operators

Here are the available operators in our solver:

AggregatedOperators: This generic operator allows the possibility of having multiple operators by having an array of sub-operators and concatenating their generated neighbourhoods into a single one.

RelocateOperator: This operator relocates a customer into a different route as specified in Section 4.2.1.

SwapOperator: This operator swaps two customers from different routes. For more details, see Section 4.2.2.

KExchangeOperator: This operator extends the `AggregatedOperators`. It regroups the `K2ExchangeOperato` and the `K3ExchangeOperator` that generate K-exchange type moves (see Section 4.2.3). Note that other K-exchange operators could be added.

CrossOperator: This operator generates cross-exchange moves as defined in Section 4.2.4.

ClarkeWrighOperator: This operator merges two routes into one. It can be used in a Clarke and Wright route construction strategy. (see Section 4.3.2)

ChangeDepotOperator: This operator is used in the Multiple Depots VRP variant to change the start and end depots of a route. When initialised, a boolean field can be used to specify if a route can have different start and end depots.

ChangeVehicleOperator: This operator is used in the Heterogeneous Vehicles VRP variant. It changes the vehicle assigned to a route.

5.6 Evaluation functions

In this section, we describe the available evaluation functions in our library and their implementation. Basically an evaluation function is an object that implements two methods: `evaluate` which takes the instance in parameter and returns a double value corresponding to the evaluation of the instance and `evaluateDelta` which has a move as additional parameter and evaluates the delta of the evaluation that this move would cause if applied.

To have a modular and easy implementation, we decided to have the same interface for the constraint functions and the cost functions. This implies that a constraint function can be used as a cost function where the returned violation would be considered as an additional cost. The inverse is true and a cost function could thus be used as a constraint function even if the applications are quite limited.

Another point of our implementation is that we encourage to create the evaluation functions in a hierarchised way. The idea is to use an evaluation function that hosts some sub functions. When an evaluation method is called, it is propagated to the sub functions and the results are aggregated (for example with a sum) and returned.

Adding an additional constraint in our solver consist thus in passing the evaluation function corresponding to this constraint to the aggregation function that serves as general constraint function.

Note that currently, our implementation does not allow to use constraints or cost functions only on a specific part of the instance (for example only on some routes or some customers). When a constraint or a cost function is used it is applied on the whole instance. However, such system would be implementable with a new abstraction that would remember the elements on which the function has to be applied and would only consider them in the evaluation. This abstraction would then have to be extended by the part-specific evaluation functions.

5.6.1 Architecture

The implementation of our evaluation functions follows an incremental evaluation approach as described at the end of Section 3.2. It uses two different components: *invariants* and *differentiable objects*.

Invariants (see Section 3.2.1) are the functional part of the evaluation function. They are defined by the abstract class of the same name.

Differentiable objects (see Section 3.2.2 are helper objects that maintain some values to ease the computation of the changes by the invariants.

For example, the `RouteCost` invariant (which calculates the total cost of each route) uses a differentiable object called `CumulatedCost`. This object is a table of the cumulated cost of each route at each customer. When `RouteCost` evaluates the cost of a move, it uses this table to compute the delta of cost in constant time. Indeed the cost of a segment can be computed as the cumulated cost at its last customer minus the cumulated cost at its first customer.

The differentiable objects are hosted in an object called `DiffStore`. It consists in a hashtable that maps the differentiable objects with a keyword corresponding to their name. When the `DiffStore` is initialised, it is passed to all the invariants which can add their required differentiable objects if needed.

During the search, the invariants have access to the `DiffStore` and can thus look for a specific differentiable object. This system allows multiple invariants to use the same differentiable object. At each iteration of the search, the manager propagates the changes to all the differentiable objects which are thus updated.

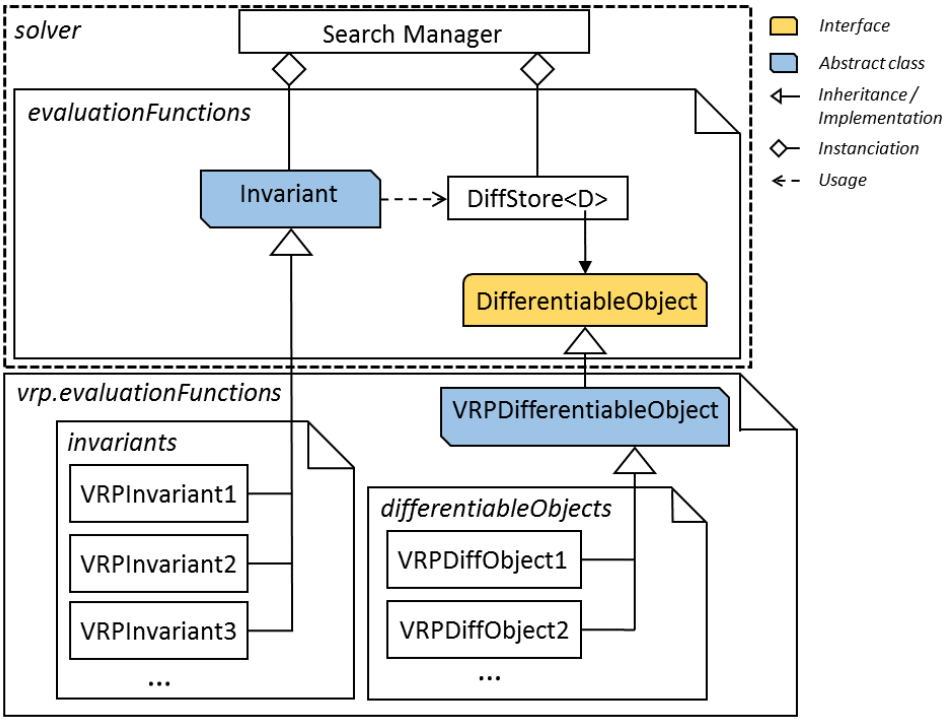

Figure 5.7: Evaluation functions architecture

In Figure 5.7, you can see the architecture of our invariant system. The `DiffStore` is a generic object that does not need to be specified for each problem. To be added in the store, the differentiable objects must implement the interface `DifferentiableObject`. For the VRP, we provided an abstract class that already implements this interface and provides common methods for all the VRP differentiable objects. The VRP invariants need to extend the abstract class `Invariant`.

5.6.2 Available Invariants

Here are the available invariants in our solver:

SumInvariant: This invariant is generic. It aggregates sub-invariants by summing their results.

RouteCost: This invariant computes the travel cost of each route.

WaitingCost: This invariant can be used in some variants of the VRP with time windows. If the delivery vehicle arrives at a customer before the beginning of the time window, it has to wait. In some variants this waiting time has to be minimised. This invariant computes the total waiting cost of each vehicle.

RouteLength: This invariant computes the route length violation: For each route, if the route is longer than a given threshold, it adds the excess as violation.

RouteCapacity: This invariant computes the violation of the capacity constraints of each route.

TimedDelivery: This invariant enforces the delivery on time constraint by summing the latenesses when a vehicle arrives after the end of the time window of a customer.

RoutePDCapacity: This invariant is a special version of the `RouteCapacity` invariant for the VRP with Pick-ups and Deliveries variant. Indeed, as the load of the vehicles can augment, the capacity violation is more complicated to compute.

5.7 Heuristics

As we explained in Section 5.4.3, the neighbourhood is implemented as a stream. The logical extension of this system is to implement our heuristics as filters that return only one element of the stream. For the basic heuristics, we separated them into smaller filters called explorers and selectors that can be combined to create a heuristic.

Each of these filters is basically a static function. We provided a class that receives in parameter one selector function and an undetermined number of explorer functions. When queried for the next element of a neighbourhood, this object applies the eventual explorers on the neighbourhood and then the selector to return a move. You can see a functional representation of our heuristic system in Figure 5.8.

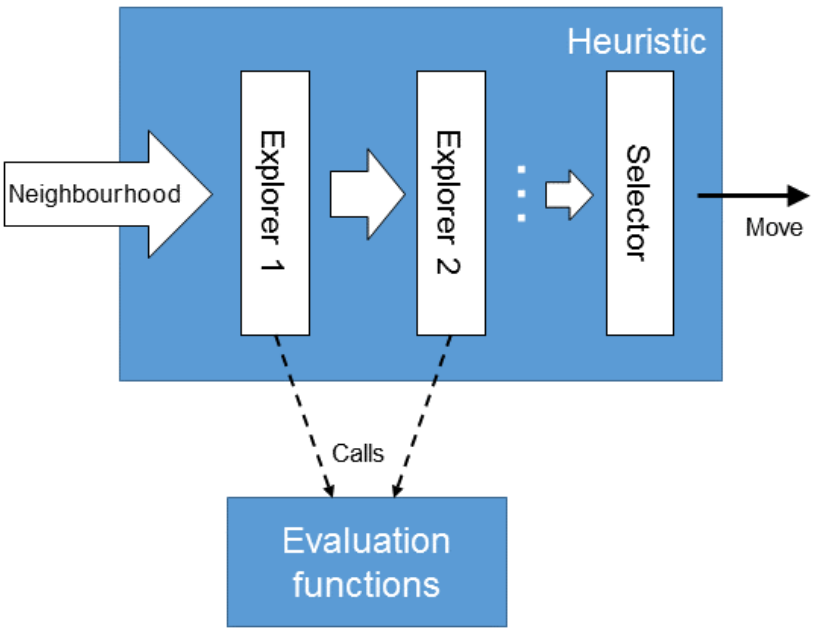

Figure 5.8: Functional representation of the heuristic.

5.7.1 Explorers

The explorers filter out a part of the neighbourhood by applying a restriction on it. Currently, there are three explorers:

Better which returns the moves that improve the current solution.

BetterOrEqual which does the same as better but also allows the moves that lead to an equivalent solution.

Best which only returns the best move(s) of the neighbourhood. Note that this is an explorer as there can be more than one best move. It has also a bigger performance cost as it requires to explore the whole neighbourhood to return the best moves.

5.7.2 Selectors

The selectors return only one move of the neighbourhood. We provided two selectors:

First consists in returning the first element of the neighbourhood. It is bad in terms of diversification but has a huge advantage in terms of performances when combined with the *better* explorer.

Random consists in returning a random move of the neighbourhood.

Note that the random selector has a performance cost. Indeed, as the stream is a lazy structure, the function that we use to return a random element has to iterate the stream until its end to guarantee the same probability for each element. To address this issue, we compute the size of the neighbourhood before generating it in the operators so that we can choose beforehand a move to return. However, there are still some limitations to this solution:

First, it may not be always possible to compute the size of the neighbourhood beforehand. We are able to do it with the operators present currently in our solver but one might want to add an operator where the number of perturbations generated cannot be computed. In this case, we need to fall back on the basic random selector approach.

Secondly, we need to iterate the stream until this move which means that on average we still generate the half of the neighbourhood.

Finally, when the neighbourhood is filtered by an explorer, its size changes. In this case, we are also forced to fall back on the original random method. In practical, this problem is minimised with the *best* explorer as we are able to compute the size of the restricted neighbourhood in our implementation.

Note that not all of our heuristics are implemented as a combination of these filters. Indeed, the metropolis heuristic for example is more complicated since it requires updates of its temperature parameter. It is thus implemented as another class.

5.7.3 Metropolis

As explained in Section 3.3.5, the metropolis heuristic uses a temperature parameter t which is used to have a small probability of selecting a worsening move in order to introduce diversification in the search.

Due to this parameter t, we had to create a separated class for this heuristic. This class is generic and can be used with different types of problems. It uses the random selector to obtain a random move. The move is then evaluated and following the metropolis equation, the move is returned or an empty move instead. The parameter t is updated by the `SimulatedAnnealing` search strategy explained in Section 5.8.1.

5.8 Search

The search process is driven by the Search class. This class interacts with the search manager during the whole process. A key point of our implementation was to provide a way to implement easily different meta-heuristics and search strategies while keeping the system modular. Indeed, as explained in Section 3.4 a meta-heuristic can take many forms.

To solve this problem, we introduced the concept of *search strategy*. A search strategy is basically a set of directives that alter the basic behaviour of the search to introduce diversification or intensification.

In practical, to implement that, we created the `SearchStrategy` interface. This interface provides five methods that are called at different parts of the search. These method receive the search manager in parameter and can be used to alter the behaviour of the search or modify components.

To implement a new meta-heuristic, one has to implement this interface and use the appropriate method(s) to alter the default behaviour of the search. The methods that are not needed can be left empty.

Here are the five available methods of this interface:

preProcessing: This is called before a new search. It can be used to assign a first solution to the instance or launch a sub-search to find one.

iterProcessing: This method is called at the end of each iteration just after a new move is applied and the best solution is updated.

postProcessing: This is called at the end of the search. It can be used to restart the search or launch a new search.

restrict: This method receives the neighbourhood as an additional parameter. It is called just before applying the heuristic on the neighbourhood and can be used as an additional filter before the heuristic. For example, it is used in the tabu search. For more details see Section 5.8.1.

moveHasBeenAppliedEvent: This method is called when a new move is applied to the current instance. It has the move in parameter.

Note that other methods could easily be added if needed. For example, we could imagine a method that would be called each time that a new best solution is found and that would launch a sub-search with this best solution to refine it.

We also created a `StackableSearchStrategy` that allows to use multiple search strategies at the same time. When initialised, this class takes multiple search strategies as parameters. The methods of the given search strategies are called in the order in which they are passed.

5.8.1 Available search strategies

We provided four different search strategies that can be combined. These are described in details in the following sections.

Iterated Minimum

This strategy is the one used in the original VRPLS module. It has been described in the Section 4.3.1. This strategy assigns randomly the customers to a minimum number of routes which is computed based on the capacity of the vehicles and the total quantity to deliver. A search is then launched until no better move can be found. The search is then restarted from a new random assignation with eventually more routes if no feasible solution has been found.

Singleton routes

This search is our implementation of the vehicle routing problem using the Clarke and Wright approach (See Section 4.3.2). Basically, the search starts by assigning each customer to a different route and merge these step by step following the saving criterion. We extended this original method by mixing different operators to improve the routes during the construction.

The main advantage of this method is that it is fast and it performs very well on instances with clustered groups of customers. However, on other instances it can fail to find a good solution. Another problem is that a minimal number of routes is not guaranteed.

Simulated annealing

The Simulated Annealing search (see Section 3.4.2) is implemented by using the Metropolis heuristic described in Section 5.7.3. This search strategy updates the temperature parameter t of the heuristic following a temperature function. This function is specified by the user at the initialisation. Depending on the given temperature function, the performances of the search might vary.

This kind of search is particularly adapted with a large search space where a lot of diversification is needed but might fail to find a good solution compared to other techniques due to its weaker intensification.

Tabu search

As described in Section 3.4.3, the tabu search consists in declaring some moves as tabu to avoid re-exploring parts of the search space. The difficulty with a tabu search is to manage the store that keeps all the moves or solutions that are tabu.

To do this, our implementation is based on the Taillard tabu search *Tabu List* structure which is explained in [37]. It consists in using the cost of a solution as tabu value. The idea is to use a tabu store of a given size T. When a solution is found, the slot of the tabu store at the index $i = cost \pmod T$ is updated with an iteration number which corresponds to the iteration at which the solution is available again.

When a move is considered, if the current iteration is lower than the one in the slot corresponding to the cost, the move is rejected. A boolean value passed at the initialisation of the search strategy allows to define an aspiration criterion which bypasses the tabu if the solution found is better than the best one so far.

The weakness of this approach is that it might consider solutions not yet explored as tabu since two solutions could have the same cost or the size of the tabu list could lead to a collision. However, it leads to a good diversification and prevents the search to stagnate. The main advantage is in terms of simplicity and performances. Indeed, using the cost of a solution as tabu value is very easy to implement and efficient in terms of performances. Finally, this implementation allows us to provide a generic tabu method as it uses the cost function provided by the user. As long as the cost function is properly defined, this method can be used with any kind of problem.

Note that this is implemented as a search strategy rather than a heursitic as it is easier to deal with the store this way. Our implementation uses two methods from the `SearchStrategy` interface: `restrict` which performs a pre-filter on the neighbourhood that allows only moves that are not yet tabu and `moveHasBeenAppliedEvent` which updates the tabu store.

5.9 Supported VRP variants

Currently, our solver supports 6 VRP variants:

1. Capacitated VRP (2.1)

2. Distance constrained VRP (2.2)
3. Time windows (2.3)
4. Multiple depots (2.4)
5. Heterogeneous Vehicles (2.6)
6. Pick ups and deliveries (2.5)

These variants can be mixed together. Also, as explained in Section 5.6, the constraint functions can be used as cost functions and inversly. This extends the flexibility and the range of problems that we are able to deal with.

Adding new problem variants, heuristics or search strategies should be easy and does not require extensive modifications to the existing parts of our solver. Furthermore, the local search engine can be used to tackle different problems.

5.10 Others components

Beside the components explained previously, our library also contains tools that are not necessary for the functioning of the search but may help the user.

5.10.1 VRP Solution checker

The VRP solution checker is a tool that checks a potential VRP Solution to see if the constraints are respected and the cost is calculated correctly. It takes a VRPState object as parameter.

It is completely separated from the solver and implemented independently to guarantee that a potential error in the solver would not be replicated in the solution checker and would be detected.

5.10.2 App models

The app models are classes that provide a template for an application using the solver. We provided two templates:

BasicApp which specifies a search in a main class.

CommandLineApp which specifies a search based on command line arguments.

5.10.3 Listeners

State listeners are objects that serve as output during the search. They are triggered at each step of the search and can be used to report the advance of the search in a graphical interface, on the standard output or in a file. We provided three listeners:

SolutionFilePrinter: Each time a new solution is found, it writes it in a given file.

SolutionOutPrinter: Writes the new solutions on the standard output.

VRPGraphViewer: Displays the advance of the search on an interface with the evolution of the violation and the cost of the current solution and a graphical display of the best solution so far.

5.10.4 Readers

Readers are utility classes used to read VRP problem instances in different formats and return a `VRPData` object corresponding to the problem.

5.11 Conclusion

In this section, we provide a brief summary of this chapter and the key points of our version of the VRPLS module.

Rather than extending the original module which would have been difficult due to it's architecture and lack of modularity, we created a new version. This version is designed to be modular and easy to extend. To do so, we separated the local search engine from the VRP specific part. We also divided these two parts in several different components. Each one of them deals with a specific part of the search and can be modified or extended mostly independently.

We implemented different operators, heuristics, evaluation functions and search strategies that can be combined to provide various approaches to VRP problems. Our version of the module is also able to tackle 6 different variants of the VRP and can easily be extended to deal with other variants or even other optimisation problems.

Chapter 6

Experimental Plan

In this chapter, we discuss the test results of our solver and the test protocol that we used. These measures have been done on several sets of standard benchmarks instances from the literature.

First, we detail our test protocol and the instances used. Then, we present our results and provide a small analysis of them along with comparisons between our different approaches and the original algorithm.

6.1 Test protocol

Our test protocol is done in several steps:

- The first step is to make the solver run on a set of benchmark instances with the configurations (i.e. the heuristic, search strategy, operators and other elements) to test. Ideally, multiple runs have to be done on each instance in order to cover a representative range of possible performances due to the randomness of our local search approach.
- The next step is to gather these results and aggregate them in order to provide meaningful data without introducing bias.
- Finally we were interested in providing a way to easily compare our different approaches between them as well as with other solvers. To do so, we used performances profiles and geometric mean.

Our test scripts were done in Python 3.5.1 [38], using an executable jar [39] of our solver. We tried to make them easy to use and parametrisable in order to allow any user of the library to use them to perform his own tests. They are provided in annex of this document along with an user manual (see Section B).

In the following sections, we further explain the details of our test protocol and its implementation. First, we explain how we performed several runs of the same tests and how we aggregated these. The second part presents the performance profile tool that we use to compare our results. We also detail the material used and the conditions of our tests. Finally, the last part describes the benchmark instances that we used, their features and the reasons why we choose them.

6.1.1 Results Computation

The main difficulty of testing a local search approach is due to the randomness that many variants introduce in order to diversify the search. This means that, to have meaningful results and avoid bias, the algorithm has to be tested multiple times and then the results aggregated.

The script that runs the tests is called `BenchmarkRunner.py`. It uses a pool of threads to perform multiple runs at the same time. Note that in order to not impact the performances of

the solver and thus the results, the number of threads in the pool must be inferior or equal to the number of available cores on the testing machine. This way, an active run is never be put to sleep.

The script `Main.py` allows to set up a test configuration and calls `BenchmarkRunner` with it. The test configuration is a string of parameters which is send as line command parameters along with the instance path to the jar executable when it is called.

This script can take many parameters that can be used to specify the directory of the test instances, the directory where the results are saved, the number of runs to do by instance, the number of threads to use, the allocated time per instance and eventually a file containing best known solutions that are used to stop the search sooner when these solutions are reached.

The results of each run are saved in a specific file which contains for each update of the best solution, the solution found along with its cost, its violation and the time elapsed since the beginning of the search. From these information, we can compute meaningful statistics such as the minimum or average cost found, the variance of the runs, the minimum or average time taken to reach a given threshold and so on.

Timelines

We can also compute what we call a *timeline*. It consists in a decreasing function of the cost of the best solution according to the time elapsed. This function represents the evolution of the solution found along the advance of the search. In practical, a timeline can be stored as a sorted set of pairs time-cost. The cost at any given time t is the cost at the last entry in the timeline before t. Note that the timeline can also be computed for the violation but we are mainly interested by feasible solutions. The timeline starts thus at the first feasible solution.

A timeline can also be generated for multiple runs. In this case, for each entry in one of the timelines at a given time, we take the mean of all the costs for each run at that time. For example if at a time t one of the runs is updated, the value in the timeline is the mean of this cost and all the costs at the last updates of the other solutions.

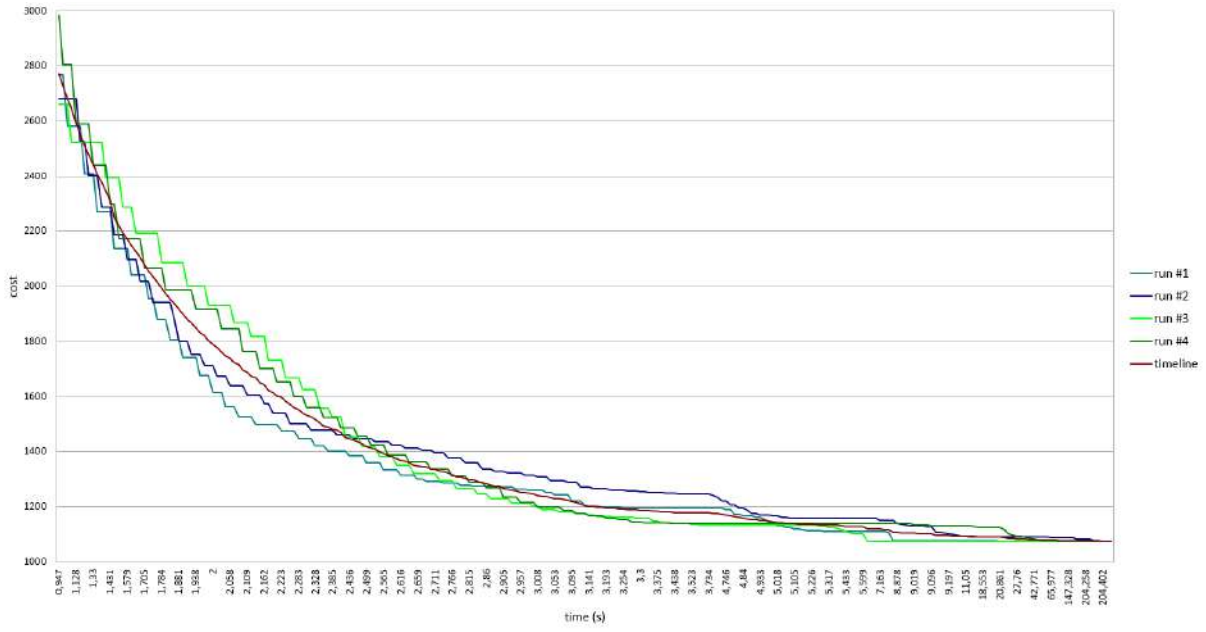

Figure 6.1: Illustration of timeline computation

In Figure 6.1, you can see an example of timeline. The timeline which is in red was aggregated from four different runs with the iterated minimum search approach (4.3.1) and the random best

(5.7.2, 5.7.1) heuristic on the instance `A-n55-k09` from the Augerat set (described in Section 6.1.4).

This timeline is quite representative of the variance of our different runs which is not so important as we expected. Indeed, as the violation is minimised before the cost, even if the initial solution differs a lot due to the random assignation, the first feasible solutions are close of each other because the moves applied to satisfy the constraints generally lead to similar solutions. The more restrictive the constraints are, the closer the initial solutions are.

Note also that the time scale of this graph is not linear as each x value corresponds to one entry in the timeline. Typically, the best solution evolves a lot at the beginning of the search before the first restart. For the rest of the search, the solution evolves by small steps leading to a near flat curve.

At the end, we have timelines that represents the average evolution of the cost during the search for each instance. Similarly, we can create timelines that represent the evolution of the time needed to reach a given cost. In this case, we measure the average time for each cost value reached by one of our runs. We can then use them to compare the configurations between them or retrieve data that is used to plot graphs or performance profiles. For example, if we are interested by the average time taken to reach a threshold of cost, we can take the y value of the time based timeline when it crosses the x value corresponding to this threshold.

6.1.2 Performance profiles

After computing the results, we want a way to compare the performances of our solver with different approaches or algorithms. Furthermore, we want to be able to do so without having to rerun all the tests when adding a solver to compare. Performance profiles [40] allow this.

They consist in cumulative distributions of a performance metric that can be used to plot and compare solver performances. Here is how a performance profile is computed:

- For each $t_{p,s}$ (the result of the benchmark of the search method $s \in S$ on the instance $p \in P$), we compute a performance ratio $r_{p,s}$ which is a comparison of the performance of s with the best search method for p:

$$r_{p,s} = \frac{t_{p,s}}{min(t_{p,s} : s \in S)} \tag{6.1}$$

- Then, we define $\rho_s(\tau)$ as the performance metric of s which is the probability to have a performance ratio within a factor τ of the best possible ratio:

$$\rho_s(\tau) = \frac{1}{|P|} size(p \in P : r_{p,s} \leq \tau) \tag{6.2}$$

- The performance profile is the cumulative distribution function of ρ.

We can then plot different performance profiles on a graph to graphically compare them. You can see an example where two of our heuristics are compared in Figure 6.2. The y axis indicates the percentage of solved instances by a heuristic. The x axis uses a time ratio scale. It corresponds to the ratio of the time used by the heuristic compared to the time used by the best baseline solver. For example it means that if the curve of a solver is at a y value of 65% at a time ratio of 1.5, if given 1.5 times the time needed by the best solver, this solver is able to solve 65% of the instances. In short, if the curve of a solver is higher than another, it means that this solver is able to solve more instances in the same time.

In the first example, both heuristics are used as baseline. That means that the time used in the time ratio is the best time between both. The two curves thus start at 1. However, we can select a specific solver as baseline. You can see an example in Figure 6.3 where the random best

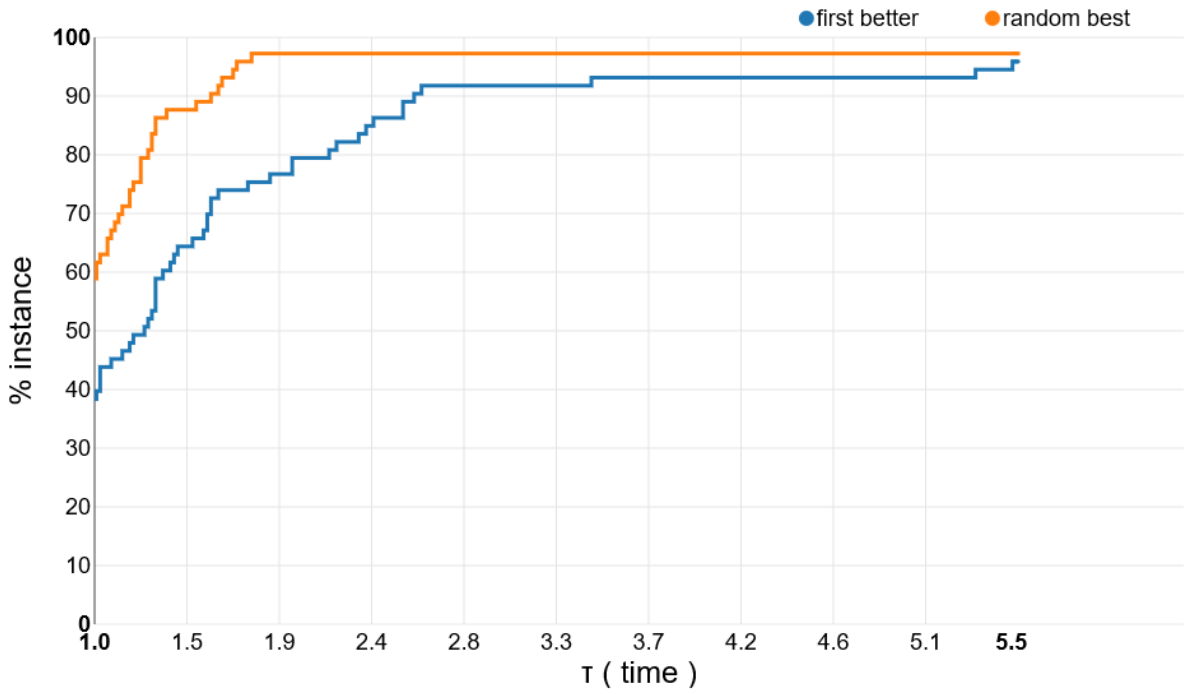

Figure 6.2: Performance profiles of first better and random best heuristics

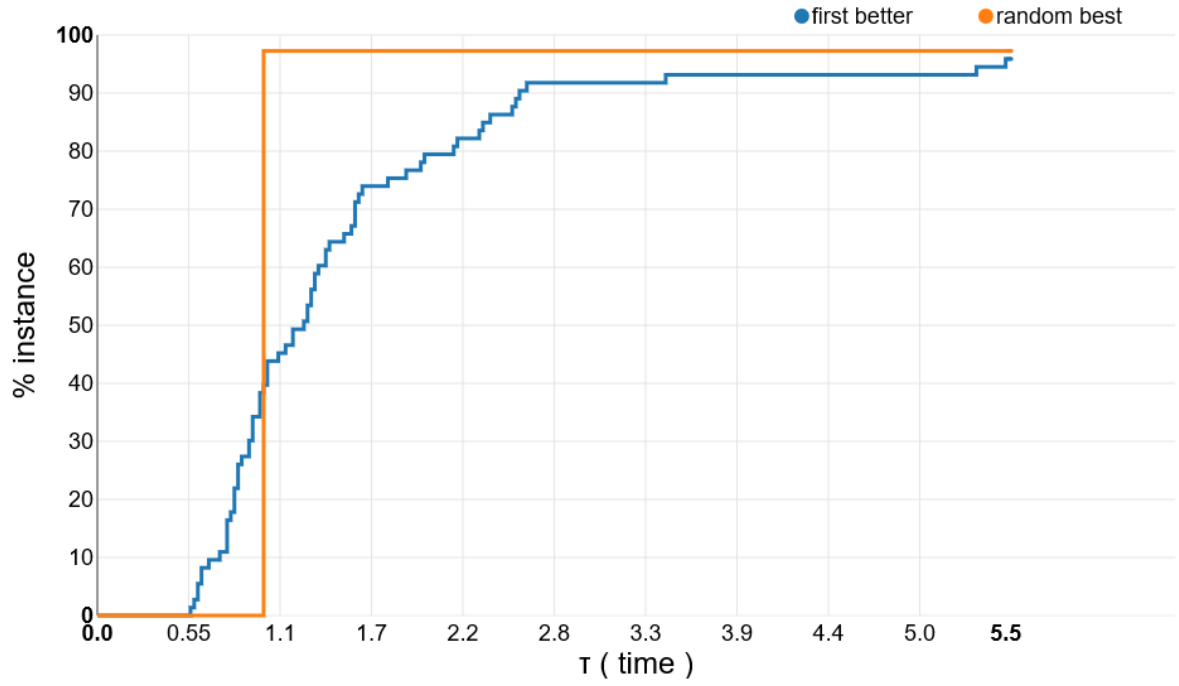

Figure 6.3: Performance profiles of first better and random best heuristics with random best as baseline

heuristic has been selected as baseline. This means that now the time ratio scale on the x axis is expressed as a ratio of the time taken by the baseline. We can see in this graph that for around 40% of the instances, the first better heuristic takes less time or an equal time than random best but that in order to solve 90% of the instances it needs around 2.5 times the time taken by random best and that ultimately, it is unable to solve as much instances as random best. Note also that both heuristics were unable to solve a few instances which is why both curves do not

go higher than 98%.

These two graphs were plotted based on four different runs of the whole Augerat set of instance (described in section 6.1.4) with the iterated minimum search approach (4.3.1). To plot our graphs, we used an online tool currently in development by Sascha Van Cauwelaert [41]. This tool allows us to upload our data as a `Json` file and then to plot our profiles with various parameters such as the baseline to select or the scale used.

6.1.3 Material used and test conditions

All our tests were done on the workstations of the Intel computer room. These computers use Intel(R) Core(TM)2 Quad Q6600 @ 2.40GHz processors and have 4 Go of RAM memory [42]. We ran our tests during the night or on unused machines in order to be able to use the full extend of the machines' resources and make sure that our measure are not influenced by processes ran by other users.

Due to these limitations on the available material, the high number of different components to test, we were forced to limit ourselves in the time allocated to the tests and the number of runs done on each instance. For each test on each instance, we did at least four different runs of 15 minutes. While ideally we should have more runs with more time allocated, we hope that the results that we obtained are sufficiently representative of the real performances of the different components tested.

Note that we made sure that no anomaly was present in our test data by checking the variance of the results obtained on the different runs. For a few tests, the variance was too high compared to the average value found. We thus remade these tests in order to obtain meaningful results.

As a final remark, for all our time measurements, we consider an instance as solved when the solution obtained by our solver is feasible (ie. has a violation of 0) and has a cost which is higher than the best known solution by at most 5%. This approximation was necessary for three reasons:

First, finding the optimal solution of an instance can require a lot of time and can be an unreliable objective to compare algorithms: one could have "luck" and find really quickly the optimal solution on specific instances whereas another which has near optimal results on most instances would be considered as less efficient. Besides, in most real-life applications, we are more interested in finding quickly an efficient solution rather than taking a lot of time to obtain the optimal solution.

Secondly, as the best known solutions were obtained by other solvers programmed with other languages and possibly different rounding rules, the evaluation of the same solution could yield different values. Having a success threshold rather than a fixed objective value allows us to accommodate of this problem.

Finally, many of the best know results were obtained with highly specific algorithms or exact approaches. As explained in Chapter 3 local search is better suited to find a near-optimal solution with good performances than the optimal solution.

6.1.4 Benchmark instances

To measure the performance of our solver, we selected some well-known benchmark instances. We could have generated our own instances but we found more interesting to test our solver on existing sets of instances. The difficulty was to find instances corresponding to each variant of the vehicle routing problem along with best known results. Many sets of instances are provided without their solutions. Ideally, we want at least to have the cost of the best known result in order to measure how close our own solution is. It is even better if the best known results is proven optimal.

Another difficulty was to accommodate for the various file format in which the instances are provided. Indeed, rather to have a common file format for VRP related instances, each author

has its own way to encode the information. We thus had to program different readers for the different sets of instances that we tested.

Furthermore, many instances had their solutions provided without any information about how the solution was generated. We had to track down the original paper in which the instances were proposed in order to have information about the rounding rules or the variations of the constraints used.

We obtained our instances mainly from two different sources: The first one is the Networking and Emerging Optimisation (NEO) research group's site [43]. The NEO research group is a part of the department of languages and computer sciences of the University of Malaga (Spain). Their site has a section dedicated to the VRP where many instances are provided for different variants of the problem along with some best known solutions.

Our second source was the VRP-REP site [44]. The vehicle routing problem Repository is a project which aims at providing the VRP research community with a collaborative open data platform. This platform proposes an universal format for instance and solution files. Sadly, the site has currently only instances for a few variants of the VRP.

Here are the instances that we used for each variant of the problem:

CVRP The Capacited Vehicle Routing Problem (see Section 2.1) has been tested on the sets from `Augerat` [45]. It contains three different sets:

- `Augerat-A` contains 27 instances in which customers have random localisation inside a grid of 100 x 100. Their demands are picked randomly following a uniform distribution between 0 and 30. For 10 percent of the customers, the demand is multiplied by 3.
- `Augerat-B` contains 23 instances containing customers placed inside a grid of 100 x 100. In this set, the customers are clustered. Their demands are generated in the same way as *Augerat-A*.
- `Augerat-P` contains 24 instances. All of them are either from *Augerat-A*, *Augerat-B* or another existing set. The particularity of this set is that the capacity is reduced in order to increase the number or routes required to satisfy all the customers.

DVRP To test the distance constraint variant of the problem (see Section 2.2), we used the set of instances from `Christofides, Mingozzi and Toth (CMT)` [13]. This set is a mix of CVRP and DVRP instances. It contains 14 instances. seven of them are basic CVRP instances with either randomly dispersed customers or clustered customers. The seven others are the same instances with a service time and an additional distance constraint.

Note that we obtained these instances from two different sources. Initially we used the files from the VRP-REP site due to their easier format. However, this set was incorrectly labelled as a CVRP set and the additional constraint along with the service time was missing in the seven DVRP instances. For these instances, we thus obtained solutions better than the best known ones as we did not considered these constraints. This made us loose a lot of time trying to find an nonexistent error in our solver as we did not thought that the instance files could be wrong.

For these tests, we used the instance set obtained from the NEO site which has the correct instance files with all the constraints.

MDVRP For the Multiple Depots variant (see Section 2.4), we used the instances from the `Cordeau` dataset [2]. This set is composed of 23 instances which are generated randomly. Note that some of these instances are really big with up to 360 customers and 9 depots.

VRPTW To test the VRP with Time Windows variants (see Section 2.3), we used the well known sets of `Solomon` [46]. These instances were originally proposed in 1987 and are the most used for this variant. Each instance contains 25, 50 or 100 customers. Three

alternatives for customer locations are considered: in the sets `R1` and `R2`, the customers are positioned randomly inside the grid, in `C1` and `C2`, the customers are clustered and in `RC1` and `RC2`, a mix clusters and randomly positioned customers is proposed. The sets `R2`, `C2` and `RC2` have a larger capacities, allowing more customers per route.

VRPPD For the VRP with pick up and deliveries (see Section 2.5), we had difficulties to find a suitable set of instances. Indeed, all the sets that we found were for the periodic VRP variant which we do not support currently in the library.

We tested our solver on the `Breedam` set which has both pick-ups and deliveries and time windows. Unfortunately, these instances were provided without best known results. We thus have no basis of comparison for our results.

HVRP We were unable to find test instances for the heterogeneous vehicles variant in the literature. To ensure that this variant works correctly, we used custom made instances and our solution checker utility to check the solutions found. However, we have no proof that these solutions are optimal.

6.2 Results and analysis

In this section we present a few interesting results of our solver. We start by a comparison of our basic search methods and heuristics. Then, we compare more advanced methods such as the *tabu search* and the *simulated annealing*. We follow by a comparison with the original solver and finally, we provide results for different problem variants.

6.2.1 Heuristics Comparison

This first comparison is between what we consider as the three basic search methods of our solver:

Iterated minimum search with random best heuristic (explained in Sections 4.3.1, 5.8.1, 5.7.1 and 5.7.2): This search method is basically our own implementation of the one used by the original library. The iterated minimum routes approach guarantees a minimal number of routes and a good diversification while the random best heuristic is good in terms of intensification.

Iterated minimum search with first better heuristic (explained in Sections 4.3.1, 5.8.1, 5.7.1 and 5.7.2): This search method uses the same search as the previous one but a different heuristic. The first better heuristic is weaker in terms of intensification but faster which means that more solutions are explored. It might perform better on some instances.

Clarke and Wright search with random best heuristic (explained in Sections 4.3.2, 5.8.1, 5.7.1 and 5.7.2): This search method is completely different as we start from a solution with one route per customer and merge these routes until it is no longer possible. Initially, it is a route construction approach. However, by mixing it with other VRP operators that can optimise the routes along the merges, we use it as a local search approach. In most cases, it yields worse results than the other approaches but it might perform very well on some instances, particularly if these are clustered.

Note that we have not included the random better heuristic in our comparison as it has a poorer intensification than random best and does not benefits from as faster exploration as first better. This heuristic is thus not competitive with the two others.

This comparison was done on three sets of Augerat instances (described in Section 6.1.4) with 4 different runs of 15 minutes per instance per method. In Figure 6.4, you can see the results of this comparison.

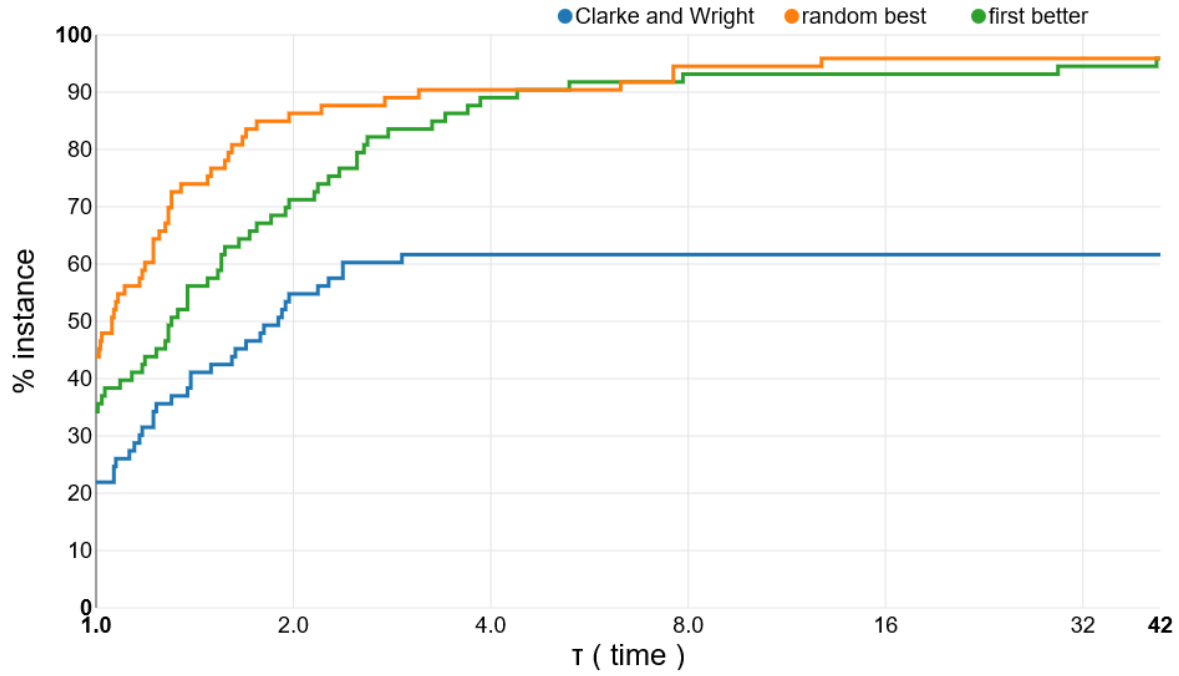

Figure 6.4: Performance profiles of first better, random best and Clarke and Wright methods.

From this figure and the comparison done in Section 6.1.2, we can see that with the iterated minimum method, the random best performs better than the first better heuristic. Note also that the scale of the x axis is logarithmic in order to display the whole range of values without shrinking the most interesting part of the graph which is the beginning.

As expected the Clarke and Wright method is far less efficient with only 60% of the instances solved. However, as you can see in Sections 6.2.2 and 6.2.6. This method is very efficient on clustered instances and works very well combined with a tabu search.

6.2.2 Tabu search

In this subsection, we compare each method with and without the TabuCost search (explained in Section 5.8.1. In practice, the TabuCost generates an overhead due to the management of the tabu store and the additional filtering of the neighbourhood. However, it may provide better results due to the additional diversification.

These tests have been done on the whole Augerat set, described in Section 6.1.4 with 4 runs of 15 minutes per instance.

First Better with Tabu

You can see in Figure 6.5 the comparison of the first better approach with and without tabu search. The original first better approach has been selected as baseline.

We observe that if for around 33% of the instances the tabu allows a small gain of time, in order to solve as much instances as the basic version, it requires up to six times more time. Interestingly, we can see that the curve of the tabu version starts at around 1%. This is due to the fact that for some instances, the tabu version is able to provide a feasible solution while the basic version is not able to solve them. In this case, the ratio considered is 0 since it corresponds to the time taken by tabu divided by infinity which tends to 0. We can also see that, since the curve of tabu does not go above the one of the basic approach, for around the same amount of instances the tabu approach is unable to find a solution while the original approach is.

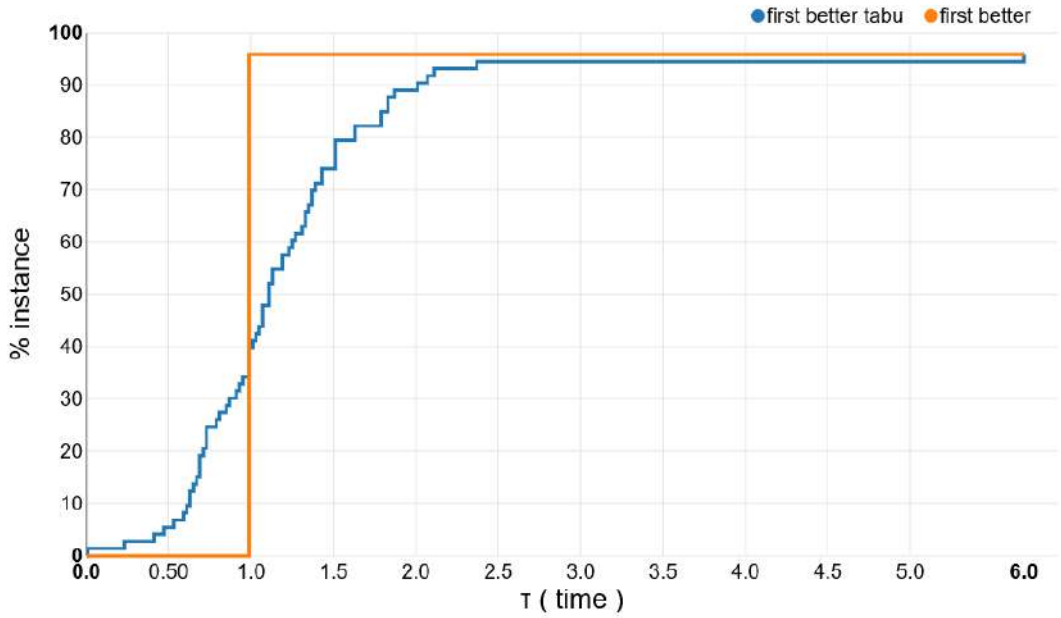

Figure 6.5: Performance profiles of the first better heuristic with and without tabu.

We can conclude that globally, there is no gain to use the TabuCost approach. Both are able to solve roughly the same number of instances but on average the approach is faster without tabu. We explain this by the fact that the first better heuristic is natively fast and already good in terms of diversification. The overhead caused by the tabu approach does not balance the additional gain of diversification.

Random Best with Tabu

In Figure 6.6, you can see the comparison of the tabu approach with the random best heuristic. Again, the original approach he baseline

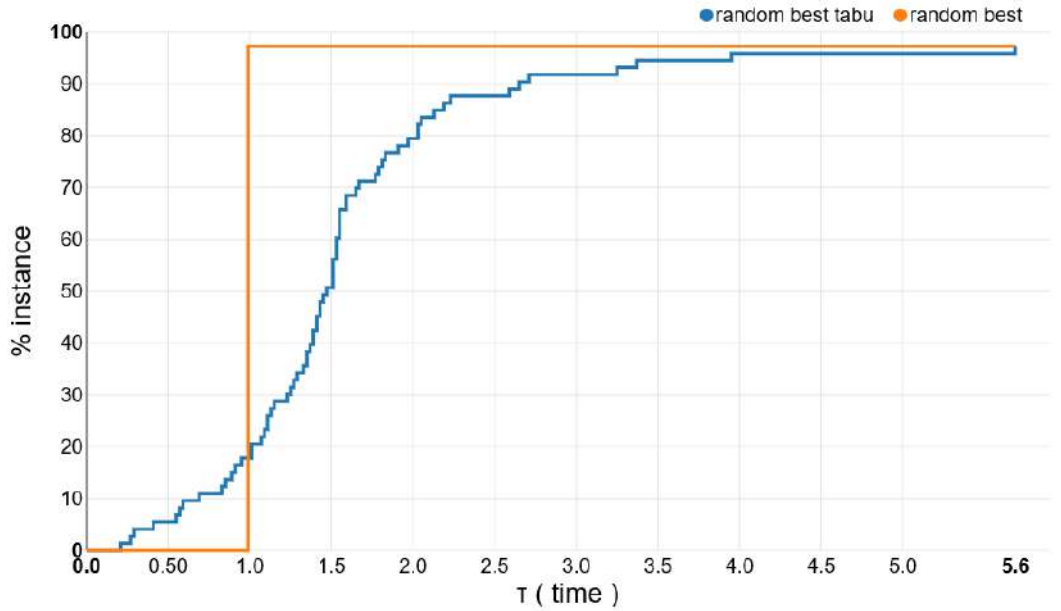

Figure 6.6: Performance profiles of the random best heuristic with and without tabu.

The conclusion is the same as for the first better heuristic: The tabu approach doesn't improve

the performances. Our hypothesis to explain this is that on the VRP, the intensification of this method is good enough to find near-optimal solutions without needing a lot of diversification.

Clark and Wright with Tabu

In Figure 6.7, You can see the gain of performance introduced by the tabu search with the Clarke and Wright method.

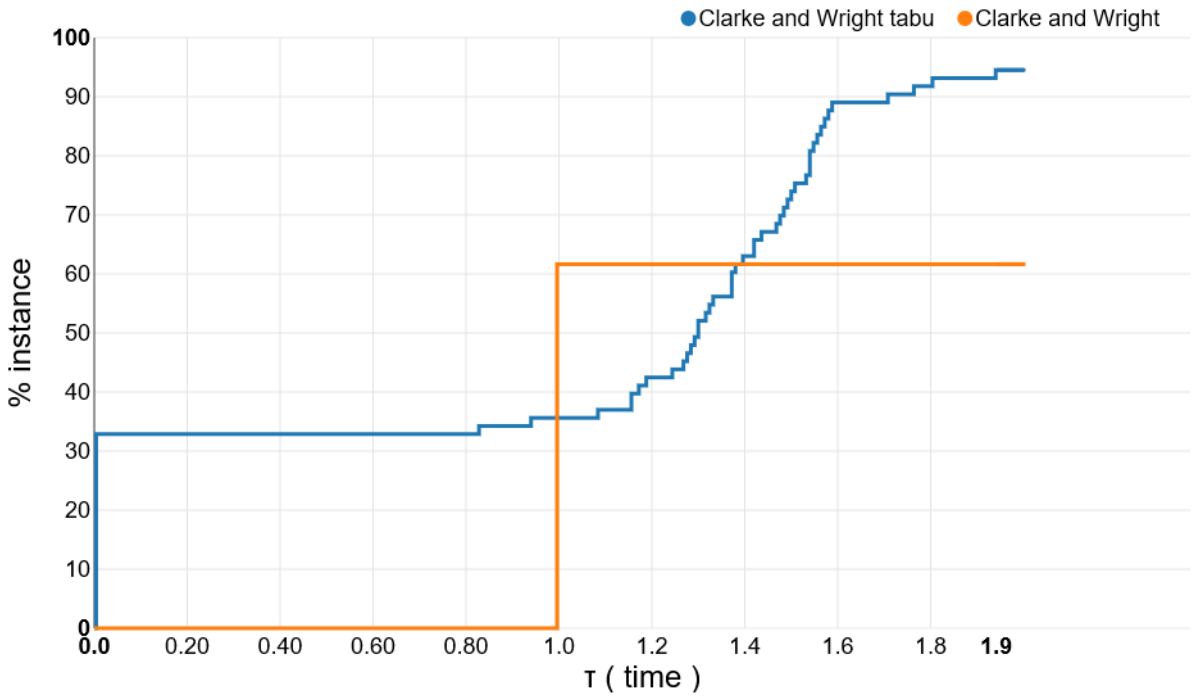

Figure 6.7: Performance profiles of the Clarke and Wright method with and without tabu.

This graph can be a bit tricky to interpret. At a ratio of 0, we can see that the curve of the tabu version of our method is at around 33%. That means that, for a third of the instances, the version with tabu was able to find a solution while the basic version timed out. Then, the evolutions of the curve correspond to the time needed to solve the instances that both version where able to solve.

For example, at a time ratio of 1.3, the tabu curve has a value of 50%. This means that, given 1.3 more time than the basic version, the tabu version is considered to be able to solve around 50% of the instances. But, bear in mind that our scale is expressed in time ratios. So, among these 50% of instances there are the 33% where we consider that the basic version takes an infinite amount of time since it is unable to solve them. Thus, there is only 17% of instances where we are actually sure to solve them given 1.3 more time than the basic version.

As the curves progresses, we can see that the tabu version requires more time but is ultimately able to solve around 94% of the instances with maximum twice the time taken by the basic version on the instances that both are able to solve.

We explain this huge gain of performances in terms of found solutions by the fact that this method has the most to gain from tabu. Indeed, the other approaches tested start with a random initial solution but this method always starts from the same solution which is having one road per customer. In its basic version, the only diversification it has is due to the random selector when more than one best solution is found in the neighbourhood. This very poor diversification leads to poor performances. Tabu allows to gain in diversification and thus makes this method competitive.

6.2.3 Simulated annealing

As explained in Sections 3.4.2 and 5.8.1, the simulated annealing is based on a random walk approach with a chance to select a degrading move in order to bring diversification. This chance depends on a temperature parameter that evolves during the search following a temperature function.

We were not expecting great results from the simulated annealing as it is quite weak in terms of intensification. Indeed, as it selects randomly only one move per iteration then accepts it based on its evaluation and the temperature parameter, in the cases were there are only a few moves in a large neighbourhood that improves the solution, it has a very small chance to find them and risks to stagnate a lot.

These expectations were confirmed by our test results. On nearly all the instances tested, it failed to reach our objective ratio of 5% more than the best known solution. We did our tests on the on the CVRP sets Augerat A, B and P with 4 runs of 15 minutes per instance. We also tested a version where we combine simulated annealing and tabu search. For the temperature function, we used a basic function where the temperature parameter decreases linearly along the time allowed for the search. We do not think that the temperature function affects a lot the performances of the search as what we observed is more a lack of intensification rather than having a too big or too small chance to select a degrading move.

The results can be seen in Appendix A in Tables A.1, A.2 and A.3. We reported the results of the basic simulated annealing, the tabu version and the Iterated random best approach as a basis of comparison. For each approach, we indicated the minimal cost found among our 4 different runs, the average cost for the 4 runs computed with a geometrical mean and the ratio with the best known solution (BKS) which is itself indicated in the last column. We did not used performance profiles as we are more interested by the solution obtained at the end of the available time rather than the time taken to reach a given threshold.

We also indicated the best result between the two simulated annealing approaches in green. and the geometrical mean of the ratios at the end of the columns. You can see that in general, the basic version performs better than the tabu one which corroborates our hypothesis that the weakness of this approach is the intensification.

Finally, an interesting observation to make is that the best known solutions are rounded. As we do not round our values during the computation, there is a small margin to take into account in the computation of the best known solution ratio. We can see this on some instances with the random best approach which most likely reached the optimal solution but has an extra cost due to the accumulated decimals in the computation.

6.2.4 Comparison with the original solver

Our solver provides a modular system separated into components. This implementation had an impact on the performances of our solver. We compared our version with the original one on the three sets of Augerat CVRP instances. In Figure 6.8, you can see a comparison with the original solver as baseline and our three most successful search methods: iterated minimum with random best and first better and Clarke and Wright with tabu.

As you can see in Figure 6.8 and in Tables A.4, A.5 and A.6, our architecture induced a loss of performances. None of our methods is able individually to solve as much instances as the original solver in the allocated time and on most instances they are much slower. This difference of performances can be explained by several reasons:

The first one is that our modular implementation does not allow us to explore the neighbourhood at each iteration as efficiently as the original implementation. Indeed, in the original solver, the evaluation is done as the same time that the move is created whereas, in our implementation, we need to create objects and pass them in a stream. Due to the modular system a lot of different objects intervene in the search and have to be created or called which has a small

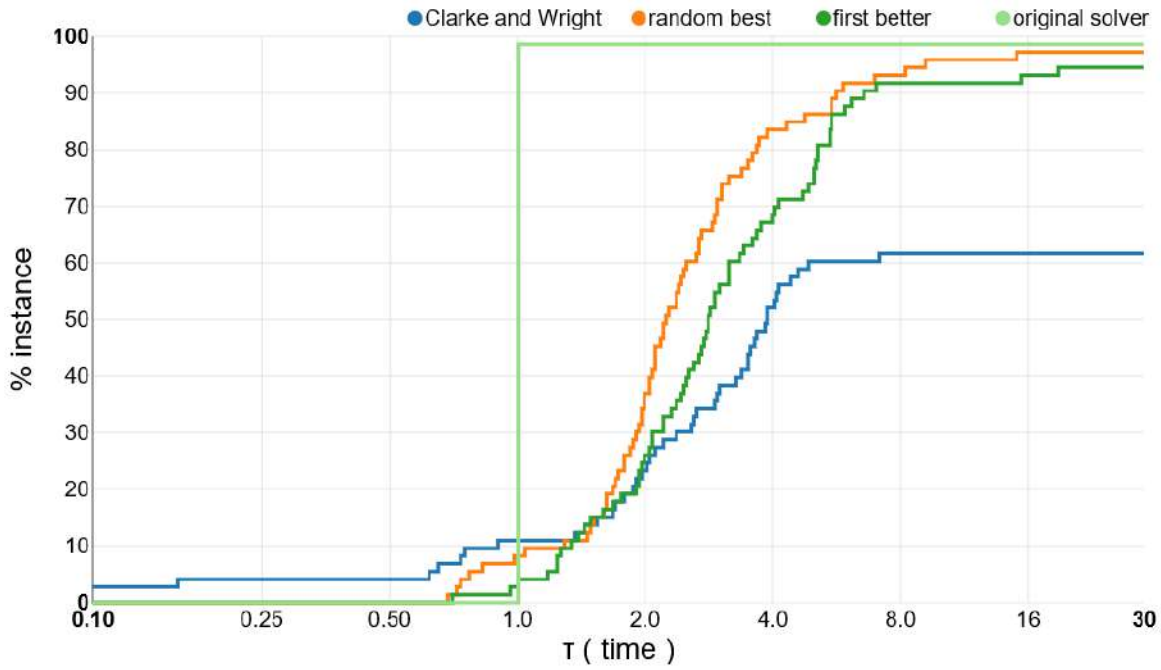

Figure 6.8: Performance profiles showing the comparison between our search methods and the original solver.

cost. However, this is not enough to explain the huge performance gap that we observe.

We suspect that the main cause behind our poor performances is the usage of streams. Initially, using the stream features from Java 8 seemed us a nice idea as it allowed us to have a flexible and lazy implementation. However, after these tests and having done researches on the subject, it seems that in practice streams in Java 8 have a huge overhead in comparison with basic loops [47,48]. To confirm this hypothesis, we ran some tests where we compare a basic loop and a stream on the iteration of an array. These tests consists in finding the maximum value inside an array of a given size filled with random integers. We ran our tests on arrays having respectively 1000, 10000, 100000, 1000000 and 10000000 entries. The results of this experiments can be seen in Figure 6.9.

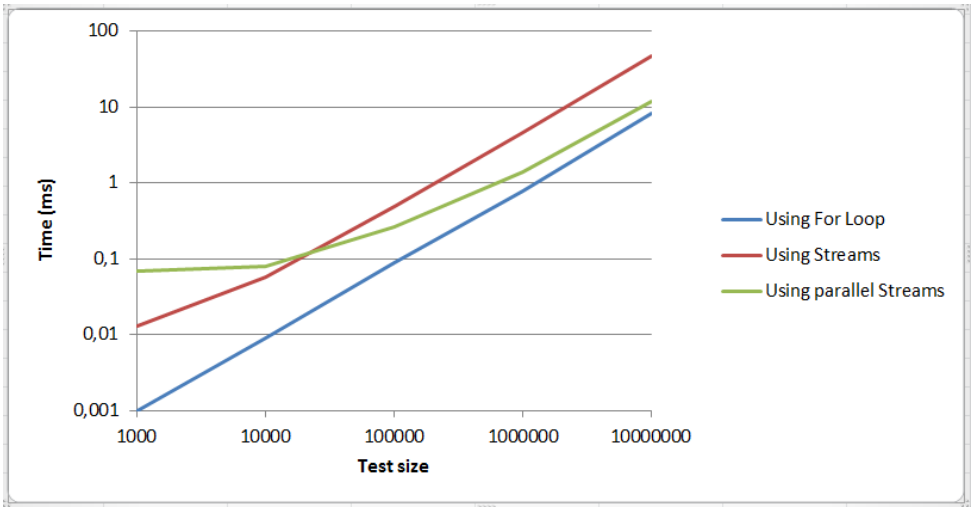

Figure 6.9: Graph showing the comparison between the usage of `For loops`, `Streams` or `Parallel Streams` in Java 8.

These results confirm our hypothesis. We can see that, on all instances, streams are slower than the basic for loop. In fact, the streams are designed to be used in a parallel way. We did not used this feature in our solver but this might drastically improve our performances and make it competitive with the original implementation. The figure shows that the usage of parallel streams improves the performances for most of the instances and are close to the performances of the loops when the size of the array grows. Note that in practical, other factors may influence this difference of performances such as the task ran during the loop or the stream or the data structure used.

Finally, a last thing to take into account is that we used relatively basic techniques in our tests. Many other resolution methods could be imagined with different combinations of our approaches. For example, we could use the Clarke and Wright method to find an initial solution that would then be improved using one of our other methods. Besides, our implementation of the simulated annealing and the tabu search. Could be improved with other techniques in the literature.

Thanks to our implementation, improving these approaches or implementing new ones should be very easy. Moreover, we are able to deal with other variants of the problem which are not supported in the original solver. The results of our tests on these different variants are presented in the next sections.

6.2.5 Distance constrained

We tested our three main methods on this variant: random best, first better and Clarke and Wright with tabu. The tests were done on the CMT set of instances with 4 different runs of 15 minutes on each instance. You can see the results in Appendix A in Table A.7.

We did not provide a performance profile graph for these results as we consider more pertinent to present a general evaluation of our performances rather than a comparison of our search methods. We can see that the performances are quite balanced between our different methods. On average, Clarke and Wright gave us the best results which is quite surprising as these instances were not clustered.

6.2.6 Time windows

We tested our three main resolution methods on the six different Solomon sets. The results can be seen in Tables A.8 to A.13. In Table 6.1, you can see an overview of the performances of each method on each set.

	Random Best	Clarke and Wright Tabu	First Better
set	**BKS ratio**	**BKS ratio**	**BKS ratio**
Solomon-c1	1,017	1,002	1,023
Solomon-c2	1,149	1,023	1,103
Solomon-r1	1,023	1,031	1,018
Solomon-r2	1,101	1,035	1,101
Solomon-rc1	1,018	1,108	1,012
Solomon-rc2	1,297	1,156	1,27
Geometric mean:	1,096	1,058	1,084

Table 6.1: Overview table of the performances of the random best, first better and the Clark and Wright tabu search approaches on the Solomon's sets.

We can see that the Clarke and Wright with tabu is mostly better. We explain that by the clustered nature of most of the test sets. However, on the `rc1` set which has both clusters and

isolated customers, it was outperformed by the other approaches.

Another interesting thing is that the performance of the random best heuristics is actually the worse among the three methods. We guess that is due to the variant which is more constrained. Indeed, as the random best approach fully explores the neighbourhood in order to find the best(s) neighbour(s), it is more impacted by additional constraints that requires more computation. This is confirmed by the performance profiles of the three approaches that we plotted in Figure 6.10. Note that a lot of instances are considered as not resolved since our algorithms failed to reach the success threshold.

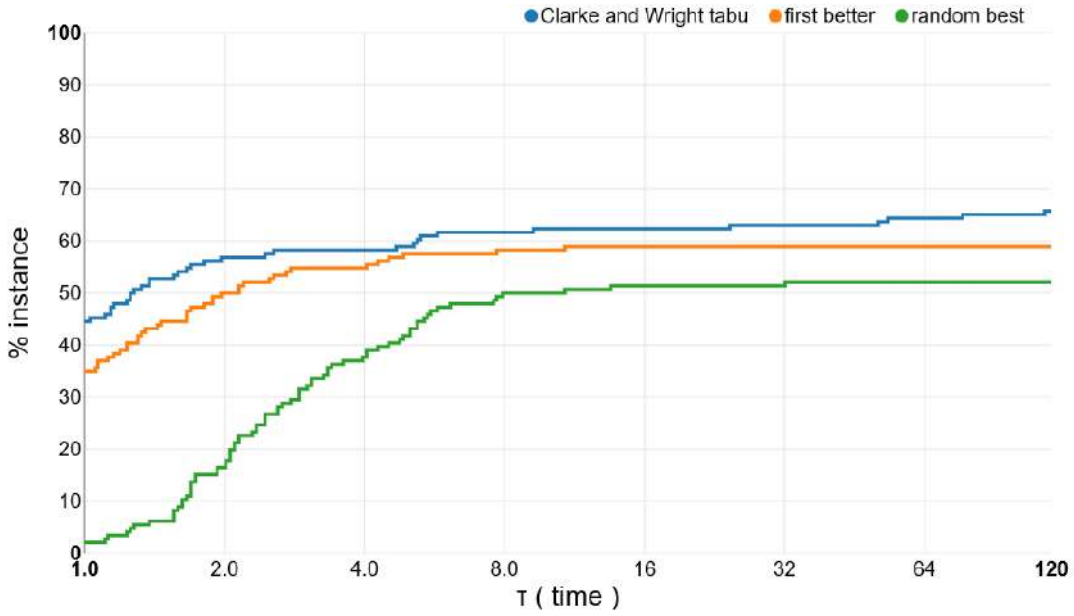

Figure 6.10: Performance profiles of our search methods on the whole Solomon set.

Finally a last interesting thing to see in the results tables is that on some instances the cost of the best solution found is a bit smaller than the one of the best known solution. We think that these solutions are actually the same but that the cost was computed differently due to other rounding rules or another language. A small error margin is thus present in our results.

6.2.7 Multiple depots

We tested our three main methods on the Cordeau set of large MDVRP instances with 4 runs of 15 minutes on each instance. You can see the results in table A.14. We did not provided a performance profile as our results are not so good. Most of the solutions found are not very close from the best known solutions. On the largest instances our search methods are unable to find good solutions. This is due to the large size of these instances and the extra layer of complexity added by the multiple depots variant.

On average, it is the first better heuristic which seems to be the most efficient. It found the optimal solution for two instances.

6.2.8 Pick ups and deliveries

To test this variant, we used the instances from the Breedam set (see Section 6.1.4). We were unable to find pure VRPPD instances in the literature and thus this set has also time windows. Unfortunately, the best known solution were not published with the instances and we were unable to find them. We thus have no means of evaluating the results obtained. You can see them in Table A.15.

We can see that in terms of number of instances where the best average cost has been obtained, *Clarke and Wright* seems to be the best method followed by *first better* and finally *random best* which is the most impacted by the extra constraints.

Chapter 7

Conclusion

In this thesis, we presented our implementation of the VRPLS library. This library aims at providing a framework of tools to tackle the vehicle routing problem and its variants using local search. We were initially tasked to study the vehicle routing problem and the state of the art local search methods. Then, to extend the library in order to deal with several variants of the problem and introduce new resolution methods.

Due to the architecture of the original library that was difficult to extend in a easy way, we decided to create our own implementation which is designed to be modular and easily extensible. We implemented different heuristics, operators, and search methods that can be combined to provide various approaches at a given problem. The local search engine of our solver is also designed to be generic and can be used to tackle other problems. We provided support for six different variants of the problem and extending the library to support new ones should be easy.

Some key features of our implementation are the distinction between the generic solver part and the problem specific components, the division into separated modules that can be modified or extended independently and the usage of streams to represent the neighbourhood.

We tested our implementation on various sets of well known instances for each variant of the problem. We also compared our implementation with the original one. Unfortunately, our implementation has a loss of performances in regards to the original implementation. Our hypothesis is that it is mainly due to the Java 8 stream implementation that is not designed to be efficient in a sequential way.

We consider that we have attained our objectives for this master thesis. Even though there is still a lot of possible improvements for the library and our results could be better compared to the original library, we are satisfied with the modular system that we created. It should now be really easy to continue to develop the library by adding new supported variants or resolution techniques. In Section 7.1, we detail all the possible improvements that could be done to the solver.

Most of all, we are thankful for all the knowledge that this thesis brought us. During our work on this library, we learned a lot on diverse subjects such as the vehicle routing problem, the optimisation techniques and in particular the local search, or the features of the java programming language.

7.1 Further work

Stream system and performances : A first improvement on the solver would be to use a parallel stream system or revert back to classical loops to explore the neighbourhood. Hopefully this would improve the performances of the solver and make it competitive with the original implementation. Many other components of the solver could be optimised. For example, we could try to find a more efficient way to select random elements among the neighbourhood.

Operators : Other operators could be added to the library in order to diversify the kind of moves considered or to support other variants of the problem. A nice addition to our operator system would be the Lin-Kernighan operator [49] which is a really effective method to generate moves.

Evaluation functions : Many other evaluation functions could be considered. Among them, relaxed versions of our existing methods that would improve the performances or evaluation functions applied only to a part of the customers.

Search strategies and heuristics : Many search methods for the VRP are detailed in the literature. Thanks to our modular architecture, it should not be difficult to implement new and more efficient approaches to solve the problem.

Problem variants : Besides the six variants explored in this thesis, many other variants exists. Our solver could be extended to support advanced variants such as the periodic VRP, the stochastic VRP or the multi-commodity VRP.

Other optimisation problems : Finally, our local search engine is designed to be modular and generic. It could be used to tackle other optimisation problems. For example, our solver could be adapted to solve the Travelling Salesman Problem. Many parts of the VRP specific part could also be reused for this problem.

Appendices

Appendix A

Results tables

In this annex, you can find the results tables of our tests. The instance is always indicated in the left-most column. If known, the best known solution (BKS) can be found in the right-most column. For each technique tested, we reported the average cost found at the end of our multiple runs along the minimal cost when pertinent or the ratio with the best known solution if applicable. We used colours to highlight some results: Green indicates the best result between the compared techniques and red indicates that no feasible solution has been found.

	Simulated Annealing			Simulated Annealing with tabu			Random Best			
instance	min cost	average cost	BKS ratio	min cost	average cost	BKS ratio	min cost	average cost	BKS ratio	BKS
A-n32-k05	787,08	854,658	1,09	787,08	827,003	1,055	787,08	787,08	1,004	784
A-n33-k05	675,16	692,325	1,047	684,9	703,312	1,064	662,11	662,11	1,002	661
A-n33-k06	742,69	766,613	1,033	744,26	772,231	1,041	742,69	742,69	1,001	742
A-n34-k05	792,92	813,98	1,046	780,94	794,392	1,021	780,94	780,94	1,004	778
A-n36-k05	827,83	844,104	1,056	822,26	836,074	1,046	802,13	802,13	1,004	799
A-n37-k05	693,27	708,614	1,059	672,47	681,742	1,019	672,47	672,47	1,005	669
A-n37-k06	976,81	1018,542	1,073	976,31	989,072	1,042	950,85	950,85	1,002	949
A-n38-k05	739,98	757,751	1,038	761,6	803,017	1,1	733,95	734,007	1,005	730
A-n39-k05	834,49	858,769	1,045	857,61	887,238	1,079	828,99	829,355	1,009	822
A-n39-k06	835,44	852,112	1,025	835,25	859,177	1,034	833,2	833,712	1,003	831
A-n44-k06	986,23	996,352	1,063	997,09	1046,839	1,117	938,18	940,263	1,003	937
A-n45-k06	1041,8	1129,619	1,197	1101,91	1278,576	1,354	956,5	967,277	1,025	944
A-n45-k07	1209,72	1257,439	1,097	1208,42	1237,988	1,08	1146,91	1149,505	1,003	1146
A-n46-k07	955,15	967,203	1,058	968	991,354	1,085	917,72	917,767	1,004	914
A-n48-k07	1105,69	1122,658	1,046	1154,95	1172,362	1,093	1074,34	1074,34	1,001	1073
A-n53-k07	1050,26	1104,258	1,093	1127,44	1172,52	1,161	1020,15	1020,56	1,01	1010
A-n54-k07	1232,7	1250,08	1,071	1315,35	1350,841	1,158	1171,78	1181,361	1,012	1167
A-n55-k09	1103,98	1113,579	1,038	1145,36	1231,078	1,147	1074,46	1074,71	1,002	1073
A-n60-k09	1447,57	1463,412	1,081	1530,37	1582,597	1,169	1360,59	1365,869	1,009	1354
A-n61-k09	1198,88	1214,225	1,174	1464,25	1499,323	1,45	1051,88	1062,86	1,028	1034
A-n62-k08	1378,31	1384,178	1,075	1532,9	1612,05	1,252	1314,33	1318,18	1,023	1288
A-n63-k09	1714,76	1753,612	1,085	1882,95	1974,436	1,222	1638,77	1645,093	1,018	1616
A-n63-k10	1381,76	1413,498	1,076	1509,29	1554,375	1,183	1324,49	1328,763	1,011	1314
A-n64-k09	1478,32	1499,096	1,07	1541,56	1630,427	1,164	1427,94	1432,896	1,023	1401
A-n65-k09	1265,82	1325,963	1,129	1573,26	1654,565	1,409	1189,76	1195,943	1,019	1174
A-n69-k09	1249,42	1271,615	1,097	1374,69	1466,224	1,265	1174,58	1178,361	1,017	1159
A-n80-k10	1884,4	1950,335	1,106	2283,45	2370,166	1,344	1823,62	1826,435	1,036	1763
Geometric mean:			1,076			1,148			1,010	1

Table A.1: Comparison of the simulated annealing with and without tabu and the random best heuristic on the Augerat-A set.

	Simulated Annealing			Simulated Annealing with tabu			Random Best			
instance	min cost	average cost	BKS ratio	min cost	average cost	BKS ratio	min cost	average cost	BKS ratio	BKS
B-n31-k05	686,99	692,433	1,03	681,69	685,876	1,021	676,09	676,09	1,006	672
B-n34-k05	792,45	799,967	1,015	795,59	801,738	1,017	789,84	789,84	1,002	788
B-n35-k05	978,51	980,709	1,027	958,95	972,922	1,019	956,29	956,29	1,001	955
B-n38-k06	808,7	824,509	1,024	808,7	813,559	1,011	807,88	808,495	1,004	805
B-n39-k05	561,06	577,989	1,053	555	577,397	1,052	553,16	553,16	1,008	549
B-n41-k06	847,67	855,72	1,032	844,69	874,711	1,055	833,81	834,4	1,007	829
B-n43-k06	749,69	759,918	1,024	754,25	758,122	1,022	746,98	746,98	1,007	742
B-n44-k07	935,29	978,267	1,076	932,09	971,916	1,069	915,02	919,391	1,011	909
B-n45-k05	768,41	804,487	1,071	793,1	840,866	1,12	756,19	759,672	1,012	751
B-n45-k06	753,22	765,461	1,129	752,24	786,891	1,161	691,55	694,405	1,024	678
B-n50-k07	744,34	748,368	1,01	754,12	756,853	1,021	744,23	744,23	1,004	741
B-n50-k08	1351,32	1354,573	1,032	1353,22	1356,892	1,034	1320,89	1326,353	1,011	1312
B-n51-k07	1036,77	1081,669	1,063	1146,19	1245,162	1,223	1038,58	1040,226	1,022	1018
B-n52-k07	756,75	759,478	1,017	760,78	789,883	1,057	750,04	750,759	1,005	747
B-n56-k07	720,48	740,134	1,047	731,87	773,772	1,094	714,29	717,196	1,014	707
B-n57-k07	1372,78	1484,237	1,297	1509,52	1629,955	1,425	1278,49	1300,014	1,136	1144
B-n57-k09	1613,73	1652,453	1,034	1696,95	1736,855	1,087	1611,7	1618,05	1,013	1598
B-n63-k10	1574,28	1584,082	1,059	1670,07	1740,034	1,163	1513,35	1532,812	1,025	1496
B-n64-k09	933,62	964,29	1,12	1211,76	1246,713	1,448	875,27	894,389	1,039	861
B-n66-k09	1354,35	1375,702	1,045	1475,53	1579,143	1,2	1330,89	1339,915	1,018	1316
B-n67-k10	1111,71	1125,358	1,09	1161,49	1217,648	1,18	1060,27	1074,257	1,041	1032
B-n68-k09	1334,85	1363,665	1,072	1437,4	1513,548	1,19	1299,58	1303,581	1,025	1272
B-n78-k10	1317,75	1338,173	1,096	1596,59	1637,847	1,341	1262,12	1280,254	1,049	1221
Geometric mean:			1,062			1,124			1,021	1

Table A.2: Comparison of the simulated annealing with and without tabu and the random best heuristic on the Augerat-B set.

	Simulated Annealing			Simulated Annealing with tabu			Random Best			
instance	min cost	average cost	BKS ratio	min cost	average cost	BKS ratio	min cost	average cost	BKS ratio	BKS
P-n016-k08	451,95	457,242	1,016	451,95	453,335	1,007	451,34	451,34	1,003	450
P-n019-k02	234,25	248,498	1,172	223,38	230,954	1,089	212,66	212,66	1,003	212
P-n020-k02	217,42	224,114	1,038	217,42	225,799	1,045	217,42	217,42	1,007	216
P-n021-k02	212,71	216,904	1,028	212,71	212,71	1,008	212,71	212,71	1,008	211
P-n022-k02	217,85	217,85	1,009	217,85	217,85	1,009	217,85	217,85	1,009	216
P-n022-k08	600,83	623,413	1,057	600,83	625,278	1,06	600,83	600,83	1,018	590
P-n023-k08	561,98	578,157	1,093	561,65	578,634	1,094	531,17	531,17	1,004	529
P-n040-k05	469,99	474,499	1,036	462,93	484,52	1,058	461,73	461,73	1,008	458
P-n045-k05	513,42	526,71	1,033	519,72	533,257	1,046	512,79	512,79	1,005	510
P-n050-k07	569,66	586,889	1,059	580,05	616,86	1,113	560,15	560,56	1,012	554
P-n050-k08	700,58	735,487	1,169	826,81	846,281	1,345	645,99	654,788	1,041	629
P-n050-k10	733,2	740,976	1,065	743,42	766,877	1,102	701,22	702,91	1,01	696
P-n051-k10	830,7	842,135	1,136	837,9	867,925	1,171	751,02	754,726	1,019	741
P-n055-k07	579,91	590,445	1,04	589,8	622,513	1,096	570,27	574,544	1,012	568
P-n055-k10	723,58	732,632	1,056	736,43	748,141	1,078	700,99	701,52	1,011	694
P-n055-k15	1161,35	1182,047	1,251	1173,72	1210,786	1,281	1027,87	1035,049	1,095	945
P-n060-k10	782,93	798,143	1,073	818,64	835,186	1,123	748,07	750,549	1,009	744
P-n060-k15	1009,61	1015,894	1,049	1075,96	1078,833	1,114	976,93	981,914	1,014	968
P-n065-k10	827,33	844,583	1,066	878,23	932,806	1,178	800,81	804,731	1,016	792
P-n070-k10	893,09	946,674	1,145	1045,73	1057,681	1,279	857,17	859,223	1,039	827
P-n076-k04	709,49	747	1,26	992,73	1045,802	1,764	613,43	613,86	1,035	593
P-n076-k05	707,01	772,74	1,232	1033,01	1072,296	1,71	646,05	648,036	1,034	627
P-n101-k04	894,75	944,153	1,386	1568,71	1622,778	2,383	698,29	701,275	1,03	681
Geometric mean:			1,104			1,194			1,019	1

Table A.3: Comparison of the simulated annealing with and without tabu and the random best heuristic on the Augerat-P set.

	First Better		Random Best		Clark and Wright tabu		Original Solver		
instance	average time	BKS ratio	average time	BKS ratio	average time	BKS ratio	average time	BKS ratio	BKS
A-n32-k05	3,163	1,004	2,022	1,004	77,918	1,004	0,978	1,004	784
A-n33-k05	1,172	1,002	2,099	1,002	18,578	1,002	0,729	1,002	661
A-n33-k06	2,314	1,001	1,573	1,001	3,27	1,001	1,151	1,001	742
A-n34-k05	1,645	1,004	1,925	1,004	4,901	1,012	0,943	1,004	778
A-n36-k05	2,998	1,005	3,63	1,004	3,808	1,004	1,161	1,004	799
A-n37-k05	2,616	1,005	2,975	1,005	4,459	1,005	1,267	1,005	669
A-n37-k06	2,877	1,002	3,148	1,002	3,958	1,002	1,242	1,002	949
A-n38-k05	3,683	1,005	4,508	1,005	4,104	1,007	1,153	1,005	730
A-n39-k05	3,083	1,009	4,799	1,009	19,398	1,01	1,333	1,009	822
A-n39-k06	2,948	1,003	2,29	1,003	4,372	1,005	1,36	1,003	831
A-n44-k06	17,818	1,007	7,815	1,003	5,547	1,005	3,509	1,001	937
A-n45-k06	233,832	1,029	101,647	1,025	6,102	1,001	8,222	1,009	944
A-n45-k07	8,001	1,003	7,017	1,003	5,868	1,006	1,916	1,001	1146
A-n46-k07	7,56	1,004	4,065	1,004	7,058	1,005	2,523	1,004	914
A-n48-k07	13,296	1,009	7,502	1,001	7,489	1,024	2,481	1,001	1073
A-n53-k07	7,58	1,01	20,095	1,01	11,319	1,019	3,822	1,01	1010
A-n54-k07	19,448	1,016	24,101	1,012	9,146	1,007	5,131	1,004	1167
A-n55-k09	5,423	1,001	8,861	1,002	8,136	1,002	4,22	1,001	1073
A-n60-k09	45,264	1,016	21,362	1,009	10,415	1,011	3,635	1,005	1354
A-n61-k09		1,034	263,563	1,028	101,6	1,012	111,645	1,027	1034
A-n62-k08	62	1,029	40,131	1,023	12,788	1,03	11,099	1,016	1288
A-n63-k09	88,717	1,031	51,667	1,018	12,193	1,016	15,084	1,015	1616
A-n63-k10	23,24	1,011	7,159	1,011	11,659	1,006	7,373	1,006	1314
A-n64-k09	55,766	1,031	24,146	1,023	50,37	1,01	12,817	1,014	1401
A-n65-k09	40,467	1,028	35,834	1,019	197,294	1,017	11,74	1,015	1174
A-n69-k09	24,515	1,021	11,142	1,017	16,524	1,012	16,705	1,013	1159
A-n80-k10		1,045	77,922	1,036	24,216	1,034	23,034	1,021	1763
Geometric mean:		1,013		1,010		1,010		1,007	1

Table A.4: Comparison of our search methods with the original solver on the Augerat-A set.

	First Better		Random Best		Clark and Wright tabu		Original Solver		
instance	average time	BKS ratio	average time	BKS ratio	average time	BKS ratio	average time	BKS ratio	BKS
B-n31-k05	1,169	1,006	1,589	1,006	2,561	1,006	0,464	1,006	784
B-n34-k05	1,717	1,002	1,744	1,002	4,544	1,002	0,631	1,002	661
B-n35-k05	1,372	1,001	1,87	1,001	4,375	1,001	0,777	1,001	742
B-n38-k06	2,083	1,004	1,936	1,004	4,511	1,005	0,932	1,004	778
B-n39-k05	3,832	1,008	3,071	1,008	5,096	1,008	0,866	1,008	799
B-n41-k06	3,349	1,006	3,15	1,006	46,539	1,048	1,756	1,006	669
B-n43-k06	3,248	1,007	2,866	1,007	5,997	1,007	1,171	1,007	949
B-n44-k07	5,156	1,007	4,281	1,007	5,311	1,007	2,244	1,007	730
B-n45-k05	6,254	1,004	3,298	1,005	5,985	1,004	1,782	1,004	822
B-n45-k06	41,369	1,019	51,177	1,006		1,058	4,487	1,004	831
B-n50-k07	4,198	1,004	2,859	1,004	7,61	1,004	1,493	1,004	937
B-n50-k08	4,04	1,007	2,894	1,006	7,926	1,016	1,456	1,005	944
B-n51-k07	10,659	1,019	7,614	1,017	13,734	1,002	2,381	1,017	1146
B-n52-k07	5,582	1,004	3,066	1,004	8,505	1,005	1,598	1,004	914
B-n56-k07	7,351	1,009	5,646	1,011	10,482	1,021	2,536	1,009	1073
B-n57-k07		1,097		1,057	14,464	0,999	71,2	1,018	1010
B-n57-k09	7,575	1,009	6,124	1,007	10,22	1,019	2,188	1,004	1167
B-n63-k10	39,316	1,023	14,963	1,013	13,86	1,03	7,198	1,008	1073
B-n64-k09	201,546	1,024	76,433	1,024		1,062	20,536	1,015	1354
B-n66-k09	8,056	1,016	7,117	1,013	184,512	1,034	4,131	1,011	1034
B-n67-k10	54,367	1,022	63,839	1,029		1,049	25,337	1,017	1288
B-n68-k09	14,204	1,023	10,057	1,021	15,619	1,025	4,815	1,017	1616
B-n78-k10	253,094	1,038	203,048	1,032	21,879	1,03	26,879	1,019	1314
A-n64-k09	55,766	1,031	24,146	1,023	50,37	1,01	12,817	1,014	1401
A-n65-k09	40,467	1,028	35,834	1,019	197,294	1,017	11,74	1,015	1174
A-n80-k10		1,045	77,922	1,036	24,216	1,034	23,034	1,021	1763
Geometric mean:		1,018		1,014		1,019		1,009	1

Table A.5: Comparison of our search methods with the original solver on the Augerat-B set.

	First Better		Random Best		Clark and Wright tabu		Original Solver		
instance	average time	BKS ratio	average time	BKS ratio	average time	BKS ratio	average time	BKS ratio	BKS
P-n016-k08	0,197	1,003	0,254	1,003	2,947	1,003	0,202	1,003	784
P-n019-k02	0,773	1,003	1,325	1,003	7,141	1,003	0,253	1,003	661
P-n020-k02	0,974	1,007	1,03	1,007	14,456	1,007	0,329	1,007	742
P-n021-k02	0,49	1,008	0,72	1,008	13,078	1,008	0,314	1,008	778
P-n022-k02	1,019	1,009	0,761	1,009	7,898	1,009	0,301	1,009	799
P-n022-k08	0,794	1,018	0,74	1,018	1,44	0,998	0,637	1,018	669
P-n023-k08	1,556	1,004	1,283	1,004	1,582	1,004	0,678	1,004	949
P-n040-k05	2,865	1,008	3,868	1,008	57,883	1,023	1,203	1,008	730
P-n045-k05	4,295	1,005	3,149	1,005	18,939	1,008	1,479	1,005	822
P-n050-k07	6,23	1,011	6,654	1,012	89,913	1,017	4,407	1,011	831
P-n050-k08		1,05		1,041	10,442	1,016	138,533	1,036	937
P-n050-k10	2,879	1,007		1,01	7,369	1,016	5,914	1,011	944
P-n051-k10	36,756	1,029	29,385	1,019	13,885	1,02	10,454	1,009	1146
P-n055-k07	4,736	1,013	6,995	1,012	92,682	1,029	2,092	1,008	914
P-n055-k10	4,861	1,009	2,956	1,011	17,313	1,008	3,301	1,01	1073
P-n055-k15		1,097		1,095	8,36	1,012		1,084	1010
P-n060-k10	8,658	1,012	7,796	1,009	83,65	1,035	9,362	1,014	1167
P-n060-k15	5,715	1,012	7,272	1,014	10,406	1,017	8,154	1,01	1073
P-n065-k10	16,982	1,017	11,625	1,016	66,765	1,026	7,483	1,018	1354
P-n070-k10	124,394	1,037	176,846	1,039	241,266	1,039	70,787	1,033	1034
P-n076-k04	334,699	1,038	220,855	1,035		1,074	59,044	1,022	1288
P-n076-k05	41,179	1,033	136,531	1,034	148,596	1,049	44,273	1,015	1616
P-n101-k04	82,339	1,033	174,153	1,03		1,105	59,264	1,029	1314
Geometric mean:		1,020		1,019		1,023		1,016	1

Table A.6: Comparison of our search methods with the original solver on the Augerat-P set.

	Random Best		Clarke and Wright Tabu		First Better		
instance	average cost	BKS ratio	average cost	BKS ratio	average cost	BKS ratio	BKS
CMT01	524,61	1	539,326	1,028	531,523	1,013	524,61
CMT02	867,991	1,039	865,807	1,037	864,555	1,035	835,26
CMT03	838,729	1,015	884,76	1,071	845,474	1,023	826,14
CMT04	1074,095	1,044	1089,64	1,06	1093,46	1,063	1028,42
CMT05	1605,26	1,243	1375,63	1,065	1647,498	1,276	1291,29
CMT06	557,49	1,004	562,35	1,012	556,929	1,003	555,43
CMT07	935,426	1,028	951,061	1,045	929,949	1,022	909,68
CMT08	880,72	1,017	905,947	1,046	883,962	1,021	865,94
CMT09	1226,642	1,055	1242,191	1,069	1235,125	1,062	1162,55
CMT10	1499,901	1,075	1537,35	1,101	1503,376	1,077	1395,85
CMT11	1220,864	1,172	1046,03	1,004	1241,974	1,192	1042,11
CMT12	847,033	1,034	821,11	1,002	851,492	1,039	819,56
CMT13	1654,005	1,073	1561,097	1,013	1667,04	1,082	1541,14
CMT14	915,345	1,057	867,17	1,001	928,175	1,071	866,37
Geometric mean:		1,059		1,039		1,068	

Table A.7: Comparison of the search methods on the Christofides, Mingozzi and Toth (CMT) set.

	Random Best		Clarke and Wright Tabu		First Better		
instance	average cost	BKS ratio	average cost	BKS ratio	average cost	BKS ratio	BKS
C101_025	191,81	1,003	191,81	1,003	191,81	1,003	191,3
C101_050	363,25	1,002	363,25	1,002	363,25	1,002	362,4
C101_100	828,94	1,002	828,94	1,002	828,94	1,002	827,3
C102_025	190,74	1,002	190,74	1,002	190,74	1,002	190,3
C102_050	362,17	1,002	362,17	1,002	362,17	1,002	361,4
C102_100	886,477	1,072	828,94	1,002	927,345	1,121	827,3
C103_025	190,74	1,002	190,74	1,002	190,74	1,002	190,3
C103_050	362,17	1,002	362,17	1,002	362,17	1,002	361,4
C103_100	893,551	1,081	828,06	1,002	894,473	1,083	826,3
C104_025	187,45	1,003	187,45	1,003	187,45	1,003	186,9
C104_050	359,7	1,005	358,88	1,002	358,88	1,002	358
C104_100	880,081	1,069	826,755	1,005	915,209	1,112	822,9
C105_025	191,81	1,003	191,81	1,003	191,81	1,003	191,3
C105_050	363,25	1,002	363,25	1,002	363,25	1,002	362,4
C105_100	828,94	1,002	828,94	1,002	856,839	1,036	827,3
C106_025	191,81	1,003	191,81	1,003	191,81	1,003	191,3
C106_050	363,25	1,002	363,25	1,002	363,25	1,002	362,4
C106_100	929,036	1,123	828,94	1,002	860,972	1,041	827,3
C107_025	191,81	1,003	191,81	1,003	191,81	1,003	191,3
C107_050	363,25	1,002	363,25	1,002	363,25	1,002	362,4
C107_100	846,666	1,023	828,94	1,002	871,98	1,054	827,3
C108_025	191,81	1,003	191,81	1,003	191,81	1,003	191,3
C108_050	363,25	1,002	363,25	1,002	363,25	1,002	362,4
C108_100	855,399	1,034	828,94	1,002	872,718	1,055	827,3
C109_025	191,81	1,003	191,81	1,003	191,81	1,003	191,3
C109_050	363,25	1,002	363,25	1,002	363,25	1,002	362,4
C109_100	843,531	1,02	828,94	1,002	893,4	1,08	827,3
Geometric mean:		1,017		1,002		1,023	

Table A.8: Comparison of the search methods on the Solomon-c1 set.

	Random Best		Clarke and Wright Tabu		First Better		
instance	average cost	BKS ratio	average cost	BKS ratio	average cost	BKS ratio	BKS
C201_025	215,54	1,004	215,54	1,004	215,54	1,004	214,7
C201_050	444,96	1,235	361,8	1,004	444,96	1,235	360,2
C201_100	598,202	1,015	591,56	1,004	591,56	1,004	589,1
C202_025	223,31	1,04	215,54	1,004	223,31	1,04	214,7
C202_050	403,81	1,121	361,8	1,004	403,81	1,121	360,2
C202_100	679,444	1,153	632,36	1,073	591,56	1,004	589,1
C203_025	223,31	1,04	215,54	1,004	223,31	1,04	214,7
C203_050	402,52	1,119	361,41	1,004	402,52	1,119	359,8
C203_100	799,605	1,358	634,23	1,077	720,753	1,224	588,7
C204_025	213,93	1,004	213,93	1,004	213,93	1,004	213,1
C204_050	362,761	1,036	351,72	1,005	354,215	1,012	350,1
C204_100	878,567	1,494	619,72	1,054	717,051	1,219	588,1
C205_025	297,45	1,385	215,54	1,004	297,45	1,385	214,7
C205_050	441,965	1,228	361,41	1,004	439,86	1,223	359,8
C205_100	588,88	1,004	615,52	1,05	588,88	1,004	586,4
C206_025	282,12	1,314	215,54	1,004	282,12	1,314	214,7
C206_050	426,44	1,185	361,41	1,004	421,916	1,173	359,8
C206_100	764,935	1,305	615,52	1,05	588,49	1,004	586
C207_025	274,78	1,281	215,34	1,004	274,78	1,281	214,5
C207_050	402,089	1,118	361,21	1,004	398,33	1,108	359,6
C207_100	734,196	1,253	615,31	1,05	588,29	1,004	585,8
C208_025	229,84	1,072	215,37	1,004	229,84	1,072	214,5
C208_050	352,12	1,005	352,12	1,005	352,12	1,005	350,5
C208_100	588,32	1,004	665,59	1,136	588,32	1,004	585,8
Geometric mean:		1,149		1,023		1,103	

Table A.9: Comparison of the search methods on the Solomon-c2 set.

	Random Best		Clarke and Wright Tabu		First Better		
instance	average cost	BKS ratio	average cost	BKS ratio	average cost	BKS ratio	BKS
R101_025	618,33	1,002	618,33	1,002	618,33	1,002	617
R101_050	1035,139	0,992	1034,54	0,991	1035,139	0,992	1044
R101_100			1767,423	1,079	1634,24	0,998	1637,7
R102_025	548,11	1,002	548,11	1,002	548,11	1,002	547,1
R102_050	912,862	1,004	915,979	1,008	910,957	1,002	909
R102_100			1541,428	1,051	1472,134	1,004	1466,6
R103_025	463,945	1,021	455,7	1,002	472,34	1,039	454,6
R103_050	779,788	1,009	787	1,018	777,467	1,006	772,9
R103_100	1282,142	1,061	1282,25	1,061	1247,805	1,032	1208,7
R104_025	417,96	1,003	417,96	1,003	417,96	1,003	416,9
R104_050	626,467	1,002	640,34	1,024	628,905	1,006	625,4
R104_100	1026,787	1,057	1059,114	1,09	1032,566	1,063	971,5
R105_025	537,645	1,013	531,54	1,002	534,584	1,008	530,5
R105_050	906,726	1,008	927,756	1,032	901,87	1,003	899,3
R105_100	1430,857	1,056	1513,18	1,116	1405,015	1,037	1355,3
R106_025	466,48	1,002	466,48	1,002	466,48	1,002	465,4
R106_050	789,353	0,995	810,95	1,023	792,496	0,999	793
R106_100	1301,602	1,054	1344,83	1,089	1281,24	1,038	1234,6
R107_025	425,27	1,002	425,27	1,002	425,27	1,002	424,3
R107_050	726,122	1,021	733,907	1,032	720,344	1,013	711,1
R107_100	1146,193	1,077	1156,56	1,086	1142,967	1,074	1064,6
R108_025	398,29	1,002	398,29	1,002	398,29	1,002	397,3
R108_050	620,918	1,005	633,265	1,025	619,228	1,002	617,7
R109_025	455,981	1,033	442,63	1,003	460,52	1,044	441,3
R109_050	799,652	1,016	796,45	1,012	797,272	1,013	786,8
R109_100	1251,861	1,092	1251,38	1,091	1216,626	1,061	1146,9
R110_025	430,99	0,97	430,99	0,97	430,99	0,97	444,1
R110_050	717,703	1,03	717,88	1,03	710,506	1,019	697
R110_100	1163,383	1,089	1148,11	1,075	1165,944	1,092	1068
R111_025	429,7	1,002	430,055	1,003	429,7	1,002	428,8
R111_050	713,737	1,009	723,48	1,023	723,991	1,024	707,2
R111_100	1143,092	1,09	1142,19	1,089	1105,271	1,054	1048,7
R112_025	394,1	1,003	394,1	1,003	394,1	1,003	393
R112_050	641,012	1,017	646,306	1,026	639,52	1,015	630,2
Geometric mean:		1,023		1,031		1,018	

Table A.10: Comparison of the search methods on the Solomon-r1 set.

	Random Best		Clarke and Wright Tabu		First Better		
instance	average cost	BKS ratio	average cost	BKS ratio	average cost	BKS ratio	BKS
R201_025	523,66	1,13	464,37	1,002	523,66	1,13	463,3
R201_050	937,804	1,184	850,793	1,074	934,801	1,18	791,9
R201_100			1383,053	1,21	1306,118	1,143	1143,2
R202_025	455,53	1,11	411,49	1,002	455,53	1,11	410,5
R202_050	820,197	1,174	756,52	1,083	821,833	1,177	698,5
R203_025	400,4	1,023	394,281	1,007	400,4	1,023	391,4
R203_050	667,8	1,103	636,05	1,051	666,062	1,1	605,3
R204_025	383,44	1,08	355,89	1,003	383,44	1,08	355
R204_050	512,467	1,012	510,63	1,008	512,431	1,012	506,4
R205_025	487,62	1,241	394,06	1,003	487,62	1,241	393
R205_050	761,693	1,104	750,809	1,088	757,334	1,097	690,1
R206_025	413,18	1,104	375,48	1,003	413,18	1,104	374,4
R206_050	664,157	1,05	676,444	1,07	662,235	1,047	632,4
R207_025	398,04	1,101	362,63	1,003	398,04	1,101	361,6
R208_025	329,33	1,003	329,33	1,003	329,33	1,003	328,2
R209_025	418,25	1,128	371,56	1,002	418,25	1,128	370,7
R209_050	675,964	1,125	614,355	1,023	664,594	1,107	600,6
R210_025	500,88	1,238	405,48	1,002	500,88	1,238	404,6
R210_050	682,922	1,058	708,111	1,097	679,037	1,052	645,6
R211_025	361,69	1,031	346,68	0,988	361,69	1,031	350,9
R211_050	568,172	1,061	556,487	1,039	565,91	1,057	535,5
Geometric mean:		1,101		1,035		1,101	

Table A.11: Comparison of the search methods on the Solomon-r2 set.

	Random Best		Clarke and Wright Tabu		First Better		
instance	average cost	BKS ratio	average cost	BKS ratio	average cost	BKS ratio	BKS
RC101_025	459,38	0,996	474,18	1,028	459,38	0,996	461,1
RC101_050	943,123	0,999	1044,91	1,107	934,835	0,99	944
RC101_100	1717,19	1,06	1851,29	1,143	1689,265	1,043	1619,8
RC102_025	346,96	0,986	405,609	1,153	346,96	0,986	351,8
RC102_050	797,644	0,97	959,9	1,167	788,8	0,959	822,5
RC102_100	1546,129	1,061	1711,09	1,174	1515,39	1,04	1457,4
RC103_025	333,92	1,003	333,967	1,004	333,92	1,003	332,8
RC103_050	708,099	0,996	869,21	1,223	707,957	0,996	710,9
RC103_100	1330,771	1,058	1438,12	1,143	1325,065	1,053	1258
RC104_025	307,14	1,002	307,14	1,002	307,14	1,002	306,6
RC104_050	546,51	1,001	626,548	1,148	546,51	1,001	545,8
RC104_100	1225,922	1,083	1220,41	1,078	1199,786	1,06	1132,3
RC105_025	408,41	0,993	461,5	1,122	408,41	0,993	411,3
RC105_050	855,67	1	959,79	1,122	855,67	1	855,3
RC105_100	1606,919	1,062	1695,03	1,12	1576,285	1,041	1513,7
RC106_025	346,51	1,003	398,06	1,152	346,51	1,003	345,5
RC106_050	744,924	1,03	927,49	1,282	722	0,998	723,2
RC107_025	298,95	1,002	298,95	1,002	298,95	1,002	298,3
RC107_050	632,89	0,985	781,121	1,215	632,89	0,985	642,7
RC107_100	1273,154	1,054	1348,54	1,117	1284,507	1,064	1207,8
RC108_025	294,99	1,002	294,99	1,002	294,99	1,002	294,5
RC108_050	599,17	1,002	600,23	1,004	599,17	1,002	598,1
RC108_100	1189,672	1,068	1163,23	1,044	1181,906	1,061	1114,2
Geometric mean:		1,018		1,108		1,012	

Table A.12: Comparison of the search methods on the Solomon-rc1 set.

	Random Best		Clarke and Wright Tabu		First Better		
instance	average cost	BKS ratio	average cost	BKS ratio	average cost	BKS ratio	BKS
RC201_025	432,3	1,2	414,71	1,151	432,3	1,2	360,2
RC201_050	843,056	1,232	955,87	1,397	838,595	1,225	684,4
RC201_100			1685,99	1,336	1525,743	1,209	1261,8
RC202_025	548,34	1,622	338,82	1,002	548,34	1,622	338
RC202_050	866,147	1,412	846,617	1,38	865,81	1,411	613,6
RC202_100			1326,71	1,215	1343,54	1,23	1092,3
RC203_025	431,32	1,319	327,69	1,002	431,32	1,319	326,9
RC203_050	674,816	1,215	659,24	1,187	674,165	1,214	555,3
RC204_025	327,33	1,092	300,23	1,002	327,33	1,092	299,7
RC205_025	386,15	1,142	364,747	1,079	386,15	1,142	338
RC205_050	892,324	1,416	811,104	1,287	819,449	1,3	630,2
RC205_100			1451,36	1,258	1434,601	1,243	1154
RC206_025	476,77	1,472	325,1	1,003	439,708	1,357	324
RC206_050	763,976	1,252	787,36	1,291	758,979	1,244	610
RC207_025	478,6	1,604	298,95	1,002	478,6	1,604	298,3
RC207_050	663,522	1,188	676,652	1,211	656,19	1,175	558,6
RC208_025	305,07	1,134	269,01	1	305,28	1,134	269,1
Geometric mean:		1,297		1,156		1,270	

Table A.13: Comparison of the search methods on the Solomon-rc2 set.

	Random Best		Clarke and Wright Tabu		First Better		
instance	**average cost**	**BKS ratio**	**average cost**	**BKS ratio**	**average cost**	**BKS ratio**	**BKS**
p01	631,917	1,095	581,644	1,008	636,802	1,104	576,87
p02	514,128	1,086	476,453	1,006	508,351	1,074	473,53
p03	739,358	1,153	653,289	1,019	741,63	1,157	641,19
p04	1086,901	1,085	1040,301	1,039	1108,366	1,107	1001,59
p05	787,256	1,05	779,874	1,04	783,507	1,045	750,03
p06	1000,634	1,142	907,039	1,035	1031,399	1,177	876,5
p07	988,33	1,116	922,437	1,041	1009,578	1,14	885,8
p08	5382,128	1,213	6306,361	1,421	5155,957	1,162	4437,68
p09	5058,604	1,297	7942,902	2,037	4820,036	1,236	3900,22
p10	4848,372	1,324	8187,529	2,235	4882,385	1,333	3663,02
p11	4777,72	1,344	8002,771	2,252	4779,623	1,345	3554,18
p12	1352,144	1,025	1329,348	1,008	1387,65	1,052	1318,95
p13	1320,292	1,001	1322,526	1,003	1318,95	1	1318,95
p14	1391,034	1,023	1406,753	1,034	1360,12	1	1360,12
p15	3181,787	1,27	2632,502	1,051	3208,348	1,281	2505,42
p16			2718,542	1,057	2720,933	1,058	2572,23
p17			2932,759	1,083	2875,916	1,062	2709,09
p18	5707,207	1,541	4676,64	1,263	5894,027	1,592	3702,85
p19			5156,004	1,347			3827,06
p20			5444,26	1,342			4058,07
p21	9987,87	1,824	39406,57	7,198	10346,352	1,89	5474,84
p22							5702,16
p23							6095,46
Geometric mean:		1,260		1,418		1,223	

Table A.14: Comparison of the search methods on the Cordeau (MDVRP) set.

	Random Best		Clarke and Wright Tabu		First Better	
instance	min cost	average cost	min cost	average cost	min cost	average cost
10PP	1065,42	1084,202	1077,33	1077,33	1089,11	1109,512
11PP	1256,5	1265,062	1169,93	1169,93	1267,29	1278,597
12PP	1526,26	1542,336	1565,56	1584,275	1546,31	1550,733
13PP	990,23	1004,918	1012,06	1019,804	991,08	998,968
14PP	3688,8	3699,92	3618,83	3639,099	3652,77	3668,62
15PP	1330,2	1340,895	1332,55	1332,55	1319,33	1329,094
1PP	1125,97	1137,771	1146	1148,497	1135,86	1168,945
2PP	1584,23	1589,587	1507,56	1507,56	1581,12	1594,992
3PP	1753,9	1754,672	1752,06	1752,06	1754,1	1754,66
4PP	1499,6	1513,378	1490,04	1500,289	1491,91	1505,18
5PP	995,33	1007,499	1019,26	1023,365	971,96	995,796
6PP	1010,93	1015,644	1038,2	1048,623	1000,36	1029,015
7PP	3857,51	3966,823	735,81	746,404	576,49	579,082
8PP	3064,43	3155,273	1133,26	1133,26	515,07	517,064
9PP	1860,42	1885,359	1795,28	1802,968	1882,49	1893,335

Table A.15: Comparison of the search methods on the Breedam (VRPPDTW) set.

Appendix B

User Manual

B.1 VRP solver

In order to use the solver, one has to create an application (i.e. a class that will read the problem data, send it to the search manager, launch the search an retrieve the results). We provided two different models of application in the package `vrp/appModels`.

The first one which is called `BasicApp` provides an example of how to use the solver with a custom made class for a specific problem. The other one, `CommandLineApp`, allows to use the solver in a generic way by specifying through command line arguments which parameters to use in the search. Here is how to use it:

Suppose you have to use the solver as a `.jar` file named `solver`. The syntax of the command is:

```
1    $ java -jar solver.jar <input file path> [options]
```

The variant of the problem will be detected based of the information retrieved when reading the instance file and the proper constraints will be applied. However, some variants might require some additional parameters such as specific operators. Here is an exhaustive list of the available options:

Option	**Usages**
`-reader <r>`	Specifies which reader should be used to read the input file (see Section 6.1.4). The possible values for the parameter `<r>` are: • `CMT` for the *Christofides, Mingozzi and Toth* instances. • `REP` (default) for the instances in the VRP-REP format. • `Cordeau` for the *Cordeau et al.* instances. • `Breedam` for the *Breedam* instances.
`-view`	Enables the graphical interface representing the evolution of the search in term of violation, cost and state.

`-operator <o>`	This parameter is used to specify the operators to use during the search. The supported values of `<o>` are the following: • `CW` corresponds to the Clarke and Wright operator. • `A4` includes 4 operators: the reallocate operator, the swap operator, the k-exchange operator and the cross operator. • `MDA4` aggregates the `A4` operator with the change depot operator. • `Mixed` (default) combines the `CW` operator with the *A4*. • `MDMixed` is a combination of the `Mixed` operator with the change depot operator.
`-heuristic <h>`	Used to specify the heuristic to use. The available options are: • `FirstBest` to use the first best heuristic. • `FirstBetter` to use the first better heuristic. • `RandomBest` (default) to use the random best heuristic. • `RandomBetter` to use the random better heuristic. • `Metropolis` to use the metropolis heuristic.
`-search <s>`	Used to specify the search strategy to use. The available options are: • `IteratedMin` to use the iterated minimum routes approach. • `Singleton` (default) to use the Clarke and Wright approach without restart. • `SA` to use the simulated annealing approach. • `Tabu` to use the iterated minimum routes approach with a tabu search. • `CWTabu` to use the Clarke and Wright approach with restart and with a tabu search.

`-out <output>`	Each time the solution is updated, the new solution will be printed in the specified file following the format:
	`New solution found!`
	`Violation: <violation>`
	`Cost: <cost>`
	`Time elapsed: <time>`
	`Vehicles: <`$v_0, v_1, ..., v_n$`>`
	`Routes:`
	`0: <r_0>`
	`1: <r_1>`
	`.`
	`.`
	`n: <r_n>`

`-comment <comment>`	Used to specify a String that will be printed at the beginning of the output file if applicable.
`-time <t>`	The maximum time allocated for the search.
`-iter <it>`	The maximum number of iterations in a single descent. If no restart strategy is defined, this option will limit the whole search.
`-obj <v>`	The objective threshold will stop the search if the current best solution is feasible and its cost is lower or equal to this threshold.
`-debug`	This option enable the debug mode which should be used to test new features inside the solver. At each step of the search, the consistency of the invariants and the instance is checked along with each solution found. Note that these checks have an overhead.
`-mute`	Disables the standard output. By default, each time the best solution found is updated, it is displayed on the standard output following the same format as with option `-out`.

B.2 Test scripts

We provided some Python scripts to test the performances of our solver (see Section 6.1.1):

The main script is `Main.py`. It runs specified configurations using a thread pool. The test configurations have to be manually entered in the file as a map of strings. The key corresponds the name of the directory in which all the results of the configuration will be saved and the value is the a single string containing the options that have to be used for the configuration. Once this is done, you can call the script like this:

```
$ Python Main.py -in <input> -out <output> -jar <solver> [options]
```

Here is a list of the available options:

Option	Usages
`-nruns <nr>`	The number of runs per instance to perform. By default, this value is 1.
`-nprocs <np>`	The number of threads in the thread pool. By default, this value is 1.
`-t <t>`	The time allowed for the search. 5 minutes by default.
`-best <b>`	The path to a best known values file used to stop the search when reached. By default, this value is None and is not considered.

`-reader <reader>`	The reader that will be use to read the instance. The possibles values for this option are exactly the same that these described in section B.1. `REP` is set by default.

Beware that if the directory specified as output already exists, its content will be erased. We also provided some additional scripts to aggregate compute the results:

PPMain.py Generates the timelines, the spreadsheets and the performance profiles. The directory and parameters to use have to be modified at the beginning of the file.

PPMainConcat.py Concatenates multiples performance profiles files into one.

The other scripts are tools that are used by these three main scripts.

List of Figures

List of Tables

Bibliography

[1] George B Dantzig and John H Ramser. The truck dispatching problem. *Management science*, 6(1):80–91, 1959.

[2] Jean-Francois Cordeau, Michel Gendreau, Gilbert Laporte, Jean-Yves Potvin, and François Semet. A guide to vehicle routing heuristics. *Journal of the Operational Research society*, pages 512–522, 2002.

[3] Chris Groër, Bruce Golden, and Edward Wasil. A library of local search heuristics for the vehicle routing problem. *Mathematical Programming Computation*, 2(2):79–101, 2010.

[4] Olli Bräysy and Michel Gendreau. Vehicle routing problem with time windows, part i: Route construction and local search algorithms. *Transportation science*, 39(1):104–118, 2005.

[5] Gilbert Laporte. The vehicle routing problem: An overview of exact and approximate algorithms. *European Journal of Operational Research*, 59(3):345 – 358, 1992.

[6] B.L. Golden, S. Raghavan, and E.A. Wasil. *The Vehicle Routing Problem: Latest Advances and New Challenges*. Operations Research/Computer Science Interfaces Series. Springer US, 2008.

[7] Michael R Garey and David S Johnson. *Computers and intractability*, volume 29, pages 222–223. wh freeman New York, 2002.

[8] Pascal Van Hentenryck and Laurent Michel. *Constraint-based local search*, pages xii–xii. The MIT Press, 2005.

[9] Hanoi university of sciences and technology. `http://en.hust.edu.vn/home`.

[10] D.L. Applegate, R.E. Bixby, V. Chvátal, and W.J. Cook. *The Traveling Salesman Problem: A Computational Study*, chapter 1. Princeton Series in Applied Mathematics. Princeton University Press, 2011.

[11] Christos H. Papadimitriou. The euclidean travelling salesman problem is np-complete. *Theoretical Computer Science*, 4(3):237 – 244, 1977.

[12] Burak Eksioglu, Arif Volkan Vural, and Arnold Reisman. The vehicle routing problem: A taxonomic review. *Computers & Industrial Engineering*, 57(4):1472 – 1483, 2009.

[13] N. Christofides. *Combinatorial optimization*, chapter 11. A Wiley-Interscience publication. John Wiley & Sons Canada, Limited, 1979.

[14] Chung-Lun Li, David Simchi-Levi, and Martin Desrochers. On the distance constrained vehicle routing problem. *Operations research*, 40(4):790–799, 1992.

[15] Groupe d'études et de recherche en analyse des décisions (Montréal in Québec) Cordeau, Jean-François. *The VRP with time windows*, pages 1–3. Montréal: Groupe d'études et de recherche en analyse des décisions, 2000.

[16] Jacques Renaud, Gilbert Laporte, and Fayez F Boctor. A tabu search heuristic for the multi-depot vehicle routing problem. *Computers & Operations Research*, 23(3):229–235, 1996.

[17] Hokey Min. The multiple vehicle routing problem with simultaneous delivery and pick-up points. *Transportation Research Part A: General*, 23(5):377–386, 1989.

[18] Michel Gendreau, Gilbert Laporte, Christophe Musaraganyi, and Éric D Taillard. A tabu search heuristic for the heterogeneous fleet vehicle routing problem. *Computers & Operations Research*, 26(12):1153–1173, 1999.

[19] Michel Gendreau, Gilbert Laporte, and René Séguin. Stochastic vehicle routing. *European Journal of Operational Research*, 88(1):3 – 12, 1996.

[20] Peter M. Francis, Karen R. Smilowitz, and Michal Tzur. *The Vehicle Routing Problem: Latest Advances and New Challenges*, chapter The Period Vehicle Routing Problem and its Extensions, pages 73–102. Springer US, Boston, MA, 2008.

[21] Pascal Van Hentenryck and Laurent Michel. *Constraint-based local search*, pages 9–10. The MIT Press, 2005.

[22] Pascal Van Hentenryck and Laurent Michel. *Constraint-based local search*, pages 11–15. The MIT Press, 2005.

[23] Scott Kirkpatrick. Optimization by simulated annealing: Quantitative studies. *Journal of statistical physics*, 34(5-6):975–986, 1984.

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

[25] Wolfgang Banzhaf. *Genetic Programming: An Introduction on the Automatic Evolution of Computer Programs and Its Applications*. The Morgan Kaufmann Series in Artificial Intelligence Series. Morgan Kaufmann Publishers, 1998.

[26] Alberto Colorni, Marco Dorigo, Vittorio Maniezzo, et al. Distributed optimization by ant colonies. *Proceedings of the first European conference on artificial life*, 142:134–142, 1991.

[27] Florence Massen. Optimization approachs for the vehicle routing problem with black box feasibility. pages 26–27, 2013.

[28] GU Clarke and John W Wright. Scheduling of vehicles from a central depot to a number of delivery points. *Operations research*, 12(4):568–581, 1964.

[29] Pham Quang Dung. Openls: A local search library for combinatorial optimization. `http://soict.hust.edu.vn/~public/openls.pdf`.

[30] Oracle. Java 8 information. `https://www.java.com/en/download/faq/java8.xml`.

[31] Oracle. Interfaces. `https://docs.oracle.com/javase/tutorial/java/IandI/createinterface.html`.

[32] Oracle. Abstract methods and classes. `https://docs.oracle.com/javase/tutorial/java/IandI/abstract.html`.

[33] Oracle. Generic types. `https://docs.oracle.com/javase/tutorial/java/generics/types.html`.

[34] Oracle. Class hashtable.
https://docs.oracle.com/javase/8/docs/api/java/util/Hashtable.html.

[35] Oracle. Class arraylist.
https://docs.oracle.com/javase/8/docs/api/index.html?overview-summary.html.

[36] Raoul-Gabriel Urma. Processing data with java se 8 streams, part 1.
http://www.oracle.com/technetwork/articles/java/ma14-java-se-8-streams-2177646.html.

[37] Éric Taillard, Philippe Badeau, Michel Gendreau, François Guertin, and Jean-Yves Potvin. A tabu search heuristic for the vehicle routing problem with soft time windows. *Transportation science*, 31(2):170–186, 1997.

[38] Python Software Foundation. Python 3.5.1.
https://www.python.org/.

[39] Oracle. Java jar information.
https://docs.oracle.com/javase/tutorial/deployment/jar/index.html.

[40] D. Elizabeth Dolan and J. Jorge Moré. Benchmarking optimization software with performance profiles. *Mathematical Programming*, 91(2):201–213, 2002.

[41] Sascha Van Cauwelaert. Performance profiles online tool.
http://sites.uclouvain.be/performance-profile/.

[42] Ingi intel room.
https://wiki.student.info.ucl.ac.be/Matériel/SalleIntel.

[43] Neo vrp instances.
http://neo.lcc.uma.es/vrp/vrp-instances/.

[44] Vrp-rep home page.
http://www.vrp-rep.org/.

[45] Philippe Augerat. *Polyhedral approach of the vehicle routing problem.* Thesis, Institut National Polytechnique de Grenoble - INPG, June 1995.

[46] Marius M Solomon. Algorithms for the vehicle routing and scheduling problems with time window constraints. *Operations research*, 35(2):254–265, 1987.

[47] Angelika Langer. Java performance tutorial – how fast are the java 8 streams?
https://jaxenter.com/java-performance-tutorial-how-fast-are-the-java-8-streams-118830.html.

[48] Nicolai Parlog. Stream performance.
http://blog.codefx.org/java/stream-performance/.

[49] Shen Lin and Brian W Kernighan. An effective heuristic algorithm for the traveling-salesman problem. *Operations research*, 21(2):498–516, 1973.

Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve **www.uclouvain.be/epl**