

Self-adapting LNS minimizing expected time to improvement, applied to the vehicle routing problem

Dissertation presented by
Antony ANCION, Benoît DARDENNE

for obtaining the Master's degree in
Computer Science and Engineering

Supervisor(s)
Pierre SCHAUS

Reader(s)
Yves DEVILLE, Sébastien MOUTHUY

Academic year 2015-2016

Contents

Introduction	9
1 The Vehicle Routing Problem	11
1.1 Problem presentation	11
1.1.1 General presentation	11
1.1.2 Extensions	11
1.2 Heuristic approaches to VRP: state of the art	14
1.2.1 Large Neighbourhood Search	14
1.2.2 Initial solution	15
1.2.3 Heuristics for VRP	17
1.3 VRP in Constraint Programming	19
1.3.1 VRP	20
1.3.2 CVRP	22
1.3.3 VRPTW	22
2 Self-Adaptive Large Neighbourhood Search	25
2.1 From LNS to ALNS	25
2.2 State of the art	26
2.2.1 The roulette wheel selection principle	26
2.2.2 Cost improvement-based adaptation	27
2.3 Self-adapting LNS Minimizing Time to Improvement	28
2.3.1 Minimizing Time To Improvement	28
2.3.2 Operators	29
3 Implementation	33
3.1 Goals	33
3.2 VRP	33
3.2.1 Models	33
3.3 ALNS	37
3.3.1 Presentation of the framework	37
3.3.2 Simple usage example : QAP	38
3.4 Implementation of Laborie	38
3.5 Implementations of ALNS-TTI	39
3.5.1 Simple ALNS-TTI	40
3.5.2 ALNS-TTI with Learning	43
3.5.3 Probabilistic ALNS-TTI	45
3.6 Applying ALNS to VRP	45
3.6.1 First solution	46
3.6.2 Operators	46
3.7 Final product	47

4 Experiments and results	49
4.1 Methodology	49
4.1.1 Instances	49
4.1.2 Parameters	50
4.2 Results for a typical instance	51
4.2.1 Gap graph	51
4.2.2 Gap over iterations graph	53
4.3 Comparison with a random selection	54
4.4 Comparison with Laborie’s ALNS	55
4.5 Operators	56
4.6 Results for other VRP variants	58
Conclusions	59
4.7 Further works	59
4.7.1 TTI-based metric with loose operators	59
4.7.2 Multi-armed bandit selection	60
4.7.3 Generic ALNS	60
4.7.4 Application to other problems	60
Appendices	64
A Gap summaries	65
A.1 X dataset [Uchoa et al., 2014]	65
A.2 R and C datasets [Solomon, 1987]	66
B Quantitative data	67

Acknowledgments

We would like to thank our promoter, professor Pierre Schaus for his help and advice, as well as our readers, professor Yves Deville and Sébastien Mouthuy. We would also like to thank Renaud Hartert for his help with **Osc**aR and his speed to apply bugfixes. Finally, we wish to thank Damien Mercier for his motivation, help and precious input.

Antony would also like to particularly thank Lara Videnovic for her support during the time needed during this master's thesis, and through the everyday life and the daily challenges. He would also like to thank Sébastien Mouthuy for his invested time, his patience, and the internship opportunity he offered that gave birth to this work.

Abstract

Large Neighbourhood Search (LNS) has been used successfully to solve many optimization problems. LNS proceeds by successive relaxations and reconstructions of an initial solution. One natural extension to LNS consists in *Adaptive LNS*, a process in which the relaxations and reconstructions methods are adapted or changed during the search. This allows the search to naturally diversify more and try different heuristics. But few such methods have been investigated in the literature. This thesis will explore the existing methods and possible frameworks for an adaptive mechanism, as well as propose a statistically-founded selection meta-heuristic and several implementations to test this meta-heuristic, called ALNS-TTI. To test and evaluate this adaptive LNS mechanism, we will apply it to several variants of the Vehicle Routing Problem. We will see that ALNS-TTI, within the specific framework proposed in this thesis, performs better than a random selection procedure but fails to outperform the best existing methods.

Introduction

The Vehicle Routing Problem (VRP) is an important optimization problems which has been extensively studied. This problem has many real-life applications in logistics, a field that deserves serious attention as minimizing transportation overhead costs can mean tremendous savings for a company.

The theoretical problem can be extended to several variants, trying to fit different situations and typical routing problems met by the industry. Each of these alternative problems can warrant some dedicated research and some ad-hoc solutions can (and are) developed to solve them independently of their father problem, VRP. The literature provides many heuristics to explore solutions of VRP and these child problems. Some are applicable to any variant of the base problem and some are more specific. But one way of generalizing the use of these heuristics would be to provide an adaptation mechanism which is able to try and evaluate the performances of a whole portfolio of heuristics and select those which seem to give the most promising results on the given problem instance. This mechanism, unaware of the nature of the problem at hand, would empirically learn from the results of the heuristics it is provided.

A successful heuristic-based approach to solving VRP and child problems is Large Neighbourhood Search (LNS). This method has been proven time and again to be able to provide good approximations of the optimal solution within acceptable time for many optimization problems, including VRP. LNS combines a fragment relaxation method with a construction strategy to explore different parts of the solution space. Since many such relaxation and construction heuristics have been proposed to help solve VRP, picking a specific pair of them can be hard: it may require tedious evaluation techniques to estimate the efficiency of a given relaxation-construction couple, and even then the evaluation only gives information about their performance with respect to the specific problem studied at the time. Here again, it seems that a learning mechanism able to handle this evaluation process at runtime and select the most promising heuristics could be a clear improvement.

Some work has been towards this purpose in the literature: [Ropke and Pisinger, 2006] and [Laborie and Godard, 2007] have proposed such adaptation schemes. But both lack a strong statistical background to ground their selection metric in a theoretical framework. We want to propose a selection mechanism that aims at minimizing the statistical expectation of the time required to improve the best solution found by LNS. We will provide an implementation (in fact, several implementations) of this proposed method and combine it with several heuristics for the VRP to see how well we can solve the problem using this technique, compared to the state of the art in this field.

In a first chapter, we will describe the general definition of the Vehicle Routing Problem as well as the main variants we will explore with constraint programming models for each of them. We will also introduce some common heuristics for VRP which will be of use. In a second chapter, we will present the state of the art methods of LNS self-adaptation and the particular metric we want to investigate. In the third chapter, we will proceed to implement the described VRP models and adaptive LNS algorithms, exposing the advantages and drawbacks of each,

and join all these elements together to solve some canonical instances of the Vehicle Routing Problem. Finally, chapter 4 will expose the obtained results and our conclusions with respects to our implementation choices. We will compare the performances of our proposed metric with the best-in-class existing adaptation mechanism.

Chapter 1

The Vehicle Routing Problem

1.1 Problem presentation

1.1.1 General presentation

The Vehicle Routing Problem (VRP) is a hard combinatorial optimization problem that has been challenging computer scientists for years. Beside its many variations, we can describe the general problem. A set of customers are spread around a map. A company disposes of a fleet of vehicles (e.g. some trucks) starting at a specific point, the warehouse (or depot), and must visit each customer once using this fleet. The objective of the problem is to find an itinerary for each vehicle, aiming at minimizing the total distance travelled (or time spent) for all vehicles, and hopefully find the optimal solution. An example of the problem is shown in figure 1.1, for an instance taken from the classical VRP dataset P from [Augerat et al., 1995].

This well-known problem is highly studied because of its obvious applications in our society, but also because finding the optimal solution to this problem has been proven to be NP-Hard[Lenstra and Kan, 1981]. There are several techniques and heuristics to approach the problem, and find solutions.

To find these solutions, there are multiple ways to explore. Some methods to find the one, exact and optimal solution exist. But other heuristic-based methods can provide solutions faster, at the trade-off of optimality, and for a larger number of instances than exact techniques. These heuristic based methods can have different advantages and drawbacks, and sometimes combining them can reveal to be a powerful tool to find approximation of the optimal solution. During this master's thesis, we will use multiple heuristics and focus on a meta-heuristic allowing to sort them according to their performances and computation time to get the best possible solution in a reasonable time.

1.1.2 Extensions

This problem can be modified to fit many real-life problems, and many variations of the canonical VRP exist. The following list covers some extensions of the basic problem which take into account various constraints that the real-life logistics industry may face:

Profitable tour: reaching each customer is not mandatory. Each visit generates a profit and each non-visited point has a cost.

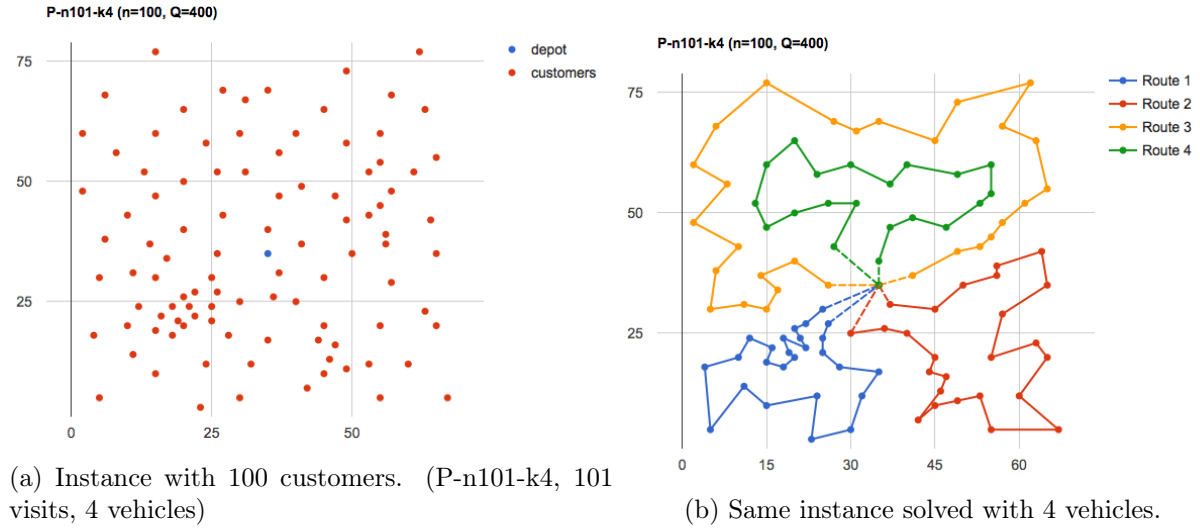


Figure 1.1: Example of capacitated vehicle routing problem [CVRPLIB, 2016]

Driver daily maximum travel time and breaks: constraints over drivers and their travel time.

Vehicle modifications: Ability to change vehicle characteristics (e.g. add a trailer)

Loaded goods constraints: Limited travel time for specific goods (e.g. blood containers that cannot stay more than some time into a vehicle)

Precedence constraints: Some customers must be visited before some other.

Incompatibilities between customers: e.g. a vehicle cannot visit specific customer after another one, or with a specific good loaded.

Multiple depots: Start and end points are not the same depot. Vehicles can start or finish their tour at any depot, or following given constraints.

Vehicles with loading modelisation: Last-in-first-out load constraint or 2D/3D load profile.

Given these multiple variations, we see that our model must be as flexible as possible to be able to modify it at will, according to the requirements of the problem we are facing. To do so, our choice was to use Constraint Programming, as we will see in section 1.3. Some very common variations of the VRP are the Capacitated VRP (CVRP), the VRP with Time Windows (VRPTW) and the pickup and delivery problem (PDP).

Capacitated VRP

In the *capacitated VRP* variant (CVRP), a demand is associated to each visit and the vehicles have a limited carrying capacity. These demands and capacity may be uni-dimensional or multi-dimensional with specific values for each commodity. The capacity can also be vehicle-dependent (in the case of a heterogeneous fleet).

In addition to the geographical information naturally required by the canonical VRP (locations of the depot and customers), the single-commodity CVRP with a homogeneous fleet (which will be a variant we will investigate) requires the following information:

- A demand for each visit.
- The vehicle capacity.

VRP with time windows

In the *VRP with time windows* variant (VRPTW), each customer is associated with a time window, representing the limits allowed for the start time of the visit. The vehicles can arrive early but cannot start the service before the opening of the time window. In this case, the vehicle has to wait. However the vehicle cannot arrive after the end of the time window. The service has an associated duration which the vehicle has to spend at the facility before departing.

We can also consider customers with multiple disjoint time windows. If the vehicle arrives between two time windows, it has to wait until the beginning of the next time window to perform its service.

And finally, it is possible to define the problem with *soft* time windows. In this case, the time window constraint can be violated, with the effect of adding a cost to the objective function. The travel time may also depend on the vehicle speed (heterogeneous vehicles) and/or be time-dependent (e.g. rush-hour, night travel, ...).

When considering a single time-window VRPTW, we require the following additional information:

- A time window for each customer (defined by a start and end times).
- The duration of a visit.
- The travel time between each pair of customers.

Pickup and delivery VRP

In the *Pickup and delivery problem* (PDP), some goods need to be moved from pick-up locations to delivery locations, instead of from/to a fixed depot. We can visualize a load profile for each vehicle, representing its load at any time of the tour.

Each customer location can either be a *pickup* location or a *delivery* location, where the vehicle load increases and decreases, respectively. The transported goods must go from one location to another one, and each pickup customer has a designated delivery location associated to it.

This particular subproblem, is characterized by the following extra information:

- The demand for each customer, positive (resp. negative) for a pick-up (resp. delivery) location.
- The capacity of the vehicles (still homogeneous)
- The delivery information: to which delivery location should each picked-up goods go.

Extensions may be combined

It is obvious that CVRP, VRPTW, PDP and many other variants simply add more constraints to the original problem, and some of them may be combined together to defined more and more constrained problems. For instance, one can define a pickup and delivery problem with time windows, or a capacitated VRP with time windows, or a CVRPTW with driver breaks, maximum goods travel time, and so on and so forth. We can therefore imagine any kind of combination we can think of to tie these problems and subproblems together.

1.2 Heuristic approaches to VRP: state of the art

As the Vehicle Routing Problems are part of the class of NP-hard problems [Lenstra and Kan, 1981], the found solutions are qualified as 'best known solution' until they are proven to optimality. To do so, some exact methods exist. Unlike the heuristic based methods, they take more time to run. Some algorithms use *Branch-and-Cut-and-Price-algorithm* (BCP), using a combination of cut and column generation, in order to exactly solve VRP [Toth and Vigo, 2014].

On the other side, heuristics-based solvers are used to find solutions to vehicle routing problems in a more practical way. Using heuristics allows to find solutions quickly using a construct-and-improve methodology. First it aims to construct a feasible solution, then improve it using local search methods or a large neighbourhood search (LNS). The heuristics, however, do not particularly attempt to find the optimal solution, but rather to minimize the gap to optimality as much as possible in a reasonable time.

1.2.1 Large Neighbourhood Search

One way of solving VRP and associated problems is *Large Neighbourhood Search*, as shown with success in [Shaw, 1998, Ropke and Pisinger, 2006, Hong, 2012, Pisinger and Ropke, 2010].

LNS is a search algorithm to partially explore large solution spaces. The idea is to start from a first solution and improve upon this solution by iterating two steps:

A relaxation phase, wherein the initial solution is relaxed, i.e. some part of the solution is temporarily forgotten while the rest of the solution is kept as it was.

A search phase, wherein a search process starts from the relaxed solution and searches for improvements on this relaxed solution.

We can see that an LNS algorithm must therefore be provided with three inputs :

1. The initial solution, x_0
2. A relaxation function, or a *large neighbourhood* $LN(x)$, which relaxes a solution x into a partial solution x' . This function may possibly take parameters, like the amount of nodes in the solution to relax or keep intact.
3. A search method, or a *completion strategy* $CS(x')$, which searches from a partial solution x' and returns a solution x'' , hopefully improving on the original solution x . This function may possibly take parameters.

Given $c(x)$, a function that maps a given solution x to its cost (the travelled distance, in the case of VRP), LNS simply iterates over the search $CS(LN(x))$, where x is of course initially x_0 , and keeps the results of the search if it improves the current best solution (i.e. $c(x'') < c(x)$). If we combine both the LN and CS operators into a single function, we can use the notation $LNS(x) = CS(LN(x))$. We will call the LNS function an *LNS operator*. A simple algorithm, inspired by the notations of [Pisinger and Ropke, 2010] and [Schaus, 2016b] is given in the Algorithm 1.

The choice of the input parameters (both the initial solution, and the couple of (CS, LN) heuristics) is obviously crucial to the performance of LNS, and this is the starting point of our work. We will especially be looking into ways of adapting the relaxation and search methods along the way of the search, in order to use appropriate couples of relaxation and search heuristics every time. This topic will be developed in chapter 2. For now, we will describe some methods

Algorithm 1 LNS

Input: x_0 $\triangleright$ an feasible initial solution
Input: LNS $\triangleright$ a relax and a rebuild operator, or LN and CS

```

1:  $x^* = x_0$   $\triangleright x^*$  stores current best solution
2: repeat
3:    $x = LNS(x^*)$   $\triangleright$  Or,  $x = CS(LN(x^*))$ 
4:   if  $c(x) < c(x^*)$  then
5:      $x^* = x$ 
6:   end if
7: until stopping criterion is met
8: return  $x^*$ 

```

which can be used (and will be used in the implementation chapter) to provide these three input parameters to LNS, in the case of the vehicle routing problem.

1.2.2 Initial solution

The initial solution which is passed to LNS clearly gives the tone of the overall results of the process. LNS will not completely revamp the first solution and will only improve the given solution by small increment, up to a certain point. This is known to give good results rather quickly, but the final output solution will often be quite different, depending on the shape of the first solution. There are many ways to find a first solution for VRP. We will briefly describe some of them.

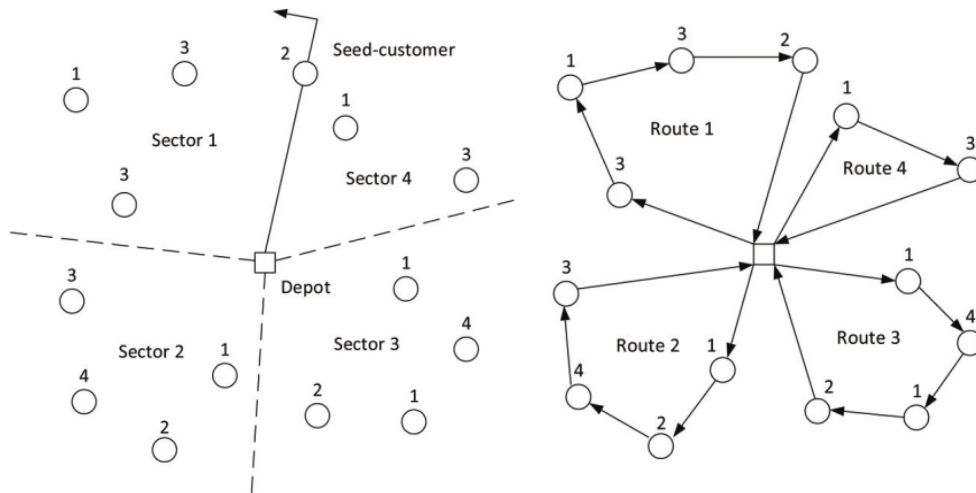


Figure 1.2: Sweep heuristic example [Labadie et al., 2016].

Sweep heuristic [Gillett and Miller, 1974] This is a "Cluster first, route second" type of algorithm. The principle, for a CVRP, is to trace a ray from the gravity center of the nodes or from the depot and sweep clockwise to create clusters of customers without exceeding vehicle capacity. When the vehicle is full, we create a new route. Finally, we solve a TSP of each cluster of customer. In practice, we can use the polar coordinates of the visits, according to the warehouse. An example of this heuristic is shown on figure 1.2.

In [Solomon, 1987]), the authors described a time-oriented sweep heuristic, to deal with

time windows. First, it uses the regular sweep solution to cluster visits. Then, in the second phase, we solve a one-vehicle scheduling problem for each cluster. Some customers might remain unscheduled. We repeat the clustering and scheduling method for these unscheduled visits.

Savings heuristic [Clarke and Wright, 1964] This heuristic is based on the computation of the savings $s(i, j)$ induced by servicing clients i and j on the same route instead of two different routes. The savings can be computed from the distances $d(i, j)$ between each pair of customers i and j . In the example below, the total distance in figure 1.3a (resp. 1.3b) is computed using equation 1.1 (resp. 1.2), where D is the depot. The saving of merging the routes is therefore computed using equation 1.4.

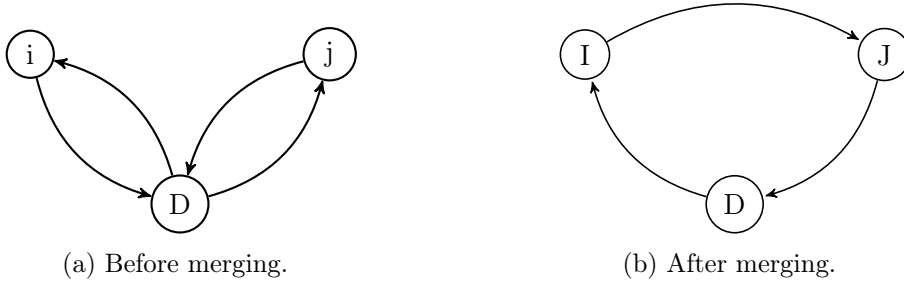


Figure 1.3: Savings principle

$$D_a = d(D, i) + d(i, D) + d(D, j) + d(j, D) \quad (1.1)$$

$$D_b = d(D, i) + d(i, j) + d(j, D) \quad (1.2)$$

$$s_{i,j} = D_a - D_b \quad (1.3)$$

$$= d(D, i) + d(D, j) - d(i, j) \quad (1.4)$$

The savings are sorted decreasing order in a *savings list*. The first element (with the greatest savings value) is removed from the list and processed as follows [Larson and Odoni, 1981], if no route constraint is violated :

- If neither i nor j are currently assigned, then a new route is created, linking i and j .
- If both i and j are included in different existing routes and neither points are interior¹ to their routes, then the routes are merged, by linking i to j .
- If exactly one of them is included to an existing route, and is not interior of that route, then the link (i, j) is added.

Note that this heuristic can also be improved to handle the time windows in addition to the distances [Solomon, 1987].

Nearest neighbour, Greedy algorithm This method adds the closest (based on distances and time aspects) unbounded customer from the previously added customer (initially, from the depot) to the current route. Once no client can be added to the route without violating a constraint, a new route is created in the same way, until no customers are left.

¹A point is interior of a route if its not directly adjacent to the depot following that route.

This simple greedy algorithm, based on distances and/or time aspect might give very good results for simple instances. Especially on CVRP instances, where the greedy algorithm applies on distances between visits.

1.2.3 Heuristics for VRP

For this work, we will mainly focus on well-known relaxation and completion VRP-specific heuristics, especially those described in [Ropke and Pisinger, 2006], [Gendreau and Potvin, 2010] and [Ribeiro and Laporte, 2011]. However, it is interesting to note that some work has been done to try and design generic relaxation methods, which are able to automatically define a fragment regardless of the problem definition. In [Mairy et al., 2010], the authors define heuristics based on metrics of the *impact*, the *variability* and the *proximity* of the variables. In [Lombardi and Schaus, 2014], it is a cost-based approach that is used while [Perron et al., 2004] uses a propagation based technique. Here, we will first focus on hand-written relaxation methods specifically designed for the VRP.

Relaxation heuristics

We will first describe 5 relaxation heuristics. Their purpose is to destroy parts of the current best found solution, in order to be able to rebuild it in a better way, improving this original solution. Each of these heuristics take as parameter the *size* of the relaxation, i.e. how large a fragment of the current solution should be destroyed. The function selects the provided number of visits, and removes them from the current solution (i.e. no route is coming from or going to these visits anymore).

Random visits relaxation This is the simplest heuristic we can imagine. We select the desired number of visits to relax randomly among all the visits, and remove them from the solution. Its simplicity gives it the advantage of being easy to implement, fast to execute and might help in diversifying the results, especially in the early stages of the search, while the problem is not too tight yet.

Related visits relaxation This relaxation randomly selects a visit and relaxes its closest neighbours. This heuristic is very efficient in CVRP problems by removing clusters of customers based on their locations. The *relatedness* is therefore defined in terms of the distance between the customers. This property may be adapted to other problems to take care of other aspects, such as the relatedness of customers time windows in addition to their location.

Path and related visits relaxation To relax n nodes, this heuristic will randomly select a customer, and removes $n/2$ nodes on this customer's route. This means that it relaxes visits being on the same route as this random customer, and being in its successors or its predecessors. The other $n/2$ nodes to relax are the closest nodes of the selected customer (geographically), among those nodes who were not yet relaxed. Figure 1.4 shows an example where nodes from three distinct routes are relaxed using this heuristic. The idea is basically to relax a part of one route, as well as one geographical area, to reorganize the routes of the nodes in this area.

Worst visits relaxation For each visit, we compute the cost of the solution if that visit were completely absent from the dataset. We select the visit minimizing the cost computed this way for relaxation. Then we link its predecessor to its successor, and restart the heuristic

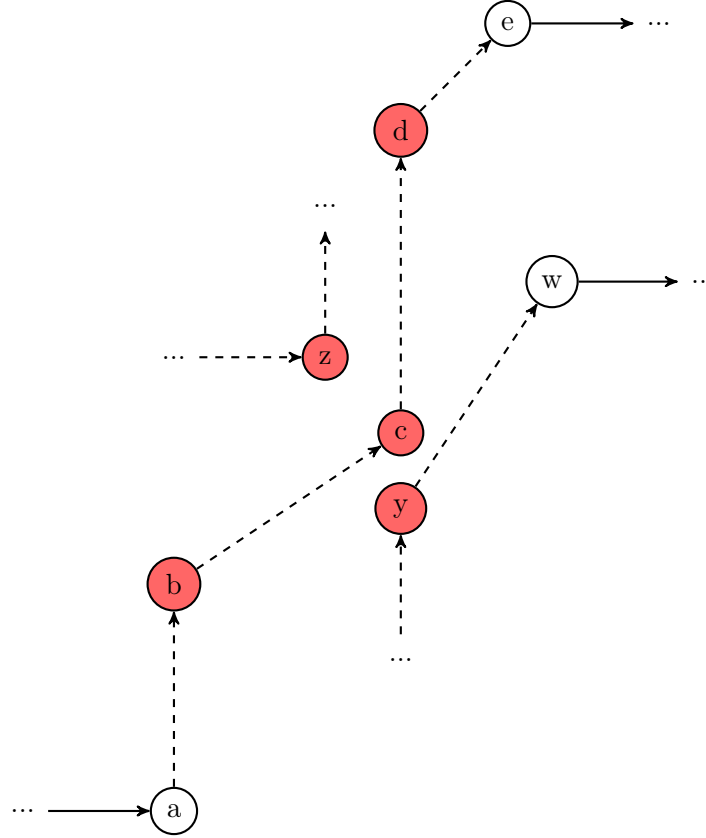


Figure 1.4: Path and related relaxation example (with $n = 5$ relaxed nodes). The red nodes are chosen by the path and related relaxation if c is selected at random

on that freshly created dataset to find another visit to relax. The process is repeated until reaching the desired number of relaxed nodes. In their algorithm (adapted here in algorithm 2) [Ribeiro and Laporte, 2011] uses the notation $Cost^-(i, s)$ to represent the cost associated with the solution s , if the customer i was completely removed from s , and R represents the set of nodes to relax.

Mixed routes visits relaxation The purpose of this heuristic is to create holes in two close routes, to have a chance of producing a good combination of both of them by exchanging free parts of routes. To do so, we randomly pick a visit, compute its closest visit that is on another route to have 2 close routes. Then we randomly remove $n/2$ nodes on each route.

Rebuilding heuristics

The following 3 rebuilding heuristics will aim to search for possible solutions based on previously relaxed partial solutions.

Greedy heuristic The greedy heuristic focuses on selecting as successor or predecessor of a given visit, the closest possible neighbour, geographically.

Deeply Greedy heuristic This is a variant of the greedy heuristic with a more global view on the variables. This heuristic focuses, like the greedy heuristic, on selecting the closest neighbour of a variable, but also to first assign the variable with the smallest distance to its neighbour.

Algorithm 2 Worst visits relaxation [Ribeiro and Laporte, 2011]

Input: s ▷ an feasible solution
Input: $size$ ▷ desired size for the relaxation

```

1:  $R = \emptyset$ 
2: repeat
3:    $L = Customers(s)$  ▷ select all customers of  $s$ 
4:   Sort  $L$  such that  $i < j \Rightarrow Cost^-(i, s) < Cost^-(j, s)$  ▷ sort according to the cost of the  

▷ solution with removed customer
5:    $R = R \cup first(L)$ 
6:   Remove  $first(L)$  from  $s$ 
7: until  $size(R) \geq size$ 
8: return  $R$ 

```

Regret-k heuristic For this research, we aim to have an even larger point of view on the problem in order to choose which variable to assign first. To do so, the regret-2 function computes the *regret value* of each visit, which is equal to the difference of cost between its best route and its second best route. This value represents the regret of not selecting its best successor. The purpose is to select first the variable with the highest regret value. If a variable with a high regret value does not select its best choice neighbour (the closest), it will have to select a visit with a far higher cost (more distance). Therefore, it's better to bound this variable as soon as possible in the decision process.

We can also extend this notion to several value of the k parameter. It defines the number of visits used in the difference to compute the regret value, and represents the difference of selecting its second, third, ..., k^{th} best route [Ribeiro and Laporte, 2011]. For this heuristic, the value has to be maximized instead of minimized as in the greedy heuristics.

Intuitively, the regret is a kind of “look-ahead” that indicates what could be lost later, if a given customer is not immediately inserted within its best route. In our context, a large regret measure means that there is a large gap between the best insertion place for a customer and its best insertion places in the other routes. Hence, un-routed customers with large regrets must be considered first, since the number of interesting alternative routes for inserting them is small. On the other hand, customers with a small regret measure can be easily inserted into alternative routes without loosing much, and are thus considered later for insertion.

Selecting a heuristic

These heuristics (both of the relaxation and reconstruction kind) are not always effective in the same way. Some are complex and take some time to be executed, while others are simpler, cheaper and faster. Some might often improve the current solution, while others are more effective when the problem is harder. Choosing a single couple of heuristic might therefore be complex, and the result of the LNS algorithm will mainly depend on this choice. This is the reason we will use an *Adaptive Large Neighbourhood Search* (ALNS) and devise schemes to select different strategies along the way.

1.3 VRP in Constraint Programming

Constraint programming has been used to solve VRP with some encouraging success in the literature [De Backer et al., 2000, Shaw, 1998, Rousseau et al., 2004]. Its declarative capabilities

and great flexibility allow to easily express the base problem [Kilby, 2013], as well as other more constrained variants. The LNS algorithm presented in section 1.2.1 is also naturally well suited to be used with a constraint programming solver:

1. Relaxing a solution in CP is simply a matter of taking the problem in reverse, i.e. instead of relaxing some nodes, we just keep a part of the current best solution. By simply adding some constraints to assign a value to the part of the solution we kept, we essentially relaxed the original solution into a fragment we can explore with a search method.
2. In the constraint programming paradigm, one typically provides the solver with a user-specified search mechanism, which means defining and re-defining a search strategy along the way is a built-in feature of CP.

In this section, we will provide CP models for the VRP variants we selected to investigate, namely: VRP, CVRP, VRPTW and PDP. We will start with a model for VRP, complete with decision variables and constraints. Then, we will see how we can extend this particular model to add extra decision variables and constraints, depending on the extension.

1.3.1 VRP

The following model is inspired by the model described in [Kilby, 2013]. We will start by defining the set of n customers as N . We will then define m , the maximum number of vehicles². For each of the m vehicles, we will define a set of additional “fake” visit points: each vehicle will have a start and end points, which have the same coordinates as the warehouse and act as a “virtual” warehouse.

We will denote as S (of size m), the set of start visits so that the point S_k is the starting point of vehicle k . Similarly, we define E (also of size m), the set of end visits so that E_k is the end point of vehicle k . In that way, vehicle k ($k \in M$ defined as $[0, m - 1]$) will start its visits at $n + k$ and end it at $n + m + k$.

Therefore, the whole set of visits will have a length of $n + 2m$ structured in this way (see figure 1.5 :

- N : the n customers, from 0 to $n - 1$
- S : the m start nodes, from n to $n + m - 1$
- E : the m end nodes, from $n + m$ to $n + 2m - 1$

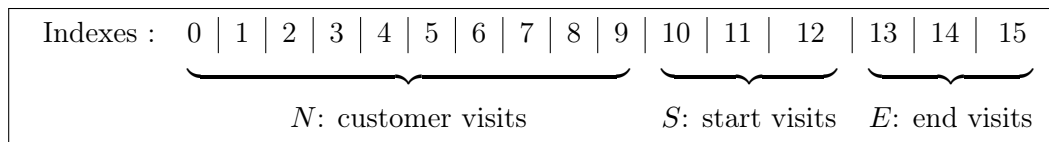


Figure 1.5: Structure of the indexes of the visits vector for 10 actual customers ($n = 10$) and 3 vehicles ($m = 3$).

Decision variables When modeling VRP in constraint programming, we need to define variables for which the solver will have to decide appropriate values respecting the constraints. Specifically, what we are looking for is:

succ(i) Successor associated with each visit i , represents the next visit performed by the vehicle.

²Note that in our variant of VRP, we do not optimize over the number of vehicles but strictly on the total distance.

pred(i) Predecessor associated with each visit i , represents the previous visit performed by the vehicle.

route(i) Route associated with each visit i , represents which vehicle is serving customer i . There are k routes, each of them corresponding to one of the k vehicles.

Strictly speaking, *succ* and *pred* contain redundant information and only one of them is required in order for the obtained solution to satisfy the constraints of VRP, but empirical evidence has shown that the use of both is often more efficient.

Constraints

1. All the successors must form a circuit.
2. All the predecessors must form a circuit.

These two constraints simply mean that the *succ* and *pred* array are such that they form a Hamiltonian cycle. In other words, following the pointers of the *succ* array from any index i , we must eventually reach the starting index, while having stepped over each possible index in the process.

3. Ensure consistency: the predecessor of the successor of i must be i itself.

$$succ(pred(i)) = i \quad \forall i \in N \cup E \quad (1.5)$$

$$pred(succ(i)) = i \quad \forall i \in N \cup S \quad (1.6)$$

4. Each visit is on the same vehicle route than its successor (except for end nodes).

$$route(i) = route(succ(i)) \quad \forall i \in N \cup S \quad (1.7)$$

5. Each visit is on the same vehicle route than its predecessor (except for start nodes).

$$route(i) = route(pred(i)) \quad \forall i \in N \cup E \quad (1.8)$$

6. Each vehicle is associated with its start and end visits.

$$route(n + k) = k \quad \forall k \in M \quad (1.9)$$

$$route(n + m + k) = k \quad \forall k \in M \quad (1.10)$$

7. The successor of an end node is the next start node (unless the successor of the last end node, that is the first start node).

$$succ(i) = i - m + 1 \quad \forall i \in E \setminus \{E_{m-1}\} \quad (1.11)$$

$$succ(i) = S_0 \quad \text{if } i = E_{m-1} \quad (1.12)$$

Constraint 6 assign one route (or vehicle) to each starting point and ending point. Constraint 7 links all routes together, with the beginning of one following the ending of another. This allows us to use one circuit constraint for the whole problem and avoids the problem of defining one such constraint for each route separately.

Objective function The objective function of a vehicle routing problem is to minimize the distance (or travel time) between each successive visits.

$$\text{minimize } \sum_{i \in N \cup S} \text{distance}(i, \text{succ}(i)) \quad (1.13)$$

1.3.2 CVRP

This variant is relatively simple. Each customer is associated with a given integer *demand*, and each vehicle will accumulate the demands of the customers along its route, without ever exceeding its *maximal capacity*. This maximal capacity is equal for all vehicles (homogeneous fleet).

Additional decision variables

load(i) Associated with the customer visit i , represents the vehicle load for the commodity after serving that customer.

CVRP Constraints

1. The load of a vehicle cannot exceed the maximum capacity, and cannot be less than zero.

$$0 \leq \text{load}(i) \leq \text{MaxCapacity} \quad \forall i \in N \cup S \cup E \quad (1.14)$$

2. The vehicles are empty at the start nodes.

$$\text{load}(i) = 0 \quad \forall i \in S \quad (1.15)$$

3. The load is accumulated along the path of a vehicle.

$$\text{load}(\text{succ}(i)) = \text{load}(i) + \text{demand}(\text{succ}(i)) \quad \forall i \in N \cup S \quad (1.16)$$

In our case, as we will see in the implementation chapter, these specific constraints can be seen as a bin packing problem, handled as such by the constraint programming solver (with some powerful and efficient propagators associated).

1.3.3 VRPTW

When dealing with time windows, we must simply add some constraints to make sure the customer's desires are respected. We will have to decide at which time to arrive at each customer, as well as the time the service will actually take place.

Additional decision variables

arrival(i) The arrival time at customer i .

service(i) The service time of customer i .

departure(i) The departure time from customer i .

VRPTW Constraints

1. Trucks depart at time 0 for start nodes S .

$$departure(i) = 0 \quad \forall i \in S \quad (1.17)$$

2. The customer can only be served after the truck has arrived.

$$arrival(i) \leq service(i) \quad \forall i \in N \quad (1.18)$$

3. Service is done in the customer's time window.

$$service(i) \geq start(i) \quad \forall i \in S \quad (1.19)$$

$$service(i) \leq end(i) \quad \forall i \in S \quad (1.20)$$

4. Vehicle leaves after service is done.

$$departure(i) = arrival(i) + service_time(i) \quad \forall i \in N \quad (1.21)$$

5. Arrival time at successor of customer i is equal to departure time from i plus the travel time³ from i to the successor of i .

$$arrival(succ(i)) = departure(i) + distance(i) \quad \forall i \in N \cup S \quad (1.22)$$

PDP

This variant is quite similar to CVRP in that each vehicle has a given maximum capacity (we will stay in the case of an homogeneous fleet here). However, some customers will have negative demands (indicating a delivery location) and each customer will have an associated *sibling* customer, each pair of siblings forming a pickup-delivery couple. Note that we must of course enforce that the vehicle goes to the the pickup location before the delivery location, and that both are located on the same route.

Additional decision variables

load(i) Associated with the customer visit i , represents the vehicle load for the commodity after serving that customer.

order(i) The order of the customer visit i along its route (first customer visit is 1, second is 2, etc.).

PDP Constraints

³In this work, we will consider that the travel time is a deterministic function of the distance, and as such, we will always use the distance as the travel time. One could simply extend the problem to specify separate travel times between each pair of customers. It is also possible to define stochastic travel times.

1. The load of a vehicle cannot exceed the maximum capacity, and cannot be less than zero.

$$0 \leq load(i) \leq MaxCapacity \quad \forall i \in N \cup S \cup E \quad (1.23)$$

2. The vehicles are initially empty and the order begins at 0

$$load(i) = 0 \quad \forall i \in S \quad (1.24)$$

$$order(i) = 0 \quad \forall i \in S \quad (1.25)$$

3. The load is accumulated along the path of a vehicle.

$$load(succ(i)) = load(i) + demand(succ(i)) \quad \forall i \in N \cup S \quad (1.26)$$

4. The order is incremented along the path of a vehicle.

$$order(succ(i)) = order(i) + 1 \quad \forall i \in N \cup S \quad (1.27)$$

5. Corresponding pickup and delivery locations must be on the same route

$$route(i) = route(sibling(i)) \quad \forall i \in N \quad (1.28)$$

6. Pickup must be performed before delivery

$$order(i) < order(sibling(i)) \quad \forall i \in \{x | x \in N \wedge x > 0\} \quad (1.29)$$

In this last constraint, the set $\{x | x \in N \wedge x > 0\}$ is of course the set of all pickup locations, having a positive demand.

Chapter 2

Self-Adaptive Large Neighbourhood Search

2.1 From LNS to ALNS

Since the choice of the relaxation and search heuristics is so important, one can wonder if it could be interesting to use different methods at different times of the search process. This is the starting point of ALNS, (Self-)Adaptive Large Neighbourhood Search. In ALNS, the algorithm is provided with several relaxation and search heuristics, and uses a meta-heuristic to select, at every iteration, which LNS operator should be run. The idea is to use different heuristics along the search to allow, for instance, the use of fast LNS operators in the early iterations while more CPU-expensive operators could become interesting later on, when the problem gets harder as we get closer to the optimum.

A version of a generic ALNS, based on the LNS presented in algorithm 1, is given in algorithm 3. The *select* function applies the selection meta-heuristic.

Algorithm 3 ALNS

Input: x ▷ an feasible initial solution
Input: $operators$ ▷ a collection of LNS operators

1: $x^o = x$ ▷ x^o stores current optimal solution
2: **repeat**
3: $op = \text{select}(operators)$
4: $x^t = op(x^o)$
5: **if** $c(x^t) < c(x^o)$ **then**
6: $x^o = x^t$
7: **end if**
8: $\text{update}(op)$ ▷ Possibly update some statistics about op
9: **until** stopping criterion is met
10: **return** x^o

The topic of this chapter will be to introduce some existing operator selection meta-heuristics and present the idea at the center of this work.

2.2 State of the art

2.2.1 The roulette wheel selection principle

In [Ropke and Pisinger, 2006], the authors present a search heuristic for solving the Pickup and Delivery Problem with Time Windows (a combination of the PDP and VRPTW variants presented in section 1.1.2). The heuristic consists in an adaptive LNS, using an operator selection method based on a probabilistic model called the *roulette wheel selection principle*.

In this method, every LNS operator i has an associated weight w_i and at every LNS iteration this operator is selected with a probability

$$p_i = \frac{w_i}{\sum_j w_j} \quad (2.1)$$

The weights are automatically determined by the algorithm by way of computing and keeping track of a score for each and every available operator. Initially, the score π_i of operator i is 0. After each run of an operator, its score is updated and increased by a factor which depends on which of the three following outcomes (if any) occurred:

- The last run of the operator found a new global best solution
- The last run of the operator found an accepted¹ solution better (in terms of cost) than the current solution
- The last run of the operator found an accepted solution worse (in terms of cost) than the current solution

Periodically, the weights of the operators are recomputed using the recorded scores. If $w_{i,k}$ is the weight of operator i after being updated at time k , its weight at the next update $k + 1$ will be:

$$w_{i,k+1} = (1 - \alpha)w_{i,k} + \alpha \frac{\pi_i}{\theta_i} \quad (2.2)$$

where:

π_i is the score of operator i .

θ_i is the number of times it was run.

α is a reaction (or decaying) factor controlling how much past information should be kept throughout the algorithm versus how fast the weights reflect changes in the effectiveness of the operator.

Conceptually, the probability distribution over the operators models a “roulette wheel”, where every operator occupies a fraction of the wheel. Initially, they are all equiprobable: they all occupy an equal fraction of the wheel, as shown on figure 2.1a. But as the algorithm progresses and LNS iterations are performed, good operators see their weight increase (in the case of [Ropke and Pisinger, 2006], proportionally to their average score), and they occupy more space on the wheel than operators that got a lower score (figure 2.1b). Operators with a good score therefore become more and more likely to be selected for future iterations. Figure 2.1 illustrates

¹In their implementation of LNS, the authors allow solutions to be *accepted* (they become the original solution to be relaxed and rebuilt upon) even if they are not better than the current best solution. We did not consider this approach in our implementation of LNS in this thesis.

a roulette wheel with 5 operators. In this example, operators B, C and E saw their selection probability increase at the expense of operators A and D.

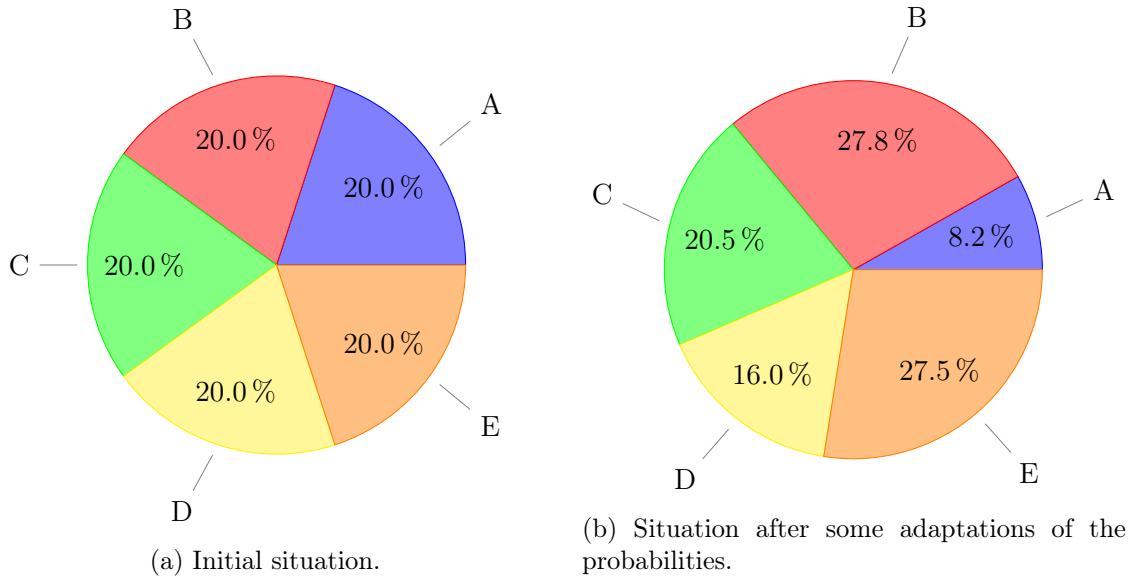


Figure 2.1: Example of the roulette wheel adaptation mechanism.

2.2.2 Cost improvement-based adaptation

In [Laborie and Godard, 2007], a similar approach is used but with a slightly different framework and a different adaptive strategy.

Here, the same roulette wheel selection method is used but the operators are rewarded proportionally to the ratio of the cost improvement on their running time. Equation 2.2 takes a more general form:

$$w_{i,k+1} = (1 - \alpha)w_{i,k} + \alpha r \quad (2.3)$$

with, in the case of [Laborie and Godard, 2007]

$$r = \frac{\Delta c}{\Delta t} \quad (2.4)$$

where:

Δc is the cost improvement if the operator found a better solution than the global best, 0 otherwise.

Δt is the cycle CPU time used by the operator.

The idea is to reward operators which improve upon the solution faster, because distinct LNS operators may have significantly different complexities and running times. Since the goal is to reach a good solution as fast as possible, it seems sensible to favor operators which improved the solution in less time.

In this paper, as in [Ropke and Pisinger, 2006], the operators are separated in two portfolios of *LN* and *CS* heuristics, as described in section 1.2.1, and both sets of relaxation and reconstruction methods are given their own probability distribution, denoted respectively $prob_{LN}$ and $prob_{CS}$. At every iteration, both the *LN* and *CS* heuristics are selected independently.

Furthermore, each heuristic i (either relaxation or rebuilding), may have a parameter vector $v_i = (v_i^1, v_i^2, \dots, v_i^n)$ with n the arity heuristic i . Each parameter v_i^k will take its values in a finite set V_i^k , and the algorithm will maintain for each parameter of each heuristic a probability distribution $prob_{V_i^k}$ to select its possible values in V_i^k .

$$h(\begin{array}{|c|c|} \hline & v_h^1 \\ \hline 10\% & 0.35 \\ \hline 20\% & 0.25 \\ \hline 30\% & 0.15 \\ \hline 40\% & 0.15 \\ \hline 50\% & 0.10 \\ \hline \end{array} \begin{array}{c} V_h^1 \\ prob_{V_h^1} \end{array} , \begin{array}{|c|c|} \hline & v_h^2 \\ \hline \text{Weak} & 0.05 \\ \hline \text{Medium} & 0.30 \\ \hline \text{Strong} & 0.65 \\ \hline \end{array} \begin{array}{c} V_h^2 \\ prob_{V_h^2} \end{array})$$

Figure 2.2: Example of probability distributions for the parameters of heuristic h with arity 2.

Figure 2.2 shows an example for a heuristic with two parameters: a percentage parameter (the amount of nodes to relax in the solution, for instance) as well as a discrete parameter with three possible values (Weak, Medium or Strong). The left columns are the possible parameter values and the right columns their associated selection probability.

2.3 Self-adapting LNS Minimizing Time to Improvement

2.3.1 Minimizing Time To Improvement

The idea at the core of this thesis is to follow the approach in [Laborie and Godard, 2007] but with some modifications to ground the selection mechanism in a well-defined statistical frame. In an unpublished draft [Schaus, 2016b], P. Schaus proposes a selection strategy wherein the operator with minimal *expected time to improvement* (hereafter referred to as *TTI*) is selected.

We will start by defining the expected time to improvement. Let p_i be the probability of improving the current best solution for operator LNS_i , and T_i the time required to execute this operator. p_i is the probability of success of a geometric distribution, which has an expected value of $1/p_i$. Intuitively, this means that an average of $1/p_i$ trials of the LNS_i operator will be required until it can improve the current best solution.

Given this expected number of trials until improvement, and the average running time of the operator, we can compute the expected time until the operator can improve the solution with:

$$TTI_i = \frac{T_i}{p_i} \quad (2.5)$$

Selecting the operator LNS_i with the lowest TTI_i gives us a selection meta-heuristic with a sound statistical basis. The p_i and T_i values can be estimated along the way by collecting statistics after each execution of an operator, and so the algorithm will “learn” the required information to estimate the TTI of all operators.

In its original version, the idea was to use the algorithm listed in algorithm 3 with the select function simply picking the operator with minimal TTI . See algorithm 4 for an instantiation of the generic ALNS algorithm using this selection mechanism. However, as we will see in chapter 3, some problems arise from such a simple construct and we will experiment with some (slightly) more complex variants to solve these problems.

Algorithm 4 ALNS-TTI

Input: x_0 ▷ a feasible initial solution
Input: *operators* ▷ a collection of operators

```

1:  $x^* = x_0$  ▷  $x^*$  stores current optimal solution
2: repeat
3:    $op = \operatorname{argmin}_i \{operators[i].TTI\}$ 
4:    $x = op(x^*)$ 
5:   update  $op.T$  and  $op.p$  ▷ update running time and success probability
6:   if  $c(x) < c(x^*)$  then
7:      $x^* = x$ 
8:   end if
9: until stopping criterion is met
10: return  $x^*$ 

```

2.3.2 Operators

It is important to notice that so far we used both the $LNS(x)$ and $CS(LN(x))$ notations depending on the context. As said in section 1.2.1, both are equivalent and the former is just a shorter version of the latter. However, both [Laborie and Godard, 2007] and [Ropke and Pisinger, 2006] (among others) separate the large neighbourhood heuristics and construction strategies, which implies that both must be selected independently of each other. We also saw that the parameters can also be varied “inside” each heuristic, so that the same heuristic with two different parameter vectors are not conceptually considered as different heuristics. When both the LN and CS heuristics are not coupled together in a LNS operator, together with parameter values, we will say the algorithm uses *loose operators*.

But if we consider that one large neighbourhood *together* with a construction strategy, both with one fixed parameter vector attached, constitute one and only one LNS operator, we will say that we use *tightly coupled operators*. This is an important distinction, and we can see immediately that this will influence both the implementations and the selection mechanism.

Let’s consider the case of a roulette wheel selection with loose operators. We can identify two problems that may arise:

1. If a heuristic is given many possible parameter vectors but only one of them constitutes an efficient combination, the heuristic will be punished and get a bad score every time it is run with a bad parameter combination, therefore decreasing the probability of this heuristic being selected, even though it would work well with a particular parameter vector.
2. If a heuristic (say an LN heuristic) works well only when combined with a particular construction strategy, then this heuristic will also get punished whenever it is run with an inappropriate companion heuristic. This reduces the chances of selecting the couple that would perform well together.

Essentially, the idea is that perhaps some relax-rebuild heuristics work well only when coupled together but poorly in all situations, and splitting the selection in two can decrease the chances of this specific couple of being picked. For these reasons, and because the literature is lacking in results concerning adaptive LNS strategies using tightly coupled operators, we opted to use them for the algorithms of our creation.

The differences between the two approaches is shown in figure 2.3. Figure 2.3a shows a selection process for tight operators whereas figure 2.3b (adapted from [Laborie and Godard, 2007]) shows the process for loose operators.

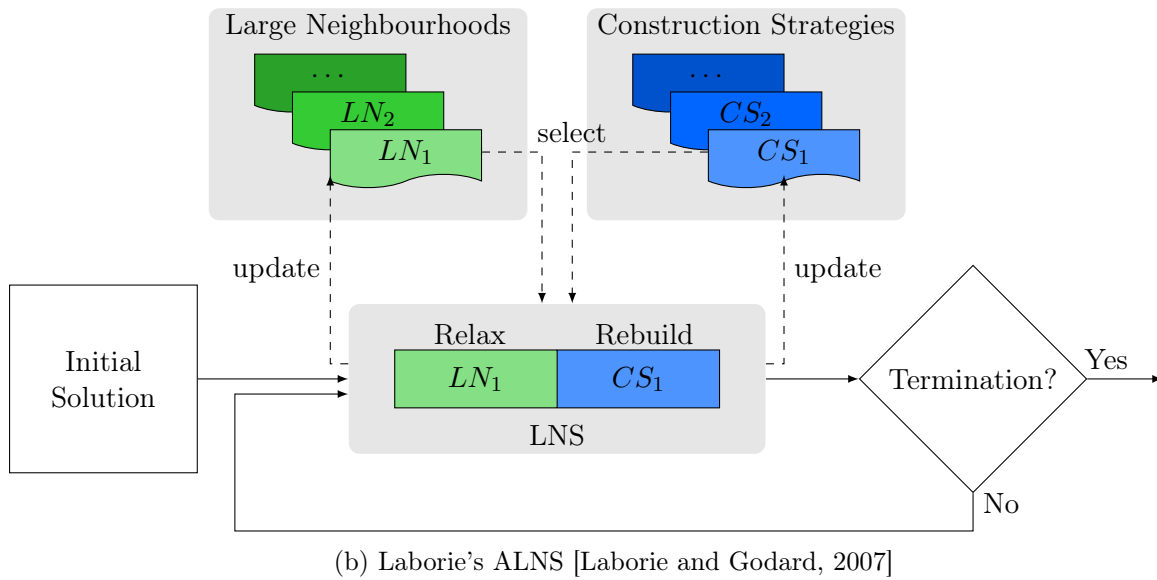
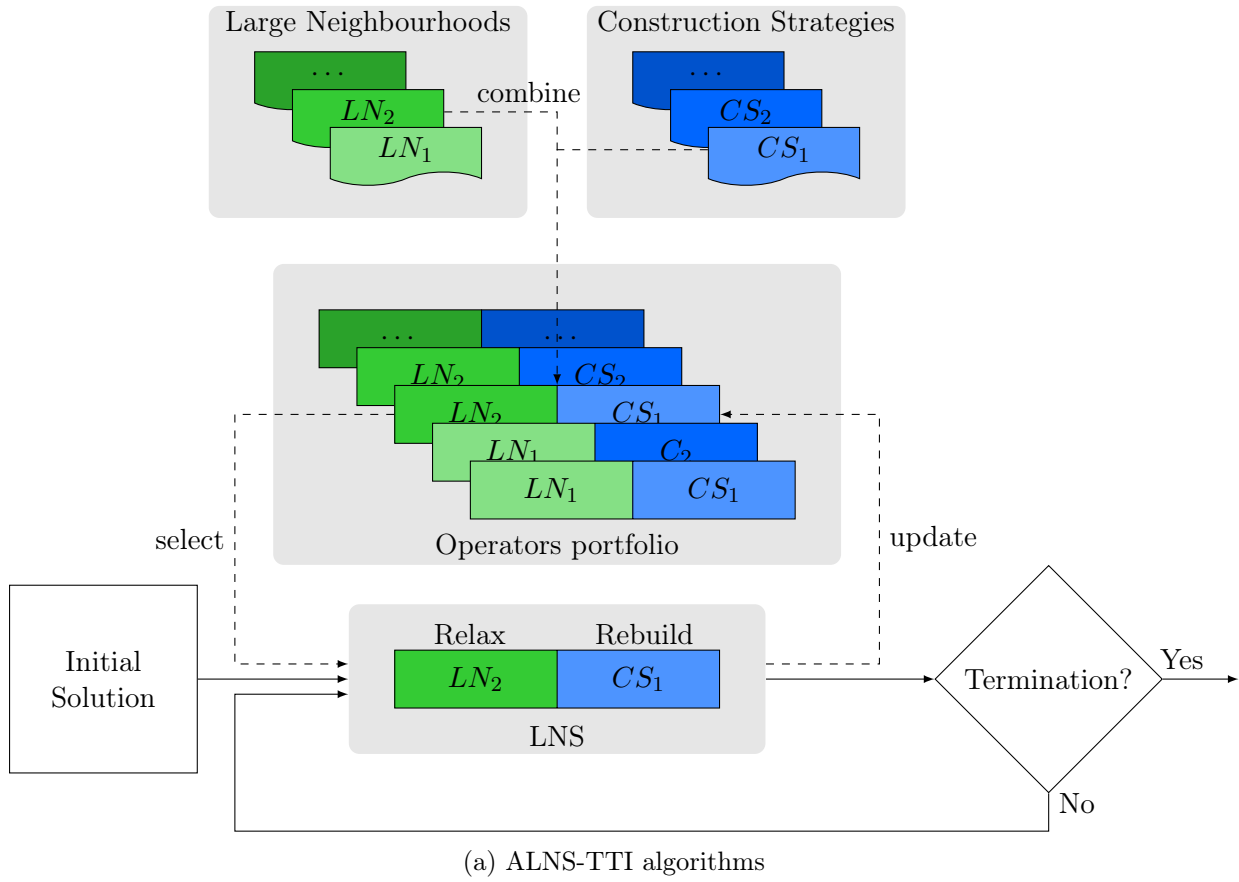


Figure 2.3: Flowcharts highlighting the selection and update process differences between the ALNS-TTI class of algorithms and Laborie's ALNS.

Operator definition

For the rest of this thesis, it is important to rigorously define what is meant by an operator and its associated metrics. An operator is composed of a pair of exactly one *LN* and one *CS* heuristics, both with fixed parameter values. We can then define, for each operator, the following statistics and metrics:

- The *runs* are the amount of executions of the operator, i.e. the number of LNS iterations where it was used.
- The *time* is the total time (in seconds) the operator has been running, aggregated on all its runs.
- The *average time* is the ratio between the total running time and the number of runs.
- The *solutions* are the total amount of new best feasible solutions found by the operator during its runs.
- The *successful runs* are the amount of runs which resulted in at least one solution found.
- The *success rate* is the ratio between the successful runs and the total amount of runs.
- The *time to improvement (TTI)* is the ratio between the average time and the success rate, which corresponds to the expected time to find a solution.

Chapter 3

Implementation

This chapter will introduce our implementation of:

1. Our VRP variants in a first section.
2. Our ALNS algorithms in a second section.

We will then show how both can be combined, together with some given heuristics, to solve the vehicle routing problem with a self-adapting large neighbourhood search.

As mentioned before, we will be using Constraint Programming, as it provides a natural way to implement LNS and VRP. More precisely, we will be using the **OscAR** library [OscAR Team, 2016].

3.1 Goals

In terms of software, the goals of our implementation were to provide a modular framework able to combine different kinds of vehicle routing variants, as well as a proof of concept for an ALNS mechanism that was simple to use, concise and clean to read.

3.2 VRP

3.2.1 Models

Since each of our VRP variants is an extension of another problem (e.g. CVRP extends VRP by adding a load constraint), it makes sense to use this hierarchy to our advantage in our software design, especially since one of our goals is to provide a modular model of the different vehicle routing problems.

We will start by defining a **VRPModel** class modeling the base problem. We will then use **Scala** traits to define more constrained sub-problems. This pattern allows us to easily implement model-specific constraints for each variant of VRP in its own trait, and then combine them together depending on the problem we are looking to solve. Figure 3.1 shows a class diagram of this design, as well as the global architecture of the system. We can see how each problem variant is implemented independently leading to little to no code redundancy and great modularity. Depending on the problem instance, the parser will build an adequate **Instance** object with the appropriate traits, which define the right data fields depending on whether the instance fits a problem with demands, time windows, or anything else. It is then the responsibility of the user to feed this instance to a model with compatible traits.

For instance, if we want to solve a simple VRP, capacitated VRP with time windows, or a pickup-and-delivery with time windows, we can use the appropriate line shown in listing 1.

Listing 1 Combining traits to define a model suited to a specific problem

```

1 val model = new VRPModel(instance) // or
2 val model = new VRPModel(instance) with CVRPModel // or
3 val model = new VRPModel(instance) with PDPMModel with VRPTWModel

```

VRPModel

The goal is to implement the constraints listed in section 1.3.1. Listing 2 provides our model in OsaR. We use a weak propagator for the `minCircuit` which uses the circuit constraint as defined in [Benchimol et al., 2012] as well as a sum constraint to keep track of the objective. We will use the weak propagator to keep reasonable execution times during propagation (stronger propagators, which use the min assignment constraint [Focacci et al., 1999] and the Held-Karp algorithm [Held and Karp, 1962] are a bit too slow for our purposes). This constraint will keep track of the objective value defined in equation 1.13. The inverse constraint makes sure that the two *succ* and *pred* arrays are consistent (see constraint 3 in section 1.3.1).

Listing 2 OsaR VRP Model

```

1 add(minCircuit(succ, distances, objective, false), Weak)
2 add(minCircuit(pred, distances, objective, false), Weak)
3 add(inverse(succ, pred)) // link predecessors and successors
4
5
6 for (i <- N ++ S)
7   add(elementVar(route, succ(i), route(i)), Strong)
8
9 for (i <- N ++ E)
10   add(elementVar(route, pred(i), route(i)), Strong)
11
12 for (k <- M) {
13   // start/end node of each vehicle is a route of that vehicle
14   add(route(S.head + k) == k)
15   add(route(E.head + k) == k)
16 }
17
18 // successor of end node is the next start node
19 for (k <- 0 until nVehi - 1)
20   add(succ(E.head + k) == S.head + k + 1)
21 add(succ(E.last) == S.head)

```

CVRPModel

Adding a load constraint is particularly easy, implementation-wise: we will use a bin packing constraint to enforce that the load of each route does not exceed the total weight. In other words, each route is a bin, each client has a weight (demand) and the sum of demands for a given route

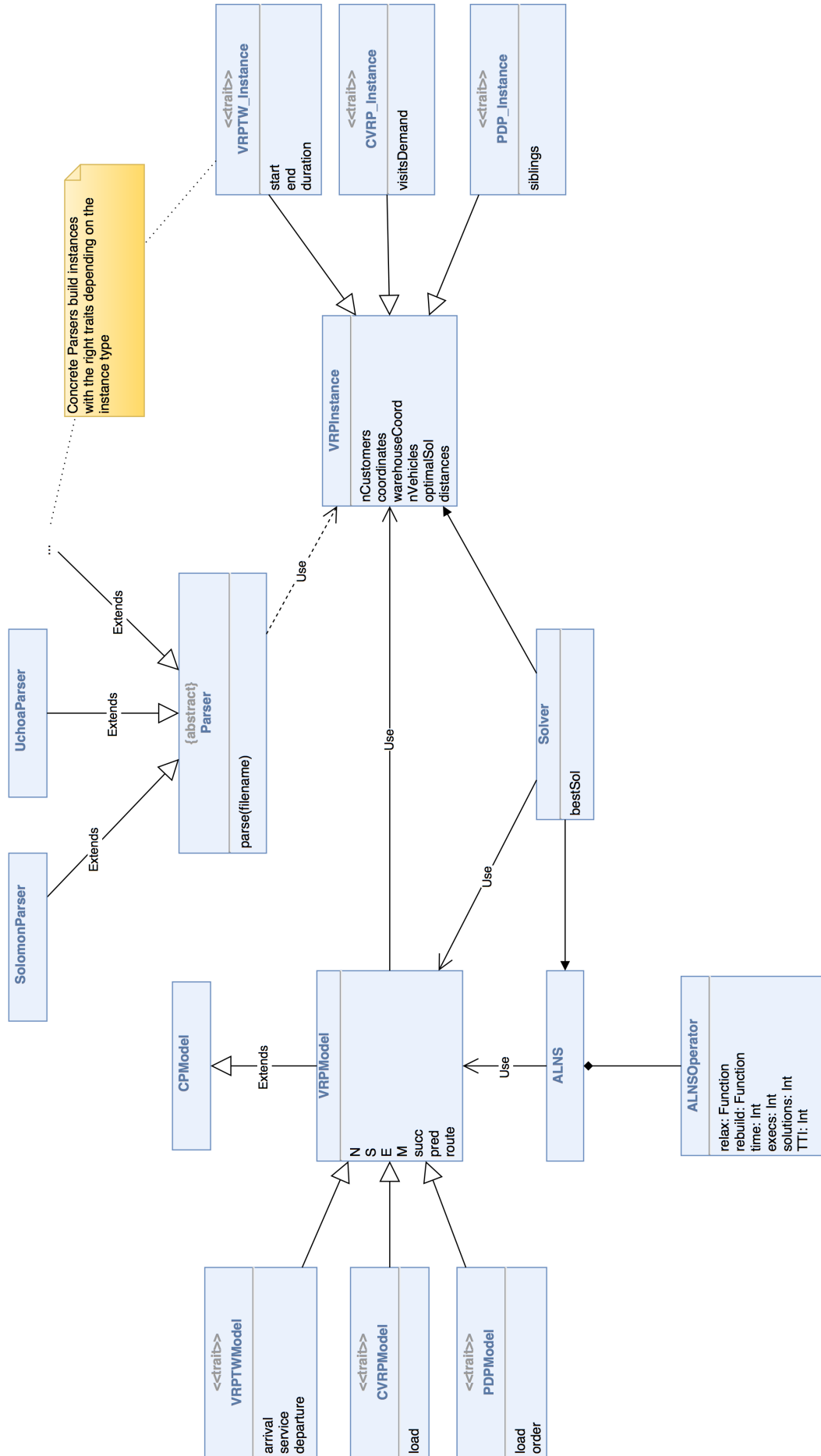


Figure 3.1: Class diagram of the implemented software architecture

must be lower than the vehicle capacity. We will make use of *Oscar*'s `binPacking` constraint which uses the propagation and lower bound presented in [Shaw, 2004] No other addition is required to the original VRP problem.

Listing 3 *Oscar* CVRP Model

```
1 add(binPacking(route, demands, load), Strong)
```

VRPTWModel

In the time-windows variant of the problem, we must enforce that the truck arrives at each customer before the end of the time window in order to serve the customer in their allowed time. The following implementation closely follows the model described earlier.

Listing 4 *Oscar* VRPTW Model

```
1 for(i <- S)
2   add(departure(i) == limitTimes._1)
3
4 for(i <- N){
5   add(service(i) >= arrival(i))
6
7   // Serve during time window
8   add(service(i) >= starts(i))
9   add(service(i) <= ends(i))
10
11   add(service(i) + serviceTimes(i) == departure(i))
12 }
13
14 // Arrival time at B is departure time from A + travelTime between A and B
15 for(i <- N ++ S)
16   add(arrival(succ(i)) == departure(i) + travelTimes(i)(succ(i)))
```

PDPModel

The following defines an *Oscar* model for the pickup and delivery problem. In this problem, we can no longer use the `BinPacking` constraint as some weights may be negative (for delivery locations).

Listing 5 OsaR PDP Model

```

1  for(i <- S) {
2      add(load(i) == 0)
3      add(order(i) == 0)
4  }
5
6  for(i <- N ++ E) {
7      add(load(i) == load(pred(i)) + demands(i))
8      add(order(i) == order(pred(i)) + 1)
9  }
10
11 for(i <- E)
12     eeadd(load(i) == 0)
13
14 for(i <- N) {
15     add(route(i) == route(siblings(i)))
16     if(demands(i) > 0)
17         add(order(i) < order(siblings(i)))
18
19     add(load(i) <= vehiCapa)
20 }

```

3.3 ALNS

In this section, we will first introduce a generic ALNS CP framework, and show how calling code can leverage this framework to solve any kind of CP model using an ALNS algorithm. Then we will instantiate this framework with a first algorithm taken from the literature [Laborie and Godard, 2007], which will serve as a performance reference, and then with our own algorithms.

The adaptive LNS facilities related to ALNS-TTI are located in the classes of the `alns` package while our implementation of the algorithm described in [Laborie and Godard, 2007] is in the `laboriealns` package.

3.3.1 Presentation of the framework

The main purpose of our development process was to produce a framework that was simple, concise and readable.

Every implementation follows the same framework: an ALNS class receives a collection of `ALNSOperators` and selects, at every iteration, which operator should be executed among this collection (depending on the selection mechanism implemented by the algorithm). The operator is then applied before searching using OsaR's `startSubjectTo` method. Listing 6 shows the main generic framework.

An `ALNSOperator` is seen as a functional object, and the class defines a simple wrapper around a function which is applied using Scala's `apply` mechanism. The wrapped function is in charge of

1. Applying the relaxation (define a large neighbourhood) by adding constraints which correspond to fixing some variables to their value in the current best solution

Listing 6 OscaR implementation of a Generic ALNS Framework

```

1 class ALNS(val model: CPModel, val operators: Iterable[ALNSOperators]){
2
3     /*... */
4
5     while(/* termination criterion not met */){
6         val op = select(operators)
7
8         val stats = startSubjectTo(/* LNS Iteration termination criterion */){
9             op()    // Apply LNS operator
10        }
11        op.update(stats)
12    }
13 }

```

2. Setting up the search strategy to be used by OscaR during this exploration of the neighbourhood, by way of OscaR's `search{}` method.

3.3.2 Simple usage example : QAP

Assuming we have an implementation of the presented framework for any given adaptive LNS algorithm, this section will present how it can be used to solve an example problem, namely the Quadratic Assignment Problem (QAP).

QAP is a well known combinatorial optimization problem wherein we are given n facilities and n locations. Each couple of locations are separated by a given distance, and each couple of facilities are associated with a given weight. We want to assign each facility to exactly one location, with minimal cost, where the cost is the product of the weight between two facilities by the distance between their locations.

A small example solving an instance of QAP (instance Chr12a taken from [Anjos, 2012] [Christofides and Benavent, 1989]) is given in the `QuadraticAssignment.scala` file. Listing 7 gives a simplified overview of how to use the ALNS framework. The model for the Quadratic Assignment Problem is largely copied from [Schaus, 2016a].

Notice how relatively easy it is to define a new operator: just instantiate the `ALNSOperator` class with a functional parameter which defines a large neighbourhood (here: keep a random 50% of the current best solution) and sets up an OscaR `search`.

In the actual `Scala` file solving QAP, the operators are generated dynamically with varying rates of relaxation. In just a dozen of lines we can create many operators and start the ALNS search.

3.4 Implementation of Laborie

The algorithm in [Laborie and Godard, 2007], which we will call LALNS (or Laborie's ALNS), will be our reference algorithm. We will compare all our results with the performance obtained using LALNS.

To implement it, we must first implement a generic roulette wheel probability adaptation mechanism. This is done in the `utils.RouletteWheel` class.

Listing 7 Solving QAP with the ALNS framework

```

1 object QAP extends App with CPMModel {
2   // CP Variables
3   val X = Array.fill(n)(CPIntVar(0 until n))
4   val D = Array.tabulate(n, n)((i, j) => element(d, X(i), X(j)))
5
6   // Constraints and objective
7   add(allDifferent(X), Strong)
8   minimize(sum(0 until n, 0 until n)((i, j) => D(i)(j) * w(i)(j)))
9
10  // Keep solution
11  onSolution{
12    (0 until n).foreach(i => bestSol(i) = X(i).value)
13  }
14
15  val op1 = new ALNSOperator({
16    // Relax 50% of the nodes
17    add(Random.shuffle((0 until n).toList).
18      take(n/2).map(i => X(i) == bestSol(i)))
19
20    // Use BinaryFirstFail
21    search{BinaryFirstFail(X)}
22  })
23
24  /* Define op2 ... */
25
26  val ALNS = new ALNS(this, List(op1, op2))
27  ALNS.start()
28 }

```

In the case of this algorithm, the operators are loosely coupled, which means the class will receive two collections of functional arguments. The `LALNS` class will maintain the two probability distributions for each set of heuristics. Listing 8 implements this algorithm based on the generic framework presented in listing 6.

The two roulette wheel probability distributions are maintained in `ln_wheel` and `cs_wheel`. The `adapt` method of the roulette wheel takes a factor and adapt the weight (and therefore the probability) of the last selected item using the formulas given in equations 2.3 and 2.4.

The `LALNSOperator` class will be slightly different from the simple wrapper that was presented before since we must keep a roulette wheel for every parameter of the operator.

3.5 Implementations of ALNS-TTI

We will now describe the implementations of the different variations of our algorithm, called ALNS-TTI. The three variants are:

1. Simple ALNS-TTI
2. ALNS-TTI with Learning
3. Probabilistic ALNS-TTI

Listing 8 Implementation of LALNS using the framework shown in 6

```

1 class LALNS(val model: CPModel, val ln_operators: Iterable[LALNSOperators],
2             val ls_operators: Iterable[LALNSOperators]){
3
4     /*... */
5
6     while(/* termination criterion not met */){
7         val ln = ln_wheel.select()
8         val cs = cs_wheel.select()
9
10        val stats = startSubjectTo(/* LNS Iteration termination criterion */){
11            ln()    // Relax into a large neighbourhood
12            cs()    // Rebuild with construction strategy
13        }
14
15        ln_wheel.adapt(costImprovement / stats.time)
16        cs_wheel.adapt(costImprovement / stats.time)
17
18        ln.update(stats)
19        cs.update(stats)
20    }
21 }

```

3.5.1 Simple ALNS-TTI

This is the simplest version of the algorithm. It is a straightforward implementation of the algorithm given in 4. In this version of ALNS, the selection mechanism simply selects the operator with the lowest TTI of all. This is done through a priority queue which orders the operators based on their TTI. This gives us something to the order of what is shown in listing 9.

The operators originally have equal TTI, successRate and executions statistics. This implies that each operator has an initial TTI of 0, and all will be run once when the algorithm start. As we will see, this is probably the biggest issue of this algorithm. Initially tie-breaking them using a random number allows us to introduce some diversification at first (which would not be the case if the operators were sorted alphabetically, for instance).

Checking for a bias towards early operators

One problem we suspected with this approach is that the operators do not get the same fair chance of gathering statistics. An operator which is selected early will try to improve the solution, and if it does, the next operator will try to find an improvement *on this new solution*, and so on. This means that the problem becomes harder and harder, and the operators which are lucky to be selected first will get good statistics early and will tend to be selected more often afterwards because they had the chance to improve an easy solution - whereas a good operator may not be ever tried again because its first run started with a very difficult solution. Perhaps more intuitively, the more we improve our solution, the less likely we get of improving it again later as the problem gets harder.

Figure 3.2 gives a bar chart illustrating this intuition. On the X axis, all the operators used for a single instance are laid out, in their order of first selection by the algorithm, from left to right. This means that the leftmost column correspond to the operator selected for the first

Listing 9 Simple ALNS-TTI implementation

```

1 var operators = new PriorityQueue[ALNSOperator]() (Ordering.by(x =>
2   (-x.timeToImprovement, x.successRate, -x.execs, Random.nextInt)))
3
4 ops.foreach(x => operators.enqueue(x))
5
6 for (i <- 1 to iterations) {
7   val op = operators.dequeue()
8
9   val stats = startSubjectTo(/*LNS iteration termination criterion */){
10    op()
11  }
12
13  op.update(stats)
14  operators.enqueue(op)
15 }

```

iteration, and the rightmost column to the operator selected last in the first phase were the algorithm tries every operator once.

The right axis gives their total executions at the end of the algorithm. The dashed blue line indicates the 50% percentile of the total number of iterations (a little over 800 of a total of a little over 1600, as indicated by the green line). This means that the operators to the left of the dashed blue line represent 50% of the total iterations performed by the algorithm. If the selection scheme were fair, and taking the operators in a random order (as we do for the first run of all operators, since they are originally shuffled), we would expect to see roughly half of the operators representing these 50% of iterations. However, the chart seems to indicate a proclivity to the left, which would confirm our suspicion of a bias towards selectors occurring early in the original shuffled order the algorithm starts with.

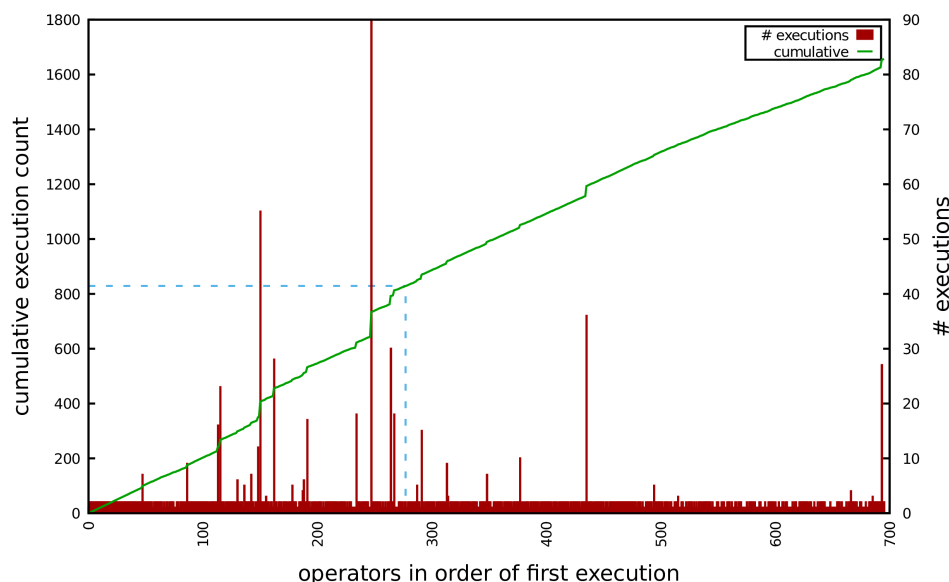


Figure 3.2: Number of executions for operators, sorted by time of first execution, for one instance

But this effect could be a product of chance, and to verify this intuition formally, we will compare the most selected operators with those which are first in the order of the input operators, originally shuffled. Formally, we want to determine if the elements in the operators selected early

Rank	Operator	Executions
0	Path&Rel(8) - Greedy	1,090
1	Related(3) - Regret(3)	409
2	Path&Rel(8) - Regret(3)	257
3	Random(8) - DGreedy	123
4	Path&Rel(13) - Regret(3)	117
5	Related(8) - DGreedy	95
6	Related(13) - Regret(3)	87
7	Worst(13) - Regret(3)	75
8	Worst(13) - Regret(2)	75
9	Worst(23) - Greedy	52
10	Related(13) - Greedy	51
11	Path&Rel(18) - Greedy	49
12	Related(13) - Regret(2)	47
13	Related(23) - Greedy	41
14	Random(18) - Greedy	36
15	Path&Rel(38) - Regret(3)	32
16	Path&Rel(38) - Greedy	32
17	Path&Rel(43) - Greedy	30
18	Related(38) - Regret(2)	26
19	Related(38) - Greedy	23
20	Worst(18) - Greedy	23
21	Path&Rel(13) - Greedy	23
22	Path&Rel(3) - Regret(2)	23
23	Random(3) - DGreedy	23
24	Worst(33) - Regret(3)	23

Table 3.1: Top 25 most selected operators for Simple ALNS for X-n120-k6.vrp

by the algorithm (let us call this subset of operators S , such that $S \subset O$) are over-represented (to an unlikely degree probability) in the subset of the most often selected operators ($T \subset O$)

The total number of operators ($|O|$) in our implementation, for a 120-node instance (see details later in section 3.6.2) is 190. Since the algorithm periodically re-runs all heuristics once (see below), most of the worst performing heuristics will have a total run number equal to this number of re-runs. In other words, all the least-used operators at the tail of the list will have an equal number of selections, while some other operators will be at the top of the list with a larger number. See for example table 3.1 where the top-25 of operators after a run of the algorithm on one instance is given. All subsequent operators have been selected exactly 23 times.

If we consider the probability that a randomly selected operator is included in the top 25 most selected operator (T), we get a probability of 25/190. Given k random operators, the probability that they are *all* in T is given by a hypergeometric distribution [Bogaert, 2006].

Intuitively, this distribution is a binomial distribution without replacement. More formally, it describes the probability of observing exactly k successes in n random draws without replacement from a population of N elements, containing K possible successes. This probability is given by:

$$P(X = k) = \frac{\binom{K}{k} \binom{N-K}{n-k}}{\binom{N}{n}} \quad (3.1)$$

In our case, we have $N = 190$ operators, of which $K = 25$ are the most selected. We will

select the $n = 25$ first elements of the shuffled list of operators that the argument takes as input (S), and we will look how many successes (k) we get, i.e. how many elements of S are also in T . Experimentally, for the instance shown above for example, we obtain only $k = 2$ successes. From this number of successes and the described distribution, we can infer the probability of obtaining the observed number of elements in $T \cap S$ or more, if no bias between the two subsets of operators existed. This probability ($P(X \geq k)$) is a p-value for a hypergeometric test, checking for over-representation of successes in the sampled population.

However, we find that the bias hypothesis is ruled out by this test, since we obtain a probability and p-value of 0.67. This means that a random draw of operators is not unlikely to give this outcome of 2 elements present in T . We cannot conclude that the first selected operators are over-represented to an unlikely high probability in the most selected operators. In fact, we can expect that on average, about 3 operators will be present in both lists if no bias exist between the two, because the mean of a hypergeometric distribution is given by $\frac{nK}{N} = 25 \times 25/190 \approx 3.29$.

Problems

A problem with this algorithm is, as mentioned, that the operators start with equal statistics, as we do not know how well they perform originally and we have no way of estimating their TTI. We are therefore constrained to run each operator once at the beginning to get some initial values for the TTI. The time required to run all heuristics in the first place can be tremendous and make the whole algorithm virtually unusable for all intents and purposes when the number of operators is large and each of them take a non-negligible time to execute (which heavily depends on the difficulty of the problem instance). For a difficult problem and many operators, it is simply too impractical to waste so much time trying all the different operators at first.

We also have to determine a scheme to attribute a value to operators which did not find a solution during this first run (or even subsequent runs, for that matter): we cannot compute a TTI value (remember that this is the ratio of time over success rate) if the success rate is 0! Our solution is to give these operators the worts existing TTI of the set. As such, these operators end up at the end of the queue but with a TTI which is still finite, and they get a chance of being re-executed again if the other operators see their TTIs increase. This is not an undesired effect since we want to give operators a second chance in case their first run was unlucky, or they perform better on another kind of base solution. But it creates another problem, which is that periodically, the algorithm will re-run every operator once! Since the TTI of effective operators grows and grows as time passes, it eventually overcomes that of these “tail operators” which have yet to find a solution, and so all these tail operators will get re-executed.

3.5.2 ALNS-TTI with Learning

This variant of the algorithm originated as an improvement to the original ALNS-TTI algorithm. Since at the beginning of the execution of the program, all operators have equal statistics, the algorithm necessarily runs each and every operator exactly once to gather some first statistics to initiate the process. However, as noted above, we could suspect that the operators selected early get a better chance of obtaining good statistics as they are run on a simpler problem, which is easier to optimize.

We wanted to find a way to gather fair statistics on all operators at the beginning, to improve the selection mechanism: the more data it gets, the more likely good operators are to be selected. To do this, we introduced a learning phase at the beginning of the algorithm. The idea is that since we are going to run every operator once anyway, why not start by *not* improving the

solution, and run them on the initial solution first. This way the algorithm can learn some nice performance measures on the heuristics and improve its selection mechanism.

The outline is similar to what is given in algorithm 4, except that, during the learning phase, we do not store any improvement found over the initial solution. Only when the learning phase is over do we start actually improving on the solution.

We also take advantage of the learning phase to filter the operators set and drop those performing particularly poorly. In our implementation, we chose to drop the 50% worst operators, but this is of course very dependent on the operators and how many of them are given. One could imagine other, more sophisticated filtering criteria.

Problem

This approach does usually provides a nice and quite fast convergence after the learning phase (see the results section for some details). However, it suffers from the same problem as the first version of the algorithm: it requires us to run all the operators at once at the beginning of the algorithm.

Link to Multi-Armed Bandit algorithms

The problem we are attempting to solve (selecting the operator most likely to improve the solution fast) can be seen as an instance of the *Multi-Armed Bandit problem*. In this problem, a set of actions (here: operators) each have an unknown probability distribution of giving a reward. The problem is to figure out which action to take at each iteration to maximize the sum of obtained rewards. In practice however, most algorithms to solve this problem in fact try to minimize the *regret* of the used strategy, i.e. the difference between the expected reward of playing the best action and the expected reward of our strategy.

One popular and simple algorithm to solve this problem is called UCB1. If we have N actions, each of which has an estimated reward $\bar{\mu}_j \in [0, 1]$ and a number of executions n_j , the UCB1 algorithm works as follows:

1. Run each action once to obtain a first approximation of $\bar{\mu}_j, \forall j \in N$
2. For each iteration $i = N, N + 1, \dots$:
 - (a) Select and run the action $a = \operatorname{argmax}_j (\bar{\mu}_j + \sqrt{\frac{2 \ln i}{n_j}})$
 - (b) Observe the obtained reward and update $\bar{\mu}_a$ accordingly.

Without going into details outside the scope of this thesis, [Auer et al., 2002] proved that the maximized expression above achieves a near-optimal regret. However, with the multi-armed bandit problem, the distributions are generally assumed to be stationary, which (as we discussed earlier) is not really the case in an optimization problem where improving on the solution becomes less and less likely. Another problem with this approach is that it does not solve the problem of initialization, i.e.: how to avoid running each operator once to estimate a first value for the $\bar{\mu}_j$. This class of problems could be interesting to consider in the context of ALNS, but the straightforward application of an algorithm like UCB1 falls in the same problems as Simple ALNS-TTI and ALNS-TTI with Learning if we use tight operators.

One possible solution is the following: use loose operators and a probabilistic selection for the parameter values inside. With this approach, we can run each couple of heuristic together to get a first approximation of the UCB reward, and the parameters, which can cause the number of

operators to explode when using tight operators, are confined inside the heuristic and selecting them with a probabilistic selection avoids having to try each and every one of them.

Such an algorithm has been implemented in our solution (class `UCB1`). We use the selection scheme and reward system from [Laborie and Godard, 2007] for the parameter values, but select the heuristics based on the UCB1 criterion.

3.5.3 Probabilistic ALNS-TTI

To solve this problem of initialization, we tried, in our third variant of the algorithm, to use a probabilistic approach, using a selection mechanism similar to what is used in both [Laborie and Godard, 2007] and [Ropke and Pisinger, 2006].

The idea is to use a roulette wheel, but to reward operators based on their TTI. We want to reward operators with a small TTI, and so we would like to use a reward factor (see equation 2.3) which is inversely proportional to the TTI of the operator. Some possibilities, amongst others, include:

- $\frac{1}{TTI}$
- $\frac{TimeLimit}{TTI}$, to reward operators proportionally to how many solutions we expect them to find in the overall allowed time.
- $\frac{RemaningTime}{TTI}$, to reward operators based on how much time remains to improve the solution. The idea is to take into account how much time remains versus how long this operator usually takes to find a solution.

These possible factors have some advantages and drawbacks, with the first having the significant issue of being < 1 . We opted to use the second factor as it seemed to give the best results, empirically. It is also a sensible metric: we want to reward operators based on how many solutions we expect them to find in the whole run. The third possible factor might also introduce a bias towards early selected operator, since the remaining is of course strictly decreasing.

3.6 Applying ALNS to VRP

In this section, we will combine all the elements presented above to finally apply a self-adapting large neighbourhood search to the vehicle problem.

At this point, we have a CP model of VRP and several of its variants. We also have a generic framework for an adaptive LNS, as well as several implementations following this framework. But as mentioned earlier, LNS requires three inputs:

1. An initial solution
2. A set of large neighbourhood heuristics
3. A set of construction strategy heuristics

We will therefore describe how we obtained and combined these elements in our implementation.

3.6.1 First solution

The first problem is to find the first solution. In 1.2.2, we presented several possible ways of finding a first approximation for VRP, but in our specific case we required several criteria for the first solution:

- We wanted this first solution to be “deterministic”, in the sense that we needed each algorithm to be fed the same first solution so that the comparisons we would perform would make sense and be fair.
- We also wanted this first solution to be computed relatively fast: we could not afford to waste a lot of time computing a top-notch first solution because it was important that we could run tests on a substantial number of instance, and the time spent computing a good first solution was time that was not spent testing our algorithms of interest.
- But we also wanted this first solution to be non-trivial, and have a starting cost which wasn’t unreasonably high. This was because we felt a real application would probably start with a good estimate of the first solution, and starting with a bad first solution would make us spend many iterations reaching a point where optimizing becomes hard, which is where the real deal is and our area of interest.

Our choice was a simple greedy algorithm (closest neighbour). This option also had the added advantage of being quite easily implementable in constraint programming. Specifically, we ask the solver to assign the `succ` CP variables in order of their index, selecting as value the index of the closest geographical node which is not a virtual depot node (unless no other possibility is available). See figure 3.3 for an example of a first solution.

3.6.2 Operators

Having determined a way of obtaining the first solution to feed the LNS process, we must now build a set of operators to apply on this first solution and its subsequent improvements.

If we refer to the heuristics listed and described in 1.2.3, we will use the following:

Relaxation heuristics

- Path and related visits
- Related visits
- Worst visits
- Random visits

Construction strategies

- Greedy
- Deeply greedy
- Regret-2
- Regret-3

Having selected these heuristics, we must now use them to instantiate some LNS operators. To do this, we must provide the relaxation heuristics with some parameter values: all of the provided relaxations take an argument giving a number of nodes to relax in the solution. We chose the following relaxation range: we relax from 3 nodes to 50% of the amount of nodes in

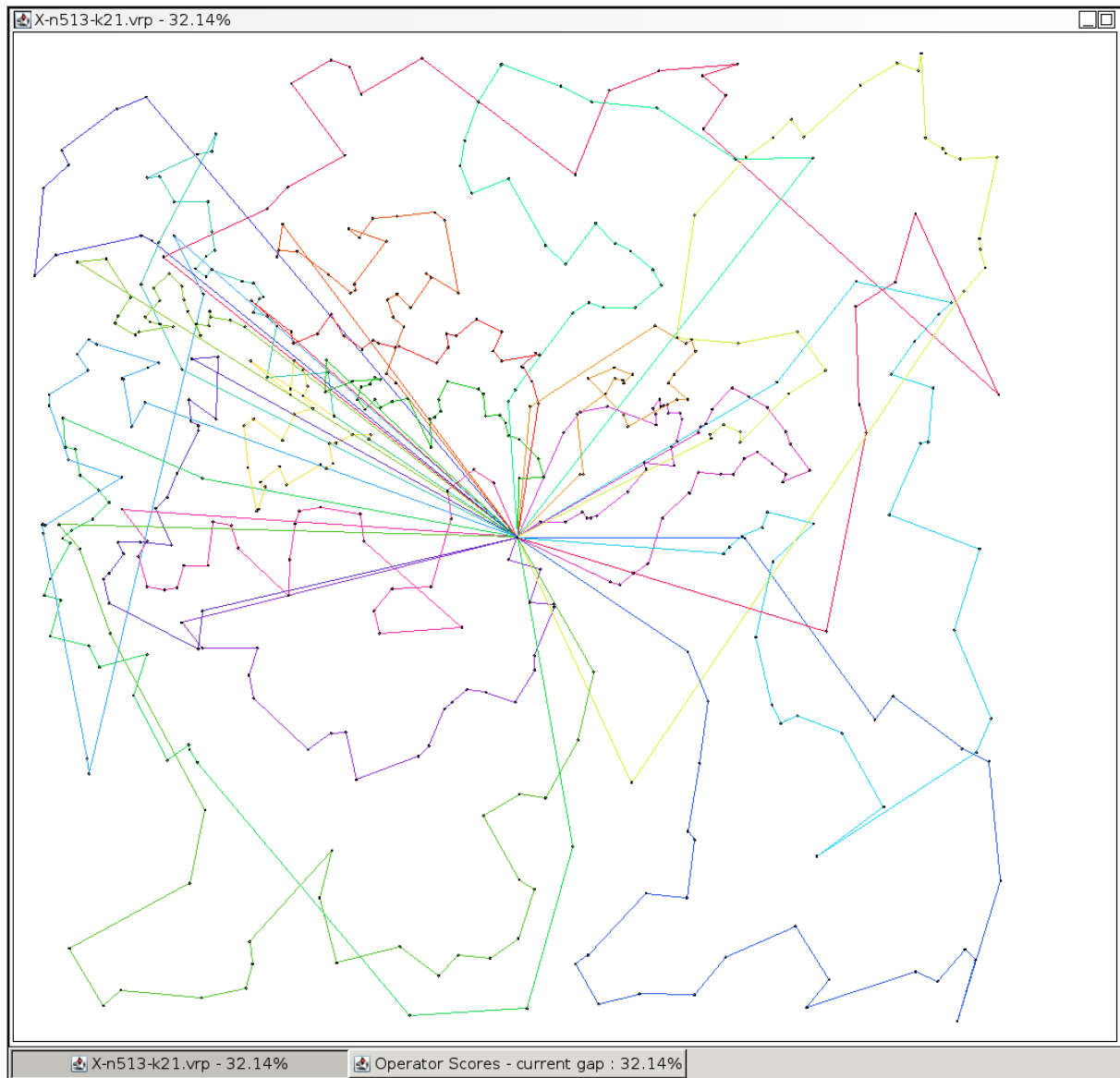


Figure 3.3: Example of a first solution obtained by the program with a greedy algorithm (X-n513-k21, 513 visits, 21 vehicles)

the problem, by increments of 5%. For example, for a 200 node instance, we get the following amounts of relaxed nodes: 3, 13, 23, 33, 43, 53, 63, 73, 83, 93.

which gives us ten different relaxation rates. Each relaxation heuristic will be instantiated with each of these possible arguments, giving a total of 40 different relaxations. Each of these will in turn be coupled with each of the 4 construction strategies, giving a grand total of 160 operators for a 200-nodes instance.

3.7 Final product

The obtained product consists in a program which can solve and display an instance of different kinds of variants of the vehicle routing problem. Modularity is achieved with a clean construct, and the ALNS algorithms are encapsulated in classes which may be reused painlessly.

Concerning the ALNS-TTI variants of the algorithm, we provide a GUI allowing to visualize

the current best solution, as already shown in figure 3.3. The routes are updated every time a new solution is found by the solver. We also provide a table visualization of the operators, displaying their statistics and allowing the user to see the evolution of the scores as well as which operators were used the most, etc. Figure 3.4 shows an excerpt of this table, during a particular run, for the operators sorted by number of executions.

Operator Scores - current gap : 9.86%						
Relax	Rebuild	Sols	Execs	Success Rate	Avg. Time	TTI
Path&Rel(9)	Greedy	107	1280	0.047	30.968	681.983
Path&Rel(9)	Regret(3)	5	41	0.049	32.488	686.5
Related(3)	DGreedy	2	21	0.048	32.286	699
Random(9)	Regret(2)	1	19	0.053	36.263	708
Worst(9)	Greedy	1	18	0.056	37.444	692
Path&Rel(9)	DGreedy	1	16	0.062	42.5	696
Random(9)	DGreedy	1	14	0.071	48.857	698
Related(15)	Regret(3)	2	12	0.083	59.75	729
Worst(9)	DGreedy	3	11	0.091	64.182	717
Related(9)	DGreedy	3	10	0.1	69.8	708
Path&Rel(15)	DGreedy	4	3	0.333	234.667	707
Path&Rel(27)	Greedy	1	3	0.333	253	762
Path&Rel(33)	DGreedy	0	2	0	283.5	681
Path&Rel(57)	Regret(2)	0	2	0	329	681
Random(21)	DGreedy	0	2	0	17.5	681

Figure 3.4: Operators visualization

Chapter 4

Experiments and results

This chapter will present our methodology to evaluate :

1. The performance of our algorithms, and how close they get to the known optimal solution for some standard instances
2. The performance of our algorithms compared to a strictly random selection, i.e. are the described selection mechanisms more efficient than selecting a random operator at every iteration
3. The performance of our algorithms compared to Laborie's ALNS

As well as the obtained results and their interpretation.

4.1 Methodology

4.1.1 Instances

The implementation contains parsers for several canonical instances taken from the literature, namely:

1. Sets A, B and P from [Augerat et al., 1995]
2. Set X from [Uchoa et al., 2014]
3. Sets 25, 50 and 100 from [Solomon, 1987] (for the CVRPTW)
4. Sets 100, 200 and 400 from [Li and Lim, 2003] (for the PDPTW)

For the following results and experiments, we will mostly be using the X dataset. This dataset contains instances for the CVRP which have the advantage of not being computer-generated, and have varying warehouse locations as well as different customer location configurations :

- The customers are sometimes grouped into clusters spread over the map.
- The customers are sometimes distributed in a seemingly random cloud of points, with the warehouse located at the center (see figure 1.1) or somewhere else.
- The customers may be distributed together but all far away from the depot (see figure 4.1).
- And many other configurations.

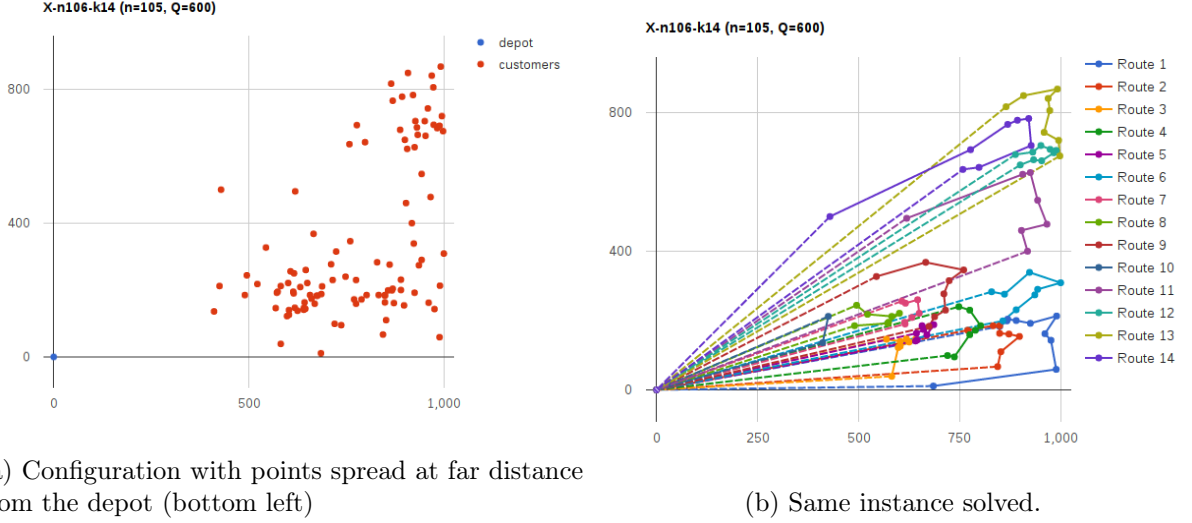


Figure 4.1: An example of map configuration from the X dataset (X-n106-k14, 106 visits, 14 vehicles) [CVRPLIB, 2016]

This dataset is very generic and contains many instance types, which gives us a more realistic approach where our algorithms are used in different contexts which match distinct but actual real-life situations. Specifically, we will use the 30 first instances (up to 21 vehicles).

4.1.2 Parameters

Common heuristics and models

In order to obtain fair and comparable numbers for all algorithms involved, it is important to specify the possible parameter values and run each test with the same circumstances so that every algorithm gets the same tools to try and improve the solution. Only then can we legitimately compare these algorithms together.

For this reason, all the tests we ran use exactly the same set of heuristics, with a shared implementation. The sets of possible parameter values are always strictly identical and the VRP models are all the same. The construction strategies are implemented independently, as are the relaxation methods, which are always applied strictly with the same piece of code, regardless of the ALNS algorithm used.

Our initial solution also uses a strictly deterministic algorithm for this reason : each tested implementation receives the same first solution and we can compare honestly how each selection mechanism improve on this solution. Our final choice for the first solution was to use a simple nearest neighbour heuristic to find the first solution to apply ALNS to. This heuristic quickly finds a starting point with a quite good gap value (less than 50%), for all Augerat instances (of size 100 or less) and several Uchoa instances (of size 100 or more). ALNS is therefore able to gather statistics on the operators when the problem is already hard to solve.

Reaction factor α

Both Laborie's ALNS and our own Probabilistic ALNS-TTI use a roulette wheel selection principle. We have seen that this mechanism requires the choice of a parameter α , the reaction factor. The choice of this reaction factor is of course crucial and could warrant some research

of its own, possibly with some statistical learning to determine a good value for this parameter. In [Ropke and Pisinger, 2006], the authors perform some parameters tuning and obtain a value of 0.1. In our implementation, we obtained a value of $\alpha = 0.2$ which seemed to work well in practice. In any case, since our goal was not so much to find near-optimal solutions but more to compare selection mechanisms between them, the most important aspect is that we use the same value both for Laborie’s ALNS and our Probabilistic ALNS-TTI. Readers interested in increasing the performance of either algorithm may perform some more elaborated parameter tuning depending on the application and context in which the roulette wheel is used.

Termination criteria

A first criterion to determine is when to end the search of one LNS operator. *Oscar* provides several facilities to determine the termination criterion. Obviously it is once again crucial to select a termination criterion for one LNS operator and use the same in every test we want to perform. We need to allow expensive operators enough time so that they can actually get a chance at reaching a solution but we also want one LNS iteration to be reasonably short, since we will run many of them. We settled for a failure limit of 100 nodes.

Another criterion is the global LNS termination criterion, i.e. when should the LNS algorithm stop and return its best found solution. The difficulty here is to find a criterion which is suitable for instances of varying degrees of complexity. We decided to use an arbitrary time limit of 100 seconds, which seems long enough to get some level of optimization in most instances.

Dropped operators

We saw in our ALNS-TTI with Learning implementation that we drop some number of operators after the learning phase. This is also a parameter which can be tuned to influence the performance of the algorithm. We opted to keep the top 50% of the operators but of course this is highly dependent on the operators portfolio.

4.2 Results for a typical instance

In this section we will present what kind of results we can obtain for one sample instance of the problem. We will then, in the next section, aggregate these results over several instances.

4.2.1 Gap graph

A gap graph indicates the evolution of the gap to the optimal solution over time. This graph shows how fast the algorithm progresses to a good solution. Figure 4.2 shows one such graph for the Simple ALNS-TTI algorithm, as compared to three other selection mechanisms. We can see that, in the case of this instance, both ALNS-TTI and Laborie seem to perform better than a straight random selection (interestingly, this is not always the case, for either algorithms). The common initial solution passed to the ALNS algorithms has a cost roughly 15% larger than the best known solution (for some instance, this is the optimal solution). At the end of the executions, ALNS-TTI, Random, Laborie’s and UCB1 obtain a gap of about 6, 10, 7 and 8%, respectively.

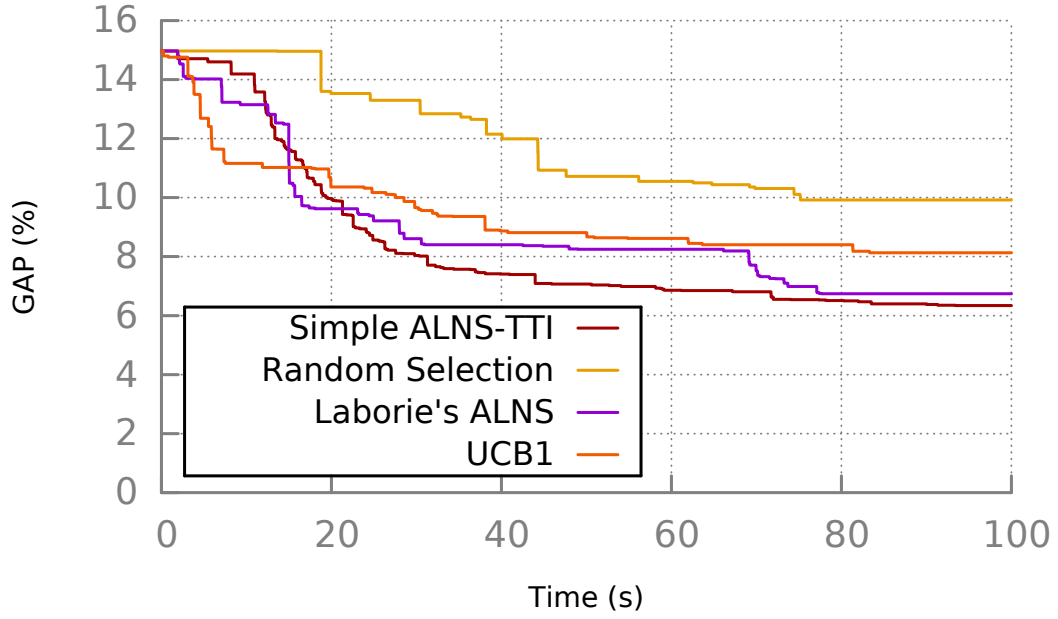


Figure 4.2: Gap over time graph for a single instance (X-n209-k16, 209 visits, 16 vehicles)

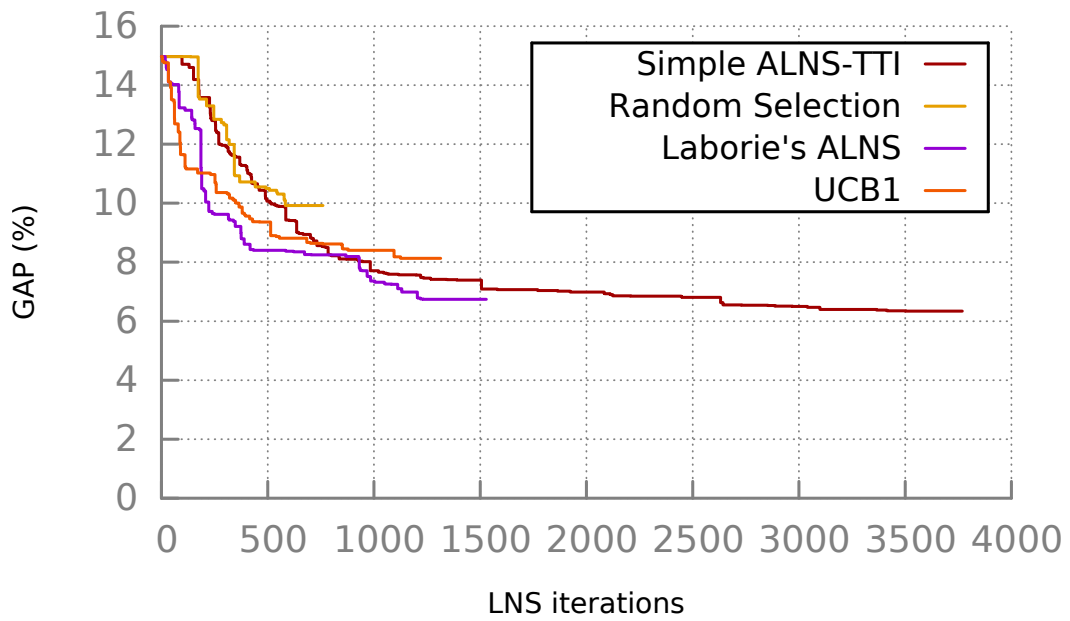


Figure 4.3: Gap over iterations graph for a single instance (X-n209-k16, 209 visits, 16 vehicles)

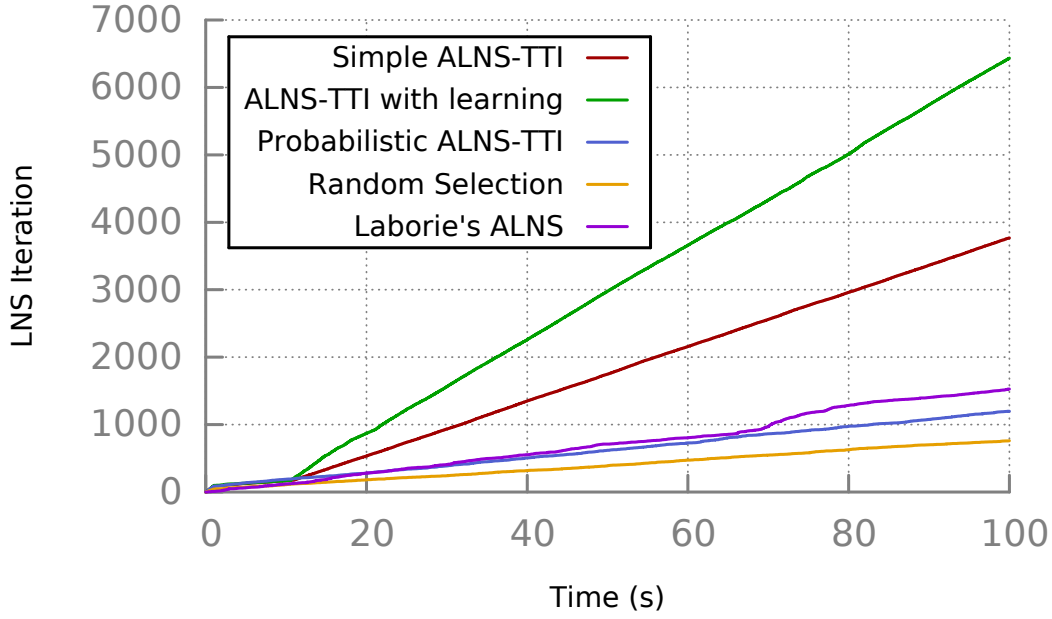


Figure 4.4: Iterations over time for a single instance (X-n209-k16, 209 visits, 16 vehicles)

4.2.2 Gap over iterations graph

Figure 4.3 shows the same gap evolution but spread over *iterations* rather than *time*. The ALNS-TTI curve is in this case much longer than the other two: this indicates that the algorithm performed much more iterations in its allowed time (which is 100 seconds). In fact, in this particular instance, it managed to run about three times more operators than Laborie and Random. Figure 4.4 shows this effect in more detail, for all 3 variants of our algorithm. Notice how the Learning version of the algorithm, in which the operators are first sorted randomly and each tried once successively, first follows the Random selection and performs the same amount of iterations in the same time, which makes sense intuitively. However, there seems to be a point at which the algorithm begins executing much more operators: this corresponds to end of the learning phase. The Simple ALNS-TTI version seems to follow a similar trend, which is not unexpected since it also begins with a phase where it tries each operator once.

We believe this is the effect of our TTI metric optimizing over the expected time to improvement, and rewarding fast operators. Our metric seems to favour fast operators much more than LALNS, which makes sense since LALNS uses a probabilistic model (in which even terribly slow operators always have a chance, however tiny, of being executed again) and while it also optimises over time, it also takes into account the order of the cost improvement performed, which may increase the selection probability of slower but more powerful operators.

For this particular instance, it is also interesting to observe on figure 4.3 that Laborie seems to use its imparted time more “wisely” and selecting more efficient operators, in the sense that in the same amount of iterations, it is almost always better than the other two algorithms. However, ALNS-TTI ends up with a slightly better gap only due to the fact that it managed to run much more operators in the given time. In other words, Laborie could appear to favour slower but more powerful operators more while simple ALNS-TTI would more tend to favour faster ones. Again, this would make sense since LALNS factors the cost improvement into its rewards.

4.3 Comparison with a random selection

We will now aggregate the results over our whole test dataset described earlier, to be able to get an overview of our the algorithms perform in general and not in a specific case. For reference, the individual gaps obtained at the end of each algorithm for all our 30 selected instances can be found in appendix A.1.

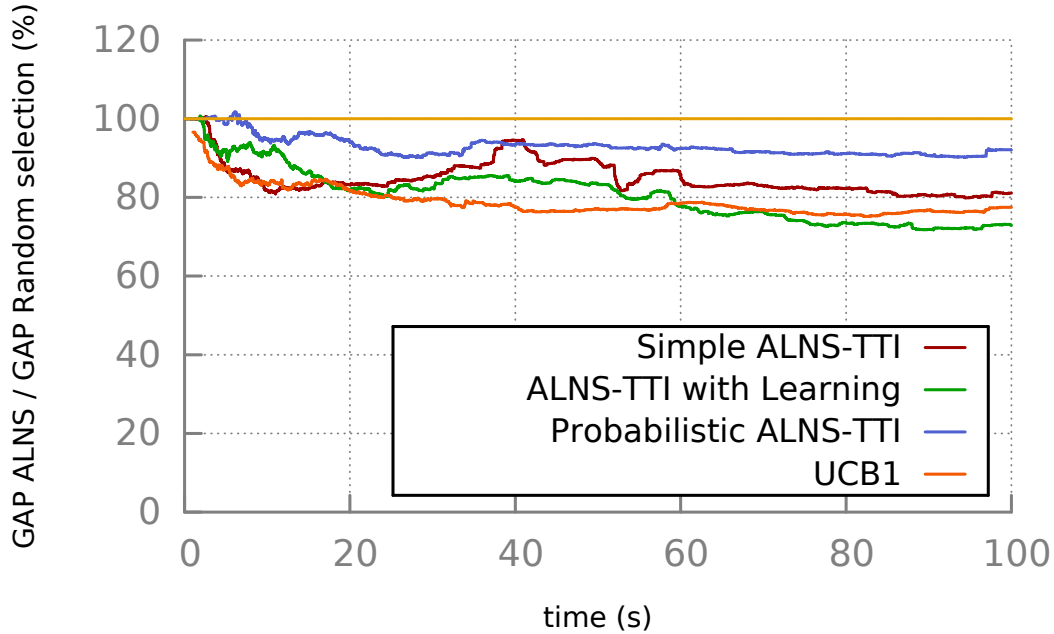


Figure 4.5: Average gap ratio of ALNS-TTI/Random over time

Figure 4.5 shows the evolution of the *gap ratio* of ALNS-TTI (the three variants) over a Random selection. This means that, at time t , we compute the ratio $\frac{gap_{ALNS-TTI}}{gap_{Random}}$ in order to see in what proportion one algorithm is better than random selection. This ratio can be aggregated meaningfully and displayed over time, as the figure does. A ratio of 100% indicates that both algorithm perform strictly identically, whereas a ratio $< 100\%$ indicates that ALNS-TTI performs on average better at this point in time, and conversely a ratio $> 100\%$ indicates that random selection performs better.

In our case, we can see that all three variants seem to perform better than random selection. The **learning** version is originally as good as the **simple** version, but seems to performing better only after some time, which likely corresponds to the end of the learning phase. The duration of this phase depends on the instance size, and so as time goes by more and more instances have finished the learning and can start improving the solution. The learning seems to give an advantage to this algorithm which starts improving the solution faster once the phase is over.

The **probabilistic** version of the algorithm seems to perform only slightly better than random selection : we believe this is because the number of operators is too large, and the probabilities do not get adapted sufficiently to make a real and more significant change compared to a strictly random approach. We will come back to this problem in the next section.

Concerning the simple version, we can see that between the 35 to 55 marks, the algorithm seems to suddenly perform slightly worse on average. This effect can be replicated throughout several runs of our test procedure. We believe this is the result of the algorithm periodically re-trying the tail operators, as mentioned in section 3.5.1. Over several instances, it averages out that this phenomenon often occurs around this time of the process and averaging the curves of

several instances reveals the performance “plateau” caused by this effect of re-trying all the tail operators.

UCB1 appears to be the overall best-performing algorithm. It is only beaten by the learning version of ALNS-TTI, and only after quite a while.

4.4 Comparison with Laborie’s ALNS

As mentioned before, the algorithm of [Laborie and Godard, 2007] will be our reference. Our goal was to get as close as possible to its performances. However we will see that our algorithms unfortunately fail to outperform Laborie’s ALNS. See figure 4.6 for a ratio plot like the one presented to compare our algorithms to random selection.

The data for our ALNS is the same as in the previous plot. We simply take the ratio over the curve for the average gap of Laborie’s, instead of the random selection. This being the case, it is not surprising that the simple version of the algorithm performs best again, here.

The interesting part of the plot is that Laborie’s ALNS is almost always better, especially early in the process. The learning version manages to compete, but needs time (too much time) to do so, because it must estimate good TTI values for all operators. UCB1, which attempts to solve the problem of having to try all possible operators by using loose operators, is slightly better, especially early in the process, but still fails to get an overall better gap than Laborie’s ALNS.

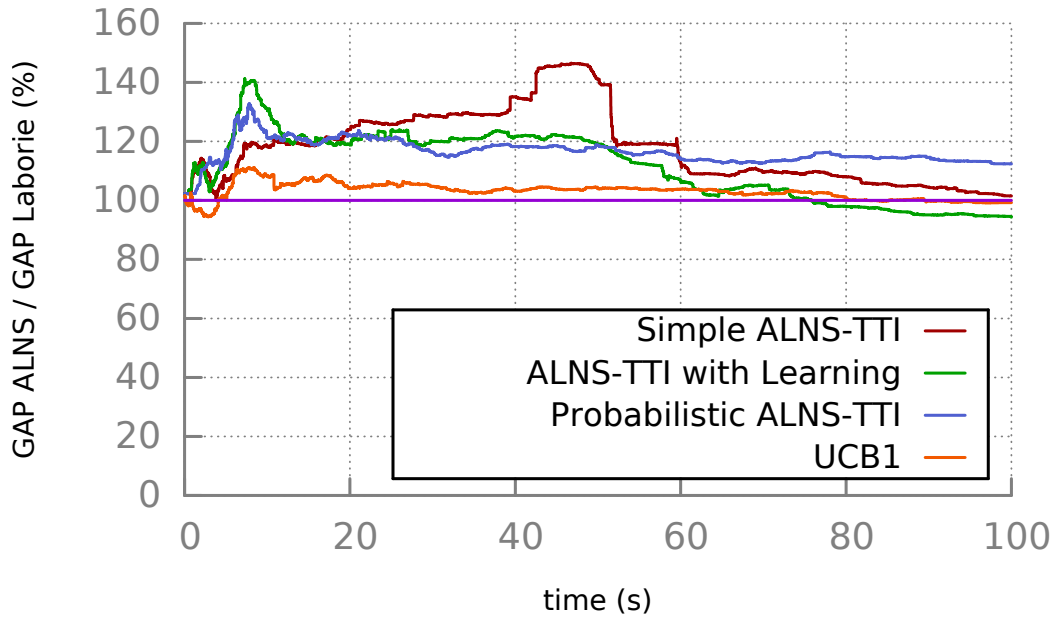


Figure 4.6: Average gap ratio of ALNS-TTI/Laborie’s over time

Our interpretation of these results is twofold: the first based on the poor performance of the probabilistic model, and the second from the evidence that the Learning version manages to catch up Laborie after some time.

A. The probabilistic model is trapped in tightly coupled operators

First, it is striking to observe how poorly the probabilistic model performs compared to Laborie’s, when both algorithms are quite similar in terms of the selection method. Remember that both use a roulette wheel selection scheme with the differences that:

1. Our version rewards the probabilities proportionally to $TimeLimit/TTI$, which is equal to

$$\frac{TimeLimit}{\frac{T}{p}} = \frac{TimeLimit \times p}{T}$$

whereas [Laborie and Godard, 2007] uses a reward of

$$\frac{CostImprovement}{T}$$

2. We use tightly coupled operators and [Laborie and Godard, 2007] uses loose operators.

The rewards being quite similar (ours having the benefit of being statistically founded), we believe this second difference is what costs us the most. Because we use tight operators, the set of operators in our algorithm is equal to the cartesian product of:

$$CS \times LN$$

Where CS and LN are the set of construction strategies and large neighbourhood heuristics, each heuristic taken with all the possible values of its parameter vectors. The cardinality of this set can grow very quickly and lead to a large set of operators, which means a roulette wheel with many possible selections. It is difficult for the mechanism of adaptation to work in these conditions because it takes a longer time to meaningfully adapt the wheel: one change in the weights has less impact when there are many many weights, and the probabilities change little by little.

Laborie's ALNS, on the other hand, has one roulette wheel for the set of CS heuristics, and one for the set of LN heuristics. Inside each heuristic, it has one wheel per parameter value. The adaptation is therefore done much faster.

B. ALNS-TTI with Learning manages to reach similar performance

A second interpretation we can make of these results comes from the fact that the learning version of the algorithm (which uses a priority queue) manages to reach similar performance after a while. The time taken to reach a performance close to Laborie is prohibitive, but it is encouraging in that, despite its flaws, this algorithm still manages to obtain somewhat good results.

These two observations lead us to believe that it might be interesting to investigate the TTI metric in a setting of loose operators, which would essentially ground Laborie's algorithm in a sound statistical basis, and possibly even obtain better performance (see later in the future works section).

All in all, it seems that the choice of tightly coupled operators was not a successful one, and ended up costing us a lot more than what we hoped we could possibly gain using this technique.

4.5 Operators

It can be interesting to compare the most selected operators between algorithms, in order to see if they tend to favour the same heuristics or not. However, since some algorithms use tight operators and others use loose operators, comparing the numbers can be tricky. Figure 4.7 gives pie charts for aggregated heuristics use in the ALNS-Simple version, for one specific instance. By

aggregated, we mean that each pie slice represents all LNS iterations where an operator which used a given heuristic was executed, regardless of the parameter values. Figure 4.8 shows such a chart for the same instance, but using Laborie’s ALNS.

We can see that both tended to use the *path-and-related* large neighbourhood more often, but interestingly they seem to disagree when it comes to construction strategies, with one preferring the *regret* heuristic and one using *deeply greedy* more often.

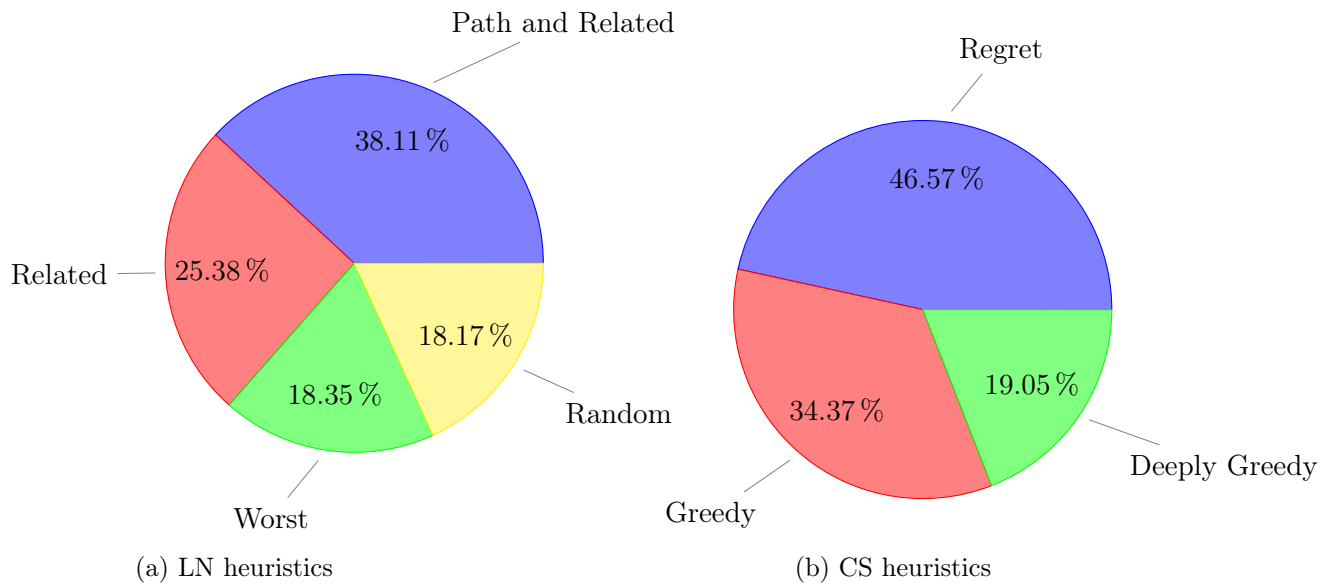


Figure 4.7: Proportion of selection for aggregated heuristics, using the simple ALNS-TTI algorithm on X-n120-k6

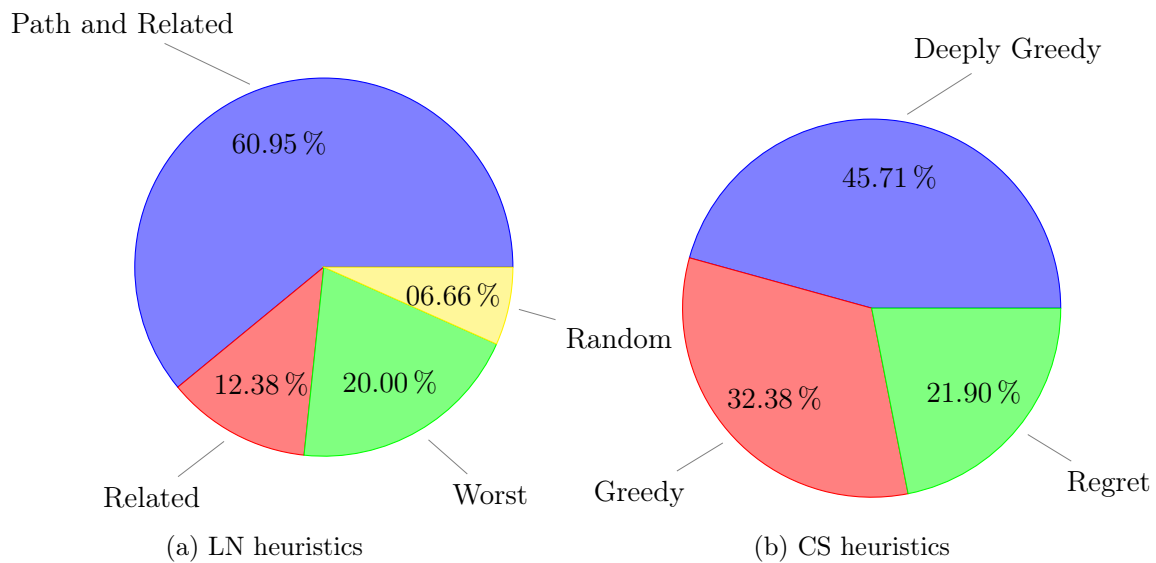


Figure 4.8: Proportion of selection for aggregated heuristics, using Laborie’s ALNS on X-n120-k6

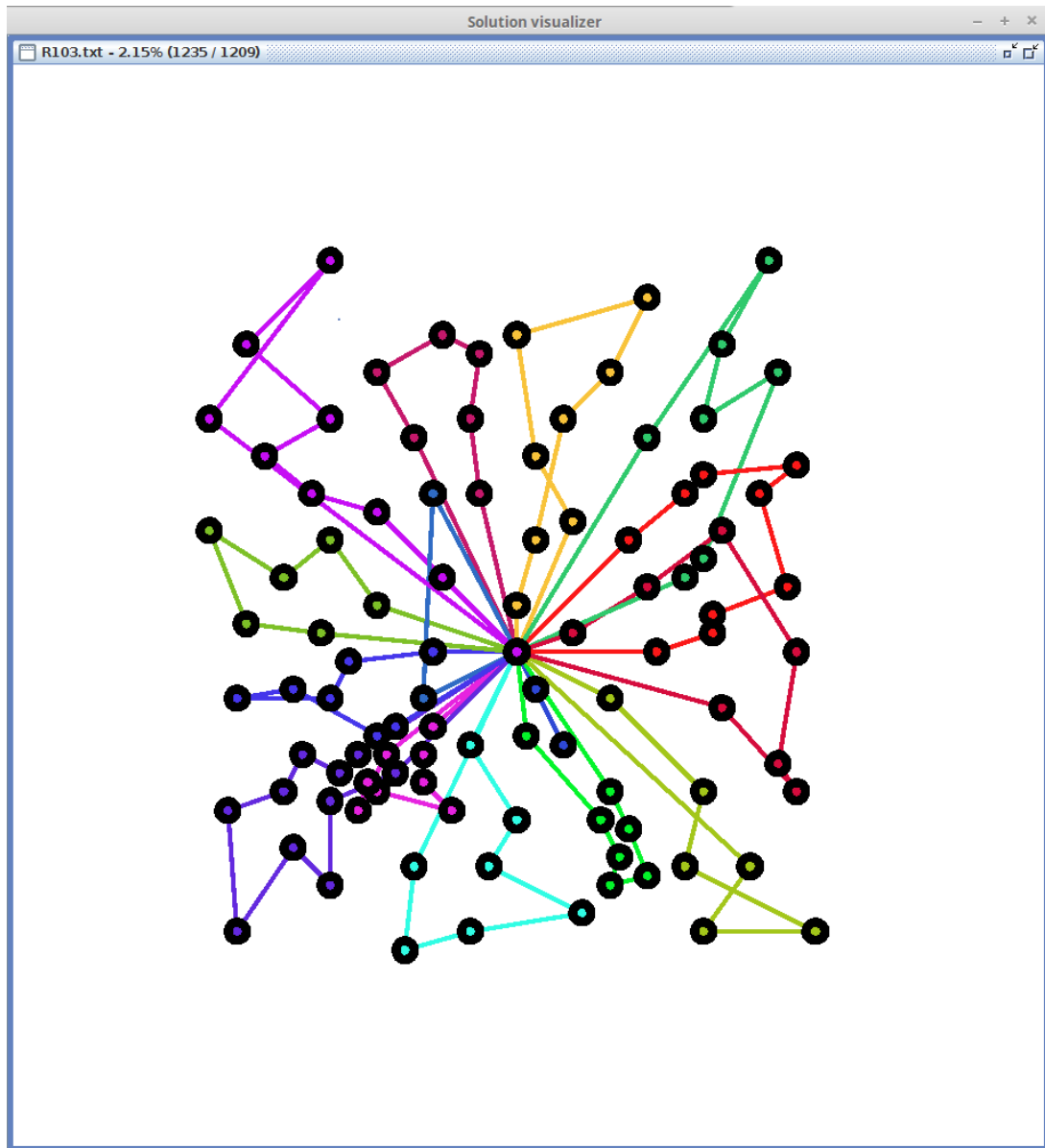


Figure 4.9: Example of a solution with a gap of 2.15%, obtained by the program with the Simple ALNS-TTI algorithm on CVRPTW instance R103 (with 100 visits) after a 100 seconds run

4.6 Results for other VRP variants

Our algorithms of course also work for other variants of the VRP. We will not go over detailed results, but this suggests that it possible to provide a generic way of solving several types of child problems of VRP without too much effort. Appendix A.2 gives the final gaps obtained for the standard instances from [Solomon, 1987]. Many of these instances are solved to optimality, regardless of the algorithms. But from a software perspective, it is interesting to see that such a modular infrastructure, with heuristics oriented towards solving CVRP without taking time windows into account, can still manage to perform well on this different problem.

Conclusions

As part of this master’s thesis, we successfully implemented a functional modular framework for the VRP and child problems, in parallel with a set of self-adapting LNS algorithms. We were able to use the latter to solve instances of the VRP with some moderate success but fell short of outperforming the existing state-of-the-art selection mechanism. We have seen that one of our premise, which was that some heuristics may be particularly strong when coupled together but inefficient when coupled with any other heuristic failed to appear or even be hinted at empirically. It seems that the possible gain we could obtain by discovering such a unlikely pair is more than matched by the cost paid to have tightly coupled operators.

However, the proposed TTI metric gives encouraging results, performing better than a random selection and staying relatively competitive with the state of the art, despite being seemingly handicapped by some of our early implementation choices.

The main point of struggle in this area seems to us to be the issue of initialization: the problem at hand is essentially to select items based on measures of their performance, but evaluating these performances may be quite costly. How can one make judicious decisions efficiently, based on values that are expensive to estimate? The probabilistic approach seems to be a good attempt, but as we have seen, it can quickly get bogged down when the number of items to select from is somewhat large.

4.7 Further works

The obtained results seem to us encouraging enough that we think some more work could be done in the same direction with some possible success of improving over the current state of the art. We had to make some decisions and implementation choices, which necessarily led to some unexplored ideas in our work. We will list here some possible leads to investigate further with regards to our topic of discussion.

4.7.1 TTI-based metric with loose operators

As we have seen, the choice of tight operators seems to have been a burden. But we believe the TTI metric to be a good selections mechanism, with a sound and intuitive reasoning behind it. It seems to us that the obtain results tend to indicate that our algorithms struggle because of the choice of tight operators, which essentially increases the number of elements to select from, thus making the problem harder. Yet it appears that once this difficulty of the large number of operators is overcome, the selection process starts performing better. We feel using a TTI-based metric with loose operators could possibly slightly improve performances over the best selection mechanisms around, and could warrant some more research.

4.7.2 Multi-armed bandit selection

We have brushed upon the subject of multi-armed bandit selection with UCB1 and we have seen that it gets quite good results. In our implementation, we combine the UCB1 approach, which starts by trying each “arm of the bandit” once (where each “arm” is a heuristic) with the roulette wheel mechanism of Laborie’s ALNS for the parameters. This makes: since the parameters is what creates the huge number of operators, we do not want to try all of them ; but we can afford to try each heuristic once since there is likely going to be only a handful of them. We essentially solve the initialization problem by having a probabilistic selection over the parameters and a learnt, statistical selection for the heuristics.

We feel this idea could be worth exploring further: in the same perspective, one could try to hybridize the TTI approach with Laborie for instance.

4.7.3 Generic ALNS

In our implementation section, we attempted to describe a generic pseudocode version of an adaptive LNS. But the actual implementation does not make use of any such pattern. We believe it could be interesting to explore the possibility to design a truly generic ALNS framework. In such a framework, the user could provide a functional argument for the selection scheme. There would be some complexities to overcome of course, as it might not be easy to encapsulate the whole mechanism in a single function. But if one could manage to implement this in a way that was friendly for the user, many different selections strategies could be implemented and tested easily. This could lead to faster improvements in the field of ALNS.

4.7.4 Application to other problems

Undoubtedly, this approach could be successful to solve many other optimization problems. We are confident that ALNS brings some improvement over straight LNS (although it could be interesting to experiment this formally: how much better does ALNS perform, compared to a straight LNS with the most selected operator?), but there does not seem to be many applications of ALNS to other problems in the literature. It could be worth looking into how ALNS manages in other situations. The proposed ALNS framework coupled with generic fragment selection algorithms such as proposed in [Mairy et al., 2010], [Lombardi and Schaus, 2014] or [Perron et al., 2004] could probably provide a generic yet efficient way of solving many optimization problems with little effort.

Bibliography

- [Anjos, 2012] Anjos, M. (2012). QAPLIB - Quadratic assignment problem library. <http://anjos.mgi.polymtl.ca/qaplib/inst.html>. Accessed : July 25, 2016.
- [Auer et al., 2002] Auer, P., Cesa-Bianchi, N., and Fischer, P. (2002). Finite-time analysis of the multiarmed bandit problem. *Machine learning*, 47(2-3):235–256.
- [Augerat et al., 1995] Augerat, P., Belenguer, J., Benavent, E., Corberán, A., Naddef, D., and Rinaldi, G. (1995). Computational results with a branch and cut code for the capacitated vehicle routing problem. *Rapport de recherche- IMAG*.
- [Benchimol et al., 2012] Benchimol, P., Van Hoes, W.-J., Régim, J.-C., Rousseau, L.-M., and Rueher, M. (2012). Improved filtering for weighted circuit constraints. *Constraints*, 17(3):205–233.
- [Bogaert, 2006] Bogaert, P. (2006). *Probabilités pour scientifiques et ingénieurs*, chapter 3, pages 52–53. De Boeck.
- [Christofides and Benavent, 1989] Christofides, N. and Benavent, E. (1989). An exact algorithm for the quadratic assignment problem. *Operations Research*, 37(5):760–768.
- [Clarke and Wright, 1964] Clarke, G. and Wright, J. W. (1964). Scheduling of vehicles from a central depot to a number of delivery points. *Operations research*, 12(4):568–581.
- [CVRPLIB, 2016] CVRPLIB (2016). Capacitated vehicle routing problem library. <http://vrp.atd-lab.inf.puc-rio.br/index.php/en/plotted-instances>. Accessed : July 20, 2016.
- [De Backer et al., 2000] De Backer, B., Furnon, V., Shaw, P., Kilby, P., and Prosser, P. (2000). Solving vehicle routing problems using constraint programming and metaheuristics. *Journal of Heuristics*, 6(4):501–523.
- [Focacci et al., 1999] Focacci, F., Lodi, A., and Milano, M. (1999). Cost-based domain filtering. In *International Conference on Principles and Practice of Constraint Programming*, pages 189–203. Springer.
- [Gendreau and Potvin, 2010] Gendreau, M. and Potvin, J.-Y. (2010). An adaptive large neighborhood search for a vehicle routing problem with multiple trips. *CIRRELT*, pages 5–7.
- [Gillett and Miller, 1974] Gillett, B. E. and Miller, L. R. (1974). A heuristic algorithm for the vehicle-dispatch problem. *Operations research*, 22(2):340–349.
- [Held and Karp, 1962] Held, M. and Karp, R. M. (1962). A dynamic programming approach to sequencing problems. *Journal of the Society for Industrial and Applied Mathematics*, 10(1):196–210.
- [Hong, 2012] Hong, L. (2012). An improved lns algorithm for real-time vehicle routing problem with time windows. *Computers & Operations Research*, 39(2):151–163.

- [Kilby, 2013] Kilby, P. (2013). Constraint programming for the vehicle routing problem. CP2013: 19th International Conference on Principles and Practice of Constraint Programming.
- [Labadie et al., 2016] Labadie, N., Prins, C., and Prodhon, C. (2016). *Metaheuristics for Vehicle Routing Problems*. John Wiley & Sons.
- [Laborie and Godard, 2007] Laborie, P. and Godard, D. (2007). Self-adapting large neighborhood search: Application to single-mode scheduling problems. In *Proceedings of the 3rd Multidisciplinary International Conference on Scheduling : Theory and Applications (MISTA 2007)*, pages 276–284.
- [Larson and Odoni, 1981] Larson, R. and Odoni, A. (1981). Urban operations research. http://web.mit.edu/urban_or_book/www/book/chapter6/6.4.12.html. Accessed : July 07, 2016.
- [Lenstra and Kan, 1981] Lenstra, J. K. and Kan, A. (1981). Complexity of vehicle routing and scheduling problems. *Networks*, 11(2):221–227.
- [Li and Lim, 2003] Li, H. and Lim, A. (2003). A metaheuristic for the pickup and delivery problem with time windows. *International Journal on Artificial Intelligence Tools*, 12(02):173–186.
- [Lombardi and Schaus, 2014] Lombardi, M. and Schaus, P. (2014). Cost impact guided lns. In *International Conference on AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, pages 293–300. Springer.
- [Mairy et al., 2010] Mairy, J.-B., Schaus, P., and Deville, Y. (2010). Generic adaptive heuristics for large neighborhood search. In *Seventh International Workshop on Local Search Techniques in Constraint Satisfaction (LSCS2010). A Satellite Workshop of CP*.
- [Oscar Team, 2016] Oscar Team (2016). Oscar: Scala in OR. Available from <https://bitbucket.org/oscarlib/oscar>.
- [Perron et al., 2004] Perron, L., Shaw, P., and Furnon, V. (2004). Propagation guided large neighborhood search. In *International Conference on Principles and Practice of Constraint Programming*, pages 468–481. Springer.
- [Pisinger and Ropke, 2010] Pisinger, D. and Ropke, S. (2010). Large neighborhood search. In *Handbook of metaheuristics*, pages 399–419. Springer.
- [Ribeiro and Laporte, 2011] Ribeiro, G. and Laporte, G. (2011). An adaptive large neighborhoods earch heuristic for the cumulative capacitated vehicle routing problem. *CIRRELT*, pages 5–13.
- [Ropke and Pisinger, 2006] Ropke, S. and Pisinger, D. (2006). An adaptive large neighborhood search heuristic for the pickup and delivery problem with time windows. *Transportation Science*, 40(4):455–472.
- [Rousseau et al., 2004] Rousseau, L.-M., Gendreau, M., Pesant, G., and Focacci, F. (2004). Solving vrptws with constraint programming based column generation. *Annals of Operations Research*, 130(1-4):199–216.
- [Schaus, 2016a] Schaus, P. (2016a). Large neighborhood search - CP for the impatient beta documentation. <https://www.info.ucl.ac.be/~pschaus/cp4impatient/lns.html>. Accessed : July 25, 2016.
- [Schaus, 2016b] Schaus, P. (2016b). Self adapting lns minimizing expected time for improvement. Unpublished.

- [Shaw, 1998] Shaw, P. (1998). Using constraint programming and local search methods to solve vehicle routing problems. In *International Conference on Principles and Practice of Constraint Programming*, pages 417–431. Springer.
- [Shaw, 2004] Shaw, P. (2004). A constraint for bin packing. In *International Conference on Principles and Practice of Constraint Programming*, pages 648–662. Springer.
- [Solomon, 1987] Solomon, M. (1987). Algorithms for the vehicle routing and scheduling problems with time window constraints. *Operations Research*, 35(2):254–265.
- [Toth and Vigo, 2014] Toth, P. and Vigo, D. (2014). *Vehicle routing: problems, methods, and applications*, volume 18. Siam.
- [Uchoa et al., 2014] Uchoa, E., Pecin, D., Pessoa, A., Poggi, M., Subramanian, A., and Vidal, T. (2014). New benchmark instances for the capacitated vehicle routing problem. Technical report, Research Report Engenharia de Produção, Universidade Federal Fluminense.

Appendix A

Gap summaries

A.1 X dataset [Uchoa et al., 2014]

Instance	Strict Min-TTI	Learning	Probabilistic	UCB1	Laborie	Random
X-n106-k14.vrp	2.87	2.62	2.6	2.44	3.46	2.1
X-n110-k13.vrp	3.79	1.72	5	1.82	6.65	1.73
X-n115-k10.vrp	2.06	2.07	1.26	1.76	1.39	3.86
X-n120-k6.vrp	4.94	3.28	2.97	3.33	1.54	2.34
X-n129-k18.vrp	3.82	7.98	5.9	4.51	6.92	6.44
X-n134-k13.vrp	4.27	12.22	5.97	6.47	6.45	13.56
X-n139-k10.vrp	7.53	5.94	4.27	5.49	7.11	11.48
X-n143-k7.vrp	7.55	4.45	3.9	2.21	2.96	6.61
X-n157-k13.vrp	3.28	3.23	2.01	2.96	2.81	3.23
X-n162-k11.vrp	2.43	5.45	3.49	2.42	4.6	5.59
X-n167-k10.vrp	12.61	5.1	7.25	7	8.1	8.85
X-n186-k15.vrp	7.27	8.13	6.76	5.52	10.12	9.87
X-n190-k8.vrp	2.99	5.8	4.32	4.37	3.6	4.15
X-n204-k19.vrp	10.55	6.55	14.29	10.9	9.79	15.25
X-n209-k16.vrp	6.34	4.71	9	8.13	6.74	9.92
X-n214-k11.vrp	13.47	15.88	25.92	16.82	20.4	21.6
X-n233-k16.vrp	58.95	43.23	63.08	59.5	52.84	65.69
X-n237-k14.vrp	6.16	7.02	10.65	6.97	7.34	12.13
X-n256-k16.vrp	21.78	17.73	27.99	30.35	23.27	29.22
X-n261-k13.vrp	8.78	10.3	17.55	12.21	14.41	21.13
X-n280-k17.vrp	14.94	13.13	22.14	15.93	17.31	23.88
X-n284-k15.vrp	9.86	7.02	15.63	13.49	10.22	17.37
X-n303-k21.vrp	15.08	9.68	22.45	20.33	17.2	25.19
X-n308-k13.vrp	13.55	17.89	24.44	17.35	15.77	33.39
X-n327-k20.vrp	9.61	8.38	19.19	14.82	16.87	20.08
X-n331-k15.vrp	7.82	6.99	11.63	9.88	9.27	14.73
X-n367-k17.vrp	12.04	11.19	25.72	24.72	21.2	25.18
X-n411-k19.vrp	38.87	24.88	38.86	38.87	39	36.43
X-n513-k21.vrp	32.14	31.94	29.56	32.11	32.06	32.14

Table A.1: Final gaps for a 100-seconds run of each algorithm for 30 instances of the X dataset from [Uchoa et al., 2014]

A.2 R and C datasets [Solomon, 1987]

Instance	Strict Min-TTI	Learning	Probabilistic	UCB1	Laborie	Random
C101-25	0	0	0	0	0	0
C101-50	0	0	0	0	0	0
C101-100	0	0	0	0	0	0
C102-25	0	0	0	0	0	0
C102-50	0	10.25	0	0	0	0
C102-100	0	0	0	0	0	0
C103-25	0	0	0	0	0	0
C103-50	11.08	0	0	0	0	0
C103-100	0	0	0	0	0	0
C104-25	0	0	0	0	0	0
C104-50	0	0	0	0	0	0
C104-100	0	2.92	0	0	0	0
C105-25	0	0	0	0	0	0
C105-50	0	0	0	0	0	0
C105-100	24.79	0	0	0	0	0
R101-25	0	0	0	0	0	0
R101-50	0.86	0	0.67	0	0	0
R102-25	0	0	0	0	0	0
R102-50	2.64	0.55	0.55	0.55	0.55	0
R103-25	1.76	1.76	0	0	0	1.76
R103-50	0.52	3.49	0.52	0	1.81	0.52
R103-100	4.71	2.98	1.32	5.38	3.64	1.49
R104-25	0	0	0	0	0	0
R104-50	4	5.92	3.36	0.64	0.64	3.36
R104-100	3.4	2.78	9.57	5.66	5.97	3.7
R105-50	0.56	0.67	0.67	0.22	0.67	0.67

Table A.2: Final gaps for a 100-second run of each algorithm for the Solomon dataset

Appendix B

Quantitative data

Table B.1 gives quantitative data of the implemented solution. The global project, including generated statistics and charts, makes up a total of 461 MB.

Files	Language	Blank	Comment	Code
27	Scala	412	246	1118
15	Bourne Shell	247	266	458
4	R	30	10	56
1	Perl	20	0	33

Table B.1: Total lines of code of the implemented solution and statistical manipulation tools

