Karlsruhe Institute of Technology

Constraint Programming based Local Search for the Vehicle Routing Problem with Time Windows

Master Thesis of

Joan Sala Reixach

At the Department of Informatics
Institute of Theoretical Computer Science

Reviewers:	Prof. Dr. Dorothea Wagner
	Prof. Dr. Peter Sanders
Advisors:	Anne Meyer (FZI, LSE Furmans)
	Julian Dibbelt (KIT, ITI Wagner)

Time Period: 1st April 2012 – 30th September 2012

Statement of Authorship

I hereby declare that this document has been composed by myself and describes my own work, unless otherwise acknowledged in the text.

Karlsruhe, September 30, 2012

Abstract

Vehicle Routing is a problem of extreme variance and at the same time, of extreme importance to companies around the world. Extensive work has been done on finding methods to solve it in an efficient way that provides a solution of good quality.

This project focuses especially on the Vehicle Routing Problem with Time Windows. It explores and tests a method based on a Constraint Programming formulation of the problem, and it implements a local search method with the power of making very powerful moves: Large Neighbourhood Search. Large Neighbourhood Search removes big sets of customers from the solutions (up to around 30% of the customers) and tries to reinsert them. Those powerful moves help solving one of the biggest problems of local search methods: Escaping local minima.

We first explore a basic path formulation of the model, as well as some alternative formulations. We implement a first basic version of a Large Neighbourhood Search engine and try to improve it through the exploration of different options for all the decisions that must be taken when designing such an engine. Concretely, the main decisions to take are about how to design the methods to remove customers from the solution and the methods to reinsert them back into the solution, desirably in a better way.

The methods are then tested using a set of benchmark problems. The results obtained are then compared to the best known solutions for these benchmarks, as well as to the results obtained by other authors that have done work regarding the Vehicle Routing Problem with Time Windows.

Deutsche Zusammenfassung

Dieses Projekt beschäftigt sich mit dem Vehicle Routing Problem. Diese Probleme sind extrem unterschiedlich und ähnlich nur in ihren Grundbegriffen. Das Problem ist aber sehr wichtig für Firmen, die im Logistikbereich arbeiten.

Dieses Projekt konzentriert sich vor allem auf das Vehicle Routing Problem mit Time Windows. Es erforscht und testet eine Methode basierend auf einer Constraint-Programming Formulierung des Problems, und es setzt eine lokale Suchmethode ein, mit der Fähigkeit, große Teilen der Lösung zu ändern: Large Neighbourhood Search. Diese Methode löscht eine große Menge von Kunden aus den Lösungen (bis etwa 30 % der Kunden) und versucht, sie besser wieder einzufügen. Diese mächtigen Moves helfen bei der Lösung eines der größten Probleme der lokalen Suchmethoden: Lokalen Minima zu entkommen.

Zuerst exploriert die Arbeit die elementare Path-Formulierung des Modells, sowie einige alternative Formulierungen. Wir implementieren eine erste Basisversion einer großen Nachbarschaftssuche. Wir versuchen danach diese Basisversion zu verbessern durch die Erforschung der verschiedenen Optionen für alle Entscheidungen, die sich präsentieren bei der Gestaltung der Suche. Konkret sind die wichtigsten Entscheidungen zu treffen, wie die Kunden gelöscht werden und wie die gelöschten Kunden wieder in die Lösung eingefügt werden.

Die Methoden werden dann unter Verwendung eines anerkannten Benchmarks getestet. Die erhaltenen Ergebnisse werden dann zu den besten bekannten Lösungen für diese Benchmarks sowie zu Ergebnissen von anderen Autoren verglichen.

Contents

1. Introduction

Vehicle routing is a key problem to management of goods distribution and has become one of the most important areas for improvement for companies around the world, and one which must be systematically solved. In the industrial practice, vehicle routing problems are extremely different and they share only a common basis. This high variability comes from the different requisites and constraints that each company will provide for the problem, and in it resides the complexity of solving these problems. Therefore it is interesting to find a method that allows us to be able to easily model constraints, as well as being able to modify them easily without the need of major changes. Its flexibility to add new constraints or modify existing ones is what makes Constraint Programming (see [RVBW06]) so useful for this problem.

We provide extensive evaluation on a suite of well-known test instances, namely the Solomon instances designed in [Sol87]. Solomon instances are sets of benchmark Vehicle Routing problems, which present different scenarios. There are sets with the customers distributed randomly, and sets of problems with clustered clients. Vehicle capacity, as well as the time windows of the customers also vary. With them, the best solution found so far for each of them is also provided. These instances are very useful for the testing of applications for the Vehicle Routing Problem.

1.1 Related Work

Extensive work has been done around the vehicle routing problem, as summarized in [Lap09]. A wide variety of exact algorithms have been developed, including branch and bound, dynamic programming, set partition and flow algorithms. However, these exact methods have proved unfeasible for problems bigger than 100 nodes. To try and solve bigger problems many heuristic methods have been introduced, one of the most popular being the Savings method proposed by Clarke and Wright in 1964 in [CW64]. The Savings method starts with an initial solution tries to merge two routes at each iteration until it is not possible to merge routes anymore. Set Partitioning heuristics [GM74], try to solve the set partitioning formulation with a subset of promising vehicle routes. Generalized assignment (GAP) approaches were intorduced in [FJ81]. These methods try to locate a given number of seeds and cluster the visits around those, then solve TSP for each cluster.

Further research has been done in the line of metaheuristics. These methods start with an initial solution and try to improve it according to some criteria by applying changes

to it. Many Local Search methods have been applied to vehicle routing problems. Local search methods differ mainly in how the neighbourhood is defined. Basic methods have been applied to the Vehicle Routing Problem, like Tabu Search in [GLS96, BGG+97] and Simmulated Annealing in [DS90, Due93] Other methods have been explored like Variable Neighborhood Search in [MH97] or Very Large Neighborhood Search in [EOSF06]. Population methods like Genetic algorithms (proposed by [HM91]) have also been proposed and can be combined with local search methods. Some research has also been made in how to include neural networks to provide learning mechanisms that can learn from experience and incrementally adjust their weights in an iterative fashion. This concept has been however found to be hardly able to be applied to VRP.

1.2 Contributions

This project focusses in a combinations of constraint programming and local search methods, which leads to the Large Neighbourhood Search. This method consists of removing a set of visits from a current feasible solution and re-inserting them in an optimal or at least better way with the help of Constraint Programming, which is used to maintain the domains of this variables through propagation rules. This allows both the high exploration from local search methods and the propagation of constraint programming methods to be used.

The main aim of this project is to further examine the constraint programming based large neighbourhood search introduced by Paul Shaw in 1998 [Sha98]. In particular, we want to find better lower bounds for the solutions, determine meaningful strategies for the large neighbourhood search—selection of clients and reinsertion, as well as design experiments for performance measuring with respect to the instances known from the literature. First of all a basic version of the Constraint Programming based Large Neighbourhood Search method will be implemented in Gecode (see [STL11]). This will serve as the starting point from which further work will be done to improve the methods that form engine and will serve as a base to test whether the proposed new methods actually perform better.

The main side of the project results serve as a test method for the Large Neighbourhood Search method, as well as the adequacy of using Constraint Programming to solve Vehicle Routing problems. It also aims to test the flexibility of Constraint Programming to easily adapt to the introduction of new side constraints.

1.3 Summary

The concepts in the project will be presented in the following order:

- **Chapter 2** reviews the basic concepts that will be used throughout the project. This consists of the description of the Vehicle Routing Problem, and key concepts about Constraint Programming and Large Neighbourhood Search.
- **Chapter 3** describes the problem modelling—the formulation of the Vehicle Routing Problem in terms of Constraint Programming, as well as alternative formulations.
- **Chapter 4** describes the search engine that implements Large Neighbourhood Search and details the various methods that have been implemented and tested.
- Finally, **Chapter 5** exposes the tests that have been run and the results obtained.
- **Chapter 6** presents the conclusions of the work.

2. Preliminaries

This chapter serves as an introduction to the project by defining and laying out the main concepts that we will work with. First we provide a formal definition of the Vehicle Routing Problem with Time Windows. Then we review Constraint Programming and Large Neighbourhood Search, laying out the main aspects and advantages of those methods. We also provide an introduction to the C++ framework that we used to implement the system, Gecode. We explain how it works generally as well as which are its main components and operations.

2.1 The Vehicle Routing Problem

The Vehicle Routing Problem (VRP) is a generalization of the famous travelling salesman problem (TSP). Whereas in the TSP we only have one vehicle to visit all the customers, VRP generalizes to a fleet of multiple vehicles. The VRP is precisely the problem of finding a set of routes to visit a set of customers exactly once with the available vehicles, and considering restrictions like vehicle capacity. We want to find a solution that is optimal according to some criteria. Usual optimization criteria are the minimization of the travelled distance or travel time, and number of vehicles required to visit all the customers. The problem can be further complicated with the addition of the time period during which a costumer can be visited. This time interval is called a time window and the resulting problem, a VRPTW. We define a Vehicle Routing Problem with Time Windows as:

- The set of n customers that must be visited, coupled with their coordinates $[x_i, y_i]$, for $i = 1..n$.
- The amount r_i of goods requested by each customer i.
- The time window $[a_i, b_i]$ for each customer indicating in which period of time the costumer i can be visited.
- The set of m vehicles, the vehicle fleet. In the case of a homogeneous fleets, it consists of a single value indicating how many vehicles are available.
- The vehicle capacity Q_j, for $j = 1..m$. If the fleet is homogeneous all vehicles have the same capacity and thus a single value Q is given for all vehicles.
- The depot coordinates $[x_d, y_d]$, where all vehicles start and must end their tours.

The problem is extremely variable and flexible, allowing to add a wide variety of side constraints. This will almost always happen in real life, as each company or situation asks for their own side constraints or the exact optimization criteria that will fit its needs. There are also several variants of the problem that arise from the depot, including problems with multiple depots, problems which don't require vehicles to return to the depot and problems where the picking up of goods is done through the route (pick-up and delivery).

2.2 Constraint Programming

Constraint programming is a declarative paradigm that, unlike imperative languages, does not specify the exact step sequence to find the solution. Constraint programming limits the programming process to a generation of constraints [Apt03]. These constraints describe a solution by specifying the requirements it must meet. As the step sequence to execute is not determined, constraint programming subsequently has to use the specified properties of a solution to find it. It makes use of domain variables representing the range of values the variable can take. Those domain variables are reduced according to the constraints during computation, pruning the space and guiding the search. It provides easy use for both mathematical constraints (e.g. $X > Y$), or more complex symbolic constraints that describe non-mathematical relations between constraints (e.g. all-different constraint, which applied to a vector will make sure all the variables in that vector take different values). These symbolic constraints allow for precise modelling for the problems [FLM02]. When all the variables are assigned to one value, a solution is found. If on the contrary the domain of any variable becomes empty, the model is failed. If there are still variables with more than one possible value, then we need to keep exploring the search space according to some search strategy.

Standard search methods that perform complete search on the whole space are usually unable to find a solution in a short time period, and have the big problem of local minima. To try and escape those minima and find solutions in shorter times, it is interesting to use some heuristic or meta-heuristic to guide the search. Iterative small improvements done by small changes in the current solution, usually have success in these problems [BFS+00]. Constraint programming provides a very rich language to model the problem, both for usual arithmetic and logical operators as well as complex constraints, for example the all-different constraint. Constraint programming also allows for great flexibility to the insertion of different side constraints to a basic model, and this is precisely what makes it a very adequate tool for vehicle routing problems, as those are usually very variable in the sense that each of them incorporates different side constraints to the basic model.

2.3 Gecode

Gecode [STL11] is a C++ software library for the development of constraint programming applications, distributed as free software under the MIT license. Despite being a mature tool, still a lot of acting development and improving is being done to provide with more reliable tools. It is open source, as all the code is available, and has a detailed and extensive documentation. It also comes with an interactive graphical debugging tool, Gist, which is useful when testing applications. Gecode provides a lot of options to the developer, as it allows the programming of new variables, propagators, branching strategies and even the search engine itself. This feature makes possible to develop elements specifically designed for our own problem.

Models are implemented in Gecode using spaces. Spaces are the basic Gecode objects, which contain variables, propagators and branchings. Propagators implement constraints and branchings describe the search tree. To implement a Gecode model, one needs to start

by creating a space and defining the variables we want to assign values to. Next step is specifying the constraints that the variable values need to satisfy. Gecode offers a rich set of propagators that can be easily posted to the model using its constraint post function. Those propagators implement a variety general of constraints, however it is interesting to design one's own propagators to post constraints specific to our own problem. Finally, the branchings define the shape of the search tree, and are implemented by branchers. Branchers take a space and create a choice, which consists of a number of alternatives to be explored. Gecode provides basic Depth-first search and Branch-and-bound engines, but again it will be interesting to implement our own branchings to meet the specific needs of our problem.

2.3.1 Modelling

The model is implemented following C++'s object orientation philosophy, and thus will be a subclass of the class *Space* which implements all spaces, and will inherit from it. Posting variables is simple, as Gecode provides operations for creation, access and update of integer, boolean and set variables, as well as arrays of said variables. These variables will later be constrained through the posting of constraints, and modified through constraint propagation and branching. Aside from the model itself, our space will need to implement a cost function that defines how good a solution is if we want to use Branch-and-bound search. Besides, a copy creator and a copy function are needed for the search engines to work.

2.3.1.1 Gecode Variables

The variables used in Gecode are not like usual C++ variables. While the last contain only the value of the variable, Gecode variables contain the set of possible values that the variable can still take. When constraint propagation occurs, the domains of these variables will reduced through the removal of the values that are no longer permitted for each one. Gecode variables can be maintained in three consistency levels [STL11]:

- **Value consistency.** Maintains the whole set of possible values for a variable and performs value propagation. Gecode will wait until a variable is assigned and then remove all the values that are not consistent with this assignment from the domains of the other variables.
- **Domain consistency.** Like value consistency, it maintains the whole set of possible values for a variable but will always keep this set consistent with the domains of the other variables.
- **Bounds consistency.** Only the lower and upper bound for the variable are maintained, giving a range of possible values from min to max.

2.3.2 Propagation

Propagators implement constraints by removing values from the variable domains that are no longer permitted, because they are in conflict with a constraint. As variables do not offer operations for modification—only for access—, propagators work with variable views instead, which serve as an interface to access modify the values of the variables. When a propagator is created, it will subscribe to some views, and will go from idle to scheduled for execution as soon as that view is modified—that is, some of its values are removed. When the *status*() function of a space is called, it will execute one of the propagators that are scheduled for execution. If no more propagators are scheduled—no more propagation can be done, then we call the space stable. When a propagator is executed, values will be removed from the views. This can cause in turn to schedule more propagators. Besides

propagating the constraints, the propagators will also report about the propagation done, reporting one of the following outcomes:

- **SS_FAILED.** A propagator will report failure if the constraint they implement is not satisfiable anymore with the current assignment of variables.
- **ES_FIX.** The propagator is at a fixpoint. We say a propagator is at a fixpoint if it cannot remove any more values from any of its views.
- **ES_NOFIX.** Propagation was successful but the propagator is not at a fixpoint.
- **ES_SUBSUMED.** The constraint the propagator implements is guaranteed to be satisfied from this point. The propagator can thus be disposed of. A propagator must report subsumption at latest when all of its views are assigned.

2.3.3 Branching

Branchers are used in modelling to determine the shape of the search tree. Spaces have a list of branchers available in a queue. Branchers will be executed following that queue order. The first brancher in the queue is called the current brancher. Three functions are basic to the implementation of branchers—these are actually virtual functions that will be used by the search engine—.

- **status()** tests whether the current brancher has anything left to do, for example if there are unassigned views on which to branch.
- **choice()** will create a choice with a number of alternatives, which describe to the search engine how to branch. It should be space independent and not contain any information about the space so it can be used with different spaces.
- **commit()** will commit to one of the previously created alternatives, typically modifying views as defined by the choice.

We will understand the use of these functions when looking at how search engines work.

2.3.4 Search Engines

Search in Gecode is based on spaces. Spaces have to implement the same three functions as branchers: $status(), choice()$ and $commit()$. When an engine wants to determine if a Space is failed, it will call its $status()$ function, that will trigger constraint propagation and check the result of the process. If the space is failed, it will return SS_FAILED. Otherwise the search engine will check the status of the current brancher. If it returns $false$, the engine will jump to the next brancher in the queue. When one of them returns $true$, the $status()$ function of the Space will return SS_BRANCH, indicating that branching is required. If all the branchers in the queue return $false$, then it means that any of the branchers has any work left to do and thus the engine will conclude that the space is solved. If branching is required, the engine will call the $choice()$ function of the space. This function will in turn call the $choice()$ function of the current brancher, which will compute a choice with a number of alternatives and return it back to the engine. Once this choice has been computed, the engine can use the $commit()$ function of the space. This function will commit the space to one of its alternatives by calling the $commit()$ member of the brancher that has generated the choice.

Besides, the space has to provide a method for cloning during search. This is used to allow the engine to backtrack. When branching is needed, the engine will always create a clone before commiting to one of the alternatives. This clone of the space can be used if in the future it needs to backtrack and commit to a different alternative, as it is a copy of the exact state of the space at that point in the search.

Finally, a cost function is also needed to guide the search. Gecode needs a way of determining whether a solution is better than another. When this is required, the cost function will be called to evaluate the current quality of the space.

2.4 Large Neighbourhood Search

Large Neighbourhood Search (LNS) is a term first coined in [Sha97, Sha98]. It is a form of local search enhanced by constraint programming techniques, following a tree-based local search. Constraint Programming makes it easy to maintain the domains of the variables and can be easily used to evaluate the legality and cost of moves [Sha11]. It tries to take advantage both from the high exploration capacity of local search methods and the flexibility of constraint programming methods. Each move in the search is defined by the removal and re-insertion of a set of costumer visits. LNS is a process of continual relaxation and re-optimization, which at each step, will remove a set of customers and try to re-insert them in a better way. As the neighbourhood of a solution is defined by the set of all other solutions that can be reached using those relax and reinsert methods, the neighbourhood of a solution is implicitly defined by the relaxation and reinsertion methods.

- **Relaxation** A method must be designed to destroy the current solution by removing a set of customers from it. It is interesting to remove a set of customers that are in some way related, for example because they are geographically close or in the same tour. For this reason, a function that measures relatedness between customers is needed. This method is very important, as it will in a strong way determine whether LNS is or is not successful. We don't want to choose variables that are likely to maintain the same value when reinserted, but variables that give way to better alternatives and improvement. Another point to consider is how many customers to remove, that is, the neighbourhood size. It is usually a good strategy to start with neighbourhood size of 1, and increase it when the search cannot improve the current solution any more, or when the search does a given number of consecutive moves without improvement. The neighbourhood size will keep increasing until some fixed limit is reached.

- **Reinsertion** Once the solution has been destroyed according to the relaxation method he following problem to solve is where to reinsert the customers we have removed. For this we will use constraint programming and metaheuristics. The relaxed visits are the constrained variables, maintained through propagation rules that also maintain capacity and time constraints along a route. A simple branch and bound method can be used to find the minimum cost insertion place for the relaxed customers, but this can take a long time in some cases. Limited Discrepancy Search (LDS) tries to avoid this by reducing the size of the search tree. This is done by allowing the removed customers only to be inserted in their n-best positions, being n the number of discrepancies. A strategy is also needed to choose the order in which the customers will be inserted.

One of the main advantages of LNS is that it doesn't suffer as much from local minima as traditional local search methods do. These methods only perform small changes to the current solution, and because of that suffer greatly from the problem of escaping local minima. We call this method *large* neighbourhood search because it can remove a big set of customers—up to 30% of the customers in the current solution. Removing such a big set of customers provides the engine a very big neighbourhood to explore, and has the potential to perform powerful moves, changing big parts of a solution. The method is able to perform far-reaching changes that allow the search to move out of local minima. The moves allowed by Large Neighbourhood Search are so powerful that it usually doesn't

need any other local minima escaping technique [Sha97]. Those powerful and far-reaching moves also help better handling side constraints, as they can drive the search over barriers in the search space created by side constraints. Cost differences are just a hint to the search and thus can be simple functions related to distance. The cost of solutions helps evaluate new found solutions, which will be accepted only if their cost is lower than the cost of the best found solution so far.

3. The Model

In this chapter we review two models proposed by [KS06] for the Vehicle Routing Problem with Time Windows. We also want to study the implementation of custom propagators and branchers.

3.1 The Path Model

First constraint programming approaches for the TSP were described by [CL97, PGPR98], using path constraints to model the routing. This formulation is adapted to the VRP by introducing the multiple-vehicle concept in [BFS+00, KPS00]. It is further developed and explored in [KS06]. It maintains a variable p_i for each customer i that represents the visit that goes immediately before it. We call the vector p formed by these variables the predecessor vector. Similarly, a successor vector s is maintained, the variables s_i of which indicate the visit that is performed immediately after costumer i. The maintenance of those vectors is redundant, because s is implicitly defined by the values of p through the coherence constraints 3.1. However, maintaining both helps to propagate the constraints quicker. Two special visits f_j, l_j are introduced for each vehicle j. These visits represent the first and last visits of each vehicle to the depot. We will refer to the sets of the first and last visits as F and L respectively. We will refer to the set of all visits as V. Additionally, two variable vectors are used to maintain time windows and capacity along the tours, we call these vectors t and q. For each visit, t_i represents the time the service at visit i starts, while q_i represents the load of the vehicle after the visit i has been performed. Finally a vector v is maintained, each variable v_i of which indicates the vehicle that visits costumer i.

All-different constraint ensures that all values of p and s must be different from each other. This is needed to guarantee that all the clients are visited exactly once.

$$p_i \neq p_j, \ \forall i, j \in V \wedge i < j$$

Coherence constraints enforce that the predecessor and successor variable vectors are consistent with one another. For convention, we will say that the last visit of a vehicle has the first visit of the vehicle as successor, consequently the first visit will have the last visit of its tour as predecessor.

$$s_{p_i} = i, \ \forall i \in V, \qquad p_{s_i} = i, \ \forall i \in V$$

Vehicle constraints indicate that all variables must be visited by the same vehicle as their predecessor. Each first visit $f_j \in F$ and last visit $l_j \in L$ will be assigned their respective vehicle j when setting up the model.

$$v_i = v_{p_i},\ \forall i \in V, \qquad v_i = v_{s_i},\ \forall i \in V$$

Capacity constraints make sure the vehicle capacity is not exceeded. We first need the constraints which will maintain the values of the quantity of goods being carried by a vehicle. For these quantities, we only need to maintain bounds consistency (see section 2.3.1.1). The load of a vehicle j after performing visit i will be equal to the load q_{p_i} after visiting the previous customer in the tour plus the demand r_i of the current customer.

$$q_i = q_{p_i} + r_i,\ \forall i \in V - F, \qquad q_i = q_{s_i} - r_{s_i},\ \forall i \in V - L$$

Then we need, for each visit, to make sure the vehicle load does not exceed the vehicle capacity Q, assuming a homogeneous fleet where all vehicles have the same capacity.

$$q_i \leq Q\ \forall i \in V$$

Time constraints make sure the planned tour meets the time window constraints for all the costumers. Recall t_i represents the time at which visit i starts. Then we have that the start of service for client i is the start of service for the predecessor t_{p_i}, plus the service time for p_i and travel time between the costumers $dist(i, p_i)$. We define $T_{p_i,i} = servicetime_i + distance_{p_i,i}$. In the same manner as with capacity, we need only maintain consistency bounds for the time values along vehicle routes. For the start of service, waiting is usually allowed, for this reason an inequality is maintained instead of an exact equality.

$$t_i \geq t_{p_i} + T_{p_i,i}\ \forall i \in V - F, \qquad t_i \leq t_{s_i} - T_{i,s_i}\ \forall i \in V - L$$

Finally we have to constraint that the start of service for each costumer i fits its time window $[a_i, b_i]$.

$$a_i \leq t_i \leq b_i$$

3.2 The Set Model

The set model described in [PGPR98] presents an alternative to the above described path formulation 3.1 for the TSP problem. The model can be easily extended to make it applicable to the VRP [KS06]. The authors of [PGPR98] report that the model for the TSP resulted in increased propagation by eliminating certain arcs from consideration or enforcing that certain costumers must be visited by different vehicles. Instead of maintaining only a predecessor and successor vector, two sets of visits A_i and B_i are maintained for each client. Those sets represent the visits that come before and after the costumer respectively. The model is defined through the constraints:

- For every costumer i, the intersection of its respective sets A_i and B_i must be empty. This is, any other visit can come either before or after costumer i, but can not come both before and after. This constraint helps forbid cycles in tours.

  $$B_i \cap A_i = \emptyset$$

- If customer j comes after customer i, then customer i must come before customer j.

  $$j \in A_i \Leftrightarrow i \in B_j$$

- If the direct successor of customer i is customer j, then the set of visits that come after i must be exactly the set of visits that come after j plus the same node j.

  $$s_i = j \Leftrightarrow A_i = A_j \cup \{\ j\ \}$$

- The transitive property that if a customer j comes after i, and in turn a costumer l comes after j, then necessarily customer l comes after customer i.

  $$j \in A_i \wedge l \in A_j \Rightarrow l \in A_i$$

- For two customers i, j, the customer j must either come before or after i in the same tour, or they are in a different tour—served by different vehicles.

  $$v_i \neq v_j \vee j \in B_i \vee j \in A_i$$

- If other visits come between i and j, then j can't be the direct successor of i.

  $$A_i \cap B_j \neq \emptyset \Rightarrow s_i = j$$

- These constraints enforce time window constraints by stating that there must be a certain time gap in between two visits when they are ordered in some way in the same tour.

  $$j \in A_i \Rightarrow t_j \geq t_i + \tau_{i,j}$$

  $$j \in B_i \Rightarrow t_j \leq t_i - \tau_{j,i}$$

- Additionally to these constraints described by the authors, we also need similar constraints for the vehicle capacity.

  $$j \in A_i \Rightarrow q_j \geq q_i + r_j$$

  $$j \in B_i \Rightarrow q_j \leq q_i - r_j$$

Despite the good results described in [PGPR98] for the TSP, which described increased propagation in the model, we have found that when applied to VRPTW, the model grows exponentially in size due to the multiple alternatives caused by multiple vehicles. The fact that the variable domain is a large set (all subsets from 0..n) causes it to run very slowly, while the quality of the solution found remains the same. To understand this we need to think about the sets that are being maintained for each variable, A_i and B_i. In the TSP, all the visits are either contained in A_i or in B_i, meaning one of this sets is completely defined by the other set, and for this reason we only have to branch on one of the sets. On the VRP, there is a third possibility: according to constraint e, a visit can either be in A_i, B_i, or visited by another vehicle. In the TSP problem, it is enough to maintain the set A_i for each visit, while in the VRP we need to maintain both A_i and B_i for each visit. Also, in TSP the domain of the A sets is much smaller, as it is strongly constrained by the fact that visits must be either in A or B, while in VRP those domains are very large due to the possibility of being visited by another vehicle. A number of preliminary tests were run on Solomon instances, mostly resulting in "heap memory exhausted" exceptions. Hence, from here on we only consider and work with the path formulation described in 3.1.

3.3 Propagation

Recall that propagators serve as implementation of the constraints 2.3.2. Propagation in Gecode occurs when the *status*() function of a space is called. This happens when the domain of any variable is modified, and all the constrained values that are affected by this modification will be updated—this is what we call propagation. Although Gecode provides many available propagators that implement a wide variety of constraints, it can be useful to implement custom own propagators for specific constraints.

3.3.1 The NoCycle Propagator

This efficient constraint for avoiding cycles was first introduced in [CL97, PGPR98]. For each variable, we maintain a pair $[b_i, e_i]$ representing the nodes at the beginning and at the end of the corresponding chain respectively. A propagation rule enforces that the successor of a variable cannot be the node at the beginning of the same chain, forbidding cycles. This constraint is not applied to last visits, which have first visits as successors. Cycles in tours are already forbidden by various constraints. The most obvious is the all-different constraint placed on the successor vector s. If there was a cycle, then for some variables j and k, $s_j = s_k$, directly violating the constraint. Capacity and time window constraints also guarantee that no cycles can happen, as the inequations maintained would contradict each other. Despite cycles being already forbidden, adding this propagator will help propagate faster.

ut we want to use it for better and quicker propagation. The idea is to fail nodes quicker and be able to explore more nodes in a shorter time.

As far as problems with a very clustered set of costumers is concerned, very similar compute times are required to find a solution. This problems however are normally easy and quick to solve. In all of the more complicated problems though, the model performs faster with the noCycle propagator, especially in problems with wide time windows. The reason for this is that in problems with small time windows, much of the constraint propagation and variable domain restrictions come from the said time windows. The wider time windows are, the less restricted the variable domains are and thus there is more room for the system to make a choice that violates the noCycle constraint.

3.4 Branching

2.3.3 Branchings determine the shape of the search tree by providing the search engine with a choice, which will keep reducing the domains of the variables and try to assign them a value. Branching occurs in two steps:

- **Select a variable.** We first need to choose one of the unassigned values to which we will try to assign a value. At its simplest it can just be the first unassigned value, but other strategies make sense like choosing to first assign the most constrained variable.
- **Select a value.** The next step is to determine which value to assign to the variable we have chosen. Again, we can choose one of the permitted values in the variable's domain at random, but we can also use information we have about the problem to make a better choice.

In the presented model, we branch on the p vector, that is, on the predecessor variables. This means we will try to assign a value to each one of the variables in p. If we can do this, then we have a solution and the other vectors s and v are implicitly defined by the values in p.

3.4.1 Selecting a variable

From the start, two options have been considered within the default branchings that Gecode offers:

- **INT_VAR_NONE.** This default brancher will simply assign a value to the first unassigned value that it encounters, without taking into account the variables assigned so far nor the domains of the remaining variables.

- **INT_VAR_MIN_SIZE.** This brancher will take into account the domains of the variables that have not yet been assigned. It will search for the variable with the smallest domain size, that is the most constrained variable so far. The idea is to assign the most constrained variables first to prevent states without possible solutions.

This strategies make sense each one in their own scenario (see section 4. Evaluation) and for this reason it has not been modified nor has the need to implement new selection methods arisen.

3.4.2 Selecting the value

The default brancher provided by Gecode assigns the minimum value from the domain to the variable. This doesn't make much sense when treating with vehicle routing problems, so it is desirable to design a better way to assign a value. Two new branchers were implemented, the minimum distance brancher and the time brancher.

3.4.2.1 The minimum distance brancher

Once a variable has been selected, this brancher will consider the distance between the node and all the possible domain values to assign the node with the minimum distance as the predecessor. The calculation of this minimum distance poses obviously an overhead in the brancher every time a new choice has to be computed, resulting in longer computation times. However, preliminary experiments show that the quality of the solution found is much better, making for a good trade-off between computation time and solution quality. This strategy works with similar results independently of the strategy used to select the variable.

3.4.2.2 The time brancher

Another idea when assigning a value to a variable is to assign as a successor to a variable the node with the smallest time window. We understand that the smaller the late window value is the more likely it is that this node will need to be visited first. The idea is to visit first the nodes that are more urgent. The interesting thing about it is to avoid making assignations to variables that will inevitably lead to failed nodes, to find solutions in a smaller computation time. As it doesn't take into account any distances, the quality of the solution is expected to go down. However, the idea is to combine this brancher with the minimum distance brancher described above to find a good balance between computation time and solution quality. However, as preliminary tests showed this brancher alone didn't present good results, we didn't explore this option further.

4. The Search Engine

Gecode provides already built engines for Depth-First Search and Branch-and-Bound search. We are however interested in Large Neighbourhood Search, and so we want to design a new Gecode search engine that implements it. Various decisions over all the available options must be taken when designing the engine. First of all, we need a construction heuristic that will be used to determine an initial solution from which to start the search. We need to define an improvement heuristic that will determine how Large Neighbourhood Search is performed. This improvement heuristic will be basically defined through the relaxation and reinsertion methods—that is, how the solution will be destroyed and rebuilt. As for the relaxation method, the main decision will be how to choose the customers to remove, the reinsertion process will determine how those removed clients are inserted. As the whole process of choosing a neighbourhood, destroying the solution and reinserting the removed clients is not exactly described in [Sha98] we want to explore various options and decisions that present themselves when designing the search engine.

4.1 General Algorithm

We present the general algorithm for Large Neighbourhood Search (Algorithm 4.1), based on [PR10]. A variable x^B is maintained throughout the algorithm, it is the best solution found until the current time. The algorithm receives an instance problem from the entry and searches for an initial solution using the construction heuristics. This solution is used as first best found solution. Then it enters the loop, the first thing it does is a copy x of the current best solution. The following steps are destroying it using the function $destroy(x)$ and rebuilding it using the function $rebuild(x^D)$. Once the solution has been rebuilt, we have to see whether we accept the new solution or not. Typically, we will accept any valid solution the cost of which is lower than the current best solution. That is, we will accept any valid solution that improves the current solution. In this case, the variable x^B is updated to the new found solution. When the neighbourhood size becomes too big, it is too expensive to determine that we are truly at a local minima, and thus some other stop condition must be placed on the search [KPS98]. This condition is usually a time limit that accounts for how long the user is disposed to wait to find a solution. In our case a

time limit of 900 seconds will be placed on the whole search to receive an answer. It will naturally end sooner if the whole search space until a neighbourhood size of 30 is explored.

Algorithm 4.1: General algorithm for the Large Neighbourhood Search.

```
1 Input instance problem initSol = find initial solution(problem)  x^B = initSol ;
2 while stop criterion not met do
3 |   x = x^B ;
4 |   x^D = destroy(x) ;
5 |   x^R = rebuild(x^D) ;
6 |   if accept solution(x^R) then
7 |   |   x^B = x^R ;
8 return x^B
```

4.2 Construction Heuristic

The first step is to determine the initial solution. It is important to choose how the start solution will be determined because it is the starting point from where to perform Large Neighbourhood Search. Various methods have been tested to evaluate various ways to obtain an initial solution.

- **Using one of the default engines.** We can use the default Branch and Bound or Depth-First Search engines, combined with our custom brancher (see section 3.4.2.1), to determine an initial solution. The solution obtained by this method is usually of a good quality. However, the BAB and DFS engines can take a very long time to find an initial solution, especially for instances with wide time windows, sometimes even exceeding the total time limit set for the whole search. The quality of the initial solution rarely pays off for this high temporal cost.

- **Start with one tour per vehicle.** A very simple option is to start with a vehicle per client [Sha98]. For the Solomon instances, this means we start with 100 vehicles. Although the quality of this initial solution is obviously as poor as it can be, the solution is very quickly improved to one of similar quality as the one found with the BAB engine. This quick improvement is possible thanks to the low cost of moves when only a single client is removed, allowing to perform a long series of successful moves in little time.

- **Savings method.** A third alternative is the savings method [CW64]. It also starts with one tour per client, but the idea is to use the savings method before running LNS to try and start from a better start point. The method computes the cost for every pair of nodes, in this case the distance and ranks them accordingly, from lowest to highest cost. It does then iteratively try to link the cheapest pair of clients until no more moves are possible. This method improves the solution in a relatively low time, but no big differences can be observed with respect to starting LNS right after assigning one tour per client.

4.3 Improvement Heuristic

After finding a first solution, the next step is to improve this solution. We will do this by iteratively removing sets of customers and trying to reinsert them in the solution.

4.3.1 Relaxation

The next step is to determine how we will destroy the current solution. To do that we want to remove a set of clients from the solution. The most important decision in this step is deciding which clients to remove, and the relaxation method will be implicitly defined by the function that makes this choice. The most basic method is to remove a set of clients chosen at random, but this doesn't make any use of the information we have about the problem or the state of the current solution. Using these informations, we ideally want the costumers that we choose to remove to be maximally related according to some relatedness measure, to avoid removing sets of clients that are very likely to be reinserted in the same positions, leading to no improvement in the solution. The main part of designing the relaxation method consists in defining said relatedness measure.

4.3.1.1 Neighbourhood Size

Another important question that arises is how many clients to remove from the solution. It is best to remove a set of costumers as small as possible, as their reinsertion step is very cheap, but we also want to remove sets of clients big enough as to allow large parts of the solution to be changed and make it easier to escape local minima. Therefore we will start removing a single client and increase the neighbourhood size once no more improvements have been found after a certain number of failed attempts. A parameter will indicate how many consecutive failed attempts will be performed before jumping to the next neighbourhood size. The higher this number is, the more stubbornly will the engine try to find a solution in a given neighbourhood size before increasing the number of clients to remove.

4.3.1.2 Relatedness Function

As said in section 4.3.1 we want to remove clients that are maximally related to each other. The set of customers to be removed from the current solution is implicitly determined by the relatedness function. A very basic method that considers relatedness is to choose a random seed node and remove the set of visits that are closer geographically. A better option suggested in [Sha98] is to consider also whether the visits are or not on the same tour as the seed node. This helps by optimization of the number of vehicles used, as to reduce the number of vehicles we need to relax all the visits on one tour so that they can be re-inserted somewhere else.

$$rel(i,j) = \frac{1}{dist(i,j)+v(i,j)}$$

Where $dist(i,j)$ is the distance between the nodes normalized in the range [0..1) and $v(i,j) = 0$ if the clients are in the same tour and $v(i,j) = 1$ otherwise.

This function will always attempt to remove first all the clients in the tour, as their relatedness will always be higher than those in other tours. This is important for vehicle optimization, as reducing the number of vehicles used requires the removal of entire tours.

Determinism

A determinism parameter is used to introduce a certain degree of randomness to the algorithm. This randomness is used when removing clients and it will cause the engine to take not always the most related client to the seed node. It indicates how strongly the engine relies in the relatedness function. With determinism = 0, relatedness is completely ignored and the nodes to remove are randomly selected. With determinism = ∞, the algorithm takes always the most related client. The importance of this factor lies in

the fact that if we rely entirely on the relatedness function, it is pointless to perform more attempts than the total number of clients, as the removal of a given seed client will always result in the removal of the same neighbourhood and perform the same reinsertion attempts. By introducing a certain randomness we allow that different attempts, even with the same seed node, result in slightly different neighbourhoods.

4.3.2 Reinsertion

Once the solution has been destroyed, the next step is to try and reinsert those clients in the solution, hopefully in a better way that improves the quality of the solution. It is possible to use the default branchers or the custom ones described in section 3.4.2.1 to explore the whole search tree of the destroyed solution, that is, consider every single insertion position for each one of the removed customers with hopes of finding a better solution. However, although for some reinsertion steps, especially with neighbourhoods of small size, the engine can quickly find out a solution or determine that no better solution exists, in other occasions exploring the whole search tree of the destroyed solution can take a very long time. As a great number of reinsertions are made during the whole process, we wish that this step is as fast as possible, and a high overhead on this process can lead to very high computation times. The alternative to try and compute this reinsertion step faster, is to not explore the whole search tree but only a part of it. If we can find a way to explore only the best branches of the search tree, then we will quickly find out whether there is a solution that improves our current best or not. For this we will use Limited Discrepancy Search introduced in [HG95]. It is also important to choose a way to determine in which order will the customers be reinserted. As for the reinsertion method, a series of different strategies have been implemented and tested to the effect of finding the optimal way to reinsert the removed nodes. The main reinsertion variants revolve around fixing or not fixing the customers when they are inserted. Fixing them will mean that, when inserting customer j after customer i, we will unequivocally set $p_j = i$ and $s_i = j$. The alternative is not to set this and keep the domains bigger to allow other insertions to be made between those nodes.

4.3.2.1 Limited Discrepancy Search

The idea behind LDS (see [HG95]) is to explore only the best possible insertion positions for the removed customers, assuming that, if there is a better solution to be found within the search tree, it will be one involving those values for the removed costumers. We will explore only the d best insertion points for the nodes, being d the number of discrepancies. We call a discrepancy the insertion of a removed visit in its second best position. We consider two discrepancies either the insertion of a visit in its third best position or the insertion of two visits in their second best position. The parameter d indicates how stubbornly the engine will try to find a new solution within a given neighbourhood. The more discrepancies we allow, the longer the engine will try to find a better solution by inserting the removed costumers in different positions, even when they are not the best positions for those customers.

4.3.2.2 General Algorithm

We present the general algorithm for the reinsertion process based on [Sha98], using Limited Discrepancy Search. It is implemented with a recursive function (see Algorithm 4.2). The algorithm receives the current solution, the number of discrepancies parameter and a vector that contains the customers that are yet to be inserted. On the first call, these are precisely the destroyed solution and the set of removed customers. First of all, the algorithm checks whether there are still customers to be reinserted. If this is not the case,

then it means we have effectively reinserted all the customers and thus have a solution. If it is better than the best found so far, we will accept it. If there are still customers to insert, the algorithm chooses one of the removed customers to be inserted using the insertion order heuristic, and computes the allowed insertion points for that customer and ranks from lower to higher cost. Then it tries, while there are still valid insertion points and we haven't used all the allowed discrepancies, to insert the chosen customer in the next best position, and recursively call the function to try and insert the remaining customers. Before inserting, a clone of the current solution is done so that we can backtrack to it after the reinsertion process, in the case that it has not produced a better solution.

Algorithm 4.2: General algorithm for the reinsertion step.

```
1  Input current solution x, number of discrepancies d, removed clients removed ;
2  if |removed| = 0 then
3      if cost(x) < cost(x^B then
4          x^B = x ;
5  else
6      toInsert = choose customer to insert(removed) ;
7      insertionPoints = calculate and rank insertion points(toInsert) ;
8      i = 0 ;
9      for p in insertionPoints and i <= d do
10         x^C = clone(x) ;
11         insert(x, toInsert, p) ;
12         reinsert(x, removed-toInsert, d-i) ;
13         x = x^C ;
14         i = i+1; ;
```

Choosing Which Customer to Insert

We have to reinsert all the clients back in the destroyed solution. The order of insertion of these customers is of big importance, as every insertion affects the domains of all the other variables. To choose which variable we will assign first, we have considered two heuristics, the most-constrained heuristic and the farthest insertion heuristic:

- **Most-Constrained heuristic**. Consists in assigning a value first to those variables with less possible options available, to try and avoid that their domains become empty and fail the model.
- **Farthest insertion heuristic**. For each removed client, we consider the set of its insertion points as the set of nodes after which the removed client can be inserted. The cost of each insertion point is defined as the increase in distance when inserting the removed node. We choose as the variable to insert the one for which the cheapest insertion point is largest. The basis for this is to minimize the maximum cost by inserting first the node with largest distance increases, for inserting it at the end could cause the total cost to increase highly.

Inserting Variables when Only One Position is Left As an optimization for the Farthest Insertion heuristic, as soon as one of the customers yet to be reinserted is left with only one possible value in its domain, it is immediately inserted in that point—note that the Most Constrained heuristic does that implicitly. This can hopefully avoid the failing of a state due to the domain of this variable becoming empty and increase efficiency resulting in the exploration of larger neighbourhoods in less time. This aims to combine the good parts of both methods: Using information about the problem provided

through the Farthest Insertion heuristic and avoiding explorations that will most likely lead to empty domains through inserting variables when only one position is left. Another way to see it is as using the Most Constrained heuristic only if the domain of a variable has only one value, and using the Farthest Insertion heuristic otherwise.

4.3.2.3 Fixing Clients

In this versions, when an insertion point is decided for a client, the predecessor variable for this client is fixed to be equal to the client after which it is inserted. This means that if customer j is inserted after customer i, then $p_j = j$ and $s_i = j$. This method is very simple, as it allows us to use Gecode both for solution checking as well as constraint propagation. The main issue with this version is that, after this insertion, no more nodes can be inserted between the nodes i and j. This is so because domains in Gecode can never become bigger, meaning that once an assignment has been made, it is not reversible and that customer j will always have the customer i as predecessor, unless we decide the search has failed and we decide to backtrack. An alternative to these methods is presented in the next section which aims to solve this problem. Another question is what to do with the clients that we don't remove: we can either leave them fixed or allow removed clients to be inserted between them:

Fixing the non-removed clients

The first option implies that we will only allow to change the part of the solution that we have destroyed, re-organizing those clients in the same tour or moving them to other tours next to other removed clients, while the rest of the solution will remain exactly the same (see Figure 4.1). The main advantage of having everything fixed is that it is much faster to explore the neighbourhoods due to them being much smaller, unfortunately this very same fact also causes that some solutions that would be possible are passed away and not found.

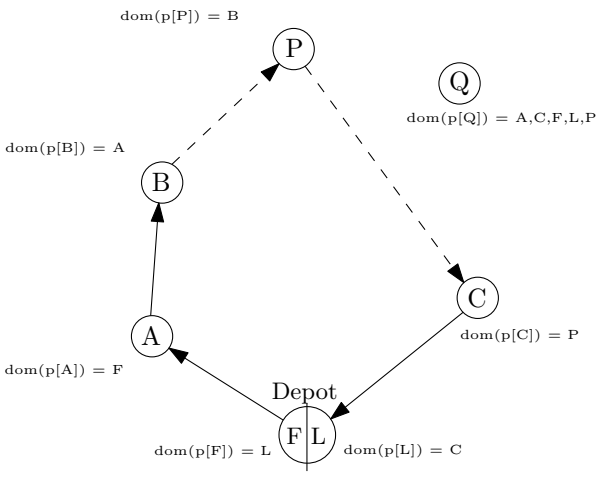

Figure 4.1: Domains of the clients after reinserting the client P in the tour in the all nodes fixed version.

Allowing clients to be inserted anywhere

The second option allows for clients to be reinserted anywhere in the solution. All non-removed nodes here have as predecessor variable's domain the set of removed costumers plus their predecessor in the current solution. When a node is inserted its predecessor will

be automatically removed from the domains of the other clients, as per the all-different constraint (see Figure 4.2). Obviously the neighbourhood is here bigger, and although it will take longer to explore the removed neighbourhoods, there is the hope that better solutions will be found. The method still suffers from the fact that no nodes can be further inserted before the node that we have just inserted, which limits the search options.

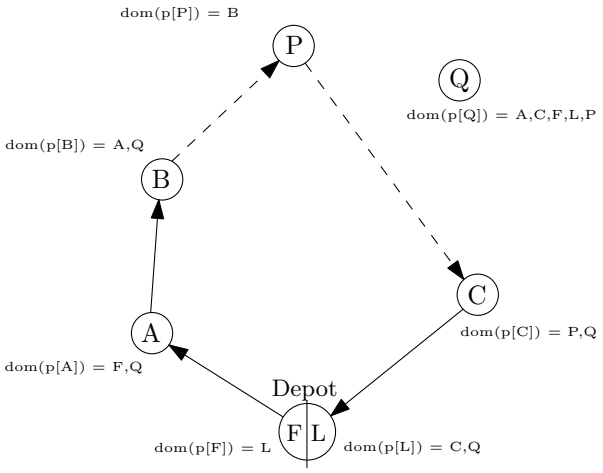

Figure 4.2: Domains of the clients after reinserting the client P in the tour in the version allowing clients to be inserted everywhere.

4.3.2.4 Costumers Left Free

The idea behind this method is to allow removed clients to be inserted after any of the non-removed nodes. To allow this, no node link can be fixed until the end, because this would cause the model to always fail when only one removed client is left, as the domain of the predecessor for this client is empty. This is because due to the all-different constraint, all p_i variables must be different, and at the same time a node can't have itself as predecessor. The insertion of nodes will be done instead by adjusting the domains of the nodes according to the insertion. A solution is found when all the removed nodes have been successfully reinserted. If the domain of any variable becomes empty during the process, the *status*() function of the model will report that it has been failed.

As this method requires the shrinking of domains and propagation of some constraints to be done manually, which leaves the Constraint Programming environment almost only as a solution checker, used to determine whether a model can still be further explored, has been failed or is already a solution.

Passive Representation

In this method, instead of inserting clients directly to the model, the domains of the variables will be manually updated to be consistent with the current state of the solution. The model now only contains a set of constrained variables with their corresponding domains for predecessor and successor variables. As we want to keep the options open for the removed customers to be inserted anywhere, even the variables that have already been assigned a predecessor will have not only that predecessor in their domain, but also other customers that are yet to be inserted—those could come in between that variable and its current predecessor. Because of that, the model loses the information about which customers are actually assigned together in the current solution. To avoid losing this information, a passive representation will be maintained to describe the actual state of the

solution. It consists of two additional vectors. These vectors represent how would the predecessor and successor vectors be in the original model if the customers actually inserted in the solution. When costumers are removed, this passive representation represents the tours that the the non-removed customers form. When customers are inserted, the vectors are updated to include that customer in the corresponding tour.

Domain Adjusting

This method is based on adjusting the domains of the variables to make them consistent with the actual state of the model. This has to be done both when destroying the current solution—as the removal of variables changes variable domains, and as well when inserting clients, as this will make the set of possible values left smaller.

Removing Clients from the Current Solution When we have a valid solution, all the p_i and s_i variables are set and this means the domain of each of these variables has one and only one value. After removing clients, there will be many values again open for those domains. As the domains of the variables can never become bigger in Gecode, we will part from an empty model, completely unrestricted, and set the domains of the variables according to the removed clients Figure 4.3. For this task there are three kinds of clients:

- **The first visits.** Those visits representing the starting point at depot will always have the corresponding last visits as their predecessor. This is enforced through constraints in the model so no further work must be done for those clients.
- **The removed clients.** These are the clients whose domain becomes bigger. The domain for their predecessor variables will include virtually all the non-removed clients. This is, they can be inserted after any of the clients that have not been removed as long as they are not forbidden by time windows or capacity constraints. The rest of the removed clients must also be included in the domain, because although they will not be considered as insertion points while they remain free, they could be once if they are inserted before the current node, as explained in the following section.
- **The non-removed clients**. The predecessor variable for these clients can take any of the removed clients yet to be inserted plus their predecessor in the current solution, that is, they can either keep their current predecessor or get one of the removed clients inserted before them—again, unless time windows or capacity constraints forbid it.

Reinserting Removed Nodes in the Model Each time a removed node is inserted, the domains of most of the rest of variables will be shrunk Figure 4.4. Again, the first visits don't require any treatment, they keep the corresponding last visit as predecessor. There are four different cases that must be considered when inserting a node:

- **The inserted node.** The inserted node becomes a completely normal non-removed node when inserted and will be treated as such from here on. This means that the domain for this node will be, as for all the other non-removed nodes, the current predecessor in the solution plus the nodes that are still free. This translates in removing all the non-removed nodes from the domain of this client except for the one after which it has been inserted.
- **The successor of the inserted node.** The old predecessor for this node is not a valid option anymore, as a node has been inserted in between and this node is now the current predecessor. As it was already in the domain of that variable, the only thing that must be done is remove the old predecessor which is not valid anymore from its domain.

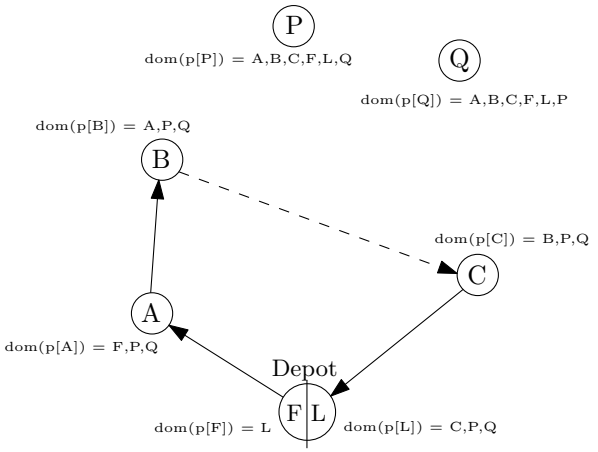

Figure 4.3: Domains of the clients after removing the nodes P and Q from the tour.

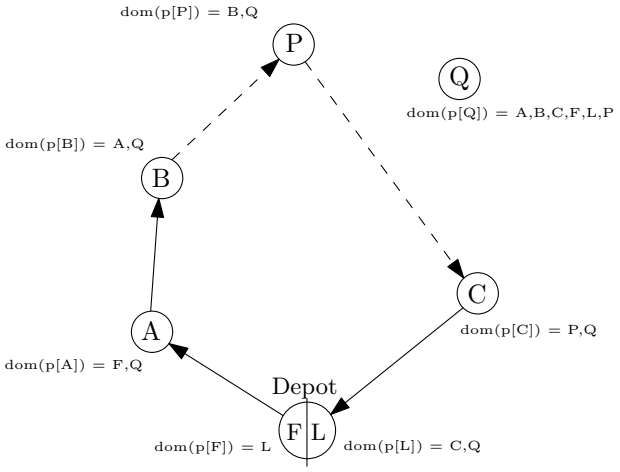

Figure 4.4: Domains of the clients after reinserting P back in the tour.

- **The rest of non-removed clients.** Those nodes can no longer have the inserted node as their predecessor because this node is no longer free. The inserted node must thus be removed from their domains.
- **The nodes that remain free.** Nothing must be done for the nodes that are yet to be inserted. They keep all the options open, including all the non-removed clients, the removed ones and the inserted node as well.

Constraint Propagation

We need to remember that constraint propagation in Gecode happens when the *status*() function of the space is called. When a variable is assigned, the domains of all the other variables will be reduced according to that assignment. Due to the variables not being fixed in this version, the *status*() function cannot perform propagation on the time and capacity vectors for that assignment, because although the customer has been assigned a predecessor in the passive representation, the domain of this variable in the active model can still contain other values. As we want to keep them coherent with the current state of the solution that we have in the passive representation, constraint propagation for those variables must be performed by hand. In practice, this means that when a node is inserted

in a tour, the time window and capacity variables must be updated for the nodes in the same tour Figure 4.5. Always starting from the inserted node until we reach the depot, we need to do two kinds of propagation:

- **Forwards Propagation.** The minimum values for t and q must be updated for all the nodes that come after the inserted client, as they are no longer valid. This means that the insertion of a node before them will most probably cause the start of service for those clients to start later.
- **Backwards Propagation.** In the same manner, maximum values for the nodes that come before the inserted client must be updated. The insertion of a client after those visits means that the service time for visits before will be constrained to start sooner.

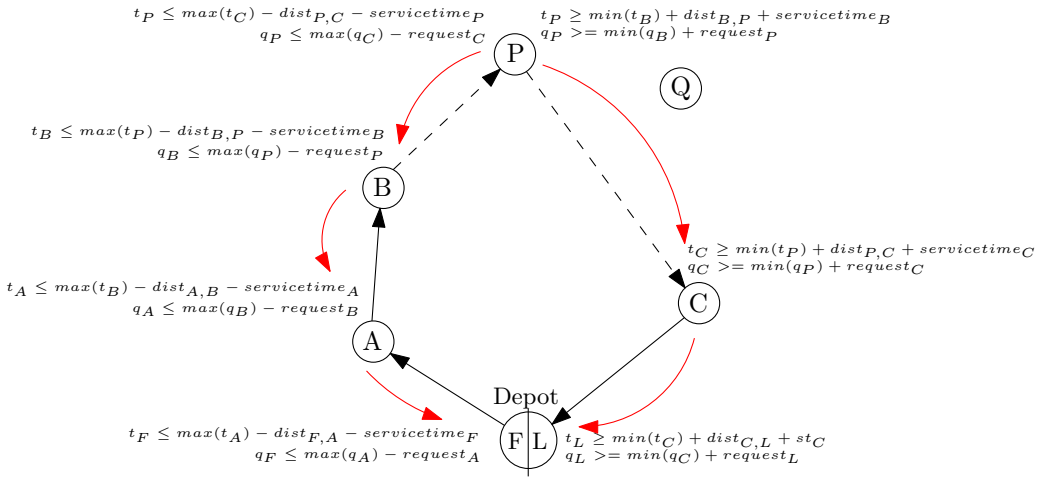

Figure 4.5: Forwards and backwards propagation of the time windows and capacity constraints after reinserting node P in the tour.

Finally, we also have to take this into account when destroying the solution, and propagate this constraints after the removal of clients. It is done in the exact same way as for the inserted clients, but it must be done for all tours where clients have been removed from. For simplicity the constraints are propagated forwards from the first to the last node and similarly backwards from the last to the first node.

4.3.2.5 Time Limit on Reinsertion

Observing the results of each iteration, one can see that most of the iterations where a solution is found, this solution is found quickly. In other words, it can be seen that most of the reinsertion iterations that take long time don't end up in the finding of a better solution, while on the contrary in most of the iterations where new solutions are found the reinsertion time is relatively low. Based on this observation a time limit can be placed on each iteration to avoid long reinsertion steps that most likely won't find any better solution.

4.3.2.6 First-Found, all-solutions

Another decision to take is whether to consider the first solution found or keep searching for better ones. Finding all the solutions does not seem a good option, as when a better solution is found it is unlikely to become any better by finding more solutions due to our

LDS engine inserting the clients in their best positions. It does seem clear then that we will halt when a solution is found. We have however two options:

- Stop when a solution better than the current best solution is found.
- Stop when a solution is found, be it better or not than the current best solution. It can be similarly argumented that the first solution found is most likely to be the best we will find in during that reinsertion iteration.

4.3.3 Objective Function

The search in Gecode needs an objective function or cost function that returns the cost associated with the current solution. This function will help decide whether a solution found is better or not than the current best solution. As an important part of our problem is to optimize the number of vehicles used, using the distance travelled is not enough, and a high penalty for each vehicle used is added to the cost function besides distance travelled. In addition to providing better guidance for the search, it allows small increases in the total distance travelled if this means using less vehicles. As we never accept worse solutions, this would never be possible if the cost function only takes the distance into account, but with this penalty per vehicle, even if the distance goes slightly up, the total value of the objective function decreases, making this move possible.

4.3.4 Implementation in Gecode

Gecode doesn't provide a way to unassign values, because this is a really complex problem that doesn't only involve the value itself but the propagation of all the constraints and rules that come with this assignment. Relaxing visits in Gecode thus involves creating a new unconstrained model and adjusting the domains of the non-relaxed variables to take the corresponding allowed values. Reinsertions will always be done on a clone of the current Space. In case backtracking is needed because the reinsertion we tried did not work and we need to try another choice, then we still have the original state of the model at that point. This need to constantly clone the current state of the model before each reinsertion causes this clone operation to be one of the most time consuming in the whole process, taking up to around half the execution time.

5. Evaluation

The objective of this section is to analyse the data obtained from the run tests to evaluate the performance of each method and understand why some methods perform better than others. First of all we will quickly evaluate and review the NoCycle propagator from 3.3.1 and the Minimum Distance brancher from 3.4.2.1. Then we will proceed to analyse the LNS engine. The construction heuristics will be evaluated to see if starting with the savings method is useful. As for the improvement heuristics, we will present a number of tests regarding the three main reinsertion methods and the farthest insertion heuristic. Relaxation will be analysed based on tests on the relatedness function. Many tests have been also run to determine the effect the main parameters in search have—number of discrepancies, attempts and determinism. Finally, we will test other options like inserting a variable immediately when only one client is left, objective function and stopping search when the first solution is found.

Environment

We compiled the programs using GCC v4.5.0 with full optimization—using the flag -O3. All tests have been run on one core of a 4x12-core AMD Opteron Processor 6172 machine clocked at 2.6 GHz, running Linux Suse 2.6.34. It has 128 KiB L1 cache (per core), 512 KiB L2 cache (per core), 12 MiB L3 cache (per socket) and a total of 256 GiB RAM.

Inputs

To test the implemented models, the Solomon benchmark instances [Sol87] have been used. These are problems for which best-known solutions are provided. The instances can be divided using two criteria: The geographic distribution of the customers and the vehicle capacity.

Regarding the distribution of the clients we have three kinds of instances:

- **The C instances.** These instances present a set of clustered customers (see Figure 5.1). Given the clustered distribution, these problems are usually easy to solve with an optimal number of vehicles.
- **The R instances.** These instances have a set of customers that are randomly distributed (see Figure 5.2).
- **The RC instances.** These instances have a mix of clustered and randomly distributed clients (see Figure 5.3). They are a mix of the C and R instances.

All the instances in each of these sets presents the exact same distribution of customers. In each set, the instances are divided in groups of four—that is 101..104, 105..108, etc., except for c109 that is left alone. Within each group, time windows widen progressively, with the first in the group having very narrow time windows and the last in the group having very wide time windows—in some cases almost no time window restrictions.

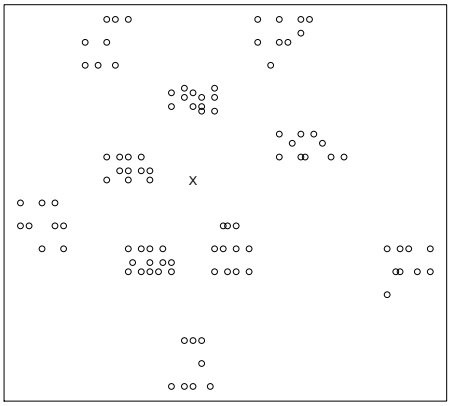

Figure 5.1: Distribution of clustered customers found in the C-Type instances. The point marked with a cross represents the depot.

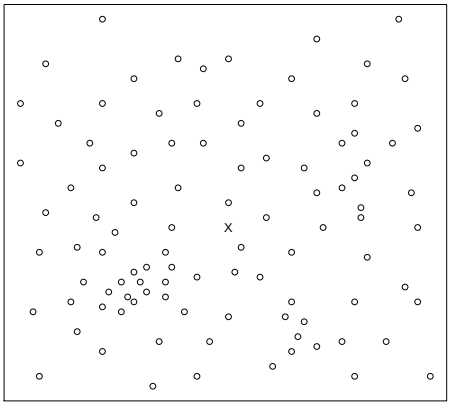

Figure 5.2: Distribution of randomly placed customers found in the R-Type instances. The cross is marked with a red point.

With respect to tour length we have two kinds of instances:

- **C1, R1, RC1 instances.** The vehicle capacity for this instances is relatively low, resulting in a relatively high number of tours of short length, as the capacity constraints do not allow the tours to become much longer.
- **C2, R2, RC2 instances.** This instances have a high vehicle capacity, which allows the tours to be very long. These instances usually can be solved with 2-3 vehicles that do a very long tour.

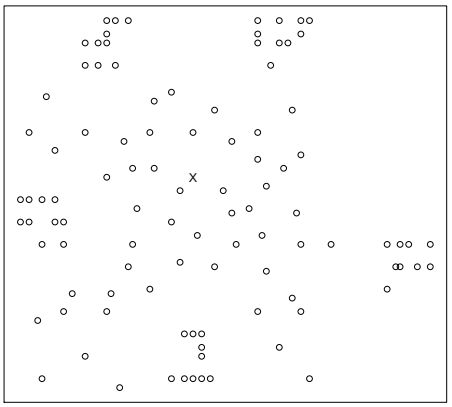

Figure 5.3: Mixed distribution of clustered and randomly placed customers found in the RC-Type instances. The point marked with a cross represents the depot.

This evaluational chapter of the project focuses on the C1, R1 and RC1 instances, as they are the ones that are suitable to be effectively solved by Large Neighbourhood Search methods. The C2, R2 and RC2 are usually solved with three or four vehicles, each of which has a length of about 25 customers. This means that to remove a whole tour we need to remove a very large set of customers. As vehicle optimization usually requires the removal of two or even three complete tours, that would mean removing around 50 or 60 customers, which is more than half of the solution. The LNS engine proposed gets to remove only a 30% of the customers, which is already a large number.

Methodology

As our algorithm has a part of randomness, for each instance ten runs of the program have been executed in parallel with different random seeds. The best solution obtained from the ten runs has been taken to analyse. A time limit of 900 seconds has been placed on those tests, unless specified otherwise. In most of the tests we left the C-Type instances out of the analysis, the fact is that these problems are the easiest kind and most methods always find the optimal solution, at least in number of vehicles, and with results in terms of distance varying only very slightly. This fact causes the analysis of this data not to bring a lot of insight into the performance of the methods.

5.1 Propagators and Branchers

Although our custom brancher and propagator are not finally used in our final Large Neighbourhood Search engine, it is interesting to quickly test their performance on a Branch and Bound or Depth-First Search engine. We first present the results for the NoCycle propagator. Secondly, we analyse the in-built variable selection methods provided by Gecode, and finally we evaluate our custom brancher, the distance brancher.

5.1.1 NoCycle Propagator

The NoCycle propagator forbids cycles within a chain of customers. It aims to increase propagation and speed up the process of finding an initial solution. Tests have been run to find out whether it really accomplishes this objective, as well as running an analysis on propagation data to see how this affects search. As evidenced by Table 5.1, using the

propagator results in increased propagation. Around 90% more fails occur when using the nocycle propagator, which means spaces are failed before. This results in avoiding a lot of useless search. It is also evidenced by the number of nodes explored. Around 75% more nodes are explored in less time when using the nocycle propagator. For the C-Type instances this doesn't result in a tangible win in terms of computation time, as these problems are easy to solve and an initial solution is very quickly found. On the R and RC instances though, the computation time for the initial solution is reduced almost a 15%.

Table 5.1: Table showing execution times, fails and nodes explored, as well as win percentages when using the nocycle propagator.

	C-Type	R/RC-Type
Average Time	0.32	0.41
Average Fails	9.71	16.95
Average Nodes Eplored	144.41	158.89
Average Time (NoCycle)	0.32	0.35
Average Fails (NoCycle)	269.18	297.34
Average Nodes Explored (NoCycle)	663.35	719.68
%Time Win	0,91	14.49
%Fail increase	96.39	94.30
%Nodes explored increase	78.23	77.92

5.1.2 Branchers

Recall that branching in Gecode has two steps: Selecting a variable and selecting a value (see section 2.3.3). A new brancher has been implemented to improve the default branchers that use no information about the problem. It uses information about the problem, concretely the distance between nodes, to assign a value to a variable, and uses the built-in strategies for variable selection.

Selecting the Variable

We have two already built-in options for variable selection:

- **INT_VAR_NONE.** Assign a value to the first unassigned variable.
- **INT_MIN_SIZE.** Assign a value to the variable with smallest domain—that is, most constrained variable.

Although it could initially seem that assigning first to the variable with the smallest domain would yield the best results, this is not always the case. For problems with a big fleet available, it is much better to assign a value to the first unassigned variable first. This is due to the fact that the problem is much less constrained and because of that, the domains of the variables remain considerably big. On the other hand, with problems with a narrow number of vehicles where the variables are much more constrained, the domain of the variables are quickly reduced, leading to more failings of the space. Fore these problems, it is much better to choose the most constrained variable to minimize this failing. Using this method, a solution is found faster. On these problems, assigning the first unassigned variable performs poorly.

Preliminary tests have shown that using the most constrained selection when the number of vehicles is very restricted is not a good idea, even failing to find solutions for some of

the C-Type instances. When the number of vehicles is not that restricted, as is the case of the Solomon instances, using first unassigned variable performs really good for C-Type instances. This is again, because those instances are easy. Using most constrained variable takes longer to find a solution, and even fails to find one in the time limit placed for the instance c103. However, when moving to more complicated problems, namely R-Type and RC-Type, the first unassigned method fails to find a solution for 55% of the instances on the placed time limit, while using the most constrained only fails in 25% of the instances, namely those with very wide time windows.

Selecting the Value: Distance Brancher

The distance brancher aims to reduce total distance values by always assigning as a predecessor for a customer, the customer that is geographically closer. Tests have been run to see if this is accomplished and which impact does that have on computation time. We show analysis based on those instances for which an initial solution could be determined within the 900 seconds time limit for both methods. There were a series of instances with wide time windows for which both methods failed to find a solution, and other when only one of them could—. For those instances that both methods could solve, tests result in the whole computation being made around 80% faster. The solution quality in terms of distance improves—that is total travelled distance goes down, around 44%.

5.2 Construction Heuristics

We want to determine which of the options we have to obtain the initial solution is better: Using one of the default engines, starting with the savings method or starting directly with one vehicle per customer. Preliminary tests have shown that it is not viable to find an initial solution with the default engines 5.1.2, as for problems with very wide time windows it can take more than the 15 minutes placed as time limit for the whole search only to find that initial solution.

Having discarded the default engines option, it remains to see if the savings method adds any benefit with respect to searching directly with LNS after starting with one tour per vehicle. As can be seen in Table 5.2, the results are very similar for most instances. However, in some cases using the savings method before launching LNS results in better solutions (r110, r111, rc107), using one vehicle less, and it only performs worse for the instance r106 (one vehicle more). Although there is not a big difference between the two methods, applying the savings method seems to help with some instances, and the biggest downside it presents is a low increase of travelled distance on most instances.

The fact that both methods perform so similarly is because when starting with a vehicle per customer, the search engine does moves that are very similar to the savings method. Recall that at the start of LNS the neighbourhood size is one, meaning only one single client is removed (see section 4.3.1.1). While the savings method will always take one of the customers that is still alone in a tour and try to insert it in another tours, LNS selects a customer randomly and as such can choose a customer that is already together with other customers in the same tour. However, removing a costumer that is already coupled with another one will never result in opening a new tour due to the vehicle penalty causing the objective function to rise greatly. On the other hand, when removing a customer that is alone in a tour, we will always put it together with other nodes—as long as capacity and time window constraints allow it, due to the objective function now sinking when eliminating a vehicle. Finally, as the lowest tested number of attempts is 250, even with the randomness factor we can be almost sure that all nodes will be chosen at some point, and so they will be placed within other tours. All together, we could say that, in the first step where the neighbourhood size is one, the LNS engine does perform in a very similar

Table 5.2: Comparative of the performance on distance and number of vehicles starting with one vehicle per tour (init) and for the savings method.

Instance	Vehicles (init)	Distance (init)	Vehicles (savings)	Distance (savings)
r101	20	1695.79	20	1684.59
r102	19	1529.07	19	1520.44
r103	14	1265.32	14	1277.91
r104	11	1062.91	11	1042.70
r105	16	1403.68	16	1405.82
r106	13	1289.56	14	1281.91
r107	12	1116.89	12	1123.89
r108	11	994.33	11	1006.51
r109	14	1233.84	14	1239.85
r110	13	1146.32	12	1139.32
r111	13	1104.25	12	1103.78
r112	11	989.92	11	1048.89
rc101	17	1711.34	17	1709.37
rc102	15	1547.61	15	1553.86
rc103	13	1354.11	13	1402.81
rc104	11	1194.70	11	1303.46
rc105	16	1598.80	16	1584.40
rc106	14	1468.65	14	1524.74
rc107	13	1303.16	12	1309.16
rc108	12	1222.51	12	1220.08
Sum	278	26232.76	276	26483.49

way to the savings method. We can see that in the time the savings method takes to find the initial solution, the search engine reaches a solution of even a better quality by making lots of moves removing only one customer, as shown in Table 5.4 . However in the three mentioned instances the savings method performs better, which indicates that in those very concrete cases the pairings the savings method does provide a better basis for further optimization.

5.3 Improvement Heuristics

This section aims to analyse the performance of the different improvement heuristics and explore all the options and parameters that are involved in the process. First of all we evaluate the three main insertion methods and the heuristics for choosing the node to reinsert. We also develop the question of when should the reinsertion process be stopped, for example by accepting the first found solution or placing a time limit on reinsertion. Secondly we evaluate the relatedness functions, and its effectiveness in comparison to random selection of clients. Finally we evaluate various values for the main parameters of the search engine.

5.3.1 Reinsertion

The objective of this section is to evaluate the performance of the three main reinsertion methods, summarized in Table 5.3. First of all, we want to compare the two methods that fix customers when reinserting them. These are the Fixed and Partially Fixed methods.

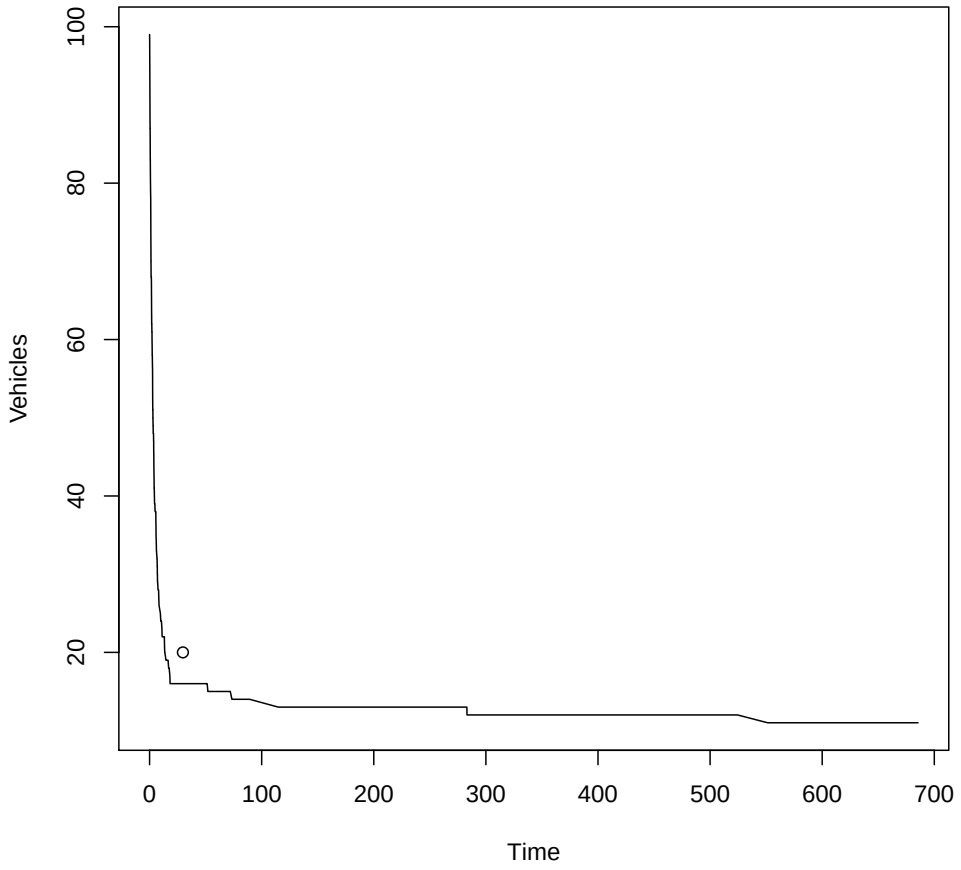

Figure 5.4: Evolution of the number of vehicles starting directly with LNS. The point represents the initial solution obtained by the savings method.

By comparing those methods we want to find out which impact has fixing all the non-removed customers with respect to leaving them open. Secondly, we want to see how important is the problem of not allowing further insertions before reinserted nodes, and whether the Costumers Left Free method—which solves this problem, performs better.

Table 5.3: Summary of the three reinsertion methods classified by whether removed customers are fixed upon reinsertion and by whether the rest of the non-removed nodes are fixed.

		Fix removed clients upon reinsertion?	
		YES	NO
Non-removed clients fixed?	YES	Fixed Method (Fix)	-
	NO	Partially Fixed Method (PF)	Clients Left Free Method (CLF)

Within the two methods fixing the costumers when they are inserted (see section 4.3.2.3), the one that also fixes the non-removed costumers performs better as can bee seen in Table 5.4. Not fixing the non-removed costumers provides a larger neighbourhood to explore. Although a larger neighbourhood allows the engine to explore more options to find solutions, it also takes more time to be completely explored, because more reinsertions and backtracks have to be done, each of them involving a clone operation (see section 4.3.4). The Customers Left Free method does not fix anything, allowing free customers to be

Table 5.4: Comparative of the results obtained with the three reinsertion methods: Clients Left Free (CLF), Only fixing the reinserted costumers (Partially Fixed, PF) and Everything Fixed (fix).

Instance	Veh. (CLF)	Dist (CLF)	Veh. (PF)	Dist (PF)	Veh. (Fix)	Dist (Fix)
r101	20	1695.79	22	1764.76	22	1764.25
r102	19	1529.07	20	1608.47	20	1542.66
r103	14	1265.32	16	1358.48	17	1346.82
r104	11	1062.91	13	1166.12	14	1126.54
r105	16	1403.68	18	1547.00	19	1506.38
r106	13	1289.56	16	1390.05	17	1370.58
r107	12	1116.89	14	1205.06	14	1180.66
r108	11	994.33	12	1063.71	13	1072.42
r109	14	1233.84	16	1350.24	16	1271.52
r110	13	1146.32	15	1303.54	15	1245.88
r111	13	1104.25	15	1286.18	15	1235.94
r112	11	989.92	15	1237.87	14	1098.97
rc101	17	1711.34	19	1881.81	19	1901.39
rc102	15	1547.61	19	1929.07	18	1690.82
rc103	13	1354.11	17	1624.49	15	1473.92
rc104	11	1194.70	14	1568.24	14	1401.39
rc105	16	1598.80	20	1966.50	18	1738.78
rc106	14	1468.65	17	1751.27	17	1569.44
rc107	13	1303.16	15	1488.59	16	1504.73
rc108	12	1222.51	16	1533.95	15	1398.83
Sum	278	26232.76	329	30025.40	328	28441.92

reinserted anywhere in the current solution. This provides a much bigger neighbourhood for the engine as the nodes can be inserted in a much bigger set of positions, and although it also becomes a bit slower because of the increase of size in the neighbourhoods, much better solutions are found, as can be seen in Table 5.4. As seen in the table, the results obtained by this method are clearly better for all instances, both in vehicles and distance.

The fact that the Customers Left Free method performs so much better demonstrates how big a problem fixing the customers upon insertion is: It greatly reduces the neighbourhood causing the engine to miss a lot of choices that could potentially lead to better solutions. The better performance of the Partially Fixed Method with respect to the Fixed Method indicates that a big neighbourhood to explore is useless if the neighbourhood isn't properly chosen. This method does provide a bigger search tree, but the big problem it has makes it useless, because the values we want to find are most likely left out because of this problem. The only thing it accomplishes is wasting more time exploring a bigger search tree that won't lead to any better solution.

5.3.2 Heuristics for Insertion Sequence

Another decision that can be important when reinserting clients is the choice of which client to reinsert first. Two options have been presented (see section 4.3.2.1). Tests have been run to determine which of them yields the best results. Although the Farthest Insertion Heuristic performs better on the instance r103, the Most Constrained Heuristic does give better results in almost all the RC-Type instances—concretely for the rc101, rc105, rc106, rc107 and rc108 (see Table 5.5). The better performance of the Most Constrained Heuristic can come from the fact that a lot of searches that end up with a failed spaced are avoided.

Inserting first the variables with less possible values attempts to minimize the risk of the domain of a variable becoming empty. Avoiding the search of those neighbourhoods that will end up in a fail is important because it increases the speed of the reinsertion process and can take the engine to bigger neighbourhoods within the placed time limit. These results indicate that it is important to try and minimize distance travelled, but it is also important to guide search but taking the current domains into the variables into account.

Table 5.5: Results for the Farthest Insertion Heuristic (FI) and the Most Constrained Heuristic (MC)

Instance	Vehicles (FI)	Distance (FI)	Vehicles (MC)	Distance (MC)
r101	20	1695.79	20	1680.45
r102	19	1529.07	19	1520.05
r103	14	1265.32	15	1243.72
r104	11	1062.91	11	1058.80
r105	16	1403.68	16	1400.01
r106	13	1289.56	13	1305.13
r107	12	1116.89	12	1125.59
r108	11	994.33	11	976.78
r109	14	1233.84	14	1241.51
r110	13	1146.32	14	1241.51
r111	13	1104.25	12	1093.04
r112	11	989.92	11	988.96
rc101	17	1711.34	16	1706.99
rc102	15	1547.61	15	1534.57
rc103	13	1354.11	13	1353.73
rc104	11	1194.70	11	1183.79
rc105	16	1598.80	15	1619.02
rc106	14	1468.65	13	1469.65
rc107	13	1303.16	12	1274.19
rc108	12	1222.51	11	1163.03
Sum	278	26232.76	274	26180.52

The fact that the Most Constrained heuristic works better than the Farthest Insertion heuristic comes from the fact that it avoids failing of spaces sooner. However, it is interesting to compare these results with those obtained when variables with only one valid insertion point left are immediately inserted (see Table 5.6).

5.3.3 Inserting Variables when Only One Position is Left

We want to see if inserting variables immediately if at some point they are left with one possible value 4.3.2.2 has any influence on both the solution quality and computation time. It can be seen in Table 5.6 inserting variables when only one position is left results in better vehicle optimization for some instances, specifically r102, r109, r111, rc107 and rc108. It also performs better on distance for most of the instances. Inserting variables when only one value is left in their domain avoids many useless explorations that will end in failed spaces, greatly accelerating the search and allowing the engine to explore bigger neighbourhoods in less time. This is reflected in the neighbourhood size reached within the 900 time limit, which is greater in all instances. This is especially important for instances with wide time windows. These instances are much less constrained and thus

reinsertion takes longer, as many possibilities are available and must be explored. Due to this extra cost on reinsertion, for some of the instances the engine only reaches a very small neighbourhood size—for example for the instance r104, it only reaches neighbourhood size of 8 removed customers. On the other hand, by inserting the customers when they have only a single allowed value brings the engine to reach neighbourhood size of 14 removed customers. Getting to remove bigger sets of customers is important because it allows changing bigger parts of the solution, which is important to escape local minima (see section 2.4).

Table 5.6: Results on distance for the plain Farthest Insertion heuristic (FI) and the Farthest Insertion heuristic with immediate insertion of variables with only one position left (FI-INS) , as well as neighbourhood size reached.

Instance	Veh. (FI)	Dist (FI)	Veh. (FI-INS)	Dist (FI-INS)	NSize (FI)	NSize (FI-INS)
r101	20	1695.79	20	1682.16	30	30
r102	19	1529.07	18	1556.08	30	30
r103	14	1265.32	14	1285.53	24	30
r104	11	1062.91	11	1071.13	11	13
r105	16	1403.68	16	1403.62	30	30
r106	13	1289.56	13	1302.59	22	30
r107	12	1116.89	12	1147.18	18	28
r108	11	994.33	11	990.75	11	17
r109	14	1233.84	13	1199.86	23	29
r110	13	1146.32	13	1162.69	19	22
r111	13	1104.25	12	1099.56	24	26
r112	11	989.92	11	994.25	10	16
rc101	17	1711.34	17	1708.74	30	30
rc102	15	1547.61	15	1551.35	30	30
rc103	13	1354.11	13	1345.78	21	19
rc104	11	1194.70	11	1166.51	8	14
rc105	16	1598.80	16	1591.16	30	30
rc106	14	1468.65	14	1496.01	24	29
rc107	13	1303.16	12	1295.62	18	26
rc108	12	1222.51	11	1198.44	12	16
Sum	278	26232.76	273	26249.01	425	495

It is also interesting to compare these results obtained with the Farthest Insertion Heuristic against the ones obtained with the Most Constrained heuristic (see Table 5.5). With this optimization, the Farthest Insertion heuristic gives distances slightly higher, but performs better in number of vehicles on 20% of the instances (r102, r109, r111, rc107, rc108), so inserting variables immediately when only one insertion point is left indeed succeeds in bringing some of the advantage of the most Constrained Heuristic into the Farthest Insertion heuristic.

5.3.4 Time Limit on Reinsertion

As solutions are usually found relatively quick within a neighbourhood, we want to see if placing a 20 second time limit on the reinsertion step 4.3.2.5 accelerates the whole search, allowing the engine to reach bigger a neighbourhood size faster. We also want to see if cutting the search at this point has any significant effect on the solution quality because it will prevent the engine from finding a better solution, or on the contrary it is safe to assume that from there on no better solutions will be found in the search. We can see

that the results obtained are better for many instances in Table 5.7, namely r102, r109, r111, rc103, rc107 and rc108. Only for the instance rc106 the number of vehicles goes up. The reason for it performing better for many instances is that due to the reinsertion process being accelerated, we get to remove bigger sets of clients—that is, we get to explore bigger neighbourhoods. Removing big sets of customers is a requirement regarding vehicle optimization, as this allows the engine to perform more powerful moves, which favours escaping from local minima.

Table 5.7: Comparative of the results obtained by placing a time limit on the reinsertion step against the ones obtained by not doing so.

Instance	Veh.	Dist	Veh. (Inslimit)	Dist (Inslimit)	NSize	NSize (Inslimit)
r101	20	1695.79	20	1682.16	30	30
r102	19	1529.07	18	1556.08	30	30
r103	14	1265.32	14	1285.53	24	26
r104	11	1062.91	11	1067.69	11	19
r105	16	1403.68	16	1403.62	30	30
r106	13	1289.56	13	1302.59	22	30
r107	12	1116.89	12	1124.46	18	26
r108	11	994.33	11	990.75	11	14
r109	14	1233.84	13	1199.86	23	30
r110	13	1146.32	13	1162.69	19	24
r111	13	1104.25	12	1108.79	24	23
r112	11	989.92	11	994.25	10	14
rc101	17	1711.34	17	1708.74	30	30
rc102	15	1547.61	15	1547.20	30	27
rc103	13	1354.11	12	1357.21	21	27
rc104	11	1194.70	11	1223.04	8	13
rc105	16	1598.80	16	1615.90	30	30
rc106	14	1468.65	15	1500.46	24	29
rc107	13	1303.16	12	1311.02	18	23
rc108	12	1222.51	11	1199.34	12	16
Sum	278	26232.76	273	26341.38	425	491

5.3.5 Stopping Search When a Solution is Found

We want to test when should the reinsertion step be stopped. Preliminary tests have shown that exploring the whole neighbourhood even when a solution that is better than the current best found solution is not worth it, as it takes a lot of time to explore neighbourhoods for which already a better solution has been found. Based on 4.3.2.6 we now want to test if it is safe to assume the first solution found will be the best, stopping search in the current neighbourhood even if this first found solution is not as good as our current best solution, or if contrarily we should keep searching for better solutions.

The results in Table 5.8 show that it is better to keep searching for a solution that is better than the current best. Doing so yields better results for some of the instances (r102, r109, r110, r111, rc107).

To understand what is happening we have to look at the evolution of the number of vehicles through time. As can be seen in the Figure 5.5, at the beginning both strategies present almost the same results, going exponentially down. This is because the initial solution with one vehicle per tour is of very poor quality, and as a almost any solutions

Table 5.8: Results for the both strategies, stopping when a solution is found (FF), or stopping when a solution better than the current one is found (FB).

Instance	Distance_FF	Vehicles_FF	Distance_FB	Vehicles_FB
r101	1695.79	20	1703.19	20
r102	1529.07	19	1556.08	18
r103	1265.32	14	1285.53	14
r104	1062.91	11	1067.69	11
r105	1403.68	16	1403.62	16
r106	1289.56	13	1302.59	13
r107	1116.89	12	1124.46	12
r108	994.33	11	990.75	11
r109	1233.84	14	1199.86	13
r110	1146.32	13	1164.06	13
r111	1104.25	13	1099.56	12
r112	989.92	11	994.25	11
rc101	1711.34	17	1708.74	17
rc102	1547.61	15	1613.88	15
rc103	1354.11	13	1345.78	13
rc104	1194.70	11	1166.51	11
rc105	1598.80	16	1612.75	16
rc106	1468.65	14	1496.01	14
rc107	1303.16	13	1295.62	12
rc108	1222.51	12	1198.44	11
Sum	26232.76	278	26329.37	273

that are found will also be better than the current one. Note that if the first solution found within a neighbourhood is better than the current one, then both methods behave exactly the same. However, as the solution gets better, its more difficult to found a better solution. The assumption that the best solution will always be the first found—or at least most times, is incorrect, as evidenced by the fact that although both methods do similarly through time, by searching for a solution that improves the current best the engine finds one last solution at the end that the other strategy fails to find. This means that there are better solutions hidden deeper in the search tree that will be found only if it is explored. Although exploring deeper into the tree means using more discrepancies or the removed customers and consequently inserting more nodes in locally less optimal positions, it can happen that the overall solution becomes better. It is worth being explored, as otherwise this new improvement in the solution is skipped by the engine.

5.4 Relatedness Function

In this section we want test the various ways of choosing the clients to remove. First of all we want to evaluate if the concept of using relatedness between customers is indeed better than choosing clients to be removed at random. Table 5.9 shows that random selection of customers performs poorly in comparison with the relatedness functions, both in number of vehicles and distance. This confirms that indeed using the information we have available about the problem and current state of the solution by removing related clients is important.

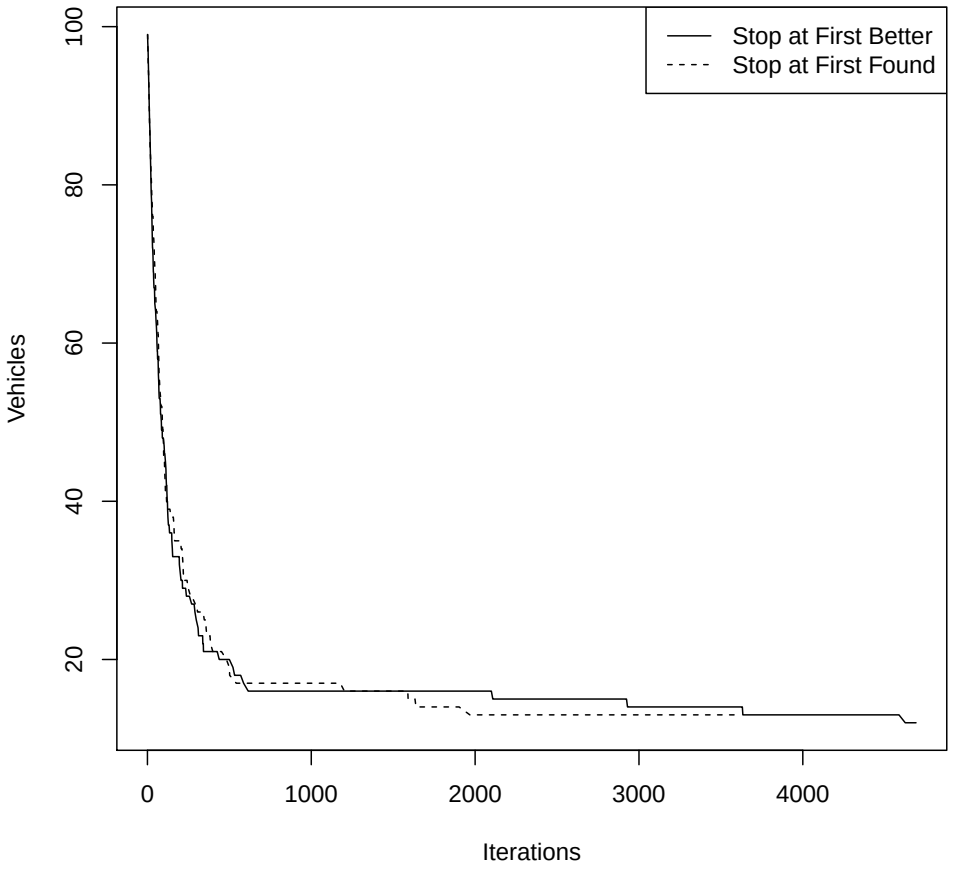

Figure 5.5: Evolution of the number of vehicles using the both strategies.

Secondly, we want to compare the suggested relatedness function 4.3.1.2 to a simple function that only considers geographic closeness. Our relatedness functions perform very similarly regarding distance, but there are some differences regarding number of vehicles 5.9. While the only geographically related function performs slightly better on a few instances (r102, r109, r111, rc108), it also performs slightly worse on some others (r103, r106, rc107), and much worse on some others (rc102, rc103) where the solution obtained is up to 4 vehicles over the solution found with the same tour relatedness function. This indicates that while both relatedness functions seem to perform similarly on some of the instances, there are other instances where it is indeed required to remove entire tours to successfully optimize the number of vehicles. Those are the instances where using our relatedness function really performs much better than the only geographically related.

5.5 Parameters

In these section we want to test the three main parameters of our search engine: Number of discrepancies, number of attempts and determinism. We first make an analysis of each parameter on its own to understand how the search is affected by each of those parameter variations, and then we have an global overview of the results obtained with various combinations of the parameters.

Table 5.9: Table showing a comparative of the results obtained removing a set of clients at random (RDM), using a geographically-close relatedness mesure (GEO) and using a geographic and same tour relatedness function (Tour).

Instance	Veh. (RDM)	Dist (RDM)	Veh. (GEO)	Dist (GEO)	Veh. (TOUR)	Dist (TOUR)
r101	20	1688.52	20	1659.69	20	1695.79
r102	20	1568.67	18	1546.79	19	1529.07
r103	16	1312.34	15	1264.12	14	1265.32
r104	13	1086.99	11	1070.10	11	1062.91
r105	18	1470.34	16	1410.20	16	1403.68
r106	16	1360.65	14	1276.95	13	1289.56
r107	13	1149.53	12	1128.71	12	1116.89
r108	15	1175.51	11	983.12	11	994.33
r109	15	1287.92	13	1236.46	14	1233.84
r110	13	1158.45	13	1139.38	13	1146.32
r111	13	1261.94	12	1189.23	13	1104.25
r112	13	1092.31	11	998.22	11	989.92
rc101	19	1765.28	17	1690.12	17	1711.34
rc102	17	1637.01	17	1690.12	15	1547.61
rc103	13	1400.84	17	1690.12	13	1354.11
rc104	13	1335.27	11	1192.47	11	1194.70
rc105	16	1594.54	16	1579.12	16	1598.80
rc106	16	1540.73	14	1448.55	14	1468.65
rc107	14	1422.15	14	1448.55	13	1303.16
rc108	13	1342.75	11	1176.85	12	1222.51
Sum	306	27651.74	283	26818.87	278	26232.76

5.5.1 Discrepancies

Discrepancies indicate how stubbornly the engine will try to find a solution within the neighbourhood of a removed solution. A number of discrepancies that is too low will cause the engine to discard neighbourhoods very soon when a solution might still be there to find, while a number of discrepancies that is too high will cause the engine to keep exploring the search tree for solutions where none is to be found, trying values for the removed customers that are far from optimal. This is the most important parameter of the three and it has a strong influence on how the engine performs, as it determines how the neighbourhoods are explored. Adjusting the number of discrepancies is thus very important. We have run tests for values of 2, 3, 5, 10, 15 and 20 discrepancies. To perform those tests, we have fixed the other parameters at $D = 15$ and $a = 250$. The obtained results are shown in Table 5.10. The best results are obtained with $d = 10$ and $d = 15$. While $d = 10$ has slightly worse results in number of vehicles, $d = 15$ presents higher travelled distance values. Results slightly worse in both distance and number of vehicles are obtained with $d = 3$ and $d = 5$, which perform very similarly compared to each other. $d = 2$ is clearly an insufficient number of discrepancies, as the results obtained are significantly worse, and although we can almost always explore until neighbourhoods of 30 removed costumers as the reinsertion is very cheap, we don't explore deep enough in each of them to find good solutions. On the contrary, for $d = 20$ the results go worse again with respect to $d = 10$ or $d = 15$. The number of discrepancies is too high and exploration of a neighbourhood takes a very long time. For this reason, we don't reach big neighbourhoods—for some instances with wide time windows, the engine only reaches neighbourhood size of five or six, which is clearly insufficient.

Table 5.10: Average number of vehicles and travelled distance obtained for values of 2, 3, 5 10, 15 and 20 discrepancies, with a=250 and D=15.

	d=2	d=3	d=5	d=10	d=15	d=20
Average number of vehicles	14.70	13.70	13.90	13.40	**13.30**	13.60
Average travelled distance	1355.76	1316.24	1311.64	**1310.78**	1323.02	1325.72

5.5.2 Attempts

The number of attempts determines how many consecutive reinsertion tries we will allow before increasing the neighbourhood size. A value that is too low will cause the engine to jump too soon to the next neighbourhood size when solutions could still potentially be found. On the contrary, a number too high will cause the engine to spend so much time attempting to find solutions with a neighbourhood size when there are probably no more to be found. Tests have been run for 250, 500 and 1000 attempts. Table 5.11 shows that $a = 1000$ is too large a number of attempts. In this case reinsertion takes much longer and thus very little exploration is large neighbourhoods. On the other hand, $a = 500$ performs slightly better than $a = 250$, showing that 250 is a too small number of attempts. It seems to be the parameter that has a lower effect on the search.

Table 5.11: Average number of vehicles and travelled distance obtained for 250, 500 and 1000 attempts, with d=5 and D=15.

	a=250	a=500	a=1000
Average number of vehicles	13.90	**13.60**	13.75
Average travelled distance	1311.64	1304.92	**1303.67**

5.5.3 Determinism

Determinism will indicate how strongly we rely on the relatedness function when removing a set of clients. Values of determinism of 5, 10 and 15 have been tested. As can be seen in Table 5.12, $D = 10$ and $D = 15$ perform almost the same, both producing the best results in number of vehicles. In terms of distance, they also perform almost exactly the same but $D = 10$ is slightly better. The results for $D = 5$, show that a high randomness parameter is even worse. With $d = 5$, we are too close to removing sets of costumers at random, which is coherent and reinforces the results obtained in section 5.4, which indicate that the guidance of the relatedness function is indeed important and needed to appropriately guide the search. Determinism has however a low impact on the solution quality. As can be seen in the table the results vary only minimally. Of the three parameters, it is the one which has the less effect on the results. An explanation of why that is so is that, while discrepancies and attempts affect directly the search process, determinism affects only the relatedness function, varying it slightly.

5.5.4 Overall

It seems that, when tested individually, $d = 15$, $a = 500$ and $D = 15$ is the best setup. It is interesting however to test if those values do actually produce the best results when combined. Running tests with that exact configuration has shown that it is not the case (see Table 5.13). The reason is that $d = 15$ enforces that search is done extensively within a

Table 5.12: Average number of vehicles and travelled distance obtained for values of determinism of 5, 10 and 15, with d=5 and a=250.

	D=5	D=10	D=15
Average number of vehicles	14.00	**13.90**	13.90
Average distance travelled	1308.61	**1311.14**	1311.64

given neighbourhood, while $a = 500$ enforces that many attempts are done before increasing the neighbourhood size. As discrepancies were tested with a low value of attempts, we could afford using a high number of discrepancies. Similarly, attempts were tested with a low number of discrepancies, allowing us to afford a relatively high number of attempts. However, when a high number of discrepancies and a high number of attempts are combined, these configurations slow up the whole reinsertion process significantly, resulting in the engine not getting to explore neighbourhoods big enough—to around five or six removed customers, which is of course insufficient. In addition, while for configurations that are less search-extensive we can see that the last solution the engine finds is found relatively soon, with this configuration the engine keeps finding solutions up to the very time limit that is placed on the search. This is again caused by the fact that the engine can only get to remove small sets of customers and has not yet got to explore the bigger neighbourhoods. The fact that new solutions are found towards the end of the test hint that there are still more solutions to be found and that, with more time, they would end up finding a better solution. This hypothesis is explored in section 5.9. However, in this section we want to determine which are the best parameters for our current time limit of 900 seconds.

To find out the best configuration we want to test various combinations of values. We have taken the best values for discrepancies $d = 5, 10, 15$, the number of attempts $a = 250, 500, 1000$. While discrepancies and number of attempts are strongly related as they affect the reinsertion step directly, determinism only affects the relatedness function and seems to have a small impact on the solution, as $D = 10$ and $D = 15$ perform so similarly. He have assumed then that $D = 10$ is the best value for determinism independently of the number of discrepancies and attemtps. We have run tests for the chosen values of discrepancies and attempts with determinism fixed at $D = 10$. Table 5.13 shows the results obtained with $D = 10$. The best results are produced with $d = 10$, with both $a = 250$ and $a = 500$ producing the best value for number of vehicles, and $a = 500$ performing the best result in terms of distance. Both configurations of $d = 10, a = 1000$ and $d = 15, a = 250$ perform almost as good as $d = 10, a = 500$, only slightly worse. For bigger values of a with $d = 15$, exploration is increased too much and the results we obtain begin worse as the number of attempts increases. Similarly, for $d = 5$ too little exploration is done, also resulting in worse values. But here, the results get worse as we decrease the number of attempts, as this reduces even more the exploration done.

If we compare the results obtained with the combination $d = 10, a = 500, D = 10$ with the ones that we obtained with $d = 15, a = 250, D = 15$ from the former tests (see Table 5.10), we can see that they are slightly worse. This fact hints that, although $D = 10$ performs slightly better in the concrete case of $d = 5, a = 250$, assuming that it is always the best choice is not right and that $D = 15$ is actually a better value. We have thus repeated the tests for the chosen values of discrepancies and attempts, now with determinism fixed at $D = 15$. In the results presented in 5.14 we can see that the results obtained are worse in some cases. Those are cases with a low number of discrepancies or a high number of attempts. However, with those configurations that also obtained the best values for

Table 5.13: Results obtained for distance (right) and number of vehicles (left), with various configurations for discrepancies and attempts, with the fixed value for determinism of $D = 10$.

	a=250	a=500	a=1000
d=5	1311.14 13.9	1304.31 13.65	1294.6 13.55
d=10	**1306.76 13.4**	**1293.64 13.4**	1301.72 13.45
d=15	1318.66 13.45	1308.62 13.6	1314.10 13.7

$D = 10$, the results improve with respect to the tests previously made with $D = 10$. The best results are obtained with $d = 10$, $a = 500$, having the best score in number of vehicles. The configuration $d = 15, a = 250$ performs the same in number of vehicles, but is slightly worse in distance.

Table 5.14: Results obtained for distance (right) and number of vehicles (left), with various configurations for discrepancies and attempts, with the fixed value for determinism of $D = 15$.

	a=250	a=500	a=100
d=5	1311.64 13.9	1304.92 13.6	1303.69 13.75
d=10	1310.79 13.4	**1313.39 13.3**	1301.87 13.65
d=15	**1323.03 13.3**	1305.54 13.65	1316.02 13.75

Based on this analysis, we conclude that the best parameter combination with the current time limit of 900 seconds is $d = 10, a = 500$ and $D = 10$.

5.6 Objective Function

The objective of this section is to see whether our objective function 4.3.3 really allows the travelled distance value to become slightly worse if that implies an improvement in the number of vehicles. A penalty of 1000 has been used for each vehicle used, as suggested in [Sha98]. It is a high penalty in the sense that it means we allow the distance to get up to 1000 units worse if this means optimizing a vehicle. This, despite the increase in distance, still causes the overall objective function to go down. As an example the evolution of the values for distance and number of vehicles for the instance r106 are presented. We can clearly observe in Figure 5.6, that near iteration 5000, the travelled distance goes a bit up exactly at the point where the number of used vehicles is improved. At this point, our objective function serves its purpose and accepts a solution with a higher distance, prioritizing the optimization of the number of vehicles.

5.7 Best Found Configurations

After testing all the combinations and options presented, we want to see if all the configurations that were optimal locally also produce the best results when combined. As a summary, these are exactly the options that have performed best in each case: An initial solution with the savings method. For the relaxation step, the geographic and same tour relatedness measure has been taken. For the reinsertion, the Customers Left Free method with the Farthest Insertion Heuristic, with the optimization of inserting variables immediately when they only have one insertion point. A 20 second time limit has been placed

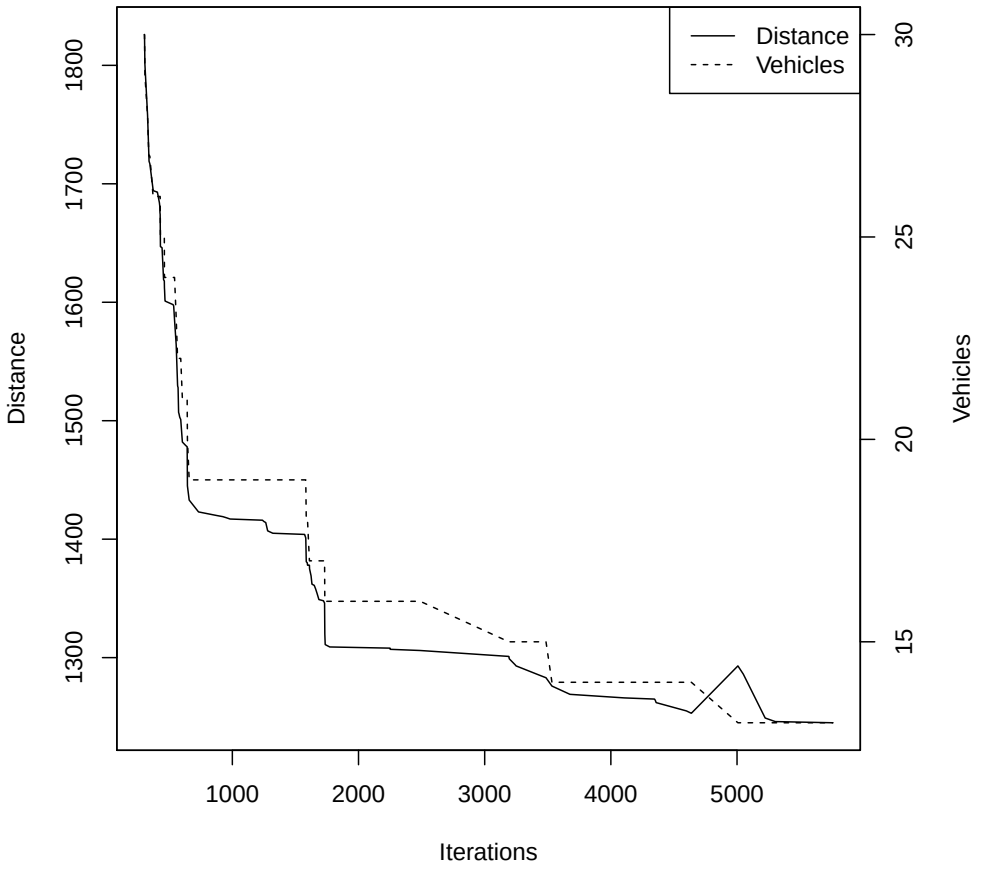

Figure 5.6: Evolution of the travelled distance and number of vehicles over iterations for the r106 instance.

on the reinsertion step. The search is stopped when a solution that is better than the current best is found. The parameter values have been set at $d = 10, a = 500$ and $D = 15$. The combination of those methods should result in better results than the ones obtained until here, or in the worst case, results that are equally good. After running a complete test with this configuration, we see that it doesn't provide the best results. That would indicate that one or more of the tested options perform good when tested individually but don't work well with the others.

Time Limit on Reinsertion. Tests have been run without the insertion time limit for the configuration described above. Preliminary tests show that this option was indeed the one that caused the whole configuration to perform worse, as the engine performs now very equally than the previously tested methods. The reason that time limit on reinsertion does not work properly now is that we now use parameters of $d = 10, a = 500$, while the insertion time limit had been tested with lower values of $d = 5, a = 250$. More discrepancies indicate that more search will be done within each neighbourhood. This increase in the search space also implies an increase on the time each reinsertion step takes. Placing this time limit on the reinsertion step with a higher number of discrepancies prevents the engine from exploring the whole neighbourhood that results from that number of discrepancies. Thus, the insertion time limit should not be used when allowing a higher number of discrepancies, or in any case, a higher time limit should be placed.

Savings method for the Initial Solution. As the savings method seem to perform very similar to starting directly with one vehicle per customer, the results differencing only in very few instances, we have also run tests without using it. The results indicate that it is indeed, with this configuration, better to start directly with one vehicle per customer instead of using the savings method.

Table 5.15 shows the results obtained by this configuration but starting with one vehicle per customer and not placing the time limit on reinsertion. We can see the results are now very similar to the ones obtained in the previous analysis, with some instances even performing better (r102).

Table 5.15: Results obtained with the best configuration for the engine. We also report the neighbourhood size reached (NSize Reached) before the time limit of 900 seconds.

Instance	Vehicles	Distance	NSize Reached
r101	19	1691.23	13
r102	17	1527.86	8
r103	14	1285.47	7
r104	11	1071.80	7
r105	15	1447.05	14
r106	13	1286.64	9
r107	12	1128.47	7
r108	11	1003.51	6
r109	13	1207.69	11
r110	12	1145.83	9
r111	12	1095.10	7
r112	11	1022.09	5
rc101	16	1678.98	9
rc102	14	1548.73	10
rc103	12	1339.68	8
rc104	11	1200.77	5
rc105	15	1591.66	12
rc106	14	1427.90	8
rc107	12	1305.01	6
rc108	12	1204.81	6

From this analysis we conclude that the best configuration for the engine is the following:

- **The initial solution** will constructed assigning one vehicle to each customer.
- **Relaxation** will make use of the relatedness function that considers geographic relation as well as whether the customers are currently on the same tour.
- **Reinsertion** will be done with the Customers Left Free method.
- As **insertion order heuristic** we will take the Farthest Insertion heuristic, with the optimization of inserting a variable immediately when it only has one insertion point left.
- **No time limit** will be placed on the reinsertion step.
- We **stop the search** after we found a solution that is **better than the best found** solution so far.

- **Parameter Configuration**. We take the best found values of bf $d = 10, a = 500$ and $D = 15$.

5.8 Robustness of the Solution

Finally we want to analyse how robust is the solution. All the tests have been run ten times with different random seeds. We want to see how big a part this randomness factor has, and see if the other solutions found differ a lot from the best found one. We have computed the standard deviation for the 10 tests corresponding to the results in Table 5.15. The results for the standard deviation are shown in Table 5.16. The results show that the solution is robust in number of vehicles, for which the standard deviation is less than one for all instances. In terms of distance, it is also robust in most of the instances, although some of them present a relatively high deviation of around 20% of their distance value.

Table 5.16: Average solution quality, in number of vehicles and distance, found by the ten runs of our engine, as well as the standard deviation.

Instance	Avg. Vehicles	Avg. Distance	StDev (Vehicles)	StDev (Distance)
r101	19.90	1684.73	0.32	16.31
r102	18.70	1531.56	0.67	7.21
r103	18.70	1531.56	0.67	13.54
r104	12.10	1083.21	0.57	30.69
r105	16.00	1423.47	0.47	23.55
r106	14.10	1299.31	0.57	21.08
r107	12.90	1145.60	0.32	22.08
r108	11.10	1022.27	0.32	16.39
r109	13.90	1231.39	0.57	26.05
r110	13.10	1166.16	0.57	14.24
r111	12.90	1130.88	0.74	17.68
r112	11.90	1046.14	0.57	25.02
rc101	16.50	1695.98	0.53	15.69
rc102	14.90	1557.89	0.32	23.08
rc103	12.90	1367.07	0.57	15.25
rc104	11.80	1250.99	0.42	25.74
rc105	16.00	1609.37	0.47	17.50
rc106	14.40	1477.63	0.70	30.02
rc107	12.60	1327.21	0.52	23.96
rc108	12.50	1276.23	0.53	48.21

5.9 Improvement over Time

We want to see how the solution evolves with more computation time. Especially, since we have a relatively high number of discrepancies and attempts ($d = 10, a = 500$) and this causes the engine not to reach very big neighbourhoods, as evidenced in Table 5.15. We want to see if giving the engine more time to explore those neighbourhoods results in better solutions for the problems.

We have run test with a half an hour limit (1800 seconds) and one hour limits (3600 seconds) to see if the solution really improves with more computation time. The results obtained for a half an hour time limit are shown in Table 5.17. While distance is improved

for almost all instances, there isn't a big influence in the number of vehicles used when giving the engine more computation time. By giving the engine half an hour, we improve a vehicle for instances r109 and rc108, and by giving it a whole hour we further improve a vehicle for instance r107. What can be observed is that the solution is more robust, meaning that from the 10 executions with different random seeds, most of them find that best solution but fail to further improve it.

Table 5.17: Results obtained with the 900 seconds time limit compared to the results obtained with 1800 seconds and 3600 seconds.

Instance	Vehicles (900)	Dist (900)	Vehicles (1800)	Dist (1800)	Vehicles (3600)	Dist (3600)
r101	19	1691.23	19	1691.23	19	1691.23
r102	17	1527.86	17	1518.67	17	1518.67
r103	14	1285.47	14	1268.52	14	1263.02
r104	11	1071.80	11	1031.55	11	1025.78
r105	15	1447.05	15	1447.05	15	1447.05
r106	13	1286.64	13	1322.07	13	1279.06
r107	12	1128.47	12	1111.10	11	1121.26
r108	11	1003.51	11	1002.58	11	988.68
r109	13	1207.69	12	1215.43	12	1215.43
r110	12	1145.83	12	1133.24	12	1132.59
r111	12	1095.10	12	1083.52	12	1083.52
r112	11	1022.09	11	1004.04	11	999.91
rc101	16	1678.98	16	1678.98	16	1678.98
rc102	14	1548.73	14	1545.67	14	1538.11
rc103	12	1339.68	12	1339.68	12	1331.30
rc104	11	1200.77	11	1184.38	11	1175.39
rc105	15	1591.66	15	1591.66	15	1591.66
rc106	14	1427.90	13	1432.01	13	1432.01
rc107	12	1305.01	12	1249.82	12	1249.82
rc108	12	1204.81	11	1175.42	11	1161.32

6. Conclusion

To conclude, a test for our methods with the combination of each best performing part is compared to the results exposed in [Sha98]. We have used the best configuration found in section 5.7. As an initial solution, we use the one obtained by initially assigning one vehicle per customer. For the relaxation step, we use the relatedness function that considers both geographic closeness and whether or not the customers are in the same tour. As for reinsertion, we use the Customers Left Free method combined with the Farthest Insertion heuristic—but inserting variables immediately if only one value is left in their domain. No time limit is placed on the reinsertion step, but we will stop it once a solution that is better than the current best solution is found. The parameters have been set at $d = 10, a = 500$ and $D = 15$.

As can be observed in table Table 6.1, while the results obtained in distance travelled are very similar, in most instances we are one or two vehicles over the solution the authors reported. Concretely, for all the C-Instances the optimal number of vehicles has been found. For the other instances, we have found the optimal solution for 10% of the instances (r101, r102). In 40% of the instances, we are 1 vehicle above and in the remaining 50% of the instances the results obtained use 2 vehicles more than the ones reported by the authors. In terms of distance, the results obtained are almost optimal, being 1.93% above for the C-Type instances and 3.39% above for the rest of the instances. It should be noted that in some cases where we obtain better results in distance this can come from the fact that our computation is done with integer variables, which could lead to some solution being accepted when it wouldn't be if computing with the decimal numbers.

The fact that the solutions obtained are not exactly optimal can come from the fact that the authors of [Sha98] do not describe exactly neither how the neighbourhoods are chosen and not many details about the implementation are given, which can have led to some misinterpretation from our part. Parameter analysis however provides a bit of insight regarding where the problem might be. Especially interesting is the observation that we obtain the best results for $d = 10, 15$ discrepancies while the authors obtain them for $d = 5, 10$. This means that we need more discrepancies to find a solution which implies that we need to try more insertion points to find it. This in turn means that the insertion points we calculate for each customer to insert are not the same as the authors'. Probably the problem lies within deciding which insertion points are candidates for reinsertion, and in which order those insertion points will be tried.

Although we were not able to perfectly reproduce the results of [Sha98], we could verify that Constraint Programming is indeed an effective way of solving Vehicle Routing Problems,

Table 6.1: Comparison of our best found solutions against the best found solutions the authors reported.

Instance	Vehicles_Shaw	Distance_Shaw	Vehicles	Distance
c101	10	828.94	10	828.94
c102	10	828.94	10	829.70
c103	10	828.06	10	882.13
c104	10	824.78	10	836.06
c105	10	828.94	10	828.94
c106	10	828.94	10	829.70
c107	10	828.94	10	829.70
c108	10	828.94	10	859.76
c109	10	828.94	10	869.12
r101	19	1650.80	19	1691.23
r102	17	1486.12	17	1527.86
r103	13	1292.68	14	1285.47
r104	9	1007.31	11	1071.80
r105	14	1377.11	15	1447.05
r106	12	1252.03	13	1286.64
r107	10	1104.66	12	1128.47
r108	9	963.99	11	1003.51
r109	11	1197.42	13	1207.69
r110	10	1135.07	12	1145.83
r111	10	1096.73	12	1095.10
r112	10	953.63	11	1022.09
rc101	14	1696.95	16	1678.98
rc102	12	1554.75	14	1548.73
rc103	11	1261.67	12	1339.68
rc104	10	1135.48	11	1200.77
rc105	14	1540.18	15	1591.66
rc106	12	1376.26	14	1427.90
rc107	11	1230.48	12	1305.01
rc108	10	1139.82	12	1204.81

as the solutions obtained are very little over the optimal. The ease with which new side constraints can be added greatly outweighs the somewhat less optimization performance. To add side constraints to the system, only the specification of the requirements the solution has to meet has to be modified. This is done by adding the new constraints to the model. Most constraints can be posted using the implemented propagators Gecode provides, but in the worst case all that must be done is implement a custom propagator. The important thing is that to add new side constraints, nothing from the engine has to be modified—the engine remains exactly the same.

Future work should further consider strategies for the reinsertion step, especially for the definition and choice of the insertion points for the customers, as well as possibly the order in which clients are inserted. Other relatedness functions and other reinsertion methods should not be discarded though, as it could be that better options exist. It should also concentrate on an in-detail evaluation of the impact of real-world constraints on the performance of the search engine.

Bibliography

[Apt03] K.R. Apt. *Principles of constraint programming*. Cambridge University Press, 2003.

[BFS+00] B.D. Backer, V. Furnon, P. Shaw, P. Kilby, and P. Prosser. Solving vehicle routing problems using constraint programming and metaheuristics. *Journal of Heuristics*, 6(4):501–523, 2000.

[BGG+97] P. Badeau, F. Guertin, M. Gendreau, J.Y. Potvin, and E. Taillard. A parallel tabu search heuristic for the vehicle routing problem with time windows. *Transportation Research Part C: Emerging Technologies*, 5(2):109–122, 1997.

[CL97] Y. Caseau and F. Laburthe. Solving small tsps with constraints. In *International Conference on Logic Programming*, pages 316–330, 1997.

[CW64] G. Clarke and JW Wright. Scheduling of vehicles from a central depot to a number of delivery points. *Operations research*, pages 568–581, 1964.

[DS90] G. Dueck and T. Scheuer. Threshold accepting: a general purpose optimization algorithm appearing superior to simulated annealing. *Journal of computational physics*, 90(1):161–175, 1990.

[Due93] G. Dueck. New optimization heuristics. *Journal of computational physics*, 104(1):86–92, 1993.

[EOSF06] Ö. Ergun, J.B. Orlin, and A. Steele-Feldman. Creating very large scale neighborhoods out of smaller ones by compounding moves. *Journal of Heuristics*, 12(1):115–140, 2006.

[FJ81] M.L. Fisher and R. Jaikumar. A generalized assignment heuristic for vehicle routing. *Networks*, 11(2):109–124, 1981.

[FLM02] F. Focacci, A. Lodi, and M. Milano. Embedding relaxations in global constraints for solving tsp and tsptw. *Annals of Mathematics and Artificial Intelligence*, 34(4):291–311, 2002.

[GLS96] M. Gendreau, G. Laporte, and R. Séguin. A tabu search heuristic for the vehicle routing problem with stochastic demands and customers. *Operations Research*, 44(3):469–477, 1996.

[GM74] B.E. Gillett and L.R. Miller. A heuristic algorithm for the vehicle-dispatch problem. *Operations research*, 22(2):340–349, 1974.

[HG95] W.D. Harvey and M.L. Ginsberg. Limited discrepancy search. In *International Joint Conference on Artificial Intelligence*, volume 14, pages 607–615. LAWRENCE ERLBAUM ASSOCIATES LTD, 1995.

[HM91] J.H. Holland and J.H. Miller. Artificial adaptive agents in economic theory. *The American Economic Review*, 81(2):365–370, 1991.

[KPS98] P. Kilby, P. Prosser, and P. Shaw. Apes report apes-01-1998. 1998.

[KPS00] P. Kilby, P. Prosser, and P. Shaw. A comparison of traditional and constraint-based heuristic methods on vehicle routing problems with side constraints. *Constraints*, 5(4):389–414, 2000.

[KS06] P. Kilby and P. Shaw. Vehicle routing. *Foundations of Artificial Intelligence*, 2:801–836, 2006.

[Lap09] Gilbert Laporte. Fifty years of vehicle routing. *Transportation Science*, 43(4):408–416, November 2009.

[MH97] N. Mladenović and P. Hansen. Variable neighborhood search. *Computers & Operations Research*, 24(11):1097–1100, 1997.

[PGPR98] G. Pesant, M. Gendreau, J.Y. Potvin, and J.M. Rousseau. An exact constraint logic programming algorithm for the traveling salesman problem with time windows. *Transportation Science*, 32(1):12–29, 1998.

[PR10] D. Pisinger and S. Ropke. Large neighborhood search. *Handbook of metaheuristics*, pages 399–419, 2010.

[RVBW06] F. Rossi, P. Van Beek, and T. Walsh. *Handbook of constraint programming*, volume 2. Elsevier Science, 2006.

[Sha97] P. Shaw. A new local search algorithm providing high quality solutions to vehicle routing problems. *Department of Computer Sciences, University of Strathclyde, Glasgow, Scottland, Tech. Rep, APES group*, 1997.

[Sha98] Paul Shaw. Using constraint programming and local search methods to solve vehicle routing problems. In Michael Maher and Jean-Francois Puget, editors, *Principles and Practice of Constraint Programming – CP98*, volume 1520 of *Lecture Notes in Computer Science*, pages 417–431. Springer Berlin / Heidelberg, 1998.

[Sha11] P. Shaw. Constraint programming and local search hybrids. *Hybrid Optimization*, pages 271–303, 2011.

[Sol87] M.M. Solomon. Algorithms for the vehicle routing and scheduling problems with time window constraints. *Operations research*, 35(2):254–265, 1987.

[STL11] C. Schulte, G. Tack, and M.Z. Lagerkvist. Modeling and programming with gecode, 2011.