University of Glasgow | School of Computing Science

Using Constraint Programming to solve the Vehicle Routing Problem with Time Windows

W. David Fröhlingsdorf (2079884)

April 23, 2018

ABSTRACT

The Capacitated Vehicle Routing Problem with Time Windows is a very important logistic problem because our economy is becoming increasingly globally connected particularly in the wake and rise of online trading. Nowadays customers can order goods at any time from anywhere in the world and the order must be delivered to the customer within days. This trend puts significant strains on supply chains. Logistic businesses need to keep transportation costs low in order to stay competitive. Businesses use computer models and programs to effectively route their delivery vehicles to customers. Various approaches have been proposed and used to solve this problem. This paper focuses on Constraint Programming models because of their high flexibility and readability. Constraint Programming models can be adapted and changed with relative ease on a day to day basis allowing businesses to react quickly to exceptional circumstances. Furthermore, this paper compares two different models as well as a number of search heuristics. Finally, it contains a brief survey of the Vehicle Routing Problem in a wider context.

1. INTRODUCTION

This section contains an informal introduction to the problem and to Constraint Programming which we used to solve the problem. We use simple and informal examples to convey the key ideas.

1.1 Capacitated Vehicle Routing Problem with Time Windows

Consider the following scenario to get an intuition for the problem: You are the owner of a grocery store. In order to increase your business's competitiveness you launched an online shopping website where your customers can place orders for next day home delivery. Each customer can pick a one hour time window for their delivery to arrive. You hired three delivery drivers and each driver has the same vehicle and abilities. Therefore you encounter the following problem every morning: Which driver should deliver to which customers and in what order such that no vehicle is overloaded, every delivery happens in the correct time window and the overall travelled distance is minimal (to minimise fuel consumption)?

This problem is formally known as the *Capacitated Vehicle Routing Problem with Time Windows*(CVPRTW). Section 2 contains a formal description of the problem and Section 6 presents numerous variants of the *Vehicle Routing Problem* (VRP) in general.

There exists a close relation between the CVRPTW and

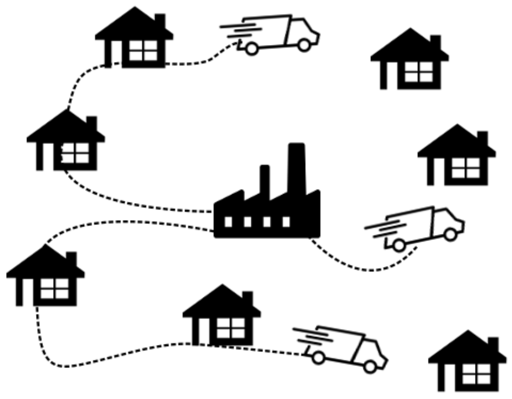

Figure 1: Each vehicle takes a different route

the better known *Travelling Salesman Problem* (TSP). In fact, the CVRPTW is equivalent to the Capacitated Travelling Salesman Problem with Time Windows if we have only **one** vehicle available for the CVRPTW (one vehicle has to visit every customer). Hence, it is clear that the CVRPTW is as least as complex as the TSP. Since finding an optimal solution for the TSP belongs to the *NP-Hard* problem class, finding an optimal solution for the CVRPTW (that is, finding the shortest combined tour for all vehicles) equally belongs to NP-Hard.

1.2 Constraint Programming

Constraint Programming (CP) is a programming paradigm which focuses on the declarative description of a problem and solves the problem automatically in the background[46]. To get a better intuition of CP we will use two independent example problems to explain how problems are modelled and solved using CP.

Imagine you are a stone carver and you are transporting goods from your workshop to the market where you are going to sell your goods. Because the market is many miles away you can not afford to make multiple trips. Your goods are very popular and from your experience you know you will likely sell everything you take to the market. Each of your items has a size and a market price. For example, statues might be very large but also very expensive. On the other hand, smaller sculptures are of medium size and much cheaper. Which items should you load into the van such that your profit will be as large as possible but the van is not overloaded?

Let us consider another example: Imagine you need to create the week rota for the night security service of a museum. Each night you need precisely one guard. A guard must not have two or more consecutive night shifts. Each guard must have two or three shifts per week. Your team consists of three guards. Guard 1 can not work in the first and second night, Guard 2 can not work in the fifth night and Guard 3 can work any night. How do you schedule the shifts of the guards such that all of these constraints are satisfied?

Even though the example of the stone carver (which is a variant of the knapsack problem[35, 10]) and the security guard rota seem to be very different, they have many things in common. First of all, they are both problems which require decisions to be made. In our first example we need to decide for each item whether we are going to take it with us to the market or leave it in the workshop. In the second example we need to decide for each shift which guard is scheduled to work the shift. In CP such decisions are captured in variables which are called *decision variables* (for simplicity they are often referred to as "variables" only). In our first problem we could have a decision variable for each item signalling whether we take the item with us to the market (`true`) or not (`false`). Such a decision variable is also called a boolean decision variable. On the other hand, in the second problem each decision variable can take one of three values; that is, for each shift we need to pick one of the three guards (formally we say that the *domain* of the decision variable consists of three values). Furthermore, we notice that both of our examples have constraints regarding the decision variables. For instance, in the second example the shift of the fifth night can not be assigned to the second guard (because he is not available for this shift).

Let us now look at how one would actually model the two examples in MiniZinc, a popular CP tool kit[37]. MiniZinc translates the source code into a low-level CP code, called FlatZinc, and uses a solver such as Gecode[53] to solve the problem encoded in the model. Figure 2 shows the model of the stone carver example.

```
1   set of int: ITEM = 1..3;
2
3   array[ITEM] of int: size = [1,4,3];
4   array[ITEM] of int: value = [2,3,2];
5   int: max_load = 4;
6
7   var set of ITEM: take;
8
9   constraint sum(i in take)
10                  (size[i]) <= max_load;
11
12  solve maximize sum(i in take)
13                      (value[i]);
```

Figure 2: CP - Stone Carver

Here we assume that we only have three items which we might take to the market. In Line 1 we define a set of three integer values (one to three) where each value represents a specific item, in other words we label each item with a unique number. In Line 3 and 4 we declare for each item its size and market value. The *parameters* `size` and `value` are immutable (read-only) arrays[1] which take an integer of the range `ITEM` as the index and map each integer of this range to an integer denoting the size and the value respectively of the corresponding item. For example, `size[2]` corresponds to the size of the item which is labelled with "2" (in our example: `size[2] = 4`). The parameter `max_load` in Line 5 defines the maximum load the van can transport. Line 7 declares our decision variable `take`. Rather than having a boolean decision variable for each item, we say that we take a subset of all our items to the market. This is equivalent to having a boolean decision variable for each item (where `true` would denote that the item is an element of the subset). In this example using set notation rather than individual boolean decision variables makes the code cleaner and therefore more intuitive. Line 9 and 10 encode a constraint. Particularly, the sum of the sizes of all items that we take to the market must not exceed the maximum load of the van. Finally, Line 12 and 13 declare the objective, namely the sum of the values of all items that we take to the market should be maximised (that is, the profit should be as large as possible).

Figure 3 shows the output of the stone carver model. By default MiniZinc shows only the values of the decision variables (in our case `take`, Line 3), however, the user might declare a more informative and richer custom output. We see that for our model we should take items one and three to the market (which give us a combined profit of four and a combined size of four which is equivalent to the maximum load of the van). Line 5 denotes that this solution is optimal. In other words, there is no other *valid* solution which gives a greater profit. A solution is called valid iff all constraints are satisfied (for example taking items one, two and three is not a valid solution because their combined size exceeds the maximum load of the van. Therefore the constraint in Line 9 and 10 is not satisfied).

```
1  Compiling stone_carver.mzn
2  Running stone_carver.mzn
3  take = {1,3};
4  ----------
5  ==========
6  Finished in 161msec
```

Figure 3: CP - Stone Carver Output

```
include "globals.mzn";
set of int: GUARD = 1..3;
set of int: NIGHT = 1..7;

array[NIGHT] of var GUARD: rota;

constraint forall(n in 1..6)
            (rota[n] !=
             rota[n+1]);
constraint global_cardinality_low_up(
            rota,
            [g | g in GUARD],
            [2 | g in GUARD],
            [3 | g in GUARD]);
constraint rota[1] != 1;
constraint rota[2] != 1;
constraint rota[5] != 2;

solve satisfy;
```

Figure 4: CP - Security Guard Rota

[1]Note that the smallest index of an array in MiniZinc is 1 rather than 0.

Figure 4 shows the MiniZinc model of the guard rota. Line 2 defines the set of guards and Line 3 the set of nights (where nights are labelled from one to seven inclusive). Line 5 defines the decision variable `rota`, namely for each night we have to pick precisely one guard. Lines 7 to 9 define that for all nights (except for the very last one) the guard scheduled to work during this night has to be different than the guard scheduled for the next night. We have to exclude the seventh night from this constraint because it does not have a successive night as it is the last night (if we include it, it would cause an out of bounds error). Lines 10 to 14 make sure that each guard is scheduled for two or three night shifts. The constraint says that each guard (Line 12) has to appear at least twice (Line 13) and at most three times (Line 14) in the decision variable array `rota` (Line 11). The number of occurrences of a constant in a decision variable array is called the *cardinality*. Cardinality constraints are not natively supported by MiniZinc but are included in the standard library which is loaded into the model in Line 1. Lines 15 and 16 define that Guard 1 can not be assigned to work in the first and second night respectively (because he is unavailable). Equally, Line 17 defines that the shift in the fifth night can not be assigned to Guard 2. Finally, Line 19 asks to solve the problem such that all constraints are satisfied.

The result of running MiniZinc on the rota model can be seen in Figure 5. A valid (but not necessarily unique solution) is the rota shown in Line 3 of the output. Other than in the previous example we do not wish to optimise our rota in any way, we simply require a valid rota.

```
1  Compiling rota.mzn
2  Running rota.mzn
3  rota = array1d(1..7 ,[3, 2, 3, 2, 1, 2,
        1]);
4  ----------
5  Finished in 222msec
```

Figure 5: CP - Security Guard Rota Output

1.2.1 Decision vs. Optimisation Problem

In the first example we try to maximise our profit. Whenever, we are maximising or minimising a number (in this case the sum of the values of the items which we take to the market, i.e. the profit) we call the problem an *optimisation problem*. While solving an optimisation problem we might find incrementally better solutions. For instance, MiniZinc might first explore subsets with only a single element, that is, we only take one item to the market. This might give us valid solutions which are not *optimal* yet (the profit is not maximal). Finding the optimal solution can often take a very long time. Therefore, optimisation problems are usually solved with a time limit. If the optimal solution can not be found within the limit, the best solution found so far is reported instead (which is often reasonably close to the optimal solution).

The second problem, on the other hand, is a *decision problem* (which belongs to the class of *Constraint Satisfaction Problems*, CSP). That is, the first valid solution that MiniZinc encounters is reported. Another way to look at decision problems is to think of them as yes or no questions (hence, decisions). Particularly: Is there at least one solution such that all the constraints are satisfied? There can only be two answers to this question, namely: “No there is no solution” or “Yes and here is the solution I found” (where the actual solution might not be unique).

Conceptually, decision and optimisation problems are very different. However, it is easy to show that optimisation problems can be built using iterative decision problems. To go back to our first example, we might first ask: ”Is there a solution such that we load the van with some of our items and go to the market without overloading the van?”. This is a decision problem so it might give us an answer such as: `take = {1};` (with a profit of two). In the next iteration we would ask the same question and add the additional constraint that the profit must be larger than two. We repeat this process until no better solution can be found (that is, the decision problem becomes *unsatisfiable*). We can also approach this problem the other way round, namely we first ask whether there is a valid solution such that the profit is bigger or equal to seven (using seven as the upper bound because it is the combined value of all items that we own). As long as the decision problem is unsatisfiable we keep decrementing the profit constraint (that is, in the second iteration we would ask for solutions with a profit of at least six and so on). The value of the optimal solution is called the *critical number* (sometimes also *crossover point*). In our small example the critical number, that is the profit, is equal to four.

Numerous previous (empirical) studies have shown that the computational complexity of a decision problem increases as we approach the critical number (in other words, the difficulty peaks at the critical number)[6, 15, 8, 16, 17, 41]. This phenomena is called the *phase transition*. Figure 6 shows a schema of the phase transition[2]. As a rule of thumb we can say that it is easy to find a valid solution for an optimisation problem but it becomes increasingly harder to improve the best found solution so far (as we are ”climbing” towards the peak).

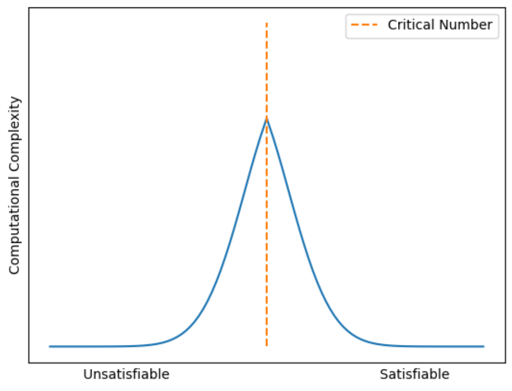

Figure 6: Phase Transition - Schema

Clearly, the CVRPTW is an optimisation problem because we wish to minimise the total distance travelled by all vehicles combined. Because of the phase transition phenomena it can be expected that finding the optimal solution

[2]Note that this schema corresponds to a minimisation problem. A maximisation problem, such as the stone carver example, would have the labels ”Unsatisfiable” and ”Satisfiable” swapped.

for an instance of the CVRPTW is very difficult. It is therefore reasonable to aim to find a solution which is fairly close to the optimal solution within a time limit instead. In reality it is often more economical to use a reasonably good solution (for example, within 5% of the optimal solution) instead of waiting for hours, days or weeks in order to obtain the optimal solution or investing into additional computational power to speed up the solving process. Therefore, our work aims to produce reasonably good solutions within a short time but optimality is not strictly required.

1.2.2 Symmetry Breaking

Often entities in a model are interchangeable. Let us revise our security guard rota example and imagine all three guards are available on all nights (this corresponds to deleting Lines 15 to 17 in the model shown in Figure 4). In this case the work schedule for Guard 1 would equally work for Guards 2 and 3. Another way to think about this is: any two guards could swap their schedule and all constraints would still be satisfied. This is called a *symmetry*. Symmetric solutions might slow the solving process down substantially. This is because symmetric solutions (regardless of whether they are valid or invalid) are explored multiple times. Concretely, if we have three symmetric values which have to fill three slots, and each slot has to have a different value, we can choose from three values in the first slot, two in the second slot and one in the last slot ($3 \times 2 \times 1 = 3! = 6$). Clearly symmetries create a huge overhead and should be avoided. Symmetries can be avoided by adding additional constraint which force the symmetric values to take a certain order, this is called *symmetry breaking*. Typically, lexicographical ordering or value proceed chains are used for symmetry breaking. In the security guard rota we might use a value proceed chain over the `rota` decision variable array. That means, the first shift of a guard must be before the first shift of the guard's successor (where the successor is the guard with the next highest number; for instance, Guard 3 is the successor of Guard 2). In other words, the first shift must be assigned to Guard 1, the second shift must be assigned to Guard 1 or 2, the third shift must be assigned to Guard 1 or 2 or, if the second shift was assigned to Guard 2, 3.

While symmetry breaking can speed up the solving process significantly, it can also break a correct model if it is not implemented correctly. Therefore, symmetry breaking should only be added to the model once we are sure that the model works correctly.

1.2.3 Backtracking and Search Heuristics

Typically a CP solver uses a *backtracking* algorithm to reach a valid solution. In particular, it takes the first decision variable which has not been *instantiated* (no value has been assigned to it yet), it instantiates the variable with the first value of the variable's domain and reduces the domains of all remaining non-instantiated decision variables using *constraint propagation*. Constraint propagation enforces that all domains of the remaining decision variables only contain values that do not conflict with the newly instantiated value of the current variable.

For example, let us consider two decision variables `var 1..3: a;` and `var 1..3: b;` (both with the initial domain $\{1, 2, 3\}$) as well as a constraint to say that `a` must be different than `b` (`constraint a != b;`). The solver takes the first non-instantiated variable and assigns it to the first value of its domain, that is `a = 1`. Now `b`'s domain can be reduced to $\{2, 3\}$ because value 1 can not satisfy the constraint `a != b`.

If during constraint propagation a domain becomes empty (called a *domain wipeout*), we trigger a backtrack. That means, we will try the next value of our current variable instead. If we run out of values for our current variable we will try the next value of the previously instantiated decision variable and so on. This way we explore all possible solutions and are guaranteed to find a solution if there is one (this is called a *complete search*). Moreover, because of constraint propagation, it is guaranteed that a found solution is valid. This property is called *soundness*.

The backtracking search strategy might not always work well because a difficult variable might be instantiated later than an easy variable (there is no general definition of what difficult and easy variables are; typically however, a difficult variable causes many backtracks because it can only take few values in a valid solution). Therefore, to reduce the number of backtrack calls, we want to instantiate more difficult variables before easier variables. Moreover, we might prefer some values in the domain over others (particularly if we know more about the nature of the problem). CP tool kits allow us to customise the order that variables are picked for instantiation and the order that values of the domain are chosen for instantiation. This is called *variable ordering heuristic* and *value ordering heuristic* respectively. If the order can change while the solver is running we also call it a *dynamic ordering heuristic*.

Choosing the right heuristics can speed up the solving process by an order of magnitude or more (equally, a badly chosen heuristic can slow the process down). Typically, heuristics are merely rules of thumb and have to be empirically evaluated. However, a common variable ordering heuristic which often leads to a significant speedup is the "Smallest Domain" variable ordering heuristic where the variable with the fewest remaining values in the domain is chosen for instantiation. Such standard heuristics are supported by most CP solvers and can often be added by changing a single line of code. Some tool kits such as Choco[42] also allow custom heuristics where the programmer can code their own heuristic.

2. PROBLEM DESCRIPTION

The previous section informally described the CVRPTW as well as CP. This section firstly argues why the CVRPTW is an important problem. Secondly, it gives a brief yet formal definition of the CVRPTW which will be used for the rest of this paper. This formal definition of the problem is loosely based on the work by Kallehauge, Larsen and Madsen[26].

2.1 Importance

The transportation of goods is a major factor in the economic world and often accounts for 10% to 20% of the gross national product of a country[13]. In previous decades the trend of globalisation and online trading has put further strains on the transportation sector[24]. Routing efficiently a fleet of vehicles over a given set of customers with additional time and capacity constraints is often the very essence of a successful logistic business. Improving supply chains can substantially help to reduce production costs and increase profit. Clearly even small improvements in the transportation sector can have a significant impact at a national level and improve prosperity.

2.2 Formal definition

A set of n customers has to be served by a set of m vehicles. Customer i has a specific demand for goods r_i and the sum of demands served by a single vehicle must not exceed the vehicle capacity Q. We only consider a homogeneous fleet of vehicles which means that each vehicle has the same capacity. c_{ij} is the cost/distance to travel from location i to location j. τ_{ij} is the time to travel from i to j, for simplicity we assume that the travel time is equivalent to the distance $\tau_{ij} = c_{ij}$. Each customer i has a time window in which service has to commence $[a_i, b_i]$. Equally, the depot has a time window $[a_i, b_i]$ which denotes the earliest and latest time a vehicle can leave and return respectively. s_i is the time needed to serve customer i. t_i corresponds to the time that service commences at customer i. The depot is represented twice: location 0 corresponds to the depot for leaving vehicles, location $n+1$ is the depot for returning vehicles. x_{ijk} is equal to 1 if vehicle k travels directly from location i to location j, 0 otherwise. Below, Table 1 lists all necessary variables for the formal definition.

Table 1: Formal definition variables

n	Number of customers
$C = \{1..n\}$	Set of customers
$N = C \cup \{0, n+1\}$	Set of customers and depot
m	Number of vehicles
$M = \{1..m\}$	Set of vehicles
$c_{ij} = \tau_{ij} \quad i,j \in N$	Cost/time to travel from i to j
$r_i \quad i \in C$	Demand of customer i
Q	Vehicle capacity
$a_i \quad i \in N$	Earliest time i can be served
$b_i \quad i \in N$	Latest time i can be served
$x_{ijk} = \{0,1\}$ $i,j \in Nk \in M$	Vehicle k travels from i to j
$s_i \quad i \in N$	Service duration at i
$t_i \quad i \in N$	Service start time at i

Objective:

$$minimise \sum_{k\in M}\sum_{i\in N}\sum_{j\in N} c_{ij}x_{ijk} \tag{2.1}$$

Subject to:

$$\sum_{i\in N}\sum_{k\in M} x_{ijk} = 1 \quad \forall j \in C \tag{2.2}$$

$$\sum_{j\in N}\sum_{k\in M} x_{ijk} = 1 \quad \forall i \in C \tag{2.3}$$

$$\sum_{j\in N}\sum_{k\in M} x_{0jk} = m \tag{2.4}$$

$$\sum_{i\in N}\sum_{k\in M} x_{i(n+1)k} = m \tag{2.5}$$

$$\sum_{i\in C}\sum_{j\in N} x_{ijk}r_i \leq Q \quad \forall k \in M \tag{2.6}$$

$$(t_i + s_i + \tau_{ij})x_{ijk} \leq t_j x_{ijk} \quad \forall i,j \in N \quad \forall k \in M \tag{2.7}$$

$$s_0 = s_{n+1} = 0 \tag{2.8}$$

$$a_i \leq t_i \leq b_i \tag{2.9}$$

The objective and the constraints above have the following meanings:

- (2.1) The objective of the problem is to minimise the sum of the cost of all used routes by all vehicles.
- (2.2) For every customer j, there must be precisely one vehicle k which goes to j from an arbitrary location i.
- (2.3) For every customer i, there must be precisely one vehicle k which leaves i to go to an arbitrary location j.
- (2.4) There are precisely m routes leaving the depot at location 0 and go to an arbitrary location j. Unused vehicles “go” directly from the depot at location 0 to the depot at location $n+1$
- (2.5) There are precisely m routes which go to the depot at location $n+1$ coming from an arbitrary location i.
- (2.6) For every vehicle k: The sum of all demands of customers visited by k must not exceed the vehicle capacity Q.
- (2.7) For all locations and vehicles: If vehicle k goes from location i to location j then the leaving time from location i (which corresponds to the service start time plus the service duration) plus the time to travel from i to j is the earliest time that service can begin at location j.
- (2.8) There is no service time associated with the depot.
- (2.9) Service at location i has to commence within the time window.

3. KEY IDEAS

We developed two models for the CVRPTW. The first model is loosely based on models discussed in the literature (see also Section 6). The second model, on the other hand, exploits the fact that we are considering a homogeneous vehicle fleet (all vehicles are equivalent). This section presents the two models using a concrete example and high-level descriptions of how the two models work.

3.1 Example Problem

We will explain the two models using an example with five customers ($n = 5$), a single depot (denoted with $n+1$) and a fleet of five vehicles ($m = 5$) each with a capacity of five units ($Q = 5$). In particular, we will use the solution shown in Figure 7 to exemplify how the two models work. Note that only three of the five vehicles are used in this particular solution.

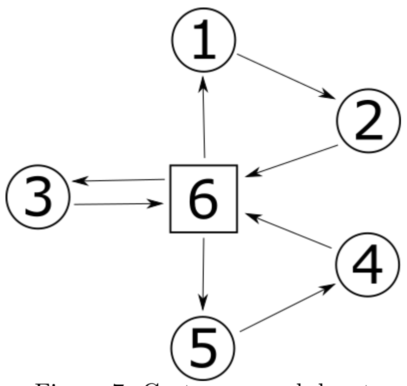

Figure 7: Customers and depot

Table 2: Distance matrix

	1	2	3	4	5	6
1	0	4	2	4	4	3
2	4	0	5	1	5	1
3	2	5	0	4	5	4
4	4	1	4	0	5	2
5	4	5	5	5	0	5
6	3	1	4	2	5	0

The solution shown in Figure 7 is valid, however, not necessarily optimal in the sense that there might be solutions with a shorter total distance. The total distance is the path length of all vehicle routes summed up using the distance matrix shown in Table 2. In our example the total distance is equal to 28 where the respective routes are of length eight ($6 \xrightarrow{|3|} 1 \xrightarrow{|4|} 2 \xrightarrow{|1|} 6$), eight ($6 \xrightarrow{|4|} 3 \xrightarrow{|4|} 6$) and 12 ($6 \xrightarrow{|5|} 5 \xrightarrow{|5|} 4 \xrightarrow{|2|} 6$) where the numbers above the arrows denote the distance between the locations.

Table 3: Demand

Customer	1	2	3	4	5
Demand	2	3	3	1	4

The sum of the customer demand (as given in Table 3) along a vehicle route must not exceed the vehicle capacity of five. In our example the summed demands along the three vehicle routes are equal to five ($6 \rightarrow 1_{[2]} \rightarrow 2_{[3]} \rightarrow 6$), three ($6 \rightarrow 3_{[3]} \rightarrow 6$) and five ($6 \rightarrow 5_{[4]} \rightarrow 4_{[1]} \rightarrow 6$) where the subscripts denote the customer's demand.

Table 4: Time Windows

Location	1	2	3	4	5	6
Earliest	5	5	5	10	5	0
Latest	50	45	50	55	45	80

Table 5: Service

Customer	1	2	3	4	5
Service time	3	6	2	1	3

Finally, each customer has a time window as shown in Table 4 in which service has to commence. Furthermore, the time window of the depot denotes the earliest leaving and latest returning time for all vehicles. For simplicity (and by convention) vehicles need one time unit to travel one distance unit (for example, a vehicle needs four time units to travel from Customer 1 to 4). Vehicles may arrive before the time window but need to wait until the earliest time of the time window to commence service. Servicing a customer takes as many time units as specified in Table 5. If we assume that each vehicle always leaves every customer as soon as possible, the three vehicles would have a schedule as shown in Table 6. Note that any schedule is valid as long as the "Start service" entries lie within the customer's time window and the vehicle's "Leave depot" and "Finish" is within the time window of the depot ([0,80]).

Table 6: Possible schedules

$6 \rightarrow 1 \rightarrow 2 \rightarrow 6$		$6 \rightarrow 3 \rightarrow 6$		$6 \rightarrow 5 \rightarrow 4 \rightarrow 6$	
0	Leave depot	0	Leave depot	0	Leave depot
3	Arrive at 1	4	Arrive at 3	5	Arrive at 5
5	Start service	5	Start service	5	Start service
8	Leave 1	7	Leave 3	8	Leave 5
12	Arrive at 2	11	Finish	13	Arrive at 4
12	Start service			13	Start service
18	Leave 2			14	Leave 4
19	Finish			16	Finish

3.2 Model 1

Model 1 has an array of decision variables for each vehicle which represents the route of the vehicle. The routes of all vehicles are saved in a single $m \times (n+1)$ matrix called *succ* (short for successor). Table 7 shows the successor matrix which corresponds to our example from Figure 7 (i.e. with instantiated decision variables).

Table 7: *succ* matrix

Vehicle \ Location	1	2	3	4	5	6
1	2	6	3	4	5	1
2	1	2	6	4	5	3
3	1	2	3	6	4	5
4	1	2	3	4	5	6
5	1	2	3	4	5	6

The first row of Table 7 shows the route of the first vehicle. Each column in a row represents a location (customer/depot) and the cell denotes which location is visited next (hence the name, successor). Cells where the cell content matches the cell's column denote a non-visit (for example, *succ*[2][4] = 4 means that Vehicle 2 does not visit Location 4). Therefore, the route of vehicle 1 goes from the depot (6) to Customer 1, to Customer 2 and back to the depot (6). Customers 3, 4 and 5 are not visited by Vehicle 1 as the according cells match with their columns.

We constrain that each vehicle which does not leave the depot, that is the cell in column 6 is equal to 6, must not visit any customer (so every other cell in this row must be equal to its column too). In our example Vehicles 4 and 5 do not leave the depot. Moreover, each customer ($1..n$) must be visited precisely once. We enforce this by constraining that for each column i (except for column $n+1$ which represents the depot) there are exactly $m-1$ cells in this column which take the value i leaving room for one row (that is, vehicle) to take a different value. Finally, we constrain that each row corresponds to the subtour constraint[4] meaning that all visited customers should build a single, circular tour.

Since we have a homogeneous fleet, we need to perform symmetry breaking (i.e. it does not matter which vehicle drives which route). We can perform symmetry breaking over the *succ* matrix by lexicographically ordering the rows in a decreasing order. That means, each row forms a number and this number has to be greater or equal than the number of the next row. Concretely, in our example symmetry breaking holds because: $263451 \geq 126453 \geq 123645 \geq 123456 \geq 123456$.

The number of used vehicles corresponds to the number of rows in *succ* where the last column (i.e. the depot column) is not equal to $n+1$ (see 3.1). That means each vehicle that leaves the depot is a used vehicle.

$$\#\text{vehicles} = \sum_{i=1}^{m} \begin{cases} 1, & \text{if } succ[i,(n+1)] \neq n+1 \\ 0, & \text{otherwise} \end{cases} \tag{3.1}$$

The total distance travelled can be easily calculated by summing up all distances from all cells to their successors for each row in *succ* (see 3.2). Each non-visited customer contributes 0 to the sum because $distance[i,i] = 0 \quad \forall i \in 1..n+1$.

$$\text{total distance} = \sum_{i=1}^{m} \sum_{j=1}^{n+1} distance[j, succ[i,j]] \tag{3.2}$$

For each vehicle we propagate the load constraint by calculating the demand of each vehicle route and limiting this route demand to be no bigger than the vehicle capacity. The route demand corresponds to the sum of all demands of customers along a vehicle route (see 3.3).

$$capacity \geq \sum_{j=1}^{n} \begin{cases} demand[j], & \text{if } succ[i,j] \neq j \\ 0, & \text{otherwise} \end{cases} \quad \forall i \in 1..m \tag{3.3}$$

The time constraint is encoded with an additional decision variable array, *arrive*, which denotes the arrival time of the serving vehicle for each location. The arrival time has to lie within the time window so that service can commence immediately (see 3.4). How can we propagate the arrival times along vehicle routes? Let s be the successor of customer c (that means, on a vehicle route s is visited immediately after c). Therefore, the arrival time of s is equivalent to the leaving time of c plus the time to travel from c to s. We know the time to travel from c to s from the distance matrix and the leaving time of customer c is the arrival time at c plus the service time ($arrive[c] + service[c]$). If the vehicle arrives too early at s (that is, before service can commence) it is forced to take longer for the drive and hence to arrive later at s (see 3.5). Every customer must be left on time such that there is enough time to return to the depot before it closes (see 3.6). For (3.5) we require the leaving time from the depot (which should be equivalent to the earliest possible leaving time) so that we can define the arrival time of the first customer of each route. As previously shown, the leaving time of a location is the arrival time plus the service time. Therefore, we set the "arrival" time at the depot to the opening time of the depot and declare that the depot's service time is equal to 0 (see 3.7). Note that the arrival time can also be used for subtour elimination similar to Model 2.

$$earliest[i] \leq arrive[i] \leq latest[i] \quad \forall i \in 1..n+1 \tag{3.4}$$

$$\begin{aligned} & succ[i,j] \neq n+1 \land succ[i,j] \neq i \rightarrow \\ & \quad arrive[succ[i,j]] = max(arrive[j] + service[j] + \\ & \qquad distance[j, succ[i,j]], earliest[succ[i,j]]) \\ & \quad \forall i \in 1..m, j \in 1..n+1 \end{aligned} \tag{3.5}$$

$$\begin{aligned} & arrive[i] + service[i] + \\ & distance[i, n+1] \leq latest[n+1] \quad \forall i \in 1..n \end{aligned} \tag{3.6}$$

$$arrive[n+1] = earliest[n+1] \land service[n+1] = 0 \tag{3.7}$$

3.3 Model 2

Model 2 takes advantage of the fact that we only consider a homogeneous fleet. For this reason, the notion of assigning a customer to an actual vehicle route is completely circumvented. Instead, we have only one decision variable array, *pred* (for predecessor), which denotes for each customer c its predecessor p (the vehicle which visits p, visits c immediately afterwards) and a second boolean decision variable array, *last*, which denotes for each customer whether the visiting vehicle returns to the depot afterwards (this customer is the last on the route). Finally, the boolean decision variable array *is_pred* indicates whether a customer is a predecessor of another customer. *is_pred* is the inverse of *last* because a customer c can only be a predecessor of another customer iff the visiting vehicle does not return to the depot after visiting c and vice versa. *is_pred* is only needed for convenience in the constraints. Table 8 shows the instantiated decision variables for the solution shown in Figure 7.

A vehicle route can be derived in reverse order using *pred* and *last*. For example, we know that each route ends at the depot (6), then we look for a customer which is the last customer on its route such as Customer 2 ($6 \leftarrow 2$), the predecessor of Customer 2 is Customer 1 so we can add 1 to the route ($6 \leftarrow 2 \leftarrow 1$), finally the predecessor of Customer 1 is the depot so we have finished the route ($6 \leftarrow 2 \leftarrow 1 \leftarrow 6 = 6 \rightarrow 1 \rightarrow 2 \rightarrow 6$).

Table 8: *pred*, *last* and *is_pred* decision variable arrays

Customer	1	2	3	4	5
pred	6	1	6	5	6
last	0	1	1	1	0
is_pred	1	0	0	0	1

Arguably, having a decision variable array for customer predecessors is less intuitive than an array of customer successors. However, using predecessors rather than successors allow the capacity and timing constraints to be much shorter and concise.

The number of used vehicles corresponds to the number of vehicles returning to the depot (see 3.8).

$$\#\text{vehicles} = \sum_{i=1}^{n} last[i] \tag{3.8}$$

If a customer i is a predecessor of another customer ($is\_pred[i] = 1$) then i has to occur in *pred* exactly once. For example, Customer 1 is the predecessor of another customer

($is\_pred[1] = 1$) meaning there must be exactly one entry in *pred* which takes the value 1 (in our example: $pred[2] = 1$).

Furthermore, each used vehicle must leave the depot. That means there must be *#vehicles* customers with the depot as the predecessor. In other words, there must be exactly *#vehicles* entries in *pred* with the value $n + 1$. In our example there are three used vehicles ($= \sum last$) which means there must be exactly three customers with the depot as the predecessor ($pred[1] = 6$, $pred[3] = 6$ and $pred[5] = 6$). We can use the cardinality constraint, which was introduced in Subsection 1.2, to implement these two properties as sketched in (3.9).

$$\mathrm{card}(pred, [1, 2, ..., 6], [is\_pred[1], ..., is\_pred[5], \#vehicles]) \tag{3.9}$$

The total distance travelled can be calculated by summing up the distance of each element in *pred* to the index of the element; that is, from the predecessor to the current customer. Moreover, we need to add the distance from the current customer back to the depot iff the current customer is the last one on the route (see 3.10).

$$\text{total distance} = \sum_{i=1}^{n} dist[pred[i], i] + dist[i, n + 1] * last[i] \tag{3.10}$$

The array *load* saves the current load of the visiting vehicle when it leaves each location. Each vehicle leaves the depot completely loaded. At each customer the vehicle's load is reduced by the customer's demand (see 3.11). By restricting the domain of *load* to take values between 0 and *capacity* (in our example 5), we enforce the load constraint to hold.

$$load[i] = load[pred[i]] - demand[i] \quad \forall i \in 1..n \tag{3.11}$$

Moreover, the load constraint (as well as the time window constraint) eliminate subtours. Proof by contradiction: Imagine a subtour between Customers 1 and 2 existed. Therefore we have:

$$\begin{aligned} load[1] &= load[pred[1]] - demand[1] \\ &= load[2] - demand[1] \end{aligned} \tag{3.12}$$

$$\begin{aligned} load[2] &= load[pred[2]] - demand[2] \\ &= load[1] - demand[2] \end{aligned} \tag{3.13}$$

Now we can insert the first equation (3.12) into the second equation (3.13) such that:

$$\begin{aligned} load[2] &= load[1] - demand[2] \\ load[2] &= (load[2] - demand[1]) - demand[2] \\ \Leftrightarrow 0 &= -demand[1] - demand[2] \\ 0 &= demand[1] + demand[2] \end{aligned} \tag{3.14}$$

Because we know that demands have to be positive Equation 3.14 is a contradiction (similar for the time window constraint). □

The timing constraint requires a further decision variable array, *leaving_time*, which, for each location, encodes the time when the serving vehicle is departing from the location. The leaving time from the depot is the earliest possible leaving time which corresponds to the depot opening time. For every customer the leaving time corresponds to the time when service is finished at the customer which is equivalent to the service start time plus the service duration of the customer. The service start time is the time that the serving vehicle arrived at this customer but not earlier than the earliest time service can start (lower bound of the customer's time window, see 3.15).

$$\begin{aligned} leaving\_time[i] = service[i] + \\ max(earliest[i], \\ leaving\_time[pred[i]] + \\ distance[pred[i], i]) \\ \forall i \in 1..n \end{aligned} \tag{3.15}$$

Eventually, service at each customer has to start before the end of the customer's time window (see 3.16) and the leaving time at each customer must permit a return to the depot before it closes (see 3.17).

$$leaving\_time[i] - service[i] \leq latest[i] \quad \forall i \in 1..n \tag{3.16}$$

$$\begin{aligned} leaving\_time[i] + \\ distance[i, n + 1] \leq latest[n + 1] \\ \forall i \in 1..n \end{aligned} \tag{3.17}$$

4. SEARCH HEURISTICS

In this section we present a number of search heuristics which we considered to be particularly promising for the CVRPTW. In particular, our work focused on heuristics for a random restart strategy, called *Luby search*[34]. This search strategy has been shown to be often a successful strategy especially for difficult problems[20].

4.1 Luby Search

The idea behind a restart search strategy is to quickly explore various areas of the search tree. Therefore, the used search heuristics must have a notion of randomness so that the search takes a new path (probabilistically) with each restart. Furthermore, the strategy needs to periodically trigger a restart (that is, the search starts from the top again) in order to prevent the search from being stuck in a local minimum (see Figure 8, a normal backtracking algorithm would explore the tree from left to right). After each restart Luby Search increases the time to the next restart in a non-linear fashion according to the Luby Sequence. The Luby Sequence is iteratively defined: The sequence starts with 1 (base case), the existing sequence is repeated and the double of the last number of the existing sequence is added to the end (1, *1, 2*, 1, 1, 2, 4, *1, 1, 2, 1, 1, 2, 4, 8*, 1, ...). Because the Luby Sequence is gradually increasing it eventually exceeds the size of the search tree. Therefore, Luby Search is a complete search strategy[34]. Moreover, we can use *no-goods* recording in order to prevent the search from exploring paths which were already explored in a previous restart. No-goods recording has a computational and space overhead, however, the overhead is typically acceptable for the speed up that no-goods provide.

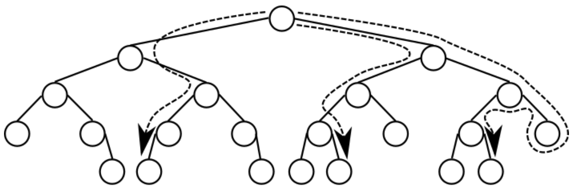

Figure 8: With each restart a new random path is explored. Note: a search tree is not a data structure but the call stack of the recursive backtracking algorithm. Note also that the Figure shows a binary search tree for simplicity.

To summarise, if we use Luby Search we need to use a variable/value ordering heuristic with a notion of randomness. Furthermore, we can optionally enable no-goods recording which often speeds up the solving process.

4.2 Variable ordering heuristics

We identified the following variable ordering heuristics as particularly promising for the CVRPTW.

- **Smallest Domain** The decision variable with the fewest remaining values in its domain is chosen for instantiation. This means, the most isolated or restricted customer is chosen for instantiation.
- **Farthest Nearest Neighbour** For each uninstantiated customer i we identify which customer j in i's domain is closest to i. From all customers the one with the greatest distance to its nearest neighbour j is chosen for instantiation.
- **Random** a random variable is chosen for instantiation.
- **Probabilistic** it has been shown that bad decision are tend to be made early in the search tree[22, 21]. The probabilistic search heuristic picks initially random decision variable for instantiation. As more and more decision variables are instantiated the heuristic will start using the Farthest Nearest Neighbour heuristic instead. In other words, the heuristic is a hybrid between the Random and the Farthest Nearest Neighbour heuristics where the former is more likely to be chosen higher in the search tree and the later at deeper depths of the search tree.

4.3 Value ordering heuristics

We identified the following value ordering heuristics as particularly promising for the CVRPTW.

- **Smallest Value** Take the first value of the domain.
- **Nearest Neighbour** Of all neighbours in the customer's domain chose the neighbour which is closest to the customer.
- **Random** Chose a random value of the domain.

5. EVALUATION

We evaluated various combinations of the heuristics shown in the previous section. In this section we briefly describe the evaluation methods and show the results of our evaluation. All our evaluations were run on the *Formal Analysis, Theory and Algorithms* (FATA) group cluster at the University of Glasgow. The cluster consists of six nodes each with the properties shown in Table 9. Each instance of each evaluation run was given 15 minutes on a single processor core of the cluster. We used the Solomon Benchmark[51] to evaluate the models and heuristics as this has been the *de-facto* standard benchmark for the CVRPTW for many years. However, we only used instances of the benchmark for which the optimal solution is known in order to gain a better estimate of how well each heuristic performs.

Table 9: Cluster node properties

Processor	Dual Intel Xeon E5-2697A v4
Memory	512GB Ram
OS	Ubuntu 17.04

Figure 9 shows the best found distance of Model 1 and Model 2 (with both Choco and MiniZinc) plotted against the optimal solution. The closer an instance is to the diagonal line the better it performed (the closer it is to the optimal solution). Note that no instance can be below the diagonal line as this would indicate that the model found a solution which is better than the optimal solution (which is impossible). In Figure 9 we only plot the results of the 25 customers instances because Model 1 was not able to find solutions for any instances of the 50 and 100 customers instances within the time limit. Both models used the "Smallest Domain" variable ordering and "Smallest Value" value ordering heuristics.

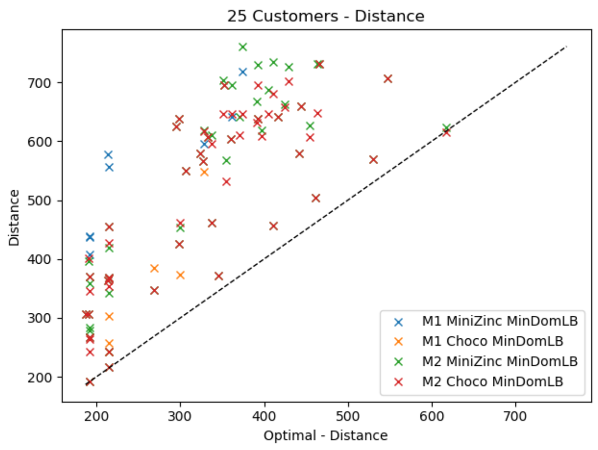

Figure 9: Model 1 (Choco/MiniZinc) vs Model 2 (Choco/MiniZinc)

Figure 10 and Figure 11 show the number of used vehicles for each solved instance plotted against the number of vehicles of the optimal solution for Model 1 and 2 respectively. We note that it is often possible to find solutions which use fewer vehicles for the cost of increasing the total combined distance of all vehicles. It is therefore **not** the case that the optimal solution uses as few vehicles as possible.

From Figure 9 we can conclude that Model 2 outperforms Model 1 significantly (keeping in mind that Model 2 found solutions for all instances while Model 1 only found solutions for some instances of the 25 customers problem instances). Moreover, we note that the Choco implementation of Model 2 performs slightly better than the MiniZinc implementation. Therefore, we used Model 2 implemented in Choco to evaluate all further heuristic combinations.

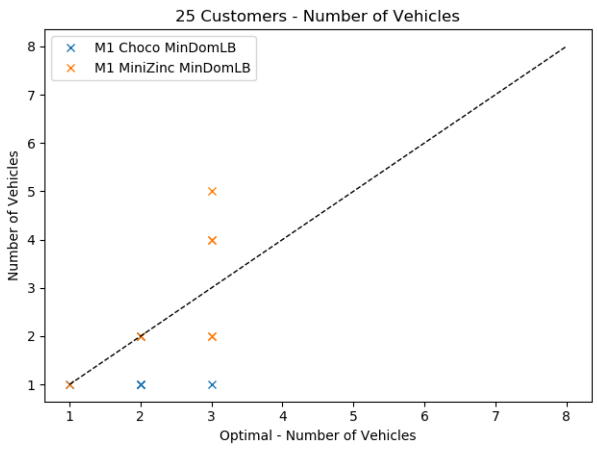

Figure 10: Model 1 (Choco/MiniZinc) number of vehicles

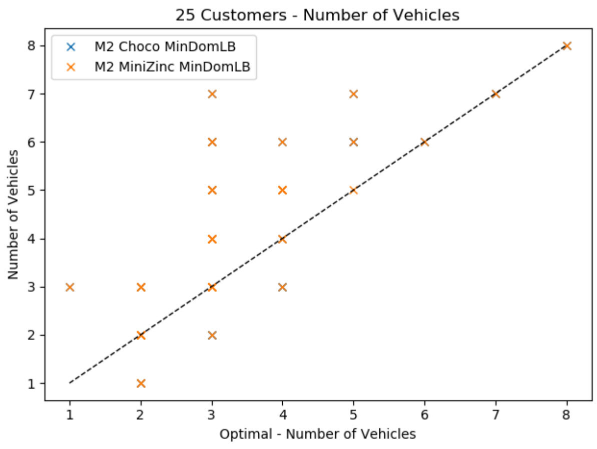

Figure 11: Model 2 (Choco/MiniZinc) number of vehicles

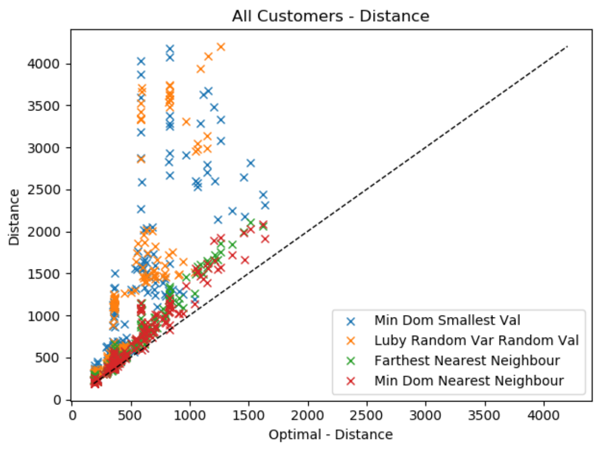

Figure 12: Performances of various heuristics

Figure 12 shows the performance of four different heuristics plotted against the optimal solution on the x-axis. Looking at the plot we can derive that the "Farthest Nearest Neighbour Variable - Nearest Neighbour Value" and "Smallest Domain Variable - Nearest Neighbour" heuristics outperform significantly the "Smallest Domain Variable - Smallest Value" and the Luby "Random Variable - Random Value" heuristics. In the next steps we aim to isolate the best performing variable and value ordering heuristics. For this we compare one to one the performance of two heuristics gathering evidence for good and bad heuristics with each step.

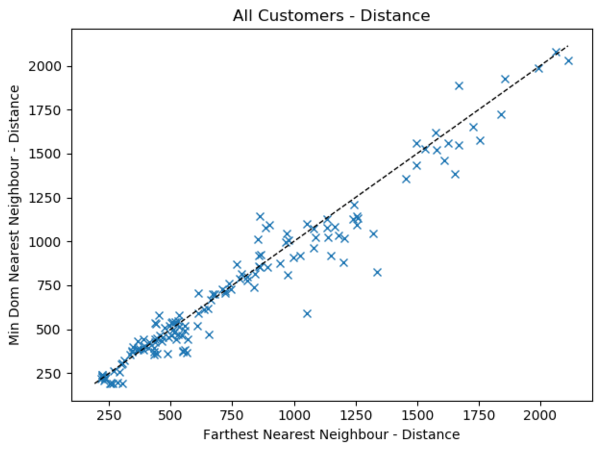

Figure 13: Smallest Domain Nearest Neighbour vs Farthest Nearest Neighbour

In Figure 12 we saw that the "Farthest Nearest Neighbour Variable - Nearest Neighbour Value" heuristic as well as the "Smallest Domain Variable - Nearest Neighbour" heuristic perform very well. Figure 13 compares the two heuristics directly to derive which one performs better. Every point in the plot corresponds to one problem instance where the x and y coordinates are determined by the total distance of the corresponding heuristic. Points on the diagonal (dashed line) correspond to instances for which both heuristics determine equally good solutions (note that this might not be the same solution). Informally: the more instances a heuristic can place deep in its opponents triangle the better it performs. Figure 13 shows that the "Smallest Domain Variable - Nearest Neighbour" heuristic seems to perform slightly better on average but the difference is not of statistical significance.

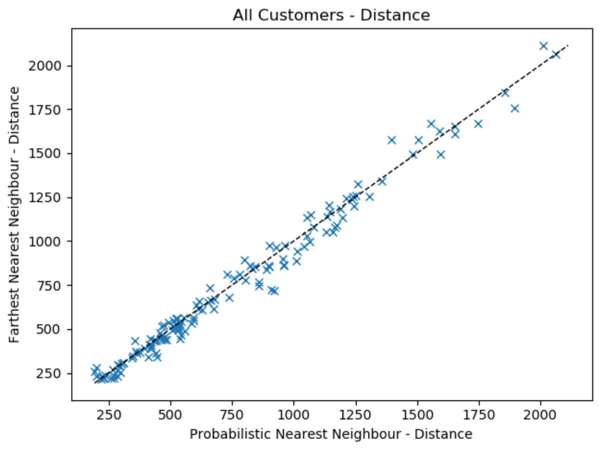

Figure 14: Performances of probabilistic heuristic

We developed the probabilistic variable ordering heuristic with the aim to improve the farthest nearest neighbour variable heuristic. However, Figure 14 shows that both heuristics perform in fact equally well. This, as well as the findings of Figure 13, lends weight to the assumption that the "Nearest Neighbour" value ordering heuristic is of greater importance than the variable ordering heuristic.

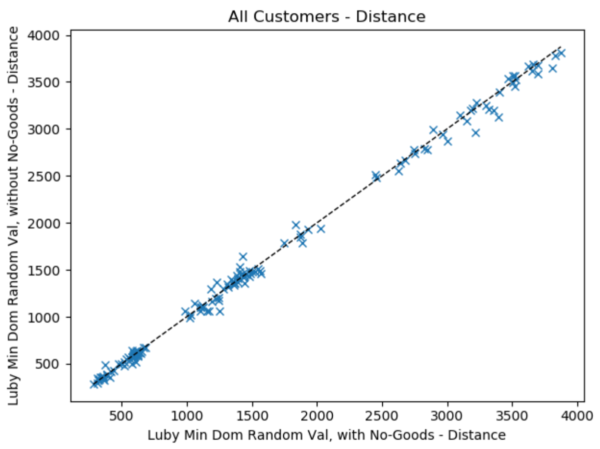

Figure 15: Effect of no-goods recording

Furthermore, we ran a number of experiments with Luby Search. Firstly, we wanted to determine the computational advantage/disadvantage of using no-goods recording. Figure 15 shows the effect of running Luby Search with (x-axis) and without (y-axis) no-goods recording (using the same search heuristics). The figure clearly shows that, in our case, no-goods recording had no significant performance impact. We decided to use no-goods recording for all further evaluation with Luby Search because the literature reports mostly positively of no-goods recording.

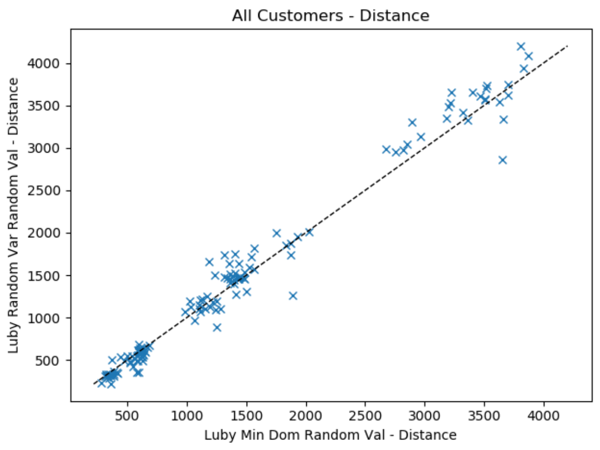

Figure 16: Luby Random Variable vs Smallest Domain (both Random Value)

Figure 16 shows the performance of Luby Search with random variables versus Luby Search where the variable with the smallest domain is chosen for instantiation (both use the "Random Value" heuristic). Both heuristics seem to perform roughly equally well.

Finally, we compared Luby Search with "Farthest Nearest Neighbour Variable - Random Value" against the "Farthest Nearest Neighbour Variable - Nearest Neighbour Value" heuristic. The results are plotted in Figure 17 and clearly show that the latter performs significantly better. This suggests the importance of the "Nearest Neighbour" value ordering heuristic.

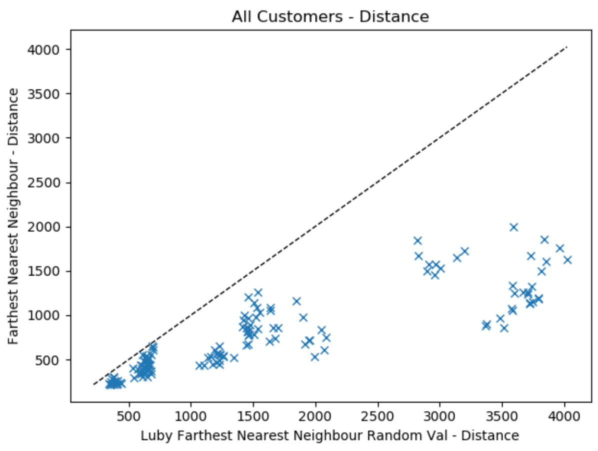

Figure 17: Farthest Nearest Neighbour vs Luby Farthest Nearest Neighbour Random Value

Section 7 contains a more thorough discussion regarding the findings presented in this section. Furthermore it considers further combinations of variable/value ordering heuristics for future work. The next section contains a brief literature review and discusses related work.

6. RELATED WORK

The VRP was first formalised by Dantzig and Ramser in 1959[9] and several refinements and extensions have been proposed since then. This section reviews a number of the most relevant and recent related work.

6.1 Variants

Most commonly, the VRP is extended with additional constraints to make it more applicable to real-world logistic problems[45]. The literature considers predominantly *timing* and *capacity* constraints[29] (CVRPTW).

The timing constraint consists of an additional time matrix to show the travel times between any two customers or the depot and a customer. Typically, the time matrix is the product of a scalar multiplication of the distance matrix. Each customer has a time-window in which the serving vehicle has to arrive. Furthermore, the depot has a time window such that no vehicle can leave the depot before it opens and all vehicles have to return to the depot before it shuts. This problem class is called the Vehicle Routing Problem with Time Windows (VRPTW)[52, 2, 3].

The second common constraint which is often added to the VRP by default is the capacity constraint. This constraint limits the amount of goods that each vehicle can transport and assigns a demand for a quantity of goods to each customer. The load of a vehicle can not be below zero or exceed its capacity at any time. This VRP variant is called Capacitated Vehicle Routing Problem (CVRP) or, if in combination with the Time Window constraint, CVRPTW[29, 27, 28].

Further common constraints and variants of the VRP which are worth being mentioned are the pickup and delivery problem (a vehicle can deliver and collect goods from a customer)[12, 33, 7, 44], the multiple trips variant (a vehicle can go back to the depot to reload)[14] and goods constraints which limit the type of goods which can be carried in the same vehicle[43, 40].

Over the last two decades much research has been done on the VRP and numerous extensions have been proposed and studied. The following are some of the more significant research streams. The Electric VRP composes routes for electric vehicles with a limited range and long charging times[23, 48], the Green VRP takes the environmental impact of the routes into account[59, 54], the Resource Distribution VRP considers the redistribution of resources amongst all nodes (for example bike sharing stations)[49, 55], the Trailer VRP calculates the VRP for HGVs by combining the towing vehicle and the trailer[5], similarly, Lam, Van Hentenryk and Kilby consider the independent movement of drivers and vehicles[31]. The Multi-Compartment VRP considers various compartments on each vehicle for different types of goods[11, 56]. All of these streams require novel constraints as they can not easily be modelled with existing ones.

6.2 Alternative Solving Approaches

Various solving approaches, other than CP, have been suggested and successfully applied for solving the VRP. While most of them are not as flexible as CP we acknowledge that many of them significantly outperform CP models (that is, they find better solutions faster). Each solving approach therefore has individual strengths and weaknesses.

Local Search takes an initial, non-optimal, solution and tries to improve the solution by applying move operators to the solution. In their influential paper "*Solving Vehicle Routing Problems Using Constraint Programming and Meta-Heuristics*" De Backer et al. propose a hybrid solver which uses Local Search and CP[1]. Their Local Search has the following well established move operators for the VRP:

- 2-Opt: Replace and insert two arcs in a route, all intermediate arcs change directions[32].
- Or-Opt: One, two or three neighbouring nodes are moved somewhere else in the route[38, 32].
- Relocate a single node to a different route and position[47].
- Swap two nodes in their positions[47].
- Swap the end paths of two routes[47].

De Backer et al. propose four local search strategies: Tabu Search, Guided Local Search (in two versions) as well as a hybrid of Tabu and Guided Local Search. Tabu Search[18, 19] aims to make "spinning" searches in a local minimum impossible (forcing the local search to climb out of the local minimum). When a move is performed, this specific move is placed into a queue, called the tabu list, which has a limited length. If the tabu list is full, the move in the front of the list is dequeued and is no longer tabu. Therefore it is impossible to repeatedly perform the same sequence of moves (up to the length of the tabu list). Guided Local Search (GLS)[58] works on a similar basis. However, rather than making specific moves tabu it penalises features of a local minimum by augmenting the objective function such that visiting the local minimum becomes more expensive. Local search can be further improved by Large Neighbourhood Search (LNS)[50] which combines related nodes and therefore reduces the overall search space.

Integer Programming (IP) has also been successfully used to model and solve the VRP. An Integer Programming problem is an optimisation problem in which all decision variables can only take integer values. The IP variant *Mixed Integer Programming* (MIP), which allows variables to take non-integer values during the solving process but must end up with integer variables in the optimal solution, has been used by Kilby and Tommaso to solve a variant of the VRP[30]. Similarly, *Integer Linear Programming* (ILP), in which the objective function and the constraints have a linear relation[39, 36] as well as *Column Generation*, in which the problem is solved to optimality or near optimality in incremental steps such that each step introduces further decision variables[25], have both been successfully used in the context of the VRP.

Finally, **Genetic Programming** (GP) is a programming technique in which an algorithm is mutated in order to derive new and ultimately better algorithms (this process is a type of *Evolutionary Algorithms*). The set of algorithms derived from a mutation is called a generation. Each algorithm in a generation is tested with a fitness function (that is, the function evaluates how well each algorithm performs) and the fittest algorithms of a generation are selected to produce the next generation. Recent attempts to use GP for solving the VRP have turned out to perform particularly well[57].

7. CONCLUSION AND FUTURE WORK

Using the evaluation data shown in Section 5 we can conclude that Model 2 significantly outperforms Model 1. We infer that primarily the more compact model design leads to fewer constraints and finally the increased performance. We can derive the compactness of the models using space complexity analysis. Model 1 has a space complexity of $O(n \times m)$ because of the *succ* matrix (where n is the number of customers and m the number of vehicles). On the other hand, Model 2 has a space complexity of $O(n)$. The number of decision variables (space complexity) says little about the total number of instantiations because we also need to take the domain sizes of the variables into account. However, constraint propagation takes longer if there are more decision variables because with each instantiation more constraints need to be checked (assuming each decision variable has the same degree of constraints as it would have in a more compact model). Finally, we note that the symmetry breaking of Model 1 is a significant performance overhead compared to Model 2 where no symmetry breaking is required by design.

Moreover, our evaluation shows that in particular the "Nearest Neighbour" value ordering heuristic leads to very good performances. However, from our results we could not determine for definite which variable ordering heuristic provides the best performance. Equally, we were not able to confirm the usefulness of Luby Search in this context.

In conclusion, Model 2 outperforms Model 1 significantly because of its more compact design. We were able to ascertain an improved performance using the "Nearest Neighbour" value ordering heuristic. However, future work should further investigate the performance of various variable ordering heuristics. We suggest to use Model 2 and focus on Luby Search in combination with random or semi-random variable ordering heuristics (like the "Probabilistic" heuristic shown in Section 4) in combination with the "Nearest Neighbour" value ordering heuristic. Using the same model and value ordering heuristic throughout the evaluation will help to focus on the effect of variable ordering heuristics.

Acknowledgments. I would like to thank my supervisor, Patrick Prosser, for his great guidance and for asking critical questions which helped to improve this thesis substantially. Furthermore, I would like to thank Ciaran McCreesh who was always there to answer any practical or technical questions.

8. REFERENCES

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

[2] O. Bräysy and M. Gendreau. Vehicle routing problem with time windows, part I: route construction and local search algorithms. *Transportation Science*, 39(1):104–118, 2005.

[3] O. Bräysy and M. Gendreau. Vehicle routing problem with time windows, part II: metaheuristics. *Transportation Science*, 39(1):119–139, 2005.

[4] Y. Caseau and F. Laburthe. Solving small tsps with constraints. In L. Naish, editor, *Logic Programming, Proceedings of the Fourteenth International Conference on Logic Programming, Leuven, Belgium, July 8-11, 1997*, pages 316–330. MIT Press, 1997.

[5] I. Chao. A tabu search method for the truck and trailer routing problem. *Computers & OR*, 29(1):33–51, 2002.

[6] P. C. Cheeseman, B. Kanefsky, and W. M. Taylor. Where the really hard problems are. In *IJCAI*, volume 91, pages 331–340, 1991.

[7] J. Clarke, V. Gascon, and J. A. Ferland. A capacitated vehicle routing problem with synchronized pick-ups and drop-offs: The case of medication delivery and supervision in the DR congo. *IEEE Trans. Engineering Management*, 64(3):327–336, 2017.

[8] J. M. Crawford and L. D. Auton. Experimental results on the crossover point in satisfiability problems. In *AAAI*, volume 93, pages 21–27. Citeseer, 1993.

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

[10] T. Dantzig. *Number: The language of science.* Penguin, 2007.

[11] U. Derigs, J. Gottlieb, J. Kalkoff, M. Piesche, F. Rothlauf, and U. Vogel. Vehicle routing with compartments: applications, modelling and heuristics. *OR spectrum*, 33(4):885–914, 2011.

[12] Y. Dumas, J. Desrosiers, and F. Soumis. The pickup and delivery problem with time windows. *European journal of operational research*, 54(1):7–22, 1991.

[13] M. Fisher. Vehicle routing. *Handbooks in operations research and management science*, 8:1–33, 1995.

[14] B. Fleischmann. *The vehicle routing problem with multiple use of the vehicles.* 1990.

[15] J. Gaschig. Performance measurement and analysis of certain search algorithms. Technical report, CARNEGIE-MELLON UNIV PITTSBURGH PA DEPT OF COMPUTER SCIENCE, 1979.

[16] I. P. Gent and T. Walsh. The sat phase transition. In *ECAI*, volume 94, pages 105–109. PITMAN, 1994.

[17] I. P. Gent and T. Walsh. The tsp phase transition. *Artificial Intelligence*, 88(1-2):349–358, 1996.

[18] F. Glover. Tabu search-part i. *ORSA Journal on computing*, 1(3):190–206, 1989.

[19] F. Glover. Tabu search-part ii. *ORSA Journal on computing*, 2(1):4–32, 1990.

[20] C. P. Gomes and T. Walsh. Randomness and structure. In Rossi et al. [46], pages 639–664.

[21] W. D. Harvey. *Nonsystematic backtracking search.* PhD thesis, stanford university, 1995.

[22] W. D. Harvey and M. L. Ginsberg. Limited discrepancy search. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence, IJCAI 95, Montréal Québec, Canada, August 20-25 1995, 2 Volumes*, pages 607–615. Morgan Kaufmann, 1995.

[23] G. Hiermann, J. Puchinger, S. Ropke, and R. F. Hartl. The electric fleet size and mix vehicle routing problem with time windows and recharging stations. *European Journal of Operational Research*, 252(3):995–1018, 2016.

[24] B. I. Intelligence. Mobile sales drive unexpected uk e-commerce growth, January 2017. [Online; posted 19-January-2017].

[25] S. Irnich and G. Desaulniers. Shortest path problems with resource constraints. *Column generation*, pages 33–65, 2005.

[26] B. Kallehauge, J. Larsen, and O. B. Madsen. Lagrangian duality applied to the vehicle routing problem with time windows. *Computers & Operations Research*, 33(5):1464–1487, 2006.

[27] P. Kilby. Constraint programming for the vehicle routing problem. In *CP2013: 19th International Conference on Principles and Practice of Constraint Programming*, 2013.

[28] P. Kilby. The vehicle routing problem with time windows. Data 61 CSIRO, 2017.

[29] P. Kilby and P. Shaw. Vehicle routing. In Rossi et al. [46], pages 801–836.

[30] P. Kilby and T. Urli. Fleet design optimisation from historical data using constraint programming and large neighbourhood search. *Constraints*, 21(1):2–21, 2016.

[31] E. Lam, P. V. Hentenryck, and P. Kilby. Joint vehicle and crew routing and scheduling. In G. Pesant, editor, *Principles and Practice of Constraint Programming - 21st International Conference, CP 2015, Cork, Ireland, August 31 - September 4, 2015, Proceedings*, volume 9255 of *Lecture Notes in Computer Science*, pages 654–670. Springer, 2015.

[32] S. Lin. Computer solutions of the traveling salesman problem. *The Bell system technical journal*, 44(10):2245–2269, 1965.

[33] Q. Lu and M. M. Dessouky. An exact algorithm for the multiple vehicle pickup and delivery problem. *Transportation Science*, 38(4):503–514, 2004.

[34] M. Luby, A. Sinclair, and D. Zuckerman. Optimal speedup of las vegas algorithms. In *Second Israel*

Symposium on Theory of Computing Systems, ISTCS 1993, Natanya, Israel, June 7-9, 1993, Proceedings, pages 128–133. IEEE Computer Society, 1993.

[35] G. B. Mathews. On the partition of numbers. *Proceedings of the London Mathematical Society*, 1(1):486–490, 1896.

[36] D. Naddef and G. Rinaldi. Branch and cut. *Vehicle Routing, SIAM*, 2000.

[37] N. Nethercote, P. J. Stuckey, R. Becket, S. Brand, G. J. Duck, and G. Tack. Minizinc: Towards a standard CP modelling language. In C. Bessiere, editor, *Principles and Practice of Constraint Programming - CP 2007, 13th International Conference, CP 2007, Providence, RI, USA, September 23-27, 2007, Proceedings*, volume 4741 of *Lecture Notes in Computer Science*, pages 529–543. Springer, 2007.

[38] I. Or. *Travelling Salesman-Type Combinatorial Problems and their Relation to the Logistics of Blood-Banking.* phdthesis, Department of Industrial Engineering and Management Sciences, Northwest University, Evanston, IL, 1976.

[39] M. Padberg and G. Rinaldi. A branch-and-cut algorithm for the resolution of large-scale symmetric traveling salesman problems. *SIAM review*, 33(1):60–100, 1991.

[40] H. Pollaris, K. Braekers, A. Caris, G. K. Janssens, and S. Limbourg. Iterated local search for the capacitated vehicle routing problem with sequence-based pallet loading and axle weight constraints. *Networks*, 69(3):304–316, 2017.

[41] P. Prosser. An empirical study of phase transitions in binary constraint satisfaction problems. *Artificial Intelligence*, 81(1-2):81–109, 1996.

[42] C. Prud'homme, J.-G. Fages, and X. Lorca. *Choco Documentation.* TASC,LS2N, CNRS UMR 6241 and COSLING S.A.S., 2017.

[43] P. Ramachandran. Vehicle routing problem with load compatibility constraints. In *Industrial Engineering and Engineering Management, 2009. IEEM 2009. IEEE International Conference on*, pages 339–343. IEEE, 2009.

[44] S. Ropke and D. Pisinger. An adaptive large neighborhood search heuristic for the pickup and delivery problem with time windows. *Transportation Science*, 40(4):455–472, 2006.

[45] S. Ropke and D. Pisinger. A unified heuristic for a large class of vehicle routing problems with backhauls. *European Journal of Operational Research*, 171(3):750–775, 2006.

[46] F. Rossi, P. van Beek, and T. Walsh, editors. *Handbook of Constraint Programming*, volume 2 of *Foundations of Artificial Intelligence*. Elsevier, 2006.

[47] M. W. P. Savelsbergh. The vehicle routing problem with time windows: Minimizing route duration. *INFORMS Journal on Computing*, 4(2):146–154, 1992.

[48] M. Schneider, A. Stenger, and D. Goeke. The electric vehicle-routing problem with time windows and recharging stations. *Transportation Science*, 48(4):500–520, 2014.

[49] J. Schuijbroek, R. C. Hampshire, and W. van Hoeve. Inventory rebalancing and vehicle routing in bike sharing systems. *European Journal of Operational Research*, 257(3):992–1004, 2017.

[50] P. Shaw. Using constraint programming and local search methods to solve vehicle routing problems. In M. J. Maher and J. Puget, editors, *Principles and Practice of Constraint Programming - CP98, 4th International Conference, Pisa, Italy, October 26-30, 1998, Proceedings*, volume 1520 of *Lecture Notes in Computer Science*, pages 417–431. Springer, 1998.

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

[52] M. M. Solomon and J. Desrosiers. Survey paper-time window constrained routing and scheduling problems. *Transportation science*, 22(1):1–13, 1988.

[53] G. Tack. *Constraint Propagation - Models, Techniques, Implementation.* phdthesis, Saarland University, Germany, 2009.

[54] O. Toro, M. Eliana, Z. Escobar, H. Antonio, E. Granada, et al. Literature review on the vehicle routing problem in the green transportation context. *Luna Azul*, (42):362–387, 2016.

[55] B. Turan, S. Minner, and R. F. Hartl. A VNS approach to multi-location inventory redistribution with vehicle routing. *Computers & OR*, 78:526–536, 2017.

[56] T. Urli and P. Kilby. Constraint-based fleet design optimisation for multi-compartment split-delivery rich vehicle routing. In J. C. Beck, editor, *Principles and Practice of Constraint Programming - 23rd International Conference, CP 2017, Melbourne, VIC, Australia, August 28 - September 1, 2017, Proceedings*, volume 10416 of *Lecture Notes in Computer Science*, pages 414–430. Springer, 2017.

[57] T. Vidal, T. G. Crainic, M. Gendreau, and C. Prins. A hybrid genetic algorithm with adaptive diversity management for a large class of vehicle routing problems with time-windows. *Computers & OR*, 40(1):475–489, 2013.

[58] C. Voudouris and E. Tsang. Partial constraint satisfaction problems and guided local search. *Proc., Practical Application of Constraint Technology (PACT'96), London*, pages 337–356, 1996.

[59] L. Zhu, J. Holden, E. Wood, and J. Gender. Green routing fuel saving opportunity assessment: A case study using large-scale real-world travel data. In *IEEE Intelligent Vehicles Symposium, IV 2017, Los Angeles, CA, USA, June 11-14, 2017*, pages 1242–1248. IEEE, 2017.