Reduce, Reuse, Recycle: On exploration of solution reuse in VRPTW

Version of August 18, 2023

Gautham Venkataraman

Reduce, Reuse, Recycle: On exploration of solution reuse in VRPTW

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Gautham Venkataraman
born in Chennai, India

Algorithmics Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.tudelft.nl/eemcs

Reduce, Reuse, Recycle: On exploration of solution reuse in VRPTW

Author: Gautham Venkataraman
Student id: 5334152
Email: g.venkataraman@student.tudelft.nl

Abstract

Solving routing problems efficiently is instrumental in minimizing operational costs in logistics. These routing problems are hard to solve and often take a lot of time to find a good solution. In this thesis, we present a methodology that tackles the challenge of efficiently solving recurring instances of the Vehicle Routing Problem with Time Windows by recycling solutions. By making more problem-specific assumptions, we introduce a solution recycling approach that can leverage shared solution structures across similar instances. This accelerates the solution-finding process. We implement our methodology in the framework of constraint programming and show that such a methodology is actually useful and is a concept that is yet unexplored in combinatorial optimization.

Thesis Committee:

Daily supervisor:	Dr. S. Dumančić, Faculty EEMCS, TU Delft
Responsible Professor:	Dr. N. Yorke-Smith, Faculty EEMCS, TU Delft
Committee Member:	Dr. H. Caeser, Faculty EEMCS, TU Delft

Preface

This thesis is the culmination of about 8 months of work. Work that would not have been possible without key figures in my life.

I had the pleasure of working with two amazing supervisors who were instrumental in the success of this thesis. I would like to thank my responsible professor, Neil, who pretty much introduced me to the field of combinatorial optimization and inspired me to take up research in this field. Furthermore, I would also like to express my gratitude to my daily supervisor, Sebastijan, for his unending support whenever I walked into his office anxious about the thesis.

I am also very thankful to the people close to me, my friends, including the ones back home, whose help cannot be overstated. I cannot express in words how much of a positive impact these people had on my life and, by extension, this thesis.

My final token of gratitude goes to my family, my *Amma* and *Appa*, for their unending trust and support, emotionally and financially, from across the world.

Gautham Venkataraman
Delft, The Netherlands
August 18, 2023

Contents

List of Figures

Chapter 1

Introduction

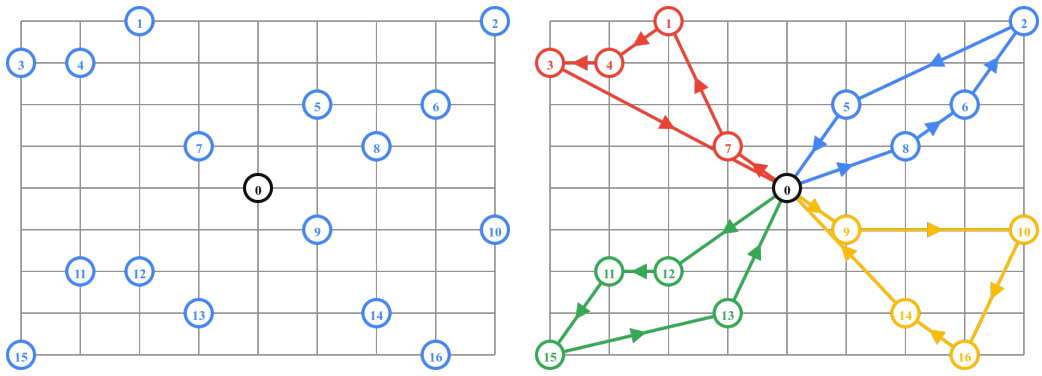
The Last Mile delivery problem involves the delivery of physical goods by a delivery company from their depots to the consumers' homes. In fact, often the most challenging and expensive part of the delivery process is the last mile delivery, often accounting for about 53% of the total cost to move goods [1]. A significant part of these costs can be directly attributed to expenses spent on fuel. It is important for such companies to route their vehicles more efficiently to save money against rising fuel costs.

The abstraction of such routing problems is the Vehicle Routing Problem (VRP). The VRP is a class of combinatorial optimization (CO) problems in the field of operations research that asks "What is the optimal set of routes for a given set of vehicles that need to visit a set of locations?". Readers familiar with the famous Travelling Salesman Problem (TSP) in computer science, can quickly draw a parallel to the VRP as it is a generalization of the TSP with multiple salesmen. An example instance of VRP and its solution can be seen in Figure 1.1. The challenging part of solving the VRP comes from how the problem scales as we deal with bigger instances.

Consider the simplest case of the VRP with one vehicle, where we have to find the shortest path starting from an initial point, visit several cities and return to the initial point. This is essentially the Travelling Salesman Problem. If the number of cities to visit is small, say 5, there are only 60 solutions we need to evaluate to find the optimal solution[1]. However, if we have to visit 15 cities, the number of solutions we need to evaluate quickly blows up to 653837184000 possible combinations. In reality, we often deal with hundreds, if not thousands, of cities that need to be visited. Using a naive brute force algorithm, small toy instances can be solved very quickly (in the order of milliseconds), but realistic instances can take a long time (in the order of thousands of years) with the most powerful of computers to find an optimal solution.

Obviously, we do not have the luxury of spending that much time, and we need smarter and more efficient ways to solve the VRP. This is without considering that in a real-world use case, we often have to deal with other restrictions, such as the load limit of every vehicle or customer preferences. The nature and frequency at which we encounter these routing problems have led to numerous developments in the field of VRP and combinatorial op-

[1] Number of solutions for the TSP with N locations = $\frac{(N-1)!}{2}$



(a) VRP instance (b) VRP solution

Figure 1.1: A VRP instance and its solution with 1 depot (node 0), 16 locations to visit using 4 vehicles.

timization itself. The methods to solve the VRP range from exact methods, which focus on finding optimal solutions, to heuristical and metaheuristical methods, which focus on finding good feasible solutions quickly but do not guarantee optimality.

The existing approaches to solving the VRP focus on solving each instance independently of other instances, i.e., information from solving one instance is not used in any capacity to solve other instances. It is inherently assumed that each VRP instance is its own problem, and the solution for one instance does not provide any information to solve another instance. This assumption is reflected in how many famous VRP benchmarks are randomly generated [2] [3].

However, in reality, VRP instances are not completely random and often exhibit certain patterns of similarity. These similarities can arise from recurring customer locations, time constraints, spatio-temporal correlations, and other factors attributed to seasonal and geographical changes.

For example, consider the case of a small logistical company that has to repeatedly solve routing problems every day as part of their operations. Our goal is to develop a methodology that can efficiently solve these types of recurring VRP instances by reusing and recycling solutions from solved instances. We posit that such recurring VRP instances have a pattern of similarity that is reflected in the solution space as well. Therefore, we may use solutions from a solved instance, which can provide valuable information when solving another instance.

In pursuit of such a methodology, we encounter some problems, the first of which is: *how do we define similarity between VRP instances?* There exist similarity metrics like the Jaccard Index, but it is not obvious how to apply such metrics in this context since we also need to take into account problem-specific information like the geographical layouts of customer locations.

Furthermore, recurring instances might have a pattern of similarity, but they are fundamentally not the same problem. So *what kind of information can we recycle from a solved instance to solve a newer instance?* This involves identifying a solved instance that is simi-

lar to the current instance we are trying to solve and extracting useful insights that can guide the search process.

Finally, not all solved instances might provide valuable insights. If the goal is to reuse and recycle solved instances to provide valuable information, *can we identify and store only the most valuable instances that provide good information?*

We attempt to answer these questions by developing a Solution Recycling Methodology for a variant of the VRP, the Vehicle Routing Problem with Time-Windows (VRPTW). We chose the VRPTW because it strikes a fine balance between real-world requirements and academic experimentation. Furthermore, we develop our methodology in the framework of Constraint Programming (CP), a paradigm to solve optimization problems such as the VRP(TW) by defining a model of the problem as a set of constraints that need to be satisfied and using specialized constraint solvers to solve the model. We would like to point out that our methodology is not limited to just the VRPTW or CP and can be expanded to other variants and optimization paradigms as well.

The rest of the thesis is organized as follows: In Chapter 2, we lay more context by discussing the related works in the field and the state-of-the-art methods to solve the VRP and its variants. This is followed by Chapter 3, where we formulate the VRPTW problem and discuss in more detail the nature of the problems we consider. We introduce our methodology in Chapter 4 and evaluate it in Chapter 5. Finally, we conclude the thesis by talking about the limitations of methodology and how future work might address those issues in Chapter 6.

Chapter 2

Related Work

The Capacitated VRP (CVRP) was first introduced by Dantzig and Ramser [4] as the "Truck Dispatching Problem" in 1959, with the goal of finding optimal routes for a fleet of gasoline delivery trucks between a terminal and the service stations supplied by the terminal. The goal was to find routes for the fleet of vehicles such that each service station is visited and is supplied the required amount of gasoline, while minimizing the total travel distance by all the vehicles.

In their work, Dantzig and Ramser provided a Linear Programming[1] (LP) formulation to solve the truck dispatching problem. The Simplex Method to solve LPs was already discovered by Dantzig, which, despite its exponential worst-case complexity, is rather efficient in practice. Despite that, the integral constraints in the original formulation by Dantzig and Ramser meant that this formulation could not be solved, with the authors remarking "At the present time there exists no general application method for solving discrete variable problems". Following this, a number of heuristical approaches to solving the CVRP were explored.

One of the most famous heuristical algorithms to solve the VRP was the Savings Algorithm, introduced by Clarke and Wright [5] in 1964. The Savings Algorithm defines the savings for a pair of nodes as the reduction in cost that is achieved when two nodes are visited in a single route. The algorithm then iterates through the decreasingly sorted pairs of nodes and attempts to merge routes that contain the pair of nodes. Despite being a greedy approach, the Savings Algorithm is able to produce good feasible solutions very quickly.

The research into exact algorithms to solve the VRP starting gaining traction after the work by Christofides et al. [6], who proposed a Mixed Integer Programming[2] (MIP) formulation that made use of tighter lower bounds on the objective of the VRP. By this point, MIP models could be solved by Branch and Bound methods, which can find optimal integer solutions. Branch and Bound methods work by recursively splitting the MIP into sub-problems and effectively pruning the search space using the upper and lower bounds of the sub-problems. The MIP formulation by Christofides et al. [6] makes use of the fact that the objective value of a solution to the TSP is a lower bound on the objective value of the

[1] Programming here refers to the assignment of values to variables

[2] Mixed Integer Programming is Linear Programming where some variables are constrained to be integers.

VRP.

Later in the 1980s, G. Laporte and Y. Nobert [7] [8] [9] use branch and bound techniques augmented with the cutting planes approach (by Gomory [10]) to exactly solve bigger instances of the VRP. Since then, the best known approaches to exactly solving the VRP are based on tighter bounds, with the best known exact implementations being by Fukasawa et al. [11] and Baldacci et al. [12], which can solve instances of size up to 135 nodes optimally. However, even the best exact solution methods do not scale well with the size of the problem.

This led to the rise of metaheuristics in combinatorial optimization. Particularly local search methods were being developed. Local search methods start with an initial solution and attempt to search the neighborhood of the solution for other solutions. For example, Croes [13] proposed the 2-opt heuristic for the TSP, which swaps two nodes in a route. These types of simple transformations allow us to search the immediate neighborhood of a solution. The 2-opt swap creates another valid solution, but not necessarily a better one. These types of transformations are usually coupled with hill-climbing strategies, where repeated transformations on an initial solution are performed and only solutions with better objectives are accepted.

This, however, has the downside of being stuck in a local optima or a plateau in the search space. To combat this, Tabu Search [14] [15] [16] was introduced, which allows the search procedure to accept solutions with worse objectives and discourages coming back to the previously visited neighborhoods by remembering the solutions in a memory table. This was successfully implemented to solve the VRP by Gendreau et al. [17].

Another powerful approach to escaping local optima is Simulated Annealing (SA). Simulated Annealing is a probabilistic technique that uses a global temperature parameter to guide exploration in the search space. Parallels can be drawn to the Epsilon-Greedy algorithm in Reinforcement Learning (RL) literature. It has been successfully used to solve the VRP in the seminal work of Kirkpatrick et al. [18].

The above-mentioned metaheuristics have one common drawback: they can only search for solutions in the neighborhood of the current solution. Variable Neighborhood Search (VNS) was introduced by Mladenović and Hansen [19] to address this issue. VNS works by finding a local optima in a neighborhood and then moves to a closer neighborhood by transforming the solution.

Large Neighborhood Search (LNS) [20] is another method that explores different neighborhoods in the solution space by repeatedly destroying and repairing parts of the solution. This allows the algorithm to jump to a completely different neighborhood in the search space. Its extension, the Adaptive Large Neighborhood Search (ALNS), was used by Ropke and Pisinger [21] [22] to solve different variants of the VRP.

Another powerful class of metaheuristics are the so-called population-based metaheuristics. A common theme among this class of metaheuristics is that instead of only considering a single solution, they maintain a population of feasible solutions. At each iteration, a new population is evolved from the current population, and a set of good solutions from both populations is selected and allowed to survive to the next iteration. Methods such as Evolutionary Algorithms (EA) or particle swarm optimization are classic examples.

Hybrid Genetic Search (HGS), introduced by Vidal et al. [23] [24] is a state-of-the-art

method to solve common variants of the VRP. HGS combines ideas from two subfields of metaheuristics: evolutionary algorithms and local search methods.

Algorithm 1 HGS-CVRP

```
Initialize population with random solutions improved by local search;
while number of iterations without improvement < It_NI and time < T_MAX do
    Select parent solutions P_1 and P_2;
    Apply the crossover operator on P_1 and P_2 to generate an offspring C;
    Educate offspring C by local approach;
    Insert C into respective subpopulation;
    if C is infeasible then
        With 50% probability, repair C (local search) and insert it into respective subpopulation;
    end if
    if maximum subpopulation size reached then
        Select survivors;
    end if
    Adjust penalty coefficients for infeasibility;
end while
Return best feasible solution;
```

Algorithm 1 shows the overall HGS algorithm for the CVRP. It can be seen that the algorithm repeatedly applies evolutionary strategies to maintain a good set of candidate solutions and local search heuristics to improve candidate solutions. The authors also introduce problem-specific heuristics to search different neighborhoods in the solution space. The HGS has been able to solve instances of large sizes (n = 1000) and outperform other competing methods, even contributing to finding new best-known solutions for benchmark instances.

The HGS has also been successfully modified to solve the VRPTW problem with additional heuristics by Kool et al. [25]. Unfortunately, we could not find an open-source implementation to compare our method against. While we do not expect our method to outperform state-of-the-art methods, it would be interesting to see how the quality of the solution compares between methods.

De Smet and Wauters [26] take a more practical route and use multithreading to speed up search in metaheuristic algorithms. They note that the solution evaluation phase in local search methods can be parallelized and present an incremental method to evaluate different solutions. They implement this in the Java-based OptaPlanner constraint solver [27].

We had already briefly mentioned MIP formulations for solving optimization problems. Another exciting paradigm to solve combinatorial optimization problems is Constraint Programming (CP). Specifically, optimization models for CP can be defined similarly to those of MIP formulations. An optimization problem takes the following form in CP:

- A set of **decision variables** which represents the variables we want to find assignments for.

- A set of **domains** on the decision variables which represent the range of values the variables can take.
- A set of **constraints** on decision variables which together represent the set of valid variable assignments for the optimization problem.
- An **objective function** which represents the quantity we want to optimize the decision variables for.

Constraint solvers also effectively search the solution space by splitting the main problem into sub-problems, albeit differently than MIP solvers. MIP solvers guide search by solving the linear relaxation of the problem and pruning infeasible solutions by calculating bounds on the sub-problem. In contrast, constraint solvers perform logical inferences to reduce the domain of variables, effectively pruning the search space.

Another stark contrast is in how these methods prove optimality. MIP solvers prove optimality by calculating the lower bound and mathematically showing that no better solution exists, whereas constraint solvers prove optimality by showing that no better solution than the current one exists, effectively performing an enumerative search through the solution space.

Finally, MIP requires that the optimization problem be linear in nature (linear constraints and an objective function). CP, on the other hand, does not have any of these restrictions and can model any constraint or objective function.

A major reason for using CP is its flexibility in defining certain constraints. For example, the AllDifferent() constraint enforces that all the decision variables in a set of variables be distinct and has special propagation techniques to remove infeasible values in variable domains. This allows us to move away from mathematically defining constraints and instead logically defining them.

Both CP and MIP are widely used across industry and academia, and commercial and open-source implementations exist for both. A lot of the heuristic and metaheuristic approaches described in the previous section are used in conjunction with CP to solve optimization problems. CP with local search heuristics has also been successfully applied to solve the VRP and its variants [20] [28] [29] [30].

Chapter 3

Problem Setting

The goal of this chapter is to accurately describe the problem we focus on. The first part of the chapter will focus on defining the VRPTW and introducing the dataset we use, which is important in establishing the practical context we consider. The second part of the chapter will focus on the problems that emerge as a consequence of trying to develop a solution-recycling methodology.

3.1 The Vehicle Routing Problem with Time-Windows

The VRPTW can be formulated as a graph $G = (N, E)$, where

- $N = \{n_0, n_1, \cdots, n_q\}$ are the nodes of the graph with n_0 representing the depot and $n_1, n_2, \cdots, n_q$ representing the customer locations.
- $E = \{(n_i, n_j) : n_i, n_j \in N\}$ is the set of edges that represents a link between any two nodes n_i and n_j with travel time $t_{i,j}$.
- $V = \{v_1, \cdots, v_k\}$ is the set of homogenous[1] vehicles.
- $W = \{(s_1, e_1), (s_2, e_2), \cdots, (s_n, e_n)\}$ is the set of time windows associated with each node.

Each customer node should be visited, or we incur a huge penalty if a node is not visited. The goal is to find a set of routes for each vehicle that start and end at the depot while minimizing the total travel time of all the vehicles and the incurred penalties.

This formulation might be slightly different from other VRPTW formulations where all the customer nodes have to be visited. We loosen this restriction since, in reality, the working hours are limited and it may not be possible for all the locations to be visited, which might render the problem infeasible.

Furthermore, this formulation can also be considered a multiobjective optimization problem, where we would like to minimize the total travel time while maximizing the number of customers visited. The introduction of the penalty term allows us to directly optimize

[1] Homogenous refers to all vehicles having the same speed.

the linear sum of these two quantities. We suspect our methodology can also be applied to the multiobjective case, but we consider the simpler case for experimentation.

We also point out that in this case, minimizing the total travel time is equivalent to minimizing the total distance traveled since travel time is directly proportional to distance.

3.1.1 The Belgian Dataset

We use the Belgian road-km/road-time/air data used in De Smet and Wauters [26], specifically the `belgium-tw-d10-n2750-k55.vrp`[2] instance. We use this dataset because the customer locations correspond to real locations on the world. This is ideal for our research as it allows us to simulate real-world operations.

The instance describes a multi-depot routing problem with 10 depots, 2750 orders, and 55 vehicles. The depots are assumed to operate for 12 hours, from 07:00 to 19:00. Each order is serviced by visiting the location of the order within the time window specified by the customer.

For the sake of simplicity, we divide the operating hours into 24 discrete half-hours, 0–23, and convert the locations from latitude-longitude to a simple x-y coordinate system[3]. We do not directly work with this instance but instead use it as an underlying distribution to randomly sample smaller VRPTW instances with a single depot.

We limit the size of instances we consider to around 100 locations or orders, 10 vehicles with 50 km/hr speed on average. While randomness is useful for simulation, it is not a desirable quality for experimentation and reproducibility. Therefore, we generate pseudo-random VRPTW instances by using a seed value to control randomness and an approximate number of orders to control the size of the instances.

Formally, we define a VRPTW instance I as a 5-tuple $I = (seed, O, d, k, speed)$, where:

- $seed$ is a random integer associated with the instance.
- $O = \{o_1, o_2, \cdots, o_q\}$ is the set of orders;
  - $q = |O|$ is the number of orders in the instance.
  - $o_i.x \in [0, 265)$ is the x-coordinate of the order location.
  - $o_i.y \in [0, 216)$ is the y-coordinate of the order location.
  - $o_i.s \in [0, 24)$ is the time window start of the order.
  - $o_i.e \in [0, 24)$ is the time window end of the order.
- d is the depot location.
- k is the number of vehicles.
- $speed$ is the travel speed of all the vehicles, assuming all vehicles have the same speed.

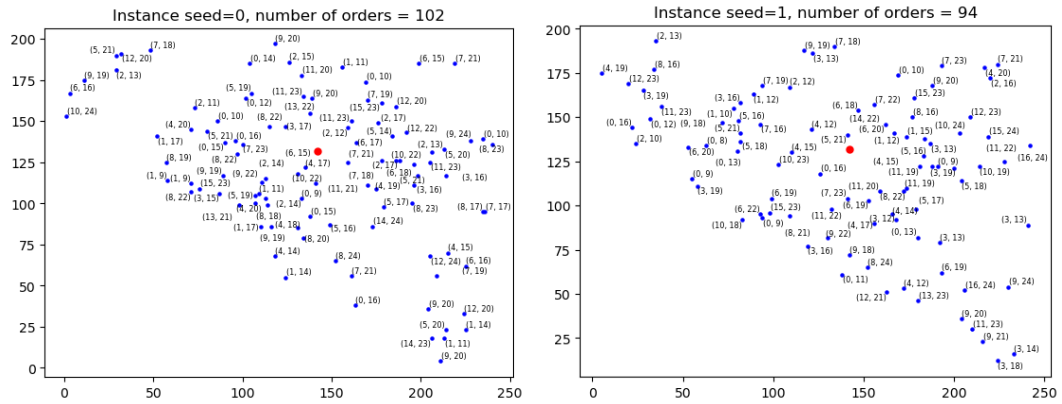


Figure 3.1: Sampled instances showing the location of the orders with their time windows.

If we fix $d, k, speed$, and by only changing the *seed*, we generate semi-realistic instances of the routing problems faced by a small-scale logistic company on a daily basis. We justify this assumption by pointing out that in reality, the depot, number of vehicles, and vehicles themselves do not change frequently, and the main difference is the different orders that have to be serviced every day. It is unrealistic to assume that all vehicles have the same speed. This is done purely for ease of problem formulation, and we note that our methodology can be extended to the heterogeneous vehicles case as well.

Furthermore, our sampled instances still reflect the underlying spatio-temporal correlations present in the original instance (`belgium-tw-d10-n2750-k55.vrp`). This is ideal as it is a reasonable simulation of real-world operations. Figure 3.1 visually represents two example instances.

3.2 Problem – Solving Recurring Instances

The overarching goal of this thesis is to effectively solve recurring VRPTW instances in a practical context. A core assumption that is made in pursuit of this method is that similarity in instances corresponds to similarity in their solution space. The idea is that if we have solved a similar instance before, we can use that experience to guide the solution process for a new instance with similar characteristics.

For example, an observation could be that a specific vehicle route is always used to service orders in a specific geographical region. If we have a new instance with orders from that geographical region, instead of solving the whole problem, we can leverage our experience by recycling the routes we have used before as a partial solution. This can massively reduce the search space while searching for a full solution.

However, this only works when we can identify some similarity between the instances, as we cannot simply use any instance to create partial solutions.

[2]The dataset is available at http://www.vrp-rep.org/datasets/item/2017-0001.html [31]

[3]This is done using PyProj [32], a coordinate transformations library for Python

3.2.1 Problem P1 – Similarity between instances

The first of our problems is to identify similar instances, which requires defining similarity in this context. There is literature on similarity in the context of VRP. For example, it is quite common to define two solutions as similar if their objective values are closer to each other. Other definitions also include the number of common arcs between two solutions. However, we are mainly interested in identifying similarity between instances.

We have loosely used the term similarity to describe instances that have a pattern. Since all the orders are being sampled from an underlying instance, every instance can be considered similar to each other. This is not particularly useful, so we turn to a more rigorous definition of similarity.

If we consider instances to only differ by their set of orders, we already have well-defined metrics, such as the Jaccard Index, to measure similarity. The Jaccard Index for two instances I_1 and I_2 can be defined in the following way:

$$J(I_1, I_2) = \frac{|I_1.O \cap I_2.O|}{|I_1.O \cup I_2.O|} = \frac{|I_1.O \cap I_2.O|}{|I_1.O| + |I_2.O| - |I_1.O \cap I_2.O|}$$

If we already have similarity metrics, what is the issue? The problem lies in the fact how the intersection between two sets ($|I_1.O \cup I_2.O|$) is calculated. In our case, the intersection roughly translates to the number of common orders between the two instances. Since our instances are randomly generated, it is very unlikely that any two instances will have orders in common. While not impossible, the number of common orders between two instances will likely be zero. This, again, is not very useful. So the first problem we encounter is:

How can we define a better similarity metric between instances?

3.2.2 Problem P2 - Recycling a Solution

Assuming that we have solved a similar instance previously, how can we make use of that solution to efficiently solve a newer instance?

One way to achieve this is to construct a feasible solution for the instance we are solving by using information from the solved instance. Constructing an initial solution (typically with a poor objective) and then using other algorithms to search for better solutions is a staple of solving combinatorial optimization problems. The main advantage an initial solution provides is an upper bound on the objective. Therefore, with an initial solution provided as a hint, a constraint solver can strictly search for better solutions than the initial solution, which can potentially reduce the search space. With this information in mind, the problem we face is:

How can we construct an initial solution from a solved instance?

3.2.3 Problem P3 - Limited Cache

Assuming that we have solutions to the above problems, we can easily implement a method that stores solved instances in a cache. Every time we solve a new instance, we can quickly

look for a similar instance in the cache and recycle its solution. The problem rears its head when our cache gets very large from solving a lot of instances. The first problem is that looking through the cache for similar instances presents a small but significant performance overhead that can add up. Secondly, the cache might also get polluted with very random or niche instances that are not necessarily similar to the other instances and therefore have no reason to be present in our cache. The problem we face is:

How can we keep a limited cache of solved instances that are useful?

Chapter 4

Methodology

In this chapter, we provide our approaches to solving the problems introduced in the previous chapter. We start by introducing a constraint model for the VRPTW, which will serve as the foundation on which we develop the rest of our methodology. Following that, we address each of the problems and present our overall methodology.

4.1 Constraint Optimization Model for the VRPTW

Defining the VRPTW in a CP framework requires reimagining the problem as a set of decision variables that represent the solution and constraints on these decision variables that represent the feasibility of the solution. The CP solver will take this model and try to find valid assignments for our decision variables that satisfy our constraints.

4.1.1 Definitions

- A node corresponds to a location on the map. We work with the node indices set $N = \{n_0\} \cup N_O \cup N_S \cup N_E$ where:
  - $n_0 = 0$ is the null node, which is the starting and ending point of the route. This corresponds to the depot in our case, but it is better to abstract this away if vehicles have different starting points (for example, in the MDVRP case). All the routes start and end at the null-node; therefore, it is the only node that can have multiple incoming and outgoing arcs, whereas all the other nodes can only have one incoming and outgoing arc.
  - $N_O = \{1, 2, \cdots, q\}$ are the order-nodes corresponding to the indices of the orders.
  - $N_S = \{1+q, \cdots, 1+q+\bar{k}\}$ are the vehicle-start-nodes.
  - $N_E = \{1+q+\bar{k}, \cdots, 1+q+2\bar{k}\}$ are the vehicle-end-nodes[1].
  - $\bar{n} = |N| \equiv 1+q+2\bar{k}$ denotes the total number of nodes.

[1] Both vehicle-start-nodes and vehicle-end-nodes correspond to the depot in this case.

Each node is identified by its node index.

- The vehicle indices set is represented by $\{v_f\} \cup \{1, 2, \cdots k\}$, where $v_f = 0$ represents the fake-vehicle to hold all the unassigned orders, i.e., customer orders that will not be serviced. Therefore, the total number of vehicles (including the fake vehicle) is denoted by $\bar{k}$.

- The distance-matrix, $d_{\bar{n}x\bar{n}}$, contains the L1 distance between each node and every other node. The distance from node i to j is given by the entry $d_{i,j} \equiv d_{j,i}$. The distance from/to the null-node to/from every other node is 0, i.e., $d_{0,i} = d_{i,0} = 0, \forall i \in N - \{n_0\}$.

- Similarly, the time-matrix, $t_{\bar{n}x\bar{n}} = \frac{d_{\bar{n}x\bar{n}}}{speed}$, contains the travel time between each node and every other node.

4.1.2 Decision Variables

- $S_i \in [0, \bar{n} - 1] \quad \forall i \in N$, is variable that represents the successor of any node i. Each node can have only one successor, except the null-node (n_0), whose successors are the vehicle-start-nodes (N_E). Every vehicle starts its journey from the respective vehicle start node and then travels along the connected successors until it reaches its respective vehicle end node.

- $P_i \in [0, \bar{n} - 1] \quad \forall i \in N$, is variable that represents the predecessor of any node i. Each node can have only one predecessor, except the null-node (n_0), whose predecessors are the vehicle-end-nodes (N_E).

- $X_{i,j} = \begin{cases} 1 & \text{if } S_i = j \text{ and } P_j = i \\ 0 & \text{otherwise} \end{cases} \quad \forall i, j \in N$, is a boolean variable that represents whether a travel arc exists between two nodes i and j. It is a redundant variable to the successor and predecessor variables, this redundancy allows us to define some constraints much simpler.

- $V_i \in [0, \bar{k} - 1] \quad \forall i \in N$, is the index of the vehicle serving any node i. The vehicles' start nodes (N_S) and end nodes (N_E) are served by themselves.

- $U_i = \begin{cases} 1 & \text{if node } i \text{ is unassigned} \\ 0 & \text{otherwise} \end{cases} \quad \forall i \in N$, is a boolean variable that represents if a node is unassigned.

- $A_i \in [0, 23] \quad \forall i \in N$, represents the arrival time at any node i. As a reminder, we consider the 12-hour shift as 24 half hours. By restricting the arrival time to the domain $[0, 23]$, we implicitly enforce the constraint that the arrival time at all nodes (including the depot) should be within the 12-hour shift.

- $T_i = \begin{cases} 0 & \text{if node } i \text{ is unassigned} \\ t_{P_i,i} & \text{otherwise} \end{cases} \in [0, max(t_{\bar{n},\bar{n}})] \quad \forall i \in N$, represents the travel time from predecessor of node i to node i only if node i is assigned. The travel time (if assigned) is given by the L1 distance between the two nodes.

4.1.3 Constraints

- The successor of vehicle end nodes and the predecessor of vehicle start nodes, respectively, is the null node.

$$\begin{aligned} S_i &= 0 \qquad \forall i \in N_E \\ P_i &= 0 \qquad \forall i \in N_S \end{aligned} \tag{4.1}$$

- The successor or predecessor of a node cannot be itself.

$$\begin{aligned} S_i &\neq i \\ P_i &\neq i \qquad \forall i \in N - \{n_0\} \\ X_{i,i} &\neq 1 \end{aligned} \tag{4.2}$$

- The channeling constraints between successors / predecessors and x.

$$\begin{aligned} (S_i = j) &\iff X_{i,j} \qquad \forall i \in N_S + N_O, \forall j \in N - \{n_0\} \\ (P_i = j) &\iff X_{j,i} \qquad \forall i \in N_O + N_E, \forall j \in N - \{n_0\} \end{aligned} \tag{4.3}$$

- The vehicle serving its start node and end node must be the vehicle itself.

$$\begin{aligned} V_i &= i - q - 1 \qquad \forall i \in N_S \\ V_i &= i - q - \bar{k} - 1 \qquad \forall i \in N_E \end{aligned} \tag{4.4}$$

- The vehicle serving a node must be the same vehicle serving the successor or predecessor of the node.

$$\begin{aligned} V_i &= V_{S_i} \qquad \forall i \in N_S + N_O \\ V_i &= V_{P_i} \qquad \forall i \in N_O + N_E \end{aligned} \tag{4.5}$$

- If an order is unassigned, it must be served by the fake vehicle.

$$(V_i = 0) \iff U_i \qquad \forall i \in N_O \tag{4.6}$$

- Time from the predecessor is derived from the time matrix and only calculated for the assigned nodes.

$$T_i = \begin{cases} 0 & \iff U_i \\ t_{P_i,i} & \iff \neg U_i \end{cases} \qquad \forall i \in N - \{n_0\} \tag{4.7}$$

- Arrival time of all vehicles at their start node should be 0.

$$A_i = 0 \qquad \forall i \in N_S \tag{4.8}$$

- The arrival time at a node is the arrival time at the predecessor node plus the time taken to travel from the predecessor.

$$A_i = A_{P_i} + T_i \qquad \forall i \in N - \{n_0\} \tag{4.9}$$

- The arrival time at a node should be within its time-window.

$$o_i.s \leq A_i < o_i.e \qquad \forall i \in N - \{n_0\} \tag{4.10}$$

- The final constraint is the AddMultipleCircuit which takes as input a set of arcs and corresponding binary variables to prevent subtours from forming, i.e., routes that do not start or end at vehicle start or end nodes, respectively.

$$AddMultipleCircuit\{(i, j, x_{i,j}); \quad \forall i, j \in N; \quad i = j \neq n_0\} \tag{4.11}$$

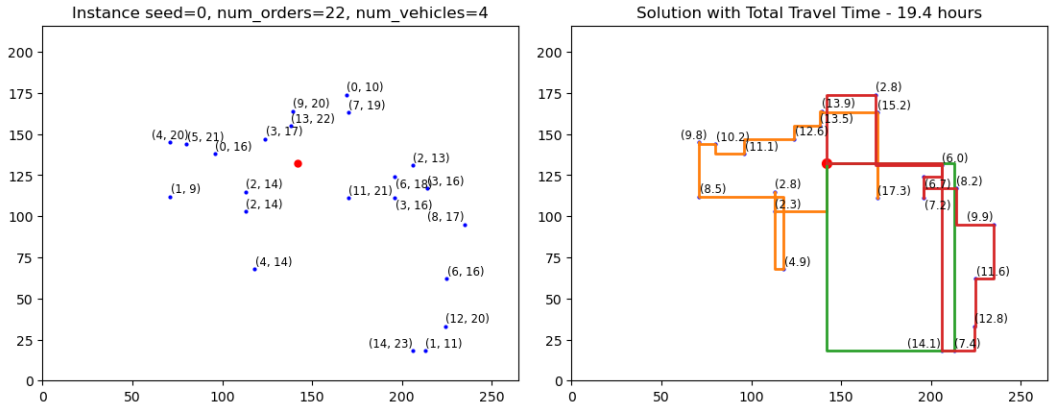
4.1.4 Objective Function

$$\text{Minimize} \qquad \sum_{i=0}^{\hat{n}} T_i + \sum_{i=1}^{q} U_i * \sum_{i=0}^{\hat{n}} \sum_{j=0}^{\hat{n}} t_{i,j} \tag{4.12}$$

The first term in the objective function represents the total travel time of all the vehicles. The second term represents the total penalty for unassigned orders, with the penalty for each unassigned order being very high. In this case, the penalty is the total sum of the time-matrix.

4.1.5 Example – Solving a small VRPTW instance

Putting together everything we discussed in this section, we solve a small VRPTW instance sampled from the Belgian dataset to optimality using our CP formulation.



(a) VRPTW instance (b) VRPTW solution

Figure 4.1: An example VRPTW instance and its solution.

Figure 4.1 shows an instance with 22 orders and 4 vehicles whose speeds are 50 km/hr. The instance is visually represented in Figure 4.1a with their corresponding time windows. Figure 4.1b shows an optimal solution to the instance with a total travel time of 19.4 hours (970 km) and no unassigned penalty. It is interesting to note that even though we have 4 vehicles, the optimal solution only uses 3 vehicles to complete all the deliveries.

4.2 A heuristic for Similarity

Defining similarity for instances starts with the orders that make up an instance. Intuitively, we can consider two orders to be similar if they are close to each other spatially and temporally. Two orders are close spatially if the distance between them is less than a threshold. Two orders are close temporally if their time window starts are less than a threshold[2].

We formalize this by defining two thresholds: the D_{THRES} and T_{THRES}, which represent the distance threshold and the time threshold, respectively. With these thresholds, we can define a binary notion of similarity, S_{order}, between two orders o_1 and o_2;

$$\begin{aligned} S_{order}(o_1, o_2, D_{THRES}, T_{THRES}) &= (|o_1.x - o_2.x| + |o_1.y - o_2.y| \leq D_{\text{THRES}}) \\ &\quad \wedge (|o_1.s - o_2.s| \leq T_{\text{THRES}}) \\ &= \begin{cases} 1 & \text{if } o_1 \text{ and } o_2 \text{ are close spatially and temporally} \\ 0 & \text{otherwise} \end{cases} \end{aligned} \tag{4.13}$$

Consider two VRPTW instances, I_1 and I_2. Every order in I_1 can be compared with every order in I_2 to find similar orders in the other instance. This presents us with another problem: each order in I_1 can have multiple similar orders in I_2. To have a more rigorous notion of similarity, each order in I_1 should only be mapped to one other order in I_2. Furthermore, the number of these mappings should be unique, so we consider the maximum number of such mappings possible.

To do this, we can consider I_1 and I_2 to form two exclusive sets of order nodes, and the similarity metric given by Equation 4.13 can represent edges from I_1 to I_2. This forms a bipartite graph like the one seen in Figure 4.2. Now the goal is to find the maximum matching containing as many of these edges as possible, such that each node is at most on one of the edges.

This is the Maximum Cardinality Matching problem in graph theory, and there exist polynomial-time algorithms, such as the Hopcroft-Karp algorithm [33], to efficiently solve this problem. Based on this, we present our overall algorithm for calculating similarity between any two VRPTW instances I_1 and I_2 in Algorithm 2. Algorithm 2 creates a bipartite graph from the orders of the two instances, with the edges representing similarity between orders (see Equation 4.13). Then we use the Hopcroft-Karp algorithm[3] to find the maximum number of matching orders between I_1 and I_2. We then use this instead of the number of common orders to calculate the Jaccard Index between I_1 and I_2.

[2]We can also choose to use the time window ends similarly.

[3]Provided by networkx [34], a python library for graph analysis.

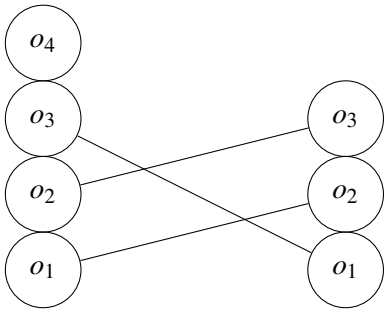


Figure 4.2: Bipartite graph with edges represented by similarity between orders in $I1$ and $I2$.

Algorithm 2 Algorithm to calculate similarity between two instances

Input I_1, I_2, D_{THRES}, T_{THRES}

Initialize $edges = []$.
for o_1 in $I_1.O$ **do**
  for o_2 in $I_2.O$ **do**
    if $S_{order}(o_1, o_2, D_{THRES}, T_{THRES})$ **then**
      Store the edge from o_1 and o_2 in *edges*
    end if
  end for
end for
Create a bipartite graph G with $I_1.O$ and $I_2.O$ as the disjoint sets and the edges represented by *edges*.
num_pairs = maximum cardinality of G using the Hopcroft-Karp algorithm.
return $\frac{num\_pairs}{|I_1.O|+|I_2.O|-num\_pairs}$

It is interesting to note the effect D_{THRES}, T_{THRES} have on our similarity heuristic. If we set these thresholds to a large value, then we will observe a lot of similarity. If we set these thresholds to a small value (0), then we fall back to the naive Jaccard Index definition proposed in the previous chapter. Therefore, setting these thresholds to reasonable values requires experimentation, which is presented in the next chapter.

4.3 Recycling Solutions

We turn our attention to the second problem we have to tackle: constructing a solution for an instance from a similar solved instance. We will assume that we have two similar instances: I_{cache}, an instance that we already know the solution for, and I_{curr}, the instance we currently aim to solve. The goal is to create an initial solution for I_{curr} from I_{cache}. We present our algorithm, which produces a valid initial set of routes, in Algorithm 3.

Algorithm 3 Algorithm for Recycling a solution

Input I_{curr}, I_{cache}

1: Create a Bidirectional map, (*cache_curr_map*), of orders from I_{curr} to I_{cache} using the Hopcroft-Karp algorithm.
2: Initialize $new\_routes = []$
3: **for** $cached\_route$ in $I_{cache}.routes$ **do**
4: Initialize arr_time for the new route to 0.
5: Initialize the vehicle location $v\_loc = depot$.
6: Initialize $new\_route = []$
7: **for** $cached\_order$ in $cached\_route$ **do**
8: **if** $curr\_order$ has a match in $cache\_curr\_map$ **then**
9: arr_time += travel_time(v_loc, $curr\_order$)
10: **if** arr_time is within $curr\_order$ time window, and we can reach the depot before the shift ends **then**
11: append $curr\_order$ to new_route
12: update v_loc to $curr\_order.loc$
13: **else**
14: arr_time -= travel_time(v_loc, $curr\_order$)
15: **end if**
16: **end if**
17: **end for**
18: append new_route to new_routes
19: **end for**
20: **return** new_routes

Algorithm 3 starts by using the Hopcroft-Karp algorithm to create a bidirectional map of orders from I_{curr} to I_{cache}. Earlier, we used the Hopcroft-Karp algorithm to find the number of such pairs to find similarity between two instances (see Algorithm 2), but here we are interested in the pairs themselves. We greedily create a new route for I_{curr} by assigning orders to vehicles (routes) by looking at how a similar order was assigned in I_{cache}. We

make sure that our new route is valid by only adding an order to the new route if it can be reached within its time window and the vehicle can reach the depot afterwards, before the shift ends. Finally, we return all the vehicle routes that were constructed.

This initial route is not good by any measure, as only a fraction of the orders in the instance will be routed. While this route can be used for variable and value selection strategies in the CP solver, we go one step further and constrain the decision variables (successors, predecessors, and vehicles) to the initial route for the orders that are present in it. By doing this, we lose any guarantee for optimality, but what we gain is a massive reduction in the search space. Therefore, the CP solver only has to solve for the assignments of the remaining orders. Intuitively, this can be thought of as having the CP solver search for solutions in promising regions of the search space, dictated by the solution of I_{cache}.

With the problem of measuring similarity and constructing an initial solution addressed, we have the necessary tools to develop a methodology to efficiently solve recurring VRPTW instances.

Algorithm 4 Solution Recycling Methodology

Input VRPTW instance I, cache of solved instances *cache*

1: Calculate the similarity between I and all the instances in *cache*
2: **if** we find similar instances in *cache* **then**
3: Choose a $candidate$ instance from *cache* by doing a similarity weighted sampling.
4: Construct an initial solution as hint (see Algorithm 3) using the $candidate$ instance.
5: Solve instance I with the CP solver using the initial solution as hints.
6: **else**
7: Solve instance I without hints.
8: **end if**
9: Update the cache with I and its solution.
10: **return** The solution for instance I.

Algorithm 4 presents our overall methodology. We take in as input the instance that needs to be solved and a cache of solved instances (which is none initially). We look for similar instances, based on a similarity threshold, to the current instance in a cache of solved instances. If there are similar instances, we choose a candidate instance by randomly choosing one based on its similarity to the current instance. We use this candidate instance to construct an initial route and use these routes as hints to solve the instance. Finally, we save the instance and its solution before returning the solution.

This methodology is designed to repeatedly solve instances, so as we solve more instances, we also have more instances to recycle solutions from. One thing we do need to take into account is the minimum similarity of candidate instances. This is another threshold that needs to be tuned based on experimentation.

4.4 Probabilistic Caching

In the previous section, we implicitly assumed that the cache has an unlimited size. However, evaluating the similarity between all the cached solutions does have a polynomial

overhead that can add up when the cache is large. Therefore, we also need to limit the number of cached solutions that are stored.

This presents a problem where we need to choose which instances to hold in the cache. A naive strategy is to only cache the last p instances. However, with instance data in our hands, we attempt to keep only the most relevant instances. We already assume that our instance has spatial and temporal correlations. We further assume that there exists an underlying probability distribution that influences the instances. The main idea is to use a probabilistic model to approximate this underlying probability distribution.

The advantage a probabilistic model offers is that we can query the model to see how likely an instance is. For example, given an instance I and a hypothetical probabilistic model that approximates the underlying distribution over instances, we can query the model to ask how likely instance I is. We can expand this idea to our cache of instances and only keep a limited number of instances that have a higher likelihood.

We use the Bayesian Gaussian Mixture Model (BGMM) as our probabilistic model. The BGMM allows us to approximate the posterior distribution of instances based on previous data. Each instance is considered as a set of orders, and each order itself is a 4-dimensional vector ($\mathcal{R}^4$). We train a BGMM in 4-dimensions with an infinite number of clusters and a full covariance matrix using a single instance (seed 0) of 1000 orders.

Practically, the number of clusters is chosen probabilistically, which depends on the instances themselves. A full covariance matrix allows the BGMM to fit any sort of distribution. To evaluate if the BGMM is trained, we also do some posterior predictive checks. The first predictive check is a visual inspection of the samples generated by the trained BGMM (see Figure 4.3).

We can note that the attributes of the sampled orders (location, time windows) are what we expect. Even though the BGMM generates unlikely data, for example, some time window starts are less than 0, we can still conclude that it is a reasonable model for querying likelihoods.

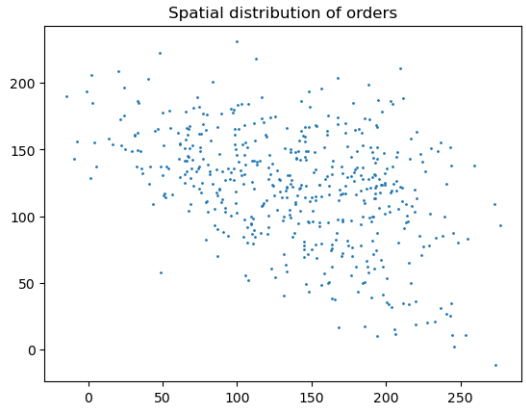
The second check is a randomized check to see if the likelihoods of true instances (an instance generated from our dataset) are always higher than the likelihoods of a random instance (generated randomly). The pseudocode of this check is given in Algorithm 5.

Algorithm 5 Random testing of BGMM

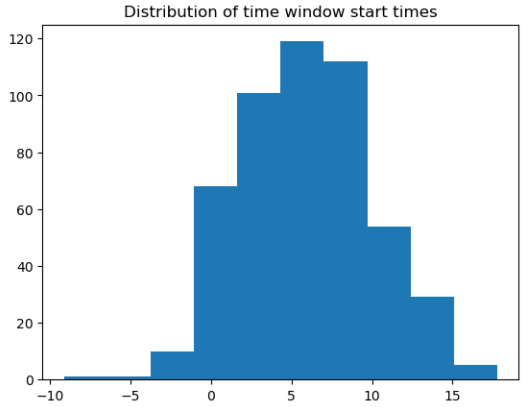
Input Trained BGMM

1: **while** True **do**
2: Generate a true instance from the Belgian dataset.
3: Generate a random instance of the same size.
4: **if** Likelihood of random instance is higher than the likelihood of the true instance **then**
5: **return** Test failed
6: **end if**
7: **end while**

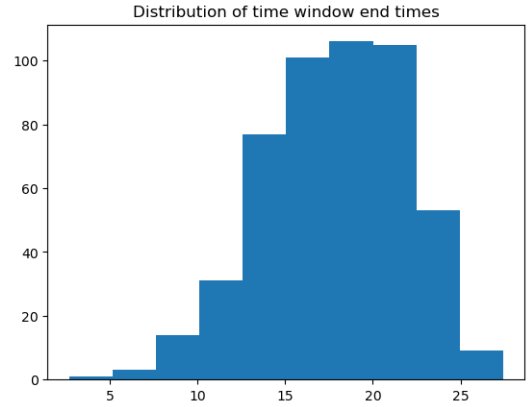
With a trained BGMM to evaluate the likelihood of any instance, we present the following extension to our Solution Recycling Methodology: specifically, an extension to line



(a) Locations



(b) Time window starts



(c) Time window ends

Figure 4.3: Distributions of different attributes of orders sampled from a trained BGMM

9 in Algorithm 4. The Probabilistic Caching algorithm shown in Algorithm 6 calculates the likelihoods of all the instances in the cache and only retains the instances (and their solutions) based on their likelihoods, as dictated by the BGMM.

Algorithm 6 Probabilistic Caching

Input Instance I, *cache*, p

1: Calculate the likelihood of all the instances in *cache* and I by querying the BGMM.
2: Do a likelihood weighted sampling of p instances with replacement.
3: Update the cache with only the p chosen instances.

This chapter presented our approaches to solving the problems introduced in Section 3.2 to put together a solution recycling methodology. The next chapter evaluates the effectiveness of these approaches.

Chapter 5

Experimental Evaluation

We presented three main ideas in the pursuit of developing an effective method to solve recurring VRPTW instances. To test our claim of being effective, we formulate the following questions in regard to the problems we discussed in Section 3.2.

P1Q1 – What effect does D_{THRES} and T_{THRES} have on our similarity metric?
To answer this question, we perform experiments comparing the similarity between a number of VRPTW instances and investigate the influence on similarity for different values of D_{THRES} and T_{THRES}.

P2Q1 – Does our solution recycling methodology allow the solver to quickly reach a first solution?
This question helps answer the influence recycling solutions have on the solver. The claim is that with our constructed initial solution, the solver would be able to quickly reach a first solution. To answer this question, we compare the time it takes for a solver to reach the first solution with and without our initial solution hints.

P2Q2 – Is our solution recycling methodology effective?
Perhaps the most important question we have to address in regard to this thesis. To answer this question, we directly compare the objectives reached by a solver augmented with our methodology (with an infinite cache) against a simple solver.

P2Q3 – How does our methodology influence decision-making for a small company?
This can be considered an extension of the previous question, which helps answer how our methodology would have influenced the operations of a small company. To answer this question, we break down the objective into its components: the number of orders that are delivered, and the total time traveled by a vehicle. We compare the performance of our methodology against a simple solver in regard to these metrics.

P3Q1 – How does the probabilistic cache influence the cache hits?
The claim is that the probabilistic cache will keep a few relevant solutions in the cache. To test this claim, we compare the probabilistic cache against a naive cache, which only keeps a cache of the latest instances it has solved. We are interested in seeing how the cache hit rate compares against the two cache types.

All our testing was done on a DelftBlue [35] computer node with 12 cores and 512 MB

memory per core, except where specified. We use OR-tools [36] to model the optimization problem and solve the problem with the CP-SAT solver. The BGMM implementation was provided by Scikit-Learn [37]. The VRPTW instances that were experimented with have the following properties.

- Approximate Number of orders in each instance = 100
- Number of vehicles = 10
- Vehicles' speed = 50 Km/hr

We define the following strategies which solve recurring VRPTW instances.

1. **Baseline Strategy** - As the name implies, the baseline strategy stands as the control of our experiment. It directly corresponds to the CP model we describe in chapter 3. This strategy has a time limit of 1 minute to solve an instance.

2. **Caching Strategy (infinite cache)** - This is an extension to the baseline strategy with our solution recycling methodology. We use this strategy with an infinite cache (no cache size limit) to answer questions **P2Q1, P2Q2, P2Q3**. This strategy has a time limit of 1 minute to solve an instance.

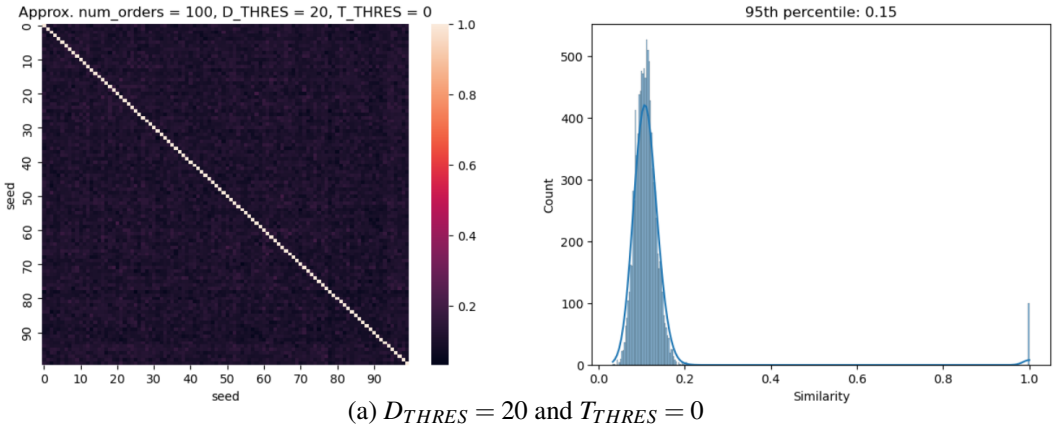
3. **Caching Strategy (finite cache)** - In a similar vein, this is an extension to the baseline strategy with our solution recycling methodology with a finite cache size p. It simply keeps the last p solved instances and their solutions. Used to answer question **P3Q1**. This strategy has a time limit of 1 minute to solve an instance.

4. **Probabilistic Caching Strategy (finite cache)** - This is an extension to the baseline strategy with our solution recycling methodology with a finite cache size p. It queries a trained BGMM to keep only the p most likely instances. Used to answer question **P3Q1**. This strategy has a time limit of 1 minute to solve an instance.

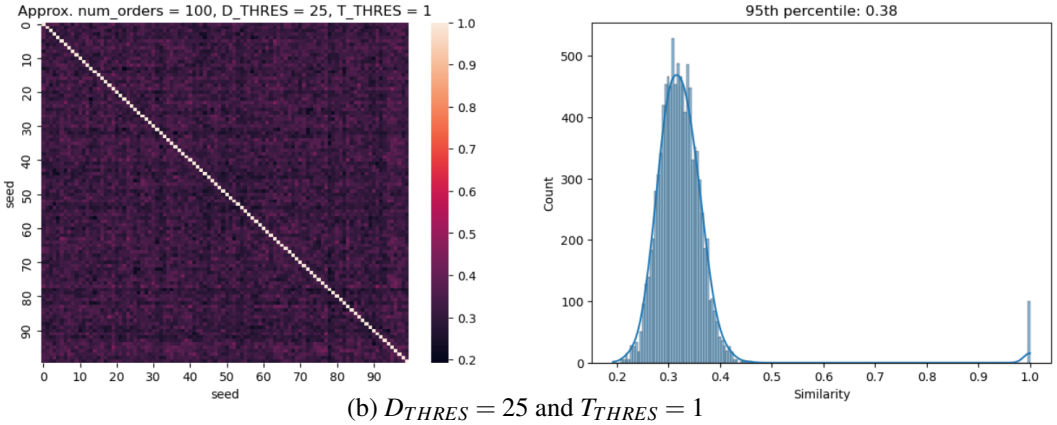
5. **Best-Known Strategy** - This strategy is basically the baseline strategy, but with a lot more computational resources (24 cores, 12 GB of memory) and 6 minutes to solve an instance. This serves as a stand in for a hypothetical optimal strategy. The best-known strategy can be used to compare how close each strategy gets to an optimal solution.

5.1 Experiment 1 – Similarity of Instances

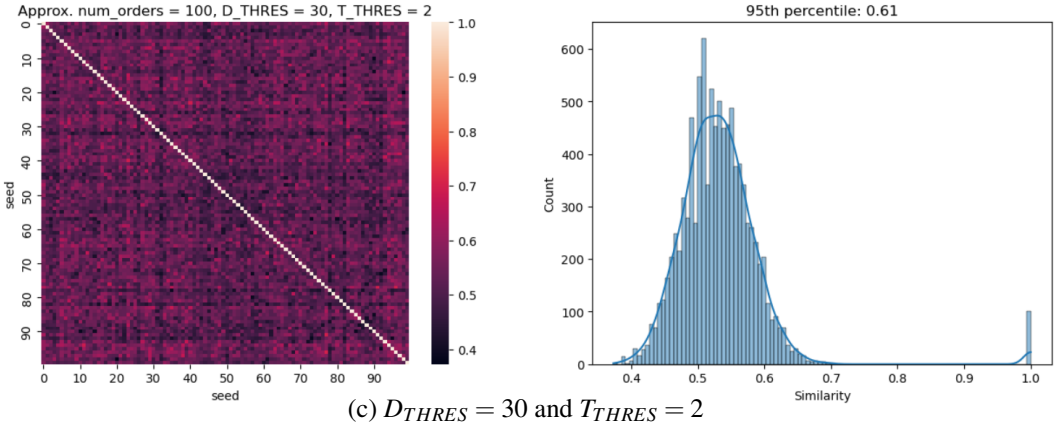
We compare the similarity of 100 instances (seed 0-100) against a combination of $D_{THRES} = [20, 25, 30]$ and $T_{THRES} = [0, 1, 2]$. Figure 5.1 shows the heatmap and the corresponding distribution of similarity values between each pair of instances.



(a) $D_{THRES} = 20$ and $T_{THRES} = 0$



(b) $D_{THRES} = 25$ and $T_{THRES} = 1$



(c) $D_{THRES} = 30$ and $T_{THRES} = 2$

Figure 5.1: Comparison of threshold values on Similarity between instances.

5.1.1 Discussion

As we expected, increasing the threshold values increases the similarity between instances. This is explicable given how heavily our measure depends on the definition of what makes two orders similar. If we increased the threshold values, we would naturally see an increase in the number of matches that could be made between two instances.

As a result, rather than an objective measure, problem-specific information has a significant influence on the similarity values. We chose $D_{THRES} = 25$ and $T_{THRES} = 1$ for further testing our methodology. This is done for two main reasons: a distance threshold of 25 corresponds to the distance traveled by any vehicle in 30 minutes, and a time threshold of 1 corresponds to 30 minutes. Essentially, we consider orders to be close if they are 30 minutes apart, spatially and temporally.

5.2 Experiment 2 – Solution Recycling Methodology

We compare the baseline, caching strategy (with infinite cache) and the best-known strategy against each other on 200 instances (seed 0-200). Our results are presented in Figure 5.2.

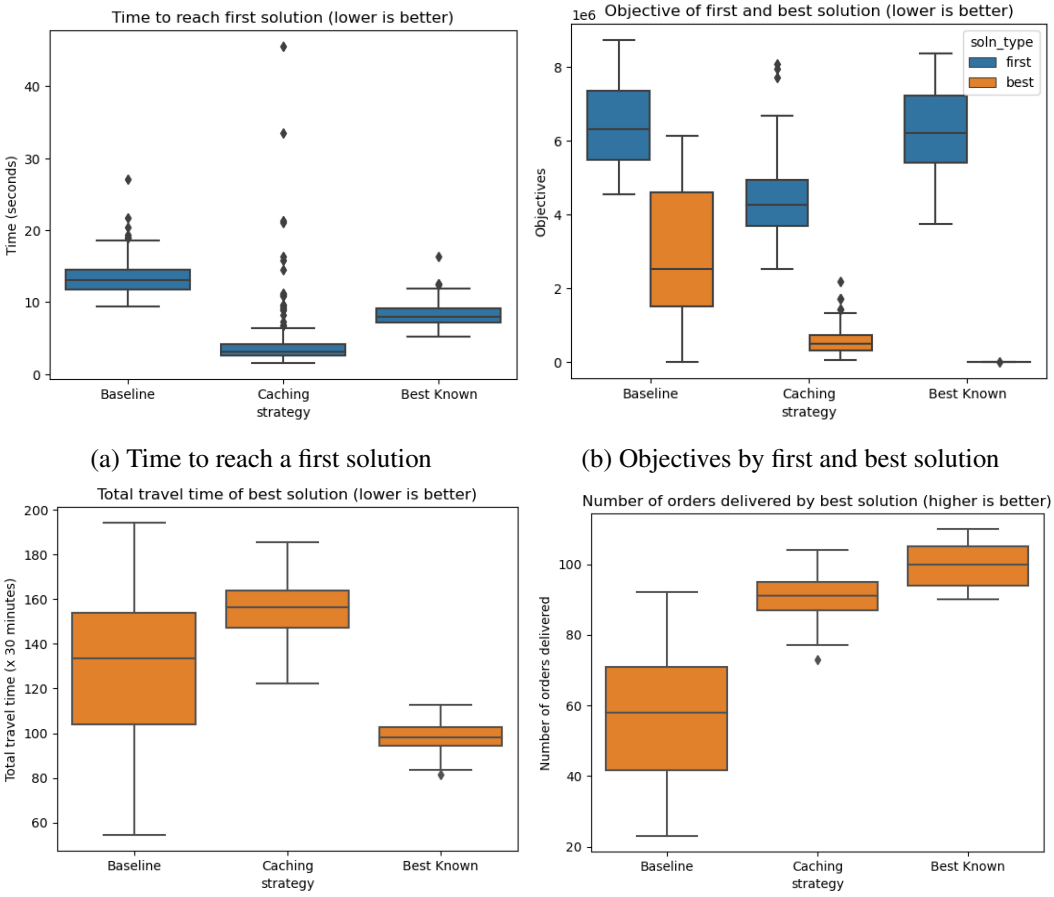
5.2.1 Discussion

We can clearly observe from Figure 5.2a that the caching strategy reaches a first solution faster than the baseline solver. This shows that our methodology lets the solver quickly reach a valid first solution, answering question **P2Q1**.

To answer question **P2Q2**, we can observe the objective values of the first and last solutions of each strategy from Figure 5.2b. Not only does the caching strategy reach a first solution faster, it also finds a much better first solution compared to the baseline strategy. The trend is continued in the overall best solution, where the caching strategy outperforms the baseline method and, in fact, performs much closer to the best-known strategy. We can conclude from this experiment that our methodology is indeed effective. We further investigate the influence our methodology has on the components that make up the objective value.

We can answer question **P2Q3** by looking at Figure 5.2c and Figure 5.2d. We can see that even though the caching strategy has a higher total travel time, it delivers much more orders than the baseline strategy. Compared to the best-known strategy (which delivered every order in all 200 instances), the caching strategy performs very closely. The caching strategy spends about 1586 more hours on travel time but delivers 5665 more orders than the baseline strategy over 200 instances.

We believe the reason our methodology works is that recycling a solution allows the CP solver to search for solutions in a promising neighborhood in the search space based on experience. Even though we lose the guarantee of optimality, our methodology performs close to optimal. This may suggest that similar instances have similar neighborhoods in the search space. By providing a partial solution to the CP solver, it can quickly leverage that information to search for more promising solutions.



(a) Time to reach a first solution

(b) Objectives by first and best solution

(c) Objective - Total vehicle travel time

(d) Objective - Number of orders delivered

Figure 5.2: Comparison of solutions by different strategies.

5.3 Experiment 3 – Probabilistic Caching

We compare the caching strategy and the probabilistic caching strategy with a finite cache size of $p = 20$. The same 200 instances (seed 0-200) are used in this case, with the only difference being the limited cache size. The only real difference between the two strategies is the instances they cache. The caching strategy naively keeps the last 20 solved instances, whereas the probabilistic caching strategy consults the BGMM for likely instances.

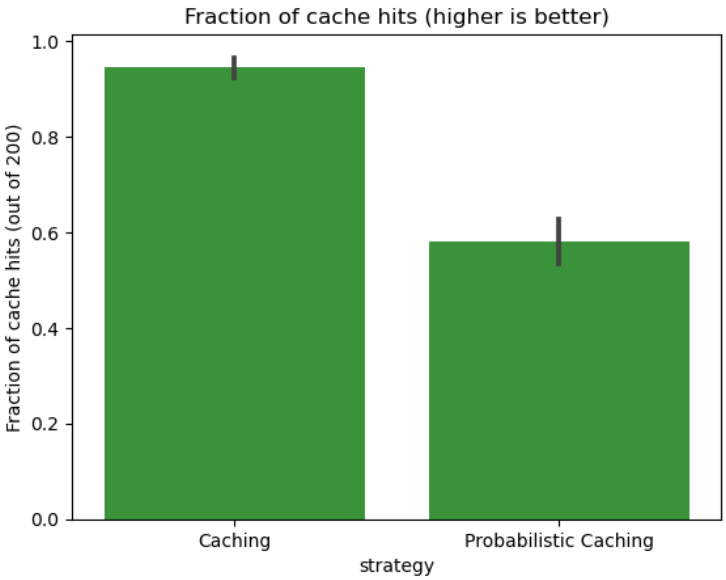


Figure 5.3: Comparison of cache hit rate

5.3.1 Discussion

To answer question **P3Q1**, we can look at Figure 5.3. We can observe that our naive caching strategy was able to perform much better than the complex probabilistic strategy, showing that our probabilistic caching method was not as effective as we thought it would be.

One theory is that the likelihoods of the instances are so small (in the order of e^{-300}) that they do not contribute to anything in the probabilistic cache. In essence, the likelihood weights do not mean anything, and we simply perform a random sampling with low weights.

The main conclusion we can draw from this experiment is that simpler methods, like naive caching, are often effective against more complex methods.

Chapter 6

Conclusion and Future Work

In this thesis, we have introduced a novel approach to address the challenge of efficiently solving recurring instances of the VRPTW by recycling solutions. We hope our approach inspires more solution recycling approaches in heuristic and metaheuristic research, as it is an approach that has not been extensively explored in the field of combinatorial optimization.

We began by establishing a clear understanding of the types of instances we are targeting. Through careful consideration of the problem characteristics and practical context, we defined a set of semi-realistic instances that mirror the routing problems faced in the real world.

A main part of our approach was redefining similarity tailored for the VRPTW. By defining and using this metric, we are able to systematically compare instances and assess their potential for sharing common solution structures, even though our metric is not necessarily an objective measure.

We have also shown that we are able to leverage this commonality in solution structures by defining a very simple solution recycling algorithm. This commonality allows us to reuse solutions from similar instances as a starting point for solving new instances.

A significant limitation of our methodology is that it requires custom heuristics that make a lot of assumptions about the problem data. For example, the similarity heuristic we use is heavily shaped by the nature of the instances we consider. Real-world problems require much more constraints and have a lot of variability, and it is not trivial to apply our methodology. A more well-suited intuition could be to approach similarity from a graph theory perspective.

Furthermore, we only evaluated our methodology against smaller instances. It would be interesting to see if our assumptions hold against larger instances as well. It might be the case that larger instances have an enormous solution space where our definition of similarity will break down. Consequently, this would also break our recycling heuristic.

Based on these two main limitations, our methodology suffers from generalizability.

A future work that should be explored is the use of our recycling methodology with metaheuristical methods. The benefits of recycling solutions integrated with a metaheuristical algorithm by building upon a partial solution might allow the algorithm to effectively search a promising neighborhood in the solution space.

For example, in step 1 of algorithm 1, instead of initializing a population with random solutions, part of the population can be recycled solutions, allowing the HGS algorithm to potentially search for solutions based on experience.

Because we recycle solutions from the method (exact or metaheuristic) that produced the solutions in the first place, there might also be algorithm-specific information that can be given to the algorithm itself. Generalizing our methodology would allow a primitive form of experience-based learning across a wide variety of algorithms.

We believe there is work to be done in introducing a more generalized method that can be applied to any combinatorial optimization problem where it makes sense to reuse and recycle solutions. In conclusion, we encourage the broader adoption of solution recycling as a powerful tool for solving challenging optimization problems.

Bibliography

[1] Shelagh Dolan. The challenges of last mile logistics & delivery technology solutions - business insider, May 2018. URL https://www.businessinsider.com/last-mile-delivery-shipping-explained?IR=T.

[2] Marius M. Solomon. Algorithms for the vehicle routing and scheduling problems with time window constraints. *Operations Research*, 35(2):254–265, 1987. ISSN 0030364X, 15265463. URL http://www.jstor.org/stable/170697.

[3] Jörg Homberger and Hermann Gehring. A two-phase hybrid metaheuristic for the vehicle routing problem with time windows. *European Journal of Operational Research*, 162(1):220–238, 2005. ISSN 0377-2217. doi: https://doi.org/10.1016/j.ejor.2004.01.027. URL https://www.sciencedirect.com/science/article/pii/S0377221704000736. Logistics: From Theory to Application.

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

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

[6] Nicos Christofides, Aristide Mingozzi, and Paolo Toth. Exact algorithms for the vehicle routing problem, based on spanning tree and shortest path relaxations. *Mathematical Programming*, 20:255–282, 1981. URL https://api.semanticscholar.org/CorpusID:18086758.

[7] Gilbert Laporte and Yves Nobert. A cutting planes algorithm for the m-salesmen problem. *The Journal of the Operational Research Society*, 31(11):1017–1023, 1980. ISSN 01605682, 14769360. URL http://www.jstor.org/stable/2581282.

[8] G. Laporte and Y. Nobert. A branch and bound algorithm for the capacitated vehicle routing problem. *Operations-Research-Spektrum*, 5(2):77–85, Jun 1983. ISSN 1436-6304. doi: 10.1007/BF01720015. URL https://doi.org/10.1007/BF01720015.

[9] Gilbert Laporte, Yves Nobert, and Martin Desrochers. Optimal routing under capacity and distance restrictions. *Operations Research*, 33(5):1050–1073, 1985. ISSN 0030364X, 15265463. URL http://www.jstor.org/stable/170853.

[10] Ralph E. Gomory. An algorithm for integer solutions to linear programs. 1958. URL https://api.semanticscholar.org/CorpusID:116324171.

[11] Ricardo Fukasawa, Humberto Longo, Jens Lysgaard, Marcus Poggi de Aragão, Marcelo Reis, Eduardo Uchoa, and Renato F. Werneck. Robust branch-and-cut-and-price for the capacitated vehicle routing problem. *Mathematical Programming*, 106 (3):491–511, May 2006. ISSN 1436-4646. doi: 10.1007/s10107-005-0644-x. URL https://doi.org/10.1007/s10107-005-0644-x.

[12] Roberto Baldacci, Nicos Christofides, and Aristide Mingozzi. An exact algorithm for the vehicle routing problem based on the set partitioning formulation with additional cuts. *Mathematical Programming*, 115(2):351–385, Oct 2008. ISSN 1436-4646. doi: 10.1007/s10107-007-0178-5. URL https://doi.org/10.1007/s10107-007-0178-5.

[13] G. A. Croes. A method for solving traveling-salesman problems. *Operations Research*, 6(6):791–812, 1958. ISSN 0030364X, 15265463. URL http://www.jstor.org/stable/167074.

[14] Fred Glover. Future paths for integer programming and links to artificial intelligence. *Computers & Operations Research*, 13(5):533–549, 1986. ISSN 0305-0548. doi: https://doi.org/10.1016/0305-0548(86)90048-1. URL https://www.sciencedirect.com/science/article/pii/0305054886900481. Applications of Integer Programming.

[15] Fred Glover. Tabu search—part i. *ORSA Journal on Computing*, 1(3):190–206, 1989. doi: 10.1287/ijoc.1.3.190. URL https://doi.org/10.1287/ijoc.1.3.190.

[16] Fred Glover. Tabu search—part ii. *ORSA Journal on Computing*, 2(1):4–32, 1990. doi: 10.1287/ijoc.2.1.4. URL https://doi.org/10.1287/ijoc.2.1.4.

[17] Michel Gendreau, Alain Hertz, and Gilbert Laporte. A tabu search heuristic for the vehicle routing problem. *Management Science*, 40(10):1276–1290, 1994. ISSN 00251909, 15265501. URL http://www.jstor.org/stable/2661622.

[18] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, 1983. doi: 10.1126/science.220.4598.671. URL https://www.science.org/doi/abs/10.1126/science.220.4598.671.

[19] N. Mladenović and P. Hansen. Variable neighborhood search. *Computers & Operations Research*, 24(11):1097–1100, 1997. ISSN 0305-0548. doi: https://doi.org/10.1016/S0305-0548(97)00031-2. URL https://www.sciencedirect.com/science/article/pii/S0305054897000312.

[20] 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*, pages 417–431, Berlin, Heidelberg, 1998. Springer Berlin Heidelberg. ISBN 978-3-540-49481-2.

[21] Stefan Ropke and David Pisinger. An adaptive large neighborhood search heuristic for the pickup and delivery problem with time windows. *Transportation Science*, 40(4): 455–472, 2006. doi: 10.1287/trsc.1050.0135. URL https://doi.org/10.1287/trsc.1050.0135.

[22] David Pisinger and Stefan Ropke. A general heuristic for vehicle routing problems. *Computers & Operations Research*, 34(8):2403–2435, 2007. ISSN 0305-0548. doi: https://doi.org/10.1016/j.cor.2005.09.012. URL https://www.sciencedirect.com/science/article/pii/S0305054805003023.

[23] Thibaut Vidal, Teodor Gabriel Crainic, Michel Gendreau, Nadia Lahrichi, and Walter Rei. A hybrid genetic algorithm for multidepot and periodic vehicle routing problems. *Operations Research*, 60(3):611–624, 2012. ISSN 0030364X, 15265463. URL http://www.jstor.org/stable/23260157.

[24] Thibaut Vidal. Hybrid genetic search for the cvrp: Open-source implementation and swap* neighborhood. *Computers & Operations Research*, 140:105643, 2022. ISSN 0305-0548. doi: https://doi.org/10.1016/j.cor.2021.105643. URL https://www.sciencedirect.com/science/article/pii/S030505482100349X.

[25] Wouter Kool, Joep Olde Juninck, Ernst Roos, Kamiel Cornelissen, Pim Agterberg, Jelke van Hoorn, and Thomas Visser. Hybrid genetic search for the vehicle routing problem with time windows: a high-performance implementation. *12th DIMACS Implementation Challenge Workshop*, 2022.

[26] Geoffrey De Smet and Tony Wauters. Multithreaded incremental solving for local search based metaheuristics with step chasing. 09 2020.

[27] Geoffrey De Smet and open source contributors. *OptaPlanner User Guide*. Red Hat, Inc. or third-party contributors, 2006. URL https://www.optaplanner.org. OptaPlanner is an open source constraint solver in Java.

[28] Paul Shaw. A new local search algorithm providing high quality solutions to vehicle routing problems. 1997. URL https://api.semanticscholar.org/CorpusID:14841431.

[29] Philip Kilby, Patrick Prosser, and Paul Shaw. *Guided Local Search for the Vehicle Routing Problem with Time Windows*, pages 473–486. Springer US, Boston, MA, 1999. ISBN 978-1-4615-5775-3. doi: 10.1007/978-1-4615-5775-3_32. URL https://doi.org/10.1007/978-1-4615-5775-3_32.

[30] Bruno Backer, Vincent Furnon, Philip Kilby, Patrick Prosser, and Paul Shaw. Local search in constraint programming: Application to the vehicle routing problem. 10 1997.

[31] Jorge Mendoza, M. Hoskins, Christelle Guéret, Victor Pillac, and D. Vigo. VRP-REP: a vehicle routing community repository. In *VeRoLog'14*, Oslo, Norway, 2014. URL https://univ-angers.hal.science/hal-03285192.

[32] Alan D. Snow, Jeff Whitaker, Micah Cochran, Idan Miara, Joris Van den Bossche, Chris Mayo, Paul Cochrane, Greg Lucas, Jos de Kloe, Charles Karney, Filipe, Bas Couwenberg, Guillaume Lostis, Justin Dearing, George Ouzounoudis, Brendan Jurd, Christoph Gohlke, David Hoese, Mikhail Itkin, Ryan May, Bill Little, Heitor, Alexander Shadchin, Bernhard M. Wiedemann, Chris Barker, Chris Willoughby, DWesl, Dan Hemberger, and Dan Mahr. pyproj4/pyproj: 3.6.0 release, June 2023. URL https://doi.org/10.5281/zenodo.8028069.

[33] John E. Hopcroft and Richard M. Karp. A n5/2 algorithm for maximum matchings in bipartite. In *12th Annual Symposium on Switching and Automata Theory (swat 1971)*, pages 122–125, 1971. doi: 10.1109/SWAT.1971.1.

[34] Aric Hagberg, Pieter Swart, and Daniel S Chult. Exploring network structure, dynamics, and function using networkx. Technical report, Los Alamos National Lab.(LANL), Los Alamos, NM (United States), 2008.

[35] Delft High Performance Computing Centre (DHPC). DelftBlue Supercomputer (Phase 1). https://www.tudelft.nl/dhpc/ark:/44463/DelftBluePhase1, 2022.

[36] Laurent Perron and Vincent Furnon. Or-tools, 2023. URL https://developers.google.com/optimization/.

[37] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.