Erasmus University Rotterdam
Erasmus School of Economics
Bachelor Thesis Econometrie en Operationele Research

Efficient Multi-start Heuristic for a Driver and Vehicle Routing Problem Including Capacity and Time Window Constraints

Floor Schooten (530188)

Supervisor:	J.H. Zhu
Second assessor:	dr D. Huisman
Date final version:	1 July 2024

Abstract

This paper explores the Driver and Vehicle Routing Problem with two depots and one exchange location where drivers switch vehicles so that both can reach their respective destination. The driver needs to finish at the same depot they started without exceeding the duration constraint, while the vehicle needs to be transported from one depot to the other. This is done while the cost is minimized. The robustness of the efficiency of the multistart heuristic is tested to the addition of a capacity constraint and the addition of time window constraints. The multistart heuristic is robust to the addition of a capacity constraint but is not robust to the addition of time window constraints. This is because the time window constraints cannot be randomly organized or easily switched around in routes, which are both important components that make the multistart heuristic efficient.

1 Introduction

Efficient supply chain management is predominantly made possible due to the classical Vehicle Routing Problem (VRP), which is in turn inspired by the Traveling Salesman Problem. The VRP is a NP-hard problem, which means the running time exponentially increases with the instance size. Due to this increase in running time, solving heuristics rather than exact problems becomes more appealing. The optimization of a VRP provides a lot of value for companies, but often drivers and vehicles are seen as a single unit, as they often cannot travel independently of one another. However, seeing vehicles and drivers as separate units gives light to new applications of the VRP.

One of these applications is the Driver and Vehicle Routing Problem (DVRP), devised by Domínguez-Martín, Rodríguez-Martín and Salazar-González (2018a). This encapsulates a problem with two depots between which the trucks are to be moved, a predetermined number of points that require service by a vehicle and a driver, an exchange location that allows drivers to switch vehicles, and lastly, a time limit within which the drivers must return to their start depot. Here, the driver and vehicle destination are divergent. Multiple drivers can travel in a single vehicle for a part of the journey, if this dismisses the necessity of employing another vehicle. The exchange location is the only customer location that can be visited multiple times by multiple vehicles.

The DVRP has a multitude of applications, ranging from supplying a crew to flights with a stopover before the final destination to long-distance truck transportation. This problem was also inspired by the airports in the Canary Islands (Salazar-González, 2014), where there are 2 main airports and several smaller ones, where the crew must end at the same airport they started to reduce layovers, but airplanes had to end at different locations for maintenance purposes. In the case of long-distance truck transportation, the DVRP can be applied to substitute a single driver for the whole route with multiple drivers who can all return home at the end of the day. However, to translate a simple long-distance truck transportation to the form of a DVRP would require a long string of depots and exchange locations in between and lacks complexity. The complexity of the DVRP mainly comes from the number of customers requiring delivery of goods and services. Domínguez-Martín, Rodríguez-Martín and Salazar-González (2023) has especially highlighted the strength of the heuristic for problems containing a large number of customers. Therefore, instead of focusing on the real-life application on situations like the Canary Island airports or long-haul transportation, it is more interesting to focus on the robustness of the heuristic to larger instance sizes.

To test the robustness of the heuristic in the face of a more realistic situation, 2 additional constraints are added to the problem, a vehicle capacity constraint and time window constraints. Furthermore, to enhance the efficiency of the multi-start loop, the least possible number of drivers that might result in a feasible solution is determined, eliminating a substantial number of iteration and enhancing the efficiency of the heuristic. The research question is presented as follows: *How robust is the optimal solution and efficiency of the multi-start heuristic to the DVRP to vehicle capacity and time window constraints?* Robustness is tested against the benchmark results in Domínguez-Martín et al. (2023) and will be considered efficient if the heuristic presents good optimal solution with a maximum time increase of 50%. This paper finds that the heuristic is robust to capacity constraint Q, but not robust to time window constraints, even when every time window spans 40% of the time limit.

The remainder of this paper is structured as follows. In Section 2, a literature review is presented. Section 3 describes the DVRP and the notation is introduced. In Section 4 a detailed explanation of all the methods is described. The numerical results are presented in Section 5. Finally, the conclusion is presented in Section 6, along with suggestions for further research.

2 Literature Review

Although many variations of the VRP have been studied to no end, this is mainly limited to theoretical problems and has not often been widely studied academically. A paper by Žunić, Đonko, Šupić and Delalić (2020) looks at this problem from a realistic angle. Especially interesting is their take on the importance of the clustering of data, even going as far as to use it as an important factor in their heuristic. They also include time windows and capacity constraint.

2.1 Capacity Constraint

Capacity constraints are widely explored in the optimization literature. Pino et al. (2011) mentions several methods and concludes that two stage local search algorithms are among the most effective. Domínguez-Martín et al. (2023) uses a similar heuristic to approach the driver time limit for the DVRP. Pino et al. (2011) also mentions a branch-and-cut heuristic to be effective, which was also previously used for the DVRP in Domínguez-Martín, Rodríguez-Martín and Salazar-González (2018b). However, Domínguez-Martín et al. (2023) concluded that the branch-and-cut heuristic was less effective than a combination of two step local search algorithms.

2.2 Time Windows

Time windows are also a commonly researched problem in past literature, both in combination with capacitated vehicles (Lau, Sim & Teo, 2003) and on its own (Bräysy & Gendreau, 2005), and even combined with both capacity restrictions and clustering analysis (Vidal, 2015). However, the heuristics often used include machine learning techniques like k-means clustering or penalty systems when the time window is exceeded, like in Balakrishnan (1993). There is a lot of literature on the use of clustering as a heuristic in the study of VRP, like for example Barreto, Ferreira, Paixao and Santos (2007); Gocken and Yaktubay (2019) and Vidal (2015).

However, when looking at simple heuristics for VRP's with time window constraints, El-Sherbeny (2010) presents multiple exact methods, meta-heuristics and heuristics. The heuristics are divided into route-building heuristics and route-improving heuristics. A variation of the savings heuristic by Lysgaard (1997) is presented as a good route building heuristic. However, for the DVRP the savings heuristic is likely quite ineffective, due to the necessity to visit the exchange location. El-Sherbeny (2010) also presents neighborhood clustering heuristics as a good building heuristic and dismisses a 2-opt local search as the most efficient option.

3 Problem description

The DVRP can be described as a graph $G = (V, A)$, where the vertex set V consists of depots $D = \{0, n+1\}$ and customers $V_c = \{0, ..., n\}$. Among the customers, who all need to be visited by a vehicle, there is a unique exchange location $e \in V_c$ that has be visited by all drivers and vehicles, and is the only customer that can be visited by multiple. The arcs connecting vertices are denoted as the set $A = \{(i, j) : i, j \in V, i \neq j\}$. Each depot $d \in D$ has a set of drivers K_d and a set of vehicles L_d. For any subset $S \subset V$, the arcs in and out of this subset are denoted by $\delta^+(S) = \{(i, j) \in A : i \in S, j \notin S\}$ and $\delta^-(S) = \{(i, j) \in A : i \notin S, j \in S\}$ respectively. The time necessary for a driver to traverse arc $(i, j) \in A$ is denoted by t_{ij}, and the cost by c_{ij}. The drivers must return to depot of origin within the time limit T, while the vehicles have no time restriction to arrive at the other depot from which they started. The objective is to find feasible routes for both drivers and vehicles with the smallest cost and can be formulated as

$$min \sum_{k \in K} \sum_{(i,j) \in A} c_{ij} x_{ij}^k \tag{1}$$

where x_{ij}^k is a binary decision variable that takes value 1 if arc $(i, j) \in A$ is traversed by driver $k \in K$ and 0 otherwise. Since driver and vehicle routes are closely related, it will suffice to only include the driver costs into the calculation of the objective function.

To expand the model we introduce additional notation. The maximum vehicle capacity is denoted by Q, and the demand of each customer is denoted by q_i. The depots do not have any demand. To incorporate time windows constraints, a_i will be defined as the earliest time a driver and vehicle can arrive and start service, while b_i will be the latest time of arrival. In this context, the service time is not considered as a separate variable.

The mathematical MIP formulation of the DVRP can be found in Domínguez-Martín et al. (2018a)

To illustrate the problem, consider the example presented in Figure 1 with 7 customers, 2 depots and an exchange location. Here drivers (dotted lines) arrive at the same depot they started from while the vehicles (solid lines) arrive at a different depot. Drivers switch vehicles at the exchange location to ensure this. The drivers and vehicles leaving from the first depot are pink, while those departing from the other depot are purple. In this situation there is one driver leaving from each depot. If the time necessary to traverse each arc is assumed to be one, this solution would hold for $T = 7$, with the resulting driver routes being 0-1-7-6-8-2-4-0 and 9-5-3-8-9, while the vehicle routes are 0-1-7-6-8-9 and 9-5-3-8-2-4-0.

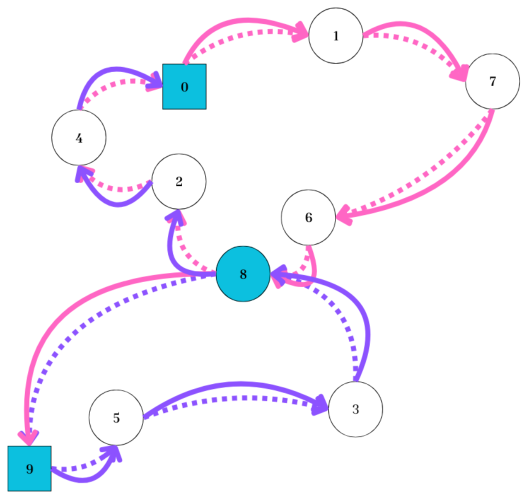

Figure 1: Example of a possible solution to the DVRP

4 Methodology

The original DVRP from Domínguez-Martín et al. (2018a) makes use of 2 heuristics to solve the MIP problem, one branch-and-cut algorithm (Domínguez-Martín et al., 2018b) and one multi-start heuristic(Domínguez-Martín et al., 2023). They found the multi-start heuristic to be the most efficient. This is the algorithm that will serve as the foundation of this paper. The DVRP is NP-complete and has polynomial time complexity, so it will get computationally more expensive when n increases. The algorithm consists of two parts, a multistart loop that contains a driver routes construction method and a local search improvement operator, and a main loop which assesses the number of drivers when a feasible solution has not been found and the generation of vehicle routes from the best feasible set of driver routes. In this section we will outline the original multistart heuristic to solve the DVRP and outline any changes and additions applied in this research.

To enhance the applicability of the research done by Domínguez-Martín et al. (2023), this research introduces more factors that can be present in real-life scenario's. The robustness of the model is tested by considering vehicle capacity constraints and time windows for customers. To investigate the applicability on real-life scenario's, the algorithm's efficacy is tested with a larger number of customer locations.

Furthermore, because we aim to test the robustness of the heuristic to certain real-life factors, the benchmark for this will be the results considered in Domínguez-Martín et al. (2023) and an extension of the multi-start heuristic will be considered efficient if the heuristic presents good optimal solution with a gap of under 5% with a maximum time increase of 50%.

4.1 Multi-start Heuristic

The multistart heuristic algorithm to solve the DVRP is described in Algorithm1: Where the input consists of a dataset, the *timeLim* and the *maxIter* and the output consists of S^*, the driver routes, R^*, the vehicle routes and f^*, the the solution value.

The original algorithm can be found in Domínguez-Martín et al. (2023), and the variation we will test is shown in algorithm 1.

Algorithm 1 2 step multistart heuristic for the DVRP

```
$f^* \leftarrow \infty,\ S^* \leftarrow \emptyset,\ nDrivers \leftarrow 1$
while $time \leq timeLim$ & $nDrivers \leq maxDrivers$ & no feasible solution for $S^*$ do
    $nIter \leftarrow 1$
    while $time \leq timeLim$ & $nIter \leq maxIter$ do
        $nDrivers \leftarrow calculateStartDrivers(n, T)$
        $S \leftarrow ConstructDriverSol(nDrivers)$
        $S \leftarrow LocalSearch(S)$
        if $S$ is feasible & $f(S) < f^*$ then
            $S^* \leftarrow S$
            $f^* \leftarrow f(S)$
        end if
        $nIter \leftarrow nIter + 1$
    end while
    if no feasible solution for $S^*$ then
        $nDrivers \leftarrow nDrivers + 1$
    end if
end while
if $S^* \neq \emptyset$ then
    $R^* \leftarrow ConstructVehicleRoutes(S^*)$
end if
return $S^*$, $R^*$ and $f^*$
```

The algorithm consists of a multi-start loop generating driver routes and is generally repeated *maxIter* times. Of these *maxIter* times, the best feasible solution $f(S^*)$ and the corresponding set of driver routes S^* are stored. The solution is evaluated for feasibility after the local search does not provide any further improvement, and a solution is considered feasible when none of the driver routes in S exceed the duration constraint T. If after *maxIter* multistart loops no feasible solution has been found, *nDrivers* is increased by 1. Here, *nDrivers* is defined as the number of drivers that leave a single depot. For each driver that leaves one depot, another driver has to leave from the other depot to ensure both the vehicle and the drivers arrive at their destination. If after *maxIter* iterations a feasible set of driver routes S^* has been found, the stopping criterion has been reached and the vehicle routes (R^*) are constructed from the optimal driver routes (S^*) once. If one of the other stopping criteria (the computing time limit or the driver maximum) have been reached without a feasible solution found, the driver routes cannot be constructed and there will be no solution found by this heuristic.

4.1.1 Start Drivers

One addition to this heuristic that was not in the original heuristic is the *calculateStartDrivers* method, which ensures the multi-start loop does not attempt to find a feasible solution when there are none. This can be determined because each arc has a minimum length. If the number of drivers cannot service all customers within T when each arc is assumed to be the minimum length, there is no feasible solution to be found. Starting with the least number of drivers necessary for a feasible solution can greatly aid the efficiency of the heuristic.

For example, if we assume each arc costs the minimum to traverse, which in this case is 0.5, and we want to service $n = 500$ customers and a driver route can take up to $T = 55$, each driver would be able to service 107 customers. If only one driver leaves from each depot, the total number of customers that can be served by those two drivers amounts to 214, which is significantly less than the 500 customers that need to be serviced. If two drivers leave from each depot, the maximum number of customers that can be serviced by the total of four vehicles amounts to 428, which is still less than 500 and a feasible solution can never be found, even when disregarding the euclidean distance that needs to be traveled.

The methods for constructing driver solutions, a local search algorithm and the driver construction methods that were present in the original heuristic are presented below.

4.1.2 Construct Driver Solutions

Similarly to Domínguez-Martín et al. (2023), the drivers routes are determined by an iterative cheapest insertion strategy for a predetermined number of drivers k that depart from a depot ($k \leq |K_d|, \forall \ d \in D$). The sequence of nodes that define the route for driver $l \in \{1, ...k\}$ from depot $d \in D$ is denoted by S_d^l. The set of routes for all drivers in the solution for the DVRP is denoted by $S = \cup_{d \in D} \cup_{l=1}^{k} S_d^l$. The driver routes in S are initialized as $S_d^l = \{d, e, d\}$, as each driver must visit the exchange location and start and return from the same depot. From that point, the iterative cheapest insertion strategy is applied to add customers into the cheapest driver route, where a randomly chosen customer $i \in V_c, i \notin S$ is inserted in the cheapest route for i in S. The cheapest route for i in S is determined by minimizing $c_{ui} + c_{iv} - c_{uv}$, where u, v are two consecutive nodes in a drivers route. Customer i is inserted in the best route in S according to the cheapest insertion criteria if it does not exceed the duration constraint T, otherwise it will be inserted in the shortest route in S at the best position. The advantage of this algorithm is that it allows for simple and fast implementation, however it does not guarantee high quality results and easily get stuck in local minima. If a customer i cannot be included in any route without exceeding the duration constraint, it can possibly made feasible later using local search algorithms. This strategy is repeated until all $i \in V_c$ are inserted in a route in S.

The cheapest insertion criteria is elected as opposed to other classical methods that share many advantages and disadvantages because it allows for insertion into an already exiting route, which allows the exchange point and the start and return from a depot to be initialized, eliminating the nearest neighbor heuristic as the best selection criteria. Furthermore, the savings heuristic proposed by Lysgaard (1997) is less efficient in this case, as it starts with a route for each separate customer and combines them. Domínguez-Martín et al. (2023) wants to use the least number of drivers possible, and using the savings heuristic may result in more solutions using more drivers than necessary.

We are looking at traditional methods because the strength of this algorithm is using these easy methods a lot of times to take away their disadvantages.

4.1.3 Local Search

After constructing feasible driver routes, the routes in S are improved upon using multiple local search algorithms to find a local minimum. After the evaluation of intra- and inter-customer relocation and 2-opt and 3-opt arc-exchanges, Domínguez-Martín et al. (2023) concluded that a simple strategy was best, consisting of an inter-route customer relocation local search followed by a 2-opt local search within a route. The difference between the inter- and intra-route customer relocation is that a customer is inserted in the same route it was removed from in the intra-route method, while for the inter-route method the customer is moved to a different route. The inter-route local search randomly selects a customer i from a route in solution S and transfers it from its current driver route to a different driver route in S according to the cheapest insertion criterion mentioned in section 4.1.2. The cheapest insertion criterion is applied to all routes in S until the cheapest insertion for i with the minimum cost increase is found, and while the solution S improves the method is applied iteratively until a local minimum is reached. The 2-opt local search is applied within each route and is applied by deleting two arcs, $(i, i+1)$ and $(j, j+1)$ and reconnecting them $(i, j+1)$ and $(j, i+1)$. After the inter-route customer relocation the 2-opt local search is applied, which deletes two arcs within a route and adds two different arcs to reconnect the route if the solution improves. This is repeated as long as the solution S decreases. Again, the advantages of these algorithms are that they are simple and fast to implement, but they do not offer high quality results and are also prone to getting stuck in local minima. This is countered by the design of the multistart loop, especially the number of iterations and the randomness in the construct driver routes method and the inter-route local search.

These methods can be key to reaching a feasible solution, construct driver routes method does not reach a feasible solution. One of the disadvantages of a local search algorithm is that the local minimum it will reach is very dependent on the starting location. This drawback is countered in this algorithm by using a number of iterations that have a randomly generated starting point, and keeping track of the best local minimum that has been found. However, when T is rather tight, there is not a lot of space to add nodes to a driver route, although it might be beneficial to do so. In this case, if after 80% of the iterations a feasible solution has not been reached, T is loosened only for the sake of the local search, in the hopes of finding a feasible solution. Not taking T into account during the local search will result in a large number of infeasible solutions and a lower chance of the optimal solution being found.

4.1.4 Construct Vehicle Routes

Vehicle routes can be generated according to the description given by Domínguez-Martín et al. (2018b). For each arc $(d, i) \in \delta^+(d)$ from depot d is traversed by a driver $k \in K_d$, a vehicle also needs to leave depot d using that arc. Because drivers are only allowed to switch at the exchange location e, the vehicle follows the drivers route until the exchange location, after which it follows the route of another driver $k' \in K_{d'}$ to depot d'. To make sure that each driver and vehicle reaches a proper destination, the number of arcs entering the other depot d' need to be equal to the number of vehicles departing from depot d. If they are not equal, a driver is traversing an arc entering d' without a vehicle and an extra vehicle needs to depart from depot d driven by a driver that is not currently driving another vehicle (but is a passenger). Also doing this analysis for the other depot d' results in a feasible solution R compatible with S. However, it is not necessary follow this algorithm to construct the vehicle routes, as the only costs that are accounted for in the optimal solution are those of a driver traversing an arc. Because of this, it is equally efficient to give each driver its own vehicle, so there are definitely an equal number of vehicles leaving and entering each depot. In Domínguez-Martín et al. (2023), it is proposed that drivers can also ride as a passenger, although this does not impact the optimal solution. With the capacity constraint that will be introduced later in mind, we have chosen to implement the latter strategy, where each driver is given a vehicle, and a vehicle route is composed of a driver route from depot d to e, and another driver route that goes from e to d'. This process is only repeated once, as the vehicle routes R are only constructed once an optimal solution S^* is found.

4.2 Capacity Constraint

The original DVRP introduces only a driver time constraint T, allowing the problem to be mostly one-dimensional. This causes swapping vehicles at the exchange point to be easy as few factors need to be taken into account. Driver routes can be generated with no regard to the number of customers visited before and after the exchange location. Adding a vehicle capacity constraint could make the number of customers visited by a single vehicle more realistic. The DVRP will only include single commodity delivery, as per the original problem. We assume any vehicle and any driver combination can service any customer. This vehicle capacity q_i^l will be tested for a single value that is homogeneous over all customers i. For this, q_i^l will denote the capacity of vehicle $l \in L$ before visiting customer $i \in V$, with Q the max capacity. This results in the following restriction:

$$\sum_{i \in V} q_i^l v_i^d \leq Q, \quad \forall l \in L \tag{2}$$

Whereas in Domínguez-Martín et al. (2018a) v_i^d is defined as the number of vehicles from depot $d \in D$. However, in the heuristic, this capacity constraint will not used exactly in this mathematical form.

To add this to the heuristic, it is necessary to check during the multistart loop whether a driver route can also produce feasible vehicle routes to prevent having to return to the inner multistart loop after driver routes have been generated as the multistart loop is computationally expensive.Furthermore, there is no guarantee the original heuristic will result in a local minimum that is also feasible for vehicles, even if the multi-start loop is repeated a multitude of times. The capacity constraint will be taken into consideration during the method constructing driver routes, during the local search and when checking feasibility. Vehicle routes are matched with pre- and post-exchange location driver routes during the construction of the driver routes to guarantee a feasible vehicle route solution. There might be some optimal solutions that are not as easily explored but it is much more efficient than having to restart the multistart heuristic when a set of driver routes produces an infeasible set of vehicle routes, or constructing vehicle routes every time a node is inserted or swapped.

The 2-opt local search that was previously elected due to its simplicity now becomes a lot more computationally expensive, as swaps between arcs before and after the exchange location will also need to comply with the capacity constraint. However, only applying the 2-opt local search to a route between a depot and the exchange location will keep the algorithm simplistic and will not require a lot of additional computational power to ensure feasible vehicle routes. To also account for switching customers between the pre- and post-exchange location route, switching between them will be done as part of the inter-route local search. 3 combinations of T and Q will be evaluated to ensure feasibility; a tight T and a loose Q, a loose Q and a tight T and loose values for both T and Q.

4.3 Time Windows

The time window for each customer $i \in V$ is defined as $[a_i, b_i]$, where a_i is the earliest time service can start and b_i is the latest time the service can start. The time it takes to travel from customer i to customer j is described in section 3 as t_{ij} and t_i^k is the time when driver k starts serving customer i. The service time is included in t_{ij} The following constraints are added:

$$a_i \leq t_i^k \leq b_i \quad \forall k \in K, \forall i \in V \tag{3}$$

$$t_j^k \geq t_i^k + t_{ij}, \quad \forall k \in K, \forall i, j \in V \tag{4}$$

$$t_d^k = 0, \quad \forall i \in V \tag{5}$$

For the time window allocation, a starting time will be randomly decided and the duration is constant, so the end time is determined according to this. To keep this consistent and not add an additional random component, the start and end time windows are the same for each numbered node. Since the generation of the nodes themselves are random, this allocation is therefore also random. This is applied by taking the constant duration as a percentage of T, which also ensures time windows expand when dealing with a larger problem. To ensure feasibility, $maxDrivers$ will need to be increased, although it is important to not choose the time window constraint too tight. A route is considered infeasible when a driver arrives at a customer after the end time. A driver is allowed to wait at a customer until the start time arrives, however this is costly and will need to be within the time limit T. We still use the cheapest insertion criterion instead of a time based criterion, but modify it to take into account starting times. The cheapest insertion criterion becomes $\min(c_{ui} + c_{iv} - c_{uv} + t_u^k, t_i^k)$. If a customer does not fit in any route that will make it feasible, a driver needs to be added, as there is no solution where the local search and two opt method will make it feasible. Therefore, the algorithm is slightly changed as drivers are added within the multi-start loop.

5 Results

There are 3 classes of data used to evaluate the heuristic, each with a different number of customers and evaluated for different T values. The number of customers in an instance is denoted by n. Including the depots this becomes $n + 2$. *Class I* contains 5 instances for each value of n where $n + 2 \in \{10, 15, 20, 25, 30\}$ and will be evaluated for T_A, T_B, T_C and T_D, with T_A being the tightest value and T_D the loosest. For *Class II*, with $n + 2 = 50$, there will be 16 instances for $T = 18$, and for *Class III*, 5 instances each will be evaluated for $n + 2 \in \{100, 200, 300, 400, 500, 600, 800, 1000\}$, for different values of T_A, T_B, T_C and T_D. The first and last node of each set of customers are the depots, $(0, n + 1)$, and the exchange location is always the last generated customer (n). The node coordinates are generated in a $[0, 100] \times [0, 100]$ square. In *Class I* the depots are also generated in this square, while in *Class II* and *Class III*, one depot is generated in $[0, 20] \times [0, 100]$, the exchange location in $[40, 60] \times [0, 100]$ and the other depot in $[80, 100] \times [0, 100]$. The instances of the data is available at: `https://data.mendeley.com/datasets/w5sbtwy8y9/4`. Here the first and last location represent the two depots and the last customer will be the exchange location.

To compute the optimal solution, the Euclidean distances between customer i and j will be used as the costs c_{ij} to traverse the arc. The time needed to traverse this arc is defined as $t_{ij} = c_{ij}/60 + 0.5$. Although it is not specified why 0.5 is added to each arc, it does ensure that customers further from the depots and exchange points will not cause the time necessary for a driver route to skyrocket in comparison to customers that are located more conveniently. One could argue that 0.5 accounts for the service time, while 60 represents the average speed when traveling along the arc. For the initial problem, the *maxDrivers* is initially set to 3 drivers to leave from each depot, resulting in a total of 6 drivers on the road This is increased when investigating time window constraints. The number of drivers will be denoted as the total number of drivers leaving both depots. *maxIter* is set to 100,000, which provides a good balance between the computing time and the value of the optimal solution, according to Domínguez-Martín et al. (2023).

In all tables below, the column labeled *name* corresponds to the set of data run, where the instance size is the number preceded by n, while the instance number is preceded by a dash. The columns T_A, T_B, T_C and T_D are the driver routes duration limits and *Sol* represents the optimal solution value. *D* corresponds to the number of drivers leaving a single depot in the optimal solution found. The percentage gap between this optimal solution and the optimal solution in Domínguez-Martín et al. (2023) and is defined as *((current solution value - previous solution value)/previous value)*100.* When a time gap is calculated, this is according to the same formula, using the times instead of the solution values. This means that negative gap values mean the current solution or time value is higher than the original value. The t column represents the running time in seconds.

Although not clearly specified in Domínguez-Martín et al. (2023), it is possible that the authors could have used multiple threads, as each iteration can be seen as independent of one another when each thread saves their own best solution, to be compared at the end of the multistart loop. As this is the only segment that unites all iterations because each iteration generates driver routes according to a random element, so it is possible that multiple independent threads can be used, as the randomness would not change this. Multi-threading allows multiple cores to be used. Our heuristic was programmed in a single thread Accounting for the fact that six cores could be used in the original results instead of one, the running times presented here have the same order of magnitude to the results in Domínguez-Martín et al. (2023). Although the speed increase of using multi-threading and multiple cores cannot exactly be translated into a factor because not each extra thread has the same effect, it does allow for a better comparison of the values in *Class III* where this makes the largest difference.

In *Class I* and *II*, there is little difference between the running times without the correction factor and the original results, although *Class II* is about 70% slower. This can be explained by the fact that the heuristic itself runs extremely fast, but each instance needs to be initialized. This cannot be done in multiple threads and therefore using additional threads does not influence the running speed as significantly. As the instance size is scaled up however, the initialization takes up proportionally less and less time, resulting in the use of multiple threads being more time effective. This reasoning could explain both why for *Class I* the running times are approximately equal to the running times in Domínguez-Martín et al. (2023), for *Class II* the running times becoming about 70% larger and for *Class III* the results becoming up to 600% as large for the largest values. To make for a fair comparison in *Class III*, a correction factor of six is taken into account, as initialization takes up an relatively small amount of time. *Class I&II* are presented without this correction factor. Of course, just including a factor of six is not exactly representative, however the authors of Domínguez-Martín et al. (2023) were unclear about the number of threads used and this is one of most plausible explanations for why the running times for large instances are about six times as large, when the order of the heuristics used is the same. Small differences can be be explained by inefficient code, but a factor of six, which coincides with the number of cores available.

5.1 Replication Results

Table 5.1 show the results for *Class I*. The number of drivers used in T_C and T_D is always equal to one, which is why they are not included as a column in the table. An interesting observation is that in table 5.1, any gaps in [-0.3, 0.3] usually present an optimal solution that differs just one from the results in Domínguez-Martín et al. (2023). Although it cannot be said with certainty, this difference can very well stem from rounding errors. The results in Domínguez-Martín et al. (2023) are presented as integers and it is unclear when they are rounded. Another possibility is that some local minima are not consistently reached within 100,000 iterations. In general, the solution values of Domínguez-Martín et al. (2023) outperform the solution values of this algorithm, with the exceptions of *n25-2* and *n30-1* for T_B. For *n25-3*, this heuristic consistently finds better solution values for all T. For *n20-2* for T_B, this heuristic fails to find a good optimal solution.

Some interesting results in terms of running time are the instances for T_A, for which this heuristic performs better than the benchmark. This is due to the start driver calculation and makes the results up to twice as fast. Table 5.1 shows the results for *Class II*. Notable here is that the heuristic needed an additional driver for *n50-5* as opposed to the results in Domínguez-Martín et al. (2023). However, the exact formulation did need this number of drivers. but found an optimal solution much higher than the optimal solution found here. The running times are somewhat slower than those of the benchmark. In Table 3, the results for *Class III* are shown. The number of drivers for T_A is always 3 for T_C always 2, which is why they are not included

name	T_A	D	Sol	g	t	T_B	D	Sol	g	t	T_C	Sol	g	t	T_D	Sol	g	t
n10-1	6	2	654	0.31	1.03	7	1	443	0.23	0.45	8	411	0.18	0.55	10	371	0.51	0.43
n10-2	5	2	486	0.00	1.20	6	1	293	0.34	0.39	7	293	0.44	0.39	10	293	0.44	0.44
n10-3	5	3	987	0.00	1.69	6	2	646	0.00	0.96	7	390	0.05	0.47	10	383	-0.08	0.43
n10-4	5	2	611	0.16	0.89	6	2	533	-0.19	0.95	7	384	-0.13	0.40	10	383	-0.07	0.39
n10-5	5	2	594	-0.17	1.00	6	1	366	0.27	0.49	7	355	-0.16	0.50	10	351	0.26	0.42
n15-1	6	2	456	0.44	1.87	8	1	349	0.00	0.64	9	349	-0.05	0.93	12	302	-0.04	0.73
n15-2	6	2	746	0.00	1.40	8	1	412	-0.48	0.59	9	405	-0.25	0.61	12	405	-0.25	0.63
n15-3	6	2	660	0.00	1.36	8	1	441	-0.23	0.60	9	387	-0.28	0.63	12	387	-0.28	0.62
n15-4	6	3	1093	-0.09	2.71	8	2	715	0.00	1.81	9	498	0.24	0.63	12	460	0.05	0.78
n15-5	6	2	788	0.13	1.43	8	2	749	-0.27	1.48	9	473	0.37	0.67	12	469	-0.01	0.66
n20-1	7	3	1263	2.68	3.81	9	2	844	0.12	2.49	10	559	0.42	0.96	14	521	0.22	1.26
n20-2	7	2	667	0.15	2.18	9	2	613	36.22	2.46	10	441	0.64	0.95	14	402	0.70	1.02
n20-3	7	2	846	0.12	1.93	9	2	757	0.00	2.63	10	540	0.02	0.90	14	508	0.17	1.26
n20-4	7	2	602	0.33	2.43	9	2	582	0.34	2.27	10	448	0.32	0.94	14	417	0.42	0.88
n20-5	7	2	651	0.62	2.20	9	2	538	0.00	2.65	10	434	0.38	1.05	14	409	-0.07	1.35
n25-1	8	2	719	0.14	3.10	10	2	708	0.14	3.05	11	501	0.48	1.25	16	483	0.01	1.28
n25-2	8	2	796	-0.13	2.47	10	2	699	-2.37	3.67	11	510	1.84	1.21	16	486	0.68	1.70
n25-3	8	2	601	-0.33	3.36	10	2	553	-0.36	3.02	11	436	-0.73	1.25	16	402	-0.62	1.35
n25-4	8	2	686	0.44	3.20	10	2	681	0.15	3.18	11	470	0.23	1.20	16	456	0.45	1.27
n25-5	8	2	724	0.42	3.17	10	2	692	0.44	3.31	11	494	0.55	1.26	16	454	0.66	1.33
n30-1	9	2	887	1.14	1.70	12	2	816	-0.97	4.64	13	616	0.11	1.65	18	582	0.25	2.56
n30-2	9	2	853	-0.12	2.32	12	2	827	0.12	4.42	13	561	0.17	1.58	18	553	0.26	1.83
n30-3	9	2	812	0.00	2.86	12	2	797	0.25	4.34	13	507	0.22	1.90	18	487	0.34	1.86
n30-4	9	2	721	0.28	1.81	12	2	663	0.15	4.31	13	536	-0.07	1.55	18	495	0.06	2.27
n30-5	9	2	750	0.13	2.81	12	2	732	0.41	4.38	13	495	0.40	1.69	18	492	0.34	1.97

Table 1: Results for *Class I* instances with $n + 2 \in \{10, 15, 20, 25, 30\}$ with different T values

name	D	Sol	g	t	name	D	Sol	g	t
n50-1	1	608	0.33	3.71	n50-9	1	607	0.66	3.60
n50-2	1	591	-0.17	3.48	n50-10	1	627	5.03	3.84
n50-3	1	586	0.86	3.98	n50-11	1	601	1.52	3.667
n50-4	1	597	0.67	3.60	n50-12	1	628	0.32	3.69
n50-5	2	757	23.49	10.21	n50-13	2	786	1.03	10.29
n50-6	1	585	0.17	3.60	n50-14	1	618	0.82	3.88
n50-7	1	550	0.36	3.53	n50-15	2	813	0.49	9.28
n50-8	1	591	0.51	3.62	n50-16	1	615	0.33	3.68

Table 2: Results for Class II instances with $n + 2 = 50$ with $T = 18$

as columns. Here, the calculate start driver method is almost always able to find the optimal number of drivers, reducing the running time significantly. However, the running times for T_D for which there is a single driver necessary. This becomes even more clear in Table 5.1, where even with the elimination multiple numbers of drivers, the running time clearly exceeds the amount listed in Domínguez-Martín et al. (2023), by an approximate factor of six. With these instance sizes, small inefficiencies are also highlighted, but as explained before this usually does not result in a difference this large. However, this heuristic often finds better optimal solutions than Domínguez-Martín et al. (2023), especially for tight values of T.

It is difficult to explain why the optimal solutions are so much better than those in Domínguez-Martín et al. (2023), but after verification of our solution to *n800-2* with a T of 85 in Microsoft Excel, all driver routes were feasible and the cost function was also calculated correctly. There is no knowing exactly why these solution differ so much, as it is very unlikely that an improvement of 20% is due to rounding errors. Since there were no optimal routes presented in Domínguez-Martín et al. (2023), there is also no way to verify any small details that might impact the solution. Furthermore, there are some parts of the methodology left up to interpretation by the authors, possibly causing these differences.

A peculiar finding is that in *n800-2*, there are 33 pairs of customers that are located at exactly the same location as another customer, having the same x and y coordinates. Although this is possible when randomly generating 800 points on a [100x100] grid. In our solution, these points always follow each other in an optimal route, and although nothing is said in Domínguez-Martín et al. (2023) about this occurrence, there is a possibility that a slight difference in handling of these points causes there to be a such a large gap in solution values. In Table 5.1, the number of

name	T_A	sol	g	t	T_B	D	sol	g	t	T_C	sol	g	t	T_D	D	sol	g	t
n100-1	15	1087	-3.81	4.11	20	2	886	0.45	3.49	30	883	0.46	2.38	40	1	741	0.41	2.53
n100-2	15	1230	-1.28	3.66	20	2	969	1.04	3.27	30	960	0.52	2.62	40	1	788	0.13	2.51
n100-3	15	1046	-16.12	4.58	20	2	889	0.23	3.74	30	848	0.71	2.97	40	1	767	0.26	2.81
n100-4	15	1168	-7.59	3.82	20	2	938	0.21	3.28	30	931	0.76	2.56	40	1	764	0.79	2.34
n100-5	15	1139	-16.00	4.42	20	2	945	-10.34	3.49	30	915	0.55	3.15	40	1	786	0.51	2.93
n200-1	25	1425	-17.20	17.25	35	2	1255	-6.97	13.95	50	1192	0.76	12.75	60	1	1143	2.05	8.72
n200-2	25	1457	-21.88	17.22	35	2	1280	-10.11	14.48	50	1203	-0.74	13.26	60	1	1134	-0.44	8.75
n200-3	25	1391	-10.83	18.58	35	2	1231	-1.52	14.88	50	1185	-0.67	11.72	60	1	1163	0.61	8.60
n200-4	25	1414	-5.92	15.26	35	2	1216	-0.25	13.02	50	1211	-0.33	9.57	60	1	1138	-0.61	8.62
n200-5	25	1375	-11.52	16.99	35	2	1222	-0.16	14.89	50	1200	-0.41	11.39	60	1	1142	-0.95	8.74
n300-1	35	1622	-10.68	42.96	45	6	1587	-2.52	35.42	70	1441	0.35	29.20	90	1	1376	0.07	22.85
n300-2	35	1732	-20.88	42.70	45	6	1658	-15.41	39.34	70	1417	0.28	31.89	90	1	1336	-0.30	21.88
n300-3	35	1674	-8.07	36.33	45	2	1598	1.27	14.81	70	1430	0.99	22.43	90	1	1336	0.15	28.08
n300-4	35	1663	-11.64	43.57	45	3	1616	-9.06	38.48	70	1452	-0.48	32.99	90	1	1406	-0.50	25.48
n300-5	35	1567	-16.20	51.29	45	3	1513	-2.13	40.33	70	1377	0.15	33.86	90	1	1339	-0.37	26.84
n400-1	45	1815	-18.57	88.11	55	3	1782	-8.99	76.65	90	1602	0.69	62.13	115	1	1555	-1.14	38.48
n400-2	45	1785	-11.06	82.85	55	3	1740	-9.42	73.16	90	1590	0.19	58.98	115	1	1547	0.72	40.98
n400-3	45	1823	-10.90	89.47	55	3	1759	-1.95	73.93	90	1616	-0.62	58.56	115	1	1579	-0.32	40.93
n400-4	45	2000	-25.48	83.00	55	3	1894	-21.51	84.39	90	1690	-9.33	70.94	115	1	1564	-1.39	43.11
n400-5	45	1942	-19.45	87.97	55	3	1883	-12.66	82.09	90	1671	0.00	63.12	115	1	1596	0.38	41.70
n500-1	55	2016	-4.36	120.92	65	3	1990	-0.60	98.95	110	1795	0.45	73.57	140	1	1755	0.98	65.98
n500-2	55	1999	-19.10	143.12	65	3	1996	-12.84	127.47	110	1819	0.06	101.85	140	2	1814	0.44	88.51
n500-3	55	2126	-22.27	160.00	65	3	2016	-22.07	140.75	110	1827	-0.05	110.50	140	2	1799	0.06	97.07
n500-4	55	2198	-21.47	167.89	65	3	2084	-20.03	152.51	110	1847	-2.17	117.02	140	2	1820	0.33	103.51
n500-5	55	1993	-12.97	144.52	65	3	1942	0.21	125.62	110	1805	0.11	99.48	140	2	1804	0.28	86.40

Table 3: Results for Class III instances with $n + 2 \in \{100, 200, 300, 400, 500\}$ with different T values

drivers is not included in the table, as all instances have 3 drivers for T_A, 2 drivers for T_B & T_C and 1 driver for T_D. Here, once again, running times are slightly slower than in the benchmark, sometimes even exceeding the set time limit, but they do result in significantly better solution values.

name	T_A	Sol	g	t	T_B	Sol	g	t	T_C	Sol	g	t	T_D	Sol	g	t
n600-1	65	2305	-13.51	218	115	1989	0.00	177	155	1975	0.56	153	205	1921	0.05	151
n600-2	65	2232	-20.40	189	115	1993	-0.50	138	155	1977	0.15	114	205	1910	0.10	116
n600-3	65	2227	-12.08	183	115	1994	0.91	133	155	1982	0.71	106	205	1926	0.36	117
n600-4	65	2128	-17.93	201	115	1942	1.04	144	155	1931	0.42	113	205	1881	0.59	132
n600-5	65	2206	-22.19	217	115	2010	-1.03	181	155	1997	0.05	152	205	1959	-0.15	135
n800-1	85	2415	-7.96	508	135	2271	0.80	420	175	2258	0.94	339	225	2234	0.59	372
n800-2	85	2644	-20.29	607	135	2398	-7.09	496	175	2283	-0.22	434	225	2245	1.22	295
n800-3	85	2411	-1.07	436	135	2267	0.35	349	175	2268	0.84	280	225	2232	1.55	379
n800-4	85	2439	-9.06	454	135	2283	0.88	389	175	2266	0.94	327	225	2229	0.32	354
n800-5	85	2507	-18.87	594	135	2305	-0.60	497	175	2245	0.54	416	225	2202	0.59	326
n1000-1	105	2755	-15.26	968	155	2579	0.98	904	205	2514	0.36	702	275	2495	1.46	497
n1000-2	105	2859	-13.39	1081	155	2666	-8.23	986	205	2554	2.20	803	275	2523	1.00	488
n1000-3	105	2781	-15.98	1088	155	2594	-6.29	964	205	2490	1.80	827	275	2447	1.03	490
n1000-4	105	2644	-10.98	961	155	2499	1.59	869	205	2496	2.51	728	275	2466	2.79	536
n1000-5	105	2694	-10.41	1007	155	2513	2.40	872	205	2495	1.63	737	275	2460	1.44	537

Table 4: Results for *Class III* instances with $n + 2 \in \{600, 800, 1000\}$ with different T values

5.1.1 Randomness

One of the biggest strengths of this heuristic is a Monte-Carlo-like approach, where each iteration is a random sample which is repeated until numerical results are found. In this method, only the optimal result from the samples is retained, focusing on finding the best solution rather than analyzing the entire distribution. This kind of stochastic optimisation counters one of the biggest downsides to the heuristics the algorithm employs, which is that the local search often ends in local minima instead of global minima, and the local minimum it ends up in is highly dependent on the starting point. By randomly sampling over many starting points, this downside is countered, but it is also the driving factor behind this heuristic.

To illustrate the effect this randomness can have, we have run the algorithm 2000 times on two separate sets of customers, namely n15-5 with a T of 7, and n25-2 with a T of 11. When running the same file multiple times, it is clear what effect the randomness has on the outcome of the optimal solution. The optimal solution found in both cases differed only slightly between runs, although the iteration in which the optimal solution was found did. For a set of customers with a clear local minimum that was easily found, the average number of iterations before the optimal solution was found was 402, while for a set of customers with loads of local minima, the iteration in which the optimal solution was found looks nearly uniformly distributed with a mean of 43166, ranging from iteration 20 to 99956. This shows that several sets of customers are greatly dependent on the randomness and the number of iterations to counter the downside of a local search ending in a local minimum. Figure 2 shows the iteration in which the optimal solution was found. It is important to note the scale on the y-axis do not match, as when *n15-5* is scaled to the same scale as *n25-2*, there is hardly any variation visible. For *n15-5*, the optimal solution is always found in the first 3500 iterations, and in 75% of the runs, the optimal solution is found before the 600$^{\text{th}}$ iteration.

In comparison, the iteration in which the optimal solution for *n25-2* is found looks uniformly distributed. Here, the optimal solution being found is highly dependent on the starting point. Furthermore, in one of the runs the optimal solution was found in the 99,956$^{\text{th}}$ iteration. So there is a likelihood the optimal solution is never found in the 100,000 iterations. However, in this case there are likely optimal solutions found that are only slightly worse, often resulting in the same value when rounded to an integer. This can also be the result of the tightness of T. Domínguez-Martín et al. (2023) also noted this when determining the ideal number of iterations, stating that the additional improvement in solution value is not compensated by the increase in running time.

5.2 Capacity Constraint

T	Q	nDr	Sol	T	Q	nDr	Sol	T	Q	nDr	Sol	T	Q	nDr	Sol
10	5	1	383	**6**	**4**	1	533	10	**4**	4	533	**5**	3	1	611
10	6	1	383	**6**	5	1	533	6	**3**	4	578	**5**	4	1	611
7	6	1	384	**6**	6	1	533	7	**3**	8	578	**5**	5	1	611
7	5	1	384	7	**4**	1	533	10	**3**	8	578	**5**	6	1	611

Table 5: n10-4 tested for different values of T and Q with the restricting value in bold

In Table 5.1 it is clear that the feasible region for certain values of T is rather small. It

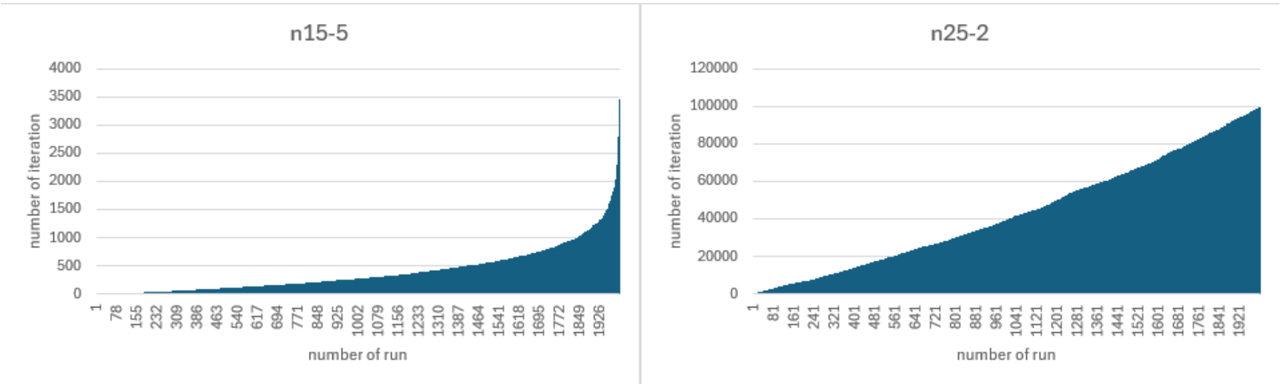

Figure 2: Sorted distribution of the iteration in which the optimal solution was found for *n15-5* and *n20-2*

is therefore important to explore what values of T and Q are interesting to explore. In Table 5.2 *n10-4* is tested for good values for $T = \{5, 6, 7, 10\}$ and $Q = \{3, 4, 5, 6\}$ with the restricting value in bold. q_i^l is assumed to be one for each customer, so from on a single customers demand will be denoted by q. $Q = 2$ was excluded, as that would ensure each vehicle could only serve a single customer besides the exchange location. Q is here defined as the number of customers visited, including the exchange location but excluding the depots. It is notable that $Q = \{5, 6\}$ is never the restricting factor for *n10-4*. Also, for $T = 5$, the duration constraint is so tight that Q also never becomes the restricting factor. $Q = \{3, 4\}$ can become the constraining factor when there is no value of T more constraining. In this particular set, it is also clear that the optimum reached when $T = 6$ and $Q = 4$ are the restraining factors are equal. This mainly holds for small instances, where there is a limited number of possible routes. For larger instances it is uncommon for both T and Q to be the constraining factor.

Nonetheless, Table 5.2 shows that there are two values of interest for T and three for Q. However, the effect for the three values of Q is similar for each T. For $T = 7$ the values $Q = \{3, 4, 6\}$ are of interest, while for $T = 6$ the values $Q = \{3, 4, 5\}$. The effect $Q = 5$ has on the optimal solution is similar to the effect of $Q = 4$. This implies that for each T there are three values of interest, one where T is the constraining factor and Q is loose and two where Q is the constraining factor, one tighter and one looser constraining factor. When T is too tight it is the only constraining factor, while if T is very loose, there is a large difference between different values in Q. To strike a balance, the value of T is either T_B or T_C. Several instances are tested for one value of T for the values Q_A, Q_B and Q_C, with Q_A being the tightest and Q_C the loosest value. For the smaller instances of *Class I*, T_B is tested, while for several instances of *Class II&III* are tested for T_C. This is also partly due to the values for parameters chosen. q and $t_{i,j}$ often have similar values. Adding heterogeneity in the customer demands might shift this, as well as defining $t_{i,j}$ differently. However, $t_{i,j}$ is defined in such a way that customers who are farther away are not considered extreme outliers, which is especially important when a lot of the customers are clustered together.

In Table 6 several values of Q were tested with the same value for T, one loose value which results in the same optimal solution value as the heuristic without Q, one somewhat tight value, restricting the optimal solution, and one very tight value for Q. The tightest value of Q

name	T	Sol	D	Q_A	D	Sol	t	g	Q_B	D	Sol	t	g	Q_C	D	Sol	t	g
n10-1	6	654	2	3	3	844	0.15	-13.65	4	2	654	0.11	-38.43	7	2	654	0.20	16.46
n10-2	6	293	1	3	3	685	0.12	84.95	4	2	486	0.10	46.43	7	1	293	0.07	3.57
n10-3	6	646	2	3	3	910	0.11	-31.35	4	2	646	0.11	-34.17	7	2	646	0.19	16.82
n10-4	6	533	2	3	3	728	0.10	-36.46	4	2	533	0.10	-39.09	7	2	533	0.17	4.53
n10-5	6	366	1	3	3	806	0.10	22.86	4	2	565	0.10	22.24	7	1	366	0.07	-14.49
n15-1	8	349	1	6	3	619	0.14	27.41	10	2	417	0.16	49.72	12	1	349	0.15	35.98
n15-2	8	412	1	6	3	1095	0.13	36.78	10	2	736	0.14	44.75	12	1	412	0.11	7.80
n15-3	8	441	1	6	3	940	0.14	34.55	10	2	636	0.15	44.68	12	1	441	0.10	3.49
n15-4	8	715	2	6	3	972	0.15	-50.30	10	2	715	0.17	-42.33	12	2	715	0.33	8.25
n15-5	8	749	2	6	3	1065	0.14	-43.35	10	2	749	0.17	-30.18	12	2	749	0.29	16.07
n20-1	9	844	2	7	3	1222	0.29	-30.03	11	2	844	0.33	-21.45	13	2	844	0.27	-34.48
n20-2	9	613	2	7	3	878	0.30	-25.98	11	2	613	0.26	-35.77	13	2	613	0.29	-29.96
n20-3	9	757	2	7	3	1015	0.30	-30.36	11	2	757	0.30	-31.01	13	2	757	0.31	-29.79
n20-4	9	582	2	7	3	774	0.32	-14.65	11	2	587	0.25	-33.22	13	2	582	0.26	-30.84
n20-5	9	538	2	7	3	741	0.29	-35.22	11	2	538	0.31	-29.14	13	2	538	0.31	-30.31
n25-1	10	708	2	8	3	963	0.44	-13.43	13	2	708	0.36	-29.29	15	2	708	0.32	-37.22
n25-2	10	699	2	8	3	1086	0.43	-29.38	13	2	699	0.44	-27.88	15	2	699	0.43	-29.38
n25-3	10	553	2	8	3	767	0.43	-14.17	13	2	553	0.34	-33.27	15	2	553	0.34	-32.17
n25-4	10	681	2	8	3	931	0.43	-18.61	13	2	681	0.35	-33.61	15	2	681	0.36	-31.59
n25-5	10	692	2	8	3	977	0.49	-10.73	13	2	692	0.37	-33.23	15	2	692	0.35	-35.74
n30-1	12	816	2	10	3	1082	0.59	-23.51	13	2	816	0.56	-27.97	13	2	816	0.56	-27.97
n30-2	12	827	2	10	3	1152	0.58	-21.73	13	2	837	0.52	-28.84	13	2	837	0.52	-28.84
n30-3	12	797	2	10	3	1127	0.59	-19.08	13	2	802	0.49	-31.77	13	2	802	0.49	-31.77
n30-4	12	663	2	10	3	888	0.56	-22.38	13	2	663	0.47	-35.21	13	2	663	0.47	-35.21
n30-5	12	732	2	10	3	1011	0.60	-18.12	13	2	735	0.50	-31.26	13	2	735	0.50	-31.26

Table 6: Different values of Q_A, Q_B and Q_C tested for T_B for *Class I*

results in the maximum number of drivers being necessary to find an optimal solution. However, one interesting result is including Q is that the algorithm can become faster when Q is tight, becoming up to 50% as fast (for $Q = 6$ in *n15-4*). This is likely due to the calculate start driver method, which can exclude more number of drivers when Q is tight. In Table 6 it is also clear that when there are more drivers necessary for a feasible solution due to Q, the time necessary increases.

Another observation is how well the algorithm performs both in terms of the optimal solution and the running time is very dependent on not only the parameters chosen, but also on the location of the randomly generated nodes. The values for T and Q for which a stable local minimum is found differs greatly, even within sample sizes.

In Table 7, the number of drivers for Q_C is always 2, and not all instances of *Class II&III* are tested. Furthermore, the algorithm does not necessarily become faster for larger size instances because the algorithm becoming slightly more complex as additional constraints are added, since more options need to be searched. It is notable that a factor of high importance is the number of drivers necessary to reach a feasible solution. As shown in Domínguez-Martín et al. (2023) has a nearly linear relation to the amount of time the algorithm has to run. This can be seen clearly for $n + 2 = 50$, where the original number of drivers ranges from 1 to 2. For the tightest value of Q, the time gap ranges from 14% faster to nearly 200% slower. This is due to two things, first the minimal number of drivers necessary here is 3, which eliminates having to search for a feasible solution with 1 or 2 drivers, which makes the heuristic faster. However, running the multistart loop is a lot less efficient for 3 drivers than when there are fewer, as the local search methods need to search more positions. For the two small gaps of less than 20%, the number of drivers necessary without Q is two, which means the original algorithm was slower to find an

Name	T	Sol	D	Q_A	D	Sol	t	g	Q_B	D	Sol	t	g	Q_C	Sol	t	g
n50-12	18	628	1	12	3	1078	1.75	184.67	16	3	1053	1.36	121.00	20	799	1.17	90.33
n50-13	18	786	2	12	3	1086	1.56	-14.18	16	3	1054	1.28	-29.49	20	786	1.34	-26.57
n50-14	18	618	1	12	3	1030	1.62	151.07	16	3	965	1.42	120.16	20	756	1.21	87.60
n50-15	18	813	2	12	3	1050	1.82	17.54	16	3	1026	1.40	-9.64	20	813	1.17	-24.60
n50-16	18	615	1	12	3	1086	1.75	185.01	16	3	1070	1.30	112.03	20	815	1.02	66.58
n100-1	30	883	2	28	3	1079	4.12	-9.41	40	2	883	3.00	-33.95	55	883	4.48	-1.33
n100-2	30	960	2	28	3	1218	3.94	-18.33	40	2	960	3.12	-35.20	55	960	4.80	-0.31
n100-3	30	848	2	28	3	1016	4.45	-11.84	40	2	848	3.74	-25.94	55	848	5.28	4.58
n100-4	30	931	2	28	3	1155	4.14	-10.34	40	2	936	3.40	-26.37	55	931	4.79	3.62
n100-5	30	915	2	28	3	1100	4.93	-7.67	40	2	918	4.11	-23.15	55	915	5.73	7.28
n300-1	70	1431	2	78	3	1577	40.66	37.44	110	2	1442	36.03	21.82	155	1444	33.66	13.79
n300-2	70	1410	2	78	3	1665	47.87	48.44	110	2	1425	40.05	24.17	155	1420	37.71	16.93
n300-3	70	1428	2	78	3	1636	40.71	78.70	110	2	1426	29.36	28.89	155	1430	25.28	10.97
n300-4	70	1461	2	78	3	1599	42.90	47.51	110	2	1455	35.40	21.71	155	1455	33.11	13.84
n300-5	70	1378	2	78	3	1503	40.90	31.11	110	2	1387	36.90	18.30	155	1381	35.71	14.49
n500-1	110	1795	2	130	2	1828	120.79	68.90	170	2	1800	93.07	30.15	250	1792	79.14	10.66
n500-2	110	1825	2	130	2	1919	136.08	38.12	170	2	1820	120.50	22.30	250	1822	112.24	13.92
n500-3	110	1825	2	130	2	1951	148.70	40.25	170	2	1888	132.80	25.25	250	1825	123.26	16.25
n500-4	110	1852	2	130	2	1980	156.34	41.68	170	2	1918	136.38	23.60	250	1870	132.22	19.83
n500-5	110	1800	2	130	2	1879	134.70	40.54	170	2	1811	119.34	24.51	250	1807	107.80	12.47

Table 7: Different values for Q_A, Q_B and Q_C for T_C of *Class II & III*

optimal solution, as can be seen in table 5.1, where especially n50-13 was quite slow. Adding the complexity of Q but eliminating several numbers of drivers. The largest gaps can also be seen with a size of 50, which is mainly due to the original heuristic being fast if there is only a single driver, rather than the algorithm including Q being slow.

It is also clear that as Q loosens, the algorithm becomes considerably quicker. This implies that the algorithm is robust in the amount of time adding Q takes, but choosing a tight Q results in not only higher optimal solutions and more drivers to be utilized, which leads to a higher running time, but also potentially saving up to 200,000 iterations when a certain number of drivers do not need to be searched for an optimal solution by the algorithm. This same phenomenon can be seen for n10-2, where it is much more that the algorithm without Q was exceptionally fast, than that the algorithm with Q is slow when compared to other times of the same size. Overall, the algorithm is only 50% slower than the original when the original algorithm was considerably faster than the rest of its size, or when the Q is really tight, so the algorithm is quite robust to the addition of capacity constraints.

5.3 Time windows

The time window constraints are defined to be 40% of T, and are consistent over all instances, as they are read from a file with a random starting time. The time window constraints are tested for several values of T, not including T_A as it is already a rather tight constraint. Furthermore, the $maxDrivers$ is increased to 5 to ensure feasibility.

Including time windows highlights a weakness of this algorithm, as predetermined time windows do not translate well to random allocation. Here, the calculate start drivers method is also not as easily effective. This causes the algorithm to search many more options before even potentially finding an optimal solution. Furthermore, searching for the best insertion position and route is a lot more computationally expensive. Not only does the feasibility of the insertion location need to be evaluated, so does the feasibility of each node after the insertion position.

name	T_B	D	Sol	t	g	T_C	D	Sol	t	g	T_D	D	Sol	t	g
n10-1	7	2	602	0.7	56.79	8	2	602	0.74	33.75	10	2	411	0.67	57.75
n10-2	6	2	486	0.91	131.38	7	2	486	0.76	95.13	10	1	293	0.57	30.82
n10-3	6	3	646	1.05	9.61	7	2	617	0.85	80.94	10	1	385	0.59	35.02
n10-4	6	2	534	0.65	-31.82	7	2	533	0.7	74.44	10	2	383	0.62	59.23
n10-5	6	2	594	0.7	41.84	7	2	565	0.72	43.65	10	1	355	0.63	50.6
n15-1	8	2	441	1.4	117.91	9	2	415	1.29	38.43	12	2	407	1.58	117.03
n15-2	8	3	744	1.56	164.41	9	2	736	1.37	123.45	12	2	413	1.49	137.36
n15-3	8	3	636	1.84	205.81	9	2	480	1.63	156.31	12	2	387	1.56	152.67
n15-4	8	3	791	1.41	-22.16	9	3	710	1.3	107.15	12	2	568	1.25	59.46
n15-5	8	3	774	1.63	9.93	9	3	753	1.67	150.83	12	2	749	1.56	135.7
n20-1	9	3	1075	3.11	24.86	10	3	943	3.07	221.57	14	2	788	2.95	134.07
n20-2	9	3	691	2.96	20.49	10	3	613	2.97	212.74	14	2	593	2.83	178.35
n20-3	9	3	823	2.73	4.08	10	3	750	2.84	217.19	14	2	708	2.95	135.38
n20-4	9	3	646	2.7	18.61	10	3	582	2.58	175.75	14	2	565	2.5	184.85
n20-5	9	3	637	3.01	13.44	10	2	573	2.86	172.38	14	2	499	2.88	113.79
n25-1	10	3	806	4.76	55.87	11	3	750	4.74	278.31	16	2	646	4.47	249.77
n25-2	10	3	907	4.8	30.8	11	3	808	4.78	296.76	16	2	614	4.74	179.82
n25-3	10	3	689	4.04	33.57	11	3	631	3.81	205.62	16	2	575	3.4	151.33
n25-4	10	3	729	4.47	40.82	11	3	701	4.55	279.25	16	2	571	3.93	209.37
n25-5	10	3	872	4.69	41.69	11	3	753	4.73	274.82	16	2	610	4.51	240
n30-1	12	3	1087	7.08	52.54	13	3	960	7.21	338.42	18	2	803	7.17	180.25
n30-2	12	3	1147	7.59	71.82	13	3	985	7.5	375.33	18	2	849	7.36	301.64
n30-3	12	3	946	6.5	49.72	13	3	847	6.54	244.13	18	2	600	5.85	214.81
n30-4	12	3	877	6.06	40.59	13	3	806	5.8	274.37	18	2	692	5.3	133.6
n30-5	12	3	873	7.48	71.02	13	3	766	7.54	345.71	18	2	736	7.57	285

Table 8: Different values of T tested with time window constraints of 40% tested for *Class I*

Name	**T**	**D**	**Sol**	**t**	**g**	**Name**	**T**	**D**	**Sol**	**t**	**g**
n50-1	30	2	857	430.33	2088	n50-9	30	2	835	512.5	2224
n50-2	30	2	839	554.64	2333	n50-10	30	2	887	511.29	2076
n50-3	30	2	869	438.94	1946	n50-11	30	2	856	436.06	2119
n50-4	30	2	881	413.19	2137	n50-12	30	2	945	496.83	2155
n50-5	30	2	888	113.35	420	n50-13	30	2	849	122.04	403
n50-6	30	2	926	489.44	2207	n50-14	30	2	848	500.36	2049
n50-7	30	2	838	438.79	2208	n50-15	30	2	907	103.07	447
n50-8	30	2	892	533.51	2228	n50-16	30	2	844	479.23	2146

Table 9: Different values for T including time window constraints of 40% tested for *Class II*

This is the case in both the construct drivers routes method as the local search method. The strength of the simple 2 opt method is also decreased, as now also within routes several criteria need to be checked. Furthermore, the 2 opt method swaps the direction of all arcs between the two swapped arc, likely rendering the solution infeasible. There are also few other heuristics that are both as simple as the 2opt local search and serve the same purpose. In Table 8, the gap calculated over the time often exceeds 20%, and even for small size instances and loose values of T, this results in a significantly increased running time.

In Table 9, the same phenomenon is encountered, but the time gap now runs to up to 200%. Because of this large time increase, *Class III* was only tested for sizes $n + 2 \in \{100, 200, 300\}$

In Table 10, the original correction for possible multi-threads has been applied, as for *Class II* the time limit of 600 seconds from the benchmark has nearly been surpassed. Still, the gaps reach up to 2000% for the sample size of 300. These results show that employing random sampling over a simple heuristic is not efficient, and that the heuristic is not robust to the addition of time windows. Even when the time windows are quite large and are scaled up as T increases, the running time increases exponentially.

Name	T	D	Sol	t	g	D	Sol	t	g	T	D	Sol	t	g
n100-1	20	4	1874	15.15	342	3	1666	17.23	279	40	2	1250	19.28	656
n100-2	20	4	2002	15.03	355	3	1651	17.59	265	40	2	1387	19.58	670
n100-3	20	4	1861	15.71	328	3	1571	16.04	218	40	2	1311	17.51	513
n100-4	20	4	1997	15.22	368	3	1732	17.61	281	40	2	1385	19.78	747
n100-5	20	4	1966	15.48	340	3	1665	17.79	233	40	2	1333	20.36	587
n200-1	35	4	2973	99.46	599	3	2510	131.08	905	60	3	2351	111.08	1161
n200-2	35	4	2982	99.57	575	3	2471	134.46	893	60	3	2343	147.18	1588
n200-3	35	5	3023	95.98	535	3	2465	128.78	975	60	3	2355	113.38	1206
n200-4	35	4	3070	98.47	637	3	2580	133.65	1277	60	3	2462	142.3	1559
n200-5	35	4	2947	104.27	590	3	2531	139.26	1099	60	3	2236	159.41	1726
n300-1	45	5	3836	323.96	1863	3	3073	482.56	1531	90	3	2767	602.89	2532
n300-2	45	5	4054	315.9	1987	3	3057	458.03	1320	90	3	2855	575.95	2532
n300-3	45	5	4147	316.05	2039	3	3197	448.71	1870	90	3	2933	540.24	1892
n300-4	45	5	4104	324.55	528	3	3201	487.1	1575	90	3	2927	658	2854
n300-5	45	5	3853	327.28	2085	3	3055	504.88	1519	90	3	2771	734.36	3256

Table 10: Different values of T including time window constraints of 40% for $n + 2 \in \{100, 200, 300\}$

6 Conclusion

In this paper, the robustness of a multi-start heuristic to a capacity and time window constraint has been researched on the Driver and Vehicle Routing Problem. Testing this gives insight in how efficient applying this heuristic in real-life might be. The heuristic is efficient at tackling the DVRP, and is also robust to a capacity constraint, as simple heuristics give good results and there is not too much added complexity. Time windows render a lot of simple heuristic ineffective, diminishing the strength of this algorithm. The running time of the algorithm increases up to 2000% for the largest two classes of instance sizes. Other algorithms might tackle it better for the DVRP, but the approach of using low time complexity heuristics a multitude of times is not effective when incorporating time windows. The strength of the multistart heuristic really lies in repeating a random sampling of simple heuristics, but therein also lies its weakness. An optimal solution is found after a certain number of tries, and often times running the algorithm results in a different outcome.

6.1 Discussion

Both the DVRP as well as the multi-start heuristic have not been studied widely in literature and there are a lot of interesting factors to explore even further, also in the name of realism. Some of these things are introducing heterogeneity to both the customer demand and the length of the time windows. Each of these scenario's could have very useful applications, especially for the DVRP.

The multi-start heuristic is slightly different. The strength of this algorithm is repeating a random sampling of simple heuristics that can be performed fast, which cannot be applied as well to certain additional constraints. Introducing time windows could be more efficient if a more complex algorithm is adapted, as the addition of time window constraints reduces the effectiveness of most simple heuristics like the cheapest insertion strategy or several variants of a local search method. It is not strange that the original heuristic employed some of the simplest heuristics, as being slightly faster is more useful when each heuristic is applied at least 100,000 times.

One of the other downsides of this algorithm is that when a constraint is tight, there is limited room for the heuristics to perform well. This was addressed in our local search method, but this makes the random sampling even more important to the algorithm. When the heuristics within the multi-start loop cannot attest to a large improvement, the starting position becomes all the more significant. The fact that the tightness of the constraints can limit the algorithm quite severely also implies that the entire problem is also very dependent on the values of the parameters. In the results it can be seen there are some instances with the same size that have a wide range of optimal solutions found. For each instance, the parameter that impacts the solution most differs, while they are all compared by the same standard. Especially with small instances, it occurred that no feasible solution could be found for the tightest time limit, so the time limit was increased.

Of course many problems are dependent on the value of parameters, but the large differences within an instance size makes the instances hard to compare to each other in terms of solution value, which also makes them more difficult to compare in terms of running time.

However, nowadays computers are fast and will only become faster, so including a simulation like element to a heuristic can be greatly beneficial, but it is important to still think about efficiency when using a lot of computing power, because running redundant iterations is not an efficient use of computing power, and also does not translate well into even larger problems.

References

Balakrishnan, N. (1993). Simple heuristics for the vehicle routeing problem with soft time windows. *Journal of the Operational Research Society*, *44*(3), 279–287.

Barreto, S., Ferreira, C., Paixao, J. & Santos, B. S. (2007). Using clustering analysis in a capacitated location-routing problem. *European journal of operational research*, *179*(3), 968–977.

Bräysy, O. & Gendreau, M. (2005). Vehicle routing problem with time windows, part ii: Metaheuristics. *Transportation science*, *39*(1), 119–139.

Domínguez-Martín, B., Rodríguez-Martín, I. & Salazar-González, J.-J. (2018a). The driver and vehicle routing problem. *Computers & Operations Research*, *92*, 56–64.

Domínguez-Martín, B., Rodríguez-Martín, I. & Salazar-González, J.-J. (2018b). A heuristic approach to the driver and vehicle routing problem. In *International conference on computational logistics* (pp. 295–305).

Domínguez-Martín, B., Rodríguez-Martín, I. & Salazar-González, J.-J. (2023). An efficient multistart heuristic for the driver and vehicle routing problem. *Computers & Operations Research*, *150*, 106076.

El-Sherbeny, N. A. (2010). Vehicle routing with time windows: An overview of exact, heuristic and metaheuristic methods. *Journal of King Saud University-Science*, *22*(3), 123–131.

Gocken, T. & Yaktubay, M. (2019). Comparison of different clustering algorithms via genetic algorithm for vrptw.

Lau, H. C., Sim, M. & Teo, K. M. (2003). Vehicle routing problem with time windows and a limited number of vehicles. *European journal of operational research*, *148*(3), 559–569.

Lysgaard, J. (1997). Clarke & wright's savings algorithm. *Department of Management Science and Logistics, The Aarhus School of Business*, *44*, 1–7.

Pino, R., Villanueva, V., Martinez, C., Lozano, J., Del Pino, B. & Andrés, C. (2011). Heuristic solutions to the vehicle routing problem with capacity constraints. *Proceedings of ICAI-Worldcomp*, *11*, 634–640.

Salazar-González, J.-J. (2014). Approaches to solve the fleet-assignment, aircraft-routing, crew-pairing and crew-rostering problems of a regional carrier. *Omega*, *43*, 71–82.

Vidal, e. a. (2015). Hybrid metaheuristics for the clustered vehicle routing problem. *Computers & Operations Research*, *58*, 87–99.

Žunić, E., Đonko, D., Šupić, H. & Delalić, S. (2020). Cluster-based approach for successful solving real-world vehicle routing problems. In *2020 15th conference on computer science and information systems (fedcsis)* (pp. 619–626).

A Programming code

The constructor MainDVRP7R initializes an instance by reading the file containing the nodes and calculating parameters that do not change during a run and initializing all variables. The constructDrivers() method initializes each route and randomly adds a node according to the cheapest insertion criterion until all nodes are added to a different route. The localSearch() method randomly selects a node which is placed in a different route according to the cheapest insertion criteria. This is repeated until all nodes have been placed in a different route. The twoOpt() method deletes two arcs and replaces them with two different arcs within a node if it improves the solution. This is repeated for all combinations of two arcs. The checkFeasibility() method checks whether there is a driver route which exceeds the constraint. The constructVehicleRoutes() method constructs vehicle routes from the best found driver routes The calculateStartDrivers() method calculates a good starting value for the parameter nDrivers The runFile() method runs the algorithm for a single instance with fixed values for all parameters. These methods are slightly altered for the addition of time windows and capacity constraints and can be found in the accompanying zip file.

```java
import java.io.BufferedReader;
import java.io.BufferedWriter;
public MainDVRP7R(String filename, int Toverwrite, int startDrivers) throws
    IOException {
        // Read file
        nDrivers = startDrivers;
        T = Toverwrite;
        FileInputStream fstream = new FileInputStream(filename);
        BufferedReader br = new BufferedReader(new InputStreamReader(
            fstream));
        this.name = br.readLine(); // NAME :
        String line = br.readLine(); // TYPE : CVRP
        line = br.readLine(); // EDGE_WEIGHT_TYPE : EUC_2D
        line = br.readLine(); // DIMENSION :
        String dimension = line.substring(11);
        dimension = dimension.replaceAll("\\s+", "");
        this.dim = Integer.parseInt(dimension);
        line = br.readLine(); // NODE_COORD_SECTION
        int[][] nodes = new int[dim][2];
        for (int i = 0; i < dim; i++) {
            line = br.readLine();
            String[] temp = line.split("   ");
            temp[2] = temp[2].replaceAll("\\s+", "");
            nodes[i][0] = Integer.parseInt(temp[1]);
            nodes[i][1] = Integer.parseInt(temp[2]);
        }
        br.close();
        // define distance
        this.exchangeLocation = dim - 2;
        this.depot1 = 0;
        this.depot2 = dim - 1;
        this.distance = new double[dim][dim];
        this.costs = new double[dim][dim];
        this.totalCost = 0;
        for (int i = 0; i < dim; i++) {
            for (int j = i; j < dim; j++) {
                if (i == j) {
                    distance[i][j] = Double.POSITIVE_INFINITY;
                    costs[i][j] = 0;

                } else {
                    costs[i][j] = Math
                            .sqrt(Math.pow(nodes[i][0] - nodes[j][0], 2) +
                               Math.pow(nodes[i][1] - nodes[j][1], 2));
                    costs[j][i] = costs[i][j];
                    distance[i][j] = (costs[i][j] / 60) + 0.5;
                    distance[j][i] = distance[i][j];
                }
            }
        }
        this.driverNodes = new ArrayList[maxDrivers];
        this.driverLength = new double[maxDrivers];
    }
    private void constructDrivers() {
        for (int i = 0; i < maxDrivers - 4; i = i + 2) {

            driverNodes[i] = new ArrayList<Integer>();
            driverNodes[i].add(0);
            driverNodes[i].add(dim - 2);
            driverNodes[i].add(0);
            driverLength[i] = distance[0][dim - 2] * 2;

            driverNodes[i + 1] = new ArrayList<Integer>();
            driverNodes[i + 1].add(dim - 1);
            driverNodes[i + 1].add(dim - 2);
            driverNodes[i + 1].add(dim - 1);
            driverLength[i + 1] = distance[dim - 1][dim - 2] * 2;
        }
        //randomly order the nodes
        random = new int[dim - 3];
        for (int i = 1; i < dim - 2; i++) {
            random[i - 1] = i;
        }

        Random rand = new Random();
        for (int i = random.length - 1; i > 0; i--) {
            int j = rand.nextInt(i + 1);
            int temp = random[i];
            random[i] = random[j];
            random[j] = temp;
        }

        for (int i : random) {
            int Route = -1;
            int Position = 1;
            double currentBestDistance = Double.POSITIVE_INFINITY;
            int shortestDriverRoute = 0;
            int shortestPosition = 0;
            double shortestBestDistance = Double.POSITIVE_INFINITY;

            while (Route < 0) {
                for (int r = 0; r < nDrivers; r++) {
                    for (int p = 1; p < driverNodes[r].size(); p++) {
                        int Vorige = driverNodes[r].get(p - 1);
                        int Volgende = driverNodes[r].get(p);
                        double currentDistance = distance[i][Vorige] +
                            distance[i][Volgende] - distance[Volgende][
                            Vorige];
                        if (currentDistance < currentBestDistance) {
                            if (driverLength[r] + currentDistance < T) {
                                Route = r;
                                Position = p;
                                currentBestDistance = currentDistance;
                            } else if (driverLength[r] <= driverLength[
                                shortestDriverRoute] && currentDistance <
                                shortestBestDistance) {
                                shortestDriverRoute = r;
                                shortestPosition = p;
                                shortestBestDistance = currentDistance;
                            }
                        }
                    }

                }
                if (Route < 0) {
                    Route = shortestDriverRoute;
                    Position = shortestPosition;
                    currentBestDistance = shortestBestDistance;
                }

                if (nDrivers > maxDrivers) {
                    System.exit(0);
                }
            }
            driverNodes[Route].add(Position, i);

            int Vorige = driverNodes[Route].get(Position - 1);
            int Volgende = driverNodes[Route].get(Position + 1);

            driverLength[Route] += (distance[i][Vorige] + distance[i][
                Volgende] - distance[Volgende][Vorige]); // currentBestRoute
                ??
        }
    }
private void localSearch(double increasedT) {
        //randomly select order of nodes
        int[] random = new int[dim - 3];
        for (int i = 0, k=0; i < nDrivers; i++) {
            for (int j = 1; j < driverNodes[i].size() - 1; j++) {
                int b = driverNodes[i].get(j);
                if (b != exchangeLocation) {
                    random[k] = 10000 * i + b;
                    k++;
                }
            }
        }

        Random rand = new Random();
        for (int i = random.length - 1; i > 0; i--) {
            int j = rand.nextInt(i + 1);
            int temp = random[i];
            random[i] = random[j];
            random[j] = temp;
        }

        for (int k : random) {
            int i = (int) k / 10000; //Route
            int j = driverNodes[i].indexOf((int) k % 10000); //Customer
                position
```

```java

            int a = driverNodes[i].get(j - 1);
            int b = driverNodes[i].get(j);
            int c = driverNodes[i].get(j + 1);

            if (b == exchangeLocation) {
                continue;
            }

            double lengthRemovej = distance[a][b] + distance[b][c] - distance[a][c];
            //find best place
            int Route = -1;
            int Position = -1;
            double currentBestDistance = Double.POSITIVE_INFINITY;
            for (int r = 0; r < nDrivers; r++) {
                if (r == i) {
                    continue;
                }
                for (int p = 1; p < driverNodes[r].size(); p++) {
                    int Vorige = driverNodes[r].get(p - 1);
                    int Volgende = driverNodes[r].get(p);
                    double currentDistance = distance[b][Vorige] + distance[b][Volgende] - distance[Volgende][Vorige];
                    if (currentDistance < lengthRemovej) {
                        if (currentDistance < currentBestDistance && (driverLength[r] + currentDistance < T + increasedT )) {
                            Route = r;
                            Position = p;
                            currentBestDistance = currentDistance;
                            increasedT = increasedT  * 0.9 ;
                        }
                    }
                }
            }
            // if a better place is found, relocate
            if (Route >= 0) {
                driverNodes[Route].add(Position, driverNodes[i].get(j));
                driverNodes[i].remove(j);

                int Vorige = driverNodes[Route].get(Position - 1);
                int Volgende = driverNodes[Route].get(Position + 1);

                driverLength[Route] += (distance[b][Vorige] + distance[b][Volgende] - distance[Volgende][Vorige]);
                driverLength[i] -= lengthRemovej;
            }
```

```
        }
    }

private void twoOpt() {
        for (int i = 0; i < nDrivers; i++) {
            for (int a = 0; a < driverNodes[i].size() - 2; a++) {
                for (int b = a + 1; b < driverNodes[i].size() - 1; b++) {

                    int a0 = driverNodes[i].get(a);
                    int a1 = driverNodes[i].get(a + 1);
                    int b0 = driverNodes[i].get(b);
                    int b1 = driverNodes[i].get(b + 1);

                    double oldDist = distance[a0][a1] + distance[b0][b1];
                    double newDist = distance[a0][b0] + distance[a1][b1];

                    if (oldDist > newDist) {
                        for (int s = driverNodes[i].indexOf(a1), r =
                            driverNodes[i].indexOf(b0); r - s > 0; s++, r--)
                             {
                            Collections.swap(driverNodes[i], s, r);
                        }
                        driverLength[i] += (newDist - oldDist);
                    }
                }
            }
        }
    }

private static boolean checkFeasibility(MainDVRP7R test) {
        for (int i = 0; i < nDrivers; i++) { // voor elke route
            if (driverLength[i] > T) {
                return false;
            }
        }
        return true;
    }

private void constructVehicleRoutes(String[] fileBestRoutes) {
        this.vehicleNodes = new ArrayList[maxDrivers];
        for (int i = 0; i < nDrivers; i++) { // loop over all routes
            fileBestRoutes[i].replaceAll("[", "");
            fileBestRoutes[i].replaceAll("]", "");
            fileBestRoutes[i].replaceAll(", ", "");
            vehicleNodes[i] = new ArrayList<Integer>();
            int j = 0;
            while (Integer.parseInt(fileBestRoutes[i].substring(j, j+1)) !=
                exchangeLocation) {
```

```
                vehicleNodes[i].add(Integer.parseInt(fileBestRoutes[i].substring(j, j+1)));
                j++;
            }

            if (i % 2 == 0) {
                int k = fileBestRoutes[i + 1].indexOf(exchangeLocation);
                while (Integer.parseInt(fileBestRoutes[i + 1].substring(k, k+1)) != depot2) {
                    vehicleNodes[i].add(Integer.parseInt(fileBestRoutes[i + 1].substring(k, k+1)));
                    k++;
                }
                vehicleNodes[i].add(depot2);
            } else {
                int k = fileBestRoutes[i - 1].indexOf(exchangeLocation);
                while (Integer.parseInt(fileBestRoutes[i - 1].substring(k, k+1)) != depot1) {
                    vehicleNodes[i].add(Integer.parseInt(fileBestRoutes[i - 1].substring(k, k+1)));
                    k++;
                }
                vehicleNodes[i].add(depot1);
            }
        }
    }

    private double calculateOptSol() {
        totalCost = 0;
        for (int i = 0; i < nDrivers; i++) {
            for (int j = 0; j < driverNodes[i].size() - 1; j++) {
                totalCost += costs[driverNodes[i].get(j)][driverNodes[i].get(j + 1)];
            }
        }
        return totalCost;
    }

    private static int calculateStartDrivers(int T, int dim) {
        double nodesPerDriver = (T-1.5)*2;
        if(nodesPerDriver*2>dim) {
            return 2;
        } else if(nodesPerDriver*4>dim) {
            return 4;
        } else {
            return 6;
        }

```

```java
   }
private static void runFile(String filename, int Toverwrite, int
   startDrivers, BufferedWriter writer2) throws IOException {

       int maxIter = 100000;
       double fileBestSol = Double.POSITIVE_INFINITY;
       String fileBestRoutes[] = new String [maxDrivers];

       double [] fileBestDriverLength = driverLength;
       boolean isFeasibleFound = false;
       long startTime = System.currentTimeMillis();
       while(isFeasibleFound == false) {
       MainDVRP7R test = new MainDVRP7R(filename, Toverwrite, startDrivers
          );

       name = name.replace(' ', '_');
       name = name.replace(':', '_');
       name = name.replace('.', '_');
       name = name.substring(9);
       String outputFile = "C:\\Users\\floor\\Documents\\Bachelor\\
          Bachelor jaar 3\\Bachelor scriptie\\Output\\" + name+ T + ".txt"
          ;
       System.out.println(outputFile);
       BufferedWriter writer = new BufferedWriter(new FileWriter(
          outputFile));
       writer.write(name);
       writer.write("\ndepot 1: " + depot1 + " \ndepot 2: " + depot2 + "\
          nexchange point:" + exchangeLocation + "\nT = " + T);

       //main loop starts here
       for (int nIter = 0; nIter < maxIter; nIter++) {
           test.constructDrivers();

           if (maxIter < 200) {
               writer.write("\nNA CONSTRUCT DRIVER");
               for (int i = 0; i < nDrivers; i++) {
                   writer.write("\nRoute" + i + ": " + driverNodes[i].
                      toString());
               }
               for (int i = 0; i < nDrivers; i++) {
                   writer.write("\nDriver Length " + i + ": " +
                      driverLength[i]);
               }
           }

           test.calculateOptSol();

           double bestSol = Double.POSITIVE_INFINITY;
```

```java
            ArrayList<Integer>[] bestDriverNodes = driverNodes;
            double[] bestDriverLength = driverLength;
            double lastOptSol = test.calculateOptSol();

            for (int i = 0; i < 100; i++) {
                if(isFeasibleFound == false && nIter > 80000) {
                    test.localSearch(2);
                } else {
                    test.localSearch(0);
                }
                double newOptSol = test.calculateOptSol();

                if (maxIter < 200) {
                    writer.write("\nNa local search");
                    for (int w = 0; w < nDrivers; w++) {
                        writer.write("\nRoute" + w + ": " + driverNodes[w].toString());
                    }
                    for (int w = 0; w < nDrivers; w++) {
                        writer.write("\nDriver Length " + w + ": " + driverLength[w]);
                    }
                }
                if (newOptSol == lastOptSol) {
                    break ;
                }
                lastOptSol = newOptSol;

            }

            bestSol = Double.POSITIVE_INFINITY;
            bestDriverNodes = driverNodes;
            lastOptSol = test.calculateOptSol();


            for (int i = 0; i < 100; i++) {
                test.twoOpt();
                double newOptSol = test.calculateOptSol();
                    if (maxIter < 200) {
                        writer.write("\nNa 2-opt");
                        for (int w = 0; w < nDrivers; w++) {
                            writer.write("\nRoute" + w + ": " + driverNodes[w].toString());
                        }
                        for (int w = 0; w < nDrivers; w++) {
                            writer.write("\nDriver Length " + w + ": " + driverLength[w]);
                        }
                    }
                    if (newOptSol == lastOptSol) {
                    break ;
                }
                lastOptSol = newOptSol;
            }

            test.calculateOptSol();
            boolean feasible = checkFeasibility(test);
            // Try if shift possible.
            double TotalDriversLenght = 0 ;
            for (int w=0 ; w < nDrivers ; w++){
                TotalDriversLenght += driverLength[w] ;
            }
            if (feasible && totalCost < fileBestSol) {
                fileBestSol = totalCost;

                for(int f = 0; f<nDrivers; f++) {
                    fileBestRoutes[f] = driverNodes[f].toString();
                }

                fileBestDriverLength = Arrays.copyOf(driverLength, nDrivers
                   ) ;
                isFeasibleFound = true;
                bestIter = nIter;
            }
        }
        if(isFeasibleFound == false) {
            startDrivers = startDrivers + 2;
            continue;
        }
        test.constructVehicleRoutes(fileBestRoutes);

        long endTime = System.currentTimeMillis();
        long totalTime = endTime-startTime;

        test.calculateOptSol();

        writer.write("\nBest Routes");
        for (int i = 0; i < nDrivers; i++) {
            writer.write("\nRoute" + i + ": " + fileBestRoutes[i]);
        }
        writer.write ("\nDriver Length: " + Arrays.toString(
           fileBestDriverLength));

        writer.write("\nTotal Best Costs: " + fileBestSol + "\n");

        writer.write("\nTime: " + totalTime);
```

```java

        writer2.write("\nLast" + name + " T: " + T + " nDrivers: " +
           nDrivers + " Total Costs: " + fileBestSol + " Time: " +
           totalTime + " best Iter: " + bestIter);
        writer2.flush();
        writer.close();
        }

    }
```