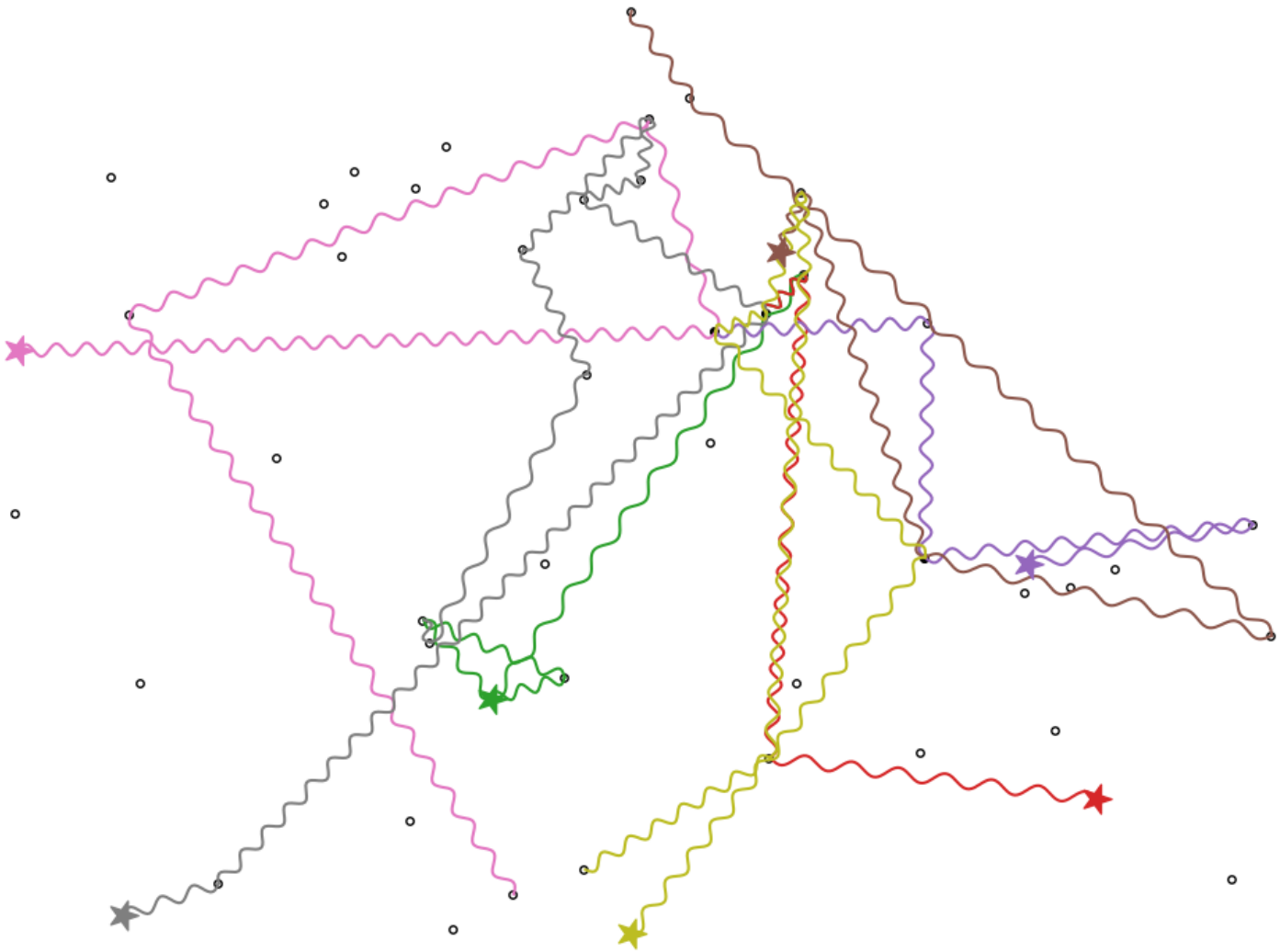
A greedy randomized destroy and repair heuristic for the dial-a-ride problem with transfers

S. J. M. Janssens de Bisthoven



TU Delft

A greedy randomized destroy and repair heuristic for the dial-a-ride problem with transfers

by

S. J. M. Janssens de Bisthoven

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on Tuesday January 21, 2019 at 10:00 AM.

Student number:	4352882	
Project duration:	March, 2019 – January, 2020	
Thesis committee:	Dr. ir. J.T. van Essen,	TU Delft, supervisor
	Prof. dr. ir. K.I. Aardal,	TU Delft
	Dr. C. Kraaikamp,	TU Delft
	J. Pierotti, MSc	TU Delft

An electronic version of this thesis is available at `http://repository.tudelft.nl/`.

TU Delft

Preface

This thesis describes the graduation research finalizing the Master of Science program Applied Mathematics at the Faculty of Electrical Engineering, Mathematics & Computer Science, Delft University of Technology, The Netherlands.
It has been a privilege to work under the supervision of Theresia van Essen and Jacopo Pierotti and I want to personally thank them for their feedback, expert judgment, and knowledge that made this thesis possible. My expressions of gratitude also go to Karen Aardal and Cornelis Kraaikamp for accepting to be part of the thesis committee. Finally, I would like to thank my family and friends for their continuous and much-appreciated support.

S. J. M. Janssens de Bisthoven
Delft, January 2020

Contents

Abstract 1

1 Introduction 3
- 1.1 Context . . . 3
- 1.2 Literature on problems without transfers . . . 4
- 1.3 Literature on problems with transfers . . . 5
- 1.4 Thesis structure . . . 5

2 Problem description 7
- 2.1 Problem formulation . . . 7
- 2.2 Problem characteristics . . . 8

3 Solution approach 9
- 3.1 Definitions . . . 9
  - 3.1.1 Local cost function . . . 11
  - 3.1.2 Multi-dimensional scaling . . . 13
- 3.2 Algorithm outline . . . 16
- 3.3 Selection criterion and cooling scheme . . . 17
- 3.4 Repair phase . . . 18
  - 3.4.1 Vehicle-request matching algorithm . . . 20
  - 3.4.2 Candidate route generation . . . 20
  - 3.4.3 Transfer generation . . . 21
- 3.5 Destroy phase . . . 23

4 Experimental analysis and results 27
- 4.1 Test instances generation . . . 27
- 4.2 MDS accuracy . . . 30
- 4.3 Results . . . 31
  - 4.3.1 Solution visualizations . . . 32
  - 4.3.2 Performance . . . 34
  - 4.3.3 User inconvenience . . . 36
  - 4.3.4 Influence of demand . . . 38
  - 4.3.5 Fleet usage . . . 42
  - 4.3.6 Running time . . . 43
  - 4.3.7 Effect of Δ_{trans} term . . . 44

5 Conclusion and recommendations 45

Bibliography 47

Abstract

Automated vehicles have the potential to create a future in which most cars are shared instead of being individually owned and used. The advantages of vehicle sharing are expected to be multiple: reduced traffic, freed-up parking space, safer trips and a lower environmental impact of traveling.
The dial-a-ride problem with transfers (DARP-T) consists in finding a set of minimum cost routes that satisfies a set of transportation requests. Requests may share a vehicle and may be transferred from one vehicle to another at any node during their journey.
In this thesis, we describe a heuristic solving moderate to large instances of the static heterogeneous DARP-T in which the requests have demanding time constraints. The heuristic builds a solution in a constructive greedy randomized procedure and subsequently improves it with a destroy and repair procedure. The heuristic is tested on instances we randomly generated containing up to 1000 requests traveling between any two of the 100 most populated cities in the Netherlands inside a four-hour timescale. These instances are also solved without transfers to determine the benefits transfers can bring and the user inconvenience they may cause. We also investigate the influence of the demand on the objective function value, look into the influence of the fleet size on the fleet usage, and present some visualizations of the solutions found.

1 Introduction

In this thesis, we are studying the dial-a-ride problem with transfers (DARP-T). This problem is related to the traveling salesman problem (TSP), the vehicle routing problem (VRP), the pick-up and delivery problem (PDP), and the dial-a-ride problem (DARP). The TSP consists in finding, for a vehicle, the minimum cost route that visits a set of locations and returns to its starting location. The VRP is a generalization of the TSP that consists in finding, for a fleet of vehicles, the set of minimum cost routes that start and end at the same location, and for which each location is visited by one route. The PDP is a generalization of the VRP that consists in finding, for a fleet of vehicles, the set of minimum cost routes that satisfy a set of location-to-location transportation requests. The vehicles can transport several requests simultaneously as long as the capacity of the vehicles is not exceeded. The DARP is a generalization of the PDP in which transportation requests have time windows and user inconvenience constraints, as it considers the transportation of people instead of goods. The DARP-T is a generalization of the DARP in which the vehicles are allowed to exchange requests. The DARP and the DARP-T are said to be *static* if all information relevant to the decision making (appearance of new requests, disturbances, ...) is provided before the start of operations. On the other hand, if the information relevant to the decision making is progressively revealed during operations, the problem is said to be *dynamic*.
We start by explaining the context motivating this research in Section 1.1. Because most of the literature on similar problems does not consider transfers, we first review the research on related problems without transfers in Section 1.2. We review the literature on similar problems with transfers and explain how our research fits in the existing literature in Section 1.3. The objective of this thesis and the thesis organization is given in Section 1.4.

1.1. Context

Automated vehicles are expected to revolutionize mobility and could be used to create an affordable on-demand mobility service. Fagnant and Kockelman (2015) estimate that every automated vehicle could save on average $4000 per year in parking benefits, crash savings, fuel efficiency, and travel time reduction.
In densely populated cities, many authors (International Transport Forum (2015), Fagnant and Kockelman (2015)) estimate that, with ride-sharing, one shared autonomous vehicle could replace up to 10 traditional cars. Spieser et al. (2013) estimate that in a metropolis like Singapore, automated vehicles could satisfy the overall demand without ride-sharing with less than a third of today's fleet. Indeed, cars are currently parked 95% of the time and even during rush hours, only up to 16% of the cars are used simultaneously (Fagnant and Kockelman (2015)). This reduction in the number of cars will free up parking space. For the medium-sized European city of Lisbon, the on-street parking amounts to 1.5 million square meters that could be put to other uses (Dia and Javanshour (2017)).
Since the use of the cars is more intense, their lifespan will decrease, allowing for a faster adoption of new, cleaner technologies (International Transport Forum (2015)). Additionally, in case of a combustion engine fleet, the intense usage reduces the number of cold starts, responsible for the emission of large amounts of atmospheric pollutants (Fagnant and Kockelman (2014)). Automated vehicles will also use energy-efficient driving techniques to reduce their noise and pollution impact (Spieser et al. (2013)). By allowing automated vehicles to operate close to one another, they can take advantage of each others' slipstream, reducing their

mileage by approximately 20% (Poczter and Jankovic (2013)).
Automated vehicles are expected to be effective at reducing congestion and travel times (Spieser et al. (2013)). They will also be accessible to non-drivers such as young children, persons with a disability, and the elderly (Fagnant and Kockelman (2015)). Finally, autonomous cars are safer than traditional ones (International Transport Forum (2015)). In fact, 90% of the accidents in the U.S. are caused by human errors and more than 40% of the fatal crashes involve alcohol, distraction, drugs, or fatigue (U.S. National Highway Traffic Safety Administration (2008)).

1.2. Literature on problems without transfers

Early work on the DARP was published by Psaraftis (1983), who derives a $\mathcal{O}(n^2 3^n)$ (n being the number of requests) exact dynamic programming algorithm for the single-vehicle DARP. The existing literature on the DARP is reviewed by Cordeau and Laporte (2007). They describe the DARP variants as well as the different mathematical formulations of the problem. Then, they proceed to discuss the different heuristics used to solve the problems and compare their features and performance.
Berbeglia et al. (2012) develop a hybrid tabu search (TS) and constraint programming (CP) heuristic for the dynamic dial-a-ride problem. Tabu search explores the solution space using a variant of local search that allows worsening moves when no improving move is found and that uses memory (a tabu list) to prohibit some moves in order to avoid revisiting solutions. Their algorithm quickly decides whether to accept or refuse new requests with the constraint that, when a request is accepted, the algorithm can not refuse it later on. They use TS and CP in parallel when a new request is received. They observe that CP is good at proving the infeasibility of a request insertion while TS is good at finding feasible solutions.
Parragh et al. (2010) solve the DARP using variable neighborhood search (VNS) with three types of neighborhoods. VNS is a hill-climbing technique introduced by Mladenović and Hansen (1997) that explores increasingly distant neighborhoods of the current incumbent solution. To avoid being trapped in local minima, VNS uses the fact that a local minimum in a given neighborhood structure is not always a local minimum in all neighborhood structures (but that global minima must be local minima in all neighborhood structures). They introduce the idea of zero-split neighborhoods. Members of a zero-split neighborhood are obtained by finding two arcs in the route of a vehicle where the vehicle is not carrying any requests and reinserting the requests served between these two arcs in other routes.
Ritzinger et al. solve the DARP by first using a dynamic programming (DP) inspired heuristic to generate an initial solution and then large neighborhood search (LNS) to further improve it. LNS is a technique that explores the search space by destroying large parts of the incumbent solution before repairing the partially destroyed solution. During the dynamic programming phase of the optimization, to avoid exploring the entire search space, they use a heuristic to bound the number of branches of the decision tree that are considered. During the LNS phase, in addition to the classic LNS destroy/repair operators, they use their DP heuristic in two new destroy and repair operators based on zero-split and on tour removal. Parragh and Schmid (2013) solve the DARP using a hybrid LNS and column generation heuristic. They discover that column generation improves the convergence speed of LNS.
Li et al. (2016) solve the Share-a-Ride problem (parcels and passengers share a ride) using adaptive large neighborhood search (ALNS). Parcels do not have time windows but passengers do. Vehicles can contain several parcels but at most one passenger at the same time. ALNS is a LNS technique that keeps track of the performance of its destroy and its repair operators and decides which one to use depending on their success rates.
Toth and Vigo (2003) solve the vehicle routing problem (VRP) using a new variant of tabu search they named granular tabu search (GTS). The difference between GTS and TS is that GTS only considers neighborhoods that are likely to be part of a high-quality feasible solution by removing expensive arcs.
Qu and Bard (2014) solve a variant of the PDP with a branch price and cut algorithm. They manage to solve instances to optimality for up to 50 requests. Ropke and Pisinger (2006) solve the pick-up and delivery problem with time window (PDPTW) with ALNS, using several removal and insertion heuristics.
Rieck et al. (2014) give a mixed integer linear programming (MILP) model for a variant of the many-to-many location-routing problem. They solve small scale instances with CPLEX. For larger instances, they either use a genetic algorithm or a fix-and-optimize procedure. Inspired by the process of natural selection, genetic algorithms recombine and mutate the fittest solutions from a generation of solutions to produce the next generation. The fix-and-optimize procedure works by generating a feasible solution, fixing a subset of the decision variables and optimizing the remaining variables with the commercial solver CPLEX.

Cordeau (2006) develops a branch and cut algorithm to solve the DARP using new DARP-specific inequalities as well as inequalities from other similar problems (TSP, VRP, and PDP). They are able to solve small instances and suggest using such method as a subroutine in a metaheuristic when solving larger instances.
Afifi et al. (2013) describe a simulated annealing (SA) algorithm for the VRP with time windows and synchronization constraints. The synchronization constraints require some locations to be visited by two vehicles simultaneously. SA is a probabilistic optimization technique inspired by the slow cooling of metals. SA accepts a modification of the current solution with a probability depending (among others) on decreasing a parameter called the temperature. Afifi et al. (2013) use a variant of SA in which a reheating mechanism regularly resets the temperature. They test their algorithm on a benchmark from the literature and report finding all known optimal solutions in shorter computation times. They strictly improve the best known solutions of some instances.
Pisinger and Ropke (2007) present a general heuristic for five different variants of the VRP. They introduce a noise term in their decision heuristics to diversify their destroy and repair neighborhoods.

1.3. Literature on problems with transfers

Masson et al. (2013) solve the PDPT using ALNS. Their model includes time windows and allows transfers to happen only at predetermined locations. They manage to the total distance traveled by the vehicles in some instances from the literature and also test their algorithm on their real-life instances in which the introduction of transfers brings improvements of up to 9%. They observe that transfers are useful when people from the same geographical area need to be transported to different locations.

Rais et al. (2014) develop a MILP model for the pick-up and delivery problem with transshipment (PDPT). It includes optional time windows and requires transshipments to take place at designated transshipment nodes. The model is polynomially bounded in the size of the problem. They use the commercial solver GUROBI to solve instances containing up to 14 nodes to optimality. These instances are also solved to optimality without transshipments and they report an improvement of total distance traveled by the vehicles in 13 of their 20 solved instances, at the cost of larger CPU time.

Qu and Bard (2012) solve the PDPT using a two step algorithm. The first step generates feasible solutions with a Greedy Randomized Adaptive Procedure (GRASP) and the second step attempts to improve the results of the first step with ALNS. GRASP is a technique that finds solutions by repeatably sampling the search space using a stochastically greedy heuristic. Unable to find public data to benchmark against, they introduce their own 25 requests instances of the PDPT. Cortés et al. (2010) present an arc-based formulation of the PDPT and solve small instances to optimality using a branch and cut algorithm based on Benders decomposition. They conjecture that transfers will become more and more profitable under high demand. The size of the instances they can solve is unfortunately too small to provide experimental evidence of that conjecture.
Masson et al. (2014) solve the DARP-T using an extended version of the ALNS algorithm they previously developed to solve the PDPT (Masson et al. (2013)). In order to speed up the feasibility checks, they derive several computationally cheap necessary/sufficient conditions for feasibility. They assess the savings the transfers can provide by solving DARP instances that have been solved to optimality by Ropke et al. (2007). On the 20 instances tested, the introduction of transfers allows the ALNS algorithm to reduce the total distance traveled by the vehicles by on average 3.5%, at the cost of large CPU time.

1.4. Thesis structure

In this thesis, we describe a new constructive greedy randomized repair-and-destroy heuristic for the DARP-T. We introduce and solve DARP-T instances containing up to 1000 requests, all traveling inside a four-hour timescale. These instances require a service quality competitive with public transportation. We solve these instances with and without transfers. The goal of this research is to quantify the savings in total distance traveled by the vehicles that transfers can bring, the user inconvenience they may cause, and the influence the demand has on the fleet efficiency.

The remainder of this thesis is organized as follows. First, we define the DARP-T and discuss some of the challenges involved with the introduction of transfers in Chapter 2. Then, we define the concepts our algorithm is based on, provide the algorithm framework, and describe the repair and destroy procedure in Chapter 3. In Chapter 4, we generate instances of the problem and solve them with and without transfers. In Section 4.3, we provide a few visualizations of the solutions, investigate the user inconvenience, look into the influence of the demand on the solution quality, investigate the influence the fleet size has on the vehicle

usage, and experimentally determine the running time of the algorithm. We conclude the thesis and provide recommendations for future work in Chapter 5.

2

Problem description

This chapter gives a description of the DARP-T. Section 2.1 provides a formal definition of the problem. Section 2.2 discusses some of the challenges involved with the introduction of transfers and demanding time windows, and introduces the main ideas the algorithm described in Chapter 3 is based on.

2.1. Problem formulation

We consider the following problem: given a set of requests R and a fleet of vehicles V, maximize the number of requests served while minimizing the total distance traveled by the vehicles. The objective function is described more formally in Equation 2.1.

The road network of the problem is represented by a complete, weighted directed graph G whose set of nodes N represents a set of geographical locations and whose arc weights represent the time required to travel from the tail of an arc to the head of an arc. The weight of an arc is given by the function $d(a,b)$ for any $a,b \in N$. Note that d is non-negative and respects the triangular inequality, that is $d(a,b)+d(b,c) \geq d(a,c)$ for any $a,b,c \in N$. We use the notation $d(n_1,n_2,\ldots,n_k) := \sum_{i=1}^{k-1} d(n_i,n_{i+1})$ to denote the travel time of a sequence of arcs.

Individual requests and vehicles are defined by attributes which we denote by $object.attribute$. Every request $r \in R$ is described by a pick-up node $r.p$, a drop-off node $r.d$, an earliest pick-up time $r.e$, a latest drop-off time $r.l$ and a group size $r.gs$. Every vehicle is described by a capacity $v.c$, a position $v.pos$ and the time $v.t$ at which the vehicle reached $v.pos$. A vehicle v can move to another node by traveling along any arc of G connected to $v.pos$ and doing so increases its $v.t$ by the weight of the traveled arc. Vehicles can park at any node for any amount of time. Vehicles serve requests picking them up at their pick-up nodes after their earliest pick-up time and transporting them to their drop-off node before their latest drop-off time. Vehicles are allowed to transport several requests simultaneously as long as the sum of the group size of all the requests inside a vehicle never exceeds the capacity of the vehicle. Vehicles are also allowed to exchange requests at any node. For transfers in which one vehicle does not receive any request, that vehicle is allowed to drop-off its transferred requests and drive off before the arrival of the other vehicle.

Given a solution S serving a set A of requests, the objective function is defined to be

$$cost(S) = \frac{\sum_{v \in V} \text{time } v \text{ spent driving}}{\sum_{r \in A} d(r.p, r.d)} + (|R| - |A|)\,penalty \tag{2.1}$$

with $penalty$ a value large enough to ensure that a solution serving n requests has a lower cost than all solutions serving $n-1$ requests.

This model of DARP-T does not include explicit user inconvenience constraints such as a maximal ride time or a preferred time of departure/arrival, and therefore, one might think we are solving the PDPT. However, we assume that the inputted time windows $[r.e, r.l]$ are not much larger than the travel time $d(r.p, r.d)$ for all requests r. The time windows in the instances we solve in Chapter 4 are on average about 50% larger than the corresponding $d(r.p, r.d)$. For example, this means that a request wanting to travel between two locations 60 minutes apart must be transported within the 90 minutes time window it specified. For the Netherlands, this is competitive with public transportation. Therefore, we are indeed solving DARP-T and not PDPT instances.

2.2. Problem characteristics

The tight time windows of the DARP-T might first seem beneficial as they allow us to prune many decisions by infeasibility. While this is true, they also make the feasibility of a solution very vulnerable to change. Take, for example, the fragment of a solution shown in Figure 2.1. Nodes are represented by small black dots, requests by arrows, vehicles by stars, and routes by wavy lines. The number written under the vehicles indicates the time at which the vehicles reached their displayed position. The vehicles are empty and have the same capacity. Colors are used to distinguish the different objects. This fragment looks like it could be improved by swapping the tasks of the two vehicles (the red vehicle would pick-up the orange and green requests and the blue vehicle the blue request). Without time windows, such change improves the objective function value and does not affect the rest of the solution. Therefore, we know with certainty that the displayed fragment is not part of any optimal solution and can be discarded. In other words, the displayed fragment can be pruned by optimality. However, this is not necessarily the case with time windows. One can see that the blue vehicle has arrived at its shown position before the red vehicle. Therefore, even though the red vehicle is closer to the orange request, it might not be able to feasibly serve the orange or green request, making the route or ride-sharing (if only one request can be picked-up) no longer feasible. This reasoning extends to the rest of the solution. Therefore, the swap being feasible in the fragment does not imply that the swap is feasible in the entire solution. Another unfortunate consequence is that even if the swap is not affecting the feasibility of the entire solution, there is no guarantee anymore that the displayed fragment is not part of an optimal solution. Indeed, it is not because the swap does not affect the feasibility of some solution that it does not affect the feasibility of an optimal solution. In other words, we cannot prune the displayed case by optimality anymore.

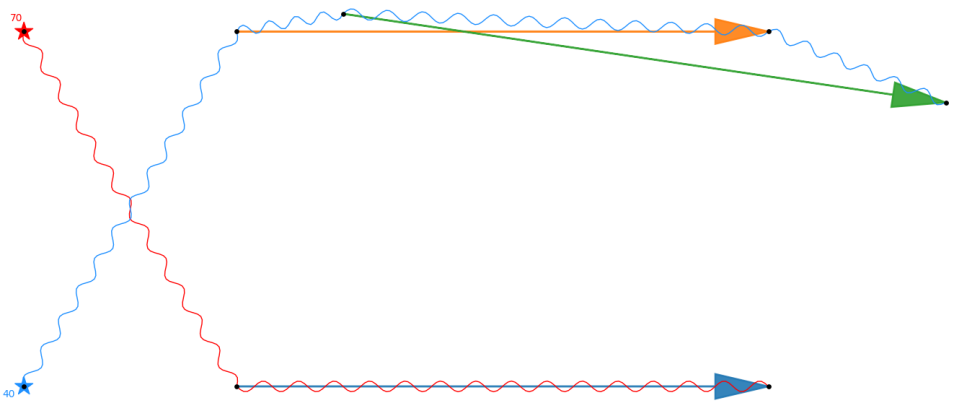


Figure 2.1: Example of a fragment of a solution with 3 requests and 2 vehicles.

The vulnerability of the feasibility of a solution to change is exacerbated by the transfers. Transfers require two vehicles to arrive at a specific node with a specific set of passengers within a tight time window. Most modifications of a route involved in a transfer are likely to make the transfer (and therefore the entire solution) infeasible. Therefore, finding new feasible solutions or exploring the neighborhood of a given solution is not easy and computationally expensive in general. Computationally cheap modifications could be done by stepping out of the solution space and considering infeasible solutions. However, because of the rarity of feasible solutions, fixing them will probably either be hard or significantly worsen their objective function value. This leads to the two main ideas the algorithm is built upon:

- Feasibility at all times: computational time spent on infeasible solutions is considered wasted and should be avoided. The algorithm presented in this thesis progressively builds a solution with the constraint that it must know at all times that the solution it is building can be completed to a feasible solution.
- Greediness: because the solution space is expensive to explore, we do not consider moves that are unlikely to be part of a good solution.

3

Solution approach

This chapter describes the algorithm used to solve the DARP-T. We start with definitions in Section 3.1 then provide an outline of the algorithm in Section 3.2. The solution selection criterion is explained in Section 3.3. The repair phase of the algorithm is described in Section 3.4 and the destroy phase in Section 3.5.

3.1. Definitions

In this section, we define some important concepts, we introduce some additional attributes for the vehicles and the requests, and we give a practical example of these definitions in a visualization of a problem. We introduce a general cost function in Section 3.1.1 and a geometrical representation of the graph in Section 3.1.2.
Let us define the following concepts:

- *Route*: a route is a sequence of (node, time, action) tuples called *units*, associated with a vehicle. Actions can be wait, pick-up, transfer, or drop-off. Every node of the route must be assigned an action except for the first node (the start node). The element time is the time at which the action is finished. A route must respect the constraints of its associated vehicle (e.g., arcs are traveled in the time specified by d) and every action must be feasible (e.g., the pick-up of request r must happen after $r.e$). An example of a route is given in Table 3.1. The following adjectives are used to refer to different types of routes.
  - *Entire*: an entire route is a route that starts at the initial position and time of its associated vehicle. The algorithm presented in this thesis builds solutions by constructing one entire route for every vehicle by repeatedly appending small non-entire routes to the entire route. For a vehicle v, these entire routes are denoted by $v.er$ and always contain at least one unit describing the initial position and time of the vehicle.
  - *Complete*: a complete route is a route that is empty (i.e., no request inside the vehicle) at its last node. The route shown in Table 3.1 is complete.
  - *Incomplete*: routes that are not empty at their last node are said to be *incomplete*. Note that there is no certainty that all the requests at the end of an incomplete route can still be feasibly dropped-off. Therefore, because of the feasibility at all time principle, incomplete routes cannot be part of the solution unless they can be proven to be completable to a complete route.

Table 3.1: Example of a complete route

Node	Time	Action
4	0	None
4	28	Wait
12	35	Pick-up request 3
34	64	Transfer with vehicle 9: exchange request 3 for request 6
26	87	Drop-off request 6

Finding the shortest route serving a set of requests is equivalent to solving the open capacitated single vehicle PDPTW. Since the algorithm presented in this report builds long routes by joining several smaller routes together, we only solve the open capacitated single vehicle PDPTW for small sets of requests and complete enumeration is not computationally costly. Therefore, all routes computed in this thesis are computed by complete enumeration.

- *Solution*: a list of complete, entire routes (one per vehicle).
- *Partial solution*: a list of possibly incomplete, entire routes (one per vehicle). Solutions are gradually built and the partial solutions are the intermediary steps in the construction of a solution.
- *Certificate of feasibility*: in Section 2.2, we explain that the solution being built — the partial solution — must always be known to be completable to a feasible solution. Therefore, if the entire route of a vehicle v is incomplete, it must come with a complete route called the certificate of feasibility and denoted by $v.cf$ proving that the content of the vehicle can indeed feasibly be dropped-off.
- *Current(ly)*: indicates that we are mentioning the most recent situation of a vehicle, that is the situation described at the last unit of its entire route. For example, if Table 3.1 is the entire route of a vehicle v, the current position of v is node 26, the current time of v is 87, v is currently empty, etc.
- $timeRoute(route)$: the time at the last unit of the route minus the time at which the route started. For the example route of Table 3.1, $timeRoute$ would return 87.
- The vehicle attribute $canBeModified$: we sometimes allow and sometimes prohibit the modification of the $v.er$ of a vehicle (more details in Section 3.4). The Boolean $v.canBeModified$ is set to true if such a modification is allowed, false otherwise.
- The request attribute ld: the latest feasible time of departure of a request r, denoted by $r.ld$, is the maximum time at which r can be picked up and still be dropped off before $r.l$.
- The request attribute $related$: the set of requests related to a request r, denoted $r.related$, is defined to be:

  $r.related = \{q \in R \mid timeRoute(\text{shortest route starting at } r.p \text{ serving } r \text{ and } q) \leq d(r.p, r.d) + d(q.p, q.d)\}$.

  It is used to find requests that could be share a vehicle or be part of a transfer,
- The request attribute $isHandled$: the Boolean $r.isHandled$ is set to true if request r is handled, false otherwise.
- The request attribute $isRejected$: the Boolean $r.isRejected$ is set to true if request r is rejected, false otherwise. Note that $r.isRejected$ implies that $r.isHandled$ (but not the other way around).

Table 3.2 summarizes the attributes of the requests and vehicles.

To make these definitions less abstract and understand in what context they are used, an example of a partial solution is shown in Figure 3.1. Nodes are represented by small dots, requests by arrows, entire routes by dashed lines, current positions of the vehicles by stars, and feasibility certificates by wavy lines. Colors are used to distinguish the different objects. To avoid overloading the illustration, some pieces of information, such as the time windows of the requests, the vehicle content, the visiting time of nodes are not explicitly shown (but can be deduced).
From the entire route of the blue vehicle, we can deduce that the blue vehicle picked up the brown, purple, and green request (in that order), then transferred the brown request to the red vehicle, then dropped-off the purple and the green request (in that order). Thus, the blue vehicle is currently empty and therefore has a trivial (empty) certificate of feasibility.
From the entire route of the red vehicle, we can see that the red vehicle transported the blue and orange requests, then picked up the yellow request and drove to the transfer point where it received the brown request from the blue vehicle. Therefore, the red vehicle now contains the yellow and the brown requests. Since the vehicle is not empty in the partially solved solution, we have a certificate of feasibility to prove that the vehicle can indeed drop off its content. In this case, the certificate proves that the brown request and the yellow request can feasibly be dropped off. Note that the certificate is not necessarily the route the algorithm will

choose for the vehicle. For example, in this case, the algorithm could make the red vehicle pick up the pink request between the drop-off node of the brown and yellow requests as it is more efficient.

Table 3.2: Summary of all attributes

Short notation	Description
$r.p$	pick-up node of request r
$r.d$	drop-off node of request r
$r.e$	earliest pick-up time of request r
$r.l$	latest drop-off time of request r
$r.gs$	group size of request r
$r.ld$	latest feasible time of departure of r
$r.related$	set of requests related to r
$r.isHandled$	Boolean set to true when r has been handled, false otherwise
$r.isRejected$	Boolean set to true if r has been rejected, false otherwise
$r.dir$	direction of r (see Section 3.1.2)
$v.c$	capacity of vehicle v
$v.pos$	current position of v
$v.t$	current time of v (time at which v reached $v.pos$)
$v.er$	entire route of the v
$v.cf$	certificate of feasibility of v
$v.canBeModified$	a Boolean set to true if $v.er$ can be modified, false otherwise

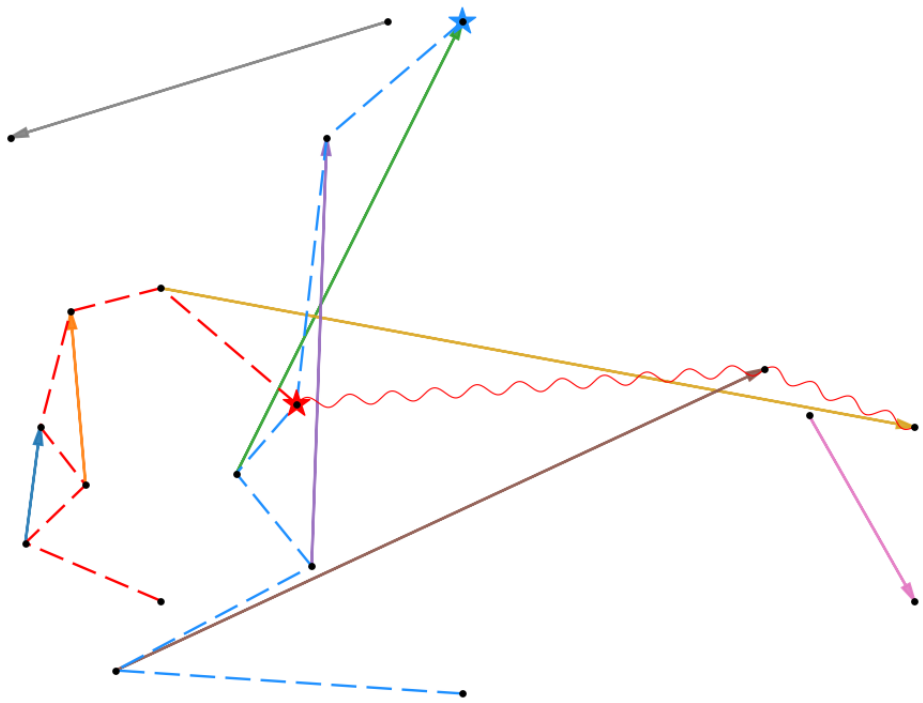


Figure 3.1: Visualization of a partially solved instance of an example problem.

3.1.1. Local cost function

The repair phase from Section 3.4 is constructive and progressively builds a solution by taking decisions based on local criteria. These criteria occasionally compare different objects. For example, we sometimes need to decide whether to do a 2-vehicle transfer or to route these two vehicles independently. Therefore, we need a general $cost$ function able to compare the quality of different objects. However, since we repeatedly partially destroy the solution (see Section 3.5), we might have to repair the same part of the solution several times. Therefore, we do not want this $cost$ function to be excessively greedy and always lead to the same decision.

Thus, whenever two objects are compared using $cost$, one of the $cost$ functions is multiplied by a factor sampled from the continuous uniform distribution $\mathcal{U}(0.85, 1.15)$. This way, the cost comparison is still greedy but will not always favor the lower cost object. This is similar to the noise term proportional to the objective function introduced by Pisinger and Ropke (2007).

Take a route ro associated with vehicle v and let us denote by C_{ro} the set of requests that are transported by ro. Note that v is not necessarily empty at the first unit of ro (e.g. certificate of feasibility) or at the last unit of ro (e.g. incomplete route) and that it does not necessarily pick up or drop off requests at their pick-up or drop-off nodes (as it might exchange requests with another vehicle). Therefore, for a request $r \in C_{ro}$, we denote by n_{in} the node corresponding to the first unit of ro containing r and n_{out} the node corresponding to the last unit of ro containing r. We measure how much a request $r \in C_{ro}$ has been transported by ro using the function

$$f(r, ro) = \begin{cases} d(r.p, r.d) - d(r.p, n_{in}) & \text{if } n_{out} = r.d \\ d(r.p, r.d) - d(n_{out}, r.d) & \text{if } n_{in} = r.p \\ d(n_{in}, r.d) - d(n_{out}, r.d) & \text{if } n_{out} \neq r.d \text{ and } n_{in} \neq r.p \end{cases}. \tag{3.1}$$

A diagram illustrating the three cases of Equation 3.1 is shown in Figure 3.2.

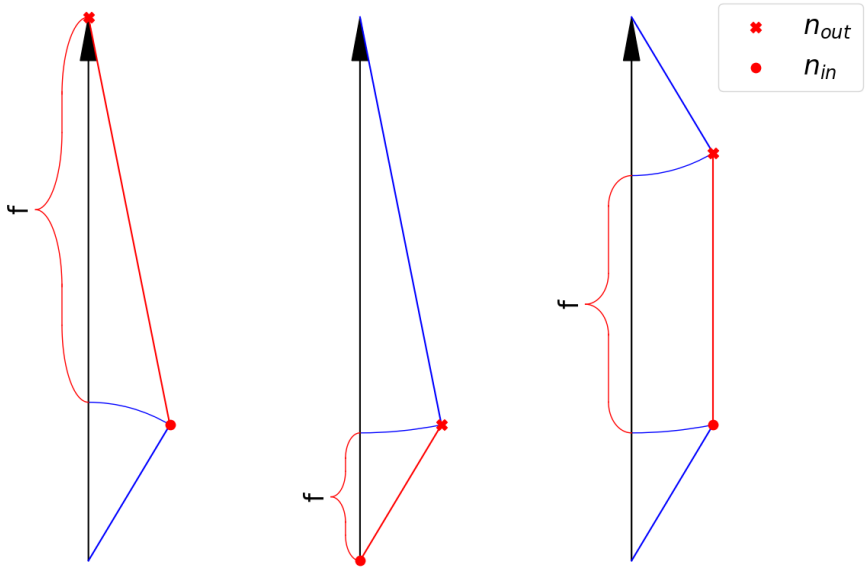


Figure 3.2: Illustration of the three cases of Equation 3.1. The request r is indicated by the arrow, the part of ro that transports r is the red line and the value of $f(r, ro)$ is indicated by the curly bracket.

We define the cost of a single route ro of vehicle v to be the time v spends transporting requests, divided by how much these requests have been transported. Formally, we have

$$cost(ro) = \frac{timeRoute(ro)}{\sum_{r \in C_{ro}} f(r, ro)}. \tag{3.2}$$

Similarly, the total cost of two independent routes ro_1 and ro_2 is defined to be

$$cost(ro_1, ro_2) = \frac{timeRoute(ro_1) + timeRoute(ro_2)}{\sum_{r \in C_{ro_1}} f(r, ro_1) + \sum_{r \in C_{ro_2}} f(r, ro_2)}. \tag{3.3}$$

Given two vehicles and two corresponding sets of routes W_1 and W_2, we define the cost function on the two sets of routes to be the total cost of the best route in W_1 and the best route in W_2, that is

$$cost(W_1, W_2) = cost(\operatorname*{argmin}_{w_1 \in W_1} cost(w_1), \operatorname*{argmin}_{w_2 \in W_2} cost(w_2)). \tag{3.4}$$

A transfer is described by two routes ro_1 and ro_2 that bring requests to the transfer point and two certificates of feasibility cf_1 and cf_2 proving that the requests brought to the transfer point can be feasibly dropped-off. The cost of the transfer is defined in Equation 3.5. The factor $\alpha \sim \mathcal{U}(0.7, 1)$ is applied to the $timeRoute$

term of the certificate of feasibility to compensate for the fact that the certificates of feasibility are likely to be improved later on.

$$cost(transfer) = \frac{\sum_{i=1,2}\left(timeRoute(ro_i) + \alpha timeRoute(cf_i)\right)}{\sum_{i=1,2}\left(\sum_{r\in C_{ro_i}} f(r,ro_i) + \sum_{r\in C_{cf_i}} f(r,cf_i)\right)} \tag{3.5}$$

3.1.2. Multi-dimensional scaling

The example given in Figure 3.1 makes it clear that, when trying to solve the problem by hand, concepts from geometry such as the relative position of nodes on a map and direction are very helpful to find good routes. However, in the instance of the problem we described, nodes and their distances from one another are represented by a complete graph in which these geometrical concepts do not exist. If we could map every node to a vector in $\mathbb{R}^d$ such that the Euclidean distance between any two of these vectors would be approximately the travel time between the two corresponding nodes, we could incorporate some geometrical heuristics from our visual intuition into the algorithm.
For example, imagine considering organizing a transfer between two routes, such as the ones shown in Figure 3.3. Nodes are represented by small black dots and the routes by lines. One can see that the nodes visited by the blue route are all far from the nodes visited by the red route. However, this does not imply that the two routes are far apart and that a transfer is not possible, as can be seen by the two routes intersecting close to a city in the figure.
Without a geometric representation of the situation, we determine whether two routes are candidates for a transfer by looking for nodes that can be cheaply inserted in both routes. This is done in $\mathcal{O}(|N|(|ro_1|+|ro_2|))$ time. Using the geometric representation of the graph, we can compute the Euclidean distance between any two routes in $\mathcal{O}(|ro_1||ro_2|)$ time. Since $|N| \gg |ro|$, this is an efficient way to assess whether a transfer between two routes is likely to be successful. Additionally, whenever two routes are found to be close to one another, candidate nodes for a transfer can efficiently be found by narrowing down the search for transfer nodes to a small zone (for example, the nodes inside the blue circle in Figure 3.5). For real-life instances, we can expect the set N to be large (for example, there are 50 thousand bus stops in the Netherlands (Zijlstra et al. (2018))) and developing transfer organizing procedures having a time complexity independent of $|N|$ is crucial.

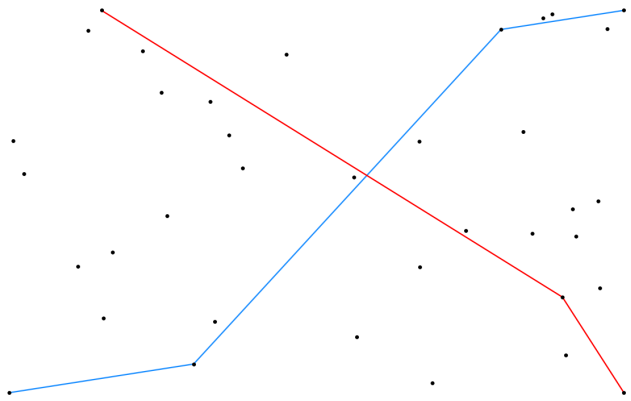


Figure 3.3: Visualization of a partially solved instance of an example problem.

The problem of representing complete weighted graphs in a Cartesian space has already been studied (see for example Borg and Groenen (2005)) and can be solved using techniques from Multi-Dimensional Scaling (MDS). MDS creates a map $mds: N \to \mathbb{R}^d$ such that $\|mds(n_1) - mds(n_1)\|_2 \approx d(n_1,n_2)$ for all $n_1, n_2 \in N$ (with $\|\cdot\|_2$ the Euclidian norm). Note that equality is not possible in general. Therefore, the map is found by solving the following optimization problem

$$\min_{x_1,\ldots,x_N\in\mathbb{R}^d} \sum_{n_i\neq n_j} (d(n_i,n_j) - \|x_i - x_j\|_2)^2.$$

Many libraries such as scikit-learn[1] have a numerical solver designed for this problem.
We use the mds map to create an additional request attribute denoted $r.dir$ that indicates the direction of a

[1] www.scikit-learn.org

request r. The vector $r.dir$ is defined to be

$$r.dir = \frac{mds(r.d) - mds(r.p)}{\|mds(r.d) - mds(r.p)\|_2}.$$

We have seen in the example illustrated by Figure 3.3 that representing routes in $\mathbb{R}^d$ can be useful. Suppose a route ro is visiting nodes $n_1, n_2, \ldots, n_J$ (in that order). We define the geometric representation of that route to be the set

$$mds(ro) = \bigcup_{i=1,2,\ldots,J-1} \{(1-\lambda)mds(n_i) + \lambda mds(n_{i+1}) \mid \lambda \in [0,1]\}.$$

We define the distance between the route ro and a vector $v \in \mathbb{R}^d$ to be

$$\min_{x \in mds(ro)} \|v - x\|_2. \tag{3.6}$$

An example is shown in Figure 3.4.

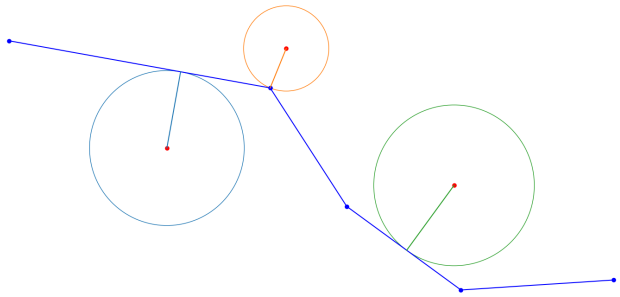


Figure 3.4: Example of the distance between a route ro and vectors in $\mathbb{R}^2$ (the red dots). The set $mds(ro)$ is colored in blue and the blue dots are the nodes visited by ro. The distance between the vectors and ro is represented by the radius of the circles drawn around the vectors.

Claim 1. *Equation 3.6 can be computed in $\mathcal{O}(|ro|)$ time. Indeed, setting $\tilde{n}_i = mds(n_i)$, $d_i = \tilde{n}_{i+1} - \tilde{n}_i$ and using $\cdot$ to denote the dot product, Equation 3.6 can be simplified to*

$$\min_{i=1,\cdots,J-1} \left\| v - \tilde{n}_i - \min\left(1, \max\left(0, \frac{(v-\tilde{n}_i)\cdot d_i}{d_i \cdot d_i}\right)\right) d_i \right\|_2. \tag{3.7}$$

Proof. Since

$$mds(ro) = \bigcup_{i=1,2,\ldots,J-1} \{\tilde{n}_i + \lambda d_i \mid \lambda \in [0,1]\},$$

we can rewrite Equation 3.7 as

$$\begin{aligned} \min_{x \in mds(ro)} \|v - x\|_2 &= \min_{i=1,\cdots,J-1} \left(\min_{x \in \{\tilde{n}_i + \lambda d_i \mid \lambda \in [0,1]\}} \|v - x\|_2 \right) \\ &= \min_{i=1,\cdots,J-1} \left\| v - \tilde{n}_i - \left(\operatorname*{argmin}_{\lambda \in [0,1]} \|v - \tilde{n}_i - \lambda d_i\|_2 \right) d_i \right\|_2. \end{aligned} \tag{3.8}$$

Let us define $f(\lambda) = v - \tilde{n}_i - \lambda d_i$. Since the square function is strictly increasing on $\mathbb{R}_{\geq 0}$, we have

$$\operatorname*{argmin}_{\lambda \in [0,1]} \|f(\lambda)\|_2 = \operatorname*{argmin}_{\lambda \in [0,1]} f(\lambda) \cdot f(\lambda).$$

We have

$$\frac{d}{d\lambda} f(\lambda) \cdot f(\lambda) = 2\lambda(d_i \cdot d_i) - 2d_i \cdot (v - \tilde{n}_i),$$

One can see that the derivative of $f(\lambda) \cdot f(\lambda)$ is a strictly increasing function and has a unique root at $\lambda_0 = \frac{(v-\tilde{n}_i)\cdot d_i}{d_i \cdot d_i}$. Therefore, $f(\lambda) \cdot f(\lambda)$ is strictly decreasing on $(-\infty, \lambda_0]$ and strictly increasing on $[\lambda_0, \infty)$. As a result,

$$\begin{aligned} \operatorname*{argmin}_{\lambda \in [0,1]} \|v - \tilde{n}_i - \lambda d_i\|_2 &= \begin{cases} 0 & \text{if } \lambda_0 < 0 \\ \lambda_0 & \text{if } 0 \leq \lambda_0 \leq 1 \\ 1 & \text{if } \lambda_0 > 1 \end{cases} \\ &= \min(1, \max(0, \lambda_0)). \end{aligned}$$

Plugging in this result into Equation 3.8 gives Equation 3.7. □

The distance between a route ro_1 and a route ro_2 is defined to be

$$\min_{x_1 \in mds(ro_1), x_2 \in mds(ro_2)} \|x_1 - x_2\|_2. \tag{3.9}$$

An example is shown in Figure 3.5.

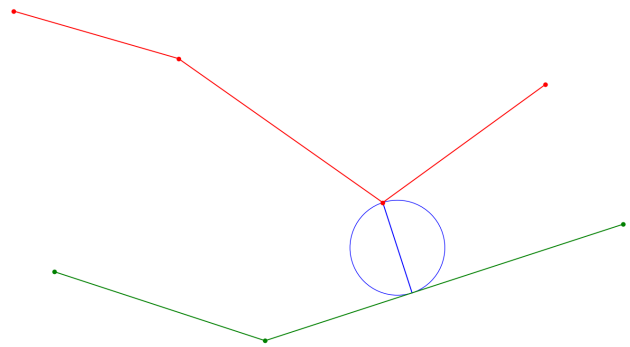


Figure 3.5: Example of the distance between two routes (in red and in green). The distance is represented by the diameter of the circle.

Claim 2. *Equation 3.9 can be computed in $\mathcal{O}(|ro_1||ro_2|)$ time.*

Proof. Let us denote by $n_1^1, n_1^2, \dots, n_1^K$ the nodes ro_1 is visiting (in that order) and by $n_2^1, n_2^2, \dots, n_2^L$ the nodes ro_2 is visiting (in that order). We denote by $d_i^j = mds(n_i^{j+1}) - mds(n_i^j)$, $p_i^j = mds(n_i^j)$, $x_i^j(\lambda_i) = p_i^j + \lambda_i d_i^j$ Using the definition of $mds(ro)$, we can rewrite Equation 3.9 as

$$\min_{k=1,\dots,K-1} \min_{l=1,\dots,L-1} \min_{x_1 \in S_1^k, x_2 \in S_2^l} \|x_1 - x_2\|_2, \qquad S_i^j = \{x_i^j(\lambda_i) \mid \lambda_i \in [0,1]\}. \tag{3.10}$$

Therefore, Equation 3.9 can be computed in $\mathcal{O}(|ro_1||ro_2|)$ time if the distance between two line segments S_1 and S_2 can be computed in constant time. Let us drop the superscript for clarity and let us compute the distance between some S_1 and S_2.
Since the square function is strictly increasing on $\mathbb{R}_{\geq 0}$, we have

$$(\widehat{\lambda}_1, \widehat{\lambda}_2) := \operatorname*{argmin}_{\lambda_1, \lambda_2 \in [0,1]} \|x_1(\lambda_1) - x_2(\lambda_2)\|_2 = \operatorname*{argmin}_{\lambda_1, \lambda_2 \in [0,1]} f(\lambda_1, \lambda_2)$$

with

$$f(\lambda_1, \lambda_2) = (x_1(\lambda_1) - x_2(\lambda_2)) \cdot (x_1(\lambda_1) - x_2(\lambda_2)).$$

The gradient of f is

$$\nabla f = \begin{pmatrix} 2d_1 \cdot (x_1(\lambda_1) - x_2(\lambda_2)) \\ -2d_2 \cdot (x_1(\lambda_1) - x_2(\lambda_2)) \end{pmatrix}.$$

Setting $\alpha_i = d_i \cdot d_i$, $\beta = -d_1 \cdot d_2$, the Hessian of f is

$$\nabla^2 f = \begin{pmatrix} 2\alpha_1 & 2\beta \\ 2\beta & 2\alpha_2 \end{pmatrix} \tag{3.11}$$

Assume that d_1 and d_2 are not parallel. Then, $\alpha_1 \alpha_2 > \beta^2$ and we have that the Hessian of f is positive definite. Solving $\nabla f(\lambda_1, \lambda_2) = \mathbf{0}$ for λ_1 and λ_2 gives

$$\tilde{\lambda}_i := \frac{\alpha_j d_i \cdot (p_j - p_i) + \beta d_j \cdot (p_i - p_j)}{\alpha_1 \alpha_2 - \beta^2}, \qquad j = \begin{cases} 2 & \text{if } i = 1 \\ 1 & \text{if } i = 2 \end{cases}. \tag{3.12}$$

Since the Hessian of f is positive definite, f is strictly convex and $(\tilde{\lambda}_1, \tilde{\lambda}_2)$ is the unique global minimum of

$$\underset{\lambda_1,\lambda_2\in\mathbb{R}}{\operatorname{argmin}} f(\lambda_1, \lambda_2).$$

Therefore, if $(\tilde{\lambda}_1, \tilde{\lambda}_2) \in [0,1] \times [0,1]$, we have $(\widehat{\lambda}_1, \widehat{\lambda}_2) = (\tilde{\lambda}_1, \tilde{\lambda}_2)$ and we have an expression for the segment to segment distance that can be computed in constant time. If not, then $(\widehat{\lambda}_1, \widehat{\lambda}_2)$ must be on the boundary of $[0,1] \times [0,1]$ and there must be an $i \in \{1,2\}$ such that $\widehat{\lambda}_i \in \{0,1\}$. Therefore, the distance between S_1 and S_2 involves one of the four extreme points of S_1 and S_2 and since the distance between a point and a segment can be computed in constant time (see Equation 3.7), we can compute the distance between S_1 and S_2 in constant time. If d_1 and d_2 are parallel, the minimum distance between S_1 and S_2 must also involve one extreme point and the distance can also be computed in constant time.

□

3.2. Algorithm outline

Now that the main concepts used by the algorithm have been discussed, we can delve into its description. In short, the algorithm works by repeatedly destroying and repairing the incumbent solution. This modification of the incumbent solution occasionally improves it, creating a new incumbent solution. This process goes on until the stopping criterion — 10^4 iterations without an improvement of the solution — is reached.

An outline of the algorithm is given in Algorithm 1. We start by transforming the instance into a trivial partial solution S (no vehicle moves and no request is handled). Then, we define the last accepted solution S_a and the best solution S_b and set them to S. To avoid getting stuck in local minima, we occasionally accept worse solutions, and thus, S_a is not necessarily the best solution found over all iterations of the destroy and repair procedure. Therefore, we remember the best solution in S_b. We set the temperature T to T_0. The temperature is a simulated annealing parameter that influences whether a new solution is accepted, as explained in Section 3.3. We made the temperature time dependent, and thus, let the variable t to be the time in seconds since the start of the algorithm.
Then, we let S be the last accepted solution. Since the vehicles involved in transfers are more likely to be affected by the destruction mechanism (see Section 3.5), we randomly protect half of them against destruction by setting their attribute $canBeModified$ to false. Then, we transform the trivial partial solution S into a solution using Algorithm 2. If S is the best solution found so far, we store it into S_b. We update the temperature T (see Section 3.3) and Equation 3.13 is used to determine whether S becomes the new S_a. After that, S is partially destroyed using the procedures described in Section 3.5. We repeat these repair, acceptance, and destruction steps until the stopping criterion is reached. S_b is the output of the algorithm.

Algorithm 1: Outline of the algorithm

Input: An instance of the DARP-T
Output: A solution S_b of the problem

```
1  Transform the instance into a trivial partial solution S;
2  Set the last accepted solution S_a = S and the best solution S_b = S; Set the temperature T = T_0;
3  Let t be the time in seconds since this line was executed and set the temperature function to be
     Equation 3.14;
4  while S_b has been updated in the last 10^4 iterations do
5      Set S = S_a;
6      Randomly set half of the attributes canBeModified corresponding to a vehicle involved in a
         transfer to false: Set all other canBeModified attributes to true;
7      Transform S into a solution using Algorithm 2;
8      If S has a lower objective cost than S_b, set S_b = S;
9      Update T (see Section 3.3);
10     Determine using Equation 3.13 whether S is accepted; If S is accepted, set S_a = S;
11     Partially destroy S using the procedures described in Section 3.5;
12 Return S_b;
```

3.3. Selection criterion and cooling scheme

The objective function value of a solution is measured by Equation 2.1. The acceptance of a new solution S depends on its objective function value $cost(S)$, the objective function value of the last accepted solution $cost(S_a)$, and a temperature $T \in (0, T_0]$ that determines how likely we are to accept worse solutions. We accept the new solution S with probability

$$\begin{cases} 0 & \text{if } S \text{ is one the last 20 accepted solutions} \\ 1 & \text{otherwise, if } cost(S) \leq cost(S_a) \\ \min\left(1, \exp\left(-\frac{cost(S)-cost(S_a)-\gamma \max(0,\Delta_{trans})}{T}\right)\right) & \text{otherwise} \end{cases} \tag{3.13}$$

with Δ_{trans} the number of transfers in S minus the number of transfers in S_a. We avoid revisiting parts of the solution space by using the last 20 accepted solutions as a short term tabu list. If S is not in the tabu list, it is always accepted when it has a lower cost than S_a. The criterion can also accept worse solutions but becomes less accepting of worse solutions as the temperature decreases. The term containing Δ_{trans} is much smaller than $cost(S)$. It increases the probability of accepting a solution containing more transfers but does not affect the probability of accepting a solution reducing the number of transfers (to prevent keeping inefficient transfers). Because the value of the Δ_{trans} term is small, it mostly plays a diversification role when the temperature is close to 0 by making the criterion accept a solution that is slightly worse, but different from S_a. Since it is not possible to keep increasing the number of transfers without significantly increasing the cost of the solution, the Δ_{trans} term cannot be the cause of the acceptance of worse solutions many times in a row. Therefore, the increase in cost this diversification mechanism can cause to S_a is bounded by the factor γ. Another advantage of the Δ_{trans} term is that it compensates for the higher destruction of the transfers by the destruction mechanism (see Section 3.5) by reintroducing transfers in the solution. An analysis of the performance of this mechanism is provided in Section 4.3.7.
The temperature is a function of the time t (in seconds) elapsed since the start of the algorithm:

$$T(t) = T_0 e^{-t/375}. \tag{3.14}$$

Equation 3.14 is plotted in Figure 3.6. The temperature was designed to take around half an hour to drop under 1% of T_0 as the algorithm was observed to usually take around 15 minutes to find a solution that minimizes the number of rejected requests.

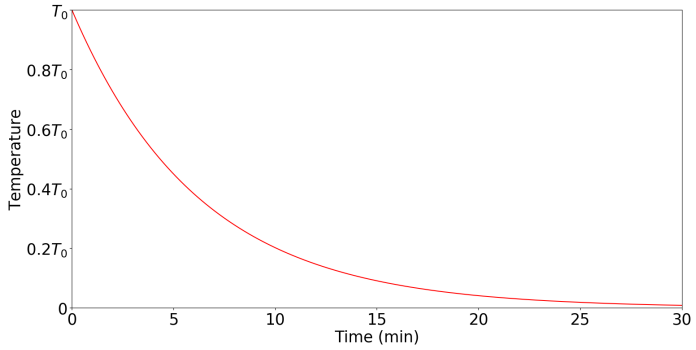


Figure 3.6: Plot of Equation 3.14

The temperature does not only affect the acceptance criterion but also influences the amount of destruction in the destruction phase (Section 3.5). At low temperatures, the amount of destruction is kept low and at high temperatures, large parts of the solutions are destroyed. Since the computational cost of repairing increases with the amount of destruction (see Section 4.3.6), the destroy-and-repair procedure at low temperatures is computationally cheap. Additionally, repairing a small part of the solution does not require taking many decisions, and therefore, we are likely to guess enough of them correctly to improve the incumbent solution. Therefore, at low temperatures, the repair-and-destroy procedure results in an efficient local search since it is computationally cheap (many solutions can be considered), greedy (the acceptance criterion is almost greedy at low temperatures) and only consider close neighbors. The repair-and-destroy procedure at low temperature is, however, incapable of escaping local minima. For this reason, when the algorithm

seems to be stuck in a local minimum, we introduce a reheating mechanism inspired by Afifi et al. (2013). The reheating mechanism increases the temperature by switching to another temperature function

$$T(t) = T_0 e^{-((t-t_0)/8+2)} \tag{3.15}$$

with t_0 the time (in seconds) at which it was decided to switch the temperature function. This decision happens when no solution has been accepted in the last 3000 iterations and when the last S_b was discovered in the last 5000 iterations. Since the algorithm stops after failing to find a new S_b for 10000 iterations, we want to ensure that we still have at least $10000 - 5000 = 5000$ iterations to compensate for the increase of $cost(S_a)$ that will follow the increase in temperature. Equation 3.15 is designed to worsen S_a just enough to displace it from the local minima. Therefore, it starts at $T(t_0) = 0.14T_0$ and reaches $0.01T_0$ in 20 seconds. A computational experiment from Chapter 4 shows the temperature function switch in Figure 3.7. One can see that the temperature increase originally causes a small increase in the cost of S_a but also resumes the objective function value descent. The effect of the Δ_{trans} term is visible in Figure 3.7 as occasional small increases of S_a. One can see that this increase is indeed always bounded.

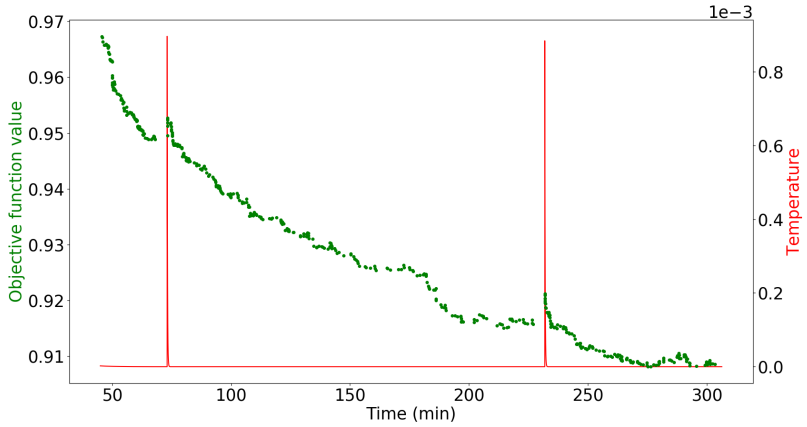


Figure 3.7: Objective function value of the accepted solutions and temperature vs time, when solving an instance from Chapter 3. The first 50 minutes are not shown to keep the range of the y axes relevant, since the initial objective function value and temperature are an order of magnitude larger.

3.4. Repair phase

This section first provides an outline of the repair phase of the algorithm. Then, the procedures used in the repair phase are explained in more details in Section 3.4.1 (vehicle-request matching), Section 3.4.2 (candidate route generation) and Section 3.4.3 (candidate transfer generation).

Usually, destroy and repair procedures clearly separate the two steps and only the destroyed part of the solution is modified in the repair step. However, since we greedily repair the partial solution with a feasibility constraint, repairing a small destruction has a high chance of simply reverting the destruction. Furthermore, since transfer organization relies on a high density of unserved requests and available vehicles, destroying and repairing a small subset of the solution is unlikely to discover transfers. Therefore, applying this procedure will most likely reduce the number of complex routes and transfers, worsening the solution. We could mitigate this problem by only destroying large parts of the solution but, as explained in Section 3.3, a large destruction is computationally expensive and unlikely to improve good solutions. Therefore, we solve this issue differently by allowing the repair operator to occasionally interact with the non-destroyed part of the solution during the repair phase. Every modification of the non-destroyed part of the solution usually results in some additional destruction of the solution. To avoid getting stuck in the repair phase because of this additional destruction, we occasionally forbid some routes to be interacted with in the repair phase by setting their attribute $canBeModified$ to false (See Algorithm 6). Eventually, enough routes will be protected and the solution can only be repaired.

Algorithm 2 gives an outline of the repair phase. It starts by matching vehicles and requests using Algorithm 3. Then, it selects the vehicle with lowest current time v_1, generates different routes for v_1 using Algorithm 4 and tries to find a transfer having a lower cost than these routes using Algorithm 5 (see Section

3.1.1 for the definition of cost). Algorithm 5 is allowed to organize a transfer between v_1 and a non-destroyed route in S. If a transfer is found, the routes that bring the requests to the transfer point are appended to the entire routes of the vehicles involved in the transfer, and the drop-off routes are set as certificates of feasibility for these vehicles. If no better transfer is found, the least cost route in W_1 is appended to $v_1.er$. This update of a vehicle is repeated until all requests are handled (i.e., served or rejected). In other words, we are constructing a solution from the partial solution by moving vehicles one by one.
Figure 3.1 could be an input partial solution of Algorithm 2. Algorithm 2 builds a solution from Figure 3.1 as follow. First, Algorithm 3 matches the blue vehicle to the gray request. The red vehicle is selected as it is the vehicle with lowest current time (see Section 3.1). Candidate routes for the red vehicle are generated using Algorithm 4 with the red certificate of feasibility as input initial complete route. Algorithm 5 tries to organize a transfer with these candidate routes. Finally, the candidate best route or a transfer is selected and appended to the partial solution. These steps are repeated until no more request can be served.

Algorithm 2: Outline of the repair phase

Input: A partial solution S of the DARP-T
Output: A solution S of the DARP-T

```
1  while requests in S can still be served do
2      Match vehicles and requests using Algorithm 3;
3      Select the vehicle with lowest current time v_1;
4      Generate a set of candidate routes W_1 for v_1 using Algorithm 4 with v_1 as input vehicle and v_1.cf
         as initial complete route if v_1 is not currently empty, otherwise with the route that starts at
         v_1.pos and serves the request matched to v_1 as input initial complete route;
5      Generate a transfer using Algorithm 5 with W_1 as input set of candidate routes;
6      if Algorithm 5 returned a transfer then
7          If v_1 is involved in a transfer with a non-destroyed route in S, partially destroy that route with
             Algorithm 6;
8          Update S by appending the routes that bring the requests to the transfer point to the entire
             routes of the involved vehicles and set the drop-off routes as certificate of feasibility.
9      else
10         Update S by appending the minimum cost route in W_1 to v_1.er;
11     Mark the requests involved in the carried out route or transfer as handled;
```

3.4.1. Vehicle-request matching algorithm

The vehicle-request matching procedure described in Algorithm 3 serves two purposes: matching idle vehicles with unserved requests and rejecting the requests that cannot be feasibly served anymore in the partial solution. The matching is found by solving the rectangular assignment problem to minimize the time required for the vehicles to pick up their request. One can imagine that solving the assignment problem is only meaningful if the vehicles have similar current time. Indeed, if the vehicles are scattered through time, few requests can be picked up by several vehicles and the assignment problem simply matches vehicles to their closest request. However, since we always move the vehicle with lowest current time in Algorithm 1, the vehicles move more or less together through time. Therefore, solving the assignment problem is expected to result in a better cooperation between vehicles by preventing short-sighted greedy moves.

Algorithm 3: Vehicle-request matching

Input: A partial solution S of the DARP-T

Output: The largest set of feasible (vehicle, request) matches that minimizes the total time spent by the vehicles travelling to (and possibly waiting for) their matched request. In the process, the algorithm also rejects the requests that cannot be feasibly served anymore

1 Create the set of empty vehicles $V_a \subset V$;
2 Create the set of available requests $R_a \subset R$ (requests for which $isHandled$ is false) ;
3 Create the vehicle-request reach-cost matrix $C \in \mathbb{R}^{|V_a| \times |R_a|}$, (c_{ij}) = time it takes for the i^{th} vehicle v in V_a (from $v.t$) to pick-up the j^{th} request r in R_a (including the waiting if v arrives sooner than $r.e$) if v can arrive at $r.p$ before $r.ld$ and if $r.gs \leq v.c$, else ∞;
4 Reject the requests that cannot be reached before their latest departure time by any vehicle by setting their attribute $isRejected$ and $isHandled$ to true. Remove them from R_a and C (delete the corresponding columns);
5 Solve the rectangular assignment problem with C as cost-matrix and remove the infeasible matchings (those with infinite cost);
6 Return the matching;

Now that all idle vehicles are matched to a request, we can select a vehicle and build candidate routes for it using the algorithm described in Section 3.4.2.

3.4.2. Candidate route generation

Algorithm 4 generates, from an initial complete route cRo, a set of candidate routes — denoted W — for a vehicle v. A set of requests we want to the routes in W to serve and a set of requests we do not want the routes in W to serve can optionally be provided.

Algorithm 4 starts with the provided initial complete route cRo and then inserts a request that can be feasibly picked-up inside cRo (i.e., not after the last drop-off of cRo) in cRo and adds this new route to W. Then, cRo is redefined to be this new route and the process is repeated. The request must be picked-up inside cRo to prevent creating a route that could be split into two complete routes. This is not desirable because it would allow the second route to skip the matching procedure described in Section 3.4.1.

Incomplete routes are also found by inserting requests into cRo without considering the feasibility of the drop-off. Therefore, the feasibility of the incomplete route relies on a future transfer. No more than one route and one incomplete route are added every iteration to keep W small (W cannot contain more routes than $2v.c$ since cRo contains one additional request every iteration and at most two routes are added every iteration). Also, adding only these two routes to W ensures that no duplicates appear in W (since at every iteration, cRo contains a different number of requests).

Algorithm 4: Candidate routes generation procedure

Input: A vehicle v, an initial complete route cRo, a set of candidate requests $candReq$ (optional), a set of requests that cannot be picked up $notCand$ (optional)
Output: A set W of candidate routes for v

```
1  Create the set of candidate routes W = {cRo};
2  Set notCand = {∅} if it is not provided;
3  Set candReq = {r | r is related to a request carried by cRo} if it is not provided;
4  Add to candReq the requests that could be picked up between the start node and the first pick-up of
     cRo;
5  Set candReq = candReq \ notCand;
6  while |candReq| > 0 do
7      Set cRo = last complete route added to W;
8      Randomly select a request r from candReq and remove it from candReq;
9      If r has already been picked up, is already in v or does not fit in v, continue;
       /* We try to insert r into cRo                                                    */
10     Try to find a route (newcRo) that serves r and the requests in cRo;
11     if newcRo exists then
12         Append newcRo to W;
13         If candR was not provided, set candR = candR ∪ r.related \ notCand;
14     else if there are less than 3 incomplete routes in W then
15         Try to find a route that only picks up r and serves the requests in cRo and store it in W (if such
             route exists);
16 Return W;
```

Now that candidate routes have been generated, we can check if we can find transfers more efficient than these candidate routes using the algorithm described in Section 3.4.3.

3.4.3. Transfer generation

Organizing a transfer adds complexity to the problem. Indeed, in order to organize a transfer, one must decide

- which vehicles are involved,
- the position (node) and time of the transfer,
- the requests the vehicles will bring to the transfer,
- the sharing of the requests between the vehicles at the transfer.

We want our transfer organizing procedure to be fast (since it is frequently called). However, there are already $2^{|V|} - |V| - 1$ different ways to select at least 2 vehicles from our fleet of $|V|$ vehicles. Therefore, we only consider a very small fraction of all the possible transfers. For example, Algorithm 5 only considers transfers involving two vehicles. As explained in Section 2.2, some locally unpromising moves are sometimes part of an optimal solution and finding the best transfer according to some local criterion was observed to not be worth the computational effort. Therefore, we settle for a transfer with a lower cost than the two routes the vehicles involved in the transfer would otherwise take. This not entirely greedy selection criterion diversifies the transfers the procedure can find.

Algorithm 5: Transfer generation procedure

Input: vehicle v_1, set of candidate routes W_1 for v_1
Output: A transfer involving two vehicles or $\emptyset$

```
1  Set bestTransfer = ∅;
2  foreach route w1 in W1, in a random order do
3      foreach vehicle v2 ≠ v1, in a random order do
4          Generate a set of candidate transfer routes W2 for v2 (more details in the text);
5          foreach route w2 in W2, in a random order do
6              foreach candidate transfer node n, in a random order do
7                  Determine where n should be inserted in w1 and w2 such that the time the vehicles
                     have to wait for each other is minimized;
8                  Set allReq to be the union of the content of both vehicles when they reach n ;
                   /* Now we are now solving the open, capacitated two vehicles VRP  */
9                  foreach way of partitioning allReq in two sets do
10                     Find the time at which the vehicles exchange their requests and leave n (if the
                         vehicle is only dropping requests, it does not have to wait for the other vehicle);
11                     Compute the drop-off routes; Continue if there are no feasible drop-off routes;
12                     if cost(transfer) ≤ min(cost(W1, W2), cost(bestTransfer)) then
13                         Set bestTransfer = transfer;
14                         Stop the procedure with probability 0.3;
15 Return bestTransfer
```

The transfer procedure is described in Algorithm 5. It takes as input a vehicle v_1 together with a set of complete and incomplete candidate routes W_1 for that vehicle. It starts by randomly selecting one candidate route w_1 for vehicle v_1 and another vehicle v_2.
Then, it creates a set of candidate transfer routes W_2 for vehicle v_2. A transfer involving v_1 and the entire route of v_2 is considered and $W_2 = \{v_2.er\}$ if $v_2.t$ is larger than the time at the last unit of w_1 and $v_2.canBeModified$ is true. If not, we have to generate the candidate transfer routes W_2 using Algorithm 4. We do so by first finding the set of requests $reqCand$ that might be interesting to bring to a transfer with w_1. More precisely, these are the requests r for which all the following statements hold:

- the group size of r fits into v_2, i.e. $r.gs \leq v_2.capacity$.
- there exists a request $q \in w_1$ such that $r.dir \cdot q.dir \geq \cos\alpha$ for some angle α. Informally, it means that r points in a direction similar to at least one request in w_1.
- $d(v_2.pos, r.p) + d(r.p, w_1) - d(v_2.pos, w_1) < 15$. Informally, it means that $r.p$ is on the way from $v_2.pos$ to w_1.
- $r.dir \cdot (z - mds(r.p)) > 0$ with z the vector in $mds(w_1)$ closest to $mds(r.p)$. Informally, it means that bringing r to w_1 brings r closer to $r.d$.

Then, we compute W_2 with Algorithm 4. v_2 is the input vehicle, $v_2.cf$ is the input initial complete route if v_2 is not currently empty, otherwise the input initial complete route is the route that starts at $v_2.pos$ and serves the request matched to v_2, $reqCand$ is the input set of candidate requests and the requests picked up by w_1 are the input set of requests that cannot be picked up. Afterwards, we randomly select a route w_2 in W_2 and find a set of candidate transfers node. These nodes are the nodes that are close to w_1 and w_2 (see Equation 3.7). Then, we randomly select a node n in the set of candidate transfer nodes and determine its insertion position in w_1 and w_2. After that, we consider a partitioning (P_1, P_2) of the requests present at the transfer node. We make sure the partitioning is feasible and passes the following quality heuristics

$$(mds(r.d) - mds(n)) \cdot (mds(q.d) - mds(n)) > 0 \ \forall r, q \in P_i, \ i \in \{1, 2\}.$$

Informally, this heuristic tests whether all requests inside the partitions have to go in the same direction. When a partitioning is accepted, the drop-off routes are computed. The quality of the transfer is then finally evaluated and accepted or rejected. Since the complete candidate routes can be executed without transfers,

they are used as a comparison to judge the quality of the transfers we find. If the transfer is accepted, a stochastic criterion decides whether to keep searching for a better transfer or not. If we keep searching, we try to organize a new transfer with a different combination of w_1, v_2, w_2, transfer node and sharing of the requests.

3.5. Destroy phase

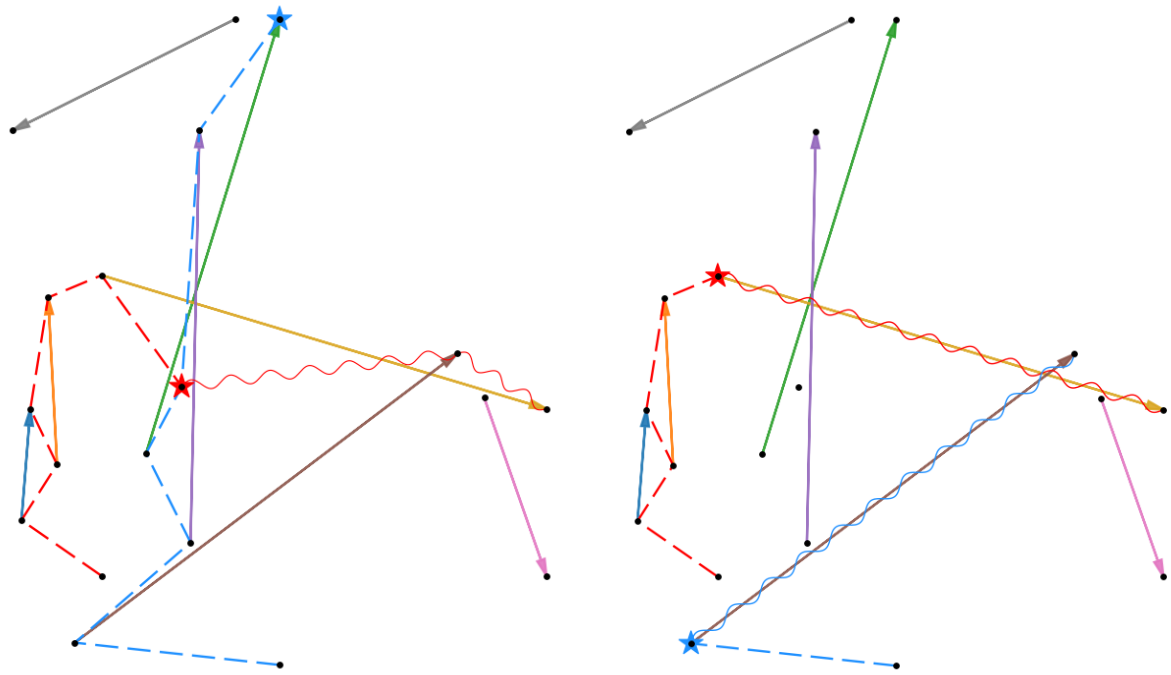
This section first describes the challenges involved with destroying parts of the solution and the basic destruction mechanism used by all destroy operators. Then, we introduce the destroy operators.
Since the time windows of the requests are tight, small modifications of the $v.er$ of a vehicle v (such as the insertion of a request along the way) can make $v.er$ infeasible. Therefore, we also destroy every unit of the route happening after the modification. If this destruction affects a transfer, the destruction is propagated to the other vehicle involved in the transfer. This mechanism is described in Algorithm 6. Unfortunately, this propagation of the destruction affects disproportionately the complex routes and the late part of the solution is more affected than the earlier part of the solution. We mitigate this problem by only solving the problem on small timescales (see Chapter 4).

Algorithm 6: Destruction mechanism

Data: A partial solution, the vehicle v corresponding to the $v.er$ we want to partially destroy and the index i of the unit of $v.er$ we want to destroy
Output: A partially destroyed partial solution
`/* The algorithm is recursive */`
Set $j = |v.er|$; $toPropagate = \emptyset$;
while $j > 0$ **do**
    Consider the j-th unit of $v.er$. If there is a transfer, remember the other vehicle involved in the transfer in $toPropagate$. If a request is being picked up from its pick-up node, set its attribute $isHandled$ to false;
    if $j \leq i$ **then**
        Try to create a certificate of feasibility for v;
        if *a certificate of feasibility is found* **then**
            Remove all units of $v.er$ with an index greater or equal to j;
            Stop the while loop;
    Decrement j;
Set $v.canBeModified$ = false with probability 0.5;
Propagate the destruction to the vehicles in $toPropagate$.

An example of the propagation of the destruction if shown in Figure 3.8. We destroy the last (the sixth) unit of the red route in Figure 3.1. At the fifth unit of the red route, the red vehicle contains the yellow request, which can be feasibly dropped-off. Therefore, the destruction of the red route can stop and the red vehicle is assigned the drop-off of the yellow route as certificate of feasibility. Then, the destruction is propagated to the blue vehicle. Since the transfer happens at the fifth unit of the blue route, everything happening after the fourth unit is destroyed. At the fourth unit, the blue vehicle contains the green, the purple and the brown requests. Unfortunately, the vehicle cannot feasibly drop-off all these requests using only ride-sharing and we do not have a certificate of feasibility. Therefore, the fourth unit is also destroyed. The third unit of the entire route is also destroyed for the same reason. At the second unit of the route, the vehicle can feasibly drop-off its content (the brown request) and the destruction stops. This is an example of destruction propagating backward in time. Note that the first unit of the route is always feasible and is therefore never destroyed.

We explain in Section 3.4 that, unlike most destroy and repair procedures, some additional destruction sometimes happens during the repair phase. Similarly, some destroy operators are not purely destroy operators as their destruction is a byproduct of a modification of the solution.



(a) Reproduction of Figure 3.1.

(b) Destroyed Figure 3.1.

Figure 3.8: Example of the propagation of the destruction.

Let us describe the destroy and repair procedure. First, the parameter Q, defined to be

$$Q(T) = \begin{cases} q & \text{with probability } 0.5 \\ \frac{T}{T_0} & \text{with probability } 0.5 \end{cases}, \tag{3.16}$$

is computed, with q a number sampled from the continuous uniform distribution $\mathcal{U}(0, 0.4)$. Q is proportional to the amount of destruction the destroy operators will carry. Therefore, we destroy more of the solution when T is high and smaller parts of the solution when T is low (see Section 3.3 for the explanations of why this is beneficial). Values of Q in $[0, 0.4]$ were observed to be likely to result in an improvement of the solution. Hence, we keep Q in that range at least 50% of the time. This also ensures Q is not always 0 when the temperature has converged to nearly 0. Then, we run all the destroy operators one by one (without repairing) in the following order:

1. *Worst removal(n)*: entirely destroy the n worst-performing entire routes that can be modified. This destroy operator is always selected and n is obtained by sampling the discrete uniform distribution $\mathcal{U}\{2, 2 + \lfloor 10Q \rfloor\}$. The term 2 ensures that this operator destroys at least two routes, even when Q is small.

2. *Zero-split(n)*: destroy at most n routes that can be modified at the tail of an arc they are traveling empty. This destroy operator is always selected and n is obtained by sampling the discrete uniform distribution $\mathcal{U}\{2, 2 + \lfloor 2Q \rfloor\}$.

3. *Greedy ride-sharing*: greedily rebuild using Algorithm 7 the entire route of a vehicle v randomly selected from V, allowing the pick-up of any request (served or rejected). When v picks up a request that is already served by another route, that other route is partially destroyed. This destroy operator is selected with probability Q^2 and builds a route in a very different manner than the repair operator, and therefore, helps diversifying the solution.

4. *Force a transfer*: using Algorithm 8 and the entire routes from the partial solution, organize a transfer. This partially destroys the routes of the involved vehicles (and potentially of other vehicles by propagation). This destroy operator is selected with probability Q.

5. *Related removal(n)*: destroy at most n routes that can be modified and containing requests related to the rejected requests in the partial solution. This destroy operator is selected with probability $\min(10Q, 1)$ and n is obtained by sampling the discrete uniform distribution $\mathcal{U}\{0, \lfloor cQ \rfloor\}$ with c the number of rejected requests in the partial solution.

6. *Force request insertion*: try to insert a request rejected in the partial solution into an existing entire route (that can be modified) from the partial solution. If the insertion is successful, update the affected entire route. This is applied to at most $\frac{Q}{2}|V|$ randomly selected vehicles.

Algorithm 7 modifies the solution by greedily rebuilding the route of a vehicle v. It starts by completely destroying the existing $v.er$ and replacing it by a route that serves a request not far from the initial position of v. Then, it finds a set of candidate requests that are likely to be insertable in $v.er$. Afterward, it tries to insert one of these candidate requests into $v.er$. If the insertion is successful, it updates $v.er$. The set of candidate requests is recomputed and this procedure is repeated until the set of candidate requests is empty. Then, we destroy the routes of the vehicles that are transporting requests that are now served by v.

Algorithm 7: Greedy ride-sharing destroy operator

Input: A vehicle v such that $v.canBeModified$ is true, a partial solution S
Output: A modified S

```
Destroy v.er completely using Algorithm 6;
Set v.er = route that serves a request close to v.pos;
Set A = {request served by v.er};
repeat
    Set reqCand = {q ∈ R | q is related to a request in v.er or a request in v.er is related to q} \ A;
    Randomly select a request r from reqCand. Remove r from reqCand and add it to A ;
    Find an entire route ro for v that serves r and the requests served by v.er; Set v.er = ro if such
      route exists;
until reqCand is empty;
Partially destroy the vehicles whose entire routes transport requests that are now served in v.er;
```

Algorithm 8 analyses the solution to find two entire routes that could be modified to create a transfer. It considers a random pair of vehicles and computes the distance between their two entire routes using the results from Section 3.1.2. If the distance is too large, it tries again with another pair. If not, it finds a set of nodes close to the two routes. Then, Algorithm 5 is used to create a transfer. If the transfer is better than the existing entire routes of the vehicles involved in the transfer, the transfer is applied to the solution and Algorithm 8 stops.

Algorithm 8: Force a transfer destroy operator

Input: A partial solution S
Output: A modified partial solution S

```
for ⌈100Q⌉ pairs of vehicles (v1, v2), v1 ≠ v2 do
    Compute the distance between v1.er and v2.er and continue if the distance is too large;
    Find a transfer trans using Algorithm 5 with w1 = v1.er, w2 = v2.er;
    Set ro1 (ro2) to be the portion of v1.er (v2.er) that will be destroyed if we carry out trans;
    if cost(trans) < cost(ro1, ro2) then
        Carry out trans;
        Break;
```

Now that the algorithm has been introduced, we experimentally test it in Chapter 4.

4

Experimental analysis and results

In this chapter, we experimentally test the algorithm from Chapter 3 and analyze the results. In Section 4.1, we generate test instances of the DARP-T. Section 4.2 investigates the accuracy of multidimensional scaling (from Section 3.1.2) for the test instances. Then, the instances are solved in Section 4.3. We provide introductory visualizations of the solutions in Section 4.3.1, we give a summary of the performance of the solutions in Section 4.3.2, we investigate the user inconvenience caused by the ride-sharing and the transfers in Section 4.3.3, we look into the influence the time and the location of the demand has on the solution quality in Section 4.3.4, we investigate the effect of the fleet size on the fleet usage in Section 4.3.5, we provide an analysis of running time of the algorithm in Section 4.3.6, and we investigate the effect of the Δ_{trans} term from Section 3.3 on the solution quality in Section 4.3.7.

4.1. Test instances generation

We did not find any public benchmark for the DARP-T. Faced with the same problem, Masson et al. (2014) tested their DARP-T solver on the DARP instances from Cordeau and Laporte (2003), since the DARP is the closest optimization problem to the DARP-T. These instances contain from 24 to 144 requests. They noticed that the structure of these instances is not really adapted to transfers. These instances do not require the high quality of service we want to provide. Indeed, the requests have to travel on average for 10.5 minutes, and half of the requests have day wide time windows and 90 minute maximal ride times. For these reasons, we generated our own test instances.

The set of nodes N represents the main train or bus station (if there is no train station) of each of the 100 most populated Dutch cities. The travel time (arc of the directed graph) between any two stations was computed using the Open Source Routing Machine (OSRM). Figure 4.1 shows an overlay of the trajectory of the $100 \cdot 99 = 9900$ routes computed by OSRM [1].

[1] http://project-osrm.org/

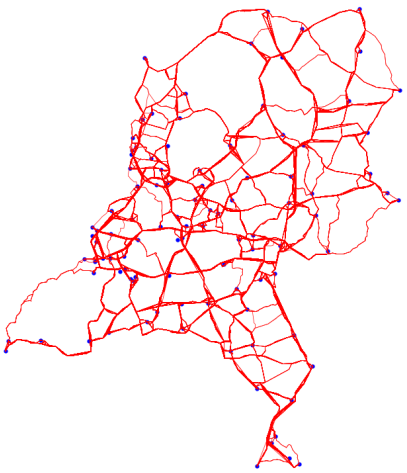


Figure 4.1: Overlay of all city-to-city trajectories computed by OSRM. Cities are represented by blue dots and the trajectories are represented in red. The Dutch road network is clearly visible.

The instances are generated using Algorithm 9. It takes as input the desired number of requests and vehicles. The time windows generated by Algorithm 9 all fit inside a 4-hour (240 minutes) timescale and no request has a travel time longer 2 hours (120 minutes). The pick-up node and drop-off node of every request are first selected randomly. Then, to reduce the number of requests with a large travel time, the drop-off node of some requests is redrawn with a probability increasing with the current travel time of the request. Each request is assigned a group size of 1, 2 or 3 with probability 0.6, 0.3 and 0.1 respectively. The earliest time of departure $r.e$ and latest time of arrival $r.l$ of a request r are chosen such that $r.l - r.e - d(r.p, r.d)$ is greater than 10 minutes but also smaller than 1 hour and $d(r.p, r.d)$. The resulting distribution of $r.e$ and $r.l$ is shown in Figure 4.2. A comparison of the resulting time window size $r.l - r.e$ with the travel time $d(r.p, r.d)$ can be seen in Figure 4.3. On average, the time windows generated with Algorithm 9 are 29 minutes longer than the travel time and are 1.52 times the travel time.

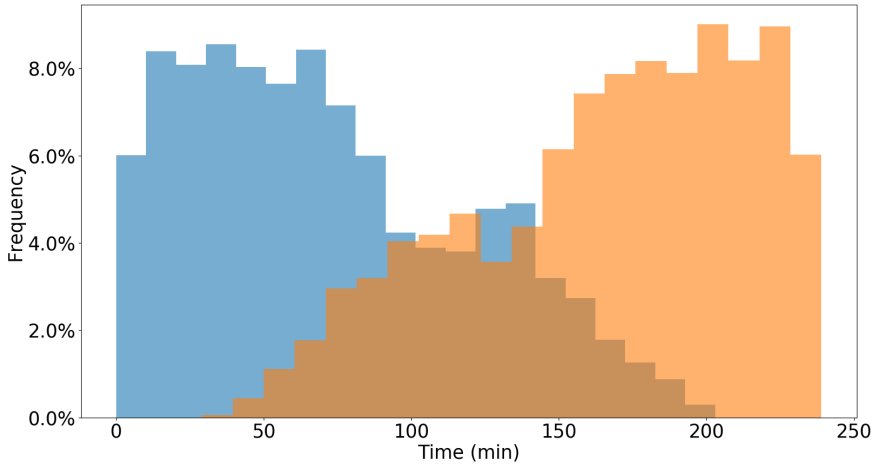


Figure 4.2: Distribution of the earliest time of departure (in blue) and latest time of arrival (in orange) of 10000 requests generated using Algorithm 9.

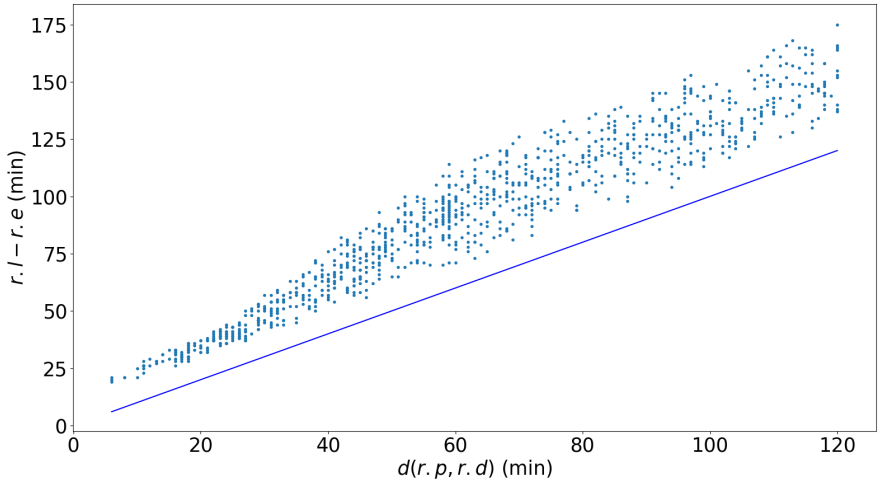


Figure 4.3: Time window size vs travel time of 1000 requests generated using Algorithm 9. The blue line is the $y = x$ line.

The vehicles are randomly assigned a capacity of 2, 4 or 6 with probability 0.2, 0.4 and 0.4 respectively. The current time of each vehicle is set to 0 and all vehicles are assigned a random current position until all requests can be reached by a vehicle before their latest time of departure plus a five minutes margin.

Algorithm 9: Instance generation algorithm

Input: Number of requests n_r, number of vehicles n_v
Output: A set of requests R, a set of vehicles V.

Create a set of n_r requests R and a set of n_v vehicles V;
Randomly assign a city to $r.p$ and to $r.d$ for every $r \in R$;
while $\exists r \in R$ *s.t.* $d(r.p, r.d) > 120$ *or* $r.p = r.d$ **do**
    If we are in the first three iterations of the while loop, redraw $r.d$ with probability $\min(1, d(r.p, r.d)/120)$ for every $r \in R$;
    For every $r \in R$, redraw $r.d$ if $d(r.p, r.d) > 120$ or if $r.p = r.d$;
foreach $r \in R$ **do**
    Set $r.gs = 1$ with probability 0.6, 2 with probability 0.3 or 3 with probability 0.1;
    /* If t_1 comes from the first (second) set, it will become $r.e$ ($r.l$). */
    Randomly select a time t_1 from the set $\{20, 21, \ldots, 104, 105\} \bigcup \{135, 136, \ldots, 219, 220\}$;
    /* t_2 is selected such that $|t_2 - t_1| = d(r.p, r.d)$ and $t_2 \in [15, 225]$. */
    Set $t_2 = \begin{cases} t_1 + d(r.p, r.d) & \text{if } t_1 \leq 105 \\ t_1 - d(r.p, r.d) & \text{otherwise} \end{cases}$;
    Set $t_e = \min(t_1, t_2)$ and $t_l = \max(t_1, t_2)$;
    /* $[t_e, t_l]$ is the smallest feasible time window for r, we pad it with t_3, t_4 */
    Randomly select two elements t_3, t_4 from the set $\{5, 6, \ldots, 10\} \cup \{5, 6, \ldots, \min(30, 0.5\, d(r.p, r.d))\}$;
    Set $r.e = \max(0, t_1 - t_3)$;
    Set $r.l = \min(240, t_2 + t_4)$;
Set $v.c = 2$ with probability 0.2, 4 with probability 0.4 or 6 with probability 0.4;
Randomly assign a city to $v.pos$ and set $v.t = 0$ for every $v \in V$;
while $\exists r \in R$ *s.t.* $r.l - d(r.p, r.d) \leq d(v.pos, r.p) + 5\ \forall v \in V$ **do**
    Redraw the positions of the vehicles;

For the computational experiments of this chapter, we generated 30 different instances: 10 with 150 requests (5 with 60 vehicles and 5 with 70 vehicles), 10 with 500 requests (5 with 175 vehicles and 5 with 180 vehicles) and 10 with 1000 requests (5 with 330 vehicles and 5 with 360 vehicles). We name the instances (number of requests)_(number of vehicles)_(instance number). For example, 150_60_3 is the third instance with 150 requests and 60 vehicles.
Note that for the instances generated by Algorithm 9, the largest vehicle capacity is 6 and the smallest group

size is 1. Therefore, no vehicle can carry more than 6 requests at once and if a solution serves all requests, we must have

$$\sum_{v \in V} \text{time } v \text{ spent driving} \geq \frac{1}{6} \sum_{r \in R} d(r.p, r.d).$$

As a result, all solutions of all instances generated by Algorithm 9 must have a cost (Equation 2.1) greater than or equal to $\frac{1}{6}$. However, note that the average group size is 1.5 and that the average vehicle capacity is 4.4.

4.2. MDS accuracy

The visualizations presented in the rest of this thesis use mds in $\mathbb{R}^2$ to display the relative positions of every node (see Section 3.1.2) in a 2D plane. The relative positions of the nodes is not always very accurate, as we observe in the example from Section 4.3.1. Therefore, in order to assess how accurate the displayed relative positions are, we compute the relative error of mds

$$\frac{\|mds(n_1) - mds(n_2)\|_2 - d(n_1, n_2)}{d(n_1, n_2)}$$

for every pair of nodes $n_1, n_2 \in N$. Given the nature of our test instances, we could also display the relative positions by projecting the geographic coordinates of every city onto a 2D plane. We compare the accuracy of the two approaches (mds in $\mathbb{R}^2$ and projection of the geographic coordinates) in Figure 4.4. One can see that the relative errors of mds are slightly less spread out. One may suspect that the relative errors can be significantly reduced by increasing the dimension of the codomain of the mds map. Figure 4.5 shows the mean and standard deviation of the relative errors for different values of the dimension of the codomain. One can see that increasing the dimension indeed improves the accuracy of the mapping. However, it also increases the computational cost of the distance computations from Section 3.1.2. Additionally, the probability that the dot product of two vectors randomly sampled from $\{v \in \mathbb{R}^d : \|v\|_2 = 1\}$ is greater than some positive value goes to 0 as d increases, causing the direction based heuristics to behave differently at higher dimensions. For these reasons, all heuristics based on mds in this thesis use the 5-dimensional mds mapping of N. The accuracy of that mapping is compared to the geographic projection in Figure 4.6.

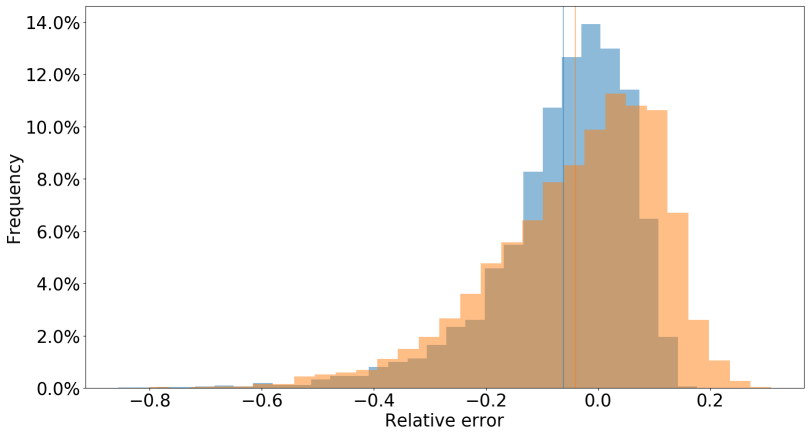


Figure 4.4: Histogram comparing the relative errors of the 2D mds map (in blue) with the relative errors of the 2D projection of the geographic coordinates of the cities (in orange). The mean is indicated by a vertical line. The blue histogram has a standard deviation of 0.12 and the orange 0.15.

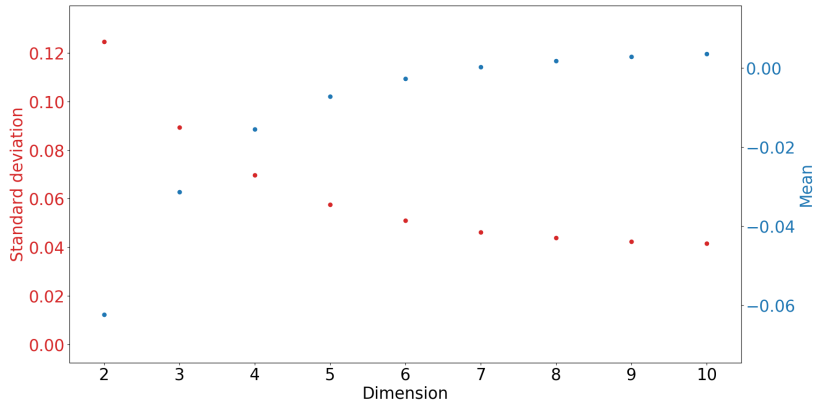


Figure 4.5: Mean and standard deviation of the relative errors of the $mds : N \to \mathbb{R}^d$ map for different values of d.

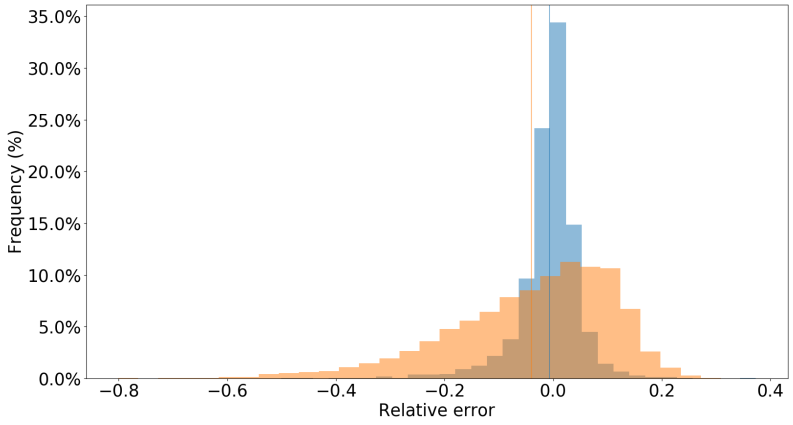


Figure 4.6: Histogram comparing the relative errors of the 5D mds map (in blue) with the relative errors of the 2D projection of the geographic coordinates of the cities (in orange). The mean is indicated by a vertical line. The blue histogram has a standard deviation of 0.06 and the orange 0.15.

4.3. Results

In order to determine the effects the transfers have on the solution, we solve the instances with and without transfers. The instances solved with transfers are said to have been solved with the DARP-T solver and the instances solved without transfers are said to have been solved with the DARP solver. The DARP-T solver is the main algorithm described in this thesis (Algorithm 1) and the DARP solver is simply the DARP-T solver without Algorithm 5 and 8.
We solve each instance three times:

1. Once with the DARP-T solver. The resulting solutions are referred to as the *t-solutions*. The solutions generated that way are named by adding (t) as a suffix to the solved instance name. For example, 150_60_3(t) refers to the solution of instance 150_60_3 obtained with the DARP-T solver.

2. Once with the DARP solver. Therefore, the resulting solutions do not contain any transfers, are referred to as the *nt-solutions* and use the (nt) suffix. For example, 150_60_3(nt) refers to the solution of instance 150_60_3 obtained with the DARP solver.

3. Once with the DARP-T solver using the nt-solution as initial guess and Equation 3.15 as temperature function. The resulting solutions are referred to as the *ntt-solutions* and use the (ntt) suffix. For example, 150_60_3(ntt) refers to the solution of instance 150_60_3 obtained with the DARP-T solver given 150_60_3(nt) as initial guess.

We generated the solutions using on an Intel® Core™ i7-8750H at 1.8 GHz, with $penalty = 0.1$, $\gamma = \frac{0.15}{|R|}$, and $T_0 = \frac{1}{150}$.

4.3.1. Solution visualizations

In this section, we introduce a few visualizations of the solutions we generated. Nodes are represented by small circles, requests by arrows, vehicle routes by wavy lines, and initial positions of the vehicles by stars. The color map on the right can be used to determine the earliest time of departure of a request (color of the tail of the arrow) and the latest time of arrival (color of the head of the arrow). The color of the vehicles is only used to differentiate them and routes share the color of their corresponding vehicle. A blacked out circle indicates that a transfer happened at that node.

Because of the large size of the instances solved, drawing all the requests, vehicles, and routes at the same time would not result in an intelligible diagram. Therefore, we only draw small subsets of the solutions. Let us call two vehicles v_1 and v_2 *connected* if there exists a sequence of vehicles starting with v_1 and ending with v_2 such that any two adjacent vehicles in the sequence are exchanging requests in the solution. If we draw the route of a vehicle involved in a transfer, it would be interesting to also draw the route of the other vehicle involved in the transfer. Therefore, a vehicle is always drawn with its connected vehicles.
In order to estimate how efficient the displayed subset of the solution is compared to the entire solution, we use Equation 2.1 only on the drawn objects, that is we compute

$$\frac{\sum_{v \in V_c} \text{time } v \text{ spent driving}}{\sum_{r \in A} d(r.p, r.d)}$$

with V_c the set of drawn connected vehicles and A the set of requests served by the vehicles in V_c.

Figure 4.7 shows a route from 500_180_1(nt). Therefore, this is a DARP (no transfer) solution of instance 500_180_1. One can see that the vehicle is fortunate to start at the pick-up node of a request with an early time window, and therefore, does not have to drive empty to its first request pick-up. Additionally, the vehicle finds many requests to carpool along its way and is never required to drive empty. Therefore, this is one of the most efficient subsets of the solution, which is confirmed by its cost of 0.600, versus 0.898 for the entire solution.

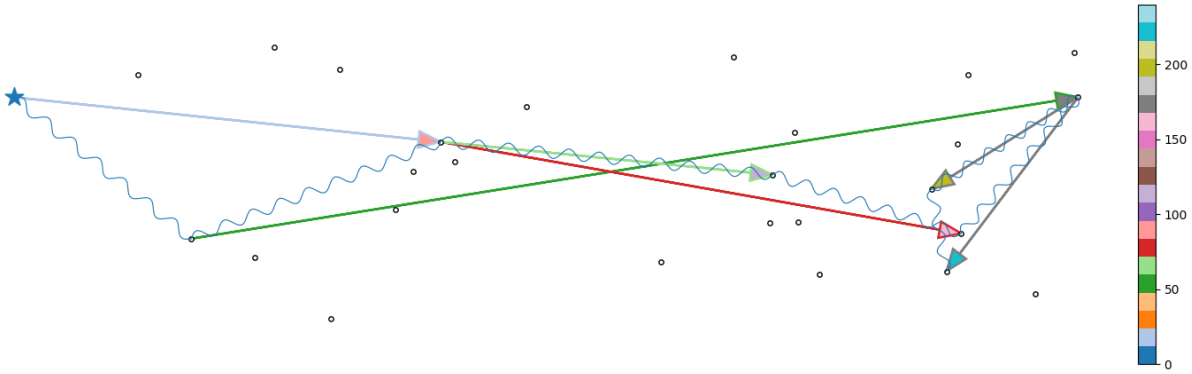


Figure 4.7: Diagram representing the routes of a vehicle in the solution of instance 500_180_1(nt).

Looking at Figure 4.8, one might question the usefulness of the transfer. Indeed, having the blue vehicle serve the two blue requests and the orange vehicle serve the orange request would result in a similar solution but without the transfer. However, the blue vehicle has a capacity of 2 and the request first picked up by the blue vehicle has a group size of 2. Therefore, it is not possible for the blue vehicle to transport the two blue requests simultaneously. The orange vehicle, having a capacity of 6, can transport the two requests. Therefore, this transfer is actually useful as it allows the two blue requests to be carpooled.
Despite the carpooling, the cost of this subset of a solution is 1.078, which is higher than 1.027, the cost of the entire solution without penalty (1.227 with penalty). Therefore, this subset of the solution is rather inefficient compared to the rest of the solution. However, this higher cost does not necessarily imply that there exists a more efficient way to serve the drawn requests.

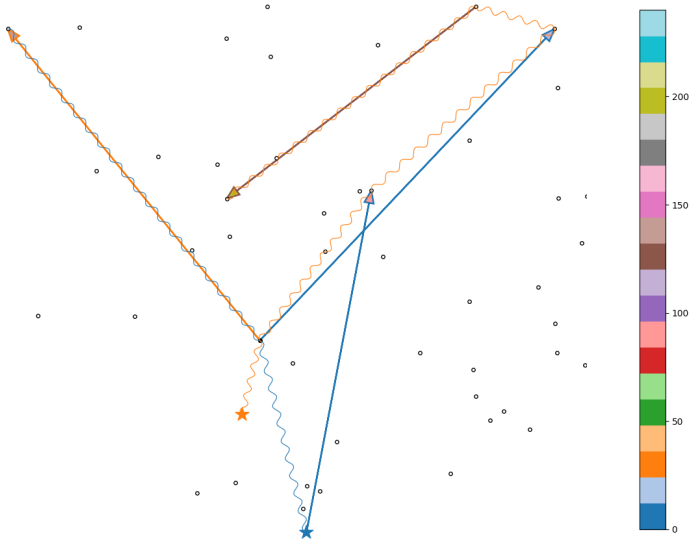


Figure 4.8: Diagram representing the routes of two connected vehicles in the solution of instance 150_60_7(t).

Figure 4.9 shows the routes of three connected vehicle from 500_180_1(t). The orange vehicle picks up the blue request, transfers it to the blue vehicle, picks up the orange request, receives the green request from the green vehicle and then carpools the orange and green requests. The blue vehicle, after receiving the blue request, is not involved in any additional transfers. The green vehicle stops after delivering the green request to the orange vehicle.
Looking at Figure 4.9, one might think that serving the orange and the green requests independently (orange vehicle serves the orange request and green vehicle serves the green request) might be more efficient. However, since the relative position of the cities can be significantly off in 2D (see Section 4.2), Figure 4.9 might be misleading. Indeed, by denoting the orange request by r_o, the green request by r_g and the transfer node by n_t, we have $d(r_o.p, r_o.d) = 90$, $d(r_o.p, n_t, r_o.d) = 97$, $d(r_g.p, r_g.d) = 97$ and $d(r_g.p, n_t, r_g.d) = 109$. Therefore, the detours caused by the transfer of the green request to the orange vehicle are not as substantial as Figure 4.9 suggests. Besides, since $d(r_o.p, n_t, r_o.d, r_g.d) + d(r_g.p, n_t) = 175 < d(r_o.p, r_o.d) + d(r_g.p, r_g.d) = 187$, the transfer is more efficient than transporting the requests independently.
Despite the complexity of the routes, the subset of the solution contains little carpooling and therefore, it is not surprising that its cost is significantly higher than the cost of the entire solution (0.974 and 0.782 respectively). Again, this does not necessarily imply that it is an inefficient way to serve the drawn requests.

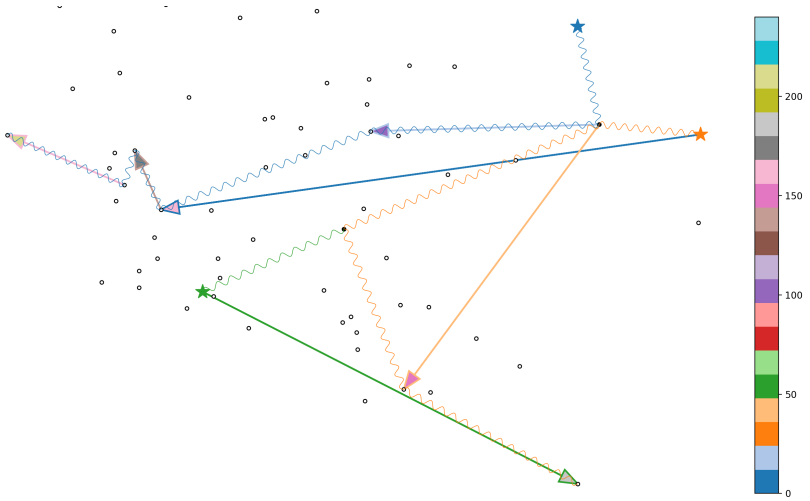


Figure 4.9: Diagram representing the routes of three connected vehicles in the solution of instance 1000_360_2(t).

4.3.2. Performance

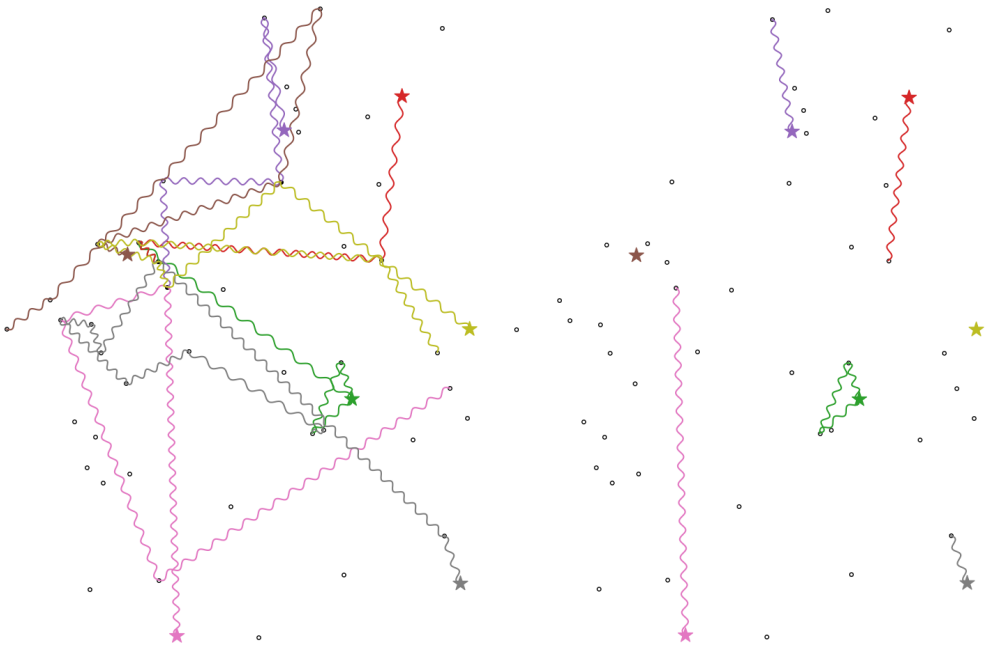
The objective function value, number of transfers, and running time corresponding to all the generated solutions are shown in Table 4.1. The ntt-solution running time does not include the time spent on the generation of the corresponding nt-solution. The relative difference between the nt-solution and the ntt-solutions and t-solutions are indicated in the corresponding gap column. The gap between two solutions that have not rejected the same number of requests is not computed as it depends on the value of $penalty$.

Since transfers are optional, the DARP-T solution space is a superset of the DARP solution space. Therefore, for a given instance, no solution of the DARP has an objective function value lower than an optimal solution of the DARP-T. Thus, we can expect the t-solutions and ntt-solutions to have a lower objective function value than the nt-solutions. One can see that in Table 4.1 that this is generally the case. Indeed, the ntt-solutions are better than the nt-solution for all 30 instances. In general, nt-solutions have the highest objective function value, followed by the t-solutions and then the ntt-solutions. The performances of the three different solutions can be summarized as follows:

- ntt-solution < nt-solution < t-solution for 8 out of the 30 instances,
- ntt-solution < t-solution < nt-solution for 12 out of the 30 instances,
- t-solution < ntt-solution < nt-solution for 10 out of the 30 instances.

The introduction of transfers in the nt-solutions has strictly improved the objective function value for all instances and has reduced the number of rejected requests for 4 of the 6 nt-solutions with rejected requests. This indicates that local minima for the DARP solver are not local minima for the DARP-T solver.

Despite its longer running times and its access to a larger solution space, the DARP-T solver fails to find better t-solutions than the nt-solutions for 8 (out of 30) instances. This suggests that the introduction of transfers introduces new local minima that are difficult to escape. Indeed, it may be that the transfers present in a solution are preventing the improvement of that solution by the destroy and repair procedure, because of the additional the vulnerability of the solution they are causing (see Section 2.2). Indeed, transfers cause routes to be very interdependent and the destruction of a single route may propagate and destroy large swathes of the solution (see Section 3.5). This can be observed in 1000_360_1(t). This solution is 0.5% worse than 1000_360_1(nt). 1000_360_1(t) contains a set of 7 connected vehicles, whose routes are shown in Figure 4.10a. Vehicles are represented by stars, cities by dots, and routes by wavy lines.



(a) Routes of the connected vehicles.

(b) Figure 4.10a after the destruction of the brown route.

Figure 4.10: Example of the propagation of the destruction.

The brown vehicle in Figure 4.10a is involved in three-vehicle transfer with the purple and the yellow vehicles. Then, the yellow vehicle exchanges requests with the pink vehicle. Afterward, the yellow vehicle participates in 4-vehicle transfer with the gray, the green, and the red vehicles. Therefore, if we destroy the route of the brown vehicle before its transfer, we also destroy the route of the yellow vehicle, which in turn destroys the routes of all the other connected vehicles. This propagation of the destruction is shown in Figure 4.10b. In Section 3.3, we argued that a large destruction of the solution is unlikely to be repaired to a better solution in a single repair iteration. Additionally, the previous repair-and-destroy iterations might have optimized the routes of the connected vehicles, decreasing even more the probability that better routes for the connected vehicles can be found in a single iteration of the repair operator. Therefore, most attempts of the repair and destroy procedure to modify the routes of the connected vehicles will be rejected. Thus, if the routes in Figure 4.10a happen to be detrimental to the quality of the entire solution, we are stuck in a local minimum.
Despite the overall superiority of the ntt-solution over the t-solutions, a third of the t-solutions are better than their corresponding ntt-solutions. Since the ntt-solutions are solved at low temperatures from elite nt-solutions, they might not be able to reach parts of the solution space containing complex multi-vehicle transfers through the repeated, gradual improvements of the entire solution required by the low temperatures. This is also suggested by the fact that the t-solutions generally contain 30% more transfers than the ntt-solutions. It is possible that these complex transfers are sometimes beneficial, advantaging the t-solutions, and sometimes trap the algorithm in a mediocre local minimum, advantaging the ntt-solutions.

One can see that the objective function values of the 150 requests solutions can be more than 40% higher than the objective function values of the 1000 requests solutions. This decrease of objective function value is observed for nt-solutions as well as for the ntt-solutions. The decrease in gap between the small and the large instances is in the order of 1%. This suggests that the conjecture of Cortés et al. (2010) that states that transfer profitability increases under high demand might be true, even though the increase in profitability attributable to the transfers is an order of magnitude smaller than the total increase in profitability due to the higher demand.
The t-solutions seem to consistently average close to 7 transfers per 100 requests. Whether this is a property of the algorithm or of the instances is unknown. The objective function value considerably decreases with an increasing number of requests. Indeed, the least-cost solution of the instances with 150 requests is 1.0034 versus 0.7467 for the 1000 requests instances. This suggests that increasing the request density increases the use of ride-sharing. Indeed, without ride-sharing, no solution can reach an objective function value lower than 1. We investigate this in more detail in Section 4.3.4. The running time of the DARP-T solver is around twice the running time of the DARP solver. Since 5.1 million destroy and repair iterations were required to generate all nt-solutions, versus 5.7 for the t-solutions, the DARP-T solver has a higher computation cost per iteration than the DARP solver. The computation cost of both solvers is investigated in more details in Section 4.3.6.

Table 4.1: Objective function value, number of transfers, and running time of all the generated solutions. The parenthesis following the objective function value indicates the number of rejected requests. The gap indicates, in percent, the relative difference between the ntt or t-solution and the nt-solution.

Instance	Objective function value					Number of transfers		Running time		
	nt	ntt	Gap	t	Gap	ntt	t	nt	ntt	t
150_70_1	1.1043 (0)	1.0811 (0)	-2.1	1.0844 (0)	-1.8	5	8	67	57	179
150_70_2	1.0200 (0)	1.0034 (0)	-1.6	1.0556 (0)	3.5	7	5	40	48	142
150_70_3	1.0445 (0)	1.0328 (0)	-1.1	1.0509 (0)	0.6	6	9	66	71	203
150_70_4	1.0502 (0)	1.0366 (0)	-1.3	1.0064 (0)	-4.2	6	13	82	63	181
150_70_5	1.1437 (0)	1.0818 (0)	-5.4	1.0992 (0)	-3.9	11	15	46	176	144
Average	1.0725 (0.0)	1.0471 (0.0)	-2.3	1.0593 (0.0)	-1.1	6.8	10.0	60	83	170
150_60_6	1.6838 (7)	1.5888 (6)	/	1.5826 (6)	/	6	10	98	73	143
150_60_7	1.2049 (2)	1.1109 (1)	/	1.2273 (2)	1.9	9	10	30	64	167
150_60_8	1.9073 (9)	1.6183 (6)	/	1.8391 (8)	/	5	10	109	103	148
150_60_9	1.5403 (5)	1.5165 (5)	-1.5	1.6801 (7)	/	7	10	60	43	139
150_60_10	1.5688 (5)	1.4522 (4)	/	1.4608 (4)	/	5	13	49	117	169
Average	1.5810 (5.6)	1.4573 (4.4)	-1.5	1.5580 (5.4)	1.9	6.4	10.6	69	80	153
500_180_1	0.8983 (0)	0.8669 (0)	-3.5	0.9160 (0)	2.0	25	29	230	391	334
500_180_2	0.8990 (0)	0.8674 (0)	-3.5	0.8920 (0)	-0.8	31	22	317	683	495
500_180_3	0.8167 (0)	0.8008 (0)	-1.9	0.8043 (0)	-1.5	20	24	266	257	676
500_180_4	0.8918 (0)	0.8430 (0)	-5.5	0.8831 (0)	-1.0	19	33	286	409	759
500_180_5	0.9069 (0)	0.8943 (0)	-1.4	0.8890 (0)	-2.0	23	46	433	250	1103
Average	0.8826 (0.0)	0.8545 (0.0)	-3.2	0.8769 (0.0)	-0.7	23.6	30.8	306	398	673
500_175_6	0.8208 (0)	0.8032 (0)	-2.1	0.8015 (0)	-2.3	30	41	1412	351	1571
500_175_7	0.8846 (0)	0.8677 (0)	-1.9	0.8663 (0)	-2.1	27	38	213	318	540
500_175_8	0.8822 (0)	0.8561 (0)	-3.0	0.8701 (0)	-1.4	30	40	362	204	746
500_175_9	0.8831 (0)	0.8535 (0)	-3.4	0.8812 (0)	-0.2	32	34	367	633	910
500_175_10	1.2830 (4)	1.2492 (4)	-2.6	1.1430 (3)	/	30	43	191	454	479
Average	0.9507 (0.8)	0.9260 (0.8)	-2.6	0.9124 (0.6)	-1.5	29.8	39.2	509	392	849
1000_360_1	0.7715 (0)	0.7467 (0)	-3.2	0.7751 (0)	0.5	54	76	1545	1448	1828
1000_360_2	0.8318 (0)	0.7785 (0)	-6.4	0.7815 (0)	-6.1	62	57	421	1581	2570
1000_360_3	0.8072 (0)	0.7839 (0)	-2.9	0.7942 (0)	-1.6	52	63	710	1043	1383
1000_360_4	0.8104 (0)	0.7955 (0)	-1.8	0.7906 (0)	-2.4	23	73	611	535	1584
1000_360_5	0.8034 (0)	0.7959 (0)	-0.9	0.7942 (0)	-1.1	26	81	1079	253	1823
Average	0.8049 (0.0)	0.7801 (0.0)	-3.1	0.7871 (0.0)	-2.2	43.4	70.0	873	972	1838
1000_330_6	0.7970 (0)	0.7782 (0)	-2.4	0.8180 (0)	2.6	54	68	1791	903	1518
1000_330_7	0.8137 (0)	0.7920 (0)	-2.7	0.7997 (0)	-1.7	47	79	1069	740	2160
1000_330_8	0.8105 (0)	0.7820 (0)	-3.5	0.7795 (0)	-3.8	55	74	724	1334	1662
1000_330_9	0.8140 (0)	0.7829 (0)	-3.8	0.7786 (0)	-4.4	49	62	725	1105	2483
1000_330_10	0.7865 (0)	0.7591 (0)	-3.5	0.7912 (0)	0.6	48	63	884	1253	1826
Average	0.8043 (0.0)	0.7788 (0.0)	-3.2	0.7934 (0.0)	-1.3	50.6	69.2	1039	1067	1930

4.3.3. User inconvenience

This section investigates the user inconvenience for the different generated solutions by looking into the delays caused by the ride-sharing and transfers. Given a request r, we define the delay to be the drop-off time minus the pick-up time minus $d(r.p, r.d)$. Similarly, the relative delay is the delay divided by $d(r.p, r.d)$.
The objective function does not include any term accounting for the user inconvenience. However, efficient vehicle routing is expected to correlate with efficient request routing. Additionally, the time windows of the requests only allow for limited delays. Indeed, on average, the time windows of the generated do not allow for delays exceeding 29 minutes and relative delays greater than 52% (see Section 4.1).
The percentage of the requests that are not delayed, the percentage of the requests that have been transferred, the average delay and the average relative delay for all solutions are shown in Table 4.2. One can see that the average delays are much lower than what is allowed by the time windows. Indeed, on average, the delay caused by the ride-sharing and the transfers is typically around 10 minutes and the relative average

delay ranges from 8 to 16%. However, since many requests are transported without delays (first column), the fraction of the requests that are delayed are more severely delayed than the average of the requests.
In our case, ride-sharing necessarily delays at least one of the requests sharing the ride (unless they share the same pick-up node and drop-off node). Indeed, there are no three distinct $n_1, n_2, n_3 \in N$ such that $d(n_1, n_2, n_3) = d(n_1, n_3)$. For the same reason, a transferred request must also be a delayed request. Therefore, it is not surprising to observe in Table 4.2 that the t-solutions and the ntt-solutions generally have a 1 to 3% greater average delay and a higher rate of delayed requests. Thus, transfers increase the user inconvenience in the solutions we generated.
We observe that fewer requests are transported without delay as the instances get larger. Similarly, the average delays increase with the size of the instance. Since delays must come from ride-sharing and transfers, we deduce that there is more ride-sharing and transfers in large instances. This is consistent with the lower objective function value of the large instances observed in Figure 4.1 that also suggests a more efficient usage of the vehicles. We investigate the effects demand has on the solution in more detail in Section 4.3.4.

Table 4.2: Percentage of the requests that have been transported without delay, percentage of the requests that have been transferred, average delay over all requests and average relative delay over all requests, for all solutions.

Instance	Without delay (%)			Transferred (%)		Average delay (min)			Average relative delay (%)		
	nt	ntt	t	ntt	t	nt	ntt	t	nt	ntt	t
150_70_1	63.4	60.0	57.5	3.3	6.7	6.6	7.4	7.6	8.8	9.5	10.8
5.0 5 150_70_2	61.8	55.6	54.1	4.7	3.3	7.1	8.5	7.7	9.6	11.7	10.2
9.0 9 150_70_3	60.7	54.5	52.7	4.7	6.0	7.0	8.2	8.1	8.6	11.4	10.8
15.0 15 150_70_4	56.8	56.8	47.6	4.0	8.7	6.5	6.8	8.2	8.9	9.6	11.6
20.0 20 150_70_5	63.5	52.7	51.0	9.3	12.7	7.0	9.0	9.1	9.9	13.0	13.0
Average	61.3	55.9	52.6	5.2	7.5	6.8	8.0	8.1	9.2	11.0	11.3
10.0 10 150_60_6	58.6	52.1	53.9	3.3	6.7	7.4	8.2	8.5	9.9	11.2	11.7
12.0 12 150_60_7	61.7	54.2	53.1	7.3	8.0	7.4	8.6	8.9	9.7	11.8	12.1
11.0 11 150_60_8	72.1	63.0	62.1	3.3	7.3	4.4	5.6	6.5	6.0	7.6	8.8
12.0 12 150_60_9	64.5	54.3	51.1	4.7	8.0	6.5	7.9	8.3	8.8	10.8	12.1
14.0 14 150_60_10	66.7	61.7	57.4	3.3	8.7	5.2	6.0	6.7	7.9	8.9	9.5
Average	64.7	57.1	55.5	4.4	7.7	6.2	7.2	7.8	8.5	10.0	10.8
35.0 35 500_180_1	47.3	43.1	43.8	5.6	6.6	9.6	10.5	9.6	13.8	15.0	14.1
27.0 27 500_180_2	49.5	41.9	44.5	7.0	5.0	8.8	10.3	9.3	12.1	14.4	13.0
29.0 29 500_180_3	45.3	40.9	42.3	4.2	5.4	10.0	10.6	10.5	13.6	14.5	14.1
40.0 40 500_180_4	44.2	41.9	40.7	4.4	7.4	9.6	10.0	10.7	14.0	14.5	15.3
58.0 58 500_180_5	49.7	45.8	46.0	5.4	11.0	9.5	10.2	10.2	13.2	14.1	14.2
Average	47.2	42.7	43.4	5.3	7.1	9.5	10.3	10.1	13.3	14.5	14.2
45.0 45 500_175_6	45.8	42.5	41.2	7.8	8.2	9.4	10.3	10.7	13.3	14.7	15.3
49.0 49 500_175_7	50.1	44.3	42.5	6.2	9.0	9.2	10.2	10.2	12.4	14.2	14.1
46.0 46 500_175_8	49.1	42.9	38.4	6.8	9.2	9.1	10.2	10.4	12.3	14.3	14.5
39.0 39 500_175_9	47.7	41.2	40.5	7.2	7.6	9.7	11.0	10.7	13.3	15.3	15.1
57.0 57 500_175_10	51.8	42.4	42.5	7.0	11.0	9.2	10.5	10.0	12.9	15.4	13.9
Average	48.9	42.7	41.0	7.0	9.0	9.3	10.4	10.4	12.8	14.8	14.6
97.0 97 1000_360_1	43.5	37.4	37.8	7.4	9.4	9.8	11.0	11.1	13.7	15.5	15.8
69.0 69 1000_360_2	44.8	37.4	38.0	7.4	6.7	10.0	11.5	11.0	14.2	16.2	15.4
86.0 86 1000_360_3	44.6	39.7	39.4	6.0	7.8	9.9	10.7	10.9	13.9	15.2	15.5
93.0 93 1000_360_4	44.6	42.2	37.2	2.8	9.0	9.9	10.5	11.1	14.1	15.0	16.3
103.0 103 1000_360_5	43.4	42.1	39.5	3.2	9.9	9.9	10.3	10.6	14.5	15.0	15.3
Average	44.2	39.7	38.4	5.4	8.6	9.9	10.8	10.9	14.1	15.4	15.6
87.0 87 1000_330_6	44.4	40.4	40.7	6.7	8.4	9.9	10.8	10.5	14.0	15.6	15.2
97.0 97 1000_330_7	46.3	41.3	38.0	6.2	9.2	9.3	10.3	10.7	13.4	15.0	15.8
97.0 97 1000_330_8	42.4	37.8	39.0	7.4	9.0	9.7	10.7	10.6	13.9	15.5	15.3
91.0 91 1000_330_9	44.2	41.5	38.9	6.2	8.4	9.6	10.4	10.6	13.3	14.5	14.7
77.0 77 1000_330_10	42.9	39.1	40.1	5.5	7.6	10.0	10.8	10.9	14.0	15.0	15.2
Average	44.0	40.0	39.3	6.4	8.5	9.7	10.6	10.7	13.7	15.1	15.2

4.3.4. Influence of demand

If a nationwide dial-a-ride service were to exist and be widely used, one can imagine that $|N| \gg 100$ and that few nodes in N would be isolated. One can see in Figure 4.1 that N consists of a central zone with a high density of nodes and an outer zone where nodes are scattered. Since the positions of the pick-up nodes are randomly selected (see Algorithm 9), the node density is proportional to the request density. One can imagine that requests that have to be transported through a low-density zone are less likely to be transported in an efficient route due to the lack of opportunity for transfers and carpooling (lack of transfer nodes and requests). Therefore, it would be interesting to quantify how the efficiency of the vehicles is affected by the density of the zone in which they are driving.

A real dial-a-ride service would also most likely be available 24/7 and there would always be cars transporting requests. However, the test instances only consider a four-hour timescale in which all vehicles are initially empty and must be empty at the end. Therefore, one can imagine that the efficiency of the vehicle routes will be poor at the beginning and at the end of the timescale. Indeed, unless a vehicle is initially located at

the pick-up node of the first request it is picking up, it will first have to drive empty to the pick-up node of that request. Similarly, unless a vehicle is transporting several requests having the same drop-off node, it will drive to its last node without ride-sharing (there is no depot constraint). This can also be seen in Figure 4.2: the number of new requests to pick up rapidly decreases after $t = 100$. Therefore, it would be interesting to quantify how the efficiency of the vehicles is affected by the start and by the end of the timescale. This could provide insight on the improvements that can be expected from solving instances with larger timescales and also the influence changes in demand (rush hour, night) can cause to the solution.
For some solution S, let us define $V_M(t) \subset V$, the set of vehicles in motion at time t and $R_{in}(v,t)$, the number of requests inside vehicle v at time t. We define the instantaneous efficiency of S at time t to be the number of requests inside a moving vehicle at time t divided by the number of vehicles moving at time t, that is

$$\eta(t) = \frac{\sum_{v \in V_M(t)} R_{in}(v,t)}{|V_M(t)|}. \tag{4.1}$$

Informally, $\eta(t)$ is the average number of requests per moving vehicle at time t.
In order to study how the density of the zone in which a vehicle is driving influences the route efficiency, we partition N into two sets: the set of requests in a dense zone D and the set of requests in an isolated zone I. Informally, we mean by dense zone a zone that many requests have to travel trough when going from their pick-up node to their drop-off node. Therefore, let us define the centroid c of N to be

$$c = \frac{1}{|N|} \sum_{n \in N} mds(n)$$

and define D to be the set of the 50 cities closest to c and I the set of the 50 cities farthest from c. Formally, the sets D and I are the sets such that $D \bigcup I = N$, $|D| = 50$, $|I| = 50$ and

$$\|c - mds(d)\|_2 \leq \|c - mds(i)\|_2 \quad \forall d \in D, i \in I.$$

We define the dense zone to be the set $Z_D = \{x \in \mathbb{R}^d \mid \|x - c\|_2 \leq \max_{d \in D} \|d - c\|_2\}$ and the isolated zone to be $Z_I = \mathbb{R}^d \setminus Z_D$. The two sets, the zone boundary, and the centroid are shown in Figure 4.11.

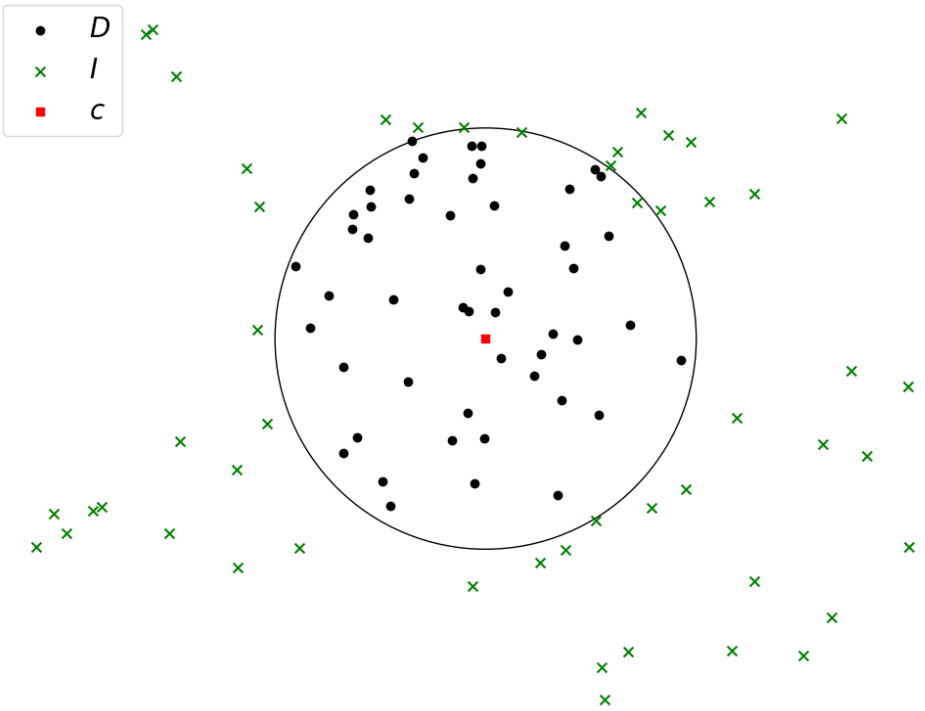


Figure 4.11: Visualization in 2D of D and I and of the boundary of the zones (black circle). Each city in D is represented by a black circle and every city in I by a green cross. The 2D centroid is represented by a red square. The presence of a green cross inside the circle is due to the fact that D and I were generated using the more accurate, 5-dimensional mds map.

In order to measure the efficiency of the vehicles depending in which zone they are driving, we use $V_M^D(t)$ ($V_M^I(t)$) to denote the set of vehicles in motion inside Z_D (Z_I) at time t and compute the instantaneous efficiency per zone using

$$\eta_D(t) = \frac{\sum_{v \in V_M^D(t)} R_{in}(v,t)}{|V_M^D(t)|}, \quad \eta_I(t) = \frac{\sum_{v \in V_M^I(t)} R_{in}(v,t)}{|V_M^I(t)|}.$$

Plots of $\eta_D(t)$ and $\eta_I(t)$ are shown for the three types of instances in Figure 4.12 (150 requests), Figure 4.13 (500 requests) and Figure 4.14 (1000 requests). The influence of the empty start and empty end is visible in all three figures as they all start with an instantaneous efficiency close to 0 (some vehicles already pick up a request at $t = 0$) that then increases to plateau from around $t = 90$ to $t = 150$ before decreasing to 1. This indicates that an increase in demand will be followed by an increase in efficiency and that a decrease in demand will be followed by a decrease in efficiency. However, note that the low efficiency at the beginning and at the end of the timescale only affects the vehicles in motion. Therefore, the effect of the low efficiency on the cost of the entire solution is not as large as the figures might suggest.
In the 150 requests case, η_D takes around 50 minutes to reach an efficiency of one and plateaus at an efficiency of 1.25. Note that this does not mean that many vehicles drive 50 minutes before picking up a request. Indeed, as can be seen from the number of vehicles moving, many vehicles are stationary at the beginning of the timescale as they wait for requests to be available for pick-up. In the 500 requests case, η_D reaches one in around 40 minutes and plateaus at around 1.6. In the 1000 requests case, these numbers further improve to 30 minutes and 1.7. Since the larger instances have a higher request density, we conclude that the vehicle efficiency increases with the request density.

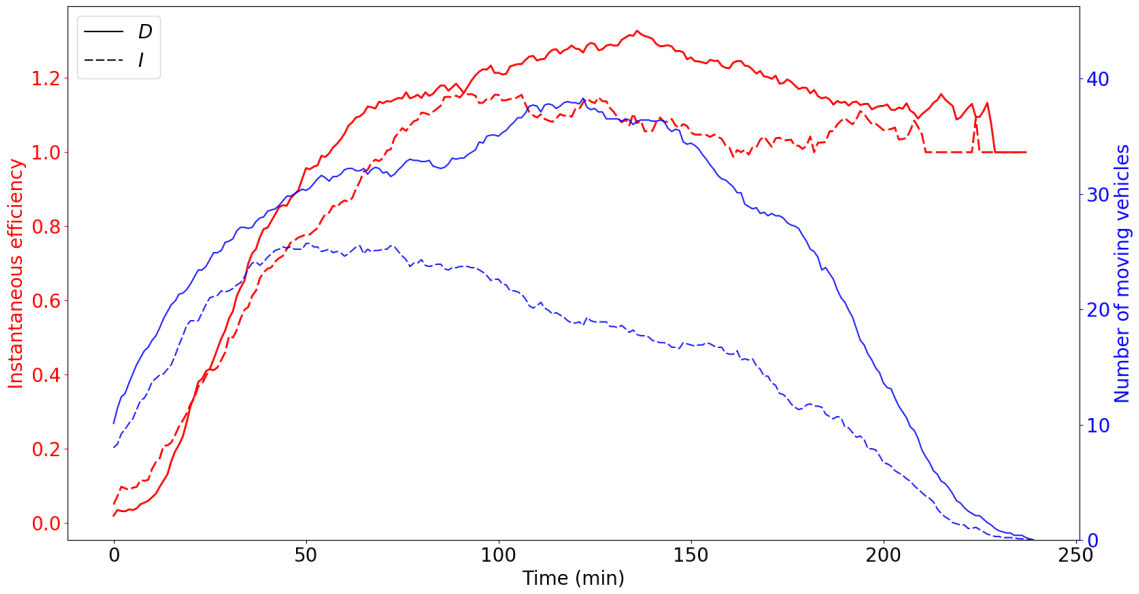


Figure 4.12: Instantaneous efficiency η_D (red line), η_I (red dashed line) and total number of vehicles V_M^D (blue line), V_M^I (blue dashed line) vs time for the ntt-solutions of the instances with 150 requests.

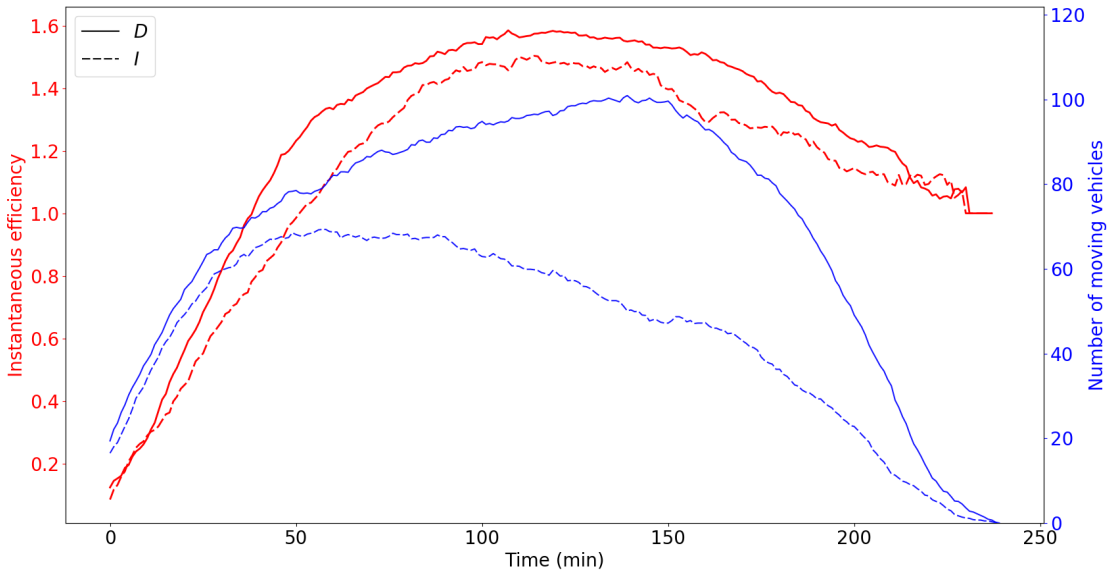


Figure 4.13: Instantaneous efficiency η_D (red line), η_I (red dashed line) and total number of vehicles V_M^D (blue line), V_M^I (blue dashed line) vs time for the ntt-solutions of the instances with 500 requests.

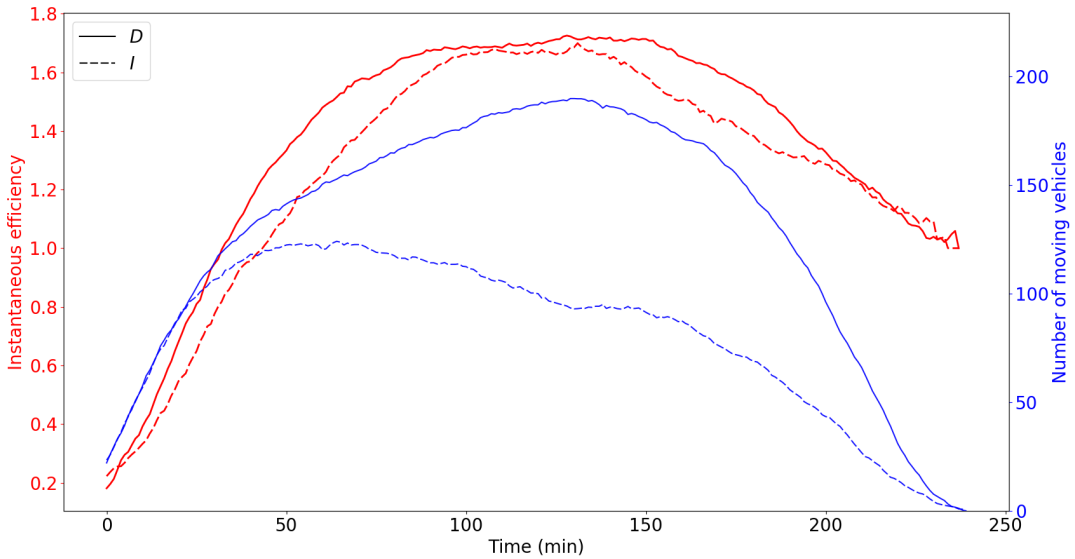


Figure 4.14: Instantaneous efficiency η_D (red line), η_I (red dashed line) and total number of vehicles V_M^D (blue line), V_M^I (blue dashed line) vs time for the ntt-solutions of the instances with 1000 requests.

The effect the zone a vehicle is driving in has on the instantaneous efficiency is visible on the three figures and the vehicles moving in zone I have a noticeably lower instantaneous efficiency throughout time. In order to quantify the global effect the zones have on the solutions, we define a function similar to Equation 4.1 that measures the average efficiency of the entire solution given a zone. The average efficiency of a solution S over a zone Z is the average (over the entire timescale) number of requests inside a vehicle moving in zone Z, divided by the average (over the entire timescale) number of vehicles moving. This can expressed as

$$\eta_{avg}^{Z}(S) = \frac{\int_0^{240} \sum_{v \in V_M^Z(t)} R_{in}(v,t)\,dt}{\int_0^{240} |V_M^Z(t)|\,dt}. \tag{4.2}$$

Note that if no request is rejected we must have

$$cost(S) \geq \frac{1}{\eta_{avg}^{D\cup I}(S)}. \tag{4.3}$$

Indeed, we have

$$\int_0^{240} |V_M(t)|dt = \sum_{v\in V} \text{time } v \text{ spent driving}$$

and since every served request spends at least $d(r.p, r.d)$ minutes in a moving vehicle, we have

$$\sum_{r\in R} d(r.p, r.d) \leq \int_0^{240} \sum_{v\in V_M(t)} R_{in}(v,t)dt$$

Therefore, we must have

$$\frac{\sum_{v\in V} \text{time } v \text{ spent driving}}{\sum_{r\in R} d(r.p, r.d)} \geq \frac{\int_0^{240} |V_M(t)|dt}{\int_0^{240} \sum_{v\in V_M(t)} R_{in}(v,t)dt}$$

which is equivalent to Equation 4.3. Due to $penalty$, Equation 4.3 also holds when requests are rejected. Since

$$\int_0^{240} \sum_{v\in V_M(t)} R_{in}(v,t)dt - \sum_{r\in R} d(r.p, r.d)$$

corresponds to the delays due to the detours caused by the transfers and the ride-sharing, the bound is tight when all requests are directly transported to their drop-off nodes.

The average efficiency per zone of the ntt-solutions and the nt-solutions solutions is shown in Table 4.3. One can see that the efficiency is significantly (12-18%) higher in the dense zone. Therefore, we can conclude that isolated zones have a considerable negative effect on the solution quality. We also observe that the gap for the ntt-solutions is 1 to 2% greater than the gap for the nt-solutions. This suggests that the introduction of transfers is more profitable in the dense zone than in the isolated zone and provides some experimental evidence for the conjecture of Cortés et al. (2010) that states that transfer profitability increases under high demand.

Table 4.3: Average efficiency per zone of the for the ntt-solutions and the nt-solutions

Number of requests	ntt			nt		
	η_{avg}^I	η_{avg}^D	Gap (%)	η_{avg}^I	η_{avg}^D	Gap (%)
150	0.906	1.068	17.9	0.893	1.028	15.1
500	1.172	1.345	14.8	1.141	1.292	13.2
1000	1.323	1.494	13.0	1.275	1.434	12.5

4.3.5. Fleet usage

We define the occupancy of a vehicle to be the number of passengers inside a vehicle divided by the capacity c of the vehicle. We give the average distance traveled by the vehicles and average occupancy of the vehicles for the different ntt-solution families and vehicle capacities in Table 4.4.
One can see that, for a fixed number of requests, decreasing the number of vehicles increases the average distance traveled by the vehicles. Therefore, as expected, reducing the size of a fleet increases the usage of the fleet. Interestingly, this increase of the usage of the fleet also comes with a general decrease of the occupancy of the vehicles. This suggests that to serve all requests, the smaller fleet has to take routes with lower occupancy. However, note that since the objective function is independent of the group size of the requests, a lower vehicle occupancy does not necessarily imply a lower objective function value.

One can see that, for all solution families, the capacity of a vehicle is positively correlated with the distance this vehicle drives. This correlation is especially strong for the vehicles of capacity 2. Additionally, the smaller fleets have a proportionally higher usage of the vehicles of capacity 2 than the larger fleets. This can be explained by the fact vehicles of capacity 2 can not carpool many passengers (and therefore requests), favoring the use of larger vehicles in elite solutions and that, when the fleet is reduced, a higher use of vehicles of capacity 2 is necessary to serve all requests.

Table 4.4: Average distance traveled by the vehicles and average occupancy of the vehicles for the different ntt-solution families and vehicle capacity.

|R|	|V|	Average distance (min)			Average occupancy (%)		
		$c=2$	$c=4$	$c=6$	$c=2$	$c=4$	$c=6$
150	70	134	143	148	55.3	40.1	25.7
150	60	161	167	165	53.3	36.6	26.6
500	180	137	157	160	60.5	46.9	36.2
500	175	140	160	163	59.4	47.4	34.6
1000	360	111	145	152	68.0	52.3	41.7
1000	330	130	155	161	66.2	51.4	40.3

4.3.6. Running time

This section experimentally investigates the influence of the instance size, the search for transfers, and Q (see Section 3.5) on the running time of the destroy and repair procedure. The results of the experiment are shown in Figure 4.15. The parameter Q determines how much destruction is applied to the solution and we see, as expected, that increasing Q increases the computation cost of an iteration for all presented cases. The DARP solver iterations seem to be around 40% cheaper than the DARP-T solver iterations for all instances size. This suggests that the solver spends 40% of its computation time looking for transfers. The red and green (dashed and full) lines seem to suggest that, for large enough instances, the size of the instance has little effect on the cost of a destroy and repair iteration. This might come as a surprise, but since the amount of destruction is mostly independent of the size of instance and since most procedures used in Algorithm 1 bound the number of cases considered independently of the size of the instance (for example, the number of routes Algorithm 4 generates does not exceed two times the capacity of the vehicle it is considering), this is to be expected. However, this also means that since the amount of change is independent of the instance size, a larger instance will require more iterations before converging to a solution. This is observed in Table 4.1.

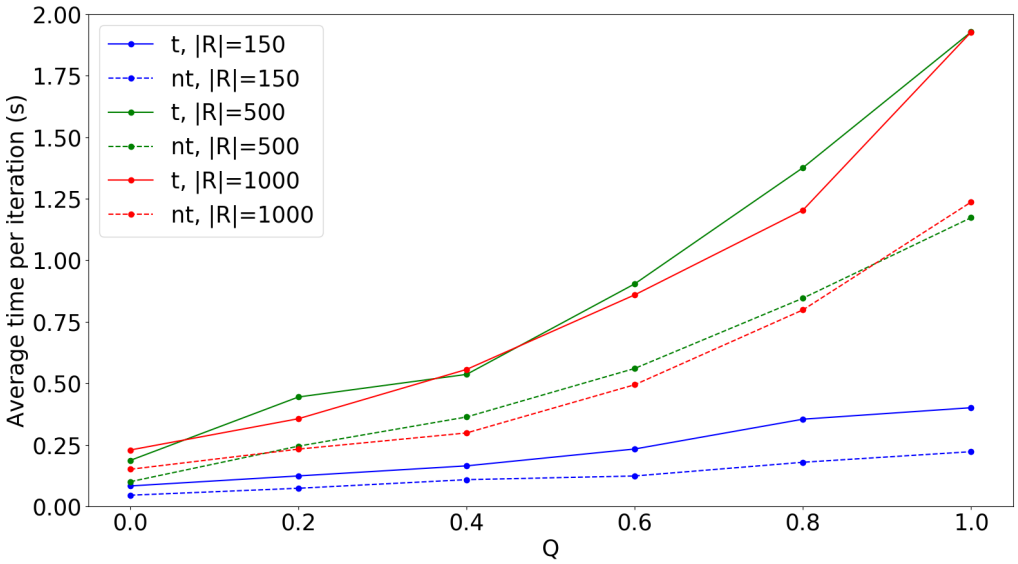


Figure 4.15: Average computation time per iteration of the repair vs Q and destroy procedure for different instance sizes. The dashed line indicate that the DARP solver was used and the full lines that the DARP-T solver was used.

4.3.7. Effect of Δ_{trans} term

In this Section, we investigate the effect the Δ_{trans} term from Equation 3.3 has on the solution quality. We run all 150 requests instances twice, once with $\gamma = 0$ (which is equivalent to removing the Δ_{trans} term) and once with $\gamma = \frac{0.15}{|R|}$. The resulting objective function values and number of transfers are shown in Table 4.5. One can see than the $\gamma = \frac{0.15}{|R|}$ solutions are better that the $\gamma = 0$ solutions for 7 out of the 10 instances. This suggests that the Δ_{trans} term is indeed improving the solution. Additionally, the $\gamma = \frac{0.15}{|R|}$ solutions contain in total 103 transfers, versus 68 for the $\gamma = 0$ solutions. This shows that Δ_{trans} term has the intended transfer-introducing effect on the solution.

Table 4.5: Objective function values and number of transfers for solutions generated with $\gamma = 0$ and $\gamma = \frac{0.15}{|R|}$. The parenthesis following the objective function value indicates the number of rejected requests. The gap indicates, in percent, the relative difference between the $\gamma = 0$ (reference) solutions and the $\gamma = \frac{0.15}{|R|}$ solutions.

Instance	Objective function value			Number of transfers	
	$\gamma = \frac{0.15}{\|R\|}$	$\gamma = 0$	Gap (%)	$\gamma = \frac{0.15}{\|R\|}$	$\gamma = 0$
150_70_1	1.0844 (0)	1.0928 (0)	0.8	8	4
150_70_2	1.0556 (0)	1.0424 (0)	-1.3	5	2
150_70_3	1.0509 (0)	1.0715 (0)	1.9	9	5
150_70_4	1.0064 (0)	1.0396 (0)	3.2	13	11
150_70_5	1.0992 (0)	1.1190 (0)	1.8	15	7
150_60_6	1.5826 (6)	1.5867 (6)	0.3	10	5
150_60_7	1.2273 (2)	1.2215 (2)	-0.5	10	8
150_60_8	1.8391 (8)	1.7561 (7)	/	10	5
150_60_9	1.6801 (7)	1.8725 (9)	/	10	11
150_60_10	1.4608 (4)	1.6429 (6)	/	13	10

5

Conclusion and recommendations

Automated vehicles have the potential to revolutionize transportation by creating a future in which transportation is an on-demand service. This model of transportation has the potential to reduce traffic, free-up parking space, be safer, and reduce the environmental impact of traveling. In this thesis, we addressed the problem of the routing of a fleet of vehicles by developing a greedy randomized destroy and repair heuristic that optimizes the routes of a heterogeneous fleet of vehicles allowed to carpool and exchange passengers. The heuristic attempts to minimize the total distance driven by the vehicles while maximizing the number of served requests. We created realistic instances of the problem containing up to 1000 requests traveling between any two of the 100 most populated cities in the Netherlands inside a four-hour timescale. The time windows in which the requests can be transported were designed to be competitive with public transportation. We showed that, even for short (four hour long) scenarios in which requests ask to travel between two cities that are on average 60 minutes apart, our fleet can satisfy the demand using a third of the number of vehicles that would otherwise be required if all requests independently traveled using their own vehicle, and reduces the total distance driven by the vehicles by up to 25%.

One of the goals of this thesis was to study the influence transfers have on the solution and all instances were solved with and without transfers for comparison. The solutions with transfers we found are typically 1 to 4% better than the solutions without transfers, and help reducing the number of rejected requests in the scenarios in which the fleet of vehicles is too small to serve all requests. The search for transfers typically increases the running time by a factor of 2 to 3. We found out that a first optimization round without transfers followed by a second optimization round with transfer was a successful optimizing strategy giving better results than a single optimization round with transfers in a comparable amount of time. In future works, we recommend introducing transfers only in a late stage of the optimization and we recommend looking into using the existing DARP solvers as initial guess generators.

We observed that the introduction of transfers causes some routes to be interdependent and that this interdependence can cause modifications of a single route to propagate to many other routes. This sensitivity of the solution to modifications is undesirable in real-life applications as it may cause a single mishap to delay numerous passengers. We suspect that these large sets of interdependent routes are trapping our destroy and repair algorithm into local minima it is unable to escape. In future works, we recommend also optimizing for solution robustness and we recommend developing new strategies designed to re-optimize sets of interdependent routes.

Despite an objective function not accounting for user inconvenience, we observed in our solutions that the delays caused by the transfers and the ride-sharing are significantly lower than those allowed by the time windows. We showed that the transfers and the ride-sharing extend the travel time of the requests by, on average, 8 to 16%. The transfers are, on average, responsible for 1 to 3% of that travel time extension.

We provided some experimental evidence for the conjecture of Cortés et al. (2010) that states that transfer profitability increases under high demand. However, the increase in profitability under a higher demand attributable to the transfers is an order of magnitude smaller than the total increase in profitability due to the higher demand. We observed that, for a fixed number of requests, decreasing the fleet size increases the fleet usage but also decreases the occupancy of the vehicles. We recommend looking into the benefits a large vehicle fleet can bring in future works.

To our knowledge, this thesis is the first work introducing and solving DARP-T instances with more than 200 requests. This thesis also distinguishes itself from the existing literature through its demanding time windows. We believe that the heuristic presented in this thesis brings us closer to solving real, very large scale instances of the DARP-T.

Bibliography

Sohaib Afifi, Duc-Cuong Dang, and Aziz Moukrim. A simulated annealing algorithm for the vehicle routing problem with time windows and synchronization constraints. In Giuseppe Nicosia and Panos Pardalos, editors, *Learning and Intelligent Optimization*, pages 259–265. Springer Berlin Heidelberg, 2013.

Gerardo Berbeglia, Jean-François Cordeau, and Gilbert Laporte. A hybrid tabu search and constraint programming algorithm for the dynamic dial-a-ride problem. *INFORMS Journal on Computing*, 24(3):343–355, 2012.

Ingwer Borg and Patrick J. F. Groenen. *Modern Multidimensional Scaling Theory and Applications*. Springer, 2005.

Jean-François Cordeau and Gilbert Laporte. The dial-a-ride problem: models and algorithms. *Annals of Operations Research*, 153(1):29–46, 2007.

Jean-François Cordeau. A branch-and-cut algorithm for the dial-a-ride problem. *Operations Research*, 54(3): 573–586, 2006.

Jean-François Cordeau and Gilbert Laporte. A tabu search heuristic for the static multi-vehicle dial-a-ride problem. *Transportation Research Part B: Methodological*, 37(6):579 – 594, 2003.

Cristián E. Cortés, Martín Matamala, and Claudio Contardo. The pickup and delivery problem with transfers: Formulation and a branch-and-cut solution method. *European Journal of Operational Research*, 200(3): 711 – 724, 2010.

Hussein Dia and Farid Javanshour. Autonomous shared mobility-on-demand: Melbourne pilot simulation study. *Transportation Research Procedia*, 22:285 – 296, 2017.

Daniel J. Fagnant and Kara Kockelman. Preparing a nation for autonomous vehicles: opportunities, barriers and policy recommendations. *Transportation Research Part A: Policy and Practice*, 77:167 – 181, 2015.

Daniel J. Fagnant and Kara M. Kockelman. The travel and environmental implications of shared autonomous vehicles, using agent-based model scenarios. *Transportation Research Part C: Emerging Technologies*, 40:1 – 13, 2014.

International Transport Forum. Urban mobility system upgrade. (6), 2015.

Baoxiang Li, Dmitry Krushinsky, Tom Van Woensel, and Hajo A. Reijers. An adaptive large neighborhood search heuristic for the share-a-ride problem. *Computers & Operations Research*, 66:170 – 180, 2016.

Renaud Masson, Fabien Lehuédé, and Olivier Péton. An adaptive large neighborhood search for the pickup and delivery problem with transfers. *Transportation Science*, 47(3):344–355, 2013.

Renaud Masson, Fabien Lehuédé, and Olivier Péton. The dial-a-ride problem with transfers. *Computers & Operations Research*, 41:12 – 23, 2014.

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

Sophie N. Parragh and Verena Schmid. Hybrid column generation and large neighborhood search for the dial-a-ride problem. *Computers & Operations Research*, 40(1):490 – 497, 2013.

Sophie N. Parragh, Karl F. Doerner, and Richard F. Hartl. Variable neighborhood search for the dial-a-ride problem. *Computers & Operations Research*, 37(6):1129 – 1138, 2010.

David Pisinger and Stefan Ropke. A general heuristic for vehicle routing problems. *Computers & Operations Research*, 34(8):2403 – 2435, 2007.

Sharon L. Poczter and Luka M. Jankovic. The google car: Driving toward a better future? *Journal of Business Case Studies (JBCS)*, 10(1):7–14, 2013.

Harilaos N. Psaraftis. An exact algorithm for the single vehicle many-to-many dial-a-ride problem with time windows. *Transportation Science*, 17(3):351–357, 1983.

Yuan Qu and Jonathan Bard. A grasp with adaptive large neighborhood search for pickup and delivery problems with transshipment. *Computers & Operations Research*, 39:2439–2456, 2012.

Yuan Qu and Jonathan F. Bard. A branch-and-price-and-cut algorithm for heterogeneous pickup and delivery problems with configurable vehicle capacity. *Transportation Science*, 49(2):254–270, 2014.

Abdur Rais, Filipe Alvelos, and Maria Carvalho. New mixed integer-programming model for the pickup-and-delivery problem with transshipment. *European Journal of Operational Research*, 235(3):530 – 539, 2014.

Julia Rieck, Carsten Ehrenberg, and Jürgen Zimmermann. Many-to-many location-routing with inter-hub transport and multi-commodity pickup-and-delivery. *European Journal of Operational Research*, 236(3): 863 – 878, 2014.

Ulrike Ritzinger, Jakob Puchinger, and Richard F. Hartl. Dynamic programming based metaheuristics for the dial-a-ride problem. *Annals of Operations Research*.

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.

Stefan Ropke, Jean-François Cordeau, and Gilbert Laporte. Models and branch-and-cut algorithms for pickup and delivery problems with time windows. *Networks*, 49:258–272, 2007.

Kevin Spieser, Kyle Treleaven, Rick Zhang, Emilio Frazzoli, Daniel Morton, and Marco Pavone. *Toward a Systematic Approach to the Design and Evaluation of Automated Mobility-on-Demand Systems: A Case Study in Singapore*. 2013.

Paolo Toth and Daniele Vigo. The granular tabu search and its application to the vehicle-routing problem. *INFORMS Journal on Computing*, 15(4):333–346, 2003.

U.S. National Highway Traffic Safety Administration. National motor vehicle crash causation survey: Report to congress. *U.S. National Highway Traffic Safety Administration Technical Report DOT HS*, 2008.

Toon Zijlstra, Peter Bakker, Lucas Harms, Anne Durand, and Hans Wüst. Busgebruikers door dik en dun. *Kennisinstituut voor Mobiliteitsbeleid*, 2018.