A scalable approach to real-time taxi ridesharing

W.J. Vaandrager

A SCALABLE APPROACH TO REAL-TIME TAXI RIDESHARING

by

W.J. Vaandrager

in partial fulfillment of the requirements for the degree of

Master of Science
in Computer Science

at the Delft University of Technology,
to be defended publicly on 16 October, 2018 at 12:00.

Student number:	4175115	
Supervisor:	Dr. M.T.J. Spaan,	TU Delft
Co-supervisor:	Dr. J. Alonso-Mora,	TU Delft
Thesis committee:	Dr. N. Tintarev,	TU Delft
External supervisor:	Ir. E. Zijp,	Transdev

TU Delft Delft University of Technology

Abstract

In an increasing urbanizing world the need for efficient transportation methods is growing. Ridesharing, sharing a taxi trip with multiple passengers, has been proposed as an effective way to contribute towards solving the traffic problems that arise in city centers. Current solutions to the problem are often more focused on optimality, and a fast algorithm is required to handle large quantities of passengers in real-time applications. We propose an algorithm that combines fast and efficient heuristics with clustering and parallelization techniques that is able to solve large problem instances within several seconds. The small response time of the algorithm is able to provide feedback to the customer rapidly, which makes it very suitable for ridesharing in real-time. A segmentation of the customers is implemented to speed up the search. This method divides the customers into subareas based on pickup location such that each subproblem contains a similar number of customers. This division also bounds the size of each subproblem to ensure a solution can be found in real-time. The primary search strategy uses a combination of effective local search heuristics to improve solutions for these subproblems rapidly. A technique reminiscent of variable neighborhood search is also implemented for diversification that can solve smaller instances to a greater degree. The full approach can handle problems containing over 5,000 requests and can effectively manage on-demand ridesharing in Manhattan where the current demand for yellow taxis has tripled.

Preface

This work marks the end of my years as student at the Delft University of Technology. During my time as student, I had the opportunity to develop many new skills, which helped me to understand the challenges that I faced during this thesis. Writing this master thesis proved to be challenging in terms of effort, persistence and knowledge and has provided me a valuable learning experience that would not have been possible without the help of the following people.

First of all, I would like to thank my supervisor, Matthijs Spaan, for all the help he has given me during this project. His advice helped me tackle the challenges I faced during the process and I am grateful for his assistance on how to approach a thesis project and write this report. I would like to thank Erik Zijp for offering me the opportunity to work on this project at Transdev and providing me with an interesting and challenging thesis project. I also would like to thank Javier Alonso-Mora for his feedback and inspiring me to take on this subject. Furthermore, to remain motivated, I would like to thank Olaf Maas and Frits Kastelein for the numerous refreshing tea breaks. Lastly, I want to thank my family and friends for always supporting me during this year.

W.J. Vaandrager
Delft, July 2018

Contents

1 Introduction

In an increasingly urbanizing world, cities have to handle a rising amount of traffic. Ridesharing is proposed as an alternative to decrease the amount of traffic [1–4]. Ridesharing or carpooling is a concept where 2 or more people share a ride other than public transport, in order to save fuel costs, toll and the stress of driving. Carpooling is traditionally used for commuting or to share long journeys, and is often organized by the people involved. However, a new form of ridesharing has come up due to the recent advances in technology, called dynamic ridesharing (also known as instant, on-demand, and real-time ridesharing). Dynamic ridesharing is a variant of ridesharing where a fast process of ride-matching is required as requests are ordered on short notice and have to be placed en-route of the current trips.

The need for efficient transportation has become more important in the last decades. City centers are not designed for the large number of vehicles they currently have to deal with which leads to many traffic jams. INRIX [5] found that in US cities, an average of 42 hours per person per year was spent in traffic due to congestion. In larger cities such as New York and Los Angeles, the average traffic delay even rose above 90 hours. The corresponding cost of the traffic congestion in the US is estimated at 300 billion dollar per year. Furthermore, one third of the CO_2 gasses is released in traffic and transport, with 80% by vehicles [6, 7]. Slow moving traffic produces even more CO_2 per mile. Reducing the of vehicles in urban areas can reduce the amount of traffic as well as the emission significantly. These factors lead cities to want to offer ridesharing services as an efficient alternative to public transport.

In the last decade, the rise of information and communication technologies have enabled many opportunities for optimizing transportation systems. A large majority of people owns a smartphone and is continuously connected to the internet and able to share their location on the fly. This connectivity allows for fast communication between all parties involved; passengers, taxi drivers and the transportation service. The passengers and drivers can be located in real-time and with the use of social networks can establish trust and accountability between ridesharing passengers and drivers. Traditional taxi services are already losing ground in multiple cities, and new mobility alternatives that have come up have rapidly increased in popularity over the last years. These services provide easy to use and convenient mobility services that provide more freedom to the customer. On-demand ridesharing increases in popularity as it suits larger ridesharing problems with a constant flow of new requests [1].

All of these factors lead to an increasing interest and potential of applying ridesharing in metropolitan areas. Centralizing taxi operations without ridesharing can already save 30% of the number of vehicles required in large cities [8] and introducing ridesharing can contribute even more towards solving the current mobility issues. The focus of this thesis is to create a scalable solution that is able to provide fast, quality solutions for a large range of ridesharing problems as well as showing the benefits of ridesharing as a viable option for urban transportation. Few of the current approaches are able to deal with a vast number of incoming requests in real-time. We present a method that can handle the taxi traffic of large cities and provide substantial results in a matter of seconds.

1.1. Problem Statement

This research wants to find a solution for the dynamic ridesharing problem that can scale and deal with large amounts of customers. The approach is able to deal with large growth in customer base while having little influence on the performance. Not much public research has been done that provide fast solutions for large ridesharing problems. Passengers want to know as soon as possible whether a ride is possible and how long it is going to take. Convenience is a crucial factor when using a mobility service, and therefore fast and clear responses are necessary. Determining rapidly whether to accept or decline a ride and where to optimally place a new passenger on a ride is important to keep satisfied customers and create an efficient allocation. The planning of a request has to be completed in seconds, where five seconds is the desired runtime and fifteen seconds is the absolute upper bound.

In order to keep up with a growing number of requests, a scalable approach is necessary to ensure that an efficient allocation can be performed for all requests at all times. A scalable approach allows the routing problem to grow in size while ensuring a similar level of solution quality. To be ready for the future a distributed algorithm makes it possible to scale the computing power to the number of trip requests and vehicles. A distributed solution allows the system to allocate the ridesharing problem over as much computing power as is necessary to stay within the maximum runtime.

1.2. RESEARCH QUESTIONS

The primary issue of this work is finding a solution to real-time ridesharing that is able to provide fast feedback to many customers in the system that achieves good results in seconds. The research questions this thesis would like to answer are:

WHAT ALGORITHMS CAN EFFECTIVELY BE SCALED TO WORK IN A DYNAMIC ENVIRONMENT?

The main reason why we want to develop a scalable algorithm is to be able to handle a large number of requests in a short period. The algorithm should be able to effectively handle a small number of requests in low-demand periods while being able to keep up with future growth and heavily increased peak moments. We want to look for an effective approach that takes advantage of multiple threads or servers and ensures that problems of all sizes can be handled.

WHAT IS THE INFLUENCE OF APPLYING DECOMPOSITION TO LARGE-SCALE ROUTING PROBLEMS?

We want to ensure that large-scale problems are solvable within a short runtime. The scale of a problem can outgrow the ability for the algorithm to hold a set time limit. Breaking the problem up into multiple smaller subproblems establishes that a similar solution quality can be achieved on all subproblems and produce valid solutions for problems of very large-scale.

IS IT POSSIBLE TO CREATE A SCALABLE ALGORITHM THAT HAS TO COMPLETE WITHIN A LIMITED TIME AND IS ABLE TO CREATE EFFECTIVE ALLOCATIONS FOR ON-DEMAND RIDESHARING PASSENGERS?

Dynamic ridesharing has to be able to handle many requests within a short period. A scalable solution enables distribution among multiple threads or servers to cope with a growing or peak number of requests. Using such an approach, the amount servers can be adjusted to the demand and can solve problems in a similar amount of time when dealing with a growing number of requests. We want to look at a specific requirement that the solution must be produced within seconds, which is a restriction that most literature does not take into account. This research question can be extended by asking two other questions that reflect on how to speed up existing or new approaches.

1.3. CONTRIBUTIONS

In this research, we develop a method for scaling the ridesharing by clustering the locations of the customer's pickup locations. The clustering enables scaling down the problem to manageable sizes and also outperforms unclustered problems in short runtimes of 10 seconds. Even problems containing up to 5000 requests could effectively be handled within this time limit.

Local search is applied to find solutions and provides an improvement of up to 20% on insertion techniques and is effective in improving solutions in the short time that is provided. Variable neighborhood search techniques are used on top of the local search approach to prevent getting stuck in local optima. In dynamic environments, the combination of clustering, local search, and diversification is able to find high-quality solutions within a few percent of the best-known solutions. We found similar results to Alonso Mora et al.[4] where ridesharing can reduce the current number of taxis by 80% in large cities when combining up to four passengers at a time.

The proposed approach can solve small and large problems within seconds and provide fast feedback to passengers. While many challenges have yet to be overcome to deploy ridesharing effectively at large scale, it shows great potential even in a fast-paced environment.

1.4. Outline

Chapter 2 will provide an overview of the ridesharing problem and which specific requirements our problem contains. The following chapter 3 shows various techniques of previous literature that have influenced the methods we have developed. In chapter 4 these different methods and algorithms that are used to construct a solution for our problem. The setup for the experiments we conducted to test the validity of our approach is discussed in chapter 5 and the results of these experiments are shown in the consecutive chapter 6. Chapter 7 offers an overview of related research and gives an impression of other solutions for ridesharing and large optimization problems in this field. Finally, in chapter 8, we discuss the results, answer the research questions of this chapter and look at how our approach could be improved in future research.

2

Problem Definition

In this chapter, an overview of the current problem is provided as well as the goal of this thesis. After that, the requirements and research questions are discussed to specify the focus of this thesis.

2.1. Ridesharing Problem

Ridesharing is the concept of sharing a ride with multiple passengers that have a similar trip. Variants of this include carpooling, vanpooling, carsharing and real-time ridesharing. Ridesharing services are growing in popularity, but still only form a small portion of the total mobility market.

The main problem of serving customers as efficiently as possible is based on the vehicle routing problem (VRP). This problem was first described in Dantzig and Ramser [9] in 1959 and defines a problem where all requests have to be served by a fleet of vehicles. The vehicles are departing from a single depot where they have to start and end. The goal is to minimize the total distance traveled by the vehicles while serving all requests. In the past half century, this problem is extended on and many different variations have been created to serve a multitude of optimization problems. These include time windows (VRPTW), multiple depots (MDVRP), capacitated and heterogeneous fleet (CVRP, HFVRP) and many more [10].

The problem of this thesis can be described as a Pickup and Delivery Problem with Time Windows (PDPTW). Each request has one or more passengers that want to travel between an origin and a destination within a certain time window. The vehicle fleet is represented by all the vehicles that are driving around in the area. The goal is to serve the customers within their time-windows in the most cost-effective way that is based on a cost function. It is a variant of the VRP and is also an NP-Hard problem [11].

2.2. Requirements

The goal of the system is to be able to plan shared taxi rides for a large number of passengers in a brief period. This concept requires a fast response time from the algorithm supported by a scalable approach that makes sure that the workload is always solvable within a short time window. Based on these conditions the system will have the following requirements:

- The algorithm needs to solve the planning problem and be scalable to the number of requests and vehicles.
- The algorithm is able to fit new requests in the current planning.
- Once a request is planned it cannot be canceled.
- A request has to be placed or declined preferably within 5 seconds and at most 15 seconds.

2.3. PROBLEM MODEL

The problem this thesis wants to solve is a dynamic ridesharing problem that wants to serve as many requests in the most efficient way as possible. As discussed in the Section 2.1, this problem is closest related to the PDPTW. This variant is an optimization problem that can be described as a mathematical model. The problem has a set $\mathcal{R}$ containing n requests and a set $\mathcal{V}$ which consists of v vehicles. Every $r \in \mathcal{R}$ contains two locations, an origin and a destination. These both have an earliest, latest and desired arrival time and contain a location. The values defined for an origin or destination are as follows; for each $r \in \mathcal{R}$: e_r as earliest time, l_r as latest time and t_r as the expected arrival time. It holds that $e_r \leq t_r \leq l_r$. Each request location has a number of passengers defined q_r as the capacity. It holds that the capacity of a destination is negative the value of the pickup's capacity: $q_r = -q_{n+r}$.

Each vehicle $m \in \mathcal{V}$ has a location l_m and a maximum capacity Q_m. For every two locations $(i, j) \in \mathcal{L}$ there is value $c_{i,j}$ which gives the cost of this route based on distance or travel time.

An overview of the general restrictions and variables of the problem are described below.

- **Precedence:** The origin of a request must be served before the destination.
- **Capacity:** Each vehicle has a maximum capacity of passengers.
- **Time Window:** Each request has a time window that must be satisfied.
- **No transfer:** A request can only be served by one vehicle.
- **Demand:** All passengers of the same request must travel together

$\mathcal{R}$	Requests
s_i	Service time of location i
q_i	Capacity of location i
e_i	Earliest arrival time of location i
l_i	Latest arrival time of location i
t_i	Estimated arrival time at location i
M_i	Maximum ride time of a request i
T_i	Planned ride time of a request i
$\mathcal{V}$	Vehicles
l_v	Start location of vehicle v
Q^v	Maximum capacity of vehicle v
q_i^v	Load of vehicle v at location i
$\mathcal{P}$	Pickup locations $\{1,..,n\}$
$\mathcal{D}$	Delivery locations $\{n+1, n+2,..,2n\}$
$\mathcal{V}_l$	Vehicle locations $\{2n+1,..,2n+v\}$
$\mathcal{L}$	All locations $\mathcal{L} = \mathcal{P} \cup \mathcal{D} \cup \mathcal{V}_l$
$x_{i,j}^v$	$\begin{cases} 1 & \text{if vehicle } v \text{ travels from location } i \text{ to } j \\ 0 & \text{otherwise} \end{cases}$
$t_{i,j}$	Travel time between locations i and j
$c_{i,j}$	Cost route from location i to j

Based on these restrictions it is possible to define the following mathematical model for the problem:

$$\min \sum_{i\in\mathcal{L}} \sum_{j\in\mathcal{L}} \sum_{v\in\mathcal{V}} x_{i,j}^v c_{i,j} \tag{1}$$

Subject to:

$$\begin{aligned}
&\sum_{i\in L} \sum_{v\in\mathcal{V}} x_{i,j}^v \leq 1 && \forall j \in \mathcal{P} && (2)\\
&\sum_{j\in L} x_{i,j}^v - \sum_{j\in\mathcal{L}} x_{n+i,j}^v = 0 && \forall i \in \mathcal{P}, v \in \mathcal{V} && (3)\\
&t_i \leq t_{n+i} && \forall i \in \mathcal{P} && (4)\\
&t_j \geq x_{i,j}^v \cdot (t_i + s_i + t_{i,j}) && \forall i, j \in \mathcal{L}, v \in \mathcal{V} && (5)\\
&q_j^v \geq (q_i^v + q_j) x_{i,j}^v && \forall i, j \in \mathcal{L}, v \in \mathcal{V} && (6)\\
&e_i \leq t_i \leq l_i && \forall i \in \mathcal{P} \cup \mathcal{D} && (7)\\
&q_i \leq q_i^v \leq Q_v && \forall i \in \mathcal{P}, v \in \mathcal{V} && (8)\\
&q_{l_v}^v = 0 && \forall v \in \mathcal{V} &&\\
&t_{l_v} = 0 && \forall v \in \mathcal{V} &&\\
&x_{i,j}^v \in \{0,1\} && \forall i, j \in \mathcal{L}, v \in \mathcal{V} &&
\end{aligned}$$

The goal is to minimize the sum of the costs taken for serving the requests in the problem (1). Constraint (2) ensures that a request is at most picked up by one vehicle. Constraints (3,4) state that a request is picked up and dropped off by the same vehicle and can not be dropped off before it is picked up. Constraint (5) keeps track of the estimated time at a location. Constraint (6) does the same for the number of passengers in the vehicle at a location. Constraints (7,8) check if the capacity and time window constraints are not violated.

$$\begin{aligned}
&t_{i,n+i} \leq T_i \leq M_i && \forall i \in \mathcal{P}, v \in \mathcal{V} && (9)\\
&t_j \leq x_{i,j}^v \cdot \min(q_i^v, 1) \cdot (t_i + s_i + t_{i,j}) && \forall i, j \in \mathcal{L}, v \in \mathcal{V} && (10)
\end{aligned}$$

An extension of this model also provides some quality of service constraints for the passengers. This extension is called the Dial-a-ride problem (DARP) in the literature. The DARP has more focus on customer service and ensures that the travel time of passengers is kept to a maximum. Constraint (9) makes sure that a passenger does not exceed its maximum ride time and constraint (10) prevents that a vehicle that is carrying at least a passenger can wait at a location. These additional constraints provide a smoother ride for the passengers as they will experience fewer moments where they feel they are unnecessary being delayed during their trip.

2.3.1. Objective Function

In the current problem four parameters are considered in calculating the cost $c_{o,d}^v$ of a request r in a trip with v as the vehicle of the trip and the pair o, d as the origin and destination of the request:

- Travel time / distance (α_1)
- Delay (α_2)
- Number of vehicles (α_3)
- Rejected requests (α_4)

Travel time or distance is the primary objective to minimize in our problem. A low total means that many rides can be shared as combining trips decreases the total travel time.

The goal of the delay factor is to minimize the detour time a passenger experiences before arriving at his destination. Focusing on this factor makes the trips shorter for passengers, but this leads to fewer detours in the trips that can decrease the overall amount of ridesharing. However, the delay factor has an advantage in the dynamic problem. As requests are served at a higher rate, the fleet has more capacity for serving incoming requests. Results show that this factor can reduce the overall travel expenses (Section 6.5.2).

In the majority of research that looks at static instances, the number of vehicles is the primary objective to minimize as drivers and vehicles can often be a costly expense [12, 13]. In this research, minimizing the number of vehicles is of less importance in a dynamic problem as the fleet size in our problem is constant. However, as we also look at static instances, a pentalty cost for a vehicle is defined, α_3, so we can adjust the importance of this objective accordingly. This value is only used when minimizing the number of vehicles is an objective.

Most variants of the VRP have a requirement to serve all requests in a problem and pose this is as a constraint in their model. Our model does not impose this as a constraint as customers book rides in real-time and their placement cannot be guaranteed. The goal is to reject as few passengers as possible though, and so the penalty for rejecting a request, α_4, is set to a high value compared to the other parameters.

In our model, the route of a vehicle is represented by $\mathcal{S}_v$ and the route without the request is defined as $\mathcal{S}_v^- = \mathcal{S}_v \setminus \{o, d\}$. The following equation describes the cost for a vehicle v to service a request with an origin o and a destination d:

$$c^v_{o,d} = \alpha_1 \cdot (\sum_{i=2}^{|\mathcal{S}_v|} t_{i-1,i} - \sum_{i=2}^{|\mathcal{S}_v^-|} t_{i-1,i}) + \alpha_2 \cdot (t_d - e_o - t_{o,d}) + \alpha_3 \cdot (1 - \min(q^v_{o-1}, 1)) + \alpha_4 \cdot (1 - \sum_{a \in \mathcal{V}} x^a_{o-1,o})$$

The route cost is determined by how much travel time or distance the request adds to the route times α_1. The delay is calculated by taking the total delay during the trip, or the earliest possible arrival time compared to estimated arrival time at the destination. When the vehicle is empty before picking up the request the penalty of α_3 is applied and penalty α_4 is relevant when the request is not served by any vehicle. The weights of these parameters are discussed in section 6.5.2.

2.4. Dynamic Ridesharing

Dynamic ridesharing is the main subject, and therefore we want to propose a solution that is applicable to on-demand taxi services. In contrast to the static variant, dynamic ridesharing includes elements of uncertainty as new requests are constantly introduced to the problem. This uncertainty creates some extra obstacles as the optimal solution might not lead to the most effective state later on.

The dynamic environment brings along a few extra challenges [2, 14]. The first one is that the problem has to be solved in real-time. As vehicles are constantly moving and new requests popping up, producing fast results is necessary to keep up with the current situation. The problem can change fast and vary on different days, and for that reason, a scalable solution is essential as this enables the problem to be solved in applicable real-time bounds at any size.

Another challenge that the dynamic variant introduces is the ability to change a current route to serve new requests also called vehicle diversion [2]. Calculating route costs and travel times is expensive, and thus estimates are often used. Planning issues can occur when vehicles are rerouted from their current trajectory, and previous estimates do no longer hold. Times have to be recalculated again as the vehicle's position has changed since the previous assignment.

The anticipation of future requests is essential to be able to serve as many new requests as possible. When only taking into account the current situation vehicles might not be located at the right locations to pick up incoming requests and lose. Taking into account future requests could save from 11-69% in travel cost according to van Hentenryck and Bent [15].

In the same line is the influence of rebalancing vehicles and this can give significant improvement in serving customers efficiently. Alonso mora et al. [4] found that rebalancing vehicles can lead to an increase of 30% in the number of passengers that a fleet of 2000 vehicles can serve in Manhattan.

While this thesis does not tackle all of these challenges, they have been considered during the process as they can have a significant influence on the results. In the following section, previous approaches for solving for static and dynamic ridesharing problems are presented.

2.5. Scalability

Scalability is an important focus of this thesis as our approach should be able to handle a vast number of requests in real-time and handle peaks in customer demand. In practice, ridesharing problems will be contained to a certain area because trying to rideshare a whole country at once is not practical. The scalable aspect of the system is focused on the size, number of requests, of the problem rather than on the request density of a region. The ridesharing problem is not dependent on the density, as the algorithm should be able to combine trips where ever possible. Therefore, this thesis will not consider request density or the size of the area directly in the problem. However, the system should be able to handle fluctuations in customer demand during the day in the same area effectively and also be able to handle a rising number of customers to accommodate future growth. While ridesharing could also be effective in rural areas, the focus of this research is on providing an extensive ridesharing service for large cities with many requests and shorter trips.

3 BACKGROUND

In this chapter, the background of the ridesharing problem and its variants are described. It also introduces techniques for ways of dealing with the scalability of this problem. Combined with the background an overview of previous research in this field is described.

3.1. SOLUTION METHODS

Many approaches have been created for solving the VRP and its variants. Finding a good trade-off between solution-quality and runtime is an important part of finding a suitable approach to the problem at hand. The VRP problem is NP-Hard and thus finding an exact solution for a problem is computationally expensive. The latest exact solvers can handle up to 100 locations within a runtime of several hours[16]. As we focus on finding solutions in real-time, exact algorithms are not considered in this thesis.

Since this problem is generally hard to solve, many research has been done in finding approximate solutions to the problem that exchange the ability to guarantee to find the optimal solution for a decreased runtime. The runtime of heuristics methods is often significantly smaller than exact algorithms, and are thus are often used in practice to find solutions in a "reasonable" time limit. These methods use heuristics that apply practical rules to find solutions more efficiently. This section discusses the commonly used heuristics in solving the general VRP as well as closest variant to our problem, the Dial-a-ride problem (DARP) or Pickup and Delivery Problem with Time Windows (PDPTW).

3.1.1. CONSTRUCTION METHODS

Constructive methods try to build feasible solutions from scratch usually to produce an initial solution that can be improved by other methods. Construction often looks at one request or route at a time and decisions made during the process cannot be reversed. The *Savings* method from Clarke and Wright [17] is one of the first methods of constructing fast solutions for the VRP. The method starts by mapping every request to a vehicle from a central depot. The next step is to merge routes that provide the most savings. For two requests i, j and a central depot d the savings are defined by $s_{i,j} = c_{d,i} + c_{j,d} - c_{i,j}$. Every iteration the merge that has the largest savings and is feasible is applied.

Another construction method is called *Sweep* proposed by Gillet et al. [18]. This method tries to create routes by looking at requests in a range from the main depot that increases in size during the process. Every sweep the ring around the depot is increased and new requests are inserted at the end of an existing trip. When a request cannot fit in any of the current routes, a new route is created.

Jaw et al.[19] created a construction heuristic for the PDPTW based on scheduling blocks. Each vehicle has its own route with a schedule, and each stop has its minimum, estimated and maximum arrival time. Every time a request is added the other locations can be shifted between these times in order to create a new valid schedule. The requests are sorted on earliest pickup time and sequentially inserted into the position in the route which adds the lowest additional cost.

3.1.2. LOCAL SEARCH

Local search is a heuristic method that is commonly used for hard optimization problems. Local search looks for solutions in a small neighborhood $\mathcal{N}$ in a discrete search space $\mathcal{S}$. The search moves around requests in the problem and looks for moves in the neighborhood $s' \in N$ with a lower cost $f(s') < f(s)$. The acceptance of a move is can also be probabilistic in which case the algorithm can accept a solution with a lower cost in order to diversify the search space. Commonly used operators are; relocation of a single request, exchanging the position of two requests and, k-opt, which optimizes the position of k locations [20].

Local search excels in intensification, improving the current solution, due to its ability to find the best position for individual requests. However, it is likely to get stuck in local optima as it lacks ways of diversification as it only affects a small number of nodes at a time. An extension to counter the diversification problem is iterated local search (ILS). ILS uses different starting positions or perturbations in order to end up in different local optima and improve the quality of the overall solution. In the following section, more exhaustive search methods are discussed some of which also apply local search.

3.1.3. METAHEURISTICS

Metaheuristics are general problem-solving heuristics for a large group of optimization problems. The general idea of these heuristics is to efficiently find good solutions in the search space that are too hard to solve to optimality. Metaheuristics differ from local search solutions by providing options to escape local minima and increase the chance of finding a better solution. Metaheuristics turned out to be less effective in improving solution quality in real-time applications because the exhaustion of a larger search space increases the runtime.

TABU SEARCH

Fred Glover introduced tabu search in 1986 [21]. The algorithm uses local search to improve on existing solutions. The algorithm stores the solutions it finds in a *tabu list* to stray from previously found solutions. Every iteration it tries to find better solutions by performing small changes on the current solution. When no improving moves are found it also accepts worse solutions to able to escape local optima.

GENETIC ALGORITHMS

Genetic algorithms [22] work with improving solutions over generations. It is based on Darwinism where better solutions thrive and are selected for future generations. The process starts from an initial solution that is randomly generated or heuristically constructed. The methods of creating new generations are derived from the natural evolution process and are mutation, cross-over, and selection of solutions. The fitness of each solution is determined and only the best solutions are allowed to generate offspring. After a number of generations the algorithm returns the best solution.

SIMULATED ANNEALING

Simulated annealing is a probabilistic search heuristic for exploring a large search space. It uses a cooling schedule to reduce the search space slowly and convert to a solution. Every solution has a probability to be selected based on its value. As the temperature cools down, the neighborhood becomes smaller and better solutions become more likely to be selected. The cooling schedule allows the heuristic to explore a more extensive search space while converging to an optimum.

LARGE NEIGHBORHOOD SEARCH

Large neighborhood search (LNS) is a metaheuristic developed by Shaw [23] that uses powerful moves to explore large neighborhoods. As a rule of thumb it follows that the larger the explored neighborhoods, the more likely the algorithm is to escape local minima and find better solutions, however, these also require more time to solve the problem. LNS tries to find better solutions by removing and reinserting subsets of customers from the problem. Shaw's method is based on removing related requests from the solution as they are more likely to be interchanged and optimized when reinserted into the route plan. The idea of 'destroy and repair' is used in other LNS methods that use various heuristics to choose what requests are removed from the solution and how they are reinserted.

Ropke et al. [24] developed an adaptive version of LNS by testing which removal and reinsertion techniques provide the largest impact in optimizing routing problems. They give weights to these heuristics such that more effective heuristics are chosen more frequently during the search.

VARIABLE NEIGHBORHOOD SEARCH

Variable neighborhood search (VNS) was developed in 1997 by Mladenović and Hansen[25]. VNS is a heuristic-based method that improves routing problems using local search but also has the option to escape local neighborhoods in order to explore large parts of solution space. VNS improves a solution using local search until a local optimum has been found and then tries to escape this neighborhood in a perturbation phase. A perturbation can exist of a random restart of the problem to start from a different base solution or apply a larger neighborhood move on the current or previously found solution that serves as a new starting point. VNS uses local search to explores a neighborhood quickly while the perturbation allows for multiple descents into various neighborhoods to find general high-quality solutions.

The use of VNS to solve very large-scale VRP problems in [26] has shown its effectiveness in solving problems of up to 20,000 locations. It is one of the few approaches that looks at routing problems of this size. Other methods have also shown the effectiveness of VNS [27, 28]. The strengths of VNS are its simplicity and effectiveness of tackling optimization problems.

3.2. Dynamic Ridesharing

In the last decade, ridesharing and especially dynamic ridesharing as a good alternative for public transport has increased in popularity. Other literature has already proposed other solutions for introducing ridesharing into the current transportation systems. Ma et al. and d'Orey et al. [29, 30] have developed some of the first solutions for on-demand taxi-ridesharing systems, and showed the promising advantages that it could bring as alternative service. However, their focus was largely on creating a full system for this problem, and their algorithms produce less substantial results than current approaches.

Alonso Mora et al. [4] presents a real-time method for ridesharing in Manhattan. They provide a solution to the ridesharing problem by mapping it to an integer linear program with reduced dimensionality to be able to handle large instances within real-time. By creating a graph of the of the viable request and vehicle matches they can find an optimal assignment between all pairs given enough runtime.

The paper shows that with 3000 vehicles, a quarter of the current fleet, with a capacity of 4 passengers can already serve 98% of the requests. Since this is most extensive research for ridesharing in Manhattan, many experiments in this thesis are similar since they have been deemed effective and provide a baseline for comparison. However, their approach requires 30 seconds to solve a batch of requests, and while their approach is fully scalable, has a greater complexity that makes it harder to scale to very large routing problems.

3.3. Scalability & Parallelization

Scalability is the ability to adapt to scale or size to keep up with the changing size of a problem. Scalability can be essential as it allows growth while minimizing the effect on performance. In this section two main concepts of scalability will be discussed; parallelization techniques in heuristic problem-solving and introducing a split of the problem into multiple subproblems or clusters.

Parallelization is the ability to do multiple calculations or processes simultaneously. Parallel computing can decrease the required runtime of a problem as the workload can be scaled over multiple processes.

According to Crainic [31], parallelization addresses two large issues that come along with larger problem instances: solving larger instances than is possible with sequential methods, and having more robustness against a broad range of problem instances. The two main reasons to use parallel methods are either to accelerate the search, find similar quality solutions in a shorter period or to broaden the search, find better solutions in the same amount of time. He states that there four main types of parallelism that can be distinguished in solving heuristic problems:

Low Level Parallelization

Low-level parallelization is the most basic form of parallelism. The problem is split up in computational intensive parts that can be solved separately. This split up does not interact with the algorithm's structure or search space and is solely for dividing the workload over all the available computing power. Not all algorithms can be used with this method efficiently as the subproblems have to be solved individually and cannot depend on each other.

Data Decomposition

Data decomposition uses separation of the search space to solve problems faster. The search space is divided into multiple subproblems that can be solved in parallel. These solutions are combined afterward to create a feasible solution for the entire problem. Data decomposition reduces the size of each problem to make them easier to solve, but also reduces the quality as the subproblems, as the problems do not contain all available information. The following section 3.4 discusses this technique in more detail as it is implemented in our approach.

Independent Multi-Search

Multiple processes try to solve the whole problem but from different angles and starting points. This technique can be effective because approaching the solution in multiple ways increases the probability of finding better solutions.

Cooperative Search

Cooperative search is similar to independent multi-search, but the different processes also share information and solutions to steer the search of others. The information can provide options for other processes to change its current parameters or to utilize parts of other solutions. However, choosing what information is exchanged or used can be difficult as this can steer different processes towards the same solutions and lead to duplicate computing. When communication between process is well structured, the cooperation can offer significant benefits compared to independent search.

3.4. Clustering

Clustering is an approach to divide a set of objects into different groups where each object in a cluster is more closely related to each other compared to any other cluster. Clustering is mainly used in data analysis to create and find patterns, but could also be used in algorithms for dividing a problem into multiple subproblems. Clustering can be classified as a type of data decomposition (Section 3.3) as it creates subproblems. These subproblems can then be solved independently by multiple processes. This section introduces different clustering techniques that can be used to create a partition of requests in routing problems.

3.4.1. K-Means Clustering

K-means is a clustering method of dividing up a set of n data-points into k clusters. Each observation belongs to the cluster which center, the mean value of all points in the cluster, is closest. The goal is to find a division where the sum of distances between observation and centers is minimal. K-Means is an NP-hard problem. However, in practice heuristics are often used that quickly converge to a local optimum.

K-means is often in and classifying large datasets as it is fast and easy to implement. The classifications of k-means are not always correct as adjacent cluster structures and the predefined number of clusters can inhibit the ability to reconstruct a classification.

K-means is not commonly used in ridesharing problems, but some researchers have looked at it for tackling large-scale vehicle routing problems[32–34]. He et al. [32] created a method for forming several clusters with the same number of requests in large cities. They show that clusters are balanced and that their method could be used for the decomposition of large-scale VRP problems.

3.4.2. Density-Based Clustering

Density-based clustering tries to find clusters of closely packed points in Euclidean space. It uses two parameters, a distance parameter for the maximum distance within a cluster and a minimum cluster size. With these two parameters, it tries to combine points that have at least the minimum number of neighbors within the maximum range. In closely packed areas it combines all neighboring points to form a large cluster. It is a quick clustering algorithm which runs in $\mathcal{O}(\mathrm{n}\log\mathrm{n})$ time.

An advantage of this technique compared to other clustering techniques is that it does not have a set amount of clusters and can extract clusters based on a strong relatedness in the data. It is also not very sensitive to outliers and noise. The disadvantage of this method is that when it cannot find clear cluster patterns in the data, it might leave many points unclustered and produce widely varying sizes of subproblems. Choosing the parameters for distance and cluster size can also be difficult to generalize as this is problem-dependent.

3.4.3. Hierarchical Clustering

Hierarchical Clustering is a clustering technique that clusters data in a hierarchal tree structure. There are two general strategies. The first strategy is agglomerative, also known as bottom-up, in which points all start as singular clusters and are merged based on closeness until all points are connected. The other one, divisive, is the reverse strategy where all points start in a single cluster that turns into a hierarchy as the algorithm splits the cluster into separate points.

The algorithm iterates over the points and each time merges the two clusters that have the least distance between them. By going down the hierarchy, the desired amount of clusters can be extracted. A disadvantage of this technique that it is slow and classifies individual points each time which can struggle with dividing adjacent cluster structures.

4 Methodology

In this chapter, the approach and implementation of the techniques created in this thesis are discussed. The first section discusses the approach and the idea behind the proposed methods. Section 4.2 provides details on the implementation of the developed clustering algorithm. The following sections 4.3 - 4.5 show the implementation of the main algorithm used for solving the ridesharing problem.

4.1. Approach

This section shows an overview of our approach and discusses why certain techniques are applied. The proposed methods are mostly of a greedy nature as the problem is NP-Hard and so finding an optimal allocation in real-time is not feasible.

The approach is split into two main components: finding an algorithm for creating a solution to the Pickup and Delivery Problem with Time Windows (PDPTW) in a short runtime and, to increase the scalability of the approach, a technique that is able to split the size of problem for speeding up the search.

Scalability is the main focus of this thesis, and hence we want to implement a parallelizable approach to reduce the runtime. In the section 3.3, four Parallelization techniques are discussed: low-level, data decompositon, independent search and cooperative search. The last two techniques, independent and cooperative search, are dismissed as they are more focused on extending the search instead of speeding up the process. Since a requirement of 5 seconds is set in this research, a decomposition strategy is chosen which downscales the problem and reduce the required computational time substantially.

The search approach is based on variable neighborhood search (VNS). Two search strategies are commonly used in heuristics and these are intensification, focusing on improving the current solution, and diversification, the search for other promising neighborhoods of the current solution. VNS combines the intensification procedure of local search, that can rapidly improve constructed solutions, with the ability to escape local optima whenever possible. As the available runtime is short, focusing on diversification is not advantageous. VNS puts more focus on intensification as primary search strategy which is beneficial as local optima are often the best possible solution that can be found in a small time window. The method is proposed as a suitable method for real-time instances as it is able to find solutions fast while providing the possibility for escaping local minima.

4.1.1. System overview

Figure 4.1 gives an overview of our approach during a planning iteration. The first step is clustering (Section 4.2) where the problem is split up based on its size. This step places all requests and vehicles in the corresponding clusters and so each subproblem can be solved individually. The second step for each subproblem is preprocessing (Section 5.3.1). This step removes all infeasible trips from the problem and for each request sorts all feasible vehicles on distance. The next step is to construct an initial solution by greedily placing all requests in the current solution (Section 4.3). Then the main algorithm tries to improve on this solution in the remaining time using LS and VNS (Sections 4.4 and 4.5). It keeps track of the currently best solution based on the objective function (Section 2.3.1) and when the time limit is reached will return this solution. When the algorithm is used for a dynamic problem, the passengers that have arrived at their destination are

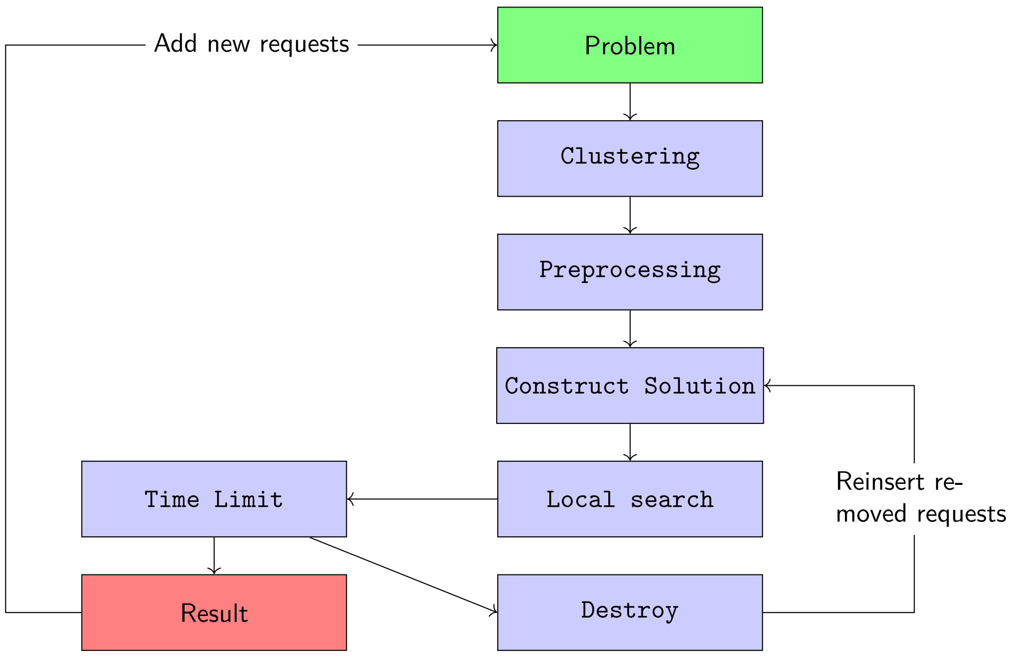


Figure 4.1: Main Approach

removed from the problem, and all newly booked requests are added to the problem.

The following section 4.2 discusses the first step in reducing the size of the problem and creating a fully scalable approach. The last three steps of the approach form the main algorithm to solve the PDPTW. This algorithm is discussed in sections (4.3 - 4.5).

4.2. CLUSTERING

Because our problem is complexionally hard to solve and our computational time is limited, a solution that bounds the number of requests in a scalable approach is necessary. It is common to divide large cities and regions into multiple zones that each have their own jurisdiction. This 'zoning' can remove potential ridesharing matches but in return makes each problem smaller and faster to solve. However, this approach is not perfect given the fact that a) the customer demand changes during the day and b) when a large event taking place results in the customer demand in a zone becoming too large to be handled effectively by the system. In order to address changes in customer demand and peak moments, this thesis creates a clustering approach that introduces dynamic zones based on the current customer demand.

Clustering allows the system to decrease the number of zones when demand is low while enabling the system to increase the number of zones for peak moments or unexpected events with an largely increased level of demand. The zones, also called subproblems, can have a maximum number of requests to scale the subproblems accordingly to the capability of the system. The focus on scalability is also applied in the size of the subproblems, where each subproblem contains a similar number of requests to prevent large subproblems from spiking the runtime. The clustering method divides the region of the problem into smaller areas that can be solved individually in parallel. The number of requests per subproblem is bounded and a larger problem will thus be divided into a larger number of subproblems. As the subproblems are solved in parallel, they can be distributed over the available computing power which can be scaled to accommodate the size of the problem.

In the general ridesharing problem, many variables can be considered for dividing the problem into multiple subproblems. Variables that could be evaluated are the locations from the request or vehicles, the time windows of the requests, the route of a passenger, etc. Area clustering is used because it is the most distinguishing factor between requests in dynamic ridesharing. As requests have to be served within short time windows, only vehicles and requests near the request can be considered for ridesharing. Due to the short pickup window for a passenger, the pickup locations of the requests are used as the input for creation of the clusters. In a fast-paced taxi service, the radius for potential ridesharing matches is small which makes the pickup location the most decisive factor for differentiating the requests.

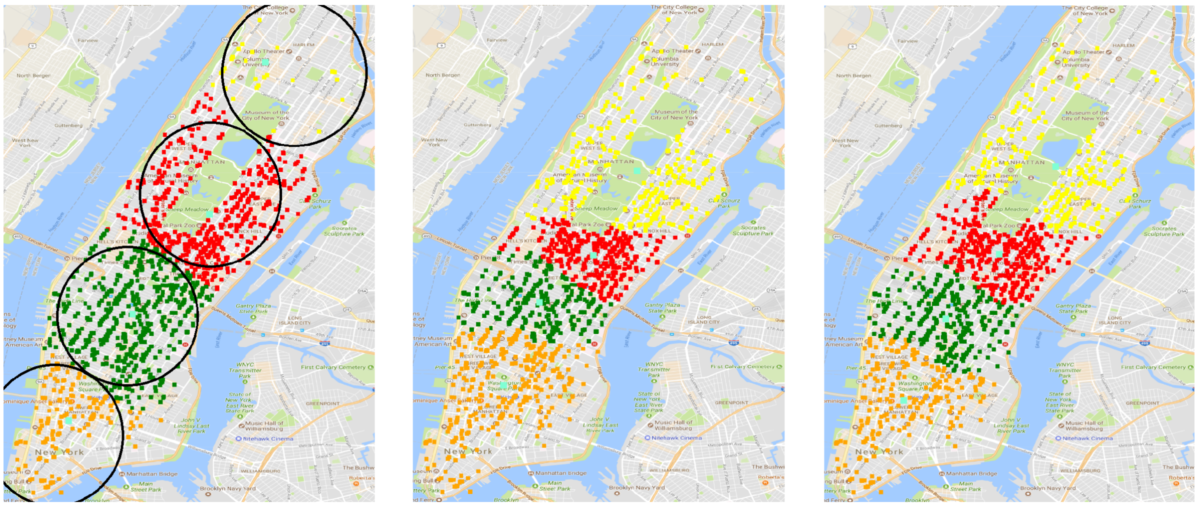


Figure 4.2: 3 phases of the proposed clustering:
1. Create clusters of similar surface area.
2. Balance the clusters to have the same number of requests.
3. Balance area and number of requests to remove elongated clusters

4.2.1. K-MEANS

K-means is a simple and effective method to divide a set of observations into multiple subproblems. The goal is to decompose the problem quickly without heavily influencing the potential ridesharing. Our method is based on He et al. [32], who created a k-means clustering technique that splits up an area into multiple subproblems with the same number of requests. They used a two-step method in which the first step is to initialize *k* clusters spread out over the area. The second step is to equally distribute the number of request per cluster. For every two clusters that share a border, a number of requests closest to the center of the smallest cluster are moved from the largest to the smaller cluster.

An issue He's algorithm has, is its preference of balancing number of requests over the area shape of the clusters. This bias can produce clusters that are elongated and contain outliers. Outliers are a problem for ridesharing as they are isolated from other requests and are thus expensive to combine. Therefore, we use the usual k-means approach in our implementation and classify all requests to their nearest cluster center. This way all requests are always best classified to their current clusters, and maintains larger cluster structures as it harder to split separate groups. The disadvantage is that the created clusters can have a larger disbalance in the number of requests per cluster as dense regions tend to have more requests which can lead to an increase in runtime.

4.2.2. IMPLEMENTATION

The first step is to create *k* initial clusters that each represent a location in the area of the problem. The starting clusters are chosen to create a large spread over the area. Every round the request whose pickup location is furthest from all current centers is chosen as new starting centroid. This step is repeated until there are *k* starting centers and the centroids are spread out over the area.

When the initial clusters are created, each request is mapped to the nearest cluster center. The center of each cluster is then recalculated as the average location of the locations it contains. These initialization steps are repeated several times to match the center with the requests in a cluster. This step is phase 1 of figure 4.2 and results in clusters of about the same area but can be varying in the number of requests.

The following step is to rebalance the number of requests per cluster. The requests in each cluster are equalized by shifting requests to neighboring clusters with fewer requests. This approach is an extension of He, in which requests are only moved between two clusters at a time. The reason to consider all neighbors at once is to speed up the balancing phase in instances with a large number of requests. The closest cluster center for each cluster is considered its direct neighbor and serves as a guideline for other neighbors. Every other cluster that is within two times this range is considered a neighbor. This range is chosen to include immediate neighbors only.

The BalanceClusters method in algorithm 1 shows the procedure of how the clusters are balanced. After every iteration, the centers are recalculated using the average location of all points in the clusters and customers are moved to their nearest center. The rebalancing step is repeated until the clusters reach a steady state, or a maximum amount of iterations is performed.

Algorithm 1 Clustering

```
1: procedure RECALCULATECENTERS
2:     for $r \in \mathcal{R}$ do
3:         $c =$ closest cluster to pickup of request $r$
4:         $c_{\mathcal{R}} = c_{\mathcal{R}} \cup r$
5:     $\forall c \in \mathcal{C} \Rightarrow c_{center} = MeanLocation(c_{\mathcal{R}})$

1: procedure BALANCECLUSTERS
2:     Find neighbors for each cluster to exchange requests.
3:     while $MaxCluster - MinCluster > threshold$ do
4:         Sort $\mathcal{C}$ on the number of requests, smallest first
5:         for $c \in \mathcal{C}$ do
6:             for $n \in c.neighbors$ do
7:                 $\lambda \leftarrow \frac{diffRequestSize(n,c)}{2}$
8:                 Move closest $\lambda$ requests to $c$ from $n$
9:         RECALCULATECENTERS()

1: procedure MAIN
2:     Input: List of requests $\mathcal{R}$, List of vehicles $\mathcal{V}$,
3:            Number of clusters $k$
4:     Output: List of clusters $\mathcal{C}$
5:     $c^0_{center} = p_0 \in \mathcal{P} \subset \mathcal{R}$
6:     $c^1_{center} = maxDistance(c^0_{center}, \forall p \in \mathcal{P})$
7:     $\mathcal{C} = \mathcal{C} \cup c^1_{center}$
8:     for $i = 2, ..., k$ do
9:         $c^i_{center} = maxDistance(\forall c \in \mathcal{C}, \forall p \in \mathcal{P})$
10:        $\mathcal{C} = \mathcal{C} \cup c^i$
11:    for $x$ times do
12:        RECALCULATECENTERS()
13:    Sort $\mathcal{C}$ on increasing number of requests
14:    for $y$ times do
15:        $c^{min} = c^0; c^{max} = c^k$
16:        BALANCECLUSTERS()
17:        RECALCULATECENTERS()
18:        Sort $\mathcal{C}$ on increasing number of requests
19:        if $c^k_{\#\mathcal{R}} - c^0_{\#\mathcal{R}} < threshold$ or $(c^{min} \equiv c^0_{\#\mathcal{R}}$ and $c^{max}_{\#\mathcal{R}} \equiv c^k_{\#\mathcal{R}})$ then
20:            break
    return $\mathcal{C}$
```

Phase 2 of figure 4.2 shows the results after the balancing step. The 4 clusters each contain the same number of requests, but the shape and size of the clusters have become more variable.

The last action is to place the vehicles in the clusters. Similarly to the requests, the vehicles are placed in the cluster with the nearest center. The last picture in figure 4.2 shows Manhattan clustered into 4 regions that are of similar size and number of requests. The results for a larger number of clusters can be seen in New York can be seen in figure 4.3.

4.2.3. ADAPTATIONS FOR REAL-TIME EXECUTION

The clustering allocation has to be performed in a dynamic environment. Therefore, the cluster step has to be repeated every time the ridesharing problem is updated, and requires a fast response to have as little impact

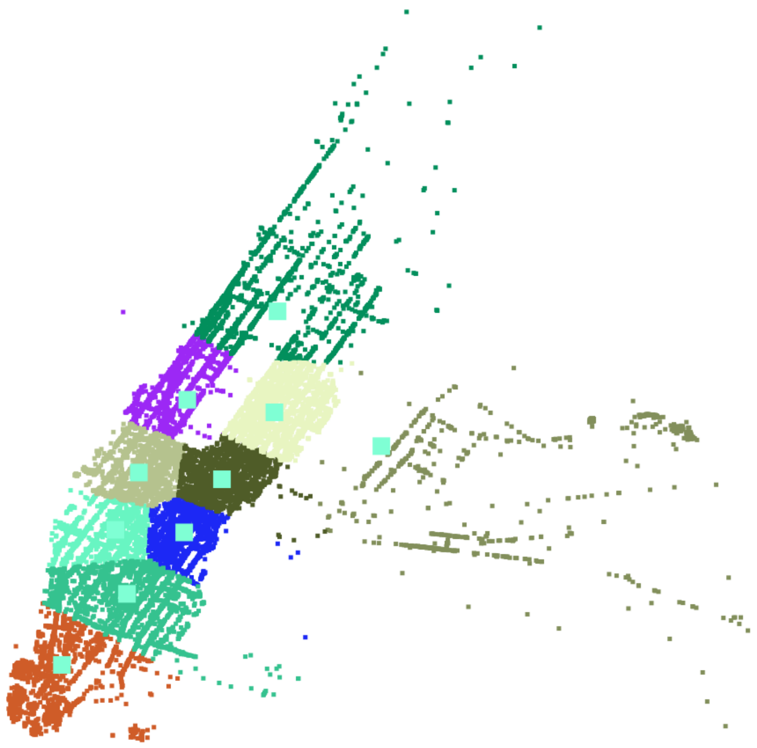


Figure 4.3: Manhattan clustered with k equals 10

as possible on the runtime of the main approach. This section contains a few solutions to deal with these issues.

SUBCLUSTERING

Dividing a large number of requests into many subproblems can be too hard to solve in real-time. However, we can speed up the clustering process by creating a decomposition of the problem.

In order to reduce the runtime of the clustering, the problem is divided into smaller problems. This way a clustering of clusters is formed that can be solved faster and in parallel. When the number of clusters k exceeds a threshold t, the whole problem is split into u subproblems. This process is repeated until there are k/t subproblems left. These are solved separately in parallel. A merge and fill step is performed afterward to remove and combine small neighboring clusters and improve cluster structure for the algorithm to solve.

DYNAMIC ALLOCATION

Vehicles may already contain routes planned for picking up requests and some of those requests can be allocated to another cluster. In order to prevent placing the requests in this trip into separate subproblems, the requests are moved to the cluster of the vehicle. As the vehicle is part of the search in this cluster, it is better to consider replacing requests in its current route as well. Hence, all passengers, who can still be moved to another trip before getting picked-up, are placed in the cluster where its current vehicle is situated.

REUSING CENTROIDS

The centers of the clusters form the base for the allocation. As the demand does not shift by much from moment to moment, these centers can be reused in following iterations of several minutes. This approach can save a time on large instances as the requests can directly be allocated to the nearest center.

Reutilizing the centers is not applied in the current implementation as the computational time of clustering is insignificant compared to the amount of runtime required for solving the ridesharing problem. However, this method provides the ability for scaling up to very large-scale instances. When the clusters are kept to a manageable size, they can all be solved within real-time given each cluster is allocated to an individual thread. In problems of over 10,000 requests, reusing the cluster centers can be effective and could be run as side-job alongside the main process.

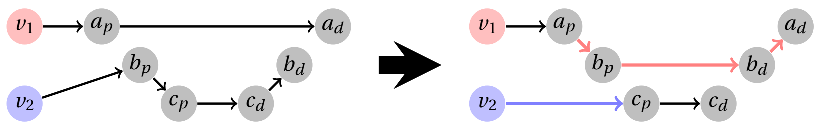


Figure 4.4: **Relocate** Request b is relocated to vehicle v_1. b_p is the pickup of request b and b_d the delivery location

4.3. CONSTRUCTION

Construction creates an initial solution by placing each request in its best position in a vehicle. The construction is done sequentially by fitting each request in the best spot in the current solution. For each request, all possible vehicles are determined and sorted by proximity. The request is inserted in all positions (Section 4.3) in the feasible vehicles and placed in the position with the lowest cost.

There are two methods for inserting a request in a vehicle: a fast method and the general method. The fast method inserts a request in a vehicle as soon as it can place the request in a vehicle. This method decreases the runtime significantly but often provides worse solutions as fewer vehicles are considered. The general method considers all feasible vehicles before deciding on placing the request in the best position.

FINDING FEASIBLE INSERTIONS

The search for feasible insertions is based on Jaw's approach [19] of creating a schedule of a route to determine if a request can fit in an ongoing trip. This method was one of the first papers tackling the construction of PDPTW and is still often used in some form in current methods due to its effectiveness in constructing solutions [35–37].

For each insertion, the request is placed in the route at all possible configurations. Before the request is placed into a trip, a check is performed to verify that no general constraints are violated: A location cannot be served after its time window and cannot cause a vehicle to surpass its maximum capacity. When the request does not violate these constraints, it is placed within the route and checked how it affects the other stops on the route. For every stop a new estimated arrival time is calculated: $t_j^v \geq (t_i^v + s_i + t_{i,j})x_{i,j}^v$. If no time restrictions are violated, the new route is added to a list of possible insertions.

The complexity of this method is $O(n^2)$. Each request contains a pickup and a delivery location the total amount of possible insertions in a route is $\frac{1}{2}(s+1)*(s+2)$, where s is the number of stops already present in the route.

4.4. LOCAL SEARCH

Local search is often used to produce a good solution when solving ridesharing problems or as an initial method prior to using meta-heuristics. The local search heuristics focus on getting many small improvements by looking for promising changes in the solution and primarily accepting moves that are leading to an improvement of the solution. In local search, small operators are used to improve the position of one to a few passengers at a time. These operators are fast in execution and allow the solution to improve quickly on its current schedule.

Local search approaches have been popular for solving VRP problems around two decades ago [12] but have been out-favored by other meta-heuristics that are more focused on diversification in recent research as local search tends to get stuck in sub-optimal solutions. However, the effectiveness of local search in improving initial solutions is beneficial in a dynamic environment where runtime is the largest factor. The heuristics provide stability by only improving small parts of the solution at a time and is focused on intensification to solely improve a single solution as primary objective. For these reasons, local search will be used as the main component in our algorithm.

The operators that our approach applies are a collection of moves from other routing research in combination with a new operator, proposed in this thesis. The operators are variable from considering a single request, two or all requests in a vehicle. This flexibility allows for more diverse exploration of the solution space. Each operator is run on all the requests in the solution and checks for each whether a better allocation exists. Most of these operators are also used in other literature [12, 38, 39]. Empty vehicle, however, is a new heuristic targeted at trying to remove a larger number of requests from vehicles. This operator turned out to be an effective improvement heuristic with similar runtime compared to the other heuristics. The local search procedure consists of the following heuristics:

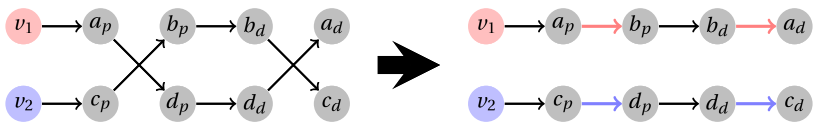


Figure 4.5: **Exchange** The position of requests b and d is exchanged

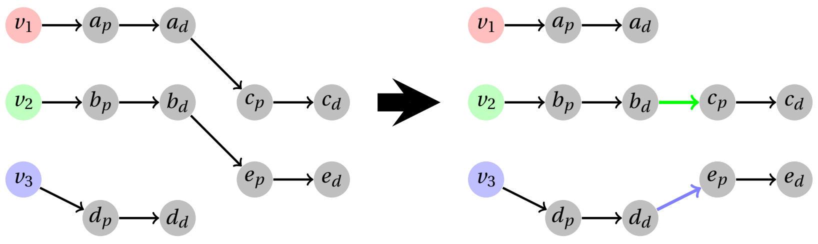


Figure 4.6: **Shift** Request e is moved to vehicle v_3 to allow v_2 to pickup request c

Relocate: Is the most basic move for improving the solution. It takes a request and tries to improve its position. The request is inserted in all feasible vehicles, including the current vehicle, and is moved to the position that has the largest improvement. If the position of the cannot be improved it is left in its current state. In figure 4.4 an example of this is shown where a request b is relocated to v_1.

Exchange: Removes two requests from two different vehicles and tries to reinsert each request in the other vehicle at the best position. Exchange is a pairwise operator and tries to place each request in the other vehicle. The move is accepted if this leads to a cost improvement of the solution. In figure 4.5 an example is shown where an exchange or 2-opt move is performed which means two crossing requests are uncrossed to decrease the total distance.

Shift: is a combination of exchange and relocate. The operator is created by Curtois et al. [39]. It tries to make room for a request by removing a request from another vehicle and inserting it in this vehicle. The removed request is then moved to yet another vehicle. This heuristic allows for more flexibility in finding better allocations as removing requests creates more options for requests to be exchanged between vehicles. Figure 4.6 shows an example of Shift where two requests are moved.

Rejected Reinsertion: This heuristic is created by Luo et al. [38] and is similar to shift but is used as an extra method for placing denied requests. Similar to shift it starts by removing a request from a vehicle to create space in the vehicle for the denied request. If this is possible, it tries to fit the original request anywhere else in the solution. If both requests are placed the move is accepted. Figure 4.7 shows a request c that cannot be placed on any trip in the current solution. However, if b is moved to v_1 it enables vehicle v_2 to serve the rejected request. Rejected reinsertion is used as support for inserting rejected requests and is only called when construction is not able to place a request.

Empty Vehicle: Empty vehicle is a move involving multiple requests to reduce the number of vehicles. The operator removes all passengers from a planned trip and tries to fit them elsewhere in the solution. The move is applied when the new solution requires fewer vehicles or has reduced in cost. The idea of this move is to decrease the number of vehicles used or to improve the position of similar requests.

The idea is a more extensive version of the savings heuristic of Clarke and Wright [17]. Their method tries

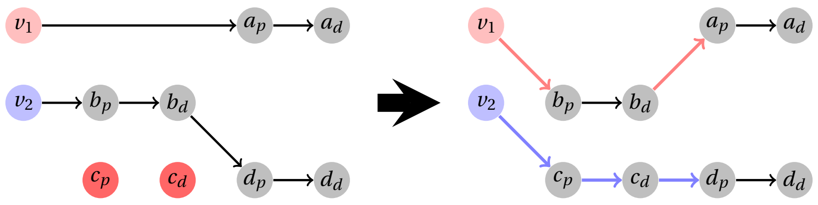


Figure 4.7: **Rejected Reinsertion** Request b is relocated to vehicle v_1 and request c can now be served by vehicle v_2

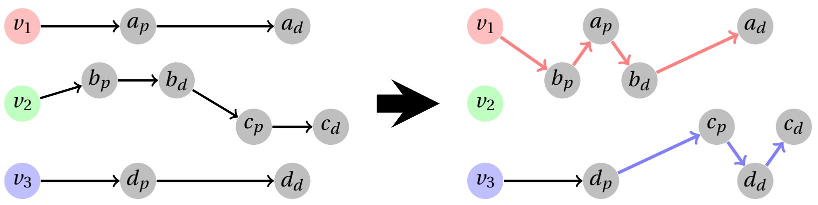


Figure 4.8: **Empty Vehicle** Requests b and c can be removed from vehicle v_2 to reduce the number of vehicles

to merge two routes into a new route if this is possible and save costs. Empty vehicle also moves a route to one or multiple other trips and attempts to reduce costs as well as redundant vehicles. The full approach is described in detail in operator 2. The move is canceled when any of the requests in the vehicle cannot be fitted elsewhere. Figure 4.8 shows a vehicle v_2 that is not necessary for serving the requests in the problem. Both requests can be moved to another route to reduce the number of vehicles and required distance.

Operator 2 Empty vehicle

```
procedure
    Input: Vehicle $V \in \mathcal{V}$, Solution $S$
    Output: Solution $S$
  if $\#Trips \in V < 2$ then return $S$
  $V' = V; \mathcal{R}' = \emptyset$
  for Request $R \in V$ do
      if $R_{status}$ is waiting for pickup then
          Remove $R$ from $V'$; $\mathcal{R}' \leftarrow R$
  $S' = S$
  $\mathcal{V}' = \mathcal{V} \setminus V$
  for $R \in \mathcal{R}'$ do
      $V_{best} = \emptyset$
      for $V' \in \mathcal{V}'$ do
          if $V'^{+}$ and $V^{-}_{best}$ or $(V' == V_{best}$ and $f(V') < f(V_{best}))$ then
              $V_{best} = V'$
      if $V_{best} == \emptyset$ then return $S$
      $S'_{V_{best}} = V_{best}$
  if $S'$ requires fewer vehicles than $S$ or $f(S') < f(S)$ then return $S'$
  return $S$
```

4.5. VARIABLE NEIGHBORHOOD SEARCH

Variable neighborhood search (VNS) is a technique that allows for exploration of local neighborhoods as well as perturbating solutions to escape local minima. Kytojoki et al. [26] applied VNS in search of solutions for VRP instances of up to 20,000 locations and showed that this technique could be effective in solving large-scale problems. The idea is to improve the solution of an instance with local search until no improving moves can be found and then escape the local neighborhood by destroying a larger part of the solution. VNS allows further improvement on problems where local search gets stuck in a local optimum. The technique is applied to provide more scalability to a larger range of problem sizes as smaller problems can be solved more effectively.

The moves of another metaheuristic, large neighborhood search (LNS), are used to perturb the solution. In the paper of Ropke et al. [24] the effectiveness of LNS is shown. Removing and inserting subsets of requests by different heuristics explores large parts of the search space and leads to high-quality solutions. The full potential of LNS cannot be achieved in a real-time environment as large problems require up to an hour of runtime. However, parts of their approach can be used for a more extensive search on smaller problem instances. We apply the removal techniques proposed by Ropke to remove large parts of the solution. Local search is still used but when the operators can no longer improve on the current solution and a local optimum is reached, one of the removal methods is used in order to escape the exhausted neighborhood. The

Algorithm 3 Variable Neighborhood Search

```
1: procedure
2:     Input: List of requests $\mathcal{R}$, list of vehicles $\mathcal{V}$
3:     Output: Solution $S_{best}$
4:     $Preprocessing(\mathcal{R}, \mathcal{V})$
5:     $S = Construction(\mathcal{R}, \mathcal{V})$
6:     $S_{best} = S$
7:     while Time limit is not reached do
8:         while Local search improves the solution do
9:             $LocalSearch(S)$
10:            if $f(S) < f(S_{best})$ then
11:                $S_{best} = S$
12:        if $f(S) > f(S_{best})$ then
13:            $S = S_{best}$
14:        $R' = Destroy(S_R)$
15:        $LocalSearch(S)$
16:        $Construction(R', S_V)$
17:        if $f(S) < f(S_{best})$ then
18:            $S_{best} = S$
       return $S_{best}$
```

removal techniques, discussed in the following subsection 4.5.1, take out a large subset of the requests and reinserts this set into their new best positions using construction (Section 4.3). Local search is reapplied on the reconstructed solution to find whether this neighborhood contains better solutions. The current best neighborhood is stored during the process and is reused when a new neighborhood did not result in an improvement. The implementation of the approach is described in algorithm 3.

4.5.1. REMOVAL

Ropke et al. describe three heuristics for removing requests: Random, Worst, and a removal technique proposed by Shaw [40] based on the relation between requests. Random removal removes a random amount of randomly selected requests from the problem. Worst removal calculates the cost of each request in its current route and removes a random amount of the worst requests from the solution. Shaw removal tries to remove similar requests from the solution as they are likely to be interchanged when reinserted. It calculates their relatedness based on four factors: origin and destination, time window overlap, capacity and the number of vehicles that can serve both requests. Shaw removal selects a random request and calculates the relatedness between this request and every other request. These values are then normalized such that the least related request has a value of 1. Then a random value between 0 and 1 is picked, and all requests that have a higher relatedness index are removed from the solution.

The removal of requests will help diversification when the local search gets stuck. The algorithm first improves the solution with the heuristics of section 4.4 until a local optimum is reached. Then it will start removing batches of requests at a time and reinserting them to provide a way to escape local optima. The use case of VNS is aimed at periods of lower demand as finding local optima in large-scale instances already is time-consuming.

5
Experimental Setup

This chapter covers the different methods and parameters that are used in the experiments of this thesis. The chapter also provides an overview of the underlying models and datasets. This chapter is split into three sections. The first section 5.1 measures the ability of our approach on solving static ridesharing instance, with tests using actual taxi data as well as an existing benchmark. The second section 5.2 provides the setup for the models and properties used in the various simulations of taxi trips in Manhattan. Finally, the last section discusses several supporting methods that are used in the validation.

5.1. Static Instances

Static instances are often used in the validation of routing problems, and most research has been performed on optimizing static routing problems. Dynamic problems introduce more variables and uncertainty to the problem, and therefore we validate our approach on static instances to have a reproducible case without uncertainty. We have chosen to measure our approach on two types of instances. The first is using a subset of actual taxi requests. The second test set is the well-known benchmark of Li & Lim [12], containing many PDPTW instances, which allows us to compare our results with other research in the field.

5.1.1. Taxi Instances

Providing a solution for dynamic ridesharing forms the main focus of this thesis. However, dynamic ridesharing introduces more variables that can influence the results which makes it harder to determine the effectiveness of a method. Hence, static tests of actual situations of a taxi service can offer a more stable way of determining the performance of our approach.

Trip Record Data

The NYC Taxi and Limousine Commission dataset (TLC) [41] is currently the largest open dataset of recorded taxi trips. The database contains data from a huge number of trips made in Yellow, Green and, FHV taxi service. The data specifies the GPS coordinates of the origin and destination of a trip as well as the corresponding recorded arrival times. The TLC data also contains the number of passengers, the distance traveled and cost of the ride. For the simulation data of 2013 is used as more recent data uses taxi zones instead of GPS coordinates which makes simulations less accurate.

After most of the simulations and experiments were run, a potential flaw in the data was found. In the factbook of 2014 [42], an average of 1.3 passengers per trip was reported over the last 5 years, while in the data we used an average of 2.1 passengers per trip was found. The data of a specific week might coincidentally have a large deviation from the average, but it is more likely incomplete or contains mistakes. Therefore, results found in the simulations might be optimistic as more passengers per trip makes combining passengers easier for ridesharing.

Setup

For test instances, we will use a small sample of recorded trips. However, a batch of requests from a planning iteration of 10 seconds contains too little requests to test the scalability of our approach. Instead, we will use intervals of 5 minutes that contain between 1000 and 2000 passengers during the daytime. This interval also better represents the number of requests in the active pool at any time as the maximum waiting time for requests is 5 minutes (Section 5.2.1). The active pool contains the requests that have not yet been picked up by a vehicle and can be moved to between trips.

5.1.2. BENCHMARK INSTANCES

To validate the effectiveness of our approach, a comparison with an existing benchmark can provide a better overview of the performance. The benchmark of Li and Lim [12] is chosen, as it is widely used in other literature [24, 39, 43, 44], and considered an effective benchmark for PDPTW instances. The benchmark consists of six sets of instances for the PDPTW, varying in size from 100 to 1000 locations. The instances are open source, and can be found on SINTEF[1]. The primary goal in these instances is to minimize the number of vehicles, and as the secondary objective to minimize the total travel distance.

Since approximate best known solutions are known for these instances, an estimation can be made of how well our approach performs compared to other methods. The best results have been calculated with far more extensive methods than presented in this thesis. The goal is not to improve these results but rather to offer an insight into the performance of our method within a short runtime.

SETUP

Scalability is the main concern of this thesis, and hence only the largest instances, containing 1000 customer locations, are considered. For solving these instances, we increase the base cost of a vehicle to a large value in the objective function (Section 2.3.1). The primary goal is to minimize vehicles, and therefore α_3 is set to value greater than the furthest distance in an instance. This makes deploying a new vehicle always more expensive than driving to the start of any other trip.

The instances are also clustered using the K-Means approach proposed in section 4.2. The vehicles are divided equally over the subproblems as there is only a single depot in these instances.

5.2. DYNAMIC RIDESHARING

As our approach is focused on serving passengers in a real-time environment, a simulation is required that estimates the effectiveness in practice. This section is split into four parts that discuss the setup of our simulations in order to create representative results for real-time ridesharing in practice. The first section 5.2.1 substantiates what values and parameters are chosen for the simulations. The second section 6.4.1 introduces the performance indicators used to measure the effectiveness of the simulations. These differ from the usual objectives as they account for factors that arise in a dynamic environment. The following section 5.2.2 discusses the underlying methods for calculating the travel time and distance between two locations in order to create more accurate simulations. The last section 5.3 presents a few supporting methods that are required for improving the results in the simulations but are not the focus of this work.

5.2.1. SIMULATION SPECIFICATION

During the simulation, certain properties and bounds are used (Table 5.1). The fleet size represents a constant amount of vehicles that are used during the full simulation and the time windows are hard bounds for the request. A request that cannot be serviced within its time window will be rejected. The service time is always 15 seconds and gives an estimate of how long it takes for passengers to hop in or out of a vehicle. Most of the values in the table are based on Alonso Mora et al. [4] who found that these values suited the density and number of requests in Manhattan. For this reason, we will use similar values for the main parameters.

When customers request a trip with each other, they are considered as a single request to respect their wish to travel together. However, when the number of customers in a trip exceeds the maximum capacity of the vehicles, it is split into multiple requests with a capacity equal to the maximum capacity and a last request with the remaining passengers.

MANHATTAN

To test the effects and scalability of our approach the data of the TLC is used 5.1.1. The data of several days in May 2013 is used containing around 450,000 customers each day. The simulations are run in iteration steps. As stated in the requirements a fast response time is desired, hence every 5 or 10 seconds the current best solution is accepted, and all vehicles and trips are updated.

[1] https://www.sintef.no/projectweb/top/pdptw/li-lim-benchmark/

Property	Default value	Description
Capacity	4	Maximum number of passengers in a vehicle
Fleet size	2000	Number of vehicles deployed in a simulation
Pickup Window (Ω)	5 min	Maximum waiting time for a request
Dropoff Window (Δ)	10 min	Maximum total delay for a request
Service time	15 seconds	Time the vehicle stops for picking up or dropping off a customer

Table 5.1: Information of settings used in the simulations

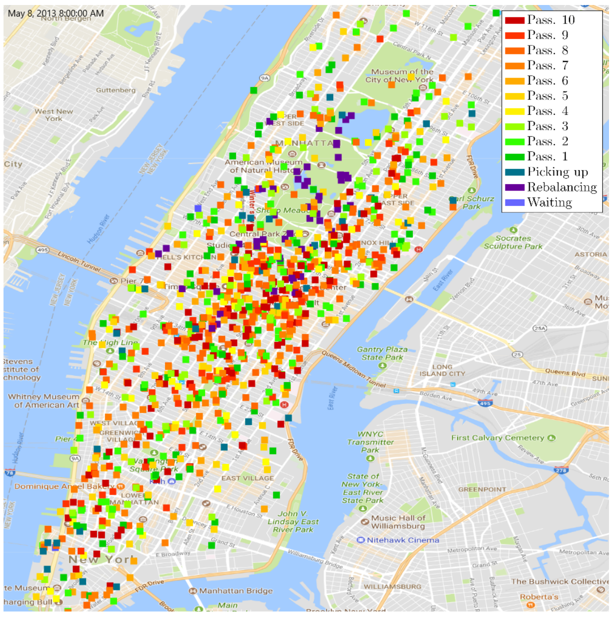


Figure 5.1: Still of the vehicle fleet during a simulation

In the simulations, we assume a homogeneous fleet of self-driving vehicles. This fleet allows some relaxations of the problem as shifts, breaks and placement of vehicles are of less importance. Removing these constraints provides clearer results of our method as fewer factors are affecting the results. For the simulation, we build a visualization tool to view the effect of the developed methods and potential problems. Figure 5.1 shows an example of the vehicle fleet operating during a simulation.

Runtime

Since this is a real-time problem, we want to strictly bound the amount of runtime a planning iteration can take. In an iteration, the requests are clustered into subproblems, and these are divided over the available threads. If the amount of clusters (C) exceeds the number of threads (T), the algorithm creates a queue of the remaining jobs that have to wait until a thread finishes. Since the maximum runtime (r) is bound, the jobs (j) have to be reduced in runtime as to finish to queue within the limit. The following equation gives the runtime of a job: $r_j = \frac{r}{\lceil \frac{C}{T} \rceil}$.

Cluster Size

The K-means algorithm has a set value k that determines in how many clusters the current problem should be split. In a real-time environment, the amount of requests fluctuates over time, and the number of clusters should be adjusted according to the size of the problem. A cluster size cs is declared that gives an average value of the number of requests per cluster. The amount of clusters is then determined by $k = \frac{n}{cs}$, where n is the total amount of requests in the problem. The cluster size allows the problem to scale to the varying amount of demand and ensures that the problems are bounded by a maximum and can be solved within the maximum runtime. The goal is to create a split in subproblems of equal size, but in order to maintain general structures of areas, the requests are mapped to the nearest clusters. The number of requests in a cluster consequently will have some variable clusters containing between half to double the number of requests set in cs.

K-Means Variables

In our clustering algorithm 1 we use two variables x and y in certain steps. The influence of these parameters is not very significant, but these parts have to be run several times to improve the cluster centers. For x we set a value of 3 as the centers are fairly stable this number of iterations. The value 10 is chosen for y because the balancing step often stabilizes within 4-7 steps and this ensures that the areas are more smoothed out. This iterations also contains a break clause if an acceptable solution or steady state is found earlier.

5.2.2. Shortest Path Calculation

In order to create realistic simulations, it is important to be able to calculate accurate ride times between two locations. Accurate times will provide a better overview of what results can be gained with ridesharing and how well they will hold up in practice. Because Much research has been performed on fast calculations for large road networks [3, 29, 45], we did not spend the time to create a system for simulating vehicles on an existing road network. Instead, we took an alternative that is commonly used in static instances; take the direct path between two locations, also known as the Euclidean distance. This method is less accurate for approximating the travel time of individual trips but can provide a representative simulation for large numbers. This section compares estimations of the travel time of measurements with and without actual routes and shows that the estimations are comparable although contain some deviation. Since the Euclidean distance is often shorter than an actual route, we lower the speed of the vehicles based on travel times estimations.

This section will look into three methods of calculating the shortest path between two points: Euclidean, Manhattan and using an origin-destination matrix (OD matrix).

Euclidean Distance

The method to calculate the ride time is based on the Euclidean distance and the original ride time of the trip. The current data provide two GPS locations for pick-up and drop-off as well as their realized times. So with this data, the distance between the two points can be calculated and divided by the ride time to get a realistic speed of the trip. When combining two trips, the speed is averaged to get an approximation. Since these points are often close in proximity, it is assumed this will give a proper estimation of the travel time.

This method will give a quite accurate representation of the ride times where the influence of traffic is taken into account and also assures that all trips are possible within their time window when a vehicle is nearby.

The driven routes will be calculated using Euclidean distance and are likely to be shorter than the shortest possible route in practice. However, since the speed is based on the original ride time, this will slow down the trip speed and provide a realistic estimation. A disadvantage of this method is the inconsistencies in the data as very similar trips can still vary in trip speed.

Distance Matrix

Distance matrices are common practice in routing problem where many route calculations have to be performed in short amounts of time. The matrix consists of all locations in the problem or a set with predefined locations in the area. Between every two locations in the set, the shortest route is calculated and stored with its drive time and distance. Because this matrix can be calculated before the actual simulation, it can provide fast estimations of routes between any two points in the area.

The matrix used in this thesis is based on GraphHopper [46], which uses Open StreetMap Data. All the main crosspoints of Manhattan are extracted from the street map data. This leads to a total of 4000 main locations of Manhattan that can be used as stops for actual requests. This method is similar to the method created in Santi et al. [3].

GraphHopper does not take into account any form of traffic, and the travel times are based on the speed limits of the roads. However, when comparing to the actual NYC data, the ride times are often 2-3 times slower. To create a more realistic simulation, a compensation for traffic is necessary. As can be seen in figure 5.2 the data shows a fluctuation over the day and is substantially lower than the general speed limit of 50 km/h. The estimated times are compensated for the traffic difference by assuming the average speed on all roads is equal to the mean traffic speed at that time.

MANHATTAN DISTANCE

The Manhattan distance is a distance calculation between two points as if they were represented on a grid. This distance is measured by adding the vertical and horizontal distance of two points. It is assumed as a decent fit for ridesharing in cities as they often consist of street blocks and form a grid.

The method takes the Manhattan distance of a route and estimates the travel time by taking the average speed at this time in the city. The average speed is determined from recorded trips.

DISTANCE MODEL

This section proposes three models for calculating travel times and distance. The Euclidean method turned out to be less reliable for matching trips due to the high variance between similar trips in the recorded data, as can be seen in the deviation of travel time in table 5.2. This row shows the average difference in travel time between the recorded trips and the estimations from the models. Due to the average deviation of almost 4 minutes, combining trips with this method can lead to inaccurate speeds overall.

Furthermore, the table compares the accuracy of the other two methods; the distance matrix and the Manhattan distance. The results represent the average results of a week of requests from Manhattan. Each request in the recorded data is measured against the calculated time, distance and speed of the two models.

	Original	Matrix	Manhattan
Mean Trip Travel time (s)	676	627	636
Mean Trip Distance (m)	3169	3122	3172
Mean Trip Speed (m/s)	5.03	5.11	5.04
Mean Time Deviation Original (s)	0	223	223
Mean Time Deviation Matrix (s)	223	0	124

Table 5.2: Overview of the accuracy of different distance models

Both models present estimated travel times that are close to the actual trips and are on average 45 seconds faster. Whereas the estimations of the Manhattan model are closer to the original data in all three regards, this method is chosen for calculating the shortest paths. The deviation between the models is significant, and thus less accurate for an individual request. However as stated before, the estimated travel times of the Manhattan model are very close to the original trip data, and therefore the simulations can provide a fairly accurate, although slightly optimistic, representation for results in practice.

TRAFFIC

In Manhattan, the amount of traffic fluctuates greatly throughout the day. The calculations of the Distance matrix uses the speed limits of the roads to calculate travel times.
Figure 5.2 shows the average driving speed in Manhattan on every hour of the day. This speed is based on a week of data of trips in Manhattan with distance driven and the corresponding drive time. As can be seen, the actual speeds driven are significantly lower. To compensate for the slower drive speeds the calculated travel times are based on the data of figure 5.2 and use the average speed over the whole trip.

5.3. SUPPORTING METHODS

In a dynamic environment, several methods are necessary to increase the effectiveness of calculations during a planning iteration. This section discusses the preprocessing step that reduces the amount of unnecessary computing time, and vehicle rebalancing, that can improve the location of the vehicles in order to increase the service rate.

5.3.1. PREPROCESSING AND BOUNDING CANDIDATE TRIPS

In several other large-scale dynamic ridesharing solutions, some form of preprocessing is performed to restrict the number of possible trips between vehicles and requests [4, 47].

The first step of preprocessing is to eliminate infeasible trips. A trip is infeasible when a vehicle cannot reach the pickup location of a request within the maximum time window. Another elimination is done based on capacity. When a vehicle cannot fit a request within its maximum pickup window, the vehicle will also be removed from the possible trips. After eliminating the infeasible vehicles for each request, the list of vehicles is sorted by proximity to speed up the insertion later on. Vehicles that are closer to a request are more likely

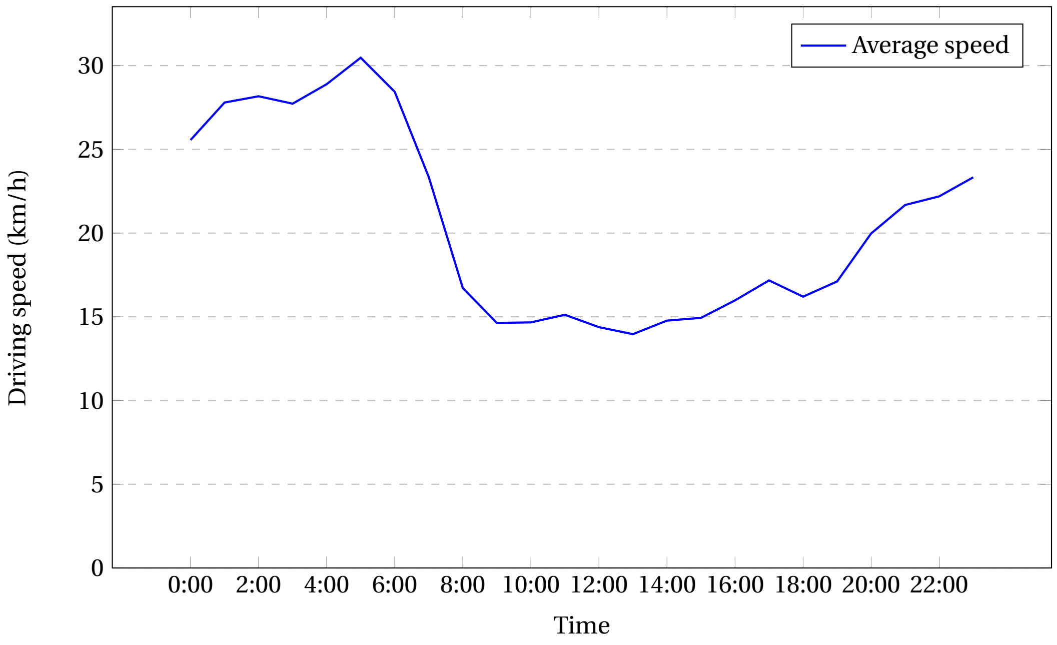


Figure 5.2: Average driving speed in Manhattan

to be a better match as the distance and waiting time is lower.

This concept can be extended by removing vehicles that are not likely to be suitable to pick up a request. Since the cost of nearby vehicles is often lower, bounding the search to a subset of the closest vehicles can provide an advantage in speeding up the placement of requests.

5.3.2. VEHICLE REBALANCING

Vehicle rebalancing is important during a simulation. During the day the pickup and dropoff hotspots are often different, and without rebalancing, it is possible for many vehicles to end up stranded in low-demand areas and have a hard time finding new requests. This causes the amount of rejected requests to rise while many vehicles are left unused. A rebalancing step, where idle vehicles are relocated to locations of high-demand or low vehicle density, provides the ability to serve more customers with the same vehicle fleet.

The current approach of rebalancing is matching idle vehicles with rejected requests. At locations where vehicles are sparse, requests are more likely to be rejected. Thus after every planning iteration where requests were rejected, several idle vehicles are redirected to the locations of these rejected requests.

For every rejected request a list of all idle vehicles with travel times is created and sorted by proximity. A subset of the closest vehicles is relocated to the rejected request. This number has a threshold, τ, which represents the maximum number of vehicles per rebalancing move. The number of vehicles that is rebalancing at any time is bounded to reduce the amount of unnecessary relocated vehicles. In our simulation is tau is set to 2% of the rebalancing bound M and M is set at 10% of the total fleet. These values worked well in the simulations but could be optimized to work more efficiently in practice or for other cities.

For every new rejected request the number of vehicles to be relocated, r, is calculated based on the number of vehicles currently rebalancing, R. The value of r is calculated as follows:

$$r = \tau * (1 - \frac{R}{M})$$

The variable number vehicles that are sent out allows for more effective rebalancing moves. It provides more efficient moves at first and reduces the number for rejections at similar locations or when the fleet is unable to serve all requests. Every rebalancing vehicle is still able to pick up requests on its way, which is also the reason to rebalance more than one vehicle, as it is likely several vehicles will start picking up customers during or after rebalancing.

5.3.3. VARIABLE CLUSTER METHOD

The goal of this research is developing a fast and scalable algorithm. However, to validate some of our results, a method that is able to find higher quality solutions can provide an indication of the solution quality that can be reached within the time limit. Instead of re-implementing a full existing approach, we chose to extend our current approach that uses the effectiveness of the clustering method. The proposed VNS approach is very slow at improving very large-scale solutions without clustering, and thus this faster exploration method is used.

Clustering introduces strict bounds that remove possible matches from the problem and can thus lose the potential for finding high-quality solutions. To minimize this effect, we propose a variable cluster method that changes the allocation of a request into subproblems during the process. The method varies the number of the cluster during the process to change the locations of the area bounds. Due to the shift of these bounds, requests are able to match with every other request at some point which diminishes the influence of the decomposition.

The method starts by solving the whole problem at once and will then switch to solve clustered instances based on varying values of k, i.e., the number of clusters. The value of k is sequentially rotated between 1 and 16, but these values can vary based on the size of the problem and the available runtime. The algorithm solves the subproblems for a period of 30 seconds for each k and continues until it cannot find any improvements in a full cluster iteration.

This method is used in this thesis for finding better solutions to static instances as it significantly faster at improving large-scale routing problem than applying our approach on a non-clustered problem.

6 Empirical Evaluation

In this chapter, the results of the different methods of chapters 4 and 5 are discussed. This section is split into 3 parts: the efficiency of the algorithm on static instances, the effect of clustering on the ridesharing problem and, lastly, the influence and results of varying factors in the dynamic taxi environment.

6.1. Overview Validation

In this chapter the validation and results of our approach are presented. At the start of this report, three research question were formulated:

RQ1 What algorithms can effectively be scaled to work in a dynamic environment?

RQ2 What is the influence of applying decomposition to large-scale routing problems?

RQ3 Is it possible to create a scalable algorithm that has to complete within a limited time and is able to create effective allocations for on-demand ridesharing passengers?

First, an answer to the first research question is provided by validating the several components of the main algorithm. Since the dynamic ridesharing is a complex problem it is important to validate the approach from the ground up to get a clearer picture of the overall effectiveness. Section 6.2 provides the validation of the algorithm of our approach. This means that the construction, local search and variable neighborhood search are tested on providing efficient and scalable results for the general ridesharing problem. Because these results are validated on a benchmark, it provides a more stable overview of its ability of solving the PDPTW in general.

Next, in order to answer the second research question, the influence of decomposition is validated by testing the proposed clustering method. Section 6.3 shows the results of the largest contribution of this thesis, the effect of decompositioning or clustering of the problem. In this section the scalability of the clustering as well as its effectiveness are presented.

Finally, the third and main research question of this thesis is discussed and the potential benefits of our method are shown by simulating a taxi ridesharing system. As the previous sections validate the ability of the approach to handle static instances, we want to further look into how well our system is able to handle real-time on-demand scenarios. This is validated in section 6.4 by running simulations with actual taxi trip data and showing how many customers can be placed and how much delay they experience.

Because dynamic ridesharing is a large and complex problem with many variables and constraints, the last section 6.5 discusses the results for multiple different dynamic ridesharing situations. The experiments look at variations of fleet size, capacity, time windows of customers, the objective function and the service rate.

6.2. Validation of the Main Algorithm

The proposed approach has two components in order to provide much scalability for solving ridesharing problems in a limited runtime. This section discusses the effectiveness of our algorithm in finding solutions

Locations in test set (LC1,LR1,LRC1) Metric (Average gap)	200 Vehicles	200 Cost	1000 Vehicles	1000 Cost
Construction	+43.63%	+65.38%	+42.95%	+50.44%
Relocate	+19.11%	+1.28%	+34.23%	+23.03%
Shift	+23.00%	+3.23%	+40.34%	+36.51%
Exchange	+32.61%	+12.17%	+42.54%	+38.58%
Empty	+11.66%	+6.21%	**+22.38%**	+25.00%
Relocate, Shift	+20.95%	+1.45%	+35.86%	+24.59%
Relocate, Exchange	+25.81%	+2.95%	+37.05%	+25.70%
Relocate, Empty	+10.91%	+0.78%	+23.28%	**+20.68%**
Shift, Exchange	+25.59%	+4.62%	+40.80%	+36.55%
Shift, Empty	+11.12%	+0.42%	+37.67%	+35.47%
Exchange, Empty	**+10.58%**	+2.19%	+33.38%	+33.04%
Relocate, Shift, Exchange	+23.97%	+1.95%	+37.08%	+25.29%
Relocate, Shift, Empty	+11.12%	-0.41%	+34.15%	+23.95%
Relocate, Exchange, Empty	+11.02%	-0.69%	+32.79%	+24.29%
Shift, Exchange, Empty	+11.56%	**-0.94%**	+39.79%	+36.53%
Relocate, Shift, Exchange, Empty	+11.45%	-0.47%	+37.05%	+25.35%

Table 6.1: Effectiveness of the local search operators with a runtime of 5 seconds. Baseline are the best known solutions on SINTEF [48]. **Average gap** shows the average relative difference between the metric in the found solution and the best known solution.

for static problem instances in bounded runtimes. First, the main results of the proposed local search operators are presented. Next, the impact of including variable neighborhood search is measured to help improving solutions for smaller instances.

6.2.1. LOCAL SEARCH

As discussed in Section 4.4, our local search method applies 4 main heuristics to improve the current solution. These are Relocate, Exchange, Shift and Empty Vehicle (Empty). In other local search approaches, all moves are commonly applied during runtime. However, as runtime is an important factor in this research and , this section validates which operators offer the most reduction in cost in a small time window.

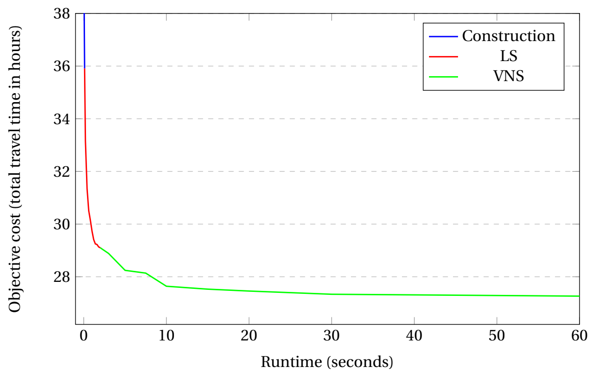
In the validation, the time window is set to 5 seconds as is discussed in the requirements (Section 2.2). Three sets (lc1,lr1,lrc1) of the Li & Lim [12] instances are used as they require more vehicles compared to the other sets which more closely resembles the dynamic taxi environment. The full algorithm (Section 4.3 - 4.5) is used and only the operators used in during local search vary. The results are compared to the best known solutions (BKS) [48] to test the effectiveness of each of the operators.

Table 6.1 provides a few noticeable results. Some of the results show costs below the BKS. This negative value is sometimes possible when a solution uses more vehicles than the BKS, but these solutions are not more optimal as the main objective is to minimize the number of vehicles. Empty vehicle is the most effective operator as it able to reduce the number of vehicles very quickly which also reduces the cost. The next best operator is Relocate. This move can decrease the cost of an instance rapidly and in combination with Empty is already able to find decent solutions quickly. The difference between applying all operators in the smaller and larger instances also stands out. In the sets with 200 locations all operators actually support each other and provide some of the best results, while some of them are too slow in the large instances. This can be seen in the combination Shift, Exchange and Empty. The faster operator, Relocate, is missing here and this leads to significantly worse solutions.

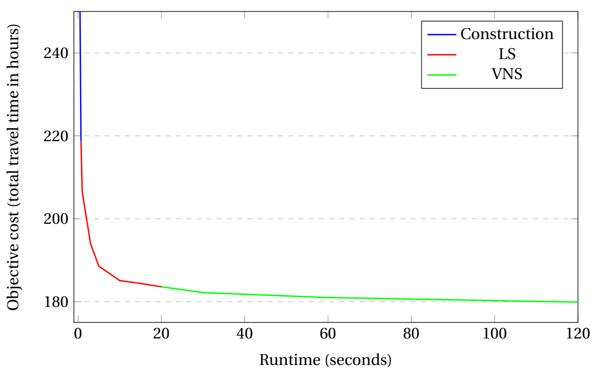
All of the operators have their own strengths and weaknesses and the broad range provides a solid base for finding solutions in real-time. However, when time is of the essence, it is better to solely employ Relocate and Empty as they are more efficient operators.

6.2.2. VARIABLE NEIGHBORHOOD SEARCH

This section shows the results of VNS in combination with the other heuristics as complementary approach after local search is no longer able to improve the current solution. VNS is not always applied in the search as local search might not reach a local optimum within the time limit. In figure 6.1 the results of the three



(a) Instance containing 300 requests



(b) Instance containing 1500 requests

Figure 6.1: Effect of heuristic methods compared to runtime

approaches are shown. The three methods are used in succession to speed up the search. When a line in these graphs ends the method cannot improve the solution any further after which the next method is deployed. The results show the ability of the different heuristics to adapt to the size of the problem in the provided runtime. On the small instance (Figure 6.1a) construction finishes within 100ms and in 2 seconds local search is no longer able to improve the solution. VNS can still reduce the total time by 5% after this within 10 seconds. However, in the larger instance (Figure 6.1b), VNS does not come into action in real-time. Local search is effective for up to 10 seconds and the total travel time can only be reduced by 3% in the following 2 minutes.

While larger instances do not benefit as much from VNS, it can help optimize smaller problems. The combination of the three methods shows to the scalability of these approaches as local search can quickly improve on a solution early on, while VNS can help with diversification in the remaining time.

6.3. CLUSTERING

This section shows the effect of K-Means clustering on static instances. The section validates the effect of clustering in 4 segments:

- The time clustering requires in a planning iteration
- Results of clustering for dynamic taxi ridesharing data
- The suboptimality introduced by decompositioning the problem into multiple regions
- Results of clustering on static PDPTW benchmark instances

This section wants to validate the proposed clustering approach from initiation time to its potential benefits and drawbacks on multiple types of routing instances.

6.3.1. RUNTIME OF CLUSTERING

The current K-Means implementation is developed to create a practical decomposition within a short runtime. This short initiation period means the clustering step can be run before every iteration as its runtime only uses a small part of a planning iteration. Clustering every step leads to more up-to-date clusters in which the incoming requests are considered as well.

In figure 6.2 the different number of requests are set out against different sizes of cluster sizes. The number of requests per subproblem or cluster size is based on the results of the previous section 6.3. However, compared to the number of requests in a problem has, the number of clusters they are split into is less significant. The runtime varies from 20 ms to up to 10 seconds for huge instances. The current approach is aimed at a scale of up to three times New York, and this is equal to up to 5000 requests with a corresponding cluster time of approximately 200 milliseconds. This period could conveniently fit within a planning iteration of 5 to 10 seconds, and therefore the clusters can be recalculated at the start of every step.

6.3.2. CLUSTERING TAXI DATA

The impact of clustering on taxi data is important to show the potential benefits of providing scalability on handling on-demand ridesharing in large urban areas. In figure 6.3a the results are shown of a 5-minute

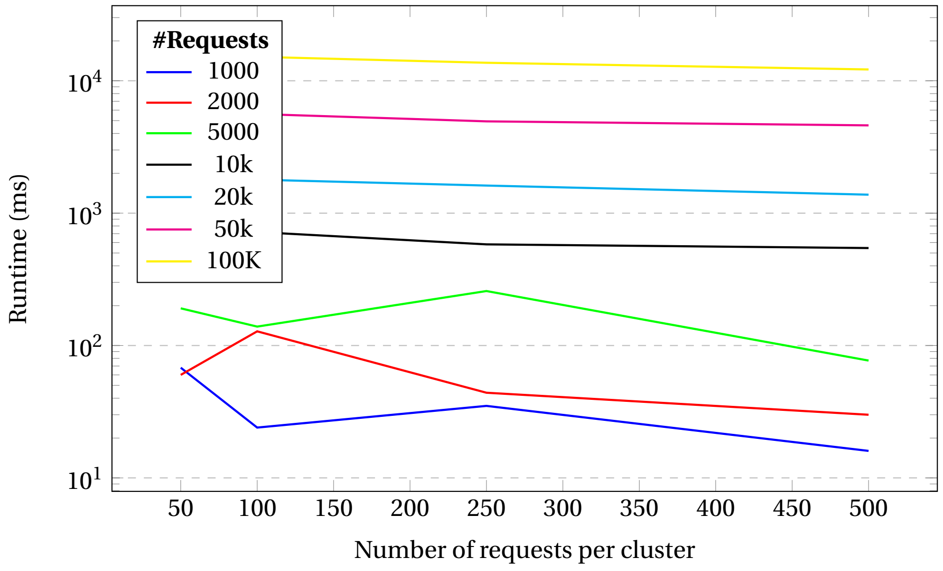


Figure 6.2: Calculation times of number of requests split into subproblems with bound number of requests

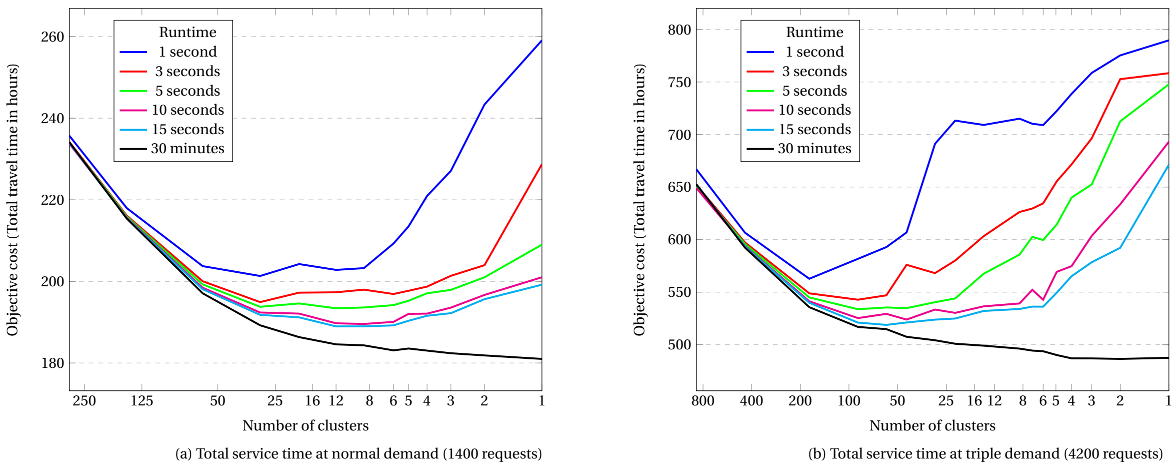


Figure 6.3: Number of clusters in the problem compared to the solution quality for different runtimes

request window around evening peak hour containing around 1400 requests. The effect of clustering is substantial in small time windows. If the algorithm is bound to a runtime of 1 second, the amount of travel time required to serve all passengers is 30% higher compared to the unclustered problem. As we approach a runtime of 15 seconds, clustering becomes less of an advantage but is still 5% more effective.

This result is even clearer if the problem is scaled up to 4200 requests (Figure 6.3b) consisting of three times the usual amount of traffic at peak hour. In this case, the fleet is also tripled to 6000 vehicles in order to reduce rejected requests as a large factor. The most efficient cluster sizes are able to outperform the unclustered problems by a large margin. At all runtimes, the clustered problems outperform the unclustered problem by 30-40%. The black line in these figures shows the performance with a longer runtime of 30 minutes and shows the best result that can be found by the algorithm. The quality loss that clustering introduces is minimal compared to the performance it provides in larger problems. When an instance is clustered into 50 or more subproblems, the algorithm can find a solution within 1% of this limit within 15 seconds. The best real-time solutions that are found using clustering are roughly only 5% worse than the best found solution.

Runtime (seconds)	1	3	5	10	15
No preprocessing (meters)	Not finished	469770	449001	412076	406669
Preprocessing (meters)	486565	427820	391099	365913	362359

Table 6.2: Total distance required when solved with and without preprocessing using different runtimes

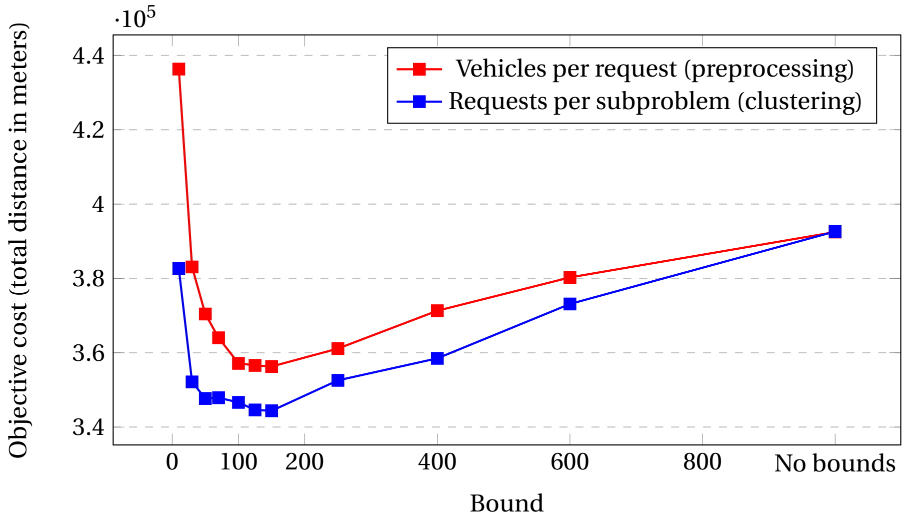


Figure 6.4: Solution quality for bounding the number of vehicles compared to our clustering approach in a runtime of 10 seconds (1000 requests)

6.3.3. Comparison to Bounding Candidate Trips

A drawback of clustering is that reduces the number of potential ridesharing matches. As decompositon also removes links from the problem the smaller subproblems might be unable to reconstruct a good overall solution. This section looks at the impact of clustering on the ridesharing problem as well as an other approach to reduce another common trade-off to speedup the search by bounding the links between vehicles and requests.

A common approach for reducing the number of problem-constraints is to limit the number of vehicles that are considered for picking up a request. Preprocessing (Section 5.3.1) is used to exclude infeasible trips when searching for ridesharing matches during runtime. In this section, two methods that reduce the size of the problem are tested. The first one is bounding the number of vehicles per request i.e., solely consider the most promising vehicles for each request. The second one is using the proposed clustering method which creates subproblems consisting of a certain number of requests.

The first step of preprocessing is to remove infeasible vehicles and sort the remaining vehicles on the proximity to pickup location of the request. The tests are run on a 5 minute batch of New York containing around 1000 requests, and a fleet of 3000 vehicles of capacity 4 is used. By tidying up the potential vehicles, an average of 10% of the total distance is saved (Table 6.2).

As previously discussed (Section 5.3.1), the preprocessing step can be extended to find better solutions at a faster pace. The closest vehicles are more likely to provide better matches for requests, and therefore considering a subset of nearby vehicles can improve rapid search. To verify this hypothesis, different bounds where chosen for both the number of vehicles and number of requests per subproblem and were run for 10 seconds in the same scenario. Both methods (Figure 6.4) show that this hypothesis holds. The elimination of less likely matches creates a significant speed-up, and better results are realized in runtimes of 10 seconds and lower. However, there are limits to this method. Too small of a bound eliminates many viable matches and using larger bounds makes the algorithm less capable of solving the problem in the short runtime. This balance means that when either method is used, it is important to use a value that provides the greatest speed-up for the least infraction of suboptimality. The proposed clustering method outperforms the vehicles per requests on every bound. Bounding the number of vehicles even leads to lower optimal solutions compared to an area division containing a similar number of requests.

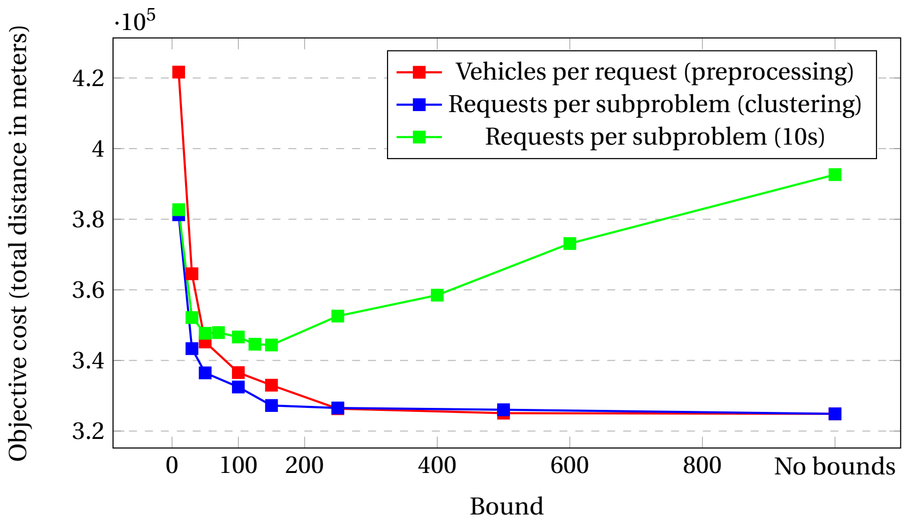


Figure 6.5: Best found solutions for bounding the number of vehicles compared to our clustering approach (1000 requests)

Effect Clustering on Optimality

Clustering and preprocessing both introduce some suboptimality as viable matches are excluded from the search space. To test the influence of these bounds on the possible lowest solution a run was performed where an extended runtime of up to a half hour was used. The different clusters sizes are compared to similar bounds of vehicle-trip matches (Figure 6.5). The graph shows the lowest total travel time required to serve all passengers. The variable cluster method of section 5.3.3 was used to speed up the process of finding better solutions.

Figure 6.5 shows that the problems solved with clustering provide better solutions than bounding the number of vehicles on for values lower than 200. After that, the best results are very close but are too insignificant to affect the solution quality in real-time. The reason that limitations on the number of vehicles provide more suboptimality is likely due to the higher number of trip eliminations as every request has its individual vehicle range. The figure also shows the effectiveness of finding solutions in clustered instances. When the number of requests per cluster reaches 50, the solution is close to the best possible solution for this decomposition. The optimality gap for the allocation with 50 requests per cluster is just 7% compared to the overall best found solution.

6.3.4. Results of Benchmark Instances

The following section contains the results for the largest Li & Lim [12] data instances containing 1000 costumer locations. The graphs show the solution gap to the best known solution (BKS) found on SINTEF [48]. The solution gap represents the relative difference to the BKS for either the number of vehicles or total cost.

In figure 6.6 the results of different cluster sizes are shown. Each instance is run for 10 seconds and the average result over 3 runs is taken. The figure shows that clustering provides better solutions in general on the benchmark instances. The best results for both the number of vehicles as well as the minimal cost is using 4 clusters. The gap to the best known solutions is 16% closer for the required number of vehicles and 27% for the travel distance when clustering is applied. A value of 4 clusters is equal to a cluster size (#Requests per cluster) of 125 which is also shown to be highly effective in dynamic ridesharing scenarios (Figure 6.4).

The required cost drops fast after the introduction of clustering and stabilizes around $k = 4$. The required number of vehicles also improves at a smaller number of clusters but starts increasing after 4 clusters. Alternative routes with a short total distance can still be found in high-clustered problems while combining rides in the least amount of vehicles becomes harder. The increased number of strict bounds in the problem makes it less likely to create long trips with a single vehicle.

Results for the Individual Sets

The Li & Lim [12] benchmark consists of six sets that focus on varying distributions of the customers over an area. The customers in LC instances are grouped together, in LR customers are randomly distributed and customers in LRC instances are partially clustered and partially placed at random. In table 6.3 the results

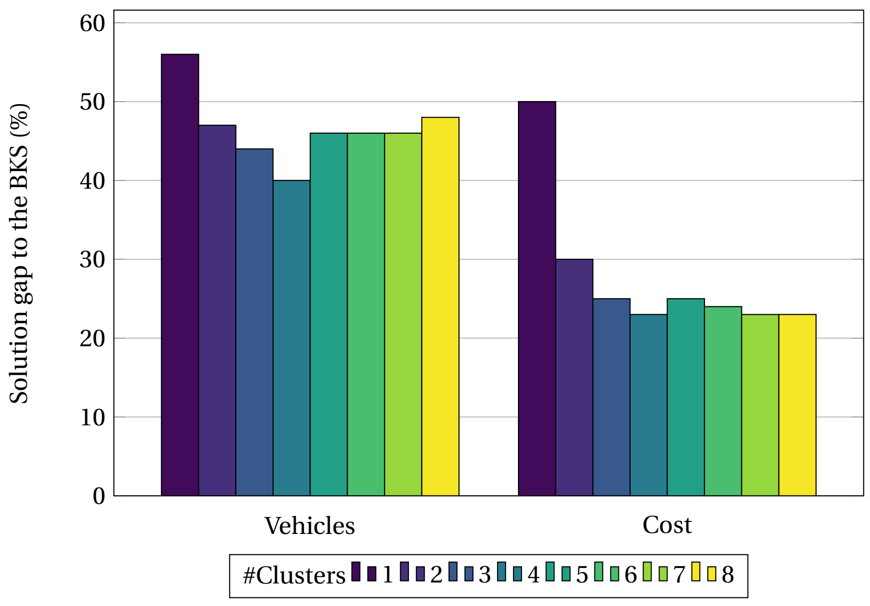


Figure 6.6: Shows the average solution gap, difference to the best known solution (BKS), for the 1000 customer instances of Li & Lim for different number of clusters

Benchmark set	**Average Vehicle Gap**	**Average Distance Gap**
lc1	+6%	+4%
lc2	+30%	+47%
lr1	+34%	+10%
lr2	+104%	+32%
lrc1	+27%	+15%
lrc2	+102%	+46%

Table 6.3: Results of the various sets of Li & Lim's 1000 customer instances using 4 clusters. The gap represents the relative difference to the best known solution

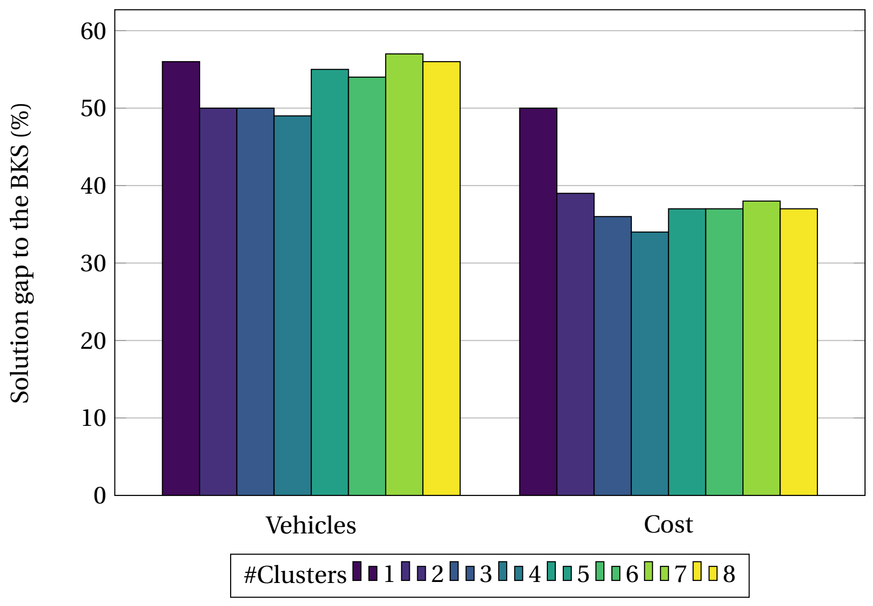


Figure 6.7: Effect of Parallelization: Shows the average gap, difference to the best known solution (BKS), for the 1000 customer instances of Li & Lim using a runtime scaled to the number of clusters

of the different instance sets are shown using 4 clusters. The algorithm struggles to produce good solutions for instances of the second group of the benchmark (lc2,lr2,lrc2). The BKSs of these instances require fewer vehicles in general to serve all customers compared to the first group. Clustering is not as effective in these instances as introducing more subproblems often also requires more vehicles. This correlation can be seen after 4 clusters in figure 6.6 as the required number of vehicles starts to increase again. However, in dynamic environments, these instance sets are less applicable because vehicles have smaller capacities and stricter time windows.

Effect of Parallelization

A likely reason why clustering is performing better than the unclustered approach has to do with parallelization. The clustered problems are solved in parallel and are thus able to use more computational power within the same period. This hypothesis is tested by scaling the available runtime of the instances to the number of clusters such that each request effectively receives the same amount of runtime. Figure 6.7 shows the results of the Li & Lim benchmark where the runtime is scaled to the number of clusters, i.e. an instance clustered into 8 subproblems gets 1/8 of the runtime. In the tests the single cluster problem receives 10 seconds of runtime which is slowly reduced to 1.25 seconds for the 8 clusters. The clustering is capped at eight clusters as the runs are performed on an 8 core machine.

The clustered instances still outperform the single clustered ones meaning the reduction of the problem size helps to lower the cost of a solution. The customers in the smaller problems can be matched more effectively in the short time frame, and clustering still provides better solutions. The influence of clustering is less significant than when parallelization is also utilized (Figure 6.6). However, the total distance is still reduced substantially compared to the unclustered instances. The amount of suboptimality introduced by the clusters seems negligible as the algorithm is still able to find decent solutions in shorter runtimes. These results show that clustering can be a very effective method for solving large-scale instances in real-time.

Comparison to Existing Methods

Not much literature has been done on finding solutions for static instances in short runtimes. To get an impression of the quality of our results, we compare them to other fast approaches or construction methods (Table 6.4). The last entry Best Known Solution (BKS) shows a comparison to best results found in the literature. The entry of proposed algorithm presents the results of this thesis. The table shows that especially in reducing cost our method is more effective than comparable methods.

Source	**Average Number of Vehicles**	**Average Total Distance**	**Runtime (s)**
Hosney (SEQ) [43]	77.25	108513.19	1.88
Sartori [44]	55.86	61106.76	15.25
Proposed algorithm	56.86	45701.68	10
BKS*[48]	44.34	38482.56	-

Table 6.4: Comparison to other construction heuristics on the Li & Lim benchmark sets of a 1000 locations. (*Best Known Solution)

6.4. DYNAMIC RIDESHARING

In this section, the results of various simulations of New York taxi trip data are presented. The results of clustering, methods, and certain parameters are shown and discussed. When a parameter is not stated in these experiments, the default value as defined in the setup 5.2.1 is used.

6.4.1. PERFORMANCE INDICATORS

To measure the effectiveness of a taxi ridesharing system, two perspectives have to be considered; the side of the operator and the passenger's side. The primary objective for the operator is to minimize the operating costs in order to save fuel and driver costs and increase revenue. The passengers, on the other hand, value the price and quality of service (QoS) with as little delay as possible.

The following metrics have been chosen to validate the effectiveness of our method for both perspectives and are similar to related research [4, 49]. The first two metrics evaluate the operator costs and the effectiveness of ridesharing, while the last two metrics consider the QoS by quantifying the experienced delays of passengers:

- **Service Rate:** Percentage of the total number of passengers that are dropped off at their destination during the simulation. This rate shows the ability of the vehicle fleet to serve the customer demand.
- **Distance Rate:** This metric determines how efficiently the requests are served. The distance rate is calculated by taking the total direct trip distance of the passengers divided by the total travel distance of the vehicles. The direct passenger distance is taken to exclude detours and provide a metric that can be compared with regular taxi services.
  This rate is commonly represented by the total travel distance. However, due to the possibility of rejecting customers and a variance in the distance of trips, this metric provides a more stable and clear representation of the overall travel efficiency.
- **Waiting Time:** The timespan a passenger has to wait for a ride. This is measured by taking $t_p - e_p$ for a passenger p.
- **Detour Time:** The period that a passenger is delayed because of detours required for ridesharing. This value is calculated by subtracting the direct travel time from the actual travel time. For a passenger with origin o and destination d this is measured as follows: $t_d - t_o - t_{o,d}$.
- **Total Delay:** The total extra time required to service a customer. This includes the time waiting for the pickup as well as the detours taken during the trip. The total delay is equal to $t_d - e_o - t_{o,d}$, for an origin o and destination d.

6.4.2. SIMULATION RESULTS

In the static instances, the different methods used showed a large difference in solution quality. In table 6.5 the comparable dynamic results of these methods are shown. This table shows the overall effect of a full day simulation. The vehicles of the fleet have a maximum capacity of 10 passengers and the Ω and Δ are 7 and 14 minutes respectively. The larger capacity and time windows make the problem harder to solve and provide a better picture of the scalability of our approach. The importance of scalability is also the reason for the smaller runtimes. The method field shows which heuristics are used and the runtime shows the time limit for a 10 second batch of requests.

While the service rate is comparable between the different methods, the distance traveled to serve the requests differs considerably. The proposed method saves 15% in distance and 100 seconds in total delay

Methods	Runtime (s)	Service Rate	Mean Waiting (s)	Mean Detour (s)	Distance Rate
Construction (Fast)	0.05	98.58%	175	378	3.14
Construction	0.25	97.57%	304	266	4.3
VNS	0.5	98.08%	256	257	4.26
VNS & Clustering	0.5	98.90%	241	244	4.68
VNS	3	98.77%	255	242	4.71
VNS & Clustering	3	98.86%	243	221	4.92

Table 6.5: Results for the variations in the main algorithm in a day simulation using a vehicle fleet of 2000 with capacity 10. The runtime is the average time used during a planning iteration for a 10-second interval of simulation time

Problem	Fleet Size	Capacity	Service Rate	Waiting (s)	Detour (s)	Distance Rate
Normal	2000	4	96.69%	187	133	2.63
Normal	2000	10	98.04%	200	116	4.29
Triple	6000	4	98.29%	179	125	2.79
Triple	6000	10	98.70%	175	161	4.85

Table 6.6: Comparison simulation results for different capacities and customer demand

compared to using construction. The effectiveness of clustering can also be seen in the very short runtimes. The distance rate of using half a second of runtime with clustering is comparable to a non-clustered approach using 3 seconds of runtime. These results are in line with can be seen in figure 6.3a. After 1 second the clustering the benefit is decreased on similar size instances. However, when the problem is scaled-up in size (Figure 6.3b) or when the runtime is decreased, the benefit of clustering becomes apparent.

6.4.3. Ridesharing at Increased Demand

To validate the scalability of our approach, we increased the number of requests in the simulation to three times the usual level of demand. The data is a combination of all the requests from three consecutive days and placed at their requested time. This increases the number of passengers on a day to around 1.3 million.

During the simulations, we use the standard values from the trip specification, found in section 5.2.1. As the amount of requests has tripled, we will also increase the increase the fleet size to 6000 vehicles to be able to serve a similar number of passengers. However, the density of the requests has tripled and thus trips are more likely to have more ridesharing matches. Hence, an increase in service rate and distance rate is to be expected when our approach can handle an increased number of requests effectively.

Table 6.6 shows the results for the increased demand. When the results of the triple demand are compared to the usual situation, the service rate has increased and waiting times, travel delays and amount of ridesharing all have improved as well. At a capacity of 10 at triple demand, the travel cost has decreased by 13% compared to the normal demand when looking at the distance rate as well as serving relatively more passengers. Furthermore, better matches can be found when the capacity or demand is higher, and thus creating a more efficient solution overall.

The improvements show the scalability of the algorithm as it is able to handle the elevated demand adequately and capable of improving ridesharing allocations in situations with a large request density.

6.4.4. Degree of Ridesharing

The distribution of passengers over the vehicle fleet provides more insight into the degree of ridesharing and how many trips can effectively be combined. The following section contains an overview of the status of the vehicles during the day and how many passengers are sharing rides. Figure 6.8 shows the state of the vehicles and the number of passengers they are carrying. At the start of the day between 0:00 and 6:00 the fleet is too large for the number of requests and most vehicles are idle. The amount of ridesharing is still quite high as even deep into the night, between 3 and 5 A.M., the active vehicles are still carrying 2.85 passengers on average.

During the day the amount of request increases and the capacity of the whole fleet is required to serve the entire demand. After 7 A.M., around 50% of the fleet is operating at their full capacity of 4 passengers. In figure 6.9 the same simulation is performed with larger capacity vehicles that can take up to 10 passengers.

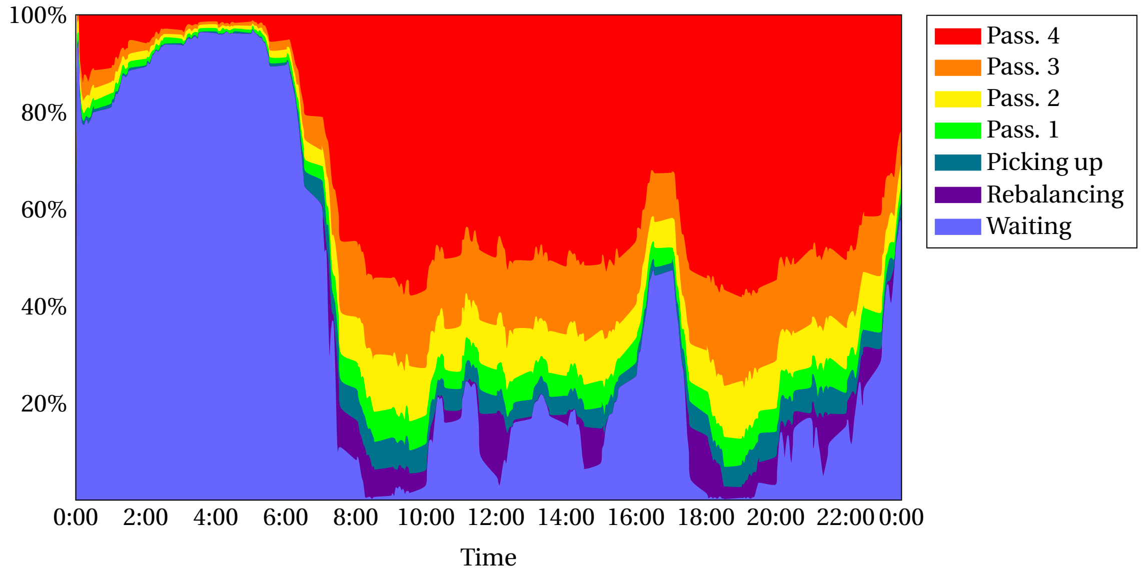


Figure 6.8: Distribution fleet with 2000 vehicles of capacity 4

The total capacity of the fleet has increased, and this leads to an increase in the number of idle vehicles. The fact that more vehicles can stay idle means that the degree of ridesharing has increased as the fleet size has remained equal in size. This conclusion can also be drawn from the results since the vehicles covered 34% less distance to serve the same number of requests. The figure also shows that the number of rides that can be combined is very high as during the day 10% of the fleet is transporting ten passengers.

6.5. Dynamic Ridesharing Experiments

As many variables influence the results of dynamic ridesharing, this section wants to show the results of varying results of parameters that are considered less important for solving static ridesharing problems. These consist of dynamic fleet size, rebalancing, various constraints on passengers and more. The results show how the algorithm performs under different circumstances and which parameters have a major influence on dynamic ridesharing.

6.5.1. Day Simulations

A general representation of the results for varying vehicles fleet and time windows can be seen in table 6.7. The runs were performed using rebalancing.

The results in the table provide an overview of the effect of various constraints and fleet sizes on the ability to service the requests of Manhattan. The value that offers the best impression of the possible ridesharing is the distance rate. This value shows how efficient passengers are serviced with ridesharing. The baseline for this value is not equal to 1 at single capacity vehicles as vehicles have to cover distance between a destination and the next pickup. This value for a fleet of 2000 vehicles with capacity 10 and $\Omega = 420$ is six times larger than using single capacity vehicles, and thus on average only 1/6th of the distance has to be covered to serve a single passenger.

The results are similar to those of Alonso-Mora et. al. [4], although slightly better in service rate. However, as a different distance model is applied, these results are not fully comparable but establishes an indication of the effectiveness of the two ridesharing approaches. The paper does not provide any direct measures of their travel efficiency, however, this value can be estimated from their data. The average trip distance is 4.2 km [42] and in a week approximately 3 million taxi trips are booked in Manhattan. The total distance than amounts to $4.2km \cdot 3,000,000 = 12,600,000km$. For a 3000 vehicle fleet of capacity 4 and a 7 minute pickup window, they report a service rate of 98% and an average of 274 km driven per vehicle per day. The total distance that the vehicles have covered over the week is about: $274km \cdot 7 \cdot 3000 = 5,754,000km$. With this total distance driven, the distance rate of their approach can be approximated to be around: $12,600,000/5,754,000 \approx 2.19$.

The results in table 6.7 report a service rate of 99% and a distance rate of 2.74. This rate shows a substantial

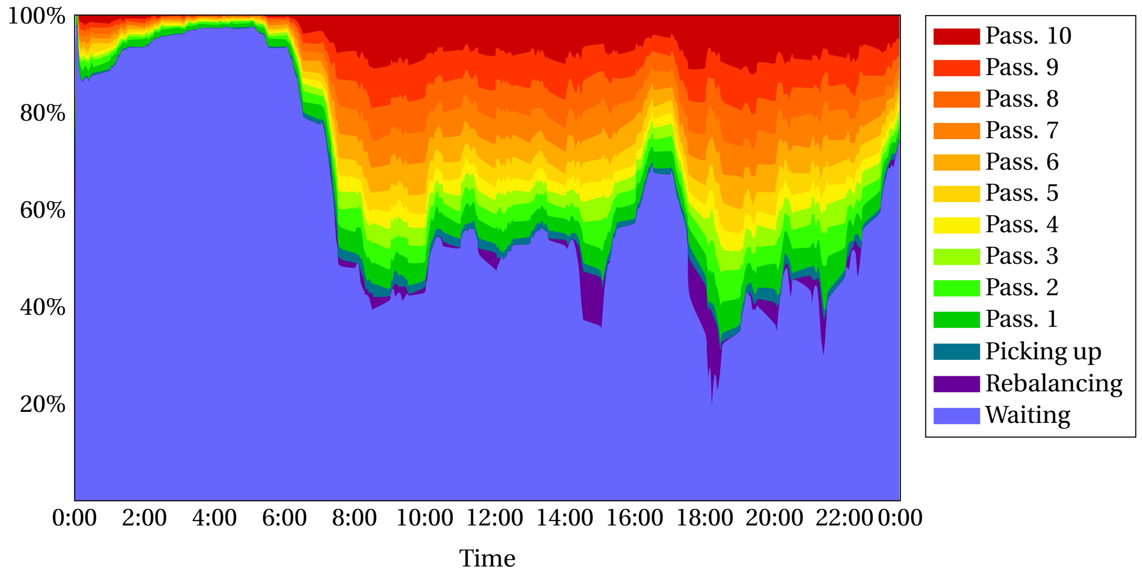
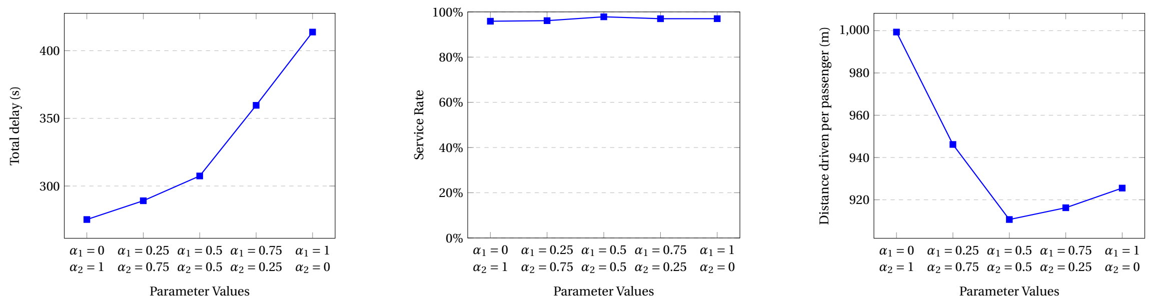


Figure 6.9: Distribution fleet with 2000 vehicles of capacity 10

Fleet size	**Capacity**	**Ω (s)**	**Δ (s)**	**Service Rate**	**Mean Waiting (s)**	**Mean Detour (s)**	**Distance Rate**
1000	1	300	600	26.13%	252	0	0.86
1000	4	120	240	48.59%	77	56	2.04
1000	4	300	600	54.42%	216	130	2.39
1000	4 no cl.*	300	600	53.52%	224	135	2.25
1000	10	420	840	84.33%	290	271	3.94
2000	1	300	600	53.20%	230	0	0.84
2000	4	120	240	83.80%	73	56	2.23
2000	4 no cl.*	300	600	96.08%	194	136	2.63
2000	4	300	600	**96.69%**	187	133	2.63
2000	4	420	840	97.16%	290	162	2.67
2000	10	120	240	92.54%	70	76	3.23
2000	10	300	600	98.14%	200	116	4.29
2000	10	420	840	99.14%	252	250	4.74
2000	10 no cl.*	420	840	98.74%	270	254	4.50
3000	4	120	240	91.09%	73	54	2.26
3000	4	300	600	**97.07%**	201	118	2.68
3000	4	420	840	99.30%	252	148	2.72
3000	10	120	240	95.75%	68.81	75.59	3.23
3000	10	300	600	99.39%	171	198	4.46

Table 6.7: Results from single day simulations with rebalancing. Ω shows the maximum waiting time and Δ shows the maximum drop-off delay. *Run without clustering



(a) Average total delay of a request (b) Percentage of passengers served (c) Mean distance to serve a passenger

Figure 6.10: Effect of objective function on ridesharing (α_1 = Travel time, α_2 = Delay)

decrease as the vehicles of our simulation require 25% less travel distance. However, while this is a substantial reduction, the differences between both simulation methods do not allow us to state that our approach is capable of these improvements in practice and further research is required.

6.5.2. Objective Function

The primary objectives set in section 2.3.1 are to maximize the service rate while minimizing the total travel distance. In this section, we also want to focus on another objective parameter that considers the delay that the customers experience. Minimizing the delay of passengers can provide two advantages. First, it is more convenient for customers as they will arrive faster at their destination. The second advantage is that it can increase the service rate when passengers are served in faster succession.

The two parameters that are considered in this section are α_1 and α_2 (Section 2.3.1). The parameters α_1 and α_2 give the weight of two cost variables; α_1 tries to minimize the total travel distance and α_2 tries to minimize the delay for each request. The parameters hold the following constraint $\alpha_1 + \alpha_2 = 1$ where α_1 starts at 0 and is increased to 1 during the simulations. This way we can approximate the full range of values between considering either one of the parameters in the objective function.

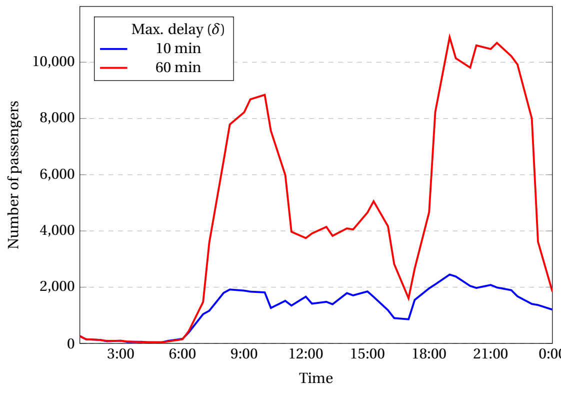
In the objective function two other parameters are considered, α_3 and α_4. Parameter α_3 gives a base cost to using a vehicle. Since a constant fleet size is used during the simulations, a base cost is not necessary and α_3 is set to 0.

Our primary focus is still to reject as few customers as possible and thus a large constant penalty is used as value for α_4. The penalty cost is set to a value larger than the service cost of a request which means the request will only be rejected when no vehicle is available to provide a ride for the request. The results were obtained from a 3-hour simulation during the evening traffic of Manhattan using the default trip setup in section 5.2.1.

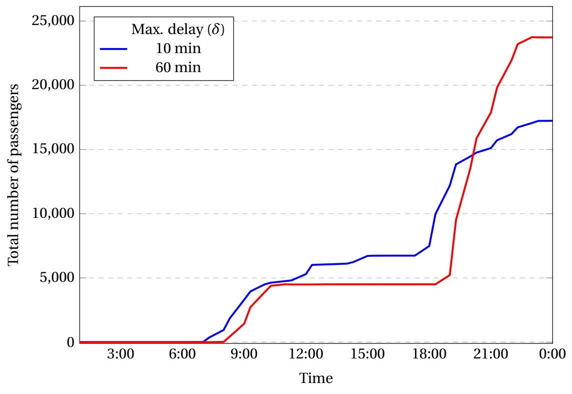
The results are shown in figure 6.10. As expected the average delay for passengers increases as the weight of the corresponding parameter α_2 decreases (Figure 6.10a). However, increasing the delay does not directly lead to an overall cheaper solution. As seen in graph 6.10c the average distance per request, which means the least total travel distance, is best at equal distribution of both parameters. Ridesharing results can thus improve when the delay is also taken into account.

The reason that the travel cost decreases when delay is taken into account is likely because the total capacity of the fleet is also a dominant factor during planning. Optimizing solely for minimizing the total travel distance results in a worse overall trip plan for new requests. Incoming requests have fewer spots in the active vehicles and have to be placed in less optimal positions. A similar result can also be seen when increasing the bounds of the time windows of customers where having flexible windows lead to worse solutions (Figure 6.11).

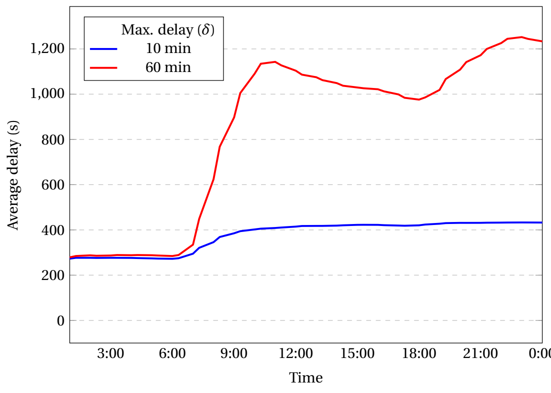
Varying the objective function during the day might be the best solution if the goal is primarily to minimize the expenses of the vehicles. When the fleet is large compared to the number of requests, it is best to minimize total travel time, while as the number of requests increases dropping off passengers in faster succession is more profitable. Considering delay as a factor at all times can also be regarded as the best option as less delay can lead to an increase in customer satisfaction.



(a) Number of passengers waiting for pickup



(b) Total amount of rejected passengers



(c) Average total delay

Figure 6.11: Influence of the maximum total delay

6.5.3. TIME WINDOWS

The current approach works with strict time windows when deciding whether to place or reject requests. This constraint leads to rejections when no vehicles are nearby while they could be served with a slightly longer wait. Hence to raise the service rate, it might seem logical to extend the time windows to always have a vehicle in range to serve the customer.

A test where to different hard bounds are used is shown in figure 6.11. In these figures, a maximum total trip delay, δ, of 10 minutes and 60 minutes is set for the request. So for a pickup p and dropoff d holds: $l_d = e_p + t_{p,d} + \delta$. In the first few hours, the simulations create similar results and longer time windows also provide little improvement in the possible placements of requests. Subfigure 6.11b shows that the point where the vehicles have to refuse passengers starts an hour later with the extended time windows. However, this directly correlates with a spike in waiting times and trip delay (Figure 6.11c). The postponed requests cause the number of rejections to rise very rapidly as the evening traffic hits at 18:00. The schedule of the vehicles is too full, and the number of passengers that are waiting for a pickup is five times as high as in comparison to stricter time windows. This effect is caused by the inefficient placement of difficult requests. As seen in the figures 6.11a and 6.11c, the amount of waiting passengers and the average delay is three times higher for $\delta = 60$ compared to the $\delta = 10$. Requests are placed in incompatible vehicles to try to serve all passengers, and this causes all incoming requests to end up in a queue, waiting for their pickup.

When deploying a larger fleet, it might be advantageous to allow requests to have relaxed time windows. However, in a situation where many rides can be shared, an extension of time windows does rarely improve the solution. This conclusion is also in line with Santi et al. [3]. They state that at 100,000 trips, or 25% of the current traffic in Manhattan should lead to near-maximum ridesharing possibilities. Relaxing constraints does not lead to improvements in areas containing a high request density.

Fleet Size	2500	1500
Capacity	4	10
Ω & Δ (min)	5 & 10	5 & 10
Delayed Requests	3.62%	4.61%
Mean Waiting Time* (s)	545	538
Mean Detour Time* (s)	150	230
Mean Waiting Time (s)	201	213
Mean Detour Time (s)	145	218
Distance Rate	2.64	4.29
Mean Active Vehicles	1221	740

Table 6.8: Results for extending time windows to obtain 100% service rate. *Data for delayed requests that could not be served within the maximum pickup delay (Ω).

6.5.4. Complete Service of Demand

The service rate of all results in table 6.7 is below 100%. Some requests are very difficult to service, and even when a large fleet with high capacity vehicles is deployed, the simulation is not able to service all requests within the constraints (Table 6.7). Some customers are positioned at distant locations and are very hard to service within several minutes. It would be interesting to see whether it is possible to serve all requests in Manhattan and how many vehicles that would require.

Currently, there are 13,000 registered yellow taxis, and it would be interesting to see how many of those would be redundant when ridesharing is applied. Since increasing the windows for all requests will only increase the size of the problem as well as decrease its ability to deal with future requests (Figure 6.11), we want to use to strict waiting times in general. Some of the requests are hard to combine or are placed in locations where the vehicles are sparse. Hence, we require a method that solves all other requests efficiently while also dealing with these harder requests. A solution to this is to relax the constraints on the maximum waiting time for difficult requests. Every time a request is rejected, its maximum waiting time (Ω) and maximum delay time (Δ) are extended by 5 minutes which makes it possible to send a vehicle further to pickup the request. The idea of continuously extending this window is to keep the time window constraint as short as possible while also making sure that at some point it can be placed into a trip as their pickup window keeps increasing. Key points for this experiment are to look whether it is possible to serve all requests and if it is, how many requests are affected and how much extra delay is introduced.

Table 6.7 shows that at capacity 4 with $\Omega = 300, \Delta = 600$ the service rate only improves from 96.69% to 97.07% when increasing the fleet size from 2000 to 3000 vehicles. As over 2000 vehicles are required to serve peak moments in Manhattan (Figure 6.8), 500 vehicles are added to be able handle the full customer demand of Manhattan.

Table 6.8 shows the results for simulations with time window extensions. The fleet of 2500 vehicles can serve the whole demand within reasonable waiting times and delays. Less than 5% of the requests cannot directly be placed on a trip, and the mean waiting for all the requests has remained comparable to previously found results (Table 6.7). The same number of rides are shared, and the average distance driven per passenger is on the same level.

The active fleet, containing vehicles that are serving passengers, consists of 1221 driving vehicles on average and require the same number of working hours. This number is only 16% of the number of working hours of service currently necessary which is around 8000 [42]. Based on these results, it might be beneficial to allow customers to extend their time windows. The general waiting time for these requests does not create a spike in waiting times and are on average only picked-up 4 minutes after their original pickup constraint. The detour time of the delayed requests is barely higher compared to the overall mean detour time. Furthermore, the average total delay just exceeds 2 minutes above their original bound of 10 minutes.

6.6. Summary of results

As a follow up to the validation steps described at the start of this chapter, this section show a short overview of the general results. The 4 steps are shortly discussed with the key values.

Approach: On benchmark instances the approach is able to reach within 10% and 30% of the best known solutions on small and large instances. Clustering increased the performance on the benchmark of the algorithm. On average, the number of vehicles was reduced by 11% and the cost by 22%. On Manhattan trip data an improvement of 8% in cost was shown when a runtime of 5 seconds was used.

Scalability: The scalability improved drastically by introducing clustering. The instances containing triple the usual demand in Manhattan improved by 38% when clustering was applied. Clustered instances with a runtime of 1 second were even able to outperform the unclustered problem with a runtime of 15 seconds in large instances of 4200 requests.

Taxi simulation: Using a fleet of 2000 vehicles the methods were able to achieve 97% and 98.5% service rate using vehicles with a capacity of 4 and 10 respectively. Using capacity 4 vehicles the approach was able to reduce the required travel cost by a factor 3 and the mini-buses with a capacity 10 were able to reduce the cost by 5 times compared to single capacity vehicle simulations.

Dynamic aspects: Incorporating delay turned out be an important factor in reducing the overall cost as well as the overall delay for the passengers. When trying to optimize for the delay alongside the overall cost in the objective function, a reduction of 35% in passenger delay as well as 2% reduction in the total driven distance. We also showed that achieving a 100% service rate seems very viable in Manhattan using a fleet of 2500 vehicles. The total delay is on average just below 6 minutes and less than 5% of requests will have to wait more than 5 minutes.

The results are based on optimistic simulations that require all parties involved to comply to all allocated routes produced by the algorithm. As drivers can get overloaded with feedback when their routing plan updates every 10 seconds, it could be hard to reach the predicted efficiency in practice. However, the results for an on-demand taxi ridesharing system are very promising and could be become a popular effective transportation service in the near future.

7
Related work

This chapter covers work in the field that is applicable to this research. The research in this chapter was considered in this thesis but could not be applied to our problem. The chapter is split into three sections, discussing other scalability approaches, real-time ridesharing methods, and related transportation solutions.

7.1. Parallel and Distributed Solutions

The following papers also looked at parallel implementations that looked at dealing with scalability issues and tackling substantial problems more effectively.

In Kalina et al. [50] static instances of the VRPTW and PDPTW are solved by multiple agents. The vehicles are represented by an agent. These agents negotiate over the requests in the problems and can trade or pass requests when another agent can provide a better position. The vehicle agents have three main moves that either moves the worst request, move all its requests to another agent or move all its requests to multiple agents. The system uses a runtime of 4 minutes for PDPTW instances of 500 requests, but this is equal to over 300 minutes in total computing time. The solutions found for these instances contain on average 15% more vehicles compared to the best-known solutions. The concept of agents is original for routing problems, however, is very expensive in computational time and is too slow for practical use in real-time applications.

A solution of using the mainstream use of smartphones in the current society is proposed by d'Orey et al. [29]. They propose a distributed algorithm in which passengers are in control of the selection of their routes. When someone is looking for immediate transportation, they send out a request that is distributed to a few nearby taxis. The taxis than each calculate what the request would cost, with how many passengers the ride is shared and when it will arrive. This leaves the passenger with an overview of possible trips to choose from and can pick the ride that suits them best. While this idea is interesting and offers a high quality of services, it does not take into account all requests at once. The solutions found will be based on human decisions and are obtained decentralized. These factors will likely have a negative impact on the overall quality of the operation. This approach will thus not be used as this thesis is focused at finding more efficient solutions for all the clients involved in the system.

7.2. Large-Scale Dynamic Taxi Ridesharing Solutions

This section contains works from similar research to this thesis. The solutions of these papers looked at ridesharing as an efficient alternative in large urban areas.

Ma et al. [30] was one of the first literature papers in creating a service for large-scale ridesharing called T-Share. Their primary focus was to decrease computation cost for calculating routes for requests. By dividing the area of a city into a grid, they are able to calculate viable trips for incoming requests quickly. By performing feasibility checks, many possible combinations are eliminated and with their method, are able to handle many requests in real-time. The placement of a request takes around 5 ms, and hence the approach can handle up to 720,000 requests per hour. They showed that with their approach, 25% more customers can be served in Beijing while also reducing the total distance by 13%.

A look at the general benefits of ridesharing in larger cities is presented in Santi et al. [3]. They looked at the tradeoff between the discomfort of individual passengers and collective gains of ridesharing. Their research is based on taxi trips in New York and found that using ridesharing it would be possible to cut at least 40% of the distance currently driven by NYC taxis every day.

Ota et al. [51] looked into scalable solutions for large-scale ridesharing simulations. They created methods for fast shortest path calculations, insertion and large simulations. Taking advantage of parallelization, they created a method for running days simultaneously for long-term simulations which results in a large speedup in the required simulation time. This approach allowed to them simulate a year of taxi data in Manhattan, 150 million requests, within 10 minutes using 1200 cores. Their results showed a reduction of 30% in the required time needed to serve all requests when allowing up to 3 passengers to share a ride at any time.

The main approach to create this speedup was not applicable to this thesis as their method did not focus on scaling real-time requests and is mostly useful for large simulations over a long period.

7.3. VERY LARGE-SCALE ROUTING PROBLEMS

The need for a solution has come up in the last decade as technology is better equipped at handling huge volumes of data-points. Most literature has focused on instances of up to a few hundred locations. Large-scale problems often consider up to a 1000 locations and this number is usually the upper limit of the scale regarded in most research. However, several optimization problems are much larger and contain over 10,000 nodes. The paper of Kytojoki et al. [26] was the one of the first that looked at instances of this scale. Their research aims at solving large groups of customers with known demands such as waste collection and mail and newspaper delivery. They examined problems of up to 20,000 customers which could be solved within a few hours.

Memory management becomes a growing issue at this scale. Distance matrices are common practice for fast shortest path calculations. However, distance matrices of 20,000 locations that only store time and distance can already need several gigabytes of RAM-memory besides the storage of preprocessing data. To reduce the size of such a matrix, they used a technique that expected less precision and combined time and distance. This technique led to a reduction of ten times less memory compared to the predetermined size while maintaining sufficient accuracy.

In Arnold et al. [52] extended on Kytojoki in the search for efficient solutions for these very large-scale instances. They created a new technique using knowledge-based heuristics. They focused on matching the most promising combinations to reduce computations and memory. They chose a subset of α neighboring nodes for a local move which reduces the complexity from $O(n^2)$ to $\alpha * n \in O(n)$. Another improvement was to direct the search to the most 'bad' locations in the solution space. They weighted the edges with a 'badness' index and tried to focus on solving these first. Compared to Kytojoki, their results were 3% better on average and required less computational time.
Both these solutions provided solutions for tackling problems that are introduced by the expanding size of routing problems. Their methods are built for the static VRP and not directly applicable to this research. However, the proposed search strategies and memory management could be beneficial in large-scale dynamic ridesharing applications as well.

7.4. TAXI FLEET OPTIMIZATION

In a recent article by Vazifeh et al. [8] the minimum vehicle fleet problem is discussed in Manhattan. In contrast to this research, their application does not look at ridesharing but on other factors that can be optimized in taxi services. They used a path cover approach to efficiently calculate the minimum number of vehicles required to serve the total trip demand. The proposed method can reduce the vehicle fleet by 30% compared to the current situation. The idea is that if the fleet is operated from a central operation the service in New York, the number of necessary vehicles can be reduced significantly. Their research showed the strength of a centralized solution in these systems and that this can already provide a large optimization in the current system. Ridesharing even allows for a more effective transportation solution and looking at factors as vehicle deployment could also provide a large benefit in future ridesharing research.

8

Conclusion & Future Work

In this chapter, the conclusions of this thesis are summarized, and the research questions that are proposed in the introduction are answered and discussed. In the last section, the limitations of this research are discussed as well as potential improvements in future research.

Conclusion

In this research, we have looked at a scalable solution for a wide range of problems and problem sizes. We found that an efficient decomposition of the problem can lead to a reduction of the objective cost when solved in limited time. The introduction of subproblems creates an upper bound for the size of a problem and allows for very large-scale instances of over 5,000 requests in real-time on a single machine. This section discusses our findings on providing a scalable algorithm for the dynamic ridesharing problem.

Scalability

At the start of this thesis, the following research question was phrased:

> *"Is it possible to create a scalable algorithm that has to complete within a limited time and is able to create effective allocations for on-demand ridesharing passengers?"*

This report has shown that decompositioning of the problem improves the scalability as well as the routing of the vehicles. By applying k-means clustering to the ridesharing problem, we were able to bound the size of each subproblem to an effective limit. Clustering provided the ability to down-scale each problem into subproblems that can be distributed and ensures solvability within seconds.

The pickup location of a request is chosen as the predominant factor in finding suitable dynamic ridesharing matches between vehicles and other passengers. Thus, decomposition of the ridesharing on pickup locations allowed for an effective split. The clustered instances were able to find better solutions in the same runtime. The results showed that within a runtime of 10 seconds, clustered solutions produced solutions which are on average 30% lower in cost and this performance increases when the problem expands in size.

The variable neighborhood search approach combines a fast and extensive approach that can be applied to a wide range of problems. The results in figure 6.1 show the fast reduction in cost that local search can achieve and the slower but effective way of perturbing this solution for smaller problems. As shown in 6.3.3 the loss in suboptimality is very small compared to the speed up the method provides in solving large-scale problems. Our approach is able to outperform itself using the proposed clustering technique. The results showed that for well known Li & Lim static instances, clustering also improved the solutions found in a short time frame. The solutions for the largest number of locations in this set that are found within 10 seconds required 30% more vehicles and 20% more cost compared to the best-known solutions.

Ridesharing Results

The results found in this study are in line with other large ridesharing papers. In comparison to Alonso-Mora [4] our results were very similar and were able to show that ridesharing using a fleet of 2000-3000 vehicles with a capacity of 4 passengers would be sufficient to serve all taxi requests in Manhattan. This number is only a fifth of the current taxi fleet but would require a fully centralized solution and the full cooperation of all passengers.

When applied to dynamic instances, our proposed method was able to place up to 1500 requests within a few percent of the best-found solution within several seconds. The experiments in this thesis also showed interesting results and implications for various constraints in dynamic ridesharing. We were able to serve every request in Manhattan by only relaxing strict short waiting for 4% of the passengers, and this would allow a fleet of 2500 vehicles to be able to serve 450,000 requests a day.

While our approach is far from ready to be directly introduced into current taxi systems, the results of this research show the possible improvements and efficiency that can be achieved using ridesharing. While further research is necessary to verify these findings in practice, real-time ridesharing in urban areas can be a profitable alternate solution to personalized transportation.

Discussion & Future Work

The research done in this thesis has its limitations that could be improved on. This section discusses several topics that this research might have missed or would like to address in future research.

Clustering Approaches

The effects of clustering are shown to be effective in real-time applications. An area-based clustering technique focused on pickup locations is represented, but other cluster techniques might produce more favorable decompositions. Taking into account time windows, delivery locations or might help improve finding better solutions or lower the required runtime.

The current approach uses Euclidean distance to determine the distance between locations, which is common in k-means. However, the actual distance between two coordinates can deviate considerably due to factors as obstacles between the locations, such as parks, or constraints on the roads. A more accurate approach for calculating distances can exclude barriers within clusters and improve the relatedness between requests in the same cluster.

Dynamic Ridesharing Challenges

Much research has been focused on solving static instances; however, factors as rebalancing and anticipating future requests can benefit the solution quality. The demand fluctuates throughout the day, and the position and status of the available fleet have a notable effect on the amount of effective ridesharing. The results show that all requests can be serviced with 2500 vehicles and on average only requires 1221 on-duty vehicles to achieve this. Therefore, incorporating improved rebalancing and fleet resizing should be able to reduce the number of dispatched vehicles at various hours substantially.

Gap with reality

Many factors in our simulations have been relaxed. Relaxations that were applied in the simulations are assuming 24-hour shifts without any breaks, simulations in Euclidean space, the full cooperation of drivers and passengers, and no vehicle breakdowns or other unforeseen delays. Hence, results found in this thesis are likely to be optimistic and in practice will require more vehicles that have less efficient allocations. A future challenge will be to tackle these problems to increase accuracy for real-time planning in practice.

This research was mainly focused on providing very fast results for an improved customer experience. However, the rate of adjusting planned routes every 5 to 10 seconds can be too fast to keep up with for drivers or confusing for the passenger when their expected vehicle changes often. This report does not discuss the effects of this high turnover, but should likely be taken into account when deploying this service in practice. At the expense of a possible increase in travel costs, keeping established matches between drivers and passengers fixed for a certain period might reduce the impact of this problem.

For future research, the distance model is a large inaccuracy that should be changed for use in practice. The current approach assumes too many generalizations to estimate the travel time and should be revisited for accurate predictions in actual situations. The model should be provided with an underlying road model and use actual traffic data. This way the system can incorporate current events and traffic into the travel time estimations, which allows the system to make accurate predictions that make it more reliable and effective overall.

Distributed Implementation

This thesis was originally focused around creating an algorithm that could be distributed over multiple servers to speed up the process. The importance of a distributed implementation has turned out to be lower than expected as problems the size of New York could effectively be handled on a single machine. It would be interesting for future research to verify whether an improvement on the current results can be obtained with multiple servers. For the calculation of larger ridesharing problems, a higher number of threads would have been helpful in solving as the number of clusters often exceeded the available threads. The current implementation works with shared memory, and this might be a challenge when implementing a fully distributed solution that is still able to guarantee real-time responses.

BIBLIOGRAPHY

[1] M. Furuhata, M. Dessouky, F. Ordóñez, M.-E. Brunet, X. Wang, and S. Koenig, *Ridesharing: The state-of-the-art and future directions,* Transportation Research Part B: Methodological **57**, 28 (2013).

[2] N. Agatz, A. Erera, M. Savelsbergh, and X. Wang, *Optimization for dynamic ride-sharing: A review,* European Journal of Operational Research **223**, 295 (2012).

[3] P. Santi, G. Resta, M. Szell, S. Sobolevsky, S. H. Strogatz, and C. Ratti, *Quantifying the benefits of vehicle pooling with shareability networks,* Proceedings of the National Academy of Sciences **111**, 13290 (2014).

[4] J. Alonso-Mora, S. Samaranayake, A. Wallar, E. Frazzoli, and D. Rus, *On-demand high-capacity ride-sharing via dynamic trip-vehicle assignment,* Proceedings of the National Academy of Sciences , 201611675 (2017).

[5] D. Schrank, B. Eisele, T. Lomax, and J. Bak, *2015 urban mobility scorecard,* (2015).

[6] M. Parry, O. Canziani, J. Palutikof, P. J. van der Linden, C. E. Hanson, *et al.*, *Climate change 2007: impacts, adaptation and vulnerability,* Vol. 4 (Cambridge University Press Cambridge, 2007).

[7] M. Barth and K. Boriboonsomsin, *Real-world carbon dioxide impacts of traffic congestion,* Transportation Research Record: Journal of the Transportation Research Board , 163 (2008).

[8] M. M. Vazifeh, P. Santi, G. Resta, S. H. Strogatz, and C. Ratti, *Addressing the minimum fleet problem in on-demand urban mobility,* Nature (2018).

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

[10] S. N. Kumar and R. Panneerselvam, *A survey on the vehicle routing problem and its variants,* Intelligent Information Management **4**, 66 (2012).

[11] M. W. Savelsbergh and M. Sol, *The general pickup and delivery problem,* Transportation science **29**, 17 (1995).

[12] H. Li and A. Lim, *A metaheuristic for the pickup and delivery problem with time windows,* International Journal on Artificial Intelligence Tools **12**, 173 (2003).

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

[14] G. Berbeglia, J.-F. Cordeau, and G. Laporte, *Dynamic pickup and delivery problems,* European journal of operational research **202**, 8 (2010).

[15] P. van Hentenryck and R. Bent, *Online stochastic combinatorial optimization* (The MIT Press, 2009).

[16] R. Baldacci, A. Mingozzi, and R. Roberti, *Recent exact algorithms for solving the vehicle routing problem under capacity and time window constraints,* European Journal of Operational Research **218**, 1 (2012).

[17] G. Clarke and J. W. Wright, *Scheduling of vehicles from a central depot to a number of delivery points,* Operations research **12**, 568 (1964).

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

[19] J.-J. Jaw, A. R. Odoni, H. N. Psaraftis, and N. H. Wilson, *A heuristic algorithm for the multi-vehicle advance request dial-a-ride problem with time windows,* Transportation Research Part B: Methodological **20**, 243 (1986).

[20] B. Funke, T. Grünert, and S. Irnich, *Local search for vehicle routing and scheduling problems: Review and conceptual integration,* Journal of Heuristics **11**, 267 (2005).

[21] F. Glover, *Future paths for integer programming and links to artificial intelligence,* Computers & operations research **13**, 533 (1986).

[22] D. E. Goldberg, *Genetic Algorithms in Search, Optimization and Machine Learning*, 1st ed. (Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1989).

[23] P. Shaw, *Using constraint programming and local search methods to solve vehicle routing problems,* in *International conference on principles and practice of constraint programming* (Springer, 1998) pp. 417–431.

[24] S. Ropke and D. Pisinger, *An adaptive large neighborhood search heuristic for the pickup and delivery problem with time windows,* Transportation science **40**, 455 (2006).

[25] N. Mladenović and P. Hansen, *Variable neighborhood search,* Computers & operations research **24**, 1097 (1997).

[26] J. Kytöjoki, T. Nuortio, O. Bräysy, and M. Gendreau, *An efficient variable neighborhood search heuristic for very large scale vehicle routing problems,* Computers & operations research **34**, 2743 (2007).

[27] O. Bräysy, *A reactive variable neighborhood search for the vehicle-routing problem with time windows,* INFORMS Journal on Computing **15**, 347 (2003).

[28] M. Polacek, R. F. Hartl, K. Doerner, and M. Reimann, *A variable neighborhood search for the multi depot vehicle routing problem with time windows,* Journal of heuristics **10**, 613 (2004).

[29] P. M. d'Orey, R. Fernandes, and M. Ferreira, *Empirical evaluation of a dynamic and distributed taxi-sharing system,* in *Intelligent Transportation Systems (ITSC), 2012 15th International IEEE Conference on* (IEEE, 2012) pp. 140–146.

[30] S. Ma, Y. Zheng, and O. Wolfson, *T-share: A large-scale dynamic taxi ridesharing service,* in *Data Engineering (ICDE), 2013 IEEE 29th International Conference on* (IEEE, 2013) pp. 410–421.

[31] T. G. Crainic, *Parallel meta-heuristic search*, Tech. Rep. (Publication CIRRELT-2015-42, Centre interuniversitaire de recherche sur les réseaux d'entreprise, la logistique et le transport, Université de Montréal, Montréal, QC, Canada, 2015).

[32] R. He, W. Xu, J. Sun, and B. Zu, *Balanced k-means algorithm for partitioning areas in large-scale vehicle routing problem,* in *Intelligent Information Technology Application, 2009. IITA 2009. Third International Symposium on*, Vol. 3 (IEEE, 2009) pp. 87–90.

[33] N. Mostafa and A. Eltawil, *Solving the heterogeneous capacitated vehicle routing problem using k-means clustering and valid inequalities,* Proceedings of the International Conference on Industrial Engineering and Operations Management (2017).

[34] R. Nallusamy, K. Duraiswamy, R. Dhanalaksmi, and P. Parthiban, *Optimization of multiple vehicle routing problems using approximation algorithms,* arXiv preprint arXiv:1001.4197 (2010).

[35] M. Schilde, K. F. Doerner, and R. F. Hartl, *Integrating stochastic time-dependent travel speed in solution methods for the dynamic dial-a-ride problem,* European journal of operational research **238**, 18 (2014).

[36] N. Marković, R. Nair, P. Schonfeld, E. Miller-Hooks, and M. Mohebbi, *Optimizing dial-a-ride services in maryland: benefits of computerized routing and scheduling,* Transportation Research Part C: Emerging Technologies **55**, 156 (2015).

[37] S. C. Ho, W. Szeto, Y.-H. Kuo, J. M. Leung, M. Petering, and T. W. Tou, *A survey of dial-a-ride problems: Literature review and recent developments,* Transportation Research Part B: Methodological (2018).

[38] Y. Luo and P. Schonfeld, *A rejected-reinsertion heuristic for the static dial-a-ride problem,* Transportation Research Part B: Methodological **41**, 736 (2007).

[39] T. Curtois, D. Landa-Silva, Y. Qu, and W. Laesanklang, *Large neighbourhood search with adaptive guided ejection search for the pickup and delivery problem with time windows,* EURO Journal on Transportation and Logistics , 1 (2017).

[40] P. Shaw, *A new local search algorithm providing high quality solutions to vehicle routing problems,* APES Group, Dept of Computer Science, University of Strathclyde, Glasgow, Scotland, UK (1997).

[41] *New york taxi data,* `http://www.nyc.gov/html/tlc/html/about/trip_record_data.shtml` (2013), accessed: 2017-10-27.

[42] M. R. Bloomberg and D. Yassky, *New york city taxicab factbook,* (2014).

[43] M. I. Hosny and C. L. Mumford, *Constructing initial solutions for the multiple vehicle pickup and delivery problem with time windows,* Journal of King Saud University-Computer and Information Sciences **24**, 59 (2012).

[44] C. S. Sartori, *Optimizing solutions for the pickup and delivery problem,* (2016).

[45] S. Ma, Y. Zheng, and O. Wolfson, *Real-time city-scale taxi ridesharing,* IEEE Transactions on Knowledge and Data Engineering **27**, 1782 (2015).

[46] GraphHopper, *https://github.com/graphhopper/graphhopper,* .

[47] J.-F. Cordeau, *A branch-and-cut algorithm for the dial-a-ride problem,* Operations Research **54**, 573 (2006).

[48] *Sintef; best known solutions li & lim,* `https://www.sintef.no/projectweb/top/pdptw/li-lim-benchmark/` (2008).

[49] P. M. d'Orey, R. Fernandes, and M. Ferreira, *Reducing the environmental impact of taxi operation: The taxi-sharing use case,* in *ITS Telecommunications (ITST), 2012 12th International Conference on* (IEEE, 2012) pp. 319–323.

[50] P. Kalina and J. Vokřínek, *Parallel solver for vehicle routing and pickup and delivery problems with time windows based on agent negotiation,* in *Systems, Man, and Cybernetics (SMC), 2012 IEEE International Conference on* (IEEE, 2012) pp. 1558–1563.

[51] M. Ota, H. Vo, C. Silva, and J. Freire, *A scalable approach for data-driven taxi ride-sharing simulation,* in *Big Data (Big Data), 2015 IEEE International Conference on* (IEEE, 2015) pp. 888–897.

[52] F. Arnold, M. Gendreau, and K. Sörensen, *Efficiently solving very large scale routing problems,* (2017).