Karlsruhe Institute of Technology

Ridesharing with Multiple Riders

Master Thesis of

Oliver Plate

At the Department of Informatics
Institute of Theoretical Informatics

Reviewers:	Prof. Dr. Dorothea Wagner
	Prof. Dr. Peter Sanders
Advisors:	Valentin Buchhold, M.Sc.
	Tobias Zündorf, M.Sc.

Time Period: 16th July 2018 – 15th Januar 2019

Statement of Authorship

I hereby declare that this document has been composed by myself and describes my own work, unless otherwise acknowledged in the text.

Karlsruhe, January 15, 2019

Abstract

Ridesharing is a far spread method for reducing traffic, pollution and minimizing the costs for driver and rider. Thus algorithms which can match drivers and riders are of high importance for ridesharing services and have been studied a long time.

In this thesis we provide a deep overview over related works. We present multiple classifications of ridesharing problems, summarize algorithms for the matching of drivers and riders and discuss approaches to optimize the quality of the solutions.

Many of these approaches rely on a "filter and refine" framework which reduces the problem size by trying to minimize the amount of drivers to be considered for a matching. These solutions perform well if the filtering is good, but else their bottleneck of calculating shortest distances one by one becomes dominating and increases the response times.

We introduce a solution for the single-driver multiple-rider ridesharing problem, which does not rely on the filtering of drivers. Instead we use a fast dynamic one-to-many (many-to-one) distance query algorithm which calculates all needed shortest distances within a fraction of the time single queries would need. We then can utilize the fact that we know all shortest distances between drivers and riders and thus can calculate the resulting detours for possible matches very fast.

Our approach can be easily adapted towards related ridesharing problems like the taxi ridesharing problem. To demonstrate this, we introduce a second algorithm, capable of scheduling taxis within a city following the same approach.

We show that our ridesharing solution achieves fast response times of under 100ms for up to 150'000 drivers within a three hour period. We compare our solution with an algorithm, which uses a filter and refine method, and point out the differences within these two approaches. Finally we run our taxi ridesharing solution with real world taxi data from New York City and can present a solution which only takes 2'101 taxis to fulfill each of the 400'000 requests that occurred within one day.

Deutsche Zusammenfassung

Mitfahrgelegenheiten bieten großes Potential für die Reduzierung von Verkehrsaufkommen und Verschmutzung. Zudem machen sie finanziell Sinn, da sich Fahrer und Mitfahrer die Kosten für Fahrten teilen. Demnach wurde das Thema bereits oft untersucht und zahlreiche Algorithmen vorgestellt, die Fahrer und Mitfahrer zusammenbringen sollen.

In dieser Arbeit wird zunächst eine detaillierte Übersicht über verwandte Arbeiten gegeben. Dabei werden mehrere Klassifikationen für das Ridesharing Problem, verschiedene Algorithmen für das Kombinieren von Fahrern und Mitfahrern und Ansätze zur Qualitätsoptimierung vorgestellt und diskutiert.

Viele dieser Arbeiten setzen auf ein Verfahren mit dem Prinzip „filtere und verfeinere", welches darauf basiert die Problemgröße zu verkleinern indem die Fahrer herausgefiltert werden, welche für einen Mitfahrer nicht in Frage kommen können. Lösungen mit diesem Ansatz sind schnell, solange die Filter entsprechend viel vorfiltern können. Ist dies nicht der Fall führt ihr Flaschenhals, die vielen resultierenden Kürzeste-Wege-Anfragen, zu längeren Laufzeiten.

Es wird eine Lösung für das *Ridesharing Problem* mit einzelnen Fahrern und mehreren Mitfahrern vorgestellt, die nicht auf diesem „filtere und verfeinere“ Verfahren aufbaut. Statdessen wird eine schnelle und dynamischen Lösung für *one-to-many* (*many-to-one*) Kürzeste-Wege-Anfragen verwendet, die alle benötigten kürzesten Wege innerhalb eines Bruchteils der für einzelne Kürzeste-Wege-Anfragen benötigten Zeit berechnet. Dies wird genutzt um für alle sinnvollen Paarungen zwischen Fahrern und Mitfahrern die entstehenden Umwege, die beim zusammenführen auftreten würden, schnell zu berechnen.

Die Lösung kann leicht auf verwandte *Ridesharing Probleme* angepasst werden, wie zum Besipiel die Taxi Variante des Problems. Dies wird mit einem zweiten Algorithmus gezeigt, welcher in der Lage ist Taxirouten für eine Stadt zu planen.

Experimente zeigen, dass unser Algorithmus kurze Antwortzeiten von unter 100ms erreicht für bis zu 150’000 Fahrer in einem Zeitraum von drei Stunden. Die vorgestellte Lösung wird mit einem Algorithmus aufbauend auf dem „filtere und verfeinere“ Verfahren verglichen. Abschließend wird die Lösung für die Taxi Variante des Problems auf echten Taxidaten von New York getestet und zeigt, dass 400’000 Anfragen, die über einen Tag verteilt waren, auf 2’101 Taxis aufgeteilt werden können, wobei pro Anfrage im Schnitt 70ms benötigt werden.

Contents

1. Introduction

Ridesharing in the scope of this thesis is the matching of two or more people, one driver and multiple riders, who wish to travel from individual starting locations to individual destinations. By matching these people they share a single vehicle. This obviously has the potential of reducing traffic, fuel consumption and pollution, but it also allows users to share their trip expenses. Therefore this type of ridesharing is an important topic in the modern society.

Initially ridesharing platforms like BlaBlaCar [1] or Karzoo [2] matched those people who have start- and end-points of their trips relatively close to each other. The search is very restrictive and matching is provided by listing drivers within a certain distance around origin and destination of the person searching for a ride. Thus possible matches where the riders trip is only a smaller part of the drivers original trip will not be considered. Same counts for solutions, which would allow that the driver takes smaller detours, even though this would clearly lead to higher matching rates.

In addition, nowadays almost everyone is connected online, people can easily offer and request trips whenever they want and wherever they are. This leads towards the need of a dynamic online ridesharing service, which must be able to answer requests within several milliseconds and provide automatic matches more sophisticated than a simple radial searches around origins and destinations of trips.

By dynamic online ridesharing we refer to a service where an automated process matches drivers and riders, possibly on very short notice, and automatically updates the trip for the driver, which then can include several smaller detours for reaching riders.

Uber [3] and Lyft[4] introduced systems for a taxi-like service with fixed drivers and only one scheduled transportation at a time. Later on Uber presented UberPool which still is a solution for the taxi-like service but schedules the trip to fulfill multiple requests within a single trip, leading to some smaller detours for the passengers but also to cheaper prices for the participants. Volkswagen presents a similar service with its subsidiary MOIA [5] for multiple cities in Germany. There exist further "ridesharing" services, but as far as we know there is no existing service that provides a full scheduling service for a private driver

[1] `www.blablacar.de`, [Online; accessed 07-January-2019]
[2] `www.karzoo.eu`, [Online; accessed 07-January-2019]
[3] `www.uber.com`, [Online; accessed 07-January-2019]
[4] `www.lyft.com`, [Online; accessed 07-January-2019]
[5] `www.moia.io`, [Online; accessed 07-January-2019]

by automatically matching him with requests and updating his scheduled trip on a short notice or even within the ride itself.

A lot of related work can be found tackling all different kinds of ridesharing. Some solutions are for commuters, some for taxis and others for carpooling. There is a sheer amount of different set-ups and approaches to solve the corresponding problems. But only a few of them tackle the same problem as we do. In our work we consider many other papers and their approaches. We present them in the Related Work section.

We then focus on the dynamic online ridesharing problem, more precisely on the more complex variant in which a driver can carry several passengers at the same time or on different sections of the route. We introduce a new approach for solving this kind of dynamic online ridesharing problem. The idea for our solution is based on the work of [GLS+10], which utilizes a fast many-to-many query solution introduced by [KSS+07], to provide fast detour computation for ridesharing. Detour computation is essential for the decision, whether trips can me merged within the tolerances set by the driver and the potential passenger. We adapt our solution afterwards to the related taxi ridesharing problem to show that the approach can also be used for solving related problems in a slightly modified way.

The remainder of this thesis is structured as follows.

In Chapter 2 we provide formal definitions and describe our ridesharing problem as well as the taxi variant of the problem. We further sketch two algorithms, *Bidirectional Dijkstra* and *Contraction Hierarchies*, which build a basis of this work.

In Chapter 3 we have a deeper insight into related work, present their classifications and optimization goals and summarize many different approaches for solving ridesharing problems.

In Chapter 4 we present our own algorithm for a ridesharing service.

In Chapter 5 we show how our previous solution can be easily adapted towards the taxi variant of the ridesharing problem.

In Chapter 6 we describe experiments for both solutions, present the measurements and compare the results to those of related work.

In Chapter 7 we discuss our conclusions and give a further outlook.

2. Preliminaries

In this chapter we define notations of graph theory and provide formal definitions for the focused problems. We further sketch two shortest path algorithms, which are of great importance for the understanding of this thesis.

2.1 Formal Definitions

We model the road network as a directed weighted Graph $G = (V, E, \omega)$ with vertices V, edges E and weight function $\omega : E \to R^+$ that assigns a positive weight to each edge. If no other function ω is specified it represents travel time of an edge $e \in E$. A path P is a series of vertices $\langle v_0, \ldots, v_n \rangle \in V$ with edges $(v_i, v_{i+1}) \in E$ between the vertices. The length $c(P)$ of a path P is the sum of the weights of the edges of P. The length of the shortest path possible between two vertices u, v is denoted as $\mu(u, v)$. Considering a non shortest path $P = \langle u, \ldots, v \rangle$ between vertices u, v with length $c(P)$ and the length of a shortest path $\mu(u, v)$, the *detour* Δ is defined as the difference between these two values. Accordingly, the *detour factor* δ is the ratio between these two values. For a given maximal ratio δ_{max} the maximum allowed detour of a path $P = \langle u, \ldots, v \rangle$ is $\Delta_{max} = \delta_{max} * \mu(u, v)$

2.2 Problem Definitions

There are different definitions of the ridesharing problem, as there are also different types of ridesharing. In chapter 3 we will provide a small overview of the existing categories of ridesharing and the different problems other papers focus on. For now we will just define the focus of this work: *We want to match requests and offers as fast as possible while generating as less as possible detour for the offers and already matched riders.*

2.2.1 Ridesharing

Users of ridesharing are either *drivers* or *riders*. Drivers are people who have a car and ride from an origin to a destination at a certain time of the day. They place an offer into the system and allow other people to share the ride. Riders are people without a car, who also want to travel from an origin to a destination at a specific time. Therefore they place requests. Each time a request is placed the system will try to match it with a set of offers in the system. If a match is found the drivers route of the matched offer gets edited so driver and rider share the ride. Detours for drivers and riders are allowed to improve the probability of finding a match. There are multiple constraints for drivers and riders which

have to be kept in mind, like free seats, maximum detours and timings of the ride. One offer can serve multiple riders at any given time.

Let an **offer** $O = (o, d, p_{max}, t_{es})$ describe the trip of an offered ride with origin $o \in V$, destination $d \in V$, a fix amount of maximum passengers p_{max} and an earliest start time t_{es}. We assume that the driver drives along the path with the shortest travel time. The latest arrival time $t_{la} = t_{es} + \mu(o, d) + \Delta_{max}$ for an offer can be calculated by taking the fastest path plus the maximum allowed detour, as the weights of the graph represent travel time.

A **request** $R = (o, d, t_{es})$ with origin $o \in V$, destination $d \in V$ and earliest start time t_{es} describes the desired trip of person looking for a fitting driver. The latest arrival time can be calculated as for the offer. We denote open requests with R and requests already matched with r. Thus r represents a rider assigned to a particular trip.

An offer $O = (o, d, p_{max}, t_{es})$ and a request $R = (o', d', t_{es'})$ form a reasonable match if:

1. There exists a path $P = \langle o, \ldots, o', \ldots, d', \ldots d \rangle$ with $c(P) \leq \mu(o, d) + \Delta_{max}$
2. For every existing rider $r_i = (o_i, d_i, t_{es_i})$ already matched with the offer O the lengths of the altered path $P_i' = \langle o_i, \ldots, d_i \rangle$ caused by the new request and shared by the rider r_i must not exceed the remaining detour of the rider. Therefore: $c(P_i') \leq \mu(o_i, d_i) + \Delta_{max}^{r_i}$
3. For every existing rider, including the driver of the offer, the altered path P', that would be caused by inserting the new request, may not lead to a situation where a riders trip is not in the time period between earliest start time and latest arrival time.
4. During the whole trip of the offer there must not be more than p_{max} riders sharing a path parallel.

The problem to be solved is to find the reasonable match with the lowest detour as fast as possible, if such a fit exists or else to inform the requester that there is no fitting offer in the system. The goal is to achieve an *online solution* for the problem where a requester doesn't have to wait longer than 100ms for an answer from the system.

Many other papers consider the waiting time of a rider as a further important constraint. This is the maximum time, the rider will wait until he wants to start his trip (in regard of the earliest starting time). But in our opinion this often is just introduced for an easier modelling of the problem which allows further pruning of the algorithm. If for example a rider is not willing to wait longer than 15 minutes for his pickup then all drivers which need more than 15 minutes to arrive at the rider can be omitted by the matching. But in our opinion it makes no difference if a driver waits 5 minutes or 50 minutes as long as he arrives before his latest arrival time. It does not matter if the additional time counts for actually driving detours or for simply waiting for a driver. We will further discuss this in the related work section 3.

2.2.2 Taxi-Ridesharing

Many other works presented in Section 3 deal with taxis instead of ridesharing drivers who simultaneously drive towards their own destination. As this is an interesting topic as well, we adapt our definition of the ridesharing problem towards a taxi ridesharing problem to see how well our approach behaves in this different situation.

This time we have taxis and riders. We presume that taxis can be spawned anywhere on the map and that there are no restrictions to the amount of taxis available. Therefore the focus will still be on the riders. As before, each rider places a request $R = (o, d, t_{es})$ into the system, with given origin o, destination d and earliest start time t_{es}. All requests made

shall be matched with the requests placed before, generating routes for the taxis. On these routes multiple riders can share a taxi and therefore minimize the total driven time. When all requests are made the generated routes shall be matched to actual taxis.

This problem is a more flexible version of the ridesharing problem before. There are no explicit routes of the drivers which have to be considered, but the less restrictions there are the more possibilities have to be considered. As this problem is more of a proof of concept for our approach it is regarded as a mixture of an online and offline problem. We still want to be as fast as possible while providing an optimal solution for every request. For every request, we match it with a trip, but since the taxis only get submitted towards the trips after the last request is made the problem gets an offline touch. A taxi can serve multiple trips.

2.3 Bidirectional Dijkstra

The Bidirectional Dijkstra is an improved version of the normal Dijkstra search and a solution for finding minimal distances between pairs of points in the graph. As a normal Dijkstra search starts at the origin o of the pair and searches for the destination d, the bidirectional Dijkstra not only starts one search, but two: One starting from the origin and one from the destination. When these two searches meet and some constraints are fulfilled the search stops and the shortest distance is obtained. Faster runtimes of the bidirectional version originate from the fact that fewer vertices have to be visited. This fact is illustrated in Figure 2.1. The unidirectional Dijkstra needs to check all the vertices inside the big circle until it finds the desired distance, whereas the bidirectional version only checks all vertices inside the two smaller circles which are clearly less.

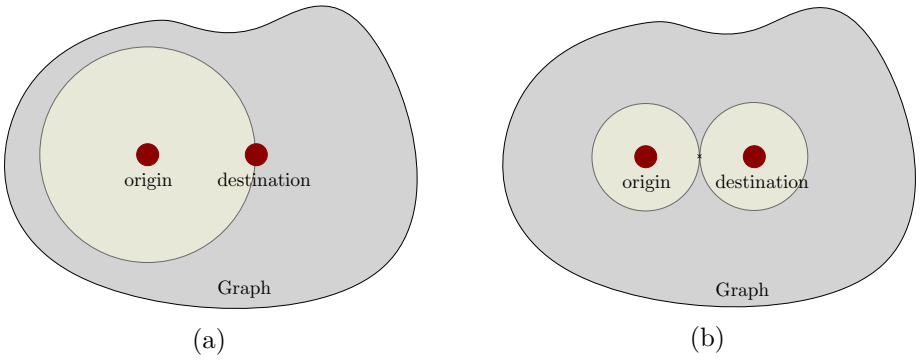


Figure 2.1: Searchspace of the unidirectional Dijkstra (A) and the bidirectional Dijsktra (B).

The exact behaviour is achieved with two priority queues which describe the next vertices to be viewed. One priority queue for the forward search, starting from origin o, and one queue for the backward search, starting from destination d. We will describe the algorithm regarding the forward search. The backward search works the same way but uses the backward graph (each edge and its associated weight are turned around). Before starting the algorithm we mark every vertex as unvisited and it's distance to the start as ∞. We put one first entry into the priority queue, a pair of our start vertex and it's distance to itself, which is zero. Now we will take the first element of the priority queue, which always represents the vertex with the shortest distance to the start of the search and is still unvisited. This vertex is the next one to be visited, therefore we have to make three steps.

1. For all neighbouring vertices of this vertex we calculate the distance as the sum of the distance of the current vertex and the corresponding edge weight.
2. We insert these neighbouring vertices and the calculated distances into the priority queue, if an entry for a vertex already exists we update the distance if it is shorter than the existing entry.
3. We mark the current vertex as visited.

We repeat these steps for the forward and backward search whereas we alternate between the searches after each visited vertex. As soon as we have a vertex m which has been visited by both searches we can stop. The path $P = \langle o, \ldots, m, \ldots, d \rangle$ is a shortest path between o and d. The speedup of unidirectional Dijkstra towards bidirectional is about 2. If the two searches are parallelized it is about 4.

2.4 Contraction Hierarchies

The Bidirectional Dijkstra will become slower with a growing number of vertices contained in the circles shown in Figure 2.1. In graphs with many thousands of vertices and edges and randomly chosen points o, d we need further improvement. Therefore we present a distance query algorithm which is much faster on big instances but still uses bidirectional Dijkstra in its core.

Contraction Hierarchies (**CH**) were presented in 2008 by R. Geisberger [Gei08] and mainly work with two phases. The preprocessing phase contracts the graph to an augmented graph, this has to be done only once for every graph and can take up to several minutes or hours. The augmented graph consists of a multi-layered vertex hierarchy where every vertex lies on a level. The second phase is the actual query which only has to search paths in the augmented graph and finds results in less than a second. For the first phase we need two constructs, *shortcuts* and *vertex-contractions*.

Shortcuts. *A shortcut is a simple edge which excludes some vertices from a Dijkstra search while maintaining correct weights. Therefore if a path visits multiple vertices in a row, where we don't need to look at every vertex individually, we can insert a shortcut-edge from the start of the row towards the end of the row with a weight combining all the weights of the edges in between, as shown in Figure 2.2. Like this a Dijkstra search only has to check one edge. A shortcut can even skip just a single vertex.*

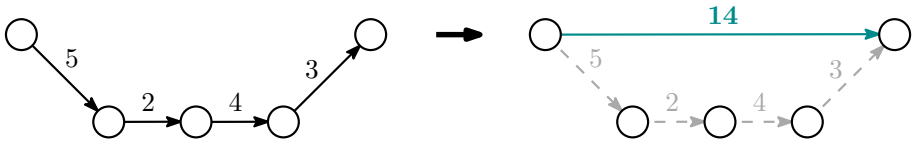


Figure 2.2: Inserting a shortcut over multiple vertices.

Vertex-Contractions. *Contracting a vertex normally means that we exclude it from the graph but add shortcut-edges between all its neighbours maintaining correct distances. In Figure 2.3 we exclude vertex u from an undirected graph and therefore have to insert edges as replacements for the paths that used to pass over u. If we get multi-edges we delete the ones with higher weight. We can also delete edges for which we can find a shorter path in the graph. This is a so called witness search which we won't describe in detail.*

With focus on CHs, vertex-contractions will be adapted. The vertex of the contraction and its edges won't be excluded from the graph. That means for Figure 2.3 that, the vertex and edges marked in grey stay in the graph.

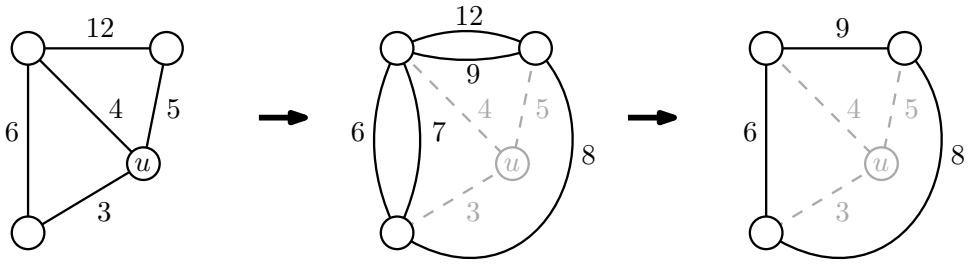


Figure 2.3: Contraction of vertex u.

2.4.1 Preprocessing

First of all we need an order for the vertices of the graph, this order can be arbitrary or chosen with regard to the underlying graph (road-network, social-media graphs, etc.) and has impact on the efficiency of the queries. How exactly a good order is achieved will not be discussed here, but there are different approaches to achieve good orders like bottom-up [GSSV12], top-down [ADGW12] or sampling path-greedy [DGPW14]. The order of the vertices implies a ranking on the same.

With the preprocessing we want to achieve an augmented graph in which a search has to follow either only upward edges, pointing from vertices with lower order to vertices with higher order, or only downward edges, pointing from vertices with higher order towards lower ordered vertices. As such a search would miss many possible paths in the original graph, we need to add one or more shortcuts for every contracted vertex to assure that later on every shortest path can also be found with this kind of search.

The shortcuts to be added can be achieved by contracting the vertices in the given order. While contracting a vertex only edges pointing towards higher ranked vertices, or coming from such, are considered. Shortcuts generated by these contractions are afterwards added to the augmented Graph assuring that a path between two higher ranked vertices formerly leading over a lower ranked vertex will also be represented by a simple shortcut between the higher ranked vertices.

In Figure 2.4 the preprocessing of a simple graph is demonstrated, where the number of each vertex also directly describes its number in the ordering. We start with the least important vertex 1, contract it and add all the shortcuts for paths connecting higher neighbouring vertices over the contracted vertex. Afterwards we do the same for the vertices 2 to 7. After the last vertex is contracted the augmented graph with all its extra shortcuts is finished.

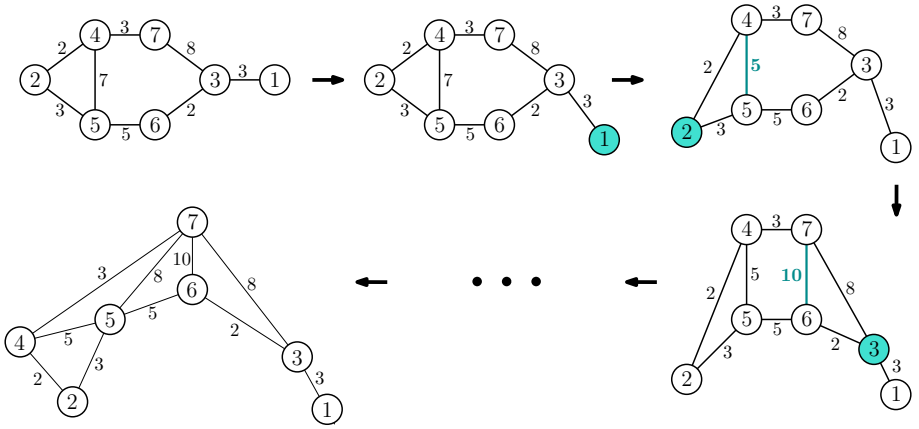


Figure 2.4: CH preprocessing performed on a simple graph. Vertex currently contracted and edges that get changed are marked turquoise. Vertices are moved to achieve a multi-layered representation of the graph.

2.4.2 Queries

A query is executed to find the path with the shortest length $\mu(o, d)$ between two chosen vertices (o, d) of the graph. Therefore a Bidirectional Dijkstra search is performed on the augmented graph. The forward search starts at the origin o, but instead of considering all edges outgoing from o, the search only considers outgoing edges pointing towards higher ranked vertices. For the backward search of the Dijkstra, we start at the destination d. Only incoming edges coming from higher ranked vertices are considered in this search. Both searches visit vertices one by one and mark them as visited. For every vertex m_i, visited by both searches, exists an upward path $P_{up} = \langle o, \ldots, m_i \rangle$ and a downward path $P_{down} = \langle m_i, \ldots, d \rangle$ resulting in a path $\langle o, \ldots, m_i, \ldots, d \rangle$ from o to d. This path gives us an upper bound for the distance $\mu(o, d) = \mu(o, m_i) + \mu(m_i, d)$. As soon as the entries within both priority queues of the searches consist of a minimum key greater than this upper bound, the current smallest upper bound for $\mu(o, d)$ also describes the actual minimum distance between o and d. In Figure 2.5 the search spaces of the two searches are illustrated. Each search space only consists of vertices with higher ranks than the origins of the search.

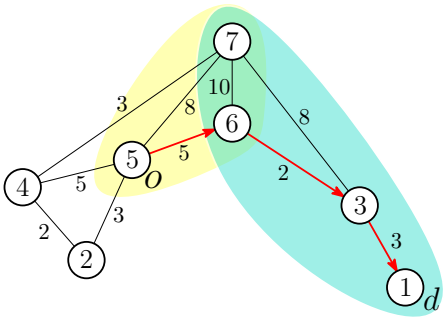


Figure 2.5: The two search spaces for the bidirectional search outgoing from o and d. The shortest path found leading from o to d is marked red.

Due to the preprocessing we can assure that the path found by this query is the shortest path. Shortcuts inserted by the previous phase added those paths into the augmented graph, which would not have been considered before, with this kind of search. As the proof of correctness is rather short we will show it here:

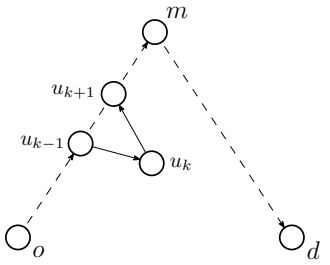


Figure 2.6: Sketch for the proof of correct CH-queries.

Consider a path found by this query like in Figure 2.6 consisting of a upward path $\langle o, \ldots, m \rangle$ and a downward path $\langle m, \ldots, d \rangle$. Assume that there exists another path which is shorter than the one we found with our query. As our searches expand upwards the shorter path must involve a vertex u_k with a rank lower than the rank of the predecessor u_{k-1} and successors u_{k+1} in the path. But if so, the preprocessing phase would have created a shortcut between u_{k-1} and u_{k+1} and u_k can be left out without changing the overall length. Therefore there can't be a shorter path than the one found by our query.

As sometimes not only the distance of the shortest path is of importance but the path itself, a post-processing can be added for the shortcuts, which unpacks the sub-paths represented by them. This step is also needed to transfer the solution for the augmented graph (with shortcuts) towards the original graph.

3. Related Work

We considered many references for this thesis. While reading these references it became clear that the definition of ridesharing is quite loose and not unique across the papers. Moreover different papers care about different aspects of ridesharing and also look for improvements in different areas. The analysis of related papers has therefore widened more than originally expected.

In this chapter we want to give an overview to achieve a better understanding of the topic "ridesharing". We first present found classifications which make sense to us. Afterwards we will cover important works, describe their solutions and classify them. For different works we will provide a small discussion from our point of view.

3.1 Classification and Constraints of Ridesharing

We mainly want to consider the definitions and classifications of four works, each providing an overview over ridesharing themselves.

The first work named "Real-Time Rideshare Matching Problem" from K. Ghoseiri, et al. published 2011 [GHHC11] not only provides a big chapter about its "dynamic rideshare matching optimization model" that is aimed at "identifying suitable matches between passengers requesting rideshare services with appropriate drivers available to carpool [...]"[GHHC11]. It also provides an overview of the state-of-the-art solutions of ridesharing projects, researches and some important classifications.

The matching model considers constraints like age, gender, smoker, etc. We won't focus on these particular constraints in our work but it has to be mentioned that if a ridesharing algorithm shall be provided there is a large amount of constraints which can be considered. Each constraint can simplify calculations and allow additional pruning of search spaces but also makes matching harder and affects the results. Table 3.1 lists some possible information that can be considered as constraints, we marked the ones important for us which are also considered in our own work.

Pricing is a big topic for real life ridesharing as it can have impact on the matching and fairness of the system. But in our work we did not focus on a fair and good pricing scheme, indeed we didn't consider pricing at all because we could not see any great influence of this aspect on the speed of the algorithms to determine optimal paths. Nevertheless, we are sure that our work can also be extended by different pricing schemes.

Table 3.1: A list of possible constraints considered in ridesharing together with typical inputs which can be provided. Important constraints for our work are bold.

Information/Constraints	**Example**
Detour	0.5 ∗ shortest trip distance
Number of seats	4 free seats in a car
Passenger amount	maximum of 3 passengers
Waiting time	a rider can wait 5 min until departure
Timing of the ride	arrival and departure time
Privacy	knowledge over driver/rider that can leak
Costs/Payment	1€ per shared kilometer
Security	private information that can leak
Smoke restriction	only non-smokers
Pet restriction	pets not allowed in vehicle
Age	only young/middle-aged/old people
Gender	only a specific gender as driver

The work also states that there is not only one definition of ridesharing, leading us to the next article "Dynamic Ridesharing" from B. Shen et al. [SHZ16] and the classification of *Dynamic RideSharing versus Static RideSharing* it provides. [SHZ16]. As among the reviewed literature many definitions can be found for dynamic ridesharing (some are listed in [GHHC11]) which all differ slightly we won't provide them all in this work. But we will give one definition each for dynamic and static ridesharing which reflect the core messages.

- **Dynamic Ridesharing** is denoted as ridesharing which matches drivers and riders in real time. Routes are flexible and even after a trip started more riders can join and thus the driven route can be altered on the fly.
- **Static Ridesharing** arranges drivers and riders before the trip starts. Once a driver started his trip no further riders can be matched towards this ride. The pre-calculated route is fix or chosen by the driver himself. Requests must be known ahead of time.

In this work we mainly focus on the dynamic variant of the problem. Therefore the further related works will mainly handle this type of ridesharing.

In the dissertation of W. M. A. Herbawi of 2012 [Her13] further interesting classifications can be found. He divides ridesharing into 4 categories of of single/multiple drivers together with single/multiple riders:

- **Single Driver and Single Rider** denotes a matching where each driver in the system can pick up only one rider and each rider wants to arrive at his destination by driving with only one driver.
- **Single Driver and Multiple Rider** denotes a matching where each driver can serve multiple requests. But each rider still only is matched with one driver.
- **Multiple Driver and Single Rider** denotes a matching where a rider can reach his destination by driving with multiple drivers. Each driver will bring him closer towards his destination. This allows for a higher possibility to find a trip for the rider. But every driver only serves one rider, even if it is only for a part of the riders trip. This also sometimes is denoted as multi hop ridesharing as the rider makes "multiple hops" [Dre12].
- **Multiple Driver and Multiple Rider** extends the previous by allowing the drivers to serve multiple riders.

Our work will focus on the Single Driver Multiple Rider problem whereas the work on which our solution is build (Section 3.3.9) solves the simpler problem of Single Driver and Single Rider. As this thesis deals with Single Driver only, we restrict the consideration of related works also to this scenario. But for interested readers an example of a solution for the Multiple Driver and Single Rider problem can be found in the publication of F. Drews and D. Luxen [Dre12].

The last work we consider for our classifications of ridesharing is "Ridesharing: The state-of-the-art and future directions" from M. Furuhata et al. published in 2013 [FDO+13]. In this paper two main types of services are presented:

- **Service operators:** "Operate ridesharing services using their own vehicle and drivers" [FDO+13]. A taxi or a normal bus also can be seen as ridesharing, as multiple persons share the same vehicle. But the drivers of these vehicles don't have a fix destination for themselves. Their only purpose of driving is the transportation of other persons. We also count companies like Uber towards this rubric as the concept is really close to the one of normal taxis (Private persons can register as driver into the system and are informed if they shall transport a rider, the private person can be seen as a taxi-driver). As in all other works we will look into, only ridesharing of taxis is considered under this rubric, we will denote to it in the future as **"Taxi Ridesharing"**. Other terms relating to these kind of operations are carpooling and dial-a-ride.
- **Matching agencies:** "Facilitate ridesharing services by matching between individual car drivers and passengers" [FDO+13]. These kind of services doesn't employ drivers on their own. Drivers in this service want to reach a destination on their own and are willing to transport other riders alongside. The service also doesn't need vehicles on their own like taxis as the drivers provide them. If we we will speak of simple "Ridesharing" we will always denote to this form of ridesharing where a driver has fix origin and destination on his own.

In the further sections we will discuss a bigger number of taxi ridesharing works than normal ridesharing, even though our main topic is closer to the non-taxi variant. This is due to the fact that we simply found more publications on the taxi topic and only some smaller publications were found for the latter. This fact also leads us towards the implementation of a taxi variant of our algorithm, in order to provide better comparisons with existing algorithms and see whether the basic idea of our algorithm can also be applied to similar problems.

The publication of M. Furuhata [FDO+13] also researched 39 "matching agencies", which are implemented solutions for the ridesharing problem, and then categorized them into six classes. Even if on first sight they may seem very important for this section on closer examination they are not. This is due to the fact that whole systems, which already exist for ridesharing, are inspected and not theoretical algorithms of the publications we consider here. For the sake of completeness we will list these six classes and describe them as good as possible. But as they are quite specific and we won't use them further.

1. **Dynamic real-time ridesharing:** "Providing an automated process of ride-matching between drivers and passengers on very short notice or even en-route" [FDO+13]. This is more or less the same as our definition of dynamic ridesharing provided above. It describes a system that calculates everything: The route of the driver, the route of the rider, pick-up and drop-off locations and matching of the drivers and riders.
2. **Carpooling:** "servicing for commuters that share transportation to work in a private vehicle with another worker" [FDO+13]. Here the participants typically have similar origins or destinations. The work locations as well as the start and end times of the work are of main importance.

3. **Long-distance ride-match:** "servicing for travellers taking long distance trips (inter-city, inter-state and inter-country)" [FDO+13]. Long-distance travellers most of the time are more flexible than the two categories mentioned above.
4. **One-shot ride-match:** Is a hybrid of carpooling and long-distance ride-match. They consider different search criteria.
5. **Bulletin-board:** "ridesharing opportunities based on notice boards" [FDO+13]. Agencies providing this kind of rid sharing want to keep everything as flexible as possible. User search methods are based on keywords and/or lists.
6. **Flexible carpooling:** "providing ridesharing opportunities without prearrangement in advance but coordinated on the spot" [FDO+13]. Normally here no matching agencies are used. The meeting time and point for a ride is publicly known among the potential participants. The ridesharing is then formed in a first-come-first-serve kind of manner.

It can be seen that there are numerous aspects that can be considered for the classification of ridesharing. In a nutshell, the main aspects of classification from our point of view is to distinguish:

$$\begin{aligned} \text{Dynamic Ridesharing} &\longleftrightarrow \text{Static Ridesharing} \\ \text{Taxi Ridesharing} &\longleftrightarrow \text{Ridesharing} \\ \text{Single Driver} &\longleftrightarrow \text{Multiple Driver} \\ \text{Single Rider} &\longleftrightarrow \text{Multiple Rider} \end{aligned}$$

In the upcoming sections of this paper we will mainly differentiate between Ridesharing and Taxi Ridesharing, leaving the rest fixed to Dynamic Ridesharing (we are focussed on response times), Single Driver and Multiple Rider, if not denoted otherwise.

3.2 Optimizations for Ridesharing

So far we have shown possible classifications and constraints related to ridesharing. When we look at the algorithms used to finally assign riders to drivers, we may rise the question what are the directions in which these algorithms may be optimized. Across the publications mainly three types of optimizations can be found which are also described in the publication "Dynamic Ridesharing" [SHZ16] mentioned before:

- **Minimizing the vehicle travel distance:** This normally makes the algorithm run slower as it has sometimes to reorganize and recalculate bigger routes and/or has to keep bigger data-structures in memory for providing better solutions.
- **Maximizing the rate of requests been matched:** Routes with a small detour or fast response time does not make an algorithm useful, if there are not enough matches between drivers and riders.
- **Minimizing the system response time:** Here the target is to get an acceptable but not necessarily optimal result as fast as possible, thus minimizing the time a rider needs to wait to get an answer from the service. This is especially important for online (or real-time) services where requests can arrive shortly before the desired departure time.

It is impossible to optimize an algorithm towards all of these three points at the same time, on some point there must be a decision made between quality and time. Regarding the quality of a solution it has to be said that finding an optimal solution can be NP-complete depending on the exact problem. In the publication "T-Share: A Large-Scale

Dynamic Taxi Ridesharing Service" from S. Ma et al. published in 2013 [MZW13] the "Total Distance Optimization Taxi Ridesharing Problem (TDOTRP)" has been proven to be a generalization of the "Travelling Salesman Problem with Time Windows (TSPTW)" which already has been proven to be NP-complete [Sav85]. The TDOTRP is formalized as decision problem: "given a stream of requests S_r, a start time t_s and a set of taxi statuses $S_{t_{status}}$, a road network RN in which each road segment is associated with a speed limit, a number $P \in [0, 100]$ and a number $D \geq 0$, plan a schedule for each taxi such that the total travel distance of all taxis is no larger than D and the fraction of satisfied queries is at least P percent" [MZW13].

In "Large Scale Real-time Ridesharing with Service Guarantee on Road Networks" published by Y. Hung et al. in 2014 [HBJW14] it is stated that the problem of "[...] real-time ridesharing is NP-hard as the classical Hamiltonian path problem can be reduced to this problem (assuming all the trips have the same ending points and requested in almost the same time)". The Hamiltonian path problem described the problem of finding a path in an undirected or directed graph that visits each vertex exactly once and has been proven to be NP-complete [Har82].

M. Asghari et al. also denoted in their work [ADS+16] that "the ridesharing problem is NP-Hard since the Vehicle Routing Problem (VRP) is reducible to the ridesharing problem in polynomial time." The VRP describes the problem of "finding the optimal set of routes for a fleet of vehicles to traverse in order to deliver to given set of customers" [DR59].

We won't further inspect the NP-completeness of the different ridesharing problems. But as has been shown the problem can be NP-complete which justifies the use of heuristics found in many solutions. Minimizing the vehicle travel distance has a "greedy nature" [HBJW14] itself, especially for dynamic online ridesharing: when additional new requests are made the past optimal matching between drivers and riders may not be the minimum any more. Shuffling trips calculated before can bring better solutions. However, regarding real-time, the best that an algorithm can achieve in a short time is often a result without reconsidering every decision made before.

We will always differentiate the papers regarding quality and speed as it is not always fair to compare those algorithms trying to achieve a high matching rate with optimal routes against those who shall provide a good user experience and scalability with fast response time. The main focus of our work is runtime, but for every request made the optimal solution is chosen, which can be achieved without changing decisions made before.

3.3 Categories and Examples of Ridesharing Systems

In this section we will provide further insight into some publications. We will describe the solutions of, from our perspective, more convincing works in detail, but for some others we will just outline the idea of the solution. We will start with solutions for the taxi ridesharing problem and later on deal with non-taxi publications, leading us towards our own work.

3.3.1 Filter and Refine Framework

"Dynamic Ridesharing" from B. Shen et al. [SHZ16] was mentioned before, as it also provides a good overview over the topic of ridesharing. This time we will cover the solution framework it presents. This framework is called "**Filter and Refine**" and summaries the structure of multiple algorithms solving the (taxi) ridesharing problem. The core idea consists of two steps.

1. **Filter:** *Filter drivers and requests that have no matching possibility.* This shall reduce the scale of the problem. When requests are made first all of the existing drivers are considered as possible matches. But it makes no sense for a rider who travels inside a city (e.g. Karlsruhe) to be matched with a driver multiple hundred of kilometres away travelling inside another city (e.g. Berlin). Like this, for a given request, the driver-set of possible drivers shall be filtered beforehand, minimizing the possible matchings.

   For example in "T-Share: A Large-Scale Dynamic Taxi Ridesharing Service" [MZW13] a spatio-temporal indexing is used to partition the road network into a grid. Each cell of the grid maintains information about travel times, travel distances and the vehicles inside or expected to enter the cell. When a new request is submitted the system uses grids from near to far to filter the candidate drivers. The stored distances for the grids can be taken to estimate the detours and remove drivers from the matching whose detours would be to big. We will have a closer look into this work in Section 3.3.3

2. **Refine:** *Calculate the matching on the filtered data set of the first step.* This step has to reschedule the routes of the possible drivers and find the best fitting, while satisfying all the constraints. Branch-and-bound and integer programming solutions can be chosen, but are "not suitable for real-time dynamic ridesharing" [SHZ16]. Also a simple insertion technique is discussed, reducing the problem size, of inserting a request into a trip with k stops, from complexity $O(k!)$ to $O(k^2)$. However, this method "does not try to achieve optimality" [SHZ16]. The most promising solution from the publication's point of view is a *Kinetic Tree structure* proposed in [HBJW14]. We will introduce this structure in the following Section 3.3.2, for now it shall be said that the structure allows to store all valid trip schedules in a tree and thus doesn't commit to a fix route as a new request is inserted, allowing further adaptations.

Most of the following algorithms can be classified as filter and refine algorithms. For this reason we considered an algorithm, which relies on this filter and refine approach, for comparison in our experiments.

3.3.2 Kinetic Tree Algorithm

"Large Scale Real-time Ridesharing with Service Guarantee on Road Networks" published by Y. Huang et al. in 2014 [HBJW14] proposes the kinetic tree algorithm mentioned before and compares it towards a branch and bound algorithm and a mixed-integer programming approach. As the two latter are only for comparison purposes and have more of an offline nature, we will focus on the kinetic tree structure and its benefits, of allowing further adjustments of the routes. This work focuses on a fast and scalable (online) solution which also improves the quality of the matching in contrary to more greedy solutions, like simple insertion techniques. The framework of this algorithm is the same as described in 3.3.1. The set of possible matches first gets reduced by estimations made with the help of a grid structure. The remaining candidates will then be tested for the matching.

Kinetic Tree Structure

The kinetic tree structure maintains calculations performed, uses them effectively for new requests and uses new matches to further extend it. More exactly the structure maintains all valid trip schedules with respect to a driver's current location. Whenever the driver moves, parts of the tree-schedule become obsolete. The root of the tree always represents the drivers current location, the branches of the tree represent valid schedules.

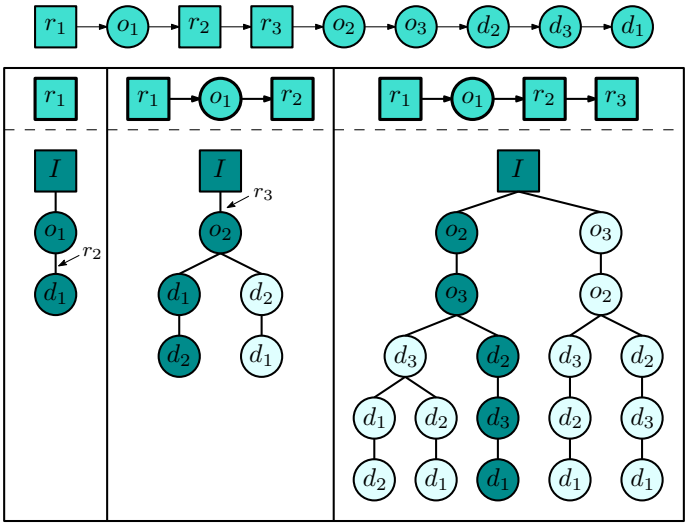


Figure 3.1: Kinetic Tree Structure

Figure 3.1 shows an example following an example of the publication. On top we see the order of requests r, pick-ups o and drop-offs d: First request r_1 arrives and is scheduled, afterwards the first rider is picked up (o_1) and then the next request arrives r_2, and so on. We will look at the kinetic tree each time a new request r_i arrives. Variable I states the current position of the driver and is always the root of the tree.

At the first step there is only one way to schedule the new rider therefore the tree has only one branch. When the second request arrives, the first rider has already been picked up. Now we assume that the option to drop the first customer before picking up the second is invalid, thus picking up rider 2 is the next step. But now we can decide on whether to drop off rider one or rider two first. This possibility is represented by creating two branches modelling the two schedules. Only after the driver picked up the second rider he would have to decide which schedule to follow, and of course he would take the shorter one. But in our scenario the third request arrives before the second rider has been picked up. This time we assume the only two valid options differ in the order rider two and three are picked up. If rider two is chosen we get the left sub-tree, else the right. For every valid combination of drop-offs the branches diverge further increasing the breadth of the tree. If a driver has to decide on the exact schedule he will always take the one with the currently shortest distance (marked in a darker color).

The exact implementational details for this structure are given in the paper [HBJW14]. To further improve the algorithm of this publication a hotspot based optimization is introduced. The size of the kinetic tree structure rises exponentially when there are multiple pick-up or drop-off locations close to each other. An example given in the paper denotes that if eight pick-ups occur close to another, for example at an airport, any permutation of the pick-ups may result in a valid schedule. This ends up in $8! = 40'320$ possibilities before even considering drop-offs. For this kind of problems a hotspot clustering algorithm checks for every new point if any other pick-ups or drop-offs appear in a close region around it. If so, they get put together and regarded as only one point. A proof for the efficiency is given in [HBJW14].

Experiment Results and Discussion

For testing the algorithm a road network of Shanghai was chosen, consisting of 122'319 vertices and 188'426 edges. Tripdata were generated out of a taxi dataset containing 432'327 trips made by 17'000 taxis on one day. Standard settings determined by the work

describe a scheduling capacity of 4 riders per car, a waiting time of 10 minutes and an allowed detour factor of 20%. The optimal amount of taxis was chosen to be 10'000 as the drop-rate of not served requests was below 0.01%. With only 5'000 taxis the drop-rate was around 8.52%. And a bigger amount of taxis can't improve the drop-rate much more but leads to longer calculations.

The results of the work seem to be very good regarding the quality and calculation times. With the given parameters the handling of a request took around $15 - 20$ ms which is fast enough for an online solution. Increasing waiting time and detour factor to 25 minutes and 50% increase this time towards 80 ms.

3.3.3 Partition-Based Methods

Partitioning the road network is an often chosen method for filtering and downsizing the original problem. Thereby the scalability of the underlying matching algorithms is improved. Many works can be found which use or discuss this approach. A basis for these approaches is being researched in the work of Möhring et al. [MSS+06], which compares several partitioning schemes with regard to speeding up the Dijkstra algorithm. One example for the usage of partitioning is described in the work of D. Pelzer et al. [PXZ+15].

They create partitions of a road network trying to represent the topology of it. Routes then can be later described as "corridors" which are sequences of partitions. The matching algorithm then compares the corridors of drivers and riders first. As for a matching only a subset of all partitions are visited by the matching, the problem size can be reduced. However, the matching rate also gets smaller as some possible legit matchings get filtered. Another method of partitioning is to overlay the graph with a grid and collapse some parts of the graph into the grid cells. We will present an example for such an approach in the following section.

Spatio-Temporal Indexing

"T-Share: A Large-Scale Dynamic Taxi Ridesharing Service", published by S. Ma et al. in 2013 [MZW13], introduces a solution for the taxi variant of the dynamic ridesharing problem. The main goals of the work are to "investigate the potential of taxi ridesharing in serving more taxi ride queries by comparing with the case where no ridesharing is conducted" and to build a system with practical use. The filter and refine method is used, moreover this work may have had big impact on the work of "Dynamic Ridesharing" and is, in regard of our knowledge, the first work to propose the spatio-temporal indexing for ridesharing problems.

Every time a new request arrives the spatio-temporal indexing is used to filter a set of feasible taxis for the trip. The simple idea behind the indexing is, that any taxi that is too far away to arrive at the origin of the rider in time (before the latest departure time) can't be matched. To check if a taxi can reach the rider a straight forward solution would be to take shortest distance queries. But as for every taxi at least one query has to be calculated this would take too much time and most of the time a matching couldn't be provided in real-time. The proposed indexing tries to approximate the distances with pre-calculated distances. Thus only approximated values for the distances are calculated, but the time required is considerably less.

First the road network gets partitioned into a grid, each cell gets represented by an "anchor" vertex, the vertex closest to the grids geographical center. For each pair of anchor vertices the shortest travel times and/or shortest distances are pre-computed and stored in a matrix. Each cell also maintains three lists: "a temporally-ordered grid cell list, a spatially-ordered grid cell list and a taxi list". The first two lists contain an entry for each other cell and

are sorted in ascending order by travel time respective distance to the currently viewed cell. These two lists are static and can be calculated with the pre-computed matrix. The taxi list records the IDs of all taxis which are scheduled to enter the grid in near future, whereas each taxi ID is also tagged with a timestamp indicating when exactly the taxi will enter the grid cell.

The work presents a single-side taxi searching as well as a double-sided one. The prior considers only the origin of the request for the filter, the latter also considers the destination and thus provides a better filtering. We will describe the more improved one, the second:

The search starts with two empty sets S_o and S_d, which later should contain taxis that can reach the origin/destination of the request in time. The sets will be filled step by step until the intersection of both sets $S = S_o \cap S_d$ is not empty or both sets can't be further extended. The taxis to be entered into a set are determined by the lists stored at the grid cell. We're looking at the case of the origin: We take the first entry in the ascending ordered list of travel times. Remember, the travel time denotes the time needed to travel from another grid towards the grid viewed. We take all taxis stored inside this grid or scheduled to enter it and calculate if it could reach the request within time. If so it gets added to the set S_o. Same is done for the destination. Like this the cut $S = S_o \cap S_d$ only will consist of taxis which are capable of reaching the origin as well as the destination of the request in time. Therefore the set S contains the taxis that will be considered in the further matching.

It has to be mentioned that this filter does not only exclude taxis not able to satisfy the request but also others, as the filter is stopped as soon as $S \neq \emptyset$. It is even possible that the taxi, with minimum increase of travel distance for a query, will be excluded from the matching by this filter. But the work states that the increase of travel distance is about a bearable 1% whereas the number of selected taxis is reduced by a further 50% in comparison to a single-sided search which does not stop early.

Discussion

Without question the provided indexing is useful and improves matching algorithms by reducing the search space. Another possibility, to estimate travel times or distances, we would like to mention, is to take advantage of the fact that many road networks also provide coordinates for their vertices. The distance between two vertices then can be estimated by calculating the air-line distance between the coordinates of these two vertices. Further a minimum travel time can be calculated if an average speed for the vehicles is determined.

3.3.4 Auction based approach

Other promising approaches are auction-based, where drivers bid in an automated way on a request and the highest bidder is matched. In the work of M. Asghari et al. from 2016 [ADS+16] such an approach is introduced. The advantage of this method is that every driver calculates his offer with his own device, submitting his result to a server which only has to determine a minimum out of all submittings. This outsources the calculations of the matching algorithm to the devices of the drivers, thus parallelizing all the calculations and therefore making them fast. But in our opinion this concept comes with some security concerns, as the most important operations are client-side and may be easily manipulated. The work also heavily focuses on a fair pricing model, but as this is not part of our work we are not going to elaborate this aspect further.

The algorithm considers each driver in the system as a bidder and the riders which make a request are viewed as goods. The bidding process is automatized and the actual driver does not engage in bidding. The server will act as an auctioneer. As soon as a request arrives at

the server it presents it towards the bidders. Each bidder computes a new schedule based on a branch-and-bound algorithm, which includes the new request, and generates a bid based on the driver's and rider's profile. Afterwards the bid is submitted to the server.

The bids are sealed, meaning that only the bidder himself and the server know how much a bidder has bid. Once the bids have arrived the server picks the highest bid and matches the request with the corresponding driver. The system is further optimized as the server only asks drivers for a bid which are close enough to the pick-up location to reach a rider in time. This is achieved due to an algorithm similar to the one described in Section 3.3.3. In case of a tie, one of the highest bidders is chosen randomly. The bid of a driver represents the profit the system can gain if the request is assigned to that driver. The fairness of the bids is achieved through the pricing model, which can be looked up in the original work [ADS+16].

Discussion

This approach definitely tackles the online ridesharing problem and may be the fastest algorithm to find a matching, thanks to its high degree of parallelization. But it is questionable if this would also work well in the real world, where devices are variable, internet connections are unstable and persons show malicious behaviour to maximize their own profits.

3.3.5 Integer Linear Programming

Another big group of solutions consider Integer Linear Programming (ILP). These algorithms mainly improve the quality of the solution and show bigger running times, which can take up to multiple hours or days. As our own work does not consider these kind of optimizations we will only provide a coarse overview over existing works and concepts.

The main idea of ILPs is to put the problem into a mathematical formulation which later on shall be optimized, like finding the minimum or maximum of a given Equation. ILPs are a form of linear programs where variables are restricted to integers. By this fact the problems become NP-hard but can provide solutions for a wide range of problems. Especially it can be used for the (taxi) ridesharing problem to find an optimal matching or routing.

J.-F. Cordeau published a work in 2005 [Cor06] introducing a solution for the Dial-a-Ride (DAR) Problem using a Branch-and-Cut Algorithm which solves a "mixed-integer programming formulation" and finds optimal or near-optimal solutions. A mixed integer program denotes that some decision variables are not discrete. Our definition of the taxi ridesharing problem can be reduced to the DAR problem by only giving the taxis a fix origin and a fix destination depot. Subsequently, the solution could be applied to the taxi ridesharing problem and with some changes maybe even for the non-taxi variant. The algorithm provided relies heavily on mathematically formulations which will be skipped here. But it is basically an improvement of a Branch-and-Bound algorithm that is applied to problems with fewer data.

"On-demand high-capacity ride-sharing via dynamic trip-vehicle assignment" by J. Alonso-Mora et al. [AMSW+17] is another interesting work trying to combine a fast greedy solution with an optimal solution of ILPs. It uses the fact that even if a request should be answered in real time, there is often much time left until the travel of the rider really begins. This time should be used for further optimizations on the matchings and routes. This work first assigns a greedy matching for a request where the quality of the matching is not important. Congruently an ILP is formulated which can produce increasingly better results over time. If a request arrives at 10:00am and the departure time is set to 11:30am there are 1 1/2 hours left for the server to optimize the matching whereas the rider can

be informed by the greedy matching if he can be served. It is further stated that the algorithm is "any time optimal" meaning that it can be stopped at any time outputting the currently best solution found by the ILP. And if enough resources (running time and running power) are provided the optimal solution for the problem will be found. Like this a dynamic on-demand solution can be provided.

This framework can be adapted to any kind of the ridesharing problem and finds solution for bigger instances (1'000-3'000 driver, ~400'000 requests in the city of New York) within a mean computation time less than one minute [AMSW+17]. But the results also show that again the calculations are heavily depending on the maximum waiting time of a passenger.

In "Activity-based ridesharing: Increasing flexibility by time geography" by Y. Wang et al. [WKW16] another solution to a ridesharing problem is introduced which uses ILPs. This work shows the possibilities of activity-based ridesharing, where the destination for a ride is flexible as long as certain criteria are met. For example if a rider wants to go to a supermarket it may not matter which supermarket it is. Therefore the algorithm can consider multiple destinations for the rider as long as it is a supermarket. This provides a bigger flexibility in the routing and shall provide a higher matching rate. To justify this approach, the improvements of matching rates shall not depend on any heuristics. Therefore ILPs were chosen for generating optimal solutions, showing a good example of the usage of optimal solutions.

V. Armant and K. N. Brown [AB14] also published a solution for the ridesharing problem with the usage of mixed integer programming. They tackled the combinatorial problem with two optimization techniques, "linearisation" and "symmetry breaking" and achieved solutions which could handle up to twice as many users within a given CPU-time as the version without optimizations.

3.3.6 Simulated Annealing

Simulated Annealing is a wide spread method of solving different kind of optimization problems. The idea is to simulate the cooling process of a structure. As long as the structure is "hot" it can be easily deformed and takes random states into consideration. Over time the structure cools down allowing less and less deformations, always keeping those states in mind which had a good/optimal structure. Once the annealing is finished at least a local minimum should be found or in the best case even the global minimum, which represents a stable state of the structure.

Y. Lin et al presented a simulated annealing solution for solving the ridesharing problem with taxis [LLQX12]. They considered rather small experiments with only 29 taxis and only a few requests but already showed the possibilities of simulated annealing for ridesharing.

In "Dynamic Shared-Taxi Dispatch Algorithm with Hybrid-Simulated Annealing" published by J. Jung and R. Jayakrishnan in 2016 [JJP16] the concept of simulated annealing is further improved to also solve the taxi ridesharing problem. They speed up the normal simulated annealing by building a Hybrid variant which considers a fast and simple insertion technique for the first solutions of the Simulated Annealing. The insertion technique is also used to constrain the randomness leading to fewer invalid results during the cooling. The algorithm can answer multiple requests at once and also optimizes already matched routes during a cooling process. For intermediate test sizes (600 Drivers, City of Seoul, 15 minutes waiting time, detour factor 2.0) this method achieved good response times for requests within several seconds, providing better solutions than a simple greedy solution. With bigger instances the simulated annealing clearly lacks speed.

3.3.7 Other Solutions and Approaches

This Section gives an overview of the remaining publications examined during the creation of this thesis.

Another problem solving category contain evolutionary algorithms such as genetic algorithms, which model the problem as a population. The starting population is then adjusted with operations of mutation, crossover and selection. Following the idea of Darwins evolution only populations with good results are selected and "survive" over time.

W. Herbawi and M. Weber developed an algorithm which combines a genetic approach with an insertion heuristic to solve the ride matching problem [HW12] which can be tuned between running time and solution quality.

The work of N. Agatz et al. [AESW11] introduces optimization-based approaches for matching drivers and riders. They only solve the problem for a single driver single rider matching but introduce the *Rolling Horizon Strategy*, which allows any matching to be changed up till the actual departure of a rider. This work is a often chosen reference as it provides a study of the possibilities of ridesharing built upon a simulation in Atlanta.

3.3.8 Tackling our Ridesharing Problem

Works presented so far are mainly listed for giving an impression of the amount of solutions and different strategies. But often the works handled the taxi variant of the ridesharing problem, DAR problems or completely other problems which are related to ridesharing. Those works could be partially adapted to our ridesharing problem but while converting them some optimizations could get lost. Therefore, in this section, we will introduce those works which have a similar definition of the ridesharing problem as we do. We will later on compare our solution with one of these works.

M. Schreieck et al. published a rather simple solution for the ridesharing problem with single driver and single rider [SSS+16]. First of all, for each graph several key vertices K are determined (for example intersections of a road network). For each route a driver wants to take, the key vertices K_i of this route store the information of the driver. If a request arrives with origin o and destination d the set of key vertices K_o and K_d within circles around o and d with radius r are looked at. A cut between the driver informations stored in the key vertices of K_o and K_d is formed. If in this cut a driver remains he can fulfill the request.

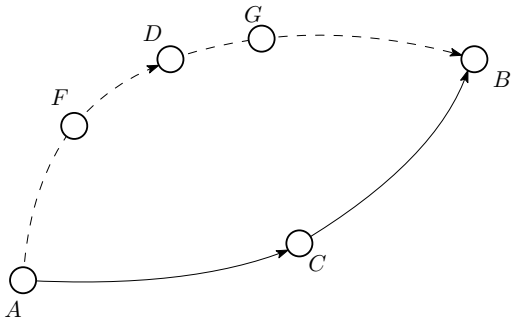


Figure 3.2: A matching between request F-G and offer A-B that can be achieved if the original route of the offer from A to B over C is altered.

But as far as we see this solution it has some problems. As only the circles around the requests origin and destination define the drivers which are considered a couple of possible drivers may be excluded. An example for this problem is illustrated in Figure 3.2. It is a typical problem of ridesharing where a driver originally takes a route from A to B over C but an alternative route of similar quality leading over D exists. When the driver makes

his offer, the route is presumed to go over C, and if a request arrives leading from F to G the method described would find that F and G are not near enough to this route and thus exclude the offer, not taking into account that a switch to the alternative route over D would allow a good match.

The solution provided by M. Schreieck et al. doesn't consider these alternatives. Furthermore, the experiments presented in the work with 10'000 drivers in Munich, showed that a single request took up to 400 ms which is to much for our online-definition.

SORS

B. Cici et al introduced a Scalable Online Ridesharing System (SORS) [CML16], which also solves a ridesharing problem similar to ours, only formulated for commuters. They refined a previous solution presented in [CML15] which solves the ridesharing problem by combining a database ("MongoDB") with a Constraint Satisfier Module and a Matching Module. With SORS they replaced the database with a suited road network data structure improving the scalability. The idea behind the solution is similar to the one presented by M. Schreieck and comes with the same problem of not considering alternative routes for an offer.

SORS utilizes a two order data structure, consisting of a tree structure, containing the intersections of the road network, which then points with the second order structure towards symbol tables consisting of timings and ids of users arriving at this intersection. Whenever a new request arrives the Constraint Satisfier will look at the intersections that are within a given distance around the origin and the destination. For each user that occurs at both searches, around origin and destination, an entry into a bipartite graph is made. The bipartite graph represents all possible matches, consisting of vertices for requesters and offers, and edges that indicate a possible matching between them. The weight of an edge is the length of the shared trip. The Matching Module then considers the bipartite graph and implements an online maximum weight matching, as this matching represents the maximum revenue for the service. In this work the profit of the service is dependent on the length of shared trips.

The advantage of the bipartite graph together with the matching module is that multiple requests can be solved at once. More requests generate more vertices and edges in the bipartite graph which also holds entries from previous requests. The matching module may take up to several seconds for its maximum weight matching but therefore solves all of the matchings at once and even improves matchings made before.

Examples of the work state that after 12 seconds of calculations for the matching algorithm a matching for around 4'000 requests was found. This would mean an amortized duration of 2.75 ms per request. Calculations were executed on a graph representing the city of New York with 61'000 users and an average distance of 16.1 km per trip.

URoad and SHAREK

Two interesting works with rather straight forward approaches following the filter and refine framework are presented by J. Fan et al. [FXH+18] and B. Cao et al. [CAMB15]. The first introduces the "URoad"-algorithm and seems to be built on the latter, which called their algorithm "SHAREK". Both follow the idea of taking the list of all drivers in the system and prune it with various techniques to reduce the amount of shortest path queries needed. URoad was published in 2018, SHAREK in 2015.

URoad The algorithm is divided into three phases: *Departure Time Pruning*, *Euclidean Distance Pruning* and *Road Network Distance Pruning*. The first two phases reduce the list of possible drivers for a request, the last phase then uses the reduced list and finds the best match regarding constraints of detour and price.

The algorithm makes use of two data structures. The first is a time index "*DriverTimeIndex*" for the drivers, which is basically a sorted hash-map with the departure time of a driver as key and the corresponding driver information as value. The second is a grid index, dividing the road network into $m * n$ grid cells allowing for fast approximations of distances. Once a request arrives, all offers in the system are entered into a list of possible matches, afterwards the three phases are run through to determine a match.

1. The **Departure Time Pruning** uses the *DriverTimeIndex* and the requester as input. It traverses the ordered *DriverTimeIndex* in sequence and finds the driver set whose key (departure time) is ahead of the latest departure time of the requester. All other subsequent drivers can't reach the request in time as their trip starts to late.
2. The **Euclidean Distance Pruning** takes the set of drivers pruned before and prunes it further. This time the arrival times of the offers at the requester origin are considered. These times can be approximated (estimated from below) by the euclidean distance between the start of the offer and the start of the request, therefore the grid index is used. For example: An offer starts at 11:00am and the latest departure time of a request is 11:30am. The euclidean distance between the start of the offer and the start of the request is 12 km. With a given average speed of 50km/h the offer can arrive at the requester within 24 minutes and thus before 11:30am. If the requesters latest departure would be at 11:15am the offer couldn't arrive in time and would be pruned.

   If an offer may arrive in time according to the euclidean distance, the pruning further approximates the detours and costs that will be generated. We ignore the costs and describe the procedure on the basis of the detour. Assume that $\mu_{eucl}(o^o, o^r)$ describes the euclidean distance between the origin of the offer o^o and the origin of the request o^r. In the same way $\mu_{eucl}(d^o, d^r)$ is defined for the destinations. Both euclidean distances can be approximated with the grid index. We know the shortest distances $\mu(o^o, d^o)$ and $\mu(o^r, d^r)$. Now we can approximate the detour that would be generated if the offer would be matched with the request using the equation:

   $$\Delta_{approx} = \mu_{eucl}(o^o, o^r) + \mu(o^r, d^r) + \mu_{eucl}(d^r, d^o) - \mu(o^o, d^o) \tag{3.1}$$

   Each offer has a maximum allowed detour Δ_{max} normally denoted by a detour factor. If the approximated detour Δ_{approx} already is bigger than this value Δ_{max} the offer can be pruned. The same can be done for the maximum price a requester is willing to pay as it can be approximated by the same way (we will skip it here).
3. The **Road Network Distance Pruning** is the last phase of the algorithm. Every remaining driver will be checked if he can fulfill the requirements of a match in the real road network without approximation. Therefore the request gets theoretically inserted into the route of the driver with a greedy algorithm. Every drivers route is presented by a sequence of stops, each marking an origin or destination of a requester or the driver himself. First the origin of the new request gets inserted. Therefore every possible position in the sequence is tested and the detour, that would be generated by inserting the new stop, is calculated. For these calculations shortest path queries are invoked (Bidirectional Dijkstra). The position with the smallest detour is stored and dependent on that position the destination of the new request gets inserted into the sequence. Denote that the destination has to be after the origin in the sequence. For each insertion the constraints of allowed detour, price and travel times are checked. This is done with all offers and the offer with the smallest detour generated will finally be matched with the request.

The experimental section of the paper states good and especially fast results for big instances of the ridesharing problem. Subsequently, we implemented this algorithm on our

own and tested it on the same instances we ran our algorithm with. But the running times we achieved differed order of magnitudes from the once stated in the paper. We further investigated this problem and will describe our thoughts and findings in the experimental section. We have chosen this work as reference mainly because it describes a similar algorithm to SHAREK and was released 3 years later, which let us assume that it is an improved version of SHAREK. Later on we had a closer look on SHAREK which shows some other improvements but only solves the single driver single rider ridesharing problem.

SHAREK The algorithm is also divided into three phases, whereas the first two phases basically do the same as the *Departure Time Pruning* and the *Euclidean Distance Pruning* of URoad. We won't describe them again, the only difference is that a request always wants to start his trip directly, so a requester doesn't provide an earliest start time as this time is always the time the request arrives in the system. Therefore we can't schedule the future and instead of considering the origin of an offer for the calculations we always have to consider the current position which may be somewhere on the path between origin and destination of the offer, dependent on the current time. We will denote this position as $currPos(o)$.

The last phase of SHAREK (*Semi-Euclidean Skyline-aware Pruning*) differs though and differentiates between four cases for all remaining offers, further pruning the candidates and reducing the amount of shortest path queries. The definition of "skyline" in this context was not common to us, but it comes closest to the definition of pareto-optimality where one property cannot be further improved without worsening another. In this case the dimensions price and waiting time are optimized.

The *Semi-Euclidean Skyline-aware Pruning* employs an incremental road network nearest-neighbour (INN) algorithm [PZMT03] that retrieves the drivers one by one in increasing order of the distance $\mu(currPos(o), o^r)$ for the pick-up of the requester. The distance $\mu(currPos(o), o^r)$ is computed as part of the INN algorithm. A variable *MAX* is keeping track of the current best price and is initialized by the maximum price a rider is willing to pay. Any offers which would get a price for the rider bigger than *MAX* will be excluded. Finally a table containing all offers not yet removed, called matching table, is sorted with the value of $\mu_{eucl}(d^r, d^o) - \mu(o \cdot d^o)$ as key, which will come in handy later. Afterwards, when going through the drivers in order, one of the four cases occurs for each:

1. **Case:** The driver can't reach the requester in time. The approximation of the time with the euclidean distance may have suggested that the driver can be in time, but as the actual road network may lead to longer rides the exact time needed by the driver may be longer than the approximated one. In this case the algorithm can be terminated completely.

2. **Case:** If the first Case is not fulfilled we test again the approximation of equation 3.1, where $\mu_{eucl}(o^o, o^r)$ gets replaced with the now exact value of $\mu(currPos(o), o^r)$. This gives a more exact estimation and we can test if the new value for Δ_{approx} results in a bigger price than *MAX* (detours and price can be mapped to each other as in the work the price is set to 1\$/km). If the price is bigger than *MAX* another offer has been tested before which lead to a smaller *MAX* value. This information can be used to prune the matching table further. We can now exclude this offer and any other offer from the table with a bigger value for $\mu_{eucl}(d^r, d^o) - \mu(o^o, d^o)$. As this is the key of the sorting, the exclusion is quick.

   The exclusion of these offers is justified, because their estimated detours are already longer than (or in the best case equal to) the real detour of the driver just checked and rejected. This holds also true for their actual detours which cannot be shorter than the approximated ones. Therefore all those entries would lead to higher prices than the current *MAX* and can be ignored.

3. **Case:** The first two cases were not fulfilled, therefore we consider the exact value for the detour and replace any euclidean distance with the real distance (in this case calculated using Bidirectional Dijkstra). If the exact values for the detour and the price that would occur is bigger then the current *MAX* value the currently viewed driver can't take the rider with him, or another driver with better values has been found before. The driver gets dropped from the matching table.

4. **Case:** If the values calculated with the exact distances for the ride beat the value for *MAX* we found a new best driver. We will mark him as current solution and update the value for *MAX* to the value determined for the driver.

Once the matching table is empty the *Semi-Euclidean Skyline-aware Pruning* is finished and can output the best found solution. The applied techniques clearly try to postpone the distance queries as long as possible and focus heavily on pruning.

Unfortunately, it seems that the presented techniques can't be considered for the single driver and multiple rider variant of ridesharing, as the presented "skyline" techniques rely heavily on the fact that an offer only consists of a single origin and a single destination.

3.3.9 Fast Detour Computation for Ridesharing

This work was published in the ATMOS Workshop of 2010 [GLS+10] and builds the base of this thesis together with the concept for fast many-to-many queries presented in [KSS+07]. The work describes an algorithm which not only matches a rider with a driver but also allows drivers to take detours or different routes to make the transportation of a rider possible. Other works, as stated in Section 3, often only match rides if origins and/or destinations of the rider are close to the recently planned route of the driver, as their solutions use techniques like radial search. This fast detour computation algorithm uses a special construct together with CHs for solving one-to-many distance queries. All in all, in this way a much higher matching rate over common simple approaches is achieved whereas the computations even for bigger instances takes less than 50ms.

The algorithm solves a simpler version of the ridesharing problem described in Section 2.2.1. Instead of allowing to match an offer with multiple riders, every offer gets matched only once. The advantage of this behaviour is that the algorithm does not need to track any modification to the route for a given offer. Once a rider got matched to a driver, the offer can be deleted from the system. All the algorithm needs to do is finding the best reasonable match of a request with all existing offers in the system. Another small difference to our work is that the algorithm does not take the daytime into account. It does not matter when a driver or rider starts his trip and when he arrives. The work only focuses on matching the path of the rider with a path of an offer. The last difference to be mentioned is the definition of a reasonable fit. As there are fewer constraints to this problem the definition of a reasonable fit is rather simple: For a given detour factor δ an offer $O = (o, d)$ and a request $R = (o', d')$ form a reasonable fit if there exists a path $P = \langle o, \ldots, o', \ldots, d', \ldots, d \rangle$ in G with $c(P) \leq \mu(o, d) + \delta * \mu(o', d')$. This at first glance somewhat strange definition for a reasonable fit with the detour of the driver dependent on the shortest path of the requester seems to be chosen for later pruning tactics.

The main part of the algorithm is the computation of multiple shortest paths. Given a single request $R = (o', d')$ and a set of n offers $O_i = (o_i, d_i)$ all $2n + 1$ distances $\mu(o_i, o')$, $\mu(d', d_i)$ and $\mu(o', d')$ for $i = 1..n$ are needed. For every offer O_i the detour, caused by inserting the request into it, can than be calculated as $\Delta_i(r) = \mu(o_i, o') + \mu(o', d') + \mu(d', d_i) - \mu(o_i, d_i)$. Figure 3.3 illustrates such a detour. The distances $\mu(o_i, d_i)$ for the offers O_i are calculated as they get added to the system.

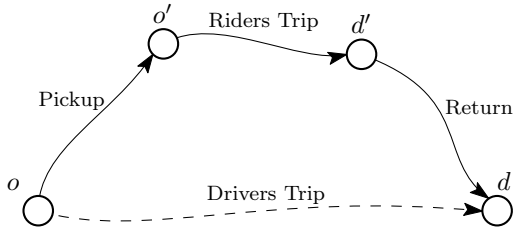


Figure 3.3: Illustration for the altered path of an offer with origin o and destination d to serve a request with origin o' and destination d'.

One-To-Many with Contraction Hierarchies and Buckets

The one-to-many shortest distances are calculated with a combination of the search spaces of the CHs and buckets holding distance entries of the offers. In Section 2.4.2 we introduced the search spaces of CHs, they shall be denoted as a forward search space $S^{\uparrow}(o)$ outgoing from an origin o and as a backward search space $S^{\downarrow}(d)$ outgoing from a destination d. For each vertex inside these search spaces the distances from/towards the origin/destination is known or can be easily calculated. Remember that for the shortest path a vertex is needed which appears in both search spaces. If later on we have to calculate the distance $\mu(o, d)$ for any pair of vertices, we know that there are two paths (in ascending / descending order) $\langle o, \ldots, m\rangle\langle m, \ldots, d\rangle$ that represent the shortest path. As $\langle o, \ldots, m\rangle$ must be in the search space $S^{\uparrow}(o)$ and $\langle m, \ldots, d\rangle$ in the search space $S^{\downarrow}(d)$ we have a fast access to $\mu(o, m)$ and $\mu(m, d)$ and thus at least an upper limit for $\mu(o, d)$. If we search for the optimum across all possible values of m we get the lowest distance which can be found.

First of all a forward bucket $B^{\uparrow}(v)$ is assigned to each vertex v in the graph. As an offer O_i with origin o_i gets added into the system the forward search space $S^{\uparrow}(o_i)$ will be computed. After that, the forward buckets $B^{\uparrow}(v)$ will be enhanced with the entries $(i, \mu(o_i, v))$, for all vertices v occurring in $S^{\uparrow}(o_i)$. In this way the search space gets indirectly stored for later queries.

To compute all distances $\mu(o_i, o')$ of a request afterwards, $S^{\downarrow}(o')$ has to be computed. All buckets of the vertices in $S^{\downarrow}(o')$ are scanned. An array of tentative distances for $\mu(o_i, o')$ is initialized with ∞ and keeps track of the found distances. As the buckets of a vertex v is scanned for each entry the distance $\mu(o_i, o')$ is calculated by adding the stored $\mu(o_i, v)$ and $\mu(v, o')$ retrieved from $S^{\downarrow}(o')$. If a calculated $\mu(o_i, o')$ is smaller than the tentative distance it will replace it. Once all buckets in $S^{\downarrow}(o')$ have been scanned the array of tentative distances holds all the shortest distances $\mu(o_i, o')$ for $i = 1..n$.

The procedure for the distances $\mu(d', d_i)$ is similar. Each vertex v of the graph also gets a backward bucket $B^{\downarrow}(v)$ assigned, holding node-distance-pairs $(i, \mu(v, d_i))$. The remaining distance $\mu(o', d')$ for the detour calculation can also be calculated by using the corresponding search spaces.

Handling of Offers and Constraints

We will have a closer look on how to insert new offers and how to delete offers (e.g. after they have been matched). For adding an offer $O = (o, d)$ the search spaces $S^{\uparrow}(o)$ and $S^{\downarrow}(d)$ have to be calculated and the corresponding entries have to be stored into the forward and backward buckets. Like this the offers are kept in the system indirectly. Therefore if a matching has been found and an offer needs to be deleted again its search spaces have to be calculated for finding the buckets in which the now invalid entries have to be deleted. Buckets are not ordered, therefore a new entry can just be appended at the end, making insertions fast. For deleting entries of a bucket the entries first have to be searched, making

the deletion of an offer much slower than the insertion. Therefore another solution is to just mark deleted offers as inactive and ignore them during matching.

The approach also allows to consider more constraints by extending the use of buckets. For example if some of the offered rides have a smoking restriction and some have not, the buckets can be split up into "smoking" and "non-smoking" buckets indicating if the driver smokes in his car or not. A new "non-smoking" offer would now only be inserted into "non-smoking" forward and backward buckets. If now a request for a "non-smoking" offer is made, the search only considers those "non-smoking" buckets, whereas "smoking" offers and requests only consider the "smoking" buckets.

Most likely all other constrains can be treated this way. This is an important construct which will also be used later in our algorithm to speed up calculations and take the constraints of Section 2.2.1 into account.

Experiments and Discussion

In the original paper also some optimizations for the algorithm are discussed. Due to the definition of a reasonable fit the scanning of forward an backward buckets can be pruned. But as this technique can't be applied to our own algorithm for ridesharing with multiple riders we won't present it here. But one thing to keep in mind is that by reducing the buckets which have to be scanned the runtime can be further improved.

The experiments presented in the paper show that even for bigger instances with a graph of Germany with 6'344'491 vertices, 13'513'085 edges and 100'000 offers in the system a request only takes 4.4ms to 45.1ms to be matched and answered. The difference in running time is caused by varying the maximum detour factor δ_{max} for an offer. If δ_{max} was chosen to be small the requests were answered faster. The choice of δ_{max} had also a strong impact on the fraction of matched requests, which seems logical. If longer detours are allowed drivers can serve also riders which are further away.

4. Ridesharing

Our solution for matching ridesharing requests with existing offers is based on the work of S. Knopp et al. [KSS+07] and the idea of the work presented in Section 3.3.9. Instead of solving the single driver and single rider variant of the problem we extended it towards the single driver and multiple rider variant. We created a version capable of tracking altered tours of an offer for further matching, and allow multiple request to be treated by only one offer. This shall lead to more matches and a more efficient way of ridesharing. We further improved the simulation of a ridesharing service by taking the daytime into account, as we want to test our solution in realistic scenarios over the course of a day. Finally, we added multi-threading to our solution for even faster matching.

Our algorithm has a service like structure allowing offers and requests to be entered arbitrarily. When a new offer arrives we enter it into our data structure thus making it available for matching later on. Whenever a requests arrives the algorithm tries to match it with all existing offers. The matching can be divided into three phases: First, we use a fast many-to-one (one-to-many) query technique to obtain all minimal distances, needed in the following phases. Second, we check for the offers, which minimal detour can be generated by inserting the two new stops (origin and destination of the request) into the route of the offer. We keep track of the offer with the smallest generated sum of detours (detours for the driver and the already matched riders). The last phase finally matches the request with the best fitting offer we found, inserts the two new stops into the route of the offer and updates the data structure of our algorithm.

In the following we will present our algorithm in detail. While reading, keep in mind that in the context of this algorithm a driver and an offer can be seen as synonyms, same counts for passenger and request.

4.1 Data Structures

In Figure 4.1 we present the model of our service. Our approach is a centralized solution, where a server receives all information of drivers and riders and then calculates a matching for them with the data provided. First of all, a driver makes an offer towards the service which then stores the offer into a database realized by a vector and several buckets. When a rider makes a request towards the service all offers in the database are looked up and the best fit is searched. Once the best fit is found, both driver and rider are informed to share the ride and get matched, and the new route for the driver is calculated and updated in the database. For each rider and driver we need to store an object representing them.

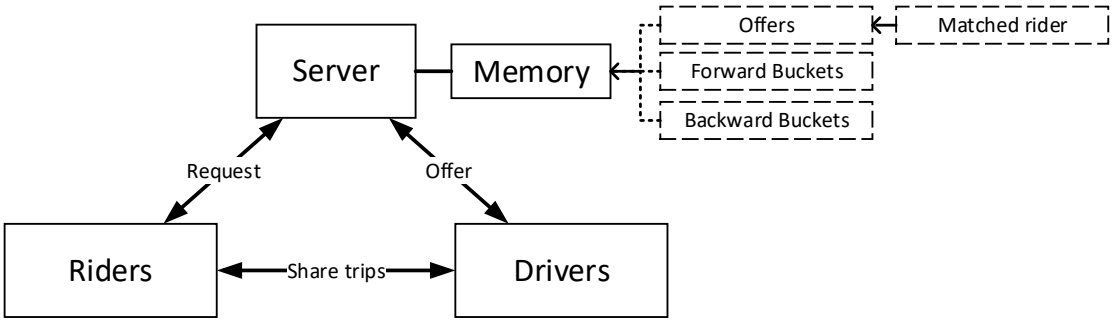


Figure 4.1: The model of our ridesharing service with driver, rider and the server. Rider and driver are interacting with the server by adding offers or making requests. When a request arrives the server tries to match it with the existing entries stored in the memory. The memory mainly stores the offers including the matched passengers as well as forward and backward buckets. If a match is found both rider and driver get informed.

4.1.1 Driver

Once a driver registers an offer he gets stored in the system by adding a driver object into an vector vec_{off}, which is part of the database. Furthermore, buckets of some vertices get additional entries storing information of the offer's trip and certain distances. We will describe the bucket approach in Section 4.2.2.1 and focus on the driver object, representing an offer, and its entries first:

- ***ID*:** A driver always gets referenced by a unique number, his id, which also represents the index of its entry in vec_{off} If a driver gets deleted another vector keeps track of the deleted drivers in vec_{off}. New offers first fill possible gaps in vec_{off}, if no gap exists a new entry gets appended to the vector.
- **Origin o^d:** The vertex representing the origin and start point of the driver.
- **Destination d^d:** The vertex representing the destination of the driver. The driver will only transport riders between his origin and destination. Once the destination is reached the offer gets deleted.
- **Distance $\mu(o^d, d^d)$:** The shortest distance between origin and destination. This value is needed for detour calculations.
- **Earliest Start Time t^d_{es}:** We assume that a driver specifies a (planned) start time, origin o^d and destination d^d of his trip. We also assume that the real start time may become later but not earlier as long as he reaches the destination at an acceptable time. In so far, the planned start time is denoted as earliest start time t^d_{es}. As soon as $\mu(o^d, d^d)$ is calculated, we can derive the earliest arrival time as

$$t^*_{ea} = t^*_{es} + \mu(o^*, d^*) \tag{4.1}$$

  As the driver would accept a detour of $\delta * \mu(o^d, d^d)$ we assume that we wants to arrive latest at

$$t^*_{la} = t^*_{ea} + \delta * \mu(o^*, d^*) = t^*_{es} + \delta * \mu(o^*, d^*) + \mu(o^*, d^*) \tag{4.2}$$

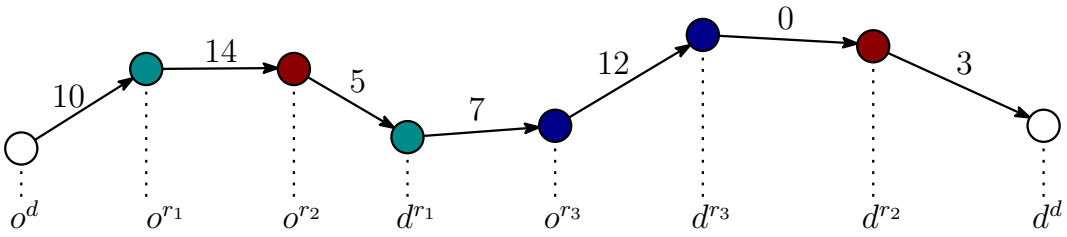
  This finally leads to a latest possible start time, which could be chosen if no detour is required at all

$$t^*_{ls} = t^*_{la} - \mu(o^*, d^*) = t^*_{es} + \delta * \mu(o^*, d^*) \tag{4.3}$$

- **Current Start Time t^d_{cs}:** Every time a rider gets matched with the offer of the driver the detour can also delay the departure time for the driver. Therefore t^d_{cs} keeps track of the time the driver is scheduled to start his trip.

- **Remaining Detour Δ^d_{rem}:** For simplicity each driver stores a value of the remaining detour he is willing to drive. Initially this value is set to value of the maximum detour, $\Delta^d_{rem} = \Delta^d_{max} = \delta * \mu(o^d, d^d)$. Every time the driver picks up a new rider the remaining detour will be updated and reduced. This value is of importance for the validity check of a matching.
- **Maximum Passengers p_{max}:** Each car can only fit a certain amount of riders, this constraint gets modelled by the value p_{max}. Also with this value a driver can set his own preferences towards the amount of riders in his car. If not denoted otherwise we will assume that $p_{max} = 3$.
- **Passengers $P[\cdot]$:** All matched passengers get stored as a passenger object in the passenger vector $P[\cdot]$. Each passenger has a unique id *subID* among the passengers of the same offer. The values stored by a passenger object are presented in the next section.
- **Passenger Order $P_{order}[\cdot]$:** A vector storing the order of pick-ups and drop-offs of the passengers in an offer. The passenger order $P_{order}[\cdot]$ has $n-2$ entries for a path with n stops, two entries (origin and destination) for each passenger. The origin and destination of the driver are not represented in this vector.

  Each passenger of an offer has a unique *subID*. For each stop we store the *subID* of the passenger of that stop into the corresponding position of $P_{order}[\cdot]$. To distinguish between a pick-up and a drop-off we simply store the positive or the negative value of the *subID*, respectively. Positive values representing pick-ups and negative values drop-offs. Therefore we always know, which stop belongs to which passenger, and as the passenger object holds his origin and destination we can also reference towards the exact vertex of each stop. A sample trip of an offer is shown in Figure 4.2. Each vertex in the path refers to a stop in the trip, either to a pick-up or a drop-off of a passenger or to the origin/destination of the driver. In the example given, the driver first picks up passenger 1 then passenger 2, next drops off passenger 1, picks up and drops off passenger 3 and finally drops off passenger 2. The very same sequence is stored in the passenger order $P_{order}[\cdot]$. Initially, this vector is empty as the offered ride is not matched with any passengers.



Passenger Order $P_{order}[\cdot]$:

1	2	-1	3	-3	-2

Step Distance $D^d_{step}[\cdot]$:

10	14	5	7	12	0	3

Figure 4.2: An example of a scheduled trip of an offer and the two vectors representing this trip. Origin o^{r_i} and destination d^{r_i} describe the two stops of rider i, Origin o^d and destination d^d belong to the driver of the offer. Remark that drop-offs or pick-ups which are on the same vertex still have entries within the two vectors representing the trip, like d^{r_3} and d^{r_2}.

- **Step Distance $D^d_{step}[\cdot]$:** As seen in Figure 4.2 for the representation of a trip we also need to store the distances of the steps from one stop towards the next. This vector also stores the distances outgoing from the origin of the driver and towards the destination of the same and thus holds one more entry than $P_{order}[\cdot]$. Initially, this vector stores only one value, the distance between the origin and destination of the driver. Each time a passenger gets added to the route, the step distances have to be updated.
- **Active Flag:** Finally, for lazily deleting offers, they can be marked as inactive. Thus the deletion of an offer can be delayed by simply deactivating it first. If an offer is deactivated it won't be considered during matching.

4.1.2 Rider

As long as a rider (passenger) is not matched towards an offer it is only referenced by the three to four values provided by the request: *The vertex of the origin* o^r, *the vertex of the destination* d^r, *the earliest start time* t^r_{es} for his trip and sometimes the detour factor δ. As varying detour factors δ make the interpretation of the results harder we normally consider a globally fixed detour factor.

Only when a rider gets matched with an offer he gets stored in the system. We create a passenger object and store it within the passenger vector $P[\cdot]$ of the matched driver. Accordingly, to access a passenger we have to access the driver first. The passenger object stores the following information:

- ***SubID*:** The id of the rider among the passengers of the same offer, starting from 1. If a driver has for example 4 passengers there will always be a passenger with *subID* 1, 2, 3 and 4. The *SubID* can't be 0 as we take advantage of switching between the positive *subID* and the negative value of it as described before (for $P_{order}[\cdot]$).
- **Origin o^r:** The vertex representing the origin of the rider.
- **Destination d^r:** The vertex representing the destination of the rider.
- **Distance $\mu(o^r, d^r)$:** The shortest distance between origin and destination.
- **Earliest Start Time t^r_{es}:** The earliest start time the rider is willing to start his trip. Together with the value of the shortest distance $\mu(o^r, d^r)$ and the maximum detour factor d_{max}, time periods for the departure $[t^r_{es}, t^r_{ls}]$ and arrival $[t^r_{ea}, t^r_{la}]$ can be calculated with Equations 4.1, 4.2 and 4.3.
- **Current Start Time t^r_{cs}:** The current start time of a rider lies between t^r_{es} and t^r_{ls}. Every detour for the matched offer can lead to detours for the matched riders and thus can also delay the starting time. Whenever a detour for the offer changes the current start time t^r_{cs} is updated.
- **Remaining Detour Δ^r_{rem}:** This variable keeps track of the detour for the passenger. Once the passenger is matched with an offer the detour Δ generated for the rider is calculated. A detour may occur by riding with an offer, which picks up and drops off other passengers as well. Together with the maximum possible detour for the rider $\Delta^r_{max} = \delta * \mu(o^r, d^r) + \mu(o^r, d^r)$ the remaining detour $\Delta^r_{rem} = \Delta^r_{max} - \Delta$ is then calculated as the maximum detour minus the already existing detour for the passenger by the trip of the driver. Later on, if further requests arrive this value serves for checking the validity of a matching, as the remaining detour must always be greater or equal to zero.

The deletion of riders was not of interest for us, therefore we do not provide such functionality. But it could be added by an vector storing an entry for each rider pointing towards the matched offer, which then can provide all information needed for a deletion.

4.2 Algorithm

The algorithm can be split into two main parts already presented in Figure 4.1, *adding offers* and *making requests*. Adding offers is straight forward as the matching only occurs after a request is made. New offers have to be stored in the database and buckets have to be filled accordingly. The request is much more complex as we try to find the best fitting offer and insert the new passenger into its ride. We will regard both procedures on its own as they can be called separately multiple times in random order. We start with adding offers.

4.2.1 Adding Offers

Adding an offer describes the concept of adding a new driver into the system who is willing to take additional riders with him during his trip. A driver needs to state three to five values for the creation of an offer:

1. **Origin o^d** of the driver.
2. **Destination d^d** of the driver.
3. **Earliest starting time t^d_{es}.**
4. **(Maximum detour factor δ):** This value is optional and most of the time a fixed standard value of 0.5., pre-set by us, is chosen as δ.
5. **(Maximum passengers p_{max}):** This value is optional an most of the time a fixed standard value of 3, pre-set by us, is chosen as p_{max}.

First of all the minimal distance $\mu(o^d, d^d)$ is obtained by a CH-query as we need this value for calculating the maximum allowed detour and checking if a valid path between origin o^d and destination d^d exists. If no explicit detour factor is stated we choose the global detour factor of 0.5 (We will discuss the influence of the detour factor later in experimental section). For making things easier we further calculate the latest starting time t^d_{ls}, earliest arrival time t^d_{ea} and latest arrival time t^d_{la} according to Equations (4.1, 4.2, 4.3) and store them. We also obtain the unique *ID* for the offer, therefore we first check a vector of beforehand deleted ids. If such a deleted id exists we reassign that id towards the new offer, else we create a new id by increasing the highest existing id by one.

After obtaining all the needed values we have to create a new offer and fill the buckets $B[\cdot]$. The buckets that get a new entry are determined by the CH search spaces outgoing from the origin and destination of the offer.

For the origin o^d we consider the upward graph of the Contraction Hierarchy, therefore we visit each vertex in the forward search space $S^{\uparrow}(o^d)$. This can be done by a Dijkstra search where no explicit target is specified, we run the algorithm following edges, going from vertices with lower rank towards vertices with higher rank, until the whole search space is visited. As a reminder, search spaces have already been illustrated in Figure 2.5. For every visited vertex v_i the distance $\mu(o^d, v_i)$ is provided by the Dijkstra search. This distance gets stored together with the unique id of the offer and a *subID* inside the **forward bucket $B^{\uparrow}[v_i]$** of vertex v_i. Thus the forward buckets store triplets of ***ID***, ***subID*** and **distance**, outgoing from the origins of the offers, where the *subID* is of no importance yet and initialized with the value of zero. Based on the search space $S^{\downarrow}(d^d)$ we do analogue calculations and create entries for the **backward buckets $B^{\downarrow}[v_i]$**.

We would also like to point out that we neither need the backward search of the origin nor the forward search of the destination as the trip of the driver can't manage stops before his own origin or after reaching his destination.

Also, this describes only the simple version of the buckets which are not optimized towards a ridesharing system taking the specific times of an offer or request into account. A more detailed description of the buckets will be provided in Section 4.2.2.1, which will also describe the use of the calculated times in the initialization.

After filling the buckets the algorithm enters a new offer object into the offer vector $vec_{off}[\cdot]$, whereas the *ID* of the offer also represents the index in the vector. A pseudo-code for adding offers is given with algorithm 4.1.

Algorithm 4.1: ADDING AN OFFER

Input: Origin o^d, Destination d^d, Earliest Starting Time t^d_{es}, detour factor δ, max passengers p_{max}

Data: Vector of offers vec_{off}, search spaces $S^{\uparrow}(\cdot)$ and $S^{\downarrow}(\cdot)$, detour factor δ

```
// Initialization
```

1 *distance* $\mu(o^d, d^d) \leftarrow CHquery(o^d, d^d)$

2 *maxDetour* $\Delta^d_{max} \leftarrow \delta * \mu(o^d, d^d)$

3 *earliest arrival* $t^d_{ea} \leftarrow t^d_{es} + \mu(o^d, d^d)$

4 *latest start* $t^d_{ls} \leftarrow t^d_{es} + \Delta^d_{max}$

5 *latest arrival* $t^d_{la} \leftarrow t^d_{ls} + \mu(o^d, d^d)$

6 $offerID \leftarrow obtainUniqueID()$

```
// Fill buckets
```

7 **foreach** $v \in S^{\uparrow}(o^d)$ **do**

8 **if** $dist(o^d, v) \leq \mu(o^d, d^d) + \Delta^d_{max}$ **then**

9 $B^{\uparrow}[v] := B^{\uparrow}[v] \cup \{(offerID, subID = 0, dist(o^d, v))\}$

10 **foreach** $v \in S^{\downarrow}(d^d)$ **do**

11 **if** $dist(v, d^d) \leq \mu(o^d, d^d) + \Delta^d_{max}$ **then**

12 s $B^{\downarrow}[v] := B^{\downarrow}[v] \cup \{(offerID, subID = 0, dist(v, d^d))\}$

```
// Save offer object
```

13 $vec_{off}[offerID] :=$ `offerObject`$(offerID, \dots)$

Improvement

Later on, scanning the buckets and their entries will be the bottleneck for our calculations. Thus we want to maintain as few entries in the buckets as possible while keeping the solutions correct. Therefore some of the entries can be pruned: each offer has a maximum detour Δ^d_{max} and with it a maximum driven distance $maxDist = \mu(o^d, d^d) + \Delta^d_{max}$. As the search spaces also indicates a distance towards or from a vertex we can skip any bucket entries if the distance is bigger than $maxDist$. Any path that could later be found with these bucket entries would lead to detours bigger then the possible Δ^d_{max} of the driver. Thus we don't consider the whole search spaces, but only a part of it. This pruning can be found in line 8 and 11 of the algorithm 4.1. Figure 4.3 illustrates the pruning of bucket entries by indicating the space within which bucket entries are made.

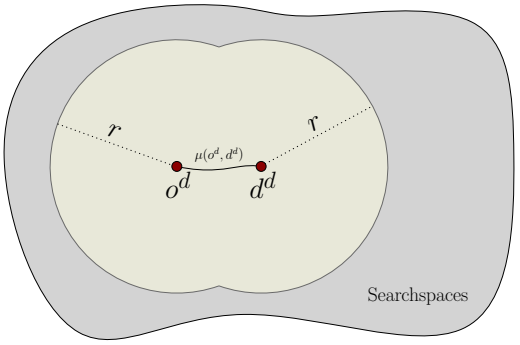


Figure 4.3: Indicating the pruning of the search space for bucket entries. Only the buckets of vertices inside the the circles around origin o^d and destination d^d with radius $r = \mu(o^d, d^d) + \Delta^d_{max}$ will get new entries. Any vertex outside is not within the reach of the offer.

4.2.2 Making Requests

A request describes a desired ride of a potential passenger. We need three values to handle a new request: the **origin o^r** of the request, the **destination d^r** of the request and the **earliest starting time t^r_{es}** of the passenger.

As seen in the related work (Section 3.3) many other algorithms also take a maximum waiting time wt_{max} of a passenger into account, which describes the maximum time a passenger is willing to wait until he starts his ride, independent of the time he loses later on while driving detours. Therefore the latest starting time would be $t^r_{ls} = t^r_{es} + wt_{max}$. But from our point of view we can summarize the two data *waiting time* and *time for detours*.

For example, if the fastest path of the passenger would take 30 minutes and the maximum allowed detour would be 15 minutes it doesn't matter where or how the detour of 15 minutes is spent. So the allowed detour or parts of it could also count as waiting time, means that e.g. 5 minutes waiting time plus 10 minutes detour is as acceptable as 15 minutes detour. Therefore in our solution, $wt_{max} = \Delta_{max}$. We just have to consider that the actual time for driven detours will decrease by the amount of waiting time used.

We first determine the distance $\mu(o^r, d^r)$ with a CH-query, by this we can also calculate the maximum allowed detour Δ^r_{max} for the requester. We consider a fixed detour factor δ (e.g. 0.5) for all passengers in our algorithm and calculate $\Delta^r_{max} = \delta * \mu(o^r, d^r)$. It would be possible to let each passenger determine his own δ but varying values would make the interpretation of the results harder. Another possible solution would be to let each passenger specify his own earliest starting time t^r_{es} and latest arrival time t^r_{la}, which would imply a detour factor $\delta = \frac{t^r_{la} - (t^r_{es} + \mu(o^r, d^r))}{\mu(o^r, d^r)}$. With the values of the shortest distance $\mu(o^r, d^r)$ (travel time), the earliest starting time t^r_{es} and the detour factor δ the time periods for the travel can be again calculated with Equations 4.1, 4.2 and 4.3.

The further procedure can be divided into three steps. First we want to *calculate all the minimal distances* between stops of the offers in the system and the origin and destination of the request. Therefore, we utilize the buckets we filled before, more exactly we use an improved version of them which are adapted to differentiate between the time of the day. Second, we use the shortest distances we got from the first step and *search the best offer to be matched*. Among all offers which would be able to pick up the rider, we chose the one with the smallest summed up detour, where the generated detours for the driver and every rider are considered. And with the final step we *insert the new rider into the route* of the best fitting match and update our data structure.

4.2.2.1 Finding minimal distances

For a given offer we want to check whether it can serve the request. The offer may has $n+2$ stops so far (origin, potentially pick-ups and drop-offs, destination). We could change the route of the offer to reach o^r after any stop $k < n+2$ and have to return later on to stop $k+1$. Here we can distinguish two cases: First, the new rider drops off before stop $k+1$ is reached (we leave the former route only once). Second, the new rider drops off after stop $k+1$ (we leave the former route twice, one time for the pick-up and later on a second time for the drop-off of the new rider).

That means, that we need for the overall distance calculations these distances:

1. Distances **from all stops** (but the destinations) of the offers **towards** the **origin** o^r of the request
2. Distances **towards all stops** (but the origins) of the offers **outgoing from** the **origin** o^r
3. Distances **from all stops** (but the destinations) of the offers **towards** the **destination** d^r of the requests
4. Distances **towards all stops** (but the origins) of the offers **outgoing from** the **destination** d^r

(1, 4) are required for the both cases mentioned before, (2, 3) only for the second case. All in all, the four different types have to be calculated by four different one-to-many (many-to-one) queries.

Figure 4.4 illustrates the distances needed by the one-to-many (many-to-one) queries for an example offer. If we look at this closer, we see that we only need two distances involving the origin and destination of the driver no matter how the route in between runs, these two distances are $\mu(o^d, o^r)$ and $\mu(d^r, d^d)$. This knowledge can be useful during the assignment of the distances.

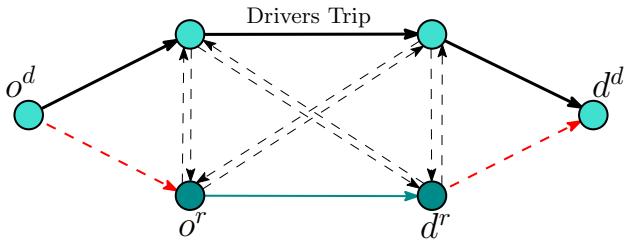


Figure 4.4: Example of the distances required from the one-to-many (many-to-one) queries for an example offer (light turquoise) and request (dark cyan). Contemplating the distances needed between orgigin/destination of a request and an offer, the only two distances needed between these pairs are marked red.

For the calculation of the required distances we will use the procedure as described before in the work of [GLS+10] using search spaces of Contraction Hierarchies. Obviously we have to calculate the forward and backward search spaces $S^{\uparrow}(\cdot)$ and $S^{\downarrow}(\cdot)$ for o^r and d^r. But the search spaces for all the vertices s_i that represent stops on a drivers tour, have already been calculated. This has been done when the offer came in (for o^d and d^d) and when the requests had been matched for the intermediate stops. These search spaces are stored implicitly in the buckets of their vertices together with distance informations.

Let's assume we want to calculate $\mu(s, o^r)$ now, for an arbitrary stop s. We know that the shortest path $\langle s, \ldots, o^r \rangle$ can be combined as $\langle s, \ldots, v \rangle \langle v, \ldots, o^r \rangle$ where the vertices in $\langle s, \ldots, v \rangle$ are ascending in ranking and descending in $\langle v, \ldots, o^r \rangle$. To find the optimal

vertex v_{opt}, we scan the backward search space $S^{\downarrow}(o^r)$. For every vertex v in the search space we obtain easily the distance $\mu(v, o^r)$. We have to find out the missing distance $\mu(s, v)$. But this distance is stored in the forward bucket of v. Thus we can calculate the values $\mu(s, v) + \mu(v, o^r)$ for all possible v. Let us denote the one with the lowest sum with v_{opt}, then the result is $\mu(s, o^r) = \mu(s, v_{opt}) + \mu(v_{opt}, o^r)$.

This is a description of what has to be done for one pair (s, o^r). In the end, we need to do it for all pairs (s, o^r), where s is any vertex representing a stop. This is done simultaneously for all s as indicated in Algorithm 4.2. Reading the algorithm keep in mind, that any stop s is identified uniquely by the pair $(ID, subID)$.

Algorithm 4.2: FINDING MINIMAL DISTANCES

Input: search space $S^{\downarrow}(o^r)$
Data: forward bucket $B^{\uparrow}[\cdot]$, bucket entry $b \in B^{\uparrow}[\cdot]$

```
// Initialization
1 foreach active ID do
2     foreach subID ∈ vec_off[ID] do
3         tentDist↓_{o^r}[ID][subID] ← ∞
4 foreach v ∈ S↓(o^r) do
5     foreach b ∈ B↑[v] do
6         ID, subID, dist ← b
7         if tentDist↓_{o^r}[ID][subID] > dist + dist(v, o^r) then
8             tentDist↓_{o^r}[ID][subID] := dist + dist(v, o^r)
```

The same procedure is done for the distances outgoing from o^r, outgoing from d^r and incoming towards d^r, representing the other three queries. This results in four different one-to-many (many-to-one) shortest path queries.

For two of these cases a special pruning can be applied. If we consider paths from o^r towards d^r that go over stops of drivers we directly can say that the path is invalid if the distance is greater then $\mu(o^r, d^r) + \Delta^r_{max}$. The detour constraint of the requester would not be fulfilled. This is used to prune the forward search space from o^r and backward search space from d^r. If the currently looked at vertex v in the search space is further away than this distance $\mu(o^r, d^r) + \Delta^r_{max}$ we can skip its bucket scans as any path over v would be to long.

Considering Time: Daytime Buckets

We just described how minimal distances can be obtained in our algorithm. But as we take the desired time periods for a trip into account we can further improve the buckets used. It should be clear that fewer bucket entries, which have to be checked, lead to a faster algorithm. We also expect that finding all the shortest distances is one of the bottlenecks of our algorithm. Thus pruning bucket scans is highly important.

Regarding the actual timings of offers and riders the following holds: If a rider wants to start his ride in a certain time period we can determine for every vertex in the graph a time period, within which an offer must be at that vertex, so that he is able to reach the rider in time. For example, if the rider wants to start between 10:00am and 10:15am at vertex o^r and we look at a vertex v in our search space which has a distance $\mu(v, o^r)$ of 20 minutes. Then we know that a offer riding over v must be there in the time period between 9:40am and 9:55am, else he can't reach the rider in time.

This leads to the daytime buckets, where we split the bucket of each vertex into multiple buckets, each representing a time period of the day. In our algorithm, we normally use 24 daytime buckets per original bucket, dividing the 24 hours of a day into intervals of an hour each. With the earliest and latest start time of an offer (a rider) plus the distance towards a vertex v we can determine the time periods when they can be at v. We now use this to make insertions into the correct daytime bucket of the vertex.

An offer that could reach v in the time period between 9:10am and 9:50am would be inserted into the bucket with index 9, representing the time period between 9:00am and 9:59am. Later on, while scanning the buckets, for finding the minimal distances, we only consider a smaller set of offers, which have entries in the correct daytime buckets of the vertices. Theoretically this could lead to 1/24 of bucket entries to be scanned, if the timings of the rides would be uniformly distributed and there would be no overlapping.

Overlapping is the main reason for why we don't want to divide the day into too many parts. If a time period of an offer (a rider) at a certain vertex overlaps multiple daytime buckets it has to be entered into each. For example, consider an offer that can reach v between 9:55am and 11:05am. In this case we would add an entry into buckets 9, 10 and 11. Same counts for scanning the buckets later on, if the time period we have to scan overlaps multiple daytime buckets we have to scan each of them completely. Therefore, dividing the day into more parts would lead to a more fine grained differentiation of time periods and buckets with fewer entries but also to a big amount of redundant data which has to be scanned, slowing down the algorithm again. We considered how to determine a good amount of daytime buckets and present our findings in the experimental Section 6. For now we will consider 24 daytime buckets per bucket.

In Figure 4.5 we illustrated a bucket search for the request of the example above. The request demands an offer to be at vertex v between 9:40am and 9:55am. We indicated the time periods of the offers and which buckets are touched. As the request only lies inside the 9:00am to 9:59am bucket we only have to consider entries inside this bucket. As the graphic indicates offers 1,3 and 5 have an entry inside this bucket, which will be scanned. Offers 2, 4 and 6 won't be considered and thus we don't have to calculate any distances for them.

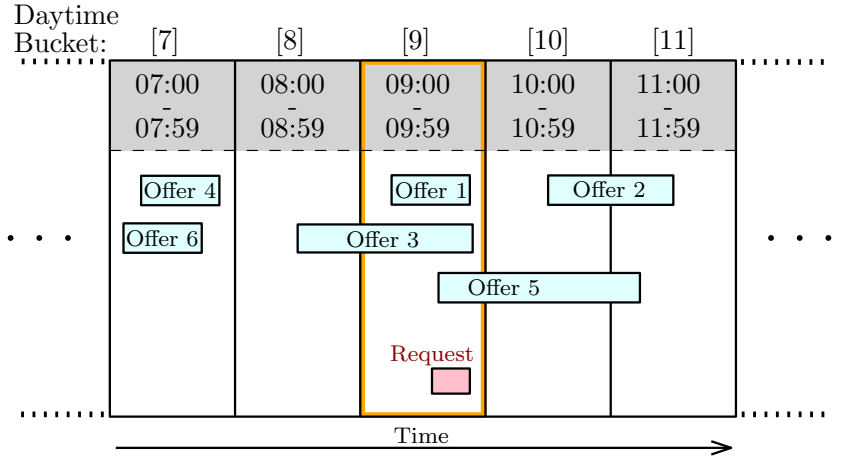


Figure 4.5: Example of a cut of a daytime bucket.

This approach may also benefit from the use of interval trees presented in [CLRS09] by Cormen et al., which also solve the problem of finding entries within a certain time period. One benefit of these interval trees is that they don't store entries multiple times if they overlap multiple buckets. Therefore finding entries always has to traverse the tree structure. We did not have time to implement this solution for daytime buckets but we think that it is quite possible that interval trees can bring further improvement. In any case interval trees would consume less memory.

4.2.2.2 Finding the best fit

After all shortest distances have been obtained, we proceed with finding the best match between drivers and rider. Our solution finds the best fit under the assumption that the sequence of stops in a route of a driver is fixed in its order. We only consider inserting the two new stops into the route, not rearranging the route completely. The global best fit may not be found, but searching it is NP-hard as stated in the related work Section 3.2 and for large routes our approach is still CPU-intensive.

We solve the problem straight forward, we consider every offer that is active and call a *getBestDetourForOffer* function for each offer, which then checks where to insert the two new stops into the route to achieve the minimal detour for that specific offer. We compare the detour of that solution with the current minimum found, and if it's smaller we save the new minimal detour together with the offer that achieves this detour plus the information we need later on to insert the two new stops into the correct position of the route. Once every offer has been looked at we know which offer achieves the smallest detour and can thus match the new rider and alter the route of the offer. For maintaining a fair matching for driver, rider and passengers, we consider the sum of all detours generated by altering the routes, this includes the detour for the driver as well as the detours for every passenger and the detour for the new requester.

We later on improved this procedure by keeping track of the offers which showed up during the minimum distance search. We then only consider those offers which are active and have not been skipped for the distance calculation as else their tentative distance is still set to "infinite" and a matching with the request is impossible anyway.

Algorithm 4.3 presents the *getBestDetourForOffer* function. First we have to consider two cases how a new request can be inserted into a route, illustrated in Figure 4.6 and Figure 4.7. The first describes the case where the request is picked up and afterwards directly dropped off. The second describes the case where the requester is picked up, afterwards one or more stops of the original route are visited and then the requester will be dropped off. We treat both cases one after another.

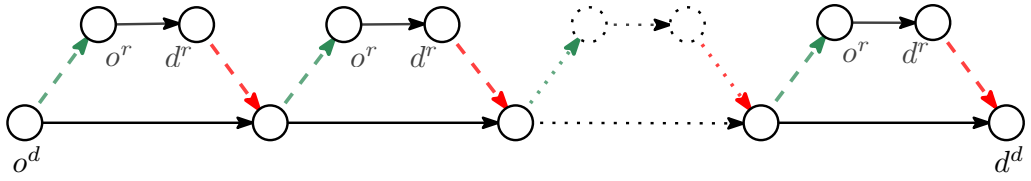


Figure 4.6: Possible positions for the first insertion case. Green denotes a pick-up, red a return.

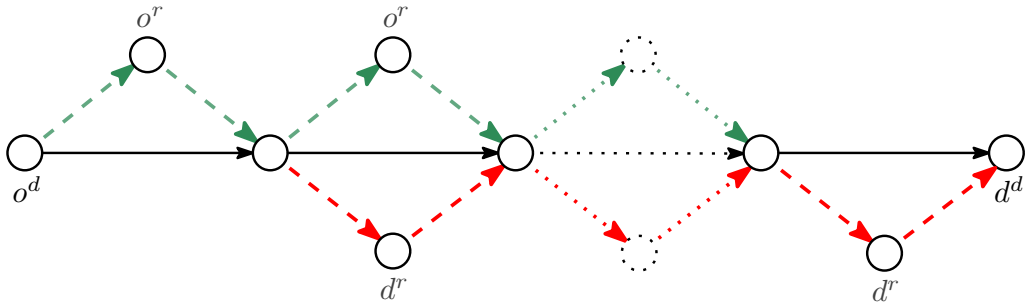


Figure 4.7: Possible positions for the second insertion case. Green denotes a pick-up, red a drop-off. Remark that a drop-off can only occur after the pick-up, and that one or more stops lie between pick-up and drop-off.

Algorithm 4.3: Best Detour For An Offer

Input: Offer ID, o^r, d^r, t^r_{es}, all required $\mu(\cdot,\cdot)$
Data: route R with stops $s_i \in R$ $(i = 1 \dots n)$, $t(s_i)$ time the offer will be at step s_i

1 $minDetour \leftarrow \infty$

`// First Case`
2 **for** $i = 1 \dots n-1$ **do**
3 $\quad subID \leftarrow abs(s_i)$
4 $\quad \boldsymbol{\Delta_{driven}} \leftarrow \mu(s_i, o^r) + \mu(o^r, d^r) + \mu(d^r, s_{i+1}) - \mu(s_i, s_{i+1})$
5 $\quad t^r_{cs} \leftarrow t(s_i) + \mu(s_i, o^r)$ `// calculated start time`
6 $\quad$ **if** $t^r_{es} < t^r_{cs}$ **then**
7 $\quad\quad \boldsymbol{\Delta^r_{wait}}[i] \leftarrow t^r_{cs} - t^r_{es}$
8 $\quad$ **else**
9 $\quad\quad \boldsymbol{\Delta^d_{wait}}[i] \leftarrow t^r_{es} - t^r_{cs}$
10 $\quad$ checkSeatContraint()
11 $\quad \Delta_{sum} \leftarrow 0$
12 $\quad$ **foreach** *rider* r_i *of offer* ID **do**
13 $\quad\quad \Delta^{r_i}_{inst} \leftarrow$ calcDetourForRider$(r_i, \Delta_{driven}, \Delta^r_{wait}[i], \Delta^d_{wait}[i])$
14 $\quad\quad$ **if** $\Delta^{r_i}_{inst} > \Delta^{r_i}_{rem}$ **then**
15 $\quad\quad\quad$ Instance is not valid $\Rightarrow$ check next instance (i++)
16 $\quad\quad$ **else**
17 $\quad\quad\quad \Delta_{sum}$ += $\Delta^{r_i}_{inst}$
18 $\quad \Delta^d_{inst} \leftarrow \Delta_{driven} + \Delta^d_{wait}[i]$
19 $\quad$ **if** $\Delta^d_{inst} < \Delta^d_{rem}$ **then**
20 $\quad\quad \Delta_{sum}$ += Δ^d_{inst}
21 $\quad\quad$ **if** $\Delta_{sum} < minDetour$ **then**
22 $\quad\quad\quad minDetour \leftarrow \Delta_{sum}$

`// Second Case`
23 **for** $i = 1 \dots n-2$ **do**
24 $\quad detourForOrigAtStep[i] \leftarrow calculateDetour(s_i, o^r, s_{i+1})$
25 **for** $i = 2 \dots n-1$ **do**
26 $\quad detourForDestAtStep[i] \leftarrow calculateDetour(s_i, d^r, s_{i+1})$
27 **for** $i = 1 \dots n-2$ **do**
28 $\quad$ **for** $j = i+1 \dots n-1$ **do**
29 $\quad\quad$ **if** $(\mu(s_i, s_{i+1}) > 0)$ *and* $(\mu(s_j, s_{j+1}) > 0)$ **then**
30 $\quad\quad\quad$ checkSeatConstraint()
31 $\quad\quad\quad \Delta_{driven} \leftarrow detourForOrigAtStep[i] + detourForDestAtStep[i]$
32 $\quad\quad\quad \Delta_{sum} \leftarrow$ checkInstance$(\Delta_{driven}, \Delta^r_{wait}[i], \Delta^d_{wait}[i])$ `// like line 11-20`
33 $\quad\quad\quad$ checkDetourForRider()
34 $\quad\quad\quad$ **if** $\Delta_{sum} < minDetour$ *and Instance is valid* **then**
35 $\quad\quad\quad\quad minDetour \leftarrow \Delta_{sum}$

36 return $minDetour$ `// actually also information for insertion`

In order to describe the procedure more easily in Algorithm 4.3 we will work with an imaginary route R consisting of stops s_i $(i = 1 \dots n)$ also including origin and destination of the offer as stops, even though originally this is modelled with the Passenger Order $P_{order}[\cdot]$ and *subIDs*. The absolute value of s_i represents the *subID* of the rider (positive values are stored for pick-ups, negative for drop-offs). The function $t(s_i)$ provides the time when the driver would reach the stop s_i within a route R.

For our two cases the detour generated can be divided into four types of detours:

1. **Actual driven detour for the driver Δ^d_{driven}:** Can be calculated by taking the length of the new route minus the length of the old route. More exactly we don't need to look at the whole route we need only to take the parts that differ. Say that o^r and d^r are origin and destination of the request, further s_i and s_{i+1} are the consecutive stops in the route in between which the request should be fit. Then the detour is

$$\Delta^d_{driven} = \mu(s_i, o^r) + \mu(o^r, d^r) + \mu(d^r, s_{i+1}) - \mu(s_i, s_{i+1})$$

   This detour affects the driver and all passengers who have not completed their ride before the pick-up of the new rider.

2. **Waiting time for the driver Δ^d_{wait}:** If a driver would arrive to early at the origin of the requester it is possible for him to wait there for the requester. This counts as "detour" as well, but is very cost intensive as all passengers will delay their ride by the same waiting time. We can calculate it as

$$\Delta^d_{wait} = t^r_{es} - t^d_{arrival}$$

   There are two possible ways to deal with this waiting time: First, the driver delays the start of his own trip. Second, the driver waits at the pick-up of the new request. We implemented the first, although it has bigger effect on the detour sum, but we don't want an already started ride to pause at a certain point. The driver should be able to drive without times where he has to sit and wait in his car.

   This detour affects the driver and all passengers as the current start time of the driver will be delayed by this value.

3. **Waiting time for the requester Δ^r_{wait}:** If the requester can't be picked up at his earliest starting time and has to wait a certain time, until the driver can pick him up, this also counts as "detour". This time is the difference between the earliest time a driver can arrive $t^d_{arrival}$ at the requester minus the earliest starting time t^r_{es} of the requester:

$$\Delta^r_{wait} = t^d_{arrival} - t^r_{es}$$

   This detour only affects the requester as the schedule for anyone else is not affected.

4. **Actual driven detour for the rider Δ^r_{driven}:** This detour is only relevant for the second case, where the two stops of the request are not consecutive. In this case it is possible that the generated route for the rider, by matching with an offer, would lead to a detour for him. Assume the rider is picked up at o^r, afterwards the driver returns on his route to s_i, stays on the original route until stop s_{i+k} and then drops off the rider at d^r. The resulting detour for the rider would be:

$$\Delta^r_{driven} = \mu(o^r, s_i) + \mu(s_i, s_{i+1}) + \dots + \mu(s_{i+k-1}, s_{i+k}) + \mu(s_{i+k}, d^r) - \mu(o^r, d^r)$$

   This detour only affects the requester.

After initialization we check all possible instances of the first case for an offer and its route. A route with n stops, including the stops of the driver, has $n-1$ instances of the first case, where the new rider gets inserted in between two consecutive stops. We check them in sequence and calculate the detour generated.

With the three detour values for the first case (1-3) we can now check the detour constraints for the driver and each passenger. If the detour would get too large for any of them, the instance is not valid and thus discarded. If the detour is within the limit for everyone and the sum of detours generated (the sum of all values that would be subtracted from the remaining detour values) is smaller than the current minimum found, we store this solution as the current best.

A special case occurs if the two consecutive stops in the passenger order of the offer rely on the same vertex (e.g. a pick-up and a drop-off at the same point). Then the step distance is zero and this position is skipped, as it makes no sense to insert the stops in a way that we have to visit a single vertex twice.

If every instance of the first case is checked, we proceed with the second. First we want to note that we can further use the values for the waiting times of the driver Δ^d_{wait} and requester Δ^r_{wait} as they only depend on the position where the origin o^r of the requester is inserted into the route. In the first case we calculated these values for all possible positions of the origin o^r.

Since the second case demands at least one stop of the route to be positioned between o^r and d^r, and o^r will be inserted first, we have $\frac{(n-1)*(n-2)}{2}$ possible combinations if n is the number of existing stops. Before we check all these combinations, we calculate the detours for each pick-up and drop-off separately as follows.

For each possible position i of o^r between two consecutive stops s_i and s_{i+1} of the route we calculate the detour $\Delta^{origin}_{driven}(i) = \mu(p_i, o^r) + \mu(o^r, s_{i+1}) - \mu(p_i, s_{i+1})$. All distances needed are given either through the calculation with the bucket scans at the beginning or are saved within the step distance vector of the offer. The same is done for the detour $\Delta^{destination}_{driven}(i)$ with possible positions i where the destination d^r can be entered into the existing route.

Now all data are ready for calculating the detour of each of the $\frac{(n-1)*(n-2)}{2}$ cases. We first choose a position i for o^r and afterwards a position j for d^r, where the actual step distance of the existing route and chosen positions is not equal to zero. Otherwise, we would construct routes which visit a vertex twice. Afterwards we can calculate the actual driven detour for the driver $\Delta^d_{driven} = \Delta^{origin}_{driven}(i) + \Delta^{destination}_{driven}(j)$ and the actual driven detour for the rider Δ^r_{driven} (described in the enumeration before). With the values for the four types of detours we now again check the detour constraints of the driver and every passenger and sum up the the detours. We also check if the passenger amount constraint is not violated. We can skip this check if the amount of passengers in the car never reached the maximum allowed value so far. If all these data indicate a new fit, with smaller detour than found before, we save it for this offer.

If all combinations have been checked we output the minimal detour for the offer plus the information needed later on to actually insert the new passenger into the route. This whole procedure is then repeated with the next offer until all offers have been checked and the global best fit has been found.

4.2.2.3 Adding Riders to an Offer

After finding the best fit, we finally have to insert the new passenger into the drivers route and update the remaining detours of every involved passenger and the driver. We also need to make entries into our bucket structure as two new stops will be entered, which should be considered by further requests. For updating the buckets we have to compute four CH search spaces. The algorithm for finding the best fit not only gives us the minimal summed up detour, more precisely it gives us an detour-object containing all information we need to update the offer.

The object contains information about the minimal detour (sum of all detours), the distances between the stops of the new route, the positions where the new stops should be entered into the route, the waiting time for the driver and the waiting time for the rider. The actual driven detour is recalculated later with the given distances.

For the sake of clarity Figure 4.8 shows an examples where we marked the information of the detour object in green and the information we have from the offer object in blue. The distance $\mu(o^r, d^r)$ is still known from the beginning of the request procedure. The passenger order vector $P^d_{order}[\cdot]$ and the step distance vector $D^d_{step}[\cdot]$ of the offer will get updated by inserting the new stops at the right position, which are specified by two entries of the passenger order. An example of the vector updates can also be seen in Figure 4.8. The remaining detours of passengers and driver are then updated by the same procedure that checked the validity of the new route in the section before, the only difference is that we really update the values this time. Finally we insert a new passenger object into the passenger vector of the offer with the next free *subID*.

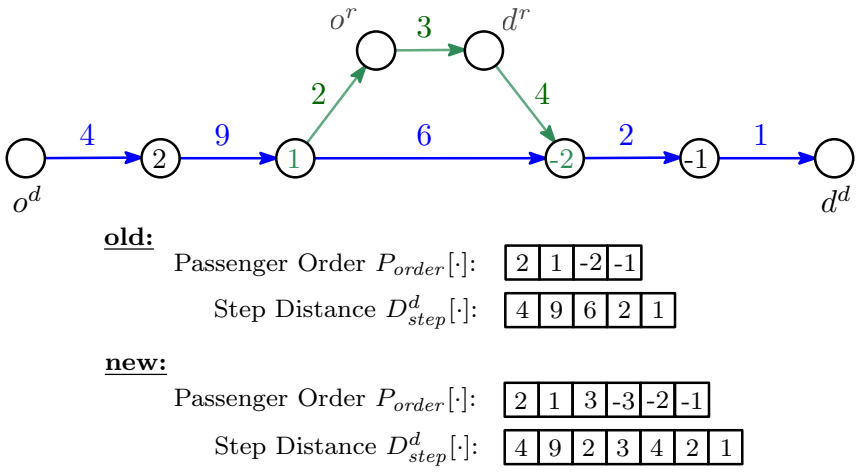


Figure 4.8: Inserting o_r and d_r of a request into the route of the matched offer. Green values are given by the detour-object, blue values are given by the offer-object. Notice that the detour-object also holds two *subID* marking the position for the insertion.

Finally we insert new bucket entries for the passenger. We take the forward and backward CH search spaces outgoing from o^r and d^r, resulting in four search spaces ($S^{\uparrow}(o^r)$, $S^{\downarrow}(o^r)$, $S^{\uparrow}(d^r)$ and $S^{\downarrow}(d^r)$). We then traverse them and insert a new entry into the forward and backward buckets of the vertices inside them. For the backward search of the origin o^r and the forward search of the destination d^r we prune the insertion with the distance $\mu(o^d, d^d) + \Delta^d_{max}$ as entries which have a bigger distance simply can't be reached by the matched offer. For the other two search spaces we can prune the bucket entries with the minimum $min(\mu(o^r, d^r) + \Delta^r_{max}, \mu(o^d, d^d) + \Delta^d_{max})$ as we're looking at distances between the new request. Therefore we have restrictions by the maximum allowed distance for the

request and the maximum allowed distance of the matched offer. Any vertex that is further away would definitely violate the detour constraint of the new passenger or the offer. This is the same pruning tactic as we used on entering offers.

We also want to point out that for every new entry we calculate the time periods that are possible for reaching a specific vertex in time, we only make entries into the affected daytime buckets. After the request is finished, the new rider is matched and with the updated buckets we are ready for the next request. Any request that would occur before the buckets have been updated may result in wrong matchings and inconsistent data. This leads to the next topic.

4.2.3 Multi-threading

During the implementation of our algorithm we saw some major possibilities for multi-threading. As we aim for fast answering times for requests we only considered optimizing the requests towards multiple cores. Therefore we used OpenMP as it provides simple mechanics for dividing the work among cores.

First we split up the one-to-many (many-to-one) queries. As stated before we have four of them and each of them can be calculated separately, therefore each query gets a task filling one out of four tentative distance vectors without critical sections. We then wait for all vectors to be filled and can split up the for-loop which checks the best detour for every offer. This can be divided among an unrestricted amount of threads, as each offer can be checked on its own. Each core gets his own best detour object which it is minimizing. After all threads finished and found their local minimum the global minimum is determined by comparing the lowest local minimums of all threads.

This multi-threading works well as every thread only has to read and combine information and never edits anything but his locally stored minimum detour object. The limiting factor for the parallelization are the four queries at the beginning, if we would allow more multi-threading in each task, critical sections would occur and thus may slow down the calculations.

4.2.4 Complexity

Finally we want to give a brief overview over the complexity of our algorithm. Therefore we consider the sequential version of it but also have a short look on the possible speedup of our multi-threaded approach. Given the algorithm it is clear that entering an offer is the faster part. Thus we will focus on the request.

For giving an upper bound for the calculation we want to assume that n is the maximum amount of steps a route of an offer has and that there are m active offers in the system which have to be checked for a new request. For checking a single offer with n stops we saw in Section 4.2.2.2 that we have $\mathcal{O}(n^2)$ possible ways of entering the new request into the existing route of the offer. Also in the worst case we have to check all m offers leading us towards an upper bound of $\mathcal{O}(m * n^2)$ which seems bad. This means the more matches we get in our system the slower our algorithm runs. But normally the amount of passengers for an offer and thus the amount n of stops for the ride stays rather small as with every match normally a detour is added and the likelihood for further matches decreases. And even if this number gets higher for one offer most of the other offers will still have a rather small amount of stops. Also the amount m of offers which have to be checked can be reduced with the stated improvements.

The four one-to-many (many-to-one) queries at the beginning of a request highly depend on the vertices inside the search spaces and the amount of bucket entries, which again

depend on the amount of vertices inside the search spaces. Therefore we have too many uncertainties and won't provide a statement for this complexity.

Last we want to have a look at the possible speedup with our multi-threading approach. As we split up the four one-to-many (many-to-one) searches into four tasks whereas each can be executed on its own without a critical section, we can achieve a maximum speedup of 4 within this part. Our implementation for checking the possible match between offer and request is modular and thus each offer can be checked on its own without conflicts between them. Thus the possible speedup here depends on the amount of threads and cores we have. For n threads a theoretical speedup of n should be possible for this part. But as the shortest path queries are only split among four threads we won't investigate more than four threads. Thus for improving the multi-threading even more in the future the focus should lie on improving the parallelization of the one-to-many (many-to-one) queries.

5. Taxi Ridesharing

As we found many related work solving the taxi ridesharing problem, which is relative to our ridesharing problem, we also wanted to test our approach within this area. Therefore we took the algorithm of the section before and expanded it. It should be mentioned that the following algorithm is only showing the feasibility. We are aware of that the algorithm is not highly optimized towards the taxi problem and that we may not achieve as good solutions as works with purely taxi ridesharing in mind. We want to show that our approach can be easily adapted and extended to similar problems. The differences between our problem and the taxi ridesharing problem were presented in Section 2.2.2.

Our solution is an adapted version of the ridesharing solution. As we have no more offers we simply deleted the possibility of adding offers. Thus, only requests are allowed. As soon as a request arrives we try to match it with the routes created by requests made before, thus generating routes which involve multiple requests. We try to enter the two stops of the new request into any route in the system, if we find a feasible match of route and request we extend the route. If no such match is found we simply open a new route with the two stops of the request. Once the last request is received we schedule the taxis. The service described is illustrated in Figure 5.1.

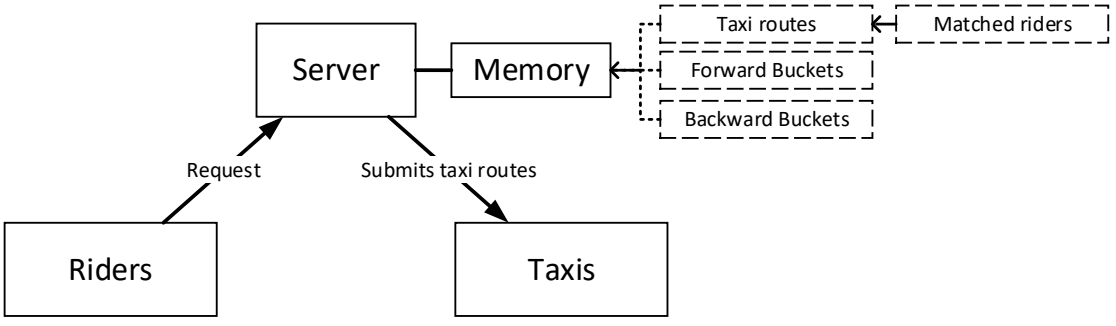


Figure 5.1: The model of our taxi ridesharing service with rider, server and memory. The server now stores taxi routes instead of offers and the only interaction that can be performed by users is making requests.

5.1 Data Structures

The server still holds exactly the same structure for fast one-to-many (many-to-one) queries as in Section 4, as the problem changes nothing on how we achieve the shortest distances. But instead of offers we now store taxi-routes. Each taxi-route stores all information about its route and the passengers which are matched towards it. Like this it is a slimmed version of an offer, which would also store information of the driver. The passengers are stored exactly as before.

Taxi-Routes

Taxi-routes are generated automatically. As soon as a request can't be matched towards existing taxi-routes a new taxi-route is generated, which later on can be extended by new requests. For a generated taxi-route holds that there are no two consecutive stops, with distance greater zero, which do not describe the transportation of at least one passenger. Hence, taxi-routes don't describe routes where the taxi is driving empty. A taxi-route holds the following information:

- $\boldsymbol{ID_t}$**:** Each taxi-route has a unique id.
- **Passengers** $\boldsymbol{P}[\cdot]$**:** All matched passengers get stored as a passenger object in the passenger vector $P[\cdot]$. Each passenger has a unique id $subID_t$ among the passengers of the same taxi-route. The values stored by a passenger object are the same as in Section 4.1.2.
- **Passenger Order** $\boldsymbol{P_{order}}[\cdot]$**:** A vector storing the order of pick-ups and drop-offs of the passengers in a taxi-route. For each stop in the taxi-route exists an entry in this vector.

  Each passenger within a taxi-route has a unique $subID_t$. For each stop we store the $subID_t$ of the passenger of that stop into the corresponding position of $P_{order}[\cdot]$. To distinguish between a pick-up and a drop-off we simply store the positive or the negative value of the $subID_t$, respectively. Positive values representing pick-ups and negative values drop-offs. Therefore we always know, which stop belongs to which passenger, and as the passenger object holds his origin and destination we can also reference towards the exact vertex of each stop.
- **Step Distance** $\boldsymbol{D^t_{step}}[\cdot]$**:** This vector stores the distances between the stops stored inside the passenger order.

5.2 Algorithm

The algorithm consists of two parts, first we handle the requests and generate taxi-routes. Afterwards we schedule the taxis and submit them towards the taxi-routes. As the routes have different start- and end-times a single taxi is capable of fulfilling multiple taxi-routes. A taxi is only allowed to drive without any passengers to reach the start of the next taxi-route. We will present a first fit solution, well knowing that other approaches specially focussed on the taxi problem may provide better results. But our main focus is still the computation time for requests. To achieve even faster requests we consider the same multi-threading approach as in the ridesharing solution.

5.2.1 Making Requests

This part of our solution is close to the solution introduced in Section 4.2.2. Therefore we will skip parts that stay unchanged and focus on the differences caused by adaptations.

In the ridesharing algorithm introduced before the request handling was divided into three phases:

1. Finding minimal distances
2. Finding the best fit
3. Adding the rider towards an offer

The first phase of finding the minimal distances stays untouched as we use the same approach for the one-to-many (many-to-one) queries as before. Also the daytime buckets are still used the same way as before, for speeding up the queries by reducing the amount of vertices considered during the search of shortest distances.

For the second phase we only have to make slight adaptations: A taxi route does not have a fix origin and destination like an offer. With taxi-routes it becomes possible to add new stops before the current first stop or after the current last stop of the route. Therefore we have to consider these new possibilities within the two cases (Section 4.2.2.2) of inserting the stops into the route. For a taxi-route with n stops we then have $n+1$ instances for the first case and $\frac{n*(n+1)}{2}$ instances for the second case.

As a taxi-route is not allowed to contain sub-routes without transporting passengers we can say for the two new instances of the first case, where both stops of the request are either entered at the beginning of the taxi-route or the end, we only allow this matching if the destination of the request is identical to the current start point of the route or the origin of the request is identical to the current end point. For the new instances of the second case of insertion we simply test them the same way as the other insertions.

The third phase stays rather untouched, we only have to allow insertion of stops as new start or endpoint into the taxi-route. Also, if no taxi-route has been found to match with, we simply create a new taxi-route consisting of the two stops of the request.

5.2.2 Submitting Taxis

To get an impression of how many taxis our approach really would need to fulfill all requests we implemented a rather simple solution for scheduling the taxis. We sort the generated taxi-routes according to their start times. Then we submit the first taxi to the earliest taxi-route, determine the destination time of the taxi-route and then submit the same taxi towards the first taxi-route it can reach afterwards. We then repeat this for the taxi until no further taxi-routes can be submitted towards it. Then we start assigning taxi-routes towards the next taxi the same way. We repeat this procedure until all taxi-routes have been submitted towards a taxi. This is in no means a balanced solution but does provide a valid solution within short time.

For every taxi-route we get the start time $t_{start}[ID_t]$ as the start time of the first passenger. We take the sum of all step distances $D^t_{step}[\cdot]$, add it to the start time and get the arrival time $t_{arrival}[ID_t]$.

We create a map of all taxi-routes that have to be assigned to a taxi. This map has one entry for every existing start time as key, therefore it is ordered in ascending order by this time. Every entry points to a vector of taxi-routes. The vector will contain one entry for every taxi-route with this very start time (keep in mind that several taxi-routes may have the same start time).

To add a taxi-route means to search the map for the appropriate start time and if this time already exists, adding an entry to the vector it points to. If the time not yet exists, we insert a new entry with the start time, mapping to a new vector that is filled with the taxi-route to be added.

We then start to submit the taxis. For the first taxi we consider the vector of the first entry of the map. We take the last taxi-route of the vector and delete it from the vector (For our algorithm, it does not matter which entry we take, thus we go for the last one, because deletion is fastest then). The deleted taxi-route is considered as assigned to the taxi.

We continue with identifying further routes for the first taxi. Therefore we have to check the entries in the map where the start time is later or equal to the arrival time for the first route. We check the possible routes in ascending order of their start times. A route can only be assigned to the taxi, if the starting point can be reached in time. Therefore we calculate the time needed for the taxi from the endpoint of the preceding tour to the start point of the considered tour. If this time is short enough to reach the start point in time, the tour is qualified by this criteria and will be assigned as the next one to the taxi. This is repeated with further taxi-routes until the arrival time of the last assigned taxi-route allows no further taxi-routes to be assigned to the taxi.

If we still have unassigned taxi-routes in the map, we start the same procedure with a new taxi. We repeat this until all taxi-routes are assigned to a taxi. Afterwards we can output the number of taxis we would need to fulfill all requests.

We are aware of that there are better ways of assigning routes to taxis, leading to less taxis needed, but as mentioned before we are mainly interested in the request times and only want to show the feasibility of solving the taxi ridesharing problem with our approach.

6. Experiments

In this chapter we want to provide deeper insight into our algorithm and analyse how it behaves while tuning the parameters. We also want to compare our ridesharing solutions against URoad which was presented in Section 3.3.8. Finally we take requests from real world taxi data of New York and test it with the taxi ridesharing variant of our algorithm.

We divide the experiments into various sections. First we want to discuss the influence of several parameters for our ridesharing solution. Therefore we differentiate between technical parameters and scenario parameters. After finding a good set of technical parameters to run our algorithm with, we will vary the scenario parameters one by one, to see if any singularities can be found where the running time or quality of our solution collapses. Afterwards we compare our ridesharing solution against URoad. As our URoad implementation did not achieve the running times stated in their paper we will also vary parameters for URoad and provide an altered set-up which achieves running times close to the ones stated in their work. We then run the same instances on URoad and our solution and compare the computation times. Finally we will test our taxi ridesharing algorithm with real taxi data of New York.

For our experiments we consider the graphs mentioned in Table 6.1. We generated random origin-destination-pairs for the graphs with different mean distances of the trips, whereas the actual distance of the trips is geometrical distributed. For the graph of New York we extracted the original taxi data from 07 May 2013 [oNY] and mapped the coordinates of the origins and destinations by their latitude and longitude informations to the closest points on the graph. The data was chosen from the year 2013 as in later years the informations of latitude and longitude of the trips got removed.

Table 6.1: Graphs used in the experiments

Graph	**Vertices**	**Edges**	**Source**
New York	64'321	154'947	extracted from OSM [Ope19]
Los Angeles	239'411	576'093	extracted from OSM [Ope19]
Europe	18'017'748	42'560'275	DIMACS Challenge [DGJ09]

All calculations were performed on a computer provided by the Institute of Theoretical Informatics (ITI), which had an Intel(R) Xeon(TM) E5-1630v3 CPU with 4 cores @ 3.7 GHz, 128 GB main memory and run openSUSE 15.0. Multiple scripts were written to

execute the various calculations, whereas first each calculation was performed three times to obtain correct values without random effects. As the first runs indicated a stable running of the experiments, where the different runs varied only slightly, we decided to lower the number of executions per setting. The time saved was then used to achieve a larger variety in parametrization of the experiments.

6.1 Parameter Evaluation

In this section we discuss the influences of the many parameters we can adjust. Table 6.2 contains a list of possible parameters. We divided the parameters that we want to test into two categories: Technical parameters (marked bold) which have direct influence on our algorithm and simulation parameters which include the parameters with main influence on the simulation.

Table 6.2: Parameters and their tested values

Parameter	**Possible values**
Threads	1, 2, 3, 4
Daytime Buckets	1, 8, 12, 24, 32, 48, 72, 96
∅ OD-Pair distance [km]	2.5, 5, 10, 20, 50, 100, 500
# Offers	1000, 10000, 50000, 100000, 200000, 400000
# Requests	1000, 10000, 50000, 100000, 200000, 400000
Detour factor δ	0.25, 0.5, 0.75, 1, 1.5, 2
Free seats per offer	1, 2, 3, 4

Obviously, it is impossible to test every of the approximately 200'000 combinations of parameters. Therefore we decided to define a standard parameter set for each graph and to run test only varying one of the parameters while the others stay unchanged with their standard value.

The pre-set standard parameters for Los Angeles and Europe can be found in Table 6.3. Our initial parameter set was chosen to simulate ridesharing during the hours from 07:00am to 10:00am with 50'000 offers and 50'000 request. We chose these numbers as when we expand these values towards a whole day we would have approximately 400'000 offers and requests during 24 hours (ignoring the variations during the day) and this amount of requests is the same as provided by the taxi data from New York for a normal work day. But obviously experiments with 50'000 offers and requests can be executed faster. For the graph of Los Angeles we did not consider Origin-Destination-Pairs (OD-Pairs) with mean distance above 50km as bigger distances in such a city don't seem to make much sense.

Table 6.3: Parameters and their pre-set values

Parameter	**Los Angeles**	**Europe**
∅ OD-Pair distance	**10km**	**50km**
# Offers	50000	
# Requests	50000	
Detour factor δ	0.5	
Free seats per offer	3	
Threads	4	
Daytime Buckets	24	

6.1.1 Technical Parameters

In this section we will test the technical parameters marked bold in Table 6.2 and discuss the improvements we can achieve with them.

6.1.1.1 Threads

First of all we want to discuss the fact that we run most of our experiments with multi-threading. Therefore we compare the timings of the programs with one, two, three and four threads. All other parameters are set towards the pre-set values.

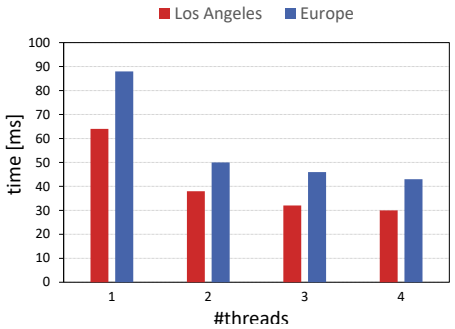


Figure 6.1: Results of multi-threading

Figure 6.1 plots the four different mean running times of a request with varying amount of threads. We see that the running time decreases, but not as expected. The speed-up from one to two threads is about 1.68 on the graph of Los Angeles and 1.76 on Europe, which seems reasonable. But by increasing two to four threads we only achieve a speed up of 1.27 for Los Angeles and 1.16 for Europe, leading to a total speed up from one to four threads of 2.13 and 2.04. This is far less than linear speed up. During the experiments we noticed that the CPUs workload went down with multiple threads running. This indicates waiting times for the CPUs as they can't run on full speed.

During implementation we considered the data sharing between multiple threads and tried to let every thread work on its own data. But as the test reveals there seems to be some kind of bottleneck we could not identify so far.

But after all the multi-threading approach with four threads still brings a speed up for the experiments of factor 2 and for a server running a ridesharing algorithm it is realistic to have at least four cores. Therefore we chose multi-threading with four threads in the pre-set.

6.1.1.2 Daytime Buckets

Next we tested the influence of daytime buckets. The original work for fast detour computation [GLS+10] did not consider the timings for the offers and requests. But with adding these time information we also became able of filtering possible matches based on their desired travel times. The technical approach to do so was the introduction of daytime buckets. As mentioned in the end of Section 4.2.2.1 there may be an alternative way of implementation using interval trees, which at least would avoid multiple entries for time periods spanning over more than one bucket. But in the end we could not test this alternative approach and therefore we restrict ourself to the daytime buckets presented.

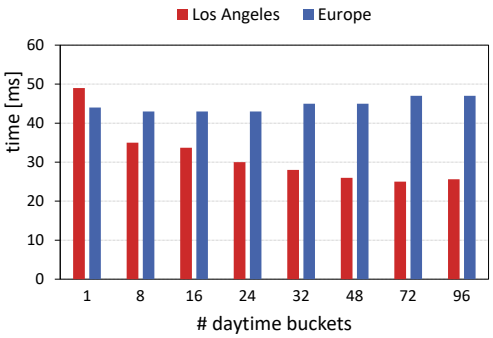


Figure 6.2: Results for varying the amount of daytime buckets.

Figure 6.2 shows the mean response times for a request with different values for the amount of daytime buckets. For Los Angeles we clearly see that the response time decreases as more daytime buckets are used. With a single bucket we have a mean response time of 49ms, this value gets almost halved towards 25ms with the use of 72 daytime buckets. This speed up is solely due to fewer bucket entries which have to be scanned.

We also can see that the improvements brought by daytime buckets is not infinite as the steps 48 to 96 all show response times of around 25ms.

Also for the step from one daytime bucket towards 24 daytime buckets we don't see an improvement near to the factor of 24 mentioned in Section 4.2.2.1. In this case the reason is simple: For the experiment we considered a time period of 3 hours. Every offer has a starting time between 07:00am and 10:00am. The arrival time may be later for some of them. This means that we will have approximately 4 buckets heavily used (covering the time from 07:00am to 11:00am) but all earlier buckets are empty and the following ones will contain only very few entries if any. Thus we may expect in theory a speedup by a factor of 4, ignoring the effects of overlaps, and got a factor of 2 in the measurements, which still is a good result.

For the experiment with Europe we don't really see an improvement by introducing daytime buckets. This can be explained by the fact that on a big graph like Europe with the same amount of offers and requests but much more vertices, the entries for the buckets are distributed across a higher number of vertices, leading to less filled buckets. On the other hand, distances in Europe have been chosen higher so that more trips will touch multiple buckets.

The latter effect seems to become dominant especially for the higher numbers of daytime buckets so that results even become a bit worse. As a first conclusion we can say that daytime buckets are a good solution if the average duration of a trip is short (significantly shorter than the time covered by a bucket) but of no advantage for longer trips. We have to leave the question open, whether interval trees would also have shown an improvement for the Europe graph.

6.1.2 Simulation Parameters

Our initial parameter set simulates ridesharing during the hours from 07:00am to 10:00am with 50'000 offers and 50'000 request. The detour factor of 0.5 for every driver and rider is an often chosen value in the literature and in the related works, therefore we also start with this value. Also, we pre-set the algorithm to run on four threads. On the one hand it seems reasonable that a ridesharing services runs on a multi core server, on the other hand we have already seen, that this saves time that can be used to run more tests in total. Also on the basis of our multi-threading approach we assumed that the speed up for four threads always should be nearly the same and that we can estimate single core timings with the multi-core ones by considering the factor of 2.

For the graph of Los Angeles, we find that a mean distance of OD-Pairs between 10km and 50km seem reasonable. Higher distances will be tested with the graph of Europe as obviously the real occurring distances across Europe are also higher than those in Los Angeles. Thus for Europe we choose a mean distance of 50km for OD-Pairs and also add an evaluation for mean distances up to 500km.

For each test we have three values which indicate the quality and speed of our solution. First of all, as we want to provide an online solution, the value of main importance is the mean response time per Request. For the quality of our Solution we then got the percentage of matched rides and the percentage of saved travel time. Table 6.4 shows the first results. All parameters not mentioned explicitly have been set to the pre-set standard values.

Table 6.4: Results for the pre-set case

Measure	**Los Angeles** (10km)	**Europe** (50km)
∅ **response time**	30 ms	43 ms
# matched	23'137/50'000 (46.3%)	14'563/50'000 (29.1%)
saved driving time	4.4%	3.8%

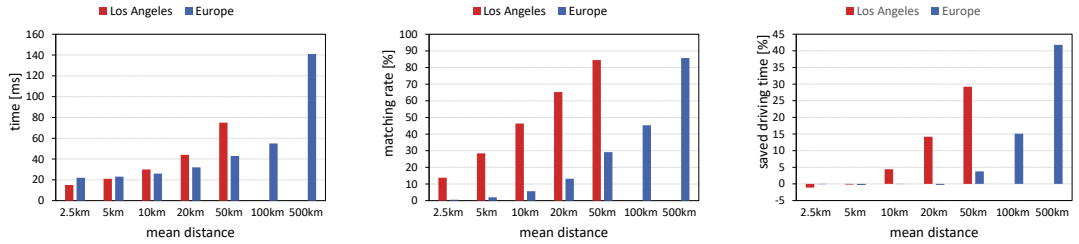
In both cases we see that the percentage of matched requests is a bit low. The main reason might be the relative even distribution of offers and requests across the whole graph without considering hot spots where origins and destinations accumulate. Thus, the probability that two routes are close to each other is reduced.

But in both cases requests are fast and only take 30ms and 43ms on average. And the difference in driven time indicates the purpose of ridesharing. We need 4.4% less driving time to transport every offer and every request in Los Angeles. In this case, this leads to 636 hours less time cars are running on the streets of Los Angeles in a period of three hours, thus saving at least 212 cars.

Despite the somewhat disappointing matching rates this is a good start for the following experiments as we do not have any extreme situations. We now consider each parameter on its own and see if our algorithm remains stable.

6.1.2.1 Origin-Destination-Pairs

The first parameter of the simulations we want test is the distance of the Origin-Destination-Pairs (OD-Pairs) we use for offers and requests. We generated different sets of OD-Pairs with geometrically distributed distances which have a mean length of 2.5km, 5km, 10km, 20km and 50km for Los Angeles and 2.5km, 5km, 10km, 20km, 50km, 100km and 500km for Europe. OD-Pairs with mean distance above 50km within a city do not seem reasonable. Also ridesharing with distances below 10km do not convince us of their need, but we added them as we need the values later on for the comparison with URoad. Figure 6.3 shows the big impact of the varying distances.



(a) Response times (b) Matched requests (c) Saved driving time

Figure 6.3: Results for varying OD-Pairs with mean distance from 2.5km to 50km in the city of **Los Angeles** and from 2.5km to 500km for **Europe**

Response Times

The response times for requests in 6.3a show: the smaller the distances the faster our algorithm. This is due to multiple reasons.

First, if the distances are smaller the number of vertices which are covered by the shortest path of an offer or the shortest path of a request is significantly lower. Hence, each request lies within the reach of fewer offers leading to less offers to be checked. Also the buckets of the vertices hold less entries, as we prune the entering if distances get too big. With 2.5km distances this is a very effective pruning, because only vertices, lying inside a circle with radius 3.75km around the origin or destination, have bucket entries for this origin or destination. Less bucket entries lead to faster computation. For the city of Los Angeles the plot indicates that by quadruplicating the distances we double the response time.

Secondly a request that can be matched needs more time than a request that is not matched. This is caused by the additional time needed to update the trip of the driver. Thus, with an increasing percentage of matches we will also see an increasing average response time.

For Europe the increase in response time is much less for smaller mean distances. The main reason is, that the number of matches does not increase fast for the smaller distances, so the second effect described for Los Angeles doesn't apply here. For longer distances we see a stronger increase for matches and also for running times. Finally, the average response time of 140ms for Europe, with distances around 500km (means quite long paths to be considered) and a matching rate of more than 80% (means quite a couple of trips to be updated) is pretty good.

Matched Requests

With increasing distances between origin and destination Figure 6.3b indicates that the matching rate goes up. This is coming as expected for the same reasons as for the response time. The longer the distances the more offers are within reach of a request. Also bigger detours are allowed and as the allowed times for detours can also be considered as waiting time the rides become far more flexible. This is an indication for why a parameter for waiting time in addition to the detour can be relevant. For smaller distances the allowed detour is also small, leading to inflexible offers.

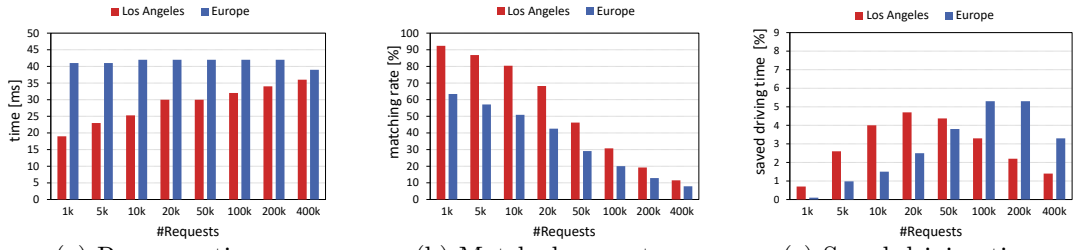
Saved Driving Time

Finally we look up on the saved driving time in Figure 6.3c. This plot indicates something interesting, ridesharing is not always valuable. Mainly for short trips the few matched rides do not decrease the usage of vehicles but increase it slightly instead. For OD-Pairs with mean distance of 2.5km the driving time increased from around 4'735 hours looking at each ride separately towards 4'787 hours driving time with ridesharing, therefore the

percentage of saved driving time is -1.1%. The longer the distances the more valuable becomes ridesharing. This is understandable because there are more matches, the length of the distance travelled together increases and less often the permitted detour has to be utilised up to the limit.

6.1.2.2 Number of Offers and Requests

The next two parameters we want to discuss are the amount of offers and requests. We hold the value of either the number of offers or the number of request at 50'000 and vary the other value. We then combine the results to present the findings.



(a) Response times (b) Matched requests (c) Saved driving time

Figure 6.4: Results for varying the amount of requests.

Figure 6.4 shows the influence of changing the amount of requests. We started with 1'000 requests and went up to 400'000 requests. For Los Angeles the mean response time for a request in Figure 6.4a goes up from initially 19ms towards 36ms. This is a rather small increase of factor 1.89 regarding that we have 400 times more requests. For Europe the results denote almost no increase at all for up to 200'000 requests. For 400'000 requests the response time even decreased slightly.

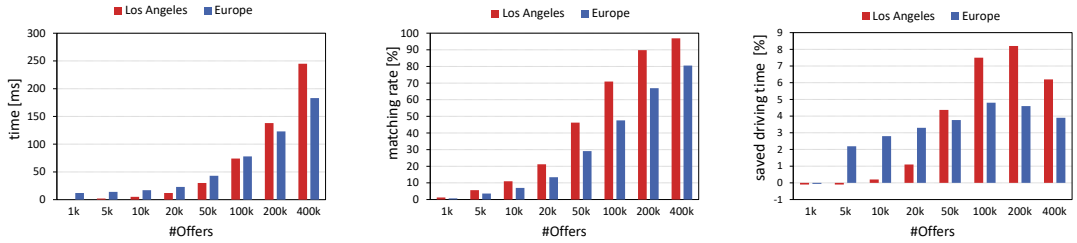
Answers for Los Angeles can be at least partially found within the matching rates shown in Figure 6.4b. As the amount of offers stays the same with 50'000 the rate of requests that can be matched with an offer drop from initially 92.4% towards 11.5%. Thus at the beginning 924 of 1'000 requests get matched and stored in the system and later on only 46'000 of 400'000.

With a very simplified view: Both times we start with 50'000 stored offers in the system giving 100'000 stops in total. In the first case 924 requests get stored additionally, in the second case 46'000 requests. This leads to 101'848 stops for the first case in the end. The second case ends up with 192'000 stops in total. Dividing these numbers we again see a factor of 1.89.

Matches are more likely to be achieved for the first requests that are checked. Requests at the end only have many offers with already no more capacity or a meanwhile reduced possible detour. Looking at the figures, we can see, that from the 46'000 matches some 23'000 happened for the first 50'000 requests, just leaving another 23'000 for the remaining 350'000. Thus the majority of the requests has really to face the higher number of stops to be checked.

For Europe this calculation does not work. We have no real increase in the response time, when the amount of request and also matched requests become higher. In general, we have two opposite effects when the number of assigned requests has increased. For the remaining requests, the distance calculations to the offers become more expensive because they now contain more stops. This proportion of the average response time is therefore increasing. On the other hand, there are fewer matches and thus fewer route updates (an update on Europe takes about 20-30ms for Europe), so this proportion of the average response times decrease.

Finally the percentage of saved driving time in Figure 6.4c tells us that within our scenario a ratio between requests and offers about 1:2 for Los Angeles and 3:1 for Europe is most effective. If we add more requests the growing percentage of non matched requests lowers the percentage of saved driving time and with fewer requests we do not use the offers in an optimal way because they pick up less riders than they have capacity for. The difference of the ratio tells us that it heavily depends on the exact scenario which ratio achieves the most effective results.



(a) Response times (b) Matched requests (c) Saved driving time

Figure 6.5: Results for varying the amount of offers.

Next we hold the amount of request on a fix value of 50'000 and vary the amount of offers in the system. Figure 6.5 plots the results. We started with 1'000 offers and increased the value towards 400'000 offers.

The response times for the requests in Figure 6.5a indicate an almost linear increase (take into account that the division of the x-axis is more of logarithmic nature than linear). For Los Angeles we see 12ms for 20'000 offers and 138ms for 200'000 offers that is a factor of 10 for the offers and a factor of 11.5 for the response times.

Figure 6.5c provides the similar informations as Figure 6.4c and was added for completeness.

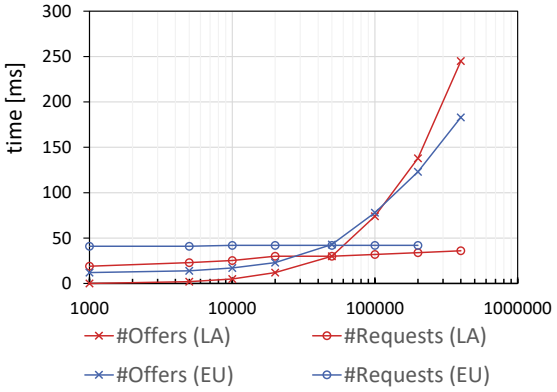
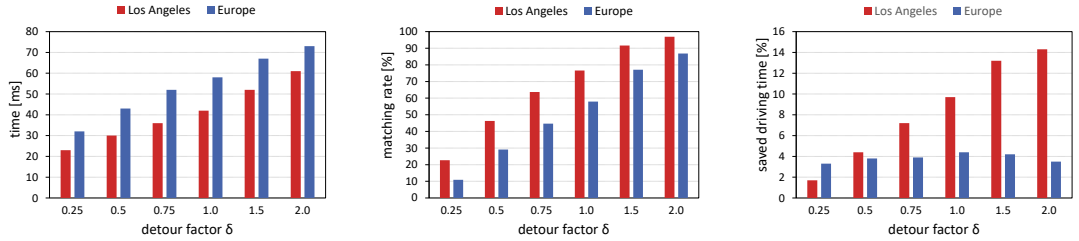


Figure 6.6: Combining the response times of both results.

In Figure 6.6 we put both results for response times of varying offer and request count together. Notice that we have a logarithmic scale with basis 10 on the x-axis. We see that for our experiments the amount of offers have way more influence on the mean response time and the saved driving time than the amount of requests. For both graphs, Los Angeles and Europe, we can say that for time periods of 3 hours (07:00am-10:00am) we can provide an *online* solution for up to 150'000 concurrent offers, with response times less than 100ms.

6.1.2.3 Detour Factor

Experiments with the detour factor δ for offers and requests are rather unspectacular. For simplicity we assume that each offer and request share the same detour factor, even though our implementation can handle different δ-s for each single driver and rider. But if we would not choose some single factor for all, the results of the experiment would be quite elusive.



(a) Response times (b) Matched requests (c) Saved driving time

Figure 6.7: Results for varying the detour factor.

Figure 6.7a presents the mean response time, Figure 6.7b the matching rate and Figure 6.7c the saved driving time for the different detour factors between 0.25 and 2.0. The response times indicate that by quadrupling the detour factor the algorithm needs almost twice as long for a request. This is logical as longer detours increase the amount of offers that can fulfill a request. Thus also the matching rate increases.

But only the percentage of saved driving time for Los Angeles is influenced. The result for Los Angeles indicates that even if, for a single driver, it seems bad to make big detours, as in his perspective this leads to way bigger driving times, larger detours for each person can actually be justified from a global point of view. At least in city areas with rather small distances.

For Europe the situation is different. If we look at the matches for Los Angeles, we see 76.6% for 1.0 and 96.9% for 2.0, means around **20.9%** (20.3 of 96.9) of the matches, need a detour factor bigger than 1.0. The corresponding figure for Europe shows 57.9% and 86.8%, means around **33%** (28.9 of 86.8) of the matches, need a detour factor grater than 1.0. This indicates, that the percentage of longer detours (with respect to the original length of a tour) is higher in Europe and hence the saved driving time does not increase.

6.1.2.4 Seats

We also tested the impact of varying the amount of free seats each offer has. As the results in Table 6.5 indicate, the timings for the request always stay the same with a mean response time of 30ms. Also the impact on the amount of matched requests and the saved driving time were rather small. In table 6.5 we can see that only by switching from one free seat to two free seats a noticeable difference exists of 1.5% and 0.5%.

Table 6.5: Results for varying the amount of free seats

	Los Angeles				**Europe**			
# seats:	1	2	3	4	1	2	3	4
∅ response time [ms]	30	30	30	30	43	43	43	43
matched [%]	44.7	46.2	46.2	46.3	28.6	29.1	29.1	29.1
saved driving time [%]	3.6	4.4	4.4	4.4	3.1	3.6	3.7	3.7

We expected a bigger impact, especially while switching from one free seat to two. We explain the small changes due to the lack of hot-spots in our OD-Pair data. Multiple seats are needed whenever many requests overlap their paths. As our OD-Pairs are generated randomly we lack most of this overlapping and thus don't need many seats. The even smaller impact on Europe can be explained as the same offers and requests are further spread across the bigger graph.

6.2 Comparing with URoad

We analysed the paper URoad [FXH+18] and compared their stated results against ours. We were convinced that their algorithm provides valuable results for comparison with our work. In our experiments we try to reproduce their experiments, while using all available information from their publication, but we can't achieve the mentioned running times. We considered all possible sources to get more information about the setup of the experiments and the data used.

In this section we first want to have a look at the original URoad implementation which uses Bidirectional Dijkstra for all shortest distance queries. We will find that their indicated response times for the algorithms differ in order of magnitudes of what our results show. Afterwards we explain why our results differ and why we believe that URoad as it is, is not capable of solving the ridesharing problem within the stated times. Afterwards we make a simple improvement by using CH-queries for every shortest distance query creating a URoad-CH implementation. We then show that with this algorithm we are capable of finding an experimental set-up in which URoad-CH generates results in order of magnitude of the publication. However, the set-up seems far of from being realistic to us. Finally we compare our ridesharing algorithm against URoad-CH.

For our research we will focus on the response times for the requests and the percentage of pre-filtered offers. Remark: URoad uses the filter and refine framework presented in Section 3.3.1. Thus the speed of the algorithm strongly relies on the amount of pre-filtered offers for a request.

6.2.1 Evaluating the original URoad

For the evaluation of the original URoad we first want to state the experimental set-up described in the publication. All experiments were executed on a road network from the City of Los Angeles (their version of the graph had 193'948 vertices and 530'977 edges). 100'000 offers and 1'000 requests were generated. The time period of the experiment was 07:00am to 09:00am and the travel times of offers and requests were randomly distributed within this period. Offers have a fix departure time and request are scheduled to wait for their ride only a certain amount of time, defined as maximum waiting time (MWT). In the experiments the MWT was set to 5 minutes. They explicitly state that they use Dijkstra's algorithm (we assume they take the faster bidirectional variant).

We tried to remodel this experiment relatively close and thus chose the parameters stated in Table 6.6 for our experiments. As we had nearly no information about the OD-Pairs we run the experiments with mean distances of 2.5km, 5km and 10km. We refrained from using even greater distances as the response times were too bad, we also decided to only make 100 requests per experiment as else they would have run simply too long.

Table 6.7 presents the results for the reconstructed experiment of the original paper. We see extremely bad response times for the request. Even for the OD-Pairs with mean distances of 2.5km, which is unrealistic for ridesharing, we achieve average response times of more than 2 seconds with 20'000 offers in the system. For 100'000 offers the average response times even increase to over 8 seconds per request, which is by any means to much for an online ridesharing service.

Table 6.6: Experimental parameters for URoad

Parameter	**Value**
OD-Pairs	2.5km, 5km, 10km
Offers	20'000, 100'000
Requests	100
Detour factor	0.5
Max waiting time	5 min
Seats	3

The increase from 2 to 8 seconds is somehow realistic, considering that the percentage of offers which are considered for the matching stay around 3%. If we increase the mean distances of the OD-Pairs the times explode. This is due to two reasons. The main reason is that the Bidirectional Dijkstra does not scale well with bigger distances, as the volume of the circular search spaces increases fast for growing distances. The second reason is that for our random generated OD-Pairs the percentage of offers, which have to be considered for the final matching phase, increases when the mean distances are increased. This leads to even more distance queries as more offers have to be considered.

Table 6.7: Results of the reconstructed experiment

# Offers (thousand):	**20k**		**100k**	
mean dist.	∅ resp. time	∅ filter	∅ resp. time	∅ filter
2.5km	2'195ms	3.1%	8'532ms	2.7%
5km	7'295ms	6.3%	30'124ms	5.6%
10km	17'074ms	10.4%	720'071ms	9.3%

For comparison we present the results stated in the publication of URoad in Table 6.8. We don't know the mean distances of the OD-Pairs or the parameters with which they've been generated, but the difference between the response times of their experiment and ours is huge. Even if we consider mean distances of 2.5km for our calculations, we need at least an order of magnitude more time. Differences in the filtering occur only if we consider bigger mean distances.

Table 6.8: Results of the original URoad publication

# Offers (thousand)	**20k**	**40k**	**60k**	**80k**	**100k**
∅ filter [%]	2.0	2.0	2.0	2.0	2.0
∅ total resp. time [ms]	102	112	120	126	135
⌞ *running time filter [ms]*	*5*	*10*	*15*	*20*	*27*
⌞ *running time matching [ms]*	*97*	*102*	*105*	*106*	*108*

Therefore we perform a thought experiment. Taking an example with 100'000 offers and a filter of 2.0% into account. For a request we first filter the offers and have 2% left, which would be 2'000. For each of these 2'000 offers we have to execute at least two distance queries for the distances $\mu(o^d, o^r)$ and $\mu(d^d, d^r)$ and a single query for the shortest distance of the request itself, giving a minimum of 4'001 queries in total. Accordingly to URoad we agree that this is the bottleneck of the algorithm as can be seen with the running times for the matching, as shown in Table 6.8. The running time for the matching includes this distance determinations.

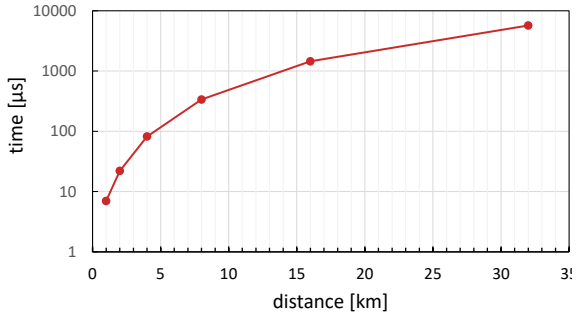


Figure 6.8: Running times for Bidirectional Dijkstra on Los Angeles with varying distances on a logarithmic scale for better legibility.

Figure 6.8 shows the mean running times for the Bidirectional Dijkstra with varying distances on the graph of Los Angeles. The running times increase as expected and we can use this information to continue our thought experiment.

To achieve a response time of 135ms, as stated in the original work, with 4'001 distance queries each query is allowed to take about $34\mu s$. According to Figure 6.8 this would correspond to a mean distance of 3km, which is rather small. Our solution for 2.5km in the other hand needs 8'532ms for some 5'341 queries leading to about $1'597\mu s$ per query. Looking into Figure 6.8, $1'597\mu s$ correspond to a distance of about 16km, which is rather big.

The different average distances to be checked for the offer that survived filtering has main impact on response times. This can at least partly explain, why our response time for URoad with an average of 8'532ms is much worse than the response times of 135ms stated in the original paper. The filter as we have implemented leaves over 33% more offers (2.67% versus 2.00%) and the distance calculations on average face differences 5 times as long as in the original paper (16km vs 3km).

This estimation shows, that we most likely have differences in the filtering of the offers, although our implementation is as close as possible to the descriptions made in the work of URoad.

There is another detail in the original work that seems to be odd: The filter in the original work always leaves about 2.0% of the offers for the matching routine. This means if we quintuple the amount of offers in the system, from 20'000 to 100'000, we also quintuple the amount of offers, which have to be considered in the matching routine. But the calculation time for the matching only increases from 97ms to 108ms by a factor of 1.1, a surprisingly low value. It is hard to imagine any algorithm that could handle 5 times as many calculations in 1.1 times of the time.

In conclusion, our implementation seems to lack some improvements, which exist in the original work but are not stated explicitly. As we couldn't get response from the authors, these improvements were inaccessible to us. They may have used techniques similar to the skyline pruning of SHAREK in Section 3.3.8 or simply a faster and more consistent distance query like CH-queries, or maybe the generated OD-Pairs for their simulations simply supported their experiments. The work definitely lacked informations for us to reconstruct the stated results.

6.2.2 Introducing URoad-CH

As the results of the previous experiments were not as expected, we considered a second implementation of URoad. Now we implemented a version which uses CH-queries instead of Bidirectional Dijkstra. We reference to this version as URoad-CH. We repeated the experiment with the same parameters as before (Table 6.6), only with the differing query algorithm.

Table 6.9: Results of the reconstructed experiment with CH

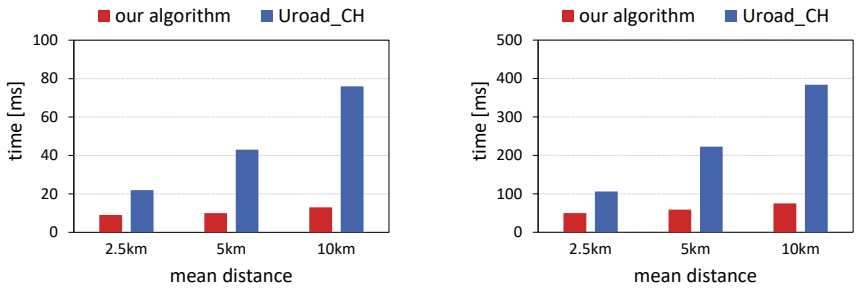
# Offers (thousand):	20k		100k	
mean dist.	∅ resp. time	∅ filter	∅ resp. time	∅ filter
2.5km	22ms	3.11%	106ms	2.67%
5km	43ms	6.29%	223ms	5.61%
10km	76ms	10.36%	384ms	9.31%

The results in Table 6.9 directly show what we already expected. The response times dropped significantly as the impact of the distance calculation was reduced. The results of URoad-CH seem quite realistic: By quintupling the amount of offers the response time almost quintuples as well, presumed the filter percentage stays about the same. If the mean distances of offers and requests gets doubled the percentage of remaining offers for the final matching phase increases as well, and with the then higher amount of possible matches the response time rises. If the percentage of the filter is doubled the response times almost double as well.

For the case that the original URoad work really did use Bidirectional Dijkstra as shortest distance query algorithm, their algorithm definitely could be improved by switching to Contraction Hierarchies, making their algorithm more robust regarding longer distances, and thus potentially better scalable for bigger graphs.

6.2.3 Comparing with URoad-CH

Now that we have an improved version of URoad, with bearable response times, we want to compare our algorithm with it.

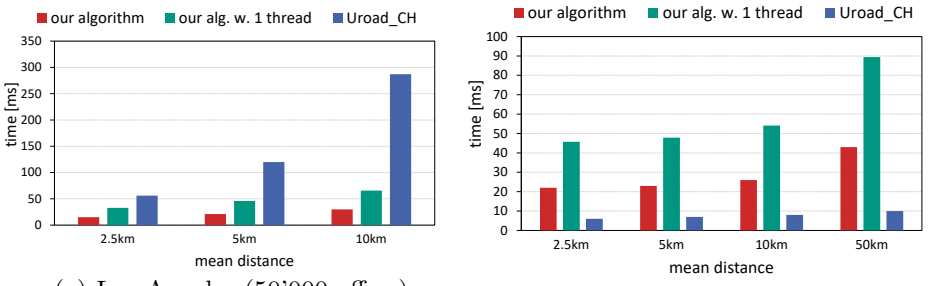


(a) Response times for 20'000 offers (b) Response times for 100'000 offers

Figure 6.9: Comparison of the average response times for a request of our algorithm and URoad-CH on Los Angeles.

Figure 6.9 shows the results of our algorithm in comparison with the results of URoad-CH (Table 6.9). Be aware that the time scales are different (100ms versus 500ms). We executed our algorithm with parameters which are as close to the ones used for URoad-CH as possible. For this reason we also added the possibility to state an explicit minimum waiting time for our algorithm if the allowed detour for an offer or request would be less.

We clearly see that the response times of our algorithms only increase slightly while increasing the mean distances of offers and requests. This is due to the fact that we have no substantial filtering and within each execution about the same amount of one-to-many and many-to-one queries have to be executed. URoad-CH on the other hand relies heavily on its filtering, which gets worse when the distances grow.



(a) Los Angeles (50'000 offers)

(b) Europe (50'000 offers)

Figure 6.10: Comparison of our algorithm with URoad-CH on the standard parameters chosen in Section 6.1. Be aware that the time scales are different.

After running the URoad experiments with our algorithm we run the experiments with the pre-set parameters of the previous section on URoad-CH, only varying the mean distances of the OD-Pairs.

The results in Figure 6.10 show that for Los Angeles (a) our approach is clearly faster than URoad, even if we run our algorithm only with one thread. We therefore decided not to increase the mean distances further. For Europe on the other hand the timings for URoad-CH are fast. For mean distances between 2.5km and 50km URoad-CH only needs about 6ms to 10ms. The answer for these fast solutions are the filters. As Europe is large, the mean distances as well as the number of offers are, related to the size of Europe, quite small. In addition the offers are distributed across Europe. Therefore the filters are very effective. For the case of mean distances of 50km the average filter resulted in only 88 out of 50'000 offers which then had to be considered for the matching.

Something else we noticed while running the experiments was that it is easier to predict the response time for our solution than for URoad or URoad-CH. As URoad relies heavily on its filters, it is possible for two different request that after filtering in one case there are about 100 possible offers left, which have to be checked for a matching, and in the other case there are 1'000 offers or more left. This may become a problem if we really consider some hotspots where offers and requests accumulate and filtering thus is less effective. Therefore even if the average response time for URoad is good, there can always be cases that take up a multiple of the average response time. There are no such big fluctuations for our algorithm.

One last thing we learn from the experiments for URoad is, that if an algorithm for ridesharing uses the filter and refine framework it should definitely not use Bidirectional Dijkstra for its distance queries as there are solutions which scale far better, for example the used Contraction Hierarchies.

6.3 Taxi Ridesharing

The final experiment only serves to show how the taxi ridesharing version of our approach deals with real taxi data. For this experiment we were able to run tests with real taxi data of New York. We are looking at all the taxi rides that took place on one day and treat each taxi ride as a request. The start time of the request is just the time the original passenger started his taxi ride.

As we focus on the calculations for the matching, and wanted to see what happens if a big amount of data has to be handled, we set the current time to a fixed value of 00:00am. That means, we assume that all requests are known already at the bigging of the day. This in fact will increase computation time, taking into account that a current time hh:mm greater than 00:00 would allow our algorithm to exclude those taxi-routes beforehand which have an end time earlier than hh:mm. In total we consider 400'000 taxi rides.

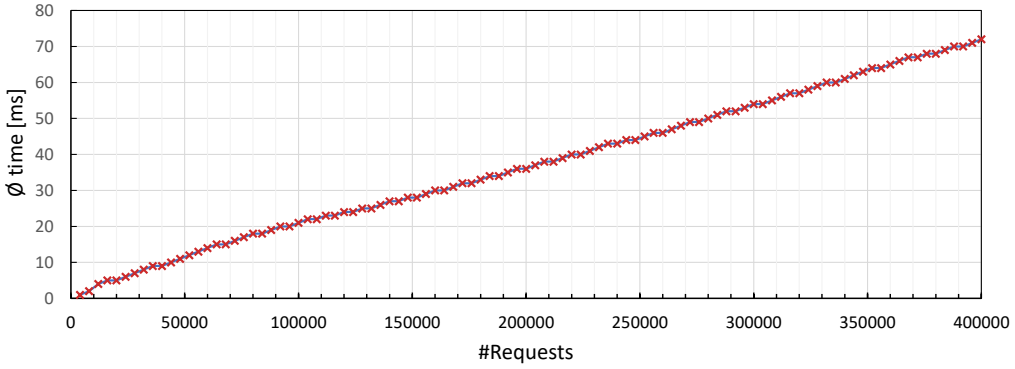


Figure 6.11: ∅ running time for each request in course of the 400'000 requests.

Figure 6.11 presents the average time a request needs to be entered into a taxi-route. At the beginning this process is very fast, as there are only few taxi-routes to be checked (the first taxi-route is generated from the first request). Later on, for each request more and more existing taxi-routes have to be considered and get tested if they can fit another passenger. The average time seems to increase rather linearly even when many taxi-routes exist in the system.

This means that the calculations for finding the best fit for a requester inside a taxi-route is fast, even though it needs $\mathcal{O}(n^2)$ operations for a route with n stops. Even for more flexible taxi-routes and 400'000 requests the amount of stops per route must be rather small, else a near linear running time for up to 400'000 request can not be explained.

This is consistent with the values for generated taxi-routes. Our algorithm generated 94'338 taxi-routes. This means 23.6% of the requests opened a new taxi-route. Hence, the remaining 76.4% (305'662 requests) could be matched with an existing taxi-route. A taxi-route therefore covers in the average a bit more than 4 requests or 8 stops. After all taxi-routes were generated our first-fit algorithm for submitting the taxi-routes to taxis resulted in 2'101 taxis that would be needed to serve all requests of one day.

7. Conclusion

In this work we dealt with ridesharing, more exactly the problem to match a requested ride with existing offers in a ridesharing system. We gave a definition of our ridesharing problem and the taxi variant of it.

In order to gain a broader overview of ridesharing and related problems, numerous publications were analysed, on the basis of which classifications of the problem, possible targets for optimization and various solution strategies were presented.

We also determined [FXH+18] as one of these publications with which we later compared our solutions with.

Afterwards we introduced our solution for the ridesharing problem: An algorithm modelling a service consisting of two functions, adding offers and making requests. The idea of our solution is based on the bucket approach of [KSS+07] for fast computations of one-to-many or many-to-one queries. For this purpose, buckets are assigned to the vertices that contain distance data between this vertex and the stop points of the drivers. As a request arrives we use these bucket entries to calculate the minimal distances between origin and destination of the request and the stops of all offers. With the help of these distances we can then determine possible matches between drivers and the requester. This is done by trying to insert the request in an optimal way into each offered tour and calculating the detours that would result for the driver and passenger(s). Finally we match the request with the best solution, means the offer with the smallest generated detours, and insert the two stops of the request into the route of the matched offer.

We presented multiple improvements for this approach: These cover optimizations within the model like daytime buckets and the pruning of bucket entries for minimizing the amount of entries within a bucket, as well as technical improvements like multi-threading for a parallel solution with faster response times.

To show that our approach is not only valid for one specific problem, we have presented how it can be adapted to the related Taxi Ridesharing problem.

Finally we run several experiments and evaluated our algorithms. We saw that our algorithm can handle up to 150'000 offers within a time period of three hours with response times under 100ms, which is a big number of offers for a city-sized graph. But we also saw that the underlying simulation for the experiments have impact on the outcome of the algorithm, and that it is rather hard to find a good setup, that simulates the real world.

The comparison with the work of [FXH+18] came with multiple problems as we could not achieve their stated results with our implementation of their solution. We had to introduce a slightly different variant of their algorithm and showed that our algorithm could answer requests faster for a city-sized problem. But we also saw that the filter and refine method achieved faster solutions when the graphs became larger, but the number of offers remained the same, resulting in widely scattered offers.

With our last experiment we showed that our solution for the taxi ridesharing problem could handle real life taxi data of New York with good results.

We believe that our solution is good for a small to medium amount of offers (up to 150'000), keeping in mind that we always have to calculate all the minimum distances between a request and the offers. If the amount of offers in the system becomes too high, it seems, that filter and refine framework with a good filter can skip enough offers in advance to compensate for the slower shortest distance calculations.

In the end we believe that our solution could be a good choice for a real world ridesharing service as the response times for a request are more predictable and more than 150'000 offers within a time period of three hours will not be seen so often.

As to a technical aspect we believe that multi-threading is the way to go. There already exist solutions which heavily rely on parallelism as presented in the related work, although some with security concerns. We already considered multi-threading, but in the future it would be interesting to explore this option further.

Also, we would suggest that, in the future, more attention should be paid to the generation of lifelike test data when no real data are available. This would include, for example, that in nearly every real network there are hotspots where origins and destinations accumulate, but also many vertices that rarely appear as the start or end point of a tour. Local accumulations will result in more matches for the same number of offers and requests than for an equal spreading. Since this in turn can influence the response times, test results could be expected to be closer to real life.

Outlook

There is still a long way to go from our solution to a ready to use application. Let us look at parts of the functionality that should be offered by such kind of application and what it could mean for our model:

- It is unlikely that every requester will accept a "blind date" means an automated match without the chance to agree or to reject. In so far, the match should be treated as a proposal and only if the requester confirms, data will be updated. Here we need to check, that the trip did not change in the meantime, we thus need some kind of versionising for trips.
- As trips are planned for the future it may happen, that the driver has to withdraw his offer or the rider his request. Which updates are required in the system and how do we treat the other persons involved. Do we just inform them or do we also present alternatives immediately (especially if the driver withdraws)?
- We assumed in our model, that the driver only specifies origin and destination and automatically chooses the optimal route calculated by the system. But the driver may want to specify his own route or at least some intermediate stops he needs, e.g. to drop off some goods. How do we need to enhance the model in order to be able to store and consider these additional data?

Any of these questions may lead to further research with the goal to extend the model we have presented, thus coming closer and closer to a real life solution.

Bibliography

[AB14] Vincent Armant and Kenneth N Brown. Minimizing the driving distance in ride sharing systems. In *Tools with Artificial Intelligence (ICTAI), 2014 IEEE 26th International Conference on*, pages 568–575. IEEE, 2014.

[ADGW12] Ittai Abraham, Daniel Delling, Andrew V. Goldberg, and Renato F. Werneck. Hierarchical Hub Labelings for Shortest Paths. Technical Report MSR-TR-2012-46, Microsoft Research, 2012.

[ADS+16] Mohammad Asghari, Dingxiong Deng, Cyrus Shahabi, Ugur Demiryurek, and Yaguang Li. Price-aware real-time ride-sharing at scale: an auction-based approach. In *Proceedings of the 24th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, page 3. ACM, 2016.

[AESW11] Niels Agatz, Alan L Erera, Martin WP Savelsbergh, and Xing Wang. Dynamic ride-sharing: A simulation study in metro atlanta. *Procedia-Social and Behavioral Sciences*, 17:532–550, 2011.

[AMSW+17] Javier Alonso-Mora, Samitha Samaranayake, Alex Wallar, Emilio Frazzoli, and Daniela Rus. On-demand high-capacity ride-sharing via dynamic trip-vehicle assignment. *Proceedings of the National Academy of Sciences*, 114(3):462–467, 2017.

[CAMB15] Bin Cao, Louai Alarabi, Mohamed F Mokbel, and Anas Basalamah. Sharek: A scalable dynamic ride sharing system. In *Mobile Data Management (MDM), 2015 16th IEEE International Conference on*, volume 1, pages 4–13. IEEE, 2015.

[CLRS09] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms*. MIT press, 2009.

[CML15] Blerim Cici, Athina Markopoulou, and Nikolaos Laoutaris. Designing an on-line ride-sharing system. In *Proceedings of the 23rd SIGSPATIAL International Conference on Advances in Geographic Information Systems*, page 60. ACM, 2015.

[CML16] Blerim Cici, Athina Markopoulou, and Nikolaos Laoutaris. Sors: a scalable online ridesharing system. In *Proceedings of the 9th ACM SIGSPATIAL International Workshop on Computational Transportation Science*, pages 13–18. ACM, 2016.

[Cor06] Jean-François Cordeau. A branch-and-cut algorithm for the dial-a-ride problem. *Operations Research*, 54(3):573–586, 2006.

[DGJ09] Camil Demetrescu, Andrew V Goldberg, and David S Johnson. *The Shortest Path Problem: Ninth DIMACS Implementation Challenge*, volume 74. American Mathematical Soc., 2009.

[DGPW14] Daniel Delling, Andrew V Goldberg, Thomas Pajor, and Renato F Werneck. Robust distance queries on massive networks. In *European Symposium on Algorithms*, pages 321–333. Springer, 2014.

[DR59] George B Dantzig and John H Ramser. The truck dispatching problem. *Management science*, 6(1):80–91, 1959.

[Dre12] Florian Drews. Multi-Hop Ride Sharing. bachelor thesis, May 2012.

[FDO+13] Masabumi Furuhata, Maged Dessouky, Fernando Ordóñez, Marc-Etienne Brunet, Xiaoqing Wang, and Sven Koenig. Ridesharing: The state-of-the-art and future directions. *Transportation Research Part B: Methodological*, 57:28–46, 2013.

[FXH+18] Jing Fan, Jinting Xu, Chenyu Hou, Bin Cao, Tianyang Dong, and Shiwei Cheng. Uroad: An efficient algorithm for large-scale dynamic ridesharing service. In *2018 IEEE International Conference on Web Services (ICWS)*, pages 9–16. IEEE, 2018.

[Gei08] Robert Geisberger. Contraction Hierarchies. Master's thesis, Universität Karlsruhe (TH), Fakultät für Informatik, 2008. `http://algo2.iti.uni-karlsruhe.de/documents/routeplanning/geisberger_dipl.pdf`.

[GHHC11] Keivan Ghoseiri, Ali Ebadollahzadeh Haghani, Masoud Hamedi, and MAUT Center. *Real-time rideshare matching problem.* Mid-Atlantic Universities Transportation Center Berkeley, 2011.

[GLS+10] Robert Geisberger, Dennis Luxen, Peter Sanders, Sabine Neubauer, and Lars Volker. Fast Detour Computation for Ride Sharing. In *Proceedings of the 10th Workshop on Algorithmic Approaches for Transportation Modeling, Optimization, and Systems (ATMOS'10)*, volume 14 of *OpenAccess Series in Informatics (OASIcs)*, pages 88–99, 2010.

[GSSV12] Robert Geisberger, Peter Sanders, Dominik Schultes, and Christian Vetter. Exact Routing in Large Road Networks Using Contraction Hierarchies. *Transportation Science*, 46(3):388–404, August 2012.

[Har82] Juris Hartmanis. Computers and intractability: a guide to the theory of np-completeness (michael r. garey and david s. johnson). *Siam Review*, 24(1):90, 1982.

[HBJW14] Yan Huang, Favyen Bastani, Ruoming Jin, and Xiaoyang Sean Wang. Large scale real-time ridesharing with service guarantee on road networks. *Proceedings of the VLDB Endowment*, 7(14):2017–2028, 2014.

[Her13] Wesam Herbawi. *Solving the ridematching problem in dynamic ridesharing.* PhD thesis, Universität Ulm, 2013.

[HW12] Wesam Mohamed Herbawi and Michael Weber. A genetic and insertion heuristic algorithm for solving the dynamic ridematching problem with time windows. In *Proceedings of the 14th annual conference on Genetic and evolutionary computation*, pages 385–392. ACM, 2012.

[JJP16] Jaeyoung Jung, R Jayakrishnan, and Ji Young Park. Dynamic shared-taxi dispatch algorithm with hybrid-simulated annealing. *Computer-Aided Civil and Infrastructure Engineering*, 31(4):275–291, 2016.

[KSS+07] Sebastian Knopp, Peter Sanders, Dominik Schultes, Frank Schulz, and Dorothea Wagner. Computing Many-to-Many Shortest Paths Using Highway Hierarchies. In *Proceedings of the 9th Workshop on Algorithm Engineering and Experiments (ALENEX'07)*, pages 36–45. SIAM, 2007.

[LLQX12] Yeqian Lin, Wenquan Li, Feng Qiu, and He Xu. Research on optimization of vehicle routing problem for ride-sharing taxi. *Procedia-Social and Behavioral Sciences*, 43:494–502, 2012.

[MSS+06] Rolf H. Möhring, Heiko Schilling, Birk Schütz, Dorothea Wagner, and Thomas Willhalm. Partitioning Graphs to Speedup Dijkstra's Algorithm. *ACM Journal of Experimental Algorithmics*, 11(2.8):1–29, 2006.

[MZW13] Shuo Ma, Yu Zheng, and Ouri Wolfson. T-share: A large-scale dynamic taxi ridesharing service. In *Data Engineering (ICDE), 2013 IEEE 29th International Conference on*, pages 410–421. IEEE, 2013.

[oNY] The City of New York. NYC Taxi & Limousine Commission - Trip Record Data. `http://www.nyc.gov/html/tlc/html/about/trip_record_data.shtml`. [Online; accessed 02-January-2019].

[Ope19] OpenStreetMap contributors. Planet dump retrieved from https://planet.osm.org . `https://www.openstreetmap.org`, 2019. [Online; accessed 02-January-2019].

[PXZ+15] Dominik Pelzer, Jiajian Xiao, Daniel Zehe, Michael H Lees, Alois C Knoll, and Heiko Aydt. A partition-based match making algorithm for dynamic ridesharing. *IEEE Transactions on Intelligent Transportation Systems*, 16(5):2587–2598, 2015.

[PZMT03] Dimitris Papadias, Jun Zhang, Nikos Mamoulis, and Yufei Tao. Query processing in spatial network databases. In *Proceedings of the 29th international conference on Very large data bases-Volume 29*, pages 802–813. VLDB Endowment, 2003.

[Sav85] Martin WP Savelsbergh. Local search in routing problems with time windows. *Annals of Operations research*, 4(1):285–305, 1985.

[SHZ16] Bilong Shen, Yan Huang, and Ying Zhao. Dynamic ridesharing. *Sigspatial Special*, 7(3):3–10, 2016.

[SSS+16] Maximilian Schreieck, Hazem Safetli, Sajjad Ali Siddiqui, Christoph Pflügler, Manuel Wiesche, and Helmut Krcmar. A matching algorithm for dynamic ridesharing. *Transportation Research Procedia*, 19:272–285, 2016.

[WKW16] Yaoli Wang, Ronny Kutadinata, and Stephan Winter. Activity-based ridesharing: increasing flexibility by time geography. In *Proceedings of the 24th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, page 1. ACM, 2016.