Available online at www.sciencedirect.com

ScienceDirect

Transportation Research Procedia 19 (2016) 272 – 285

Transportation Research Procedia

www.elsevier.com/locate/procedia

International Scientific Conference on Mobility and Transport
Transforming Urban Mobility, mobil.TUM 2016, 6-7 June 2016, Munich, Germany

A Matching Algorithm for Dynamic Ridesharing

Maximilian Schreieck*, Hazem Safetli, Sajjad Ali Siddiqui, Christoph Pflügler, Manuel Wiesche, Helmut Krcmar

Chair for Information Systems, Technical University of Munich, Boltzmannstraße 3, 85748 Garching, Germany

Abstract

Ridesharing is an important component of sustainable urban transportation as it increases vehicle utilization while reducing road utilization. By sharing rides, drivers offer free seats in their vehicles to passengers who want to travel in similar directions. Traditional ridesharing approaches are suitable for long-distance travel, especially inter-city travel, yet they are not flexible enough for short routes within cities. The aim of our research is to develop a service that enables dynamic ridesharing as part of sustainable urban mobility. Dynamic ridesharing refers to a service that automatically matches ride requests and ride offers on short notice without prior agreement between driver and passenger. We present the implementation and evaluation of a dynamic ridesharing service. The implementation part requires an automated matching algorithm that checks whether a driver can take a passenger with him without violating the maximum detour constraint he has set. As this matching algorithm needs to automatically match a relatively large number of ride offers and ride requests in real-time, we focused on building a high-performance algorithm. After implementing the algorithm, we evaluated its performance on a data set with random ride offers in and around the Munich city that is matched with different ride requests. For 10,000 rides in the system, it took less than 0.4 seconds on average to identify the best match.

Peer-review under responsibility of the organizing committee of mobil.TUM 2016.

Keywords: Dynamic ridesharing; real-time ridesharing; smart mobility service; matching algorithm

* *Corresponding author. Tel.: +49-89-289-19510; fax: +49-89-289-19533.*
E-mail address: maximilian.schreieck@in.tum.de

Peer-review under responsibility of the organizing committee of mobil.TUM 2016.
doi:10.1016/j.trpro.2016.12.087

1. Introduction

Urban road networks in many countries are severely facing traffic congestion problems. Consequently, energy consumption increases and air pollution aggravates causing major problems for society, economy and environment (May, 2013). One of the major factors is the growth in the amount of vehicles being greater than the increase in the amount of pavement to accommodate this travel. But it is made worse by the continuing growth of private automobiles. This situation presents challenges that need to be addressed in the interest of long-term urban sustainability and public concern. Ridesharing is one of the emerging solutions to increase the efficiency of the transportation networks by reducing the empty seats traveling in private vehicles.

Traditional ridesharing approaches are suitable for long-distance travel, especially inter-city travel, yet they are not flexible enough for short routes within cities. Contrarily, dynamic ridesharing provides a means by which two or more travelers can be joined in real time with taxi like responsiveness (Arena et al., 2013; Ying, Yu, Changjun, & Jiujun, 2008). However, the way that current ridesharing systems match drivers to requesting riders suffer from matching model because they are quite simple and very limited (Bin, Alarabi, Mokbel, & Basalamah, 2015).

The goal of this paper is to provide a ride matching algorithm which is more efficient and provides the most dynamic way to match the ride requests and ride offers in real time without pre-defining a meeting point in addition to their departure and destination locations. The proposed solution can act as a dynamic ride matching component that can be used as an add-on to any ridesharing system to make it more efficient and dynamic. For example, the solution could enhance the ridesharing application TUMitfahrer, an application that has been developed by a group of students from Technical University of Munich (TUM) for university students to share rides to and between different campuses of the university (Krcmar, 2016).

2. Theoretical background

In the seventies of the last century, fuel crises in the USA caused by the high prices of the fuel all over the world triggered to think about a new way to save money by sharing rides with colleagues and neighbors. Economic analysts continued to forecast that oil prices will be higher in the near future and the budget of consumers will be in a critical situation. In addition, traffic congestion and environmental pollution were a big problem in big cities and have been continuously increasing (Baláž & Londarev, 2006; Yunfei, Xu, & Chunming, 2012).

Since that time, multiple approaches of carpooling have been introduced, formal and informal approaches got together in a way to let ridesharing work. On the one hand, there is ridesharing organized by service providers who conduct the rides themselves and pick up one or several passengers. Some service provider state either a fixed pick-up or drop-off location, such as a train station, while other providers allow their users to choose both locations. Generally, the routing problem for service providers is to build a ridesharing route for a group of vehicles that minimizes the cost of servicing all the passengers.

On the other hand, there is ridesharing which is conducted by individual drivers who take passengers with them to split the fuel bill or benefit from carpool lanes. Consequently, all of the passengers must approve the costs and schedules provided by the driver, including the pick-up and drop-off locations of travelers. Therefore, prior arrangements are necessary to make a rideshare work.

A more flexible enhancement of ridesharing is dynamic ridesharing. Dynamic ridesharing is considered as ad-hoc ridesharing, real-time ridesharing, dynamic carpooling, and instant ridesharing. Comparing the traditional ride-matching process to the dynamic ridesharing ride-matching process, it is clear that the traditional process focuses on passengers traveling from fixed departure and destination locations on fixed schedules. The dynamic ridesharing process is more flexible hence the system is capable of matching random rides at any time and responding quickly to any request even if the request is created on the same day of the ride offer (Dailey, Loseff, & Meyers, 1999).

In recent days most of the ridesharing services either classic or dynamic are working on smartphones with integrated GPS which makes it possible to assume that the users are always trackable and have permanent connectivity and reachability (Furuhata et al., 2013; Stach, 2011). This increases the capability to generate much more valuable data than simple rideshare trip validation. If data were to be collected during the day, detailed travel patterns including the prevalence of trip chaining could be determined. From an urban planning and transport-modeling viewpoint, this information could be used to supplement periodic travel diaries and increase the input data used in urban modeling

activities. Traffic patterns and congestion information can be generated with the help of the very large number of these devices that collect data. Recent startups offer dynamic ridesharing services that allow drivers to offer their car seats to connect to passengers who search for a ride such as Carticipate, EnergeticX, Avego, and Flinc. They provide applications that run on mobile phones that necessarily need internet access. In order to ease the fear when sharing a ride with strangers, users can be linked with social networks like Facebook (e.g., GoLoco and Zimride) or check the driver's reputation provided by reputation systems (e.g., PickupPal).

In the last decade, most studies focused on basic route planning in road networks, developing a large amount of faster speed-up techniques. Earlier, only some classical algorithms existed which were not effective on large graphs. Usually, the new faster algorithms perform a pre-processing step for a graph before discussing the source and destination nodes. This step helps to speed-up the shortest path queries. In the following, we will explain the most important shortest-path algorithms shortly (Schultes, 2008) followed by a brief summary of matching approaches for dynamic ridesharing.

2.1. Shortest path algorithms

Dijkstra's Algorithm is the classical algorithm to find the shortest-path. It calculates the shortest paths from a single source node to all other reachable nodes in the tree (?). The algorithm visits the nodes in order of their shortest-path distances. Dijkstra's algorithm mainly solves the single-source shortest-path problem and when the number of the nodes is very large, it occupies a lot of CPU memory (Zhang & Liu, 2009). An improvement of Dijkstra's algorithm is bidirectional search. It runs two simultaneous searches: one forward from the initial node and one backward from the target node, stopping when the two meet in the middle. However, it can only be applied if the target node is known. Practically, bidirectional search nearly halves the number of settled nodes.

A* Search is a technique greatly used in artificial intelligence. It guides the search of Dijkstra's algorithm towards the target node by using lower bounds on the distance to the target. The efficiency of this approach highly depends on the lower values. A∗ search uses both path costs and heuristic values. The simplest lower value is based on the geographic coordinates of the nodes, but this results in poor performance on road networks. In case of travel time edge weights, even a slow-down of the query is possible (A. V. Goldberg & Harrelson, 2005).

In **reach-based routing** the reach of a vertex encodes the lengths of the shortest paths on which it lies. In order to have a high value of reach, a vertex must lie on a shortest path which extends a long distance in both directions from the vertex. Highway hierarchies generate a hierarchy of levels by switching between node and edge contraction. Node contraction eliminates low-degree nodes and introduces shortcut edges to reserve shortest-path distances. The edge reduction then eliminates non-highway edges that only form shortest paths of small length. The bidirectional query in this hierarchy consequently proceeds only in higher levels. Whereas contraction hierarchies assign a distinct "importance level" to each node. Then, the nodes are contracted in order of importance by removing them from the graph and adding shortcuts to preserve the shortest-path distances between the more important nodes (Gutman, 2004).

ALT depends on A*. Using the triangle contrast, strong bounds on shortest-path distances can be gained by pre-computing distances to a set of landmark nodes that are well spread over the far ends of the network. Using reasonable space and much less pre-processing time than for edge labels, these lower bounds return remarkable speedup for route planning. Whereas Edge Labels pre-calculate information for an edge which identifies a set of nodes. Faster pre-computation is possible by dividing the graph into many regions with a small number of boundary nodes (A. V. Goldberg & Harrelson, 2005).

A more effective algorithm usually results by using a mixture of the previously mentioned techniques than one single technique. Using a hierarchical technique then applying a goal-directed technique is a common way that is applied only on a core of the most important nodes identified by the hierarchical technique. This considerably reduces the pre-processing time and space of the goal-directed technique, and speeds up the query.

REAL is a combination of **RE**ach and **AL**t. It stores landmark distances only with the nodes that have high reach values, which in result can significantly reduce memory consumption (Andrew V. Goldberg, Kaplan, & Werneck, 2006).

SHARC combines **SH**ortcuts and multi-level **ARC** flags. Shortcuts allow to unset arc flags of edges that are represented by the shortcut, decreasing the search space. The query algorithm can be unidirectional, which is useful in scenarios where bidirectional search is forbidden. However, a bidirectional query algorithm is faster in the basic scenario (Bauer & Delling, 2009).

Core-ALT iteratively holds nodes that do not require too many shortcuts. Then, on the remaining Core, a bidirectional ALT algorithm is applied. As source and target node of a query are not necessarily in the core, proxy nodes in the core are used. The best proxy nodes are the core entry and exit node of a shortest path for source to target. However, as the query wants to compute a shortest path and does not know it in advance, the core nodes that are closest to source and target are used (Bauer et al., 2008).

CHASE combines **C**ontraction **H**ierarchies and **A**rc flag**S**. Initially, a complete contraction hierarchy is created. After that, on a core of the most important nodes, including shortcuts, arc flags are computed. The query requires only to use the edges with arc flags set for one of the regions where the other search direction has an entry node into the core. As a result, a very fast algorithm is gained, only algorithms based on transit nodes are faster (Geisberger, Sanders, Schultes, & Delling, 2008).

2.2. Matching algorithms in ridesharing

In dynamic ridesharing, the shortest-path problem is enhanced by the problem of matching rides to requests in a dynamic setting. Several studies have been published that tackle the matching problem in dynamic ridesharing.

Agatz, Erera, Savelsbergh, and Wang (2011) develop optimization-based approaches for dynamic ridesharing with a rolling horizon strategy assessed in a simulation study. They show that matching algorithms are important to make dynamic ridesharing useable. Their results suggest that dynamic ridesharing has potential in large US cities. Geisberger, Luxen, Neubauer, Sanders, and Volker (2010) provide an algorithm that finds suitable matches with the smallest detour. Ghoseiri, Haghani, and Hamedi (2011) offer a rich model that not only considers the match of the routes but also age, gender, smoking, pet restriction and maximum occupancy preferences. However, the problem was only solvable for small problem instances. Jung, Jayakrishnan, and Park (2013) provide solution techniques for optimization problems in dynamic ridesharing. Beside simulation, they use a simulated annealing heuristic to identify promising solutions. C.-C. Tao and Chen (2007) and C. C. Tao and Chen (2008) provide algorithms for the related problem of dynamic rideshare matching for taxipooling services. They suggest greedy heuristics and apply them in a field trial in Taiwan.

We enhance this literature by developing a simple yet powerful algorithm that can form the basis of a dynamic ridesharing service. Instead of using a complex optimization approach, we apply a smart data structure to increase the calculation speed of matches.

3. The dynamic ridesharing approach

In this section, we will explain how our dynamic ridesharing approach performs the matching of ride offers and requests. We show how route data can be stored and retrieved efficiently by the inverted index data structure. Based on this, we will explain how the algorithm for dynamic ridesharing matching works.

3.1. Inverted Index Data Structure

When a driver wants to offer a ride he has to create a ride offer $X = (X_s, X_D, X_T, X_F)$ in the system, where X_S is the driver departure location, X_D is the driver destination location , X_T is the driver departure time and X_F is the number of free seats in the driver's car (free places). After getting these information, the departure and destination addresses are geocoded using Google API, then the shortest path is created using GraphHopper with Dijsktra's algorithm.

Consequently, whenever a ride offer is created, the system generates a set of nodes that composes the shortest route between the source and destination points of the ride. In order to get the shortest route nodes N (the set of nodes composing a route), only X_s and X_D are required. We applied the inverted index data structure on the mechanism of saving ride's nodes. An inverted index is a structure used by different search engines and databases to attach search terms to files or documents, increasing the speed of later searches. There are two versions of an inverted index, a record-level index that tells which documents contain the term and a fully inverted index which tells additionally about the location of the term in the file. But here, we will use the record-level index since we only need to save a node (the term) which points to the routes (documents) that contain this node. For example, let a set of ride offers X_1, X_2, X_3, X_4 contain the following nodes:

$$\{X_1\} - [N_a, N_b, N_c, N_e, N_f, N_g]$$
$$\{X_2\} - [N_a, N_f, N_g, N_h, N_i, N_j, N_k]$$
$$\{X_3\} - [N_b, N_h, N_k, N_l, N_m]$$
$$\{X_4\} - [N_l, N_m, N_a, N_o, N_p, N_q, N_r, N_s]$$

Each node inside each ride offer has a sequence number (node order). It helps to know the predecessor node and the successor node of any chosen node along the route of the ride offer X. For example, nodes of ride offer X_1 are presented in Figure 1. Then we store them in a record-level inverted index as shown in Figure 2.

X_1

Node	Node order
N_a	1
N_b	2
N_c	3
N_e	4
N_f	5
N_g	6

Fig. 1. Ride offer representation (Source: own analysis)

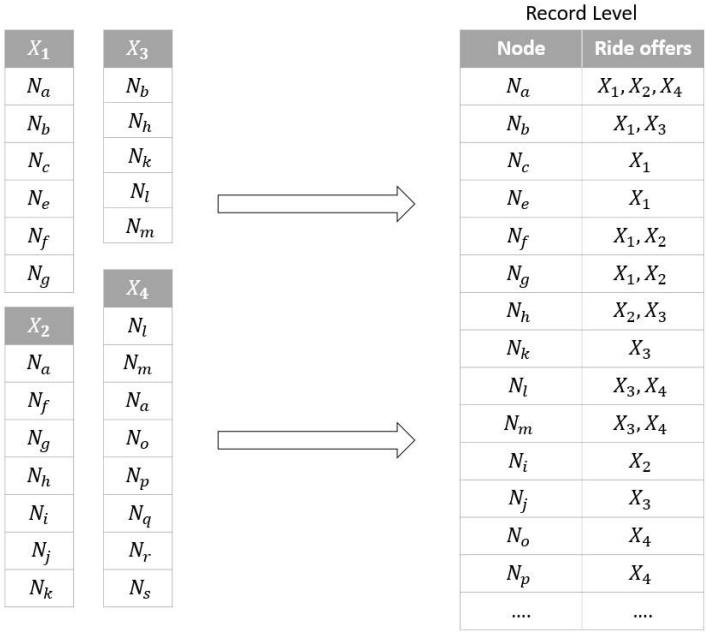

Fig.2. Record-level inverted index of rides offers' nodes (Source: own analysis)

In order to evaluate the benefits of the inverted index data structure, an assessment of the query cost is necessary. There are many possible ways to estimate cost, e.g., based on disk accesses, CPU time, or communication overhead. Disk access is the predominant cost (in terms of time) and it is relatively easy to estimate. Therefore, the number of

block transfers from/to disk is typically used as measure. Also cost of the algorithm (e.g., for join or selection) depends on database buffer size; more memory for DB buffer reduces disk accesses. Thus, DB buffer size is a parameter for estimating cost.

Record level table is ordered based on column 'Node', then the cost of selecting a row from record level table is 1 since Node is the primary key and the cost of the selection equals to $\lceil log_2(B_R) \rceil$ where B_R is the number of blocks that contains tuples of the relation R (Garcia-Molina, Ullman, & Widom, 2009).

3.2. Matching process

In order to make a ride share happen, a ride offer and a demand for a ride have to be matched. Rides of drivers are already stored in the system using the inverted index data structure. At the same time, a passenger may search for a ride $R = (R_s, R_D, R_T)$, where R_S is the passenger departure location (source), R_D is the passenger destination location (destination) and R_T is the trip departure time of passenger. After that, the system will start calculating the shortest path between the start and end locations and returns a set of nodes of the route between the mentioned points.

This process aims to get the most suitable rides for passenger request in real-time with respect to a set of conditions added by the system or the driver. In the following section we will discuss a single ride request with a single ride offer scenario and a single ride request with multiple ride offers scenario as samples.

In the single ride request with single ride offer scenario, we discuss the case of having only one ride offer created by a driver and see how the system will handle a passenger ride request. Assuming that there is one ride offer X, in the system, the set of nodes composing the shortest route between the source X_S , and destination locations X_D, is denoted by this equation:

$$SP(X_S, X_D) = (X_1, X_2, X_3, \dots, X_n) \qquad (1)$$

$SP(X_S, X_D)$ represents the set of nodes that makes the shortest route between X_S and X_D, whereas $X_1 = X_S$ and $X_n = X_D$. Similarly, assuming that R represents the ride request from a passenger, the set of nodes that makes the shortest route between the departure location and the destination location of the passenger is represented by the following equation:

$$SP(R_S, R_D) = (R_1, R_2, R_3, \dots, R_n) \qquad (2)$$

Where R_S and R_D represents the departure and destination location of the passenger respectively, whereas $R_1 = R_S$ and $R_n = R_D$ in the above equation. The matching process algorithm starts by collecting the geographical nodes around departure location R_S and destination location R_D within radius r. We represent the set of nodes around departure location by C_S and the set of nodes around the destination location by C_D. In order to get the ride offers X that are close to ride request R, we check which ride offers are close to the departure and destination locations of the ride request. This can be done by applying two intersections as the following:

$$S_S = C_S \cap SP(X_S, X_D) \qquad (3)$$

where S_s is the set of intersected nodes around the departure location.

$$S_D = C_D \cap SP(X_S, X_D) \qquad (4)$$

where S_D is the set of intersected nodes around distention location. After getting intersection sets S_S and S_D, we check if both sets S_S and S_D are not empty and the nodes of the sets belong to the same ride offer X. If the conditions are satisfied, we look for the closest node in set S_S to passenger departure location R_S (in Figure 4 node X_a is the closest node), then we do the same steps for set S_D and we check for the closest node to passenger destination location R_D, which is X_b.

S_S

Node	Distance to R_S
X_a	*0.45*
X_l	0.46
X_m	0.47
X_n	*0.48*
X_o	0.49

S_D

Node	Distance to R_D
X_b	*0.41*
X_r	0.42
X_s	0.44
X_t	*0.45*
X_u	0.46

Fig 3. Distances of intersected nodes in both sets compared to their centres (Source: own analysis)

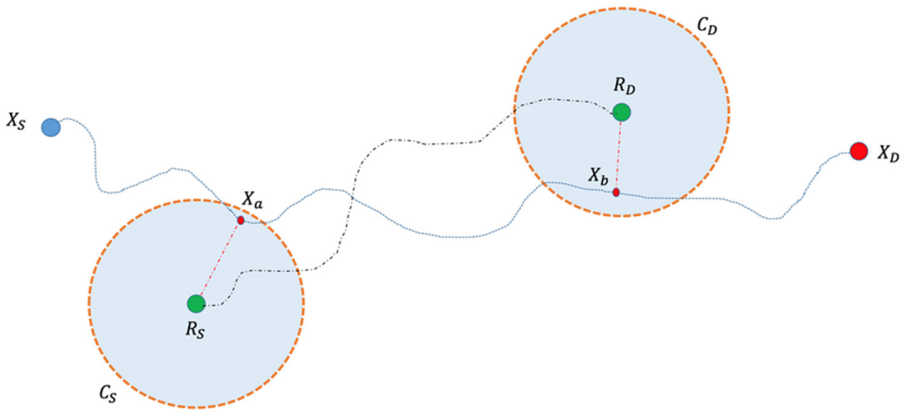

Fig 4. Matching process - single ride offer single ride request scenario (Source: own analysis)

The next step is to check if the route X (ride offer) has the same direction of route R (ride request). In order to check that, we check the node sequence (Node Order column, Figure 1) of X_a which is the closest node to passenger departure location and X_b which is closest node to passenger destination location in route X. We say that X has the same direction as R if and only if:

$$Sequence(X_a) < Sequence(X_b) \qquad (5)$$

Since X consists of a number of nodes and each node has sequence corresponding to its path. From Figure 1, we see that the lower the node order is, the closer it is to the source location and this is how we figure out rule (5) and therefore Sequence is a number that helps to show the direction of the path. We already showed that X_a is the closest point to the source location of both the passenger route R and the driver route X. Finally, if all conditions above are fulfilled, the system will send ride offer X to the user so he could join the ride offer.

In the single ride request with multiple ride offers in the system scenario, we show a case of having one ride request R (from R_S to R_D) and two ride offers X_1 (X_{1_S} is the departure location and X_{1_D} is the destination location) and X_2 (X_{2_S} is the departure location and X_{2_D} is the destination location).

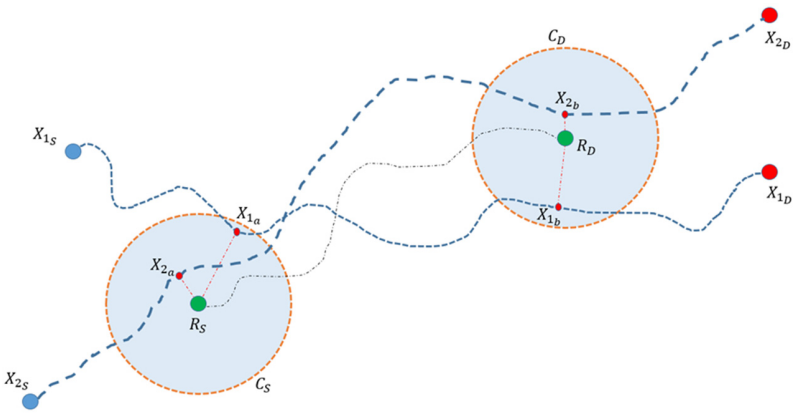

Fig 5. Single ride request, multiple ride offers (Source: own analysis)

As discussed in the single ride request/single ride offer scenario the system will get the nodes of the ride offers and the ride request by applying the shortest path function, so we get the following:

$$SP(R_S, R_D) = (R_1, R_2, R_3, \dots, R_n)$$

where $R_1 = R_S$ and $R_n = R_D$ for the current ride request, and we get the following nodes for the current two ride offers in the system:

$$SP\left(X_{1_S}, X_{1_D}\right) = \left(X_{1_1}, X_{1_2}, X_{1_3}, \dots X_{1_n}\right) \qquad (6)$$

$$SP\left(X_{2_S}, X_{2_D}\right) = \left(X_{2_1}, X_{2_2}, X_{2_3}, \dots X_{2_n}\right) \qquad (7)$$

where $X_{1_1} = X_{1_S}$, $X_{1_n} = X_{1_D}$ for X_1 ride offer, and $X_{2_1} = X_{2_S}$ and $X_{2_n} = X_{2_D}$ for ride offer X_2. Now we apply the same methodology which is used in the previous scenario, which is collecting geographical nodes around the source C_S and destination C_D locations of ride offer R then intersect these nodes with available ride offers in the system (X_1 and X_2). The inverted index structure speeds up the intersection process since it is easy to know who owns each intersected node.

$$S_S = C_S \cap SP(X_{1_S}, X_{1_D}) \cap SP\left(X_{2_S}, X_{2_D}\right) \qquad (8)$$

Now S_S contains all nodes which belong to the ride offers in the system that pass through the circle placed around the source location of R. Then we apply the same intersection around the destination location of R in order to get S_D

$$S_D = C_D \cap SP(X_{1_S}, X_{1_D}) \cap SP(X_{2_S}, X_{2_D}) \qquad (9)$$

After that, we check the distance between each node in sets S_S, S_D. For example, we suppose that the result of the intersection operation is six nodes in each set that are denoted as $S_S = \{X_{1_a}, X_{1_j}, X_{1_k}, X_{2_a}, X_{2_j}, X_{2_k}\}$ and $S_D = \{X_{1_b}, X_{1_u}, X_{1_v}, X_{2_b}, X_{2_u}, X_{2_v}\}$, and the distances to the center of each circle are as the following:

S_S

Node	Distance to R_S
X_{1_a}	*0.45*
X_{1_j}	0.47
X_{1_k}	0.49
X_{2_a}	*0.22*
X_{2_j}	0.25
X_{2_k}	0.33

S_D

Node	Distance to R_D
X_{1_b}	*0.41*
X_{1_u}	0.42
X_{1_v}	0.44
X_{2_b}	*0.15*
X_{2_u}	0.17
X_{2_v}	0.18

Fig 6. Distances of intersected nodes in both sets compared to their centres (Source: own analysis)

We can notice that nodes X_{1_a} and X_{2_a} have the least distances to R_S in ride offers X_1 and X_2 respectively. In addition, nodes X_{1_b} and X_{2_b} have the least distances to R_D in ride offers X_1 and X_2 respectively. By now in this scenario, we see that there are two ride offers which the passenger can join, so how can the passenger choose the best ride in this case?

We propose a solution to provide to the user the most appropriate ride offer to join. We return the ride which has the least average distance from the closest intersected nodes to R_S and R_D. In our scenario, we pick nodes X_{1_a} and X_{1_b} for ride offer X_1 since they are the closest nodes to R_S and R_D respectively and nodes X_{2_a} and X_{2_b} for the other ride offer X_2 for the same reason. After that, we take the average distance of the two nodes to the center of the circles of each ride offer, and so we figure out the following:

$$\frac{X_{1_a} + X_{1_b}}{2} > \frac{X_{2_a} + X_{2_b}}{2} \qquad (10)$$

As a result, we return to the user the two ride offers but with the priority to ride offer X_2 since it has the least average and the probability to have less detour than X_2 is high.

3.3. Matching process conditions

In order to provide appropriate and convenient ride offers, the system is provided with predetermined conditions that support the matching process. Some conditions are determined by the system itself and users have no involvement, and other conditions are determined by the users to ensure their convenience and satisfaction. The following section will list the conditions that are used by the system:

First, the maximum detour distance defined by the driver: it aims to limit the distance from the original driver's route to the pick-up and drop-off locations of the passenger. For example, the driver says that he can only offer 2 km as a detour for one passenger, then the system will only provide this ride offer for the ride requests which have detour distance less than 2 km.

Second, the number of available seats: this condition is also provided by the driver, it restricts the maximum number of passengers the driver wants to pick. Some drivers do not want to fill all available seats and they are satisfied with one or two passengers to join the ride. The number of available seats is decreased by 1 when a passenger joins the ride.

Third, the circle radius: when scanning the area around departure and destination locations of ride offers, this area is a circle and the radius is provided by the system depending on the number of the ride offers in the system. In other words, if the system has many ride offers in Munich, the circle radius can be reduced since it will fetch many rides and decrease the performance of the system, but if the system has few ride offers then the radius value can be increased in order to get more results.

Fourth, the departure time margin: this condition is also provided by the system. It searches for ride offers within a time-range. For example, if the ride request's departure time is 08:30, the system will search for all ride offers within

08:15 and 08:45 if the value of departure time margin is 15 minutes. In addition, the system will take into consideration the travel time from driver's departure location to passenger's pick-up location.

4. Evaluation

We have evaluated the performance of our system based on the two main processes, ride creation and ride search processes. Joining a ride is simple in terms of execution time, we tested joining a ride on the six sample sets and the result was negligible therefore we will focus on the first criteria.

4.1. Ride offer creation

In order to measure the performance of a ride offer creation, we have to measure the performance of the components that compose the ride offer creation process. Creating a ride offer has three main components, we will discuss each one separately:

First, ride offer creation includes getting the ride offer information, i.e. the address' geocode using the Google API. A sample of 10 random addresses around Munich has been prepared, we noted that the average time of sending a geocode request and receiving a response is only 0.094711 seconds. Therefore, we will ignore measuring the performance of the geocoding process since it depends on the network traffic and the processing time does not affect the whole process.

Second, ride offer creation includes generating the shortest path. In order to test the performance, we needed to have a bunch of ride offers in the system. It was not possible to bring real data and create real rides, so we chose to create random ride offers around Munich using a random function depending on java.util.Random. Random ride offers are created within a box around Munich, we restrict the coordination of departure and destination locations of random ride offers to be between [48.2791, 48.0340] for latitude and [11.7986, 11.3599] for the longitude. Figure 7 shows the area that contains the generated random ride offers.

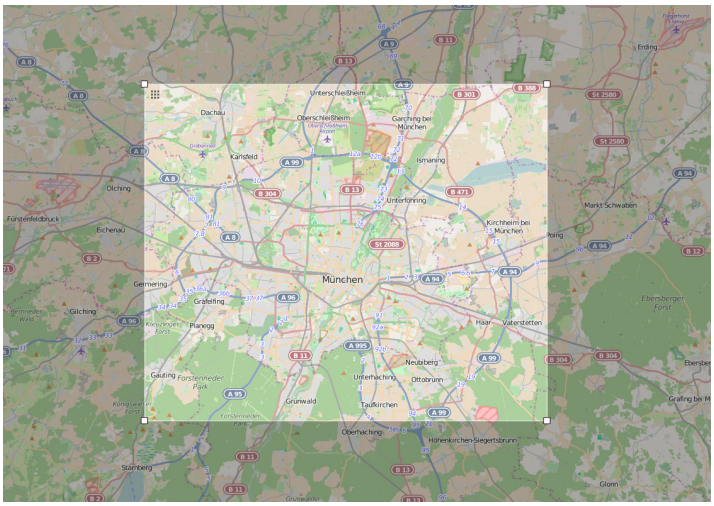

Figure 7. The area of generated random ride offers (Source: own analysis based on OpenStreetMap (OpenStreetMap, 2016))

In Table 1 below, it is clear that constructing the shortest path and getting the geographical coordinates of its points does not take that much time. For example, to create Ride 3 it only took 0.0754 seconds knowing that the distance of the ride is about 130 km and it has 1409 points. Therefore, we do not have to focus on measuring this component as well.

Third, ride offer creation includes adding the shortest path to the database. This part took the longest execution time when creating a ride offer. We measured the performance on three ride offers creation, the distance of Ride 1 is

about 4 km, the distance of Ride 2 is about 19 km, and the distance of Ride 3 is about 130 km. It is clear that adding a ride offer to the database takes longer in terms of time if the distance is longer. Ride 3 for example was not a good example for dynamic ridesharing systems, since these systems depend on short-distance rides in their definitions. However, we wanted to check the performance when we have very long ride offers.

Table 1.: Measuring the performance of ride offer creation (Source: own analysis)

	Number of Points	**Distance** (m)	**Request time** (sec)	**Save in DB time (sec)**	**Total time (sec)**
Ride 1	49	4,008.61	0.056657	0.054683	0.11134
Ride 2	291	18728.91	0.018947	1.007314	1.026261
Ride 3	1,409	130,111.26	0.075489	2.652860	2.728349

In order to show ride offers distribution of the random function that we implemented, we created 50 ride offers, then we checked the geographical points that each ride offer contains and finally we divided them into groups. As we notice from table 2 below, most ride offers have between 151 and 500 geographical points and the average distance is about 2.3 km. In addition, the time to create 50 ride offers is 35.824 seconds (0.71648 seconds per ride offer on average).

Table 2.: Ride Offers Random Distribution (Source: own analysis)

	0-50 Points	**51-150 Points**	**151-300 Points**	**301-400 Points**	**401-500 Points**	**500> Points**
Number of ride offers	0	9	23	14	4	0

4.2. Search for ride offer

In order to evaluate the search for ride offer performance, we created five sample sets. Each set contains a specific amount of ride offers, we chose to measure the performance when we have 50, 100, 500, 1,000 and 10,000 ride offers in the system (Table 3). For the testing purpose, a maximum of 10,000 rides at a time is sufficient. While Munich for example has more than 400,000 commuters, almost half of them are using public transportation. Considering the distribution of commutes over time and the typically low adaption rates of ridesharing application, 10,000 simultaneous rides would be a sufficient capacity of the system. In Table 3, *nodes count* shows the total number of nodes for all ride offers in the system. The nodes count includes *unique nodes* and *duplicated nodes* that are part of several routes.

Table 3.: Ride Offers Sample Sets (Source: own analysis)

	50 Rides	**100 Rides**	**500 Rides**	**1000 Rides**	**10000 Rides**
Nodes count	7,210	35,365	116,542	240,117	2,402,286
Unique nodes	2,289	15,431	27,145	36,646	89,543
Duplicated nodes count	4,921	19,934	89,397	203,471	2,312,743
Average distance per ride offer (m)	25,234.26	23,389.33	21,943.93	22,809.05	23,025.16

After we created our sample sets, we tested the performance of searching for a ride offer on each set. We have considered six major steps while searching for a ride offer. For a ride request, the algorithm needs to (1) identify rides around the departure location of the passenger, (2) identify rides around the destination location of the passenger, (3) intersect the two sets of rides, (4) check whether the departure time of the rides in the intersected set matches the departure time condition of the passenger, (5) check whether the new route would violate the maximum detour condition of the driver and (6) verify whether the direction of the ride is similar to the direction of the ride request.

In a first trial, we realized that identifying rides around departure and destination locations is the most time consuming step. In order to reduce this calculation time, we will replace the idea of scanning system nodes around departure and destination locations with checking if the node is within a square, and the center of this square is either the departure or the destination point (Figure 8). By this solution, we only have to know the maximum and minimum longitude/latitude around departure/destination point and provide them just once to the query before we run it, so there will be no need for further calculations.

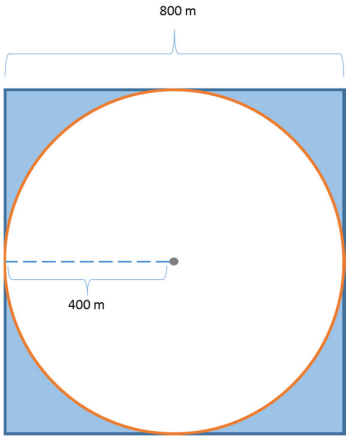

Figure 8. Search for Ride Offer- Square Edge (Source: own analysis)

After adapting the first two steps of the algorithm to a solution based on the square area, we simulated 100 random searches for rides for 50, 100, 500 and 10,000 rides in the system, assuming a 400 m radius as acceptable for the passenger. The average calculation times are given in Table 4. We can see that identifying possible rides that lay within the preset radius is still the most time consuming step. However, an overall calculation time of less than 0.4 seconds for a system with 10,000 rides shows that the performance is sufficient for real-time applications.

Table 4.: Calculation time for search for ride offers with 400 m radius (Source: own analysis)

Step		50 Rides	100 Rides	500 Rides	1,000 Rides	10,000 Rides
(1)	Rides around departure **(sec)**	0.00276	0.008364	0.02279	0.04056	0.0989
(2)	Rides around destination **(sec)**	0.002065	0.00158	0.00281	0.007450	0.0114
(3)	Intersection **(sec)**	0.000002	0.000006	0.000016	0.000047	0.00249
(4)	Check intersection departure time **(sec)**	0.000105	0.000117	0.000120	0.00014	0.1034
(5)	Check max detour condition **(sec)**	0.000001	0.000004	0.000004	0.000005	0.1791
(6)	Check ride offers direction **(sec)**	0.000001	0.000001	0.000001	0.000001	0.0028
	Total **(sec)**	0.004934	0.010072	0.02574	0.048203	0.39809

5. Discussion

Our dynamic ridesharing approach has been shown to perform well enough for real-time applications while being simpler than existing optimization based approaches. The gain in performance is reached by taking advantage of an inverted index data structure that reduces the query cost for each database query. As with any research, there are a number of limitations associated with this study. Firstly, testing samples did not carry realistic data since it was not possible to get real trips that were offered by students in Munich. Therefore, we used a random function to generate ride offers within Munich area. Secondly, using Google API for geocoding is a good solution but not the best since it increases ride offer and ride request execution time, especially if the server is handling many requests at the same time. Furthermore, Google API geocoding service is restricted to a certain number of requests per minute. Still, the provided solution in this paper has full potential to increase the efficiency and performance of existing ridesharing services in order to make them more dynamic for matching ride requests with ride offers. Besides that the obstacles such as rivers, parks or railway lines are not considered while drawing the circles around departure and destination points, they are only taking into consideration when doing the routing using GraphHopper. Offered as a web service reachable via an API it could contribute to projects such as TUMitfahrer, a ridesharing application for students of the Technical University of Munich.

For further research, we propose to take the real time traffic situation into ride-matching process calculations while searching for ride offers around departure and destination points. In this scenario, instead of collecting geographical nodes around a point within a fixed radius, now we should collect geographical nodes that are reachable within a defined amount of time around a point (the center). Another potential future work is to run a geocoding database on the server, this solution saves the time needed to retrieve geocoding data from Google's server. Consequently, there will be no risk of limiting the requests count from Google API. An alternative solution would be to geocode addresses on the user's side (mobile app) and send the coordinates directly to the server, at this stage, the server will not handle geocoding part and this lets it enhance the overall performance. Last but not least, the suggestions to use public transport options for the passenger before and after sharing the vehicle can also be integrated in the service.

6. Conclusion

The ever-growing density of smartphones and tablets all over the world, their pervasiveness among the population, and their availability on the urban territory at no cost for the public administration make them extremely valuable resources which have opened the doors for smart mobility everywhere. One example is dynamic ridesharing that has become an important component of smart mobility. To facilitate dynamic ridesharing, a dynamic ridesharing service

has to match ride offers and requests in real-time. In this paper we propose a fast algorithm that can be used in any ridesharing application as matching model for ride requests and ride offers in order to make that service more dynamic and efficient.

Acknowledgements

We thank the German Federal Ministry for Economic Affairs and Energy for funding this research as part of the project 01MD15001D (ExCELL).

References

Agatz, N. A. H., Erera, A. L., Savelsbergh, M. W. P., & Wang, X. (2011). Dynamic ride-sharing: A simulation study in metro Atlanta. *Transportation Research Part B: Methodological, 45* `(9), 1450-1464. doi:10.1016/j.trb.2011.05.017

Arena, M., Cheli, F., Zaninelli, D., Capasso, A., Lamedica, R., & Piccolo, A. (2013, 3-5 Oct. 2013). *Smart mobility for sustainability.* Paper presented at the AEIT Annual Conference, 2013.

Baláž, P., & Londarev, A. (2006). Oil and its position in the process of globalization of the world economy. *Politická ekonomie, 2006*(4), 508-528.

Bauer, R., & Delling, D. (2009). SHARC: Fast and robust unidirectional routing. *Journal of Experimental Algorithmics (JEA), 2*, 13-26. doi:10.1145/1498698.1537599

Bauer, R., Delling, D., Sanders, P., Schieferdecker, D., Schultes, D., & Wagner, D. (2008). Combining hierarchical and goal-directed speed-up techniques for Dijkstra's algorithm. *Lecture Notes in Computer Science, 5038 LNCS*, 303-318. doi:10.1007/978-3-540-68552-4_23

Bin, C., Alarabi, L., Mokbel, M. F., & Basalamah, A. (2015, 15-18 June 2015). *SHAREK: A Scalable Dynamic Ride Sharing System.* Paper presented at the Mobile Data Management (MDM), 2015 16th IEEE International Conference on.

Dailey, D. J., Loseff, D., & Meyers, D. (1999). Seattle smart traveler: dynamic ridematching on the World Wide Web. *Transportation Research Part C: Emerging Technologies, 7*(1), 17-32. doi:http://dx.doi.org/10.1016/S0968-090X(99)00007-8

Furuhata, M., Dessouky, M., Ordóñez, F., Brunet, M. E., Wang, X., & Koenig, S. (2013). Ridesharing: The state-of-the-art and future directions. *Transportation Research Part B: Methodological, 57*, 28-46. doi:10.1016/j.trb.2013.08.012

Garcia-Molina, H., Ullman, J. D., & Widom, J. (2009). *Database Systems - The Complete Book*. Upper Saddle River, NJ: Pearson Education Inc.

Geisberger, R., Luxen, D., Neubauer, S., Sanders, P., & Volker, L. (2010, 2010). *Fast Detour Computation for Ride Sharing*.

Geisberger, R., Sanders, P., Schultes, D., & Delling, D. (2008). Contraction Hierachies: Faster and Simpler Hierachical Routing in Road Networks. *Proceedings of the 7th Workshop on Experimental Algorithms (WEA 2008), 2*, 319-- 333.

Ghoseiri, K., Haghani, A., & Hamedi, M. (2011). Real-Time Rideshare Matching Problem. *Thesis*.

Goldberg, A. V., & Harrelson, C. (2005). Computing the Shortest Path: A* meets Graph Theory. *16th ACM-SIAM Symposium on Discrete Algorithms*, 156-165.

Goldberg, A. V., Kaplan, H., & Werneck, R. F. (2006). Reach for A*: Efficient Point-to-Point Shortest Path Algorithms. *In workshop on algorithm engineering & experiments*(October), 129-143. doi:10.1.1.65.521

Gutman, R. (2004). Reach-based routing: A new approach to shortest path algorithms optimized for road networks. *Proceedings of the Sixth Workshop on Algorithm Engineering and Experiments and the First Workshop on Analytic Algorithms and Combinatorics*, 100-111.

Jung, J., Jayakrishnan, R., & Park, J. Y. (2013). Design and Modeling of Real-time Shared-Taxi Dispatch Algorithms. *TRB Annual Meeting, 8*, 1-20.

Krcmar, H. (2016). TUMitfahrer. Retrieved from https://www.tumitfahrer.de/

May, A. D. (2013). Urban Transport and Sustainability: The Key Challenges. *International Journal of Sustainable Transportation, 7*(March 2015), 170-185. doi:10.1080/15568318.2013.710136

OpenStreetMap. (2016). OpenStreetMap. Retrieved from http://www.openstreetmap.de/

Schultes, D. (2008). *Route Planning in Road Networks.* (PhD), Universität Karlsruhe, Karlsruhe.

Stach, C. (2011, 21-25 March 2011). *Saving time, money and the environment - vHike a dynamic ride-sharing service for mobile devices.* Paper presented at the Pervasive Computing and Communications Workshops (PERCOM Workshops), 2011 IEEE International Conference on.

Tao, C.-C., & Chen, C.-Y. (2007). Heuristic Algorithms for the Dynamic Taxipooling Problem Based on Intelligent Transportation System Technologies. *Fourth International Conference on Fuzzy Systems and Knowledge Discovery*, 590-595. doi:10.1109/FSKD.2007.346

Tao, C. C., & Chen, C. Y. (2008). Dynamic rideshare matching algorithms for the taxipooling service based on intelligent transportation system technologies. *Proceedings of 2007 International Conference on Management Science and Engineering, ICMSE'07 (14th)*, 399-404. doi:10.1109/ICMSE.2007.4421880

Ying, F., Yu, F., Changjun, J., & Jiujun, C. (2008, 20-22 Oct. 2008). *Dynamic Ride Sharing Community Service on Traffic Information Grid.* Paper presented at the Intelligent Computation Technology and Automation (ICICTA), 2008 International Conference on.

Yunfei, H., Xu, L., & Chunming, Q. (2012, 3-7 Dec. 2012). *TicTac: From transfer-incapable carpooling to transfer-allowed carpooling.* Paper presented at the Global Communications Conference (GLOBECOM), 2012 IEEE.

Zhang, F., & Liu, J. (2009, 14-16 Aug. 2009). *An Algorithm of Shortest Path Based on Dijkstra for Huge Data.* Paper presented at the Fuzzy Systems and Knowledge Discovery, 2009. FSKD '09. Sixth International Conference on.