Single Transfer Journeys in Dynamic Taxi Sharing

Bachelor's Thesis of

Johannes Breitling

At the KIT Department of Informatics
ITI - Algorithm Engineering

First examiner:	Prof. Peter Sanders
Second examiner:	Prof. Thomas Bläsius
First advisor:	M.Sc. Moritz Laupichler
Second advisor:	M.Sc. Michael Zündorf

01. November 2024 – 03. March 2025

Karlsruher Institut für Technologie
Fakultät für Informatik
Postfach 6980
76128 Karlsruhe

Single Transfer Journeys in Dynamic Taxi Sharing (Bachelor's Thesis)

Karlsruhe, 03. March 2025

. . . .

(Johannes Breitling)

Abstract

Over the course of the last years, commercial mobility providers introduced taxi sharing services like *UberXShare*, managing dynamic fleets of vehicles and allow for more flexible and cheaper solutions than traditional taxis do. We propose an algorithm that computes single-transfer journeys for the *dynamic taxi sharing* problem. When taking on a single-transfer journey, the passenger is allowed to transfer between vehicles exactly once. Therefore, we extend the state-of-the-art dynamic taxi sharing dispatcher KaRRi. We show that our algorithm improves the no-transfer solution for up to 16.8% of the requests. We manage to reduce the total vehicle operation time by up to 5.7% while providing similar service quality for passengers at the expanse of increased running time. The running time of our algorithm drastically increases due to the combinatorial explosion of possible assignments, that now have to consider two vehicles with feasible pickups and dropoffs as well as many possible transfer points. Providing a framework for algorithms computing single-transfer journeys, our work serves as the basis of future work for efficient filter heuristics and advanced transfer point calculation techniques.

Zusammenfassung

Kommerzielle Mobilitätsdienstleister brachten im Laufe der letzten Jahre zunehmend Taxi-Sharing Dienste auf den Markt. Dienste wie *UberXShare* verwalten dynamische Flotten an Fahrzeugen und bieten Endnutzern günstigere und flexiblere Lösungen wie traditionelle Taxis. Wir stellen einen Algorithmus vor, der Single-Transfer Fahrten für das *dynamic taxi sharing* Problem berechnet. Bei einer Single-Transfer Fahrt darf der Nutzer genau einmal zwischen Fahrzeugen wechseln. Um Single-Transfer Fahrten zu berechnen, erweitern wir den State-of-the-Art dynamic taxi sharing dispatcher KaRRi. Wir zeigen, dass unser Algorithmus in bis zu 16.8% der Anfragen eine verbesserte Lösung finden kann. Weiter zeigen wir, dass unser Algorithmus die Betriebszeit der Taxis um bis zu 5.7% senkt, während den Nutzern trotzdem eine ähnliche Nutzungsqualität geboten wird. Allerdings steigt die Laufzeit unseres Algorithmus druch die kombinatorische Explosion der möglichen Lösungen, da diese nun mögliche Abhol- und Absetzpunkte, sowie viele mögliche Transferpunkte berüclsichtigen müssen. Wir bieten ein Gerüst für Algorithmen, die Single-Transfer Fahrten berechnen und legen damit die Grundlage für zukünftige Arbeiten für effiziente Filter Heuristiken und weiterentwickelten, schnellern Methoden Transferpunkte zu berechnen.

Contents

List of Figures

List of Tables

1 Introduction

Future transportation faces numerous challenges, including increasing urbanization, the environmental impact of the transport sector, higher attention drawn to sustainability by society and the need to keep up with new technologies [2]. Ridesharing seems to offer an attractive alternative to conventional taxis or individual car usage as it offers lower fares for passengers as well as reduced vehicle operation necessary for providers [8]. In ridesharing, a passengers issues a (commercial) system, e. g. via an app, to travel from a specified origin to a specified destination. The system assigns a vehicle among a fleet of vehicles to satisfy the request. The vehicles are shared among passengers, therefore it is possible to drive detours to pick up or drop off others. Due to its commercial nature, we also refer to ridesharing as taxi sharing. Ridesharing systems seamlessly integrate with modern technology, offering an enhanced user experience compared to traditional taxis. Commercial providers like UberXShare [19] already established solutions for shared taxis. While the concept of transferring between vehicles is an everyday scenario for most means of public transportation like buses, trains or trams, taxi and taxi sharing solutions almost exclusively serve single-leg journeys.

In this work we apply the common concept of transfers between vehicles to the dynamic ridesharing problem. We extend the state-of-the-art Karlsruhe Rapid Ridesharing dispatcher (KaRRi) for the dynamic taxi sharing problem (DTSP). We build a framework that computes solutions for the DTSP with exactly one transfer and evaluate it on realistic input data from the MATSim Open-Berlin scenario [21]. When introducing single-transfer journeys, a trip of a passenger now consists of one or two vehicles. To determine the best solution for a given ridesharing request, optimal combinations of pickup vehicle, pickup point, transfer point, dropoff vehicle and dropoff point have to be found and assessed. This results in a drastic increase in possible assignments that have to be evaluated. To assess the quality of a given assignment, multiple shortest path problems have to be solved to calculate the detour that is needed to satisfy a given request.

After giving an overview of prior research related to our problem in chapter 2, and explaining important route planning concepts that are necessary to understand our work, we establish an theoretical model of the DTSP with meeting points and journeys that allow exactly one transfer in chapter 4. We then outline the algorithmic foundation to find optimal single-transfer journeys and describe the limitations of our solution in chapter 5. In chapter 6 we study the details of the subproblems the algorithm has to solve to compute single-transfer journeys and how we chose to solve them for our implementation. We evaluate the chosen approaches comprehensively in chapter 7 and round off our work with a conclusive reflection and future outlook on the conducted research in chapter 8. While the runtime of our algorithm increased drastically, we were able to improve the no-transfer solution for to 16.8% of the requests. Additionally, we manage to reduce the total vehicle operation time by up to 5.7% while providing similar service quality for passengers at the expanse of increased running time.

2 Related Work

2.1 Dial-A-Ride Problem

The *dial-a-ride problem* (DARP) is a tour-planning problem for individual transport. A passenger issues a request to travel from an origin to a destination. In most problem formulations, the DARP algorithm matches a passengers request to a set of homogeneous vehicles stationed at one depot that then executes the trip of the passenger. The requests are either processed *dynamically*, where they are gradually revealed over time, also called the *online* version of the problem, or statically, where all requests are known in advance [7]. The DARP is widely known and extensive research has been conducted on it. The DARP has applications e. g. in delivery scheduling or door-to-door transportation services [18]. The dynamic DARP serves as the basis of the dynamic taxi-sharing problem (DTSP), considered in our research.

There already have been approaches to extend the DARP by allowing a passenger to transfer between vehicles [16, 9]. These efforts showed improvements in travel time and vehicle total vehicle operation time on real-life and generated instances. While most of the research on the DARP is focused on the economical optimization of a fleets operation by designing efficient transport plans, often computing solutions statically using methods from the field of operations research, we focus on the dynamic route planning aspects of the taxi sharing problem while also valuing quality of service for the individual passenger.

2.2 Dynamic Taxi Sharing

The problem of *dynamic taxi sharing* has been studied in many works. Various simulation studies for urban regions like Berlin or New York City concluded that, in comparison to regular taxis, total vehicle operation time can be decreased by well over 20% while increasing average vehicle occupancy and therefore reducing carbon emissions by a significant fraction [3, 15]. Various approaches have been implemented in order to simulate dynamic taxi dispatching. One of the dispatchers, LOUD (Local buckets dispatching), presents a novel, state-of-the-art solution for the calculation of optimal assignments in the DTSP [5]. It utilizes advanced route planning techniques such as bucket contraction hierarchies (BCHs) and elliptic pruning. Thus, LOUD is able to calculate optimal assignments for a request in a matter of milliseconds. KaRRi [14] extends the LOUD dispatcher by introducing meeting points to the dynamic taxi sharing problem. By allowing passengers to walk from their origin to a mutual meeting point with the vehicle, total vehicle operation times can be decreased by up to 15%. KaRRi overcomes the increasing complexity that the many-to-many routing problem between multiple pickup and dropoff locations by applying advanced route planning techniques, calculating optimal assignments in a few milliseconds.

2.3 Transfers in the Dynamic Taxi Sharing Problem

Integrating transfers between vehicles in the dynamic taxi sharing is not new. There are a few different approaches to calculate trips for passengers that include the transferring between vehicles. Hou et al. [12] introduce TASeT, a dynamic taxi sharing algorithm for electric taxis, that allows single-transfer journeys with transfer at charging locations that are part of the electric vehicle infrastructure. They optimize solutions using mixed integer programming. Chen, Ssu, and Chang also provide an algorithm to solve the DTSP introducing *real-time transfers* and different filter heuristics [6]. The algorithm first finds an initial no-transfer solution for the request. Then, for every potential road the vehicle drives along, that serves as a potential transfer point, a new no-transfer request is executed to check if this new request would improve the remaining cost to drive the passenger to their destination. This means that depending on the current state of the system (that is affected by requests made after the departure time of the vehicle initially assigned to the passenger), rescheduling is potentially performed. This contrasts other approaches, as e. g. in LOUD the initial assignment of the vehicle to a request will not change later [5].

The research that is most closely related to our own work is the transfer dispatching solution presented by Willich [20]. He also studies the problem of single-transfer routes in the DTSP, while also exploring multi-transfer routes. The presented dispatcher also builds upon the LOUD dispatcher [5]. To compute transfer points between vehicles, three different strategies have been implemented: The *transfer-at-stop* strategy, transfer calculation using a *geometric* sampling heuristic and transfer calculation using a *betweenness* heuristic. The transfer-at-stop algorithm considers for every eligible vehicle, that could pick up the passenger, all scheduled stops as possible transfer points. As the name suggests, geometric sampling uses the geometric distance between pickup and dropoff location for a request to determine transfer points that lie somewhere in between those both points. Sampling by betweenness first calculates the betweenness value for every vertex $x \in V$ in the graph, i. e. the number of shortest paths between any pair of vertices $u, v \in V$ that include the vertex. The algorithm then executes two LOUD requests, one from the origin with the transfer point as the destination at the given request time, and one with the transfer point as origin and the dropoff as destination with the arrival time at the transfer as request time. Willich shows, that for single-transfer routes, the heuristic betweenness approach manages to improve no-transfer routes in up to 18% of the requests and improve total vehicle operation time for the vehicles and vehicle utilization. The wait times for passengers could be decreased by around 90 seconds at the expense of increasing average trip time by up to eight minutes.

Willich uses three different heuristics to calculate possible transfer points efficiently in the one-to-many taxi sharing scenario. However, we aim to calculate optimal transfer points for the many-to-many taxi sharing with meeting points by evaluating all feasible transfer points for each request while, for now, neglecting the drastic increase in running time.

2.4 Dynamic Ridepooling

Dynamic Ridepooling is closely related to the DTSP. The main difference between ridepooling and taxi sharing is, that for ride pooling, (private) drivers register their own journeys, that they plan to take on. Therefore, these journeys have a fixed origin and a fixed destination. Other passengers that have similar origins and destinations can now be *pooled* together to share the vehicle along the trip. A commercial solution for ridepooling is BlaBlaCar, popular in France and other European countries [4, 17]. In some scenarios, the vehicles are also allowed to drive detours to pick up additional passengers along the way (e. g. see the *boost* functionality in BlaBlaCar). While ridesharing is mostly applied in urban transport and commuting, ridepooling has wide application for longer transportation and travel beyond city borders [4].

3 Preliminaries

In this section we explain shortest path algorithms that are used in this work. We also give an overview of KaRRi - a dynamic taxi sharing dispatcher with meeting points, that serves as the basis of our research. We comprehensively introduce KaRRis notation as we will extend it in chapter 4.

3.1 Dijkstra's Shortest Path Algorithm

Dijkstra's shortest path algorithm [10] computes the shortest path distances from a single source $s \in V$ to all other vertices $t \in V$ in a Graph $G = (V, E)$ with non-negative weight function $\ell : E \longrightarrow \mathbb{N}_0$. The algorithm stores a distance label $\tilde{\delta}(s, v)$ for every $v \in V$ with the best currently known distance from s to v. It utilizes an addressable priority queue Q with $key(v) = \tilde{\delta}(s, v)$ to store the vertices that are found but have not yet been processed by the algorithm. Initially, $Q = \{s\}$ with $\tilde{\delta}(s, s) = 0$ and $\tilde{\delta}(s, v) = \infty$ for $v \neq s$. The algorithm repeatedly extracts the vertex v with the smallest distance label from Q and *settles* it until Q is empty. To settle a vertex, all outward edges are *relaxed*. When relaxing an edge $e = (u, v)$, the algorithm checks, whether the best currently known distance to v, $\tilde{\delta}(s, v)$, can be improved by using the edge, i. e., whether $\tilde{\delta}(s, u) + \ell(e) < \tilde{\delta}(s, v)$.

When using Dijkstra's algorithm to calculate the shortest path distance to a given target $t \in V$ we can stop the search as soon as t is settled. The distance to t will not change afterwards. Using parent pointers, Dijkstra's algorithm can maintain the shortest path tree, allowing the reconstruction of shortest paths to every settled vertex. We define a *reverse Dijkstra search* for $G = (V, E)$ and source $s \in V$, target $t \in V$ as the Dijkstra search from t to s in the reverse graph $G^{rev} = (V, E^{rev})$ with $E^{rev} = \{(v, u) \mid (u, v) \in E\}$ and $\ell^{rev}(v, u) = \ell(u, v)$.

3.2 Contraction Hierarchies

Contraction hierarchies (CHs) [11] are a technique to speed up shortest path searches in routing problems on road networks. CHs make use of hierarchies implicitly given in a road network. The technique is made up of two phases: A *construction phase* and a *query phase*.

3.2.1 Construction

To construct a CH, a heuristic order of importance for the vertices is required. This order, also called *rank*, is used to *contract* the vertices in the original graph from the vertex with the lowest rank (i. e. importance) to the vertex with the highest rank. When contracting a vertex, the vertex itself is temporarily removed from the graph and *shortcut edges* between the neighbors of the vertex are inserted to preserve all shortest path distances in the remaining graph. When contracting vertex v, a shortcut edge between neighboring u and w is only inserted if $(u, v, w) \in E^2$ is the only shortest path between u and w. We denote the set of all original edges and all shortcut edges as E^+. The Graph $G' = (V, E^+)$ then represents the CH.

3.2.2 Query

In the query phase, we separate the edges into two sets $E^\uparrow = \{(u, v) \in E^+ \mid rank(u) < rank(v)\}$ and $E^\downarrow = \{(u, v) \in E^+ \mid rank(u) > rank(v)\}$ with $E^+ = E^\downarrow \dot\cup E^\uparrow$. Geisberger et al. [11] show that between any two gives vertices s and t, there exists a shortest path in G' that can be separated into exactly two continuous paths. The first of these paths is an *up-path* that only consists of edges $e \in E^\uparrow$. The second path is a down-path that only consists of edges $e \in E^\downarrow$. To efficiently compute this shortest path, the algorithm runs a Dijkstra search on the upwards graph $G^\uparrow = (V, E^\uparrow)$ and a reverse Dijkstra search on the downwards graph $G^\downarrow = (V, E^\downarrow)$. When a vertex is found in both searches, an up-down path between s and t is established. We can stop the searches, if either one of the smallest distance labels of the queues exceeds the best currently found s-t distance. Eventually the shortest up-down path between s and t is found. Because this is also a shortest s-t path, the CH query correctly calculates shortest path distances.

3.3 Bucket Contraction Hierarchy Searches

Bucket contraction hierarchy searches (BCHs) [11, 13] make use of contraction hierarchies to answer one-to-many distance queries for a fixed set of targets. To answer queries for a given set of target vertices $T \subseteq V$, a reverse Dijkstra search for every target $t \in T$ is ran on the downward graph $G^\uparrow$ of the CH. Every vertex v is associated with a *bucket.*

Each bucket holds bucket entries $b_v(t) = \delta^\downarrow(v, t)$ with the downward-distances from the reverse Dijkstra search for every target t. Each bucket holds bucket entries of the form $(t, \delta^\downarrow(v, t))$ with the downward-distance from vertex v to target t, that was acquired during the reverse Dijkstra search from t. To compute the distances from a vertex s to every target t, a forward search on $G^\uparrow$, rooted at s, is performed. For every vertex v reached by the forward search, the s-t distance can be computed by adding the distance $\delta^\uparrow(s, v)$ from s and the distance $\delta^\downarrow(v, t)$ from each bucket entry $(t, \delta^\downarrow(v, t))$ to find a new tentative distance from s to t. By updating the tentative s-t distances accordingly, eventually all shortest s-t paths for $t \in T$ are computed.

3.4 KaRRi and Dynamic Taxi Sharing

KaRRi - Karlsruhe Rapid Ridesharing [14] presents a novel dispatcher to solve the dynamic taxi sharing problem efficiently while also introducing meeting points, where vehicles can drop off and pick up passengers. KaRRi builds upon LOUD [5], a point-to-point dynamic taxi dispatcher that introduced elliptic pruning, a pruning technique also used by KaRRi.

3.4.1 The Dynamic Taxi Sharing Problem

In the *dynamic taxi sharing problem* (DTSP), a passenger can issue a request to the system to travel from an origin to a destination. The system then computes the best assignment of a vehicle to the request of the passenger. The assigned vehicle is tasked with picking up the passenger at their requested origin and dropping them off at their destination. Vehicles are shared among passengers, resulting in possible detours to satisfy multiple passengers requests. To determine the optimal action of the system, potential solutions are quantified and analyzed using a specified cost function.

Road Network

In the dynamic taxi sharing problem, a *road network* is considered to be a directed graph, $G = (V, E)$ with a weight function $\ell : E \longrightarrow \mathbb{N}$ that represents the travel time $\ell(e)$ for a given edge $e \in E$. Shortest path distances between u and v are denoted as $\delta(u, v)$.

Request

A request is a tuple $r = (orig, dest, t_{\text{req}})$ with $orig, dest \in V$. A passenger issues a request to get from their origin location $orig(r)$ to their destination $dest(r)$ at time $t_{\text{req}}(r)$. If the context is sufficient for a given request r, we only write $orig$, $dest$ and t_{req}.

Vehicles

A vehicle v is a tuple $v = (loc_{init}, cap, t_{\text{serv}}^{\text{min}}, t_{\text{serv}}^{\text{max}})$ where each vehicle has an initial location $loc_{init} \in V$, a seating capacity $cap \in \mathbb{N}$ (excluding the driver), and a service time interval $[t_{\text{serv}}^{\text{min}}, t_{\text{serv}}^{\text{max}})$. The algorithm has access to a Fleet F of vehicles that can be used to fulfill passengers requests.

Stops, Routes

For each vehicle v, its current *route* $R = \langle s_0, \ldots, s_{k(v)} \rangle$, a sequence of scheduled *stops*, is maintained. The route is updated so that s_0 is always the previous stop when the vehicle is driving and the current stop when the vehicle is making a stop. We define $k(v)$ as the number of scheduled stops for each vehicle v minus 1. The last stop of v therefore always is $s_{k(v)}(v)$. Each stop $s_i(v)$ is mapped to a location $loc(s_i(v)) \in V$ in the road network. When adequate context is provided, we write s_i instead of $s_i(v)$ and s_i instead of $loc(s_i(v))$. For the departure and arrival time of vehicle v at stop s_i, we write $t_{\text{dep}}(v, s_i)$ and $t_{\text{arr}}(v, s_i)$ respectively. With sufficiently clear context we omit v and just write $t_{\text{dep}}(s_i)$ and $t_{\text{arr}}(s_i)$.

Insertion

An *insertion* ι in the DTSP is a tuple (r, v, i, j) that indicates that vehicle v pickups up the passenger of request r at their origin *orig* immediately after stop s_i and drops them off at their destination *dest* immediately after stop s_j.

Constraints

An insertion has to follow certain constraints given by the overall service. Further, as the DTSP allows shared usage of vehicles, resulting in potential detours of vehicles that already contain other passengers, there is a need to control the reduction of service quality for individual riders. This results in different service quality constraints.

- **Occupancy**: It is evident that a vehicle at any given point in time can only hold at maximum $cap(v)$ passengers. So if the current occupancy for a stop between $s_i(v)$ and $s_j(v)$ is exceeded, the constraint is violated and the cost for the insertion are set to ∞.
- **Service Time**: Further, every dropoff of a given vehicle v has to be performed before its respective end of service time. Insertions that violate this constraint will not be considered by the system.
- **Maximum Trip Time**: The maximum trip time of a request r is given through model parameters α and β. The maximum trip time is defined as follows: $t_{\text{trip}}^{\max}(r) = \alpha \cdot \delta(orig, dest) + \beta$. For every rider already assigned to a vehicle, we consider the maximum trip time as a hard constraint. Every insertion for vehicle v that would cause a passenger, that is already assigned to v, to not reach their destination within the maximum trip time of their request, will not be considered.
- **Maximum Wait Time**: We use a constant maximum wait time to ensure the passenger is picked up within a certain time frame. For every passenger already assigned to the vehicle the departure at the origin $t_{\text{dep}}(orig)$ has to be before $t_{\text{req}}(r) + t_{\text{wait}}^{\max}$. If after an insertion this condition would not hold, the insertion will not be considered.

To check these constraints efficiently, we can encode the constraints in the stops as *scheduled departure times* and *maximum arrival times*. The scheduled departure time is the earliest time where a vehicle can depart at the stop. The maximum arrival time is the latest time the vehicle is allowed to arrive at the stop without breaking constraints of passengers already assigned to the vehicle.

Cost Function

The cost function used by Buchhold, Sanders, and Wagner in LOUD [5] is defined as follows:

$$c(\iota) = t_{\text{detour}}(\iota) + c^{\text{vio}}_{\text{wait}}(\iota) + c^{\text{vio}}_{\text{trip}}(\iota) \tag{3.1}$$

The cost of an insertion $\iota = (r, v, i, j)$ is the sum of the total *vehicle detour*, the *wait time violation constraint* and the *trip time violation constraint*. The total vehicle detour is defined as the detour the vehicle has to drive to pick up the passenger at the origin of the request and to drop off the passenger at the destination. It is given as $t_{\text{detour}}(\iota) = \delta(s_i, \mathit{orig}) + \delta(\mathit{orig}, s_{i+1}) + \delta(s_j, \mathit{dest}) + \delta(\mathit{dest}, s_{j+1}) - \delta(s_i, s_{i+1}) - \delta(s_j, s_{j+1})$. This formula only holds for insertions in between two consecutive stops. For insertions of a location x after the last stop of the vehicle, only the distance to x is counted. The wait time violation constraint $c^{\text{vio}}_{\text{wait}}(\iota)$ for a request r defines a penalty for an insertion exceeding the maximum wait time for the passenger at the origin. It is defined as $c^{\text{vio}}_{\text{wait}}(\iota) = \gamma_{\text{wait}} \cdot \max\{t_{\text{dep}} - t_{\text{req}} - t^{\max}_{\text{wait}}, 0\}$. The trip time violation constraint $c^{\text{vio}}_{\text{trip}}(\iota)$ for a request r defines a penalty for an insertion that breaks the maximum trip time of the request. It is defined as $c^{\text{vio}}_{\text{trip}}(\iota) = \gamma_{\text{trip}} \cdot \max\{t_{\text{arr}}(\iota) - t_{\text{req}}(r) - t^{\max}_{\text{trip}}(r), 0\}$. The model parameter γ_{wait} and γ_{trip} define the impact of the violations on the total cost of an insertion.

3.4.2 Meeting Points and Many-to-Many Routing

KaRRi introduces *meeting points* to the dynamic taxi sharing problem. Meeting points define a new set of possible pickup and dropoff points beside the origin and destination of the passenger. KaRRi then routes between those meeting points extending the classical single-origin-single-destination taxi sharing problem to a many-to-many-version. A few changes in the notation and problem statement are necessary to model the new situation.

Passenger Network

KaRRi requires a *passenger network* $G_{\text{psg}} = (V_{\text{psg}}, E_{\text{psg}})$ with shortest path distances δ_{psg} in addition to the road network from section 3.4.1. Passengers can travel along the passenger network G_{psg} without the use of vehicles by utilizing other modes of transportation (e. g. walking, cycling).

Meeting Points

In KaRRi, meeting points can formally be defined as any subset of $V \cap V_{\text{psg}}$. For a given request we define a walking radius ρ that the passenger is allowed to travel along G_{psg} to get to the *pickup* or to reach their destination from the *dropoff.* Pickups and dropoffs are the locations, where the passengers enter or leave a vehicle. Therefore the eligible set of pickups and dropoffs can be defined as $P_\rho = \{p \in V \cap V_{\text{psg}} \mid \delta_{\text{psg}}(orig, p) \leq \rho\}$ and $D_\rho = \{d \in V \cap V_{\text{psg}} \mid \delta_{\text{psg}}(d, dest) \leq \rho\}$.

Insertions

An insertion ι in KaRRi is a tuple $\iota = (r, p, d, v, i, j)$. In KaRRi, the passengers are pickup up and dropped off at meeting points that do not have to be the same as their origin and destination. Therefore, while the meaning of i and j remains the same, now we also have to consider the pickup and the dropoff $p \in P_\rho$ and $d \in D_\rho$ for an insertion ι. While the DTSP might allow forms of prior reservation, KaRRi does not. In KaRRi, the time the request was issued is the earliest possible departure time at the origin.

Cost Function

In KaRRi, assignments are evaluated using the following cost function that differs from the one presented in section 3.4.1:

$$\begin{aligned} c(\iota) = t_{\text{detour}}(\iota) + \tau \cdot (t_{\text{trip}}(\iota) + t^{+}_{\text{trip}}(\iota)) \\ + \omega \cdot t_{\text{walk}}(\iota) + c^{\text{vio}}_{\text{wait}}(\iota) + c^{\text{vio}}_{\text{trip}}(\iota) \end{aligned} \tag{3.2}$$

Note that, when model parameter ω and τ are set to zero, the cost function collapses to the cost function from section 3.4.1.

KaRRi introduces two new summands to the equation:

- $\tau \cdot (t_{\text{trip}}(\iota) + t^{+}_{\text{trip}}(\iota))$: KaRRi also includes the trip time of the passengers in the cost function. The trip times assesses service quality for the passengers by penalizing insertions that result in long time times for the passenger of r or for other passengers already assigned to v. $t_{\text{trip}}(\iota)$ is the trip time for request r with $t_{\text{trip}}(\iota) = t_{\text{arr}}(d) - t_{\text{req}}(r) + \delta_{psg}(d, dest)$. $t^{+}_{\text{trip}}(\iota)$ is called the *added trip time* for passengers already assigned to vehicle v. It is the sum of the difference in arrival times at the dropoff before and after the insertion for all affected existing passengers of the vehicle.
- $\omega \cdot t_{\text{walk}}(\iota)$: t_{walk} represent the *walking time* of an insertion that results from meeting points different from the origin or destination, as the passenger then needs to travel from the origin to the pickup and from the dropoff to the destination using the passenger network. The walking cost of an insertion is given as $t_{\text{walk}}(\iota) =$

$\delta_{\text{psg}}((, orig), p) + \delta_{\text{psg}}((, d), dest)$. The model parameter ω determines the impact of the walking cost on the total insertion cost.

3.4.3 Detour Ellipses

As described in section 3.4.1, the hard constraints of passengers are encoded in the maximum arrival time and the scheduled (minimum) departure time of a stop. The difference between those two values for consecutive stops s_i and s_{i+1} is called the *leeway* $\lambda(s_i, s_{i+1})$ for the leg starting at s_i. The leeway is the maximum detour a vehicle can drive between both stops. When leaving its route after s_i to get to vertex $x \in V$, the total detour is constraint to the leeway of the leg starting at s_i, i. e. $\delta(s_i, x) + \delta(x, s_{i+1}) \leq \lambda(s_i, s_{i+1})$. This means, that vertices that do not break this leeway constraint, form an ellipsis around stop s_i and the following stop s_{i+1}. We refer to the detour ellipsis around stop s_i and stop s_{i+1} as $E(v, i) = \{v \in V \mid \delta(s_i, v) + \delta(v, s_{i+1}) \leq \lambda(s_i, s_{i+1})\}$.

The detour ellipses are used by LOUD [5] and by KaRRi to efficiently answer many-to-one shortest path queries using *elliptic BCH queries.* In elliptic BCH searches, bucket entries are only generated for the vertices within the ellipsis.

4 Single-Transfer Journeys

In this section we introduce the conceptual changes that are necessary to extend the KaRRi model presented in section 3.4 to allow single-transfer journeys.

4.1 Single-Transfer Journey

A *single-transfer journey* is defined as a journey of a passenger, where the passenger changes vehicles exactly once somewhere along their trip. This implies that for every single-transfer journey exactly two vehicles are involved. We denote the *pickup vehicle*, the vehicle that performs the pickup, as v_p, and the *dropoff vehicle* accordingly as v_d. In KaRRi without transfers, a trip of a passenger included walking from the origin to the pickup and perhaps waiting for the vehicle v to which the request was assigned to, then using v to get to the dropoff and walking from the dropoff to the destination. Now, a trip includes walking from the origin to the pickup and perhaps waiting for v_p, then using v_p to get to the transfer point, perhaps waiting for v_d, then using v_d to get to the dropoff and finally, walking from the dropoff to the destination. In our model we allow walking only to the pickup and from the dropoff. We do not allow any form of transfer that includes walking, i. e. the transfer between vehicles takes place at a single location and both v_p and v_d will stop at this location to drop off / pick up the passenger.

4.1.1 Transfers and Single-Transfer Insertions

We define a *transfer point* as any location in the road network $tp \in V$. A *transfer* is defined as a tuple $(tp, v_\mathrm{p}, v_\mathrm{d}, i, j)$ with $tp \in V, 0 \leq i \leq k(v_\mathrm{p}), 0 \leq j \leq k(v_\mathrm{d})$. To perform the transfer, v_p picks up the passenger somewhere along the route, then drops off the passenger at tp after stop $s_i \in R(v_\mathrm{p})$. After that, the dropoff vehicle v_d picks up up the passenger after stop s_j at tp and afterwards performs the dropoff along the remaining route.

For a single-transfer journey, the dispatcher now finds an insertion for v_p and v_d simultaneously, so that the specified cost function is minimized. We formalize this insertion as a tuple $\iota = (r, v_\mathrm{p}, v_\mathrm{d}, p, d, tp, i_\mathrm{p}, j_\mathrm{p}, i_\mathrm{d}, j_\mathrm{d})$ with $p \in P_\rho, d \in D_\rho, tp \in V$. The pickup vehicle v_p picks up the passenger at $p \in P_\rho$ immediately after stop s_{i_p}. Then, immediately after stop s_{j_p}, the vehicle drops off the passenger at the transfer point $tp \in V$. The dropoff vehicle picks up the passenger at tp immediately after stop s_{i_d} and performs the dropoff immediately after stop s_{j_d}. We can decompose ι into two insertions without transfers analogous to the insertions of KaRRi, $\iota_\text{p} = (r, v_\text{p}, p, tp, i, j)$ and $\iota_\text{d} = (r, v_\text{d}, tp, d, k, l)$ for the pickup and dropoff vehicle respectively. Note that the dropoff vehicles departure at the transfer point of course has to be later than the pickup vehicles arrival there.

4.1.2 Cost Function

For single-transfer journeys we need to adjust the cost calculation presented in section 3.4.2 to represent the new wait time, vehicle cost and trip cost correctly when introducing transfers. We can keep the cost function itself, but some parts of it have to be redefined.

$$\begin{aligned} c(\iota) = t_\text{detour}(\iota) + \tau \cdot (t_\text{trip}(\iota) + t^+_\text{trip}(\iota)) \\ + \omega \cdot t_\text{walk}(\iota) + c^\text{vio}_\text{wait}(\iota) + c^\text{vio}_\text{trip}(\iota) \end{aligned} \tag{4.1}$$

- $t_\text{detour}(\iota)$: The detour for an insertion is just the sum of the detours of both vehicles: $t_\text{detour}(\iota) = t_\text{detour}(\iota_\text{p}) + t_\text{detour}(\iota_\text{d})$.
- $t_\text{trip}(\iota)$: The trip time for the passenger is now the total trip time along both vehicles: $t_\text{trip}(\iota) = t_\text{arr}(v_\text{d}, d) - t_\text{req} + \delta_\text{psg}(d, dest)$ where $t_\text{arr}(p)$ is the arrival time of the passenger at the pickup and $t_\text{arr}(v_\text{d}, d)$ is the arrival time of vehicle v_d at the dropoff.
- $t^+_\text{trip}(\iota)$: The added trip time is the sum of the added trip times for both vehicles: $t^+_\text{trip}(\iota) = t^+_\text{trip}(\iota_p) + t^+_\text{trip}(\iota_d)$. It is important to mention, that the dropoff vehicle perhaps has to wait at the transfer point until the pickup vehicle arrives at the transfer point. Therefore the minimum departure time at the transfer point is the maximum of the arrival times of both vehicles at the transfer point.
- $t_\text{walk}(\iota)$: As we do not allow walking at the transfer point in our model, the walking time remains unchanged.
- $c^\text{vio}_\text{wait}(\iota)$: The wait time violation cost now also will account for the waiting of the passenger at the transfer point: $c^\text{vio}_\text{wait}(\iota) = \max\{0, t_\text{dep}(v_\text{p}, p) - t_\text{arr}(p) + t_\text{dep}(v_\text{d}, tp) - t_\text{arr}(v_\text{p}, tp)\}$.
- $c^\text{vio}_\text{trip}(\iota)$: The trip time violation cost stays the same given the new trip time.

4.1.3 Constraints

Single-transfer journeys are also subject to all the different service constraints from section 3.4.1. A single-transfer insertion ι with no-transfer insertions $\iota_\text{p} = (r, v_\text{p}, p, tp, i, j)$ and $\iota_\text{d} = (r, v_\text{d}, tp, d, k, l)$ for pickup and dropoff vehicle respectively does not violate any of the constraints if ι_p and ι_d do not break the constraints for their vehicles v_p and v_d. This implies, that if a transfer is inserted between two consecutive stops s_i and s_{i+1} of vehicle v, the distance between stop s_i and the transfer point and between the transfer point and the next scheduled stop s_{i+1} is not allowed to exceed the leeway of s_i, i. e. $\lambda(s_i, s_{i+1})$. Note that, when the transfer is ALS, the leeway is not defined as there is no stop after the last stop and no hard constraints of passengers can be violated.

In the single-transfer case, dependencies between vehicles stops arise when a passenger transfers between them. For example, a passenger transfers between vehicle v_p at stop s_{i_p} and vehicle v_d at stop s_{i_d} with transfer location *tp*. The initial scheduled arrival time at *tp* is $t^{\mathrm{init}}_{\mathrm{arr},tp}$ and the initial scheduled departure time is $t^{\mathrm{init}}_{\mathrm{dep},tp}$. After processing some more requests, another stop is inserted in the pickup vehicles route before it reaches s_{i_p}, causing the vehicle to reach the transfer point at a later point in time. Now it is important, that the dropoff vehicle is informed about this change, as it potentially needs to wait for the passenger at the transfer point. Therefore we need to propagate the delayed arrival at the transfer point along stops of the dropoff vehicle after stop s_{i_d}. A scenario for the maximum arrival time can be imagined accordingly. In general, for a given transfer $(tp, v_\mathrm{p}, v_\mathrm{d}, s_{j_\mathrm{p}}, s_{i_\mathrm{d}})$, we now have to potentially propagate changes in scheduled departure times of stops of v_p before the transfer along the route of v_d from the transfer on. Accordingly, we have to propagate changes in the maximum arrival time after the transfer in v_d backwards along the route of v_p to the stops before the transfer.

5 Finding Single-Transfer Journeys

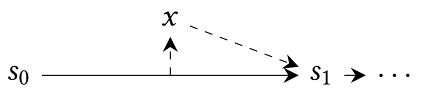
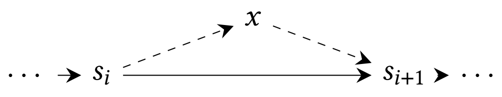
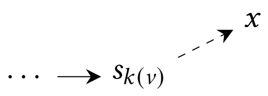
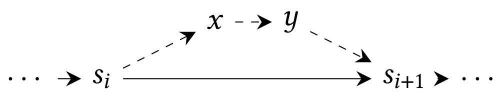
In this chapter, we outline the framework necessary for finding single-transfer journeys in the dynamic taxi sharing problem with meeting points. We give a high level outline for an algorithm that computes single-transfer journeys and illustrate the structure of the insertions that can be found.

5.1 Insertion Types

Given the route of a vehicle v and indices i, j, we consider the following four types of insertions of $x, y \in V$, analogous to the terminology introduced in KaRRi [14]. The vertices x and y represent locations where a passenger will be picked up or dropped off by vehicle v. The insertion of x happens between stops s_i and s_{i+1}, the insertion of y between stops s_j and s_{j+1}. We illustrate the different insertions in table 5.1.

- **Before Next Stop**: For an insertion of x *before the next stop*, the vehicle immediately, after receiving the request, is rerouted from its current location and drives to x. From x it then returns to the previously scheduled route and goes on to the next stop s_{i+1}.
- **Ordinary**: For an *ordinary* insertion of x into $R(v)$ after stop s_i, the vehicle leaves the scheduled route after arriving at s_i, then drives to x and thereafter drives to the next scheduled stop s_{i+1}.
- **After Last Stop**: For an insertion of x *after last stop*, the vehicle extends its route beyond the last scheduled stop $s_{k(v)}$ to satisfy a given request. After the last stop $s_{k(v)}$, instead of entering the idle state, the vehicle drives to x where the passenger is picked up / dropped off. If the passenger is dropped off at x, the vehicle then idles at the stop. If the passenger is picked up at x, the vehicle continues on directly to the passengers dropoff location.
- **Paired**: For a *paired* insertion of x and y into route $R(v)$, both x and y are inserted into the same leg, i. e. between two subsequent stops.

Table 5.1: Different types of insertions of vertices x, y into the route of vehicle v. We display insertions BNS, ORD and ALS only for pickup vertex x but the insertion would be the same for a dropoff y.

Before Next Stop (BNS)	$0 = i \leq j < k(v)$	
Ordinary (ORD)	$0 < i \leq j < k(v)$	
After Last Stop (ALS)	$0 \leq i \leq j = k(v)$	
Paired	$0 \leq i = j \leq k(v)$	

5.2 Algorithm Outline

In this section we want to give a high level overview of an algorithm that computes optimal single-transfer journeys. After an initialization phase to find meeting points and feasible pickup and dropoff locations among the set of meeting points for every vehicle, the algorithm finds assignments with a transfer. The algorithm differentiates between three types of transfer insertions: Ordinary transfers, where the insertion of the transfer point is ordinary or before next stop for both vehicles, transfers after last stop of the pickup vehicle, and transfers after last stop of the dropoff vehicle.

5.2.1 Initialization Phase

After receiving a request $r = (orig, dest, t_{\text{req}})$ the algorithm first has to compute the set of meetings points for the given origin and destination of the request. The algorithm then finds a subset of the vehicles that could potentially pick up the passenger and a subset of the vehicles that could potentially drop off the passenger.

5.2.2 Assignments with Ordinary Transfer

An *ordinary transfer* $(tp, v_{\text{p}}, v_{\text{d}}, i, j)$ with $0 \leq i < k(v_{\text{p}}), 0 \leq j < k(v_{\text{d}})$ is a transfer where the transfer point *tp* is inserted before next stop or ordinarily into both vehicles routes. The detour for the legs after stop s_i and s_j is bound by hard constraints of the other passengers that are assigned to the vehicles. When inserting a transfer point, this detour can not be exceeded. For an ordinary transfer, the pickup can be BNS or ORD. The dropoff can be BNS paired if the transfer is BNS in the dropoff vehicle v_{d}, ORD or ALS. For a given

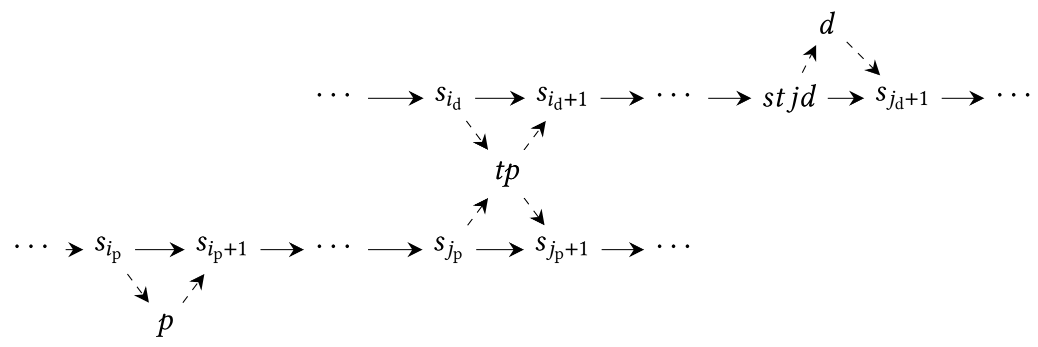


Figure 5.1: Illustrative Routes for an ordinary transfer between vehicles v_p, v_d at tp.

request r, the algorithm constructs sets of potential pickup and dropoff vehicles from the set of vehicles with relevant pickups and dropoffs. Given the set of potential vehicles, the algorithm computes the set of feasible transfer points that do not break any hard constraint. Then the algorithm enumerates all potential assignments with ordinary transfer by combining a relevant pickup (p, i_p) for v_p after stop s_{i_p}, a potential transfer tp after stop $s_{j_p}(v_p)$ and stop $s_{i_d}(v_d)$ and a relevant dropoff (d, j_d) for v_d after stop s_{j_d} into one insertion $\iota = (r, v_p, v_d, p, d, tp, i_p, j_p, i_d, j_d)$. These insertions are then evaluated under the given cost function. The structure of a ordinary transfer is illustrated in fig. 5.1.

5.2.3 Assignments with Transfer ALS

Transfers ALS for the pickup or dropoff vehicle are algorithmically challenging as the leeway of the vehicle performing the transfer als is not limited by hard constraints. At its last stop, the vehicle is either empty or only the passenger issuing the request is currently inside. Therefore the vehicle can drive to arbitrary vertices in the road network without breaking hard constraints. It is worth mentioning that the transfer cannot be ALS for both vehicles. That is because vehicle operation cost is the same among vehicles. An insertion with ALS transfer always has higher cost than an insertion without transfer where the pickup vehicle performs the dropoff as well. The cost to get from the transfer point to the dropoff for the pickup vehicle is the same as it would be for the dropoff vehicle. Therefore for optimal solutions in our model, it is sufficient to consider transfers als for only one vehicle.

Transfer ALS of Pickup Vehicle

Transfer ALS of Dropoff Vehicle

For assignments with transfer als for the pickup vehicle, pickup and dropoff can be bns, ord or als. For a given dropoff vehicle, the potential transfer points are vertices in the detour ellipses for the stops of the vehicle. For every potential pickup vehicle, the algorithm needs to compute the distance from the last stop of the vehicle to points of the detour ellipses of the potential dropoff vehicle. The algorithm enumerates possible assignments by combining

$$\begin{tikzcd}
& & & & & & & & & & & d \arrow[dr, dashed] & & \\
& & & & & \cdots \arrow[r] & s_{i_d} \arrow[rr] \arrow[dr, dashed] & & s_{i_d+1} \arrow[r] & \cdots \arrow[r] & s_{j_d} \arrow[rr] \arrow[ur, dashed] & & s_{j_d+1} \arrow[r] & \cdots \\
& & & & & & & tp \arrow[ur, dashed] & & & & & & \\
\cdots \arrow[r] & s_{i_p} \arrow[rr] \arrow[dr, dashed] & & s_{i+1} \arrow[r] & \cdots \arrow[rr] & & s_{k(v_p)} \arrow[ur, dashed] & & & & & & & \\
& & p \arrow[ur, dashed] & & & & & & & & & & &
\end{tikzcd}$$

Figure 5.2: Illustrative routes with an ALS transfer for v_p at tp.

$$\begin{tikzcd}
& & & & \cdots \arrow[r] & s_{k(v_d)} \arrow[dr, dashed] & & d & \\
& & & & & & tp \arrow[ur, dashed] \arrow[dr, dashed] & & \\
\cdots \arrow[r] & s_{i_p} \arrow[rr] \arrow[dr, dashed] & & s_{i_p+1} \arrow[r] & \cdots \arrow[r] & s_{j_p} \arrow[rr] \arrow[ur, dashed] & & s_{j_p+1} \arrow[r] & \cdots \\
& & p \arrow[ur, dashed] & & & & & &
\end{tikzcd}$$

Figure 5.3: Illustrative routes with an ALS transfer for v_d at tp.

relevant pickups, transfer points and dropoffs to an insertion $(r, v_p, v_d, p, d, tp, i_p, k(v_p), i_d, j_d)$. It then assesses the cost of the assignment to check if the currently known best solution can be improved.

For assignments with ALS transfer for the dropoff vehicle, the pickup has to be BNS or ORD. The dropoff naturally has to be ALS as well as it must be after the transfer that is ALS. Again, potential transfer points can not break detours given by other passengers hard constraints.

As the distance from the last stop of the dropoff vehicle to a dropoff is a lower bound for the distance from the last stop to the dropoff via a potential transfer point, the algorithm uses vehicles with relevant dropoffs als as the set of potential vehicles that can perform a transfer als. The algorithm then enumerates the feasible pickup vehicles. To get all potential transfer points, the algorithm needs to reconstruct the detour ellipsis for every stop of the potential pickup vehicles and calculate the distances from the last stops of the dropoff vehicles to the potential transfer points as well as the distance from the transfer point to the potential dropoffs. To find the optimal solution with transfer als for the dropoff vehicle, the algorithm enumerates all assignments by combining a potential pickup, transfer and a potential dropoff in an insertion $(r, v_p, v_d, p, d, tp, i_p, j_p, k(v_d), k(v_d))$ and evaluates the resulting assignments.

6 Computing Single-Transfer Journeys

In this section we want to explain the details of an algorithm that computes single-transfer journeys and how we choose to implement such an algorithm.

6.1 Finding Meeting Points

After receiving a request $r = (\mathit{orig}, \mathit{dest}, t_{\text{req}})$, the algorithm first determines all potential meeting points in the walking radius ρ around the origin and the destination. These pickups P_ρ and dropoffs D_ρ are computed using the implementation provided by KaRRi using a forward Dijkstra searches rooted at *orig*, and a reverse Dijkstra search from *dest*, both with the maximum search radius of ρ. This means, that the search is stopped as soon as the minimum distance label of the queue exceeds ρ.

6.2 Finding Relevant Pickups

In this section we describe how the algorithm finds vehicles that can perform a pickup for a given request and efficiently decides which of the pickups are feasible for one of those pickup vehicles. We also describe the connection to the dropoff at the transfer stop in the pickup vehicle, as no pickup is possible without dropping off the passenger later. For more details on the exact implementation and pruning techniques utilized to find relevant pickups and dropoffs, we refer to the research of Laupichler and Sanders in KaRRi [14] as we only outline the foundations of the algorithms used.

6.2.1 Ordinary Pickups

Ordinary pickups $p \in P_\rho$ for a given stop s_{i_p} of vehicle v_p have to be part of the ellipsis $E(s_{i_\text{p}}, s_{i_\text{p}+1})$ around stop s_{i_p} and stop $s_{i_\text{p}+1}$ that is defined by the maximum leeway $\lambda(s_{i_\text{p}}, s_{i_\text{p}+1})$ of the given leg. To filter out the pickups that are too far away for the given stop, we need to compute the distances from and to the pickup for the stop. A potential pickup vehicle v_p for an pickup ORD has at least one pair of consecutive stops $(s_{i_\text{p}}, s_{i_\text{p}+1})$, s. t. there is a pickup $p \in P_\rho$ with $\delta(s_{i_\text{p}}, p) + \delta(p, s_{i_\text{p}+1}) \leq \lambda(s_{i_\text{p}}, s_{i_\text{p}+1})$.

Problem Statement: For every stop s_i with $0 < i_p < k(v_p)$ of every vehicle v_p, compute the distances from the stop s_{i_p} to every pickup $p \in P_\rho$, as well as the distance from p to the next stop of the vehicle s_{i_p+1}.

Solution: To compute all ordinary pickups, our algorithm utilizes the methods provided by KaRRi. KaRRi uses bundled elliptic BCH searches running from every pickup with bucket entries for every stop of every vehicle to compute the distances from and to every pickup from and to every stop.

6.2.2 Pickups Before Next Stop

For pickups before next stop the distances computed by the elliptic BCH search from section 6.2.1 for vehicle v_p potentially can not be used anymore as soon as the vehicle has left the previous stop s_0 because the vehicle is now somewhere along its route between s_0 and s_1. Therefore, the vehicle first has to be located along the leg of s_0. Then the distances from this location $loc(v_p)$ to the pickups have to be computed.

Problem Statement: For every vehicle v_p, locate the vehicle and compute the distance between the current location of the vehicle and every pickup.

Solution: To calculate the pickups before next stop we also employ the functionality provided by KaRRi. First we note, that the distance from s_0 to a pickup p is a lower bound for the distance from s_0 to a pickup p via the current location of the vehicle $loc(v_p)$. Therefore we can use this lower bound in combination with the previously calculated distance from the pickup p to s_1 to filter out unfeasible pickups. The vehicle can be located using the departure time at s_0 and the request time t_{req} as the length and path of each leg is known, as well as the speed of the vehicle along each edge. Then KaRRi employs bundled CH searches to compute the distance from the current location of the vehicle to pickups that are still relevant.

6.2.3 Pickups After Last Stop

As with any ALS insertion, the detour which the vehicle v_p drives from its last stop $s_{k(v_p)}$ is not bound by any hard constraint of passengers already assigned to the vehicle. Note that in the pickup ALS case, the dropoff at the transfer point has to be ALS as well, therefore every pickup ALS assignment also includes a paired insertion of the transfer.

Problem Statement: Calculate the distances from the last stops of all vehicles to every pickup.

Solution: For the pickup ALS insertions, we use BCH searches, exactly like the searches used by KaRRi, to compute the distances to the pickup efficiently. We maintain bucket entries for every last stop and perform individual BCH queries rooted at every pickup to compute the shortest path distances from every last stop to every pickup.

6.3 Finding Relevant Dropoffs

In this section we describe the different types of dropoffs and how to compute them efficiently. We also describe the connection to the pickup at the transfer stop in the dropoff vehicle, as both insertions form the assignment for the vehicle.

6.3.1 Ordinary Dropoffs, Dropoffs Before Next Stop

The computation of ordinary dropoffs and dropoffs before next stop is analogous to the computation of the relevant ordinary pickups. The same filtering is applied to the dropoffs D_ρ and distances are calculated in the same manner. We explicitly point out, that a dropoff BNS presumes a transfer BNS, therefore the distance to the dropoff is calculated using the techniques for paired assignments.

6.3.2 Dropoffs After Last Stop

Similarly to the pickup ALS insertion, the detour the vehicle v_d drives from its last stop $s_{k(v_\mathrm{d})}$ is not bound by any hard constraint of passengers already assigned to the vehicle. Note that unlike the pickup ALS case, the dropoff ALS case does not require a paired insertion which affects some pruning techniques.

Problem Statement: Calculate the distances from the last stops of all vehicles to every dropoff.

Solution: To find dropoff ALS insertions, we again use BCH searches. We maintain bucket entries for every last stop and perform individual BCH queries rooted at every dropoff to compute the shortest path distances from every last stop to every dropoff.

6.4 Compute Paired Distances

For a paired insertion of x and y, we need to calculate the distance from x to y explicitly as the vehicle drives from x directly to y and only then to the next scheduled stop. Computing paired distances for single-transfer journeys introduces new challenges. In the no-transfer situation, paired assignments always have pickup-dropoff paired insertions. Therefore all paired distances can be computed by one many-to-many query between the P_ρ and the D_ρ. In the single-transfer case, paired insertions are either pickup-transfer-point or transfer-point-dropoff insertions. As the set of transfer points that the algorithm calculates along the algorithm is dependent on the selected relevant vehicles, we can not compute all direct p-tp-distances and tp-d-distances.

Problem Statement: Compute the direct distance between a pickup p and a transfer point tp or the direct distance between a transfer point tp and a dropoff d.

Solution: To compute paired distances between pickups and transfer points (PT) and between transfer points and dropoffs (TD), we employ individual CH queries. To reduce the number of CH queries we have to execute, we first compute a lower bound for the PT and the TD paired distances respectively. For this we run an any-to-any CH query, which means that we initialize every source with distance 0 in the forward search and every target with distance 0 in the reverse search. This gives us the shortest path between any source and any target. We use these lower bounds to only calculate the exact paired distance for insertions that are promising using the lower bound.

6.5 Finding Possible Transfer Points

In this section we describe the subproblems the algorithm from section 5.2 has to solve in order to calculate the relevant transfer points for a given request.

6.5.1 Ordinary Transfers

The distances to and from an ordinary transfer tp between consecutive stops $(s_{j_\mathrm{p}}, s_{j_\mathrm{p}+1})$ of the v_p and between consecutive stops $(s_{i_\mathrm{d}}, s_{i_\mathrm{d}+1})$ of the v_d are constraint by the leeways of both legs. As all the vertices that could be inserted for leg s_i without breaking the leeways constraint form an ellipsis around stop s_i and s_{i+1}, finding the optimal solution for an ordinary transfer results in the following problem statement.

Problem Statement: For each potential pairs of vehicles involved, $(v_\mathrm{p}, v_\mathrm{d})$, calculate the intersection of the detour ellipses $E(v_\mathrm{p}, j_\mathrm{p}) \cap E(v_\mathrm{d}, i_\mathrm{d})$ for every stop pair $(s_{j_\mathrm{p}}(v_\mathrm{p}), s_{i_\mathrm{d}}(v_\mathrm{d}))$.

Solution: To find the optimal solution with ordinary transfer we implemented two different strategies. First, we implemented a naive Dijkstra strategy that runs four Dijkstra searches to determine the intersection of the ellipses. The algorithm performs a forward Dijkstra search from $s_{j_\mathrm{p}}(v_\mathrm{p})$ and a reverse Dijkstra search from $s_{j_\mathrm{p}+1}(v_\mathrm{p})$ with maximum radius $\lambda(s_{j_\mathrm{p}}, s_{j_\mathrm{p}+1})$ and a forward Dijkstra search from $s_{i_\mathrm{d}}(v_\mathrm{d})$ and a reverse Dijkstra search from $s_{i_\mathrm{d}+1}(v_\mathrm{d})$ with maximum radius $\lambda(s_{i_\mathrm{d}}, s_{i_\mathrm{d}+1})$. For the first search we store the search space completely. For all other searches we mark the vertices in the first search space if they are found int the other search as well. After the searches we construct the set of feasible transfer points $E(v_\mathrm{p}, j_\mathrm{p}) \cap E(v_\mathrm{d}, i_\mathrm{d})$ by adding only vertices that were marked by all searches and where the sum of the distance to and from the transfer point does not exceed the leeway, i. e. $\delta(s_i, tp) + \delta(tp, s_{i+1}) \leq \lambda(s_i, s_{i+1})$ for $i \in \{j_\mathrm{p}, i_\mathrm{d}\}$. Note that the Dijkstra searches guarantee to compute the required shortest path distances for every vertex in the ellipse, i. e. the distance from $s_{j_\mathrm{p}}(v_\mathrm{p})$ and to $s_{j_\mathrm{p}+1}(v_\mathrm{p})$ and from $s_{i_\mathrm{d}}(v_\mathrm{d})$ and to $s_{i_\mathrm{d}+1}(v_\mathrm{d})$.

The second, more efficient, strategy uses the elliptic reconstruction functionality that is implemented for the KaRRi repository[1]. The ellipse reconstruction, as the name suggests, reconstructs the detour ellipsis of a given stop for a vehicle. We can find all vertices lying

[1]https://github.com/molaupi/karri/tree/ellipse$_reconstruction/$

within the detour ellipse of consecutive stops s_i and s_{i+1} by running a topological downward search in the CH that computes for each vertex v the distances from s_i to v and from v to s_{i+1}. The search makes use of the existing source bucket entries for s_i which are constructed s. t. for each vertex v that lies within the ellipse, the shortest path from s_i to v is an up-down path whose highest-ranked vertex has a bucket entry in the source bucket of s_i. Analogously, the shortest path from v to s_{i+1} is an up-down path whose highest-ranked vertex has a bucket entry in the target bucket of s_{i+1}. Thus, the topological downward search can be initialized using these bucket entries and is then guaranteed to find the required distances for all vertices that lie within the ellipse and correctly identify them.

For a transfer BNS we need to execute additional computation. If the transfer is BNS in the pickup vehicle, we need to compute the paired distance from the pickup, that is also BNS, to the transfer point. If the transfer is BNS in the transfer vehicle, we perhaps have to locate the dropoff vehicle and then calculate the BNS distance from the last stop to the transfer via the current location of the dropoff vehicle.

6.5.2 Transfer ALS

To find optimal solutions for a transfer after last stop between given vehicles v_a, v_b, where the transfer is ALS for vehicle v_a we would need to reconstruct the detour ellipses of the stops of vehicle v_b and then calculate the distances from the last stop of vehicle v_a to every point in the ellipses. This would result in an immense amount of possible transfer points and distance calculations. Therefore we chose, for now, to give up optimality for the solutions with the transfer ALS for one of both vehicles. We restricted our algorithm to compute solutions where only v_a, the vehicle with transfer ALS, is allowed to drive the detour to the transfer point. This implies, that transfer points must be along the current route of the vehicle not driving the detour. We further reduce complexity and the number of transfer points by only allowing transfers at existing stops of v_b. In fig. 6.1 we sketch possible routes for a transfer at an existing stop for v_b with transfer ALS for v_a.

$$\begin{tikzcd}
\cdots \arrow[r] & s_{k(v_\text{a})} \arrow[d, dashed] & & \\
\cdots \arrow[r] & s_i(v_\text{b}) = tp \arrow[r] & s_{i+1}(v_\text{b}) \arrow[r] & \cdots
\end{tikzcd}$$

Figure 6.1: Transfer point at existing stop for transfer ALS.

For our algorithm, finding a transfer ALS for a vehicle results in the following problem statement.

Problem Statement: For a given pair of pickup and dropoff vehicle (v_p, v_d), calculate the distances from the last stop of the vehicle that performs the ALS dropoff to all stops of the other vehicle. If the transfer is ALS in the pickup vehicle, the distances from the ALS pickups to all stops of the other vehicles also have to be computed. If the transfer is ALS in the dropoff vehicle, the distance from the transfer point to all dropoffs have to be computed as well.

Solution: We use bundled CH searches to compute the point-to-point distances between the necessary vertices to calculate all necessary distance for the transfer ALS.

6.6 Additional Remarks

When dealing with our single-transfer journeys, just as with any other dynamic taxi sharing dispatching, we have to keep the routes updated according to the results that the dispatcher produces. In our case this includes the scheduled departure times, scheduled arrival times and maximum arrival times for all vehicles and stops. In the single-transfer case we face the challenge of, that now a request with its hard constraints is satisfied by two different vehicles. For every transfer between two vehicles, the algorithm needs to keep track of this dependency and needs to propagate changes in minimum departure times of the pickup vehicle and maximum departure times of the dropoff vehicle also along the route of the other vehicle to ensure that the pickup vehicle arrives at the transfer point on time and to ensure, that the dropoff vehicle does not leave the transfer point to early. To ease implementation we chose to force the pickup vehicles to arrive at the transfer point at the scheduled time when the request was assigned by not allowing any further detours before the transfer. The dropoff vehicle keeps the full flexibility for changing its route, while the pickup vehicle is not allowed to delay its arrival at the transfer point.

7 Evaluation

In this section we evaluate the results of our work. To test the effect of transfer routes on the DTSP, we conducted several experiments on realistic input data.

For our implementation of the algorithm from chapter 5 and chapter 6 we extend the source code of KaRRi[1]. The source code for KaRRi with transfers[2] is written in C++17 and compiled with the GCC 11.4 GNU C++ Compiler using –O3. The experiments are run on a compute server with the Rocky Linux 9.4 operating system, 64 GB of memory and an Intel® Xenon® CPU E5-2650 at a clock speed of 2.60GHz.

7.1 Overview Benchmark Instances

First, we want to give a brief overview of the benchmark instances used for our experiments. We use the same underlying road network that KaRRi uses. The road network is extracted from the OpenStreetMap data for Berlin.[3] To determine the travel time for a given edge of the graph, the length and the known speed limit of the represented road segment is used. The Berlin road network consists of 94422 vertices and 193212 edges ($|V| = 94422$, $|E| = 193212$). We also use the same request and vehicle data that KaRRi used to represent taxi sharing demand and the vehicle fleet. The request and vehicle data is drawn from the artificially generated `Berlin-1pct` set of the Open Berlin scenario [21] for the MATSim transport simulation [1]. This request set represents 1% of taxi sharing demand in the Berlin metropolitan area on a weekday in a given time frame of 3 hours. Road capacities are adjusted accordingly.

As the running time of our algorithm massively increases in comparison to the dispatcher without transfer because of the expensive transfer point calculation and the sheer amount of assignments, that have to be tried, we reduce the number of requests and vehicles to run our experiments. For the instance `B1%-400-all` we only reduce the number of vehicles from 1000 down to 400. For instance `B1%-400-hd` we also remove every second request to halve the density and the total amount of the requests. We also extract request data of the second hour (`h2`) for `B1%-400-h2` and the third hour (`h3`) for `B1%-200-h3` where we also reduce the number of vehicles down to 200. We update the minimum and maximum service times of the vehicles accordingly. An overview of the request and vehicle sets used by our

[1]https://github.com/molaupi/karri/

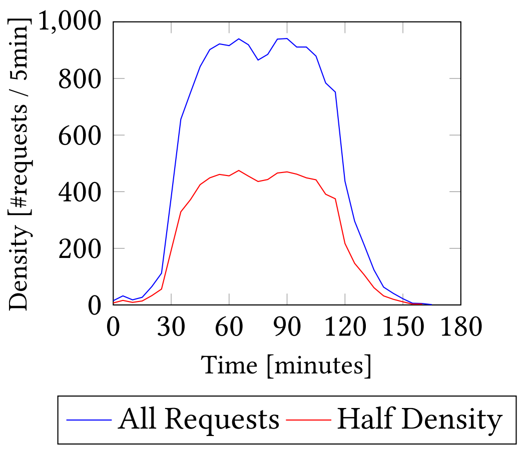
[2]https://github.com/JohannesBreitling/karri-with-transfers

[3]https://download.geofabrik.de/europe/germany.html, accessed Mar 03th 2025.

experiments is given in table 7.1. The distribution of the requests over time is shown in fig. 7.1.

Table 7.1: Overview of requests and vehicles in the `Berlin-1pct` experimental instances.

Instance	#requests	#vehicles	time frame	first request	last request
`B1%`	16569	1000	3h	0:00	2:45
`B1%-400-all`	16569	400	3h	0:00	2:45
`B1%-400-hd`	8284	400	3h	0:00	2:43
`B1%-400-h2`	10652	400	1h	1:00	1:59
`B1%-200-h3`	1216	200	1h	2:00	2:45



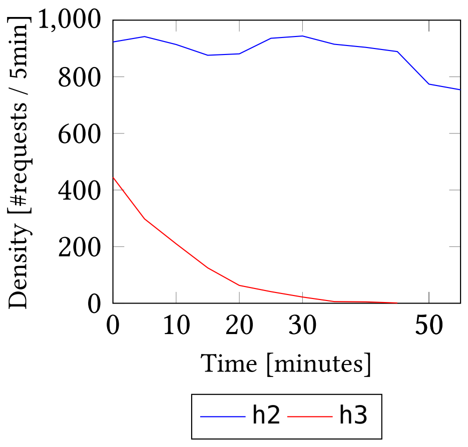


Figure 7.1: Request density for the different instances.

7.2 Comparison with the KaRRi Dispatcher

In this section we want to compare the baseline KaRRi dispatcher (K) to our algorithm (KT). We investigate the effect of the introduction of STJs on dispatch quality and running time. Although our algorithm technically can handle meetings points and finds assignments using those, we were not able to conduct meaningful experiments, as monitoring of execution suggested estimated running times for instances like the `B1%-400-h2` to exceed multiple days. For our experiments we use the default parametrization of the cost function ($\alpha = 1.7$, $\beta = 2\text{min}$, $t_{\text{wait}}^{\text{max}} = 5\text{min}$, $t_{\text{stop}}^{\text{min}} = 60\text{sec}$, $\tau = 1$). We also tried a different parametrization of the cost function, where we set the influence of passengers trip cost to zero, but we were not able to observe further decrease in vehicle operation time, as the trip times of passengers seem to affect the total cost only marginally. We leave a detailed investigation of the influence that different cost functions have on the dispatch quality open for future research.

Table 7.2: Dispatch quality of K and KT. We investigate average trip and wait times of passengers (trip, wait) and average operation times (op), drive times (drive), occupancies (occ) and number of stops (#stops) of the vehicles and average cost of the best assignment found (cost).

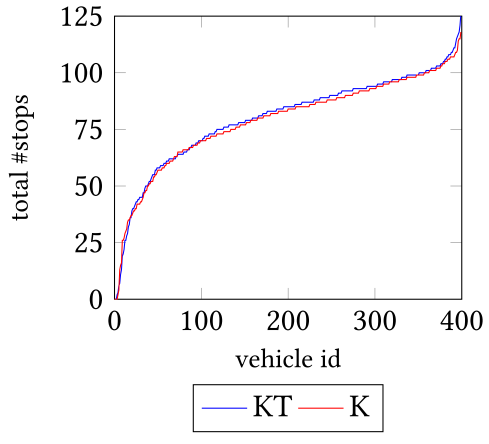
Instance	Dispatcher	wait	trip	op	drive	occ	#stops	cost
`B1%-400-hd`	K	05:22	18:54	00:37:06	00:33:14	0.765	39	24772
	KT	05:35	18:42	00:36:06	00:32:11	0.761	39	24409
`B1%-400-all`	K	07:19	21:43	01:14:21	01:06:48	0.813	78	32819
	KT	07:47	21:30	01:12:05	01:04:13	0.808	79	31470

7.2.1 Dispatch Quality

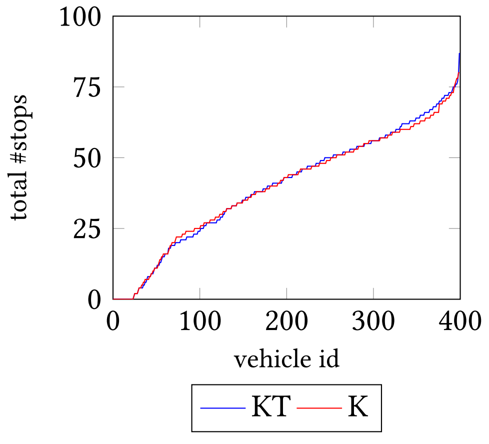
In the following we explore the quality of the assignments found by the baseline dispatcher and the single-transfer assignments found by our algorithm. We try to gain insight in how vehicle occupancy and operation time could be improved and how wait and trip time of the passengers are affected when allowing single-transfer trips. Table 7.2 displays the dispatch quality of our algorithm in comparison to KaRRi.

While we expected single-transfer journeys to have a bigger effect, we still notice a decrease of total operation time of up to 5.7% in the `B1%-400-all` instance and around about 2.7% in the `B1%-400-hd` instance. The vehicles drive time is experiencing a slightly steeper decline in comparison to the overall operation time. The total occupancy unexpectedly showed a slight decrease of around 2%. The underlying causes for that remain open for future investigation. We also notice a decrease in total cost of up to 4% in the `B1%-400-all` instance. In both instances, the wait time of passengers increased less than a minute while total trip slightly decreases. The total number of stops of the vehicles also seem to be unaffected by the single-transfer routes. Figure 7.2 shows the distribution of total number of stops among all vehicles. We notice that our algorithm was able to distribute operation slightly more evenly, producing a slightly smoother curve of stops among the vehicles. We further observe that with increasing request density, both algorithms were able to distribute vehicle utilization better among the vehicles. We argue that the decrease in vehicle operation time and trip times for passengers result from a better utilization of the vehicles when allowing single-transfer routes. The algorithm has more flexibility to choose vehicles that are feasible for a given request, even if they only are good for a small portion of the total trip, as another vehicle then can finish the trip. We speculate that because of the single-transfer journeys picking up a passenger ALS at the pickup or the transfer point is more common than in the no-transfer case. This would explain the reduced average occupancy, because for a pickup ALS, the vehicle first drives from last stop to the pickup or transfer location with an occupancy of 0.

KaRRi allows passengers to directly walk to their destination if the walking cost is smaller than the total cost for the trip using a vehicle. During the evaluation process, we noticed, that our algorithm significantly reduces the number of assignments that are not using a vehicle as shown in table 7.3. When not allowing transfers, a request might be satisfied by



(a) `B1%-400-all` Instance



(b) `B1%-400-hd` Instance

Figure 7.2: Distribution of Stops Among All Vehicles, Sorted by Total Number of Stops

Table 7.3: Percentage of No-Vehicle Solutions Produced by K and KT

Instance	% no veh K	% no veh KT	Δ
`B1%-400-all`	6.4%	2.6%	-59%

walking, if the total detour to pick up and drop off the passenger is to big for every vehicle as it would break hard constraints. When allowing single-transfer trips the algorithm is able to assign a vehicle to only perform the pickup and one to assign the dropoff and meet at a transfer point, which is much more likely to produce a feasible assignment than in the no-transfer case. This explains the drastic decrease of no-vehicle trips when allowing single-transfer journeys.

We performed another experiment, where we disabled the option of no-vehicle solutions for KaRRi and our algorithm. As the results displayed in table 7.4 suggest, when not allowing walking-only assignments, the overall dispatch quality is slightly improved on the small `B1%-200-h3`. While the result only shows a small increase in dispatch quality, we argue that with higher request densities and resulting higher vehicle utilization, the effect should increase more noticeably.

Table 7.4: Differences in dispatch quality stats between KT compared to K, evaluated on the `B1%-200-h3` instance.

Instance	Dispatcher	wait	trip	op
`no-veh allowed`	K	09:15	26:14	00:13:03
`no-veh allowed`	KT	09:40	26:11	00:12:35
`no-veh disallowed`	K	11:29	27:27	00:13:50
`no-veh disallowed`	KT	11:21	26:48	00:13:17

Table 7.5: Average running times of K and KT per request (in ms).

Instance	$t_{\text{total,K}}$	$t_{\text{total,KT}}$	t_{ord}	$t_{\text{als,p}}$	$t_{\text{als,d}}$
`B1%-400-hd`	0.278	2027.18	1975.94	39.44	11.40
`B1%-400-all`	0.293	2261.39	2144.45	94.18	22.30

7.2.2 Running Time

To evaluate the running time of both algorithms we measured running times for our algorithm and for the KaRRi dispatcher using the same code base but disabling the calculation for STJs. In table 7.5 we display the average running times per request in milliseconds for KaRRi ($t_{\text{total,K}}$) and our algorithm ($t_{\text{total,KT}}$) as well as the running times for the different types of transfers (t_{ord}, $t_{\text{als,p}}$, $t_{\text{als,d}}$). We find that the running time of our algorithm exceeds the running time of KaRRi by a factor of up to 7800. The ordinary transfer point calculation has the main effect on the running time of KT. We will further investigate the reasons behind that in section 7.3.3.

7.3 Algorithm Analysis

In this section we want to analyze the dispatching results that our algorithm produces. We break down the structure of the found assignments with transfer, analyze the performance of the different steps our algorithm executes and compare the running times of the Dijkstra strategy for finding ORD transfers with the ellipses reconstruction strategy based on the CH topological search.

7.3.1 Overall Quality

We first want to analyze the requests where our algorithm finds an improving assignment with transfer in comparison to the corresponding assignment without transfer. In table 7.6 we show the total percentage of requests that could be improved using a transfer (% hit) and the percentage of requests that already used a vehicle and could be improved (% hit use veh). We also show the differences in cost, vehicle detour and trip time for the requests that were improved. Our algorithm was able to find up to 16.8% of requests that could be improved while also decreasing cost, trip time and total detour. We observe, that due to higher request density, the portion of improved requests doubled between the `B1%-400-all` and `B1%-400-hd` instances. We argue, that with even higher densities, the impact of single-transfer routes could be further increased significantly. Transfers play a more important role for higher request densities as they provide more flexible scheduling options. With higher densities, vehicle utilization naturally increases, therefore more passengers and their hard constraints are subject of consideration for each vehicle. Transfers allow for insertions, where trips with no transfer are more likely to break hard constraints and thus, are not feasible.

In fig. 7.3 we display the distribution of cost reduction over all improved requests. We plotted the percentage of cost improvement on the x- and the percentage of improved requests with cost improvement higher than that on the y-axis. The data reveals that, as expected, the instance with the higher density, `B1%-400-all`, shows an overall more significant cost improvement. In the `B1%-400-hd` instance, 20% of the requests were improved by at least 30% while in the `B1%-400-all` instance, 20% of the requests where improved by at least 35%. Even higher densities would probably result in even higher improvements.

Table 7.6: Quality of the found improving assignments compared to the dispatch solution with no transfer.

Instance	% hit	% hit use veh	Δ_{cost}	Δ_{detour}	Δ_{trip}
`B1%-400-all`	16.80%	12.91%	-9462	-00:31	-01:44
`B1%-400-hd`	7.64%	6.24%	-6457	-00:40	-01:00

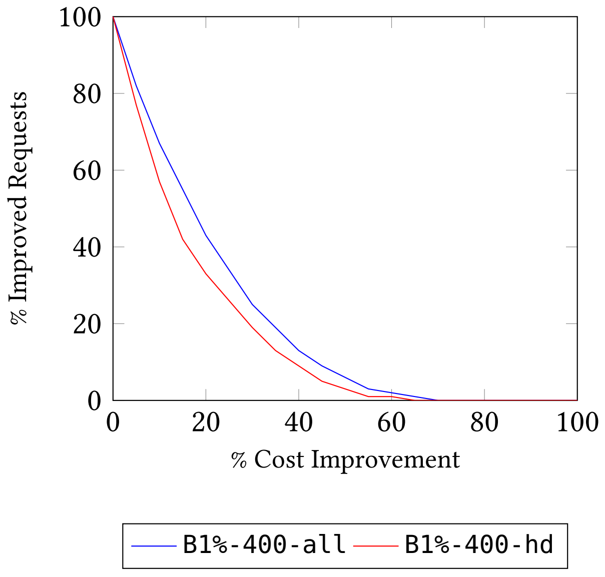


Figure 7.3: Distribution of the improvement of the assignment cost.

7.3.2 Assignment Structure

In this section we want to analyze the structure of the assignments found by our dispatcher. Table 7.7 displays the fractions of the different insertion types for the found single-transfer routes that do not break any constraints and therefore could be inserted. We observe that the impact of ORD transfers is almost negligible compared to the transfer ALS, with a share of only around 2-3%. This is a notable finding, as the running time is mostly dominated by the expensive ORD transfer point calculation and as the transfers ALS only provide transfer at stop and not optimal solutions in contrast to the ORD transfer points. It indicates, that when a passenger is already assigned to a vehicle, it is hard to find solutions where the taxi is shared, as the cost for a shared insertion seems to strongly affected by that. It is yet to explore how the portions may shift for scenarios with even higher request densities, as the routes are then expected to be longer with more options for ORD insertions and less availability of ALS insertions.

Table 7.7: Fractions of the different transfer types.

Instance	ord_{tr}	$\text{als}_{\text{tr,p}}$	$\text{als}_{\text{tr,d}}$
`B1%-400-all`	2.87%	68.62%	28.51%
`B1%-400-hd`	1.97%	59.66%	38.37%

7.3.3 Performance

To find ordinary transfers we implemented both strategies as described in section 5.2.2, the Dijkstra strategy and the strategy using CH based ellipses reconstruction. We investigated running times of both strategies on the `B1%-200-h3` instance. The running times are display in table 7.8. We notice a speedup of factor 45 when using the CHs based approach that result from the smaller search space of the CH based approach in comparison to the brute force Dijkstra solution.

Table 7.9 shows, that, as expected the number of ordinary transfer points exceeds the number of the ALS transfer points. This is because our algorithm only allows transfers at stop for the transfer ALS. Due to flawed code, further investigation of the performance of the transfer dispatching could not be done. The output files were flawed and rerunning the experiments would have exceeded the deadline.

Table 7.8: Average running times of our algorithm using the dijkstra and the ellipses reconstruction strategy on the `B1%-200-h3 dijkstra` (in ms).

Strategy	t_{total}	t_{ord}	$t_{\text{als,p}}$	$t_{\text{als,d}}$
Ellipses Reconstruction	989	958	25	5
Dijkstra	42312	42278	29	6

Table 7.9: Number of possibe transfer points for the different transfer types.

Instance	#tps ord	#tps als pveh	#tps als dveh
`B1%-400-all`	570997	700	514
`B1%-400-hd`	578931	352	270

7.4 Comparison to Heuristic Transfer Points (HTPS)

To close our evaluation, we want to briefly compare our results to the results of Willich from their work [20]. We can not compare the results directly as we do not use the same input data as well as different cost functions. But we will compare some quality values to gain a first idea of the differences between both approaches. It seems that vehicle utilization and service quality.

For the vehicle utilization of our algorithm, we see score very similar results compared to (HTPS). Willich was able to show a decrease in total vehicle operation time between 4% and 12% while we managed to achieve up to 5.9% in our experiments. While the HTPS increased travel time in almost every scenario by up to 8 minutes, we were able to reduce travel time for passengers. Of course, the runtime of our algorithm is drastically higher than the algorithm of Willich. We direct future work to a closer investigation of the difference between the solutions of Willich and the solutions produced by our algorithm. For that, the same cost function should be implemented and the algorithms for HTPS could be integrated into our framework to ensure comparability when running the experiments on the same hardware and inputs.

8 Conclusion

In this work we extended the notation of the Karlsruhe Rapid Ridesharing Dispatcher to also regard single-transfer journeys. We proposed an algorithmic outline that, when followed, computes solutions to the dynamic taxi sharing problem. We described the individual subproblems that have to be solved along the way. We then implemented an algorithm that computes single-transfer journeys and evaluated its performance and the produced solutions extensively on the real Berlin road network and realistic taxi sharing demand of the Berlin metropolitan area.

Our algorithm was able to find improved solutions for up to 16.8% of requests. We showed, that the enabling of transfers benefits the total operation cost of taxi sharing fleets by reducing the total operation time by up to 5.7% while preserving service quality for the passengers. The results promise even further improvements when demand and requests density is higher, than in our scenarios.

8.1 Future Work

As mentioned in section 6.6, our implementation of the constraint propagation is not optimal. While the short average route lengths do not promise a noticeable improvement, it would still be useful to implement the exact propagation as a basis for further code extensions. The CH searches to find the distances between a last stops or a pickup to all stops of another vehicle, as described in section 6.5.2, could be substituted by general BCH queries to improve overall runtime for transfer ALS computations.

Beyond these minor changes, we observe that transfers ALS are crucial on the overall improvements as the results from section 7.3 state, the transfer ORD do not have a big impact on the dispatch quality while also being the main factor for the increase in running time. Therefore, we direct future work to finding optimal transfers for the ALS case and decrease running times of our algorithm. It seems very promising to increase dispatch quality and reduce vehicle operations time further by allowing the second vehicle to drive a detour as well. It might also be worth to try disabling ordinary transfers to decrease running time by a factor of 40.

It further would be interesting to explore how vehicles with bigger capacities effect the single-transfer scenario as it seems natural to group multiple riders with similar origin and destination together in e. g. a bus. We argue that presently with the current running times multi-transfer trips are not feasible. The introduction of multi-hop journeys would probably result in an exponential increase of possible assignments, which the current version of our algorithm surely can not handle. But if in future work, the sets of transfer points could be reduced by efficient filtering and the running time could be improved, multi-transfer routes might be interesting to consider.

Bibliography

[1] Kay W. Axhausen and ETH Zürich. *The Multi-Agent Transport Simulation MATSim*. en. Ed. by ETH Zürich et al. Ubiquity Press, Aug. 2016. DOI: 10.5334/baw.

[2] David Banister. “Cities, mobility and climate change”. en. In: *Journal of Transport Geography* 19.6 (Nov. 2011), pp. 1538–1546. DOI: 10.1016/j.jtrangeo.2011.03.009.

[3] Joschka Bischoff, Michal Maciejewski, and Kai Nagel. “City-wide shared taxis: A simulation study in Berlin”. en. In: *20th International Conference on Intelligent Transportation Systems (ITSC)*. Yokohama: IEEE, Oct. 2017, pp. 275–280. DOI: 10.1109/ITSC.2017.8317926.

[4] BlaBlaCar. *Carpool Bus Travel Deals | BlaBlaCar*. URL: https://www.blablacar.co.uk/ (visited on 03/03/2025).

[5] Valentin Buchhold, Peter Sanders, and Dorothea Wagner. “Fast, Exact and Scalable Dynamic Ridesharing”. In: *Proceedings of the Symposium on Algorithm Engineering and Experiments (ALENEX)*. SIAM, 2021, pp. 98–112. DOI: 10.1137/1.9781611976472.8.

[6] Yen-Long Chen, Kuo-Feng Ssu, and Yu-Jung Chang. “Real-time Transfers for Improving Efficiency of Ridesharing Services in the Environment with Connected and Self-driving Vehicles”. en. In: *2020 International Computer Symposium (ICS)*. Tainan, Taiwan: IEEE, Dec. 2020, pp. 165–170. DOI: 10.1109/ICS51289.2020.00041.

[7] Jean-François Cordeau and Gilbert Laporte. “The dial-a-ride problem: models and algorithms”. en. In: *Annals of Operations Research* 153.1 (June 2007), pp. 29–46. DOI: 10.1007/s10479-007-0170-8.

[8] Pedro M. d’Orey and Michel Ferreira. “Can ride-sharing become attractive? A case study of taxi-sharing employing a simulation modelling approach”. en. In: *IET Intelligent Transport Systems* 9.2 (Mar. 2015), pp. 210–220. DOI: 10.1049/iet-its.2013.0156.

[9] Samuel Deleplanque and Alain Quilliot. *Transfers in the on-demand transportation: the DARPT Dial-a-Ride Problem with transfers allowed*. en. URL: https://hal.science/hal-00917197v1 (visited on 03/03/2025).

[10] Edsger W. Dijkstra. “A note on two problems in connexion with graphs”. In: *Numerische Mathematik* 1.1 (Dec. 1959), pp. 269–271. DOI: 10.1007/BF01386390.

[11] Robert Geisberger et al. “Contraction Hierarchies: Faster and Simpler Hierarchical Routing in Road Networks”. In: *Experimental Algorithms*. Ed. by Catherine C. McGeoch. Springer, 2008, pp. 319–333. DOI: https://doi.org/10.1007/978-3-540-68552-4_24.

[12] Yunfei Hou et al. “TASeT: Improving the Efficiency of Electric Taxis With Transfer-Allowed Rideshare”. en. In: *IEEE Transactions on Vehicular Technology* 65.12 (Dec. 2016), pp. 9518–9528. DOI: 10.1109/TVT.2016.2592983.

[13] S. Knopp et al. “Computing Many-to-Many Shortest Paths Using Highway Hierarchies”. In: *Proceedings of the 9th Workshop on Algorithm Engineering and Experiments (ALENEX 2007) and the Fourth Workshop on Analytic Algorithmics and Combinatorics, January 6, 2007, New Orleans, Louisana. Ed.: D. Applegate.* SIAM proceedings series. SIAM, 2007, pp. 36–45. DOI: `10.1137/1.9781611972870.4`.

[14] Moritz Laupichler and Peter Sanders. “Fast Many-to-Many Routing for Dynamic Taxi Sharing with Meeting Points”. In: *2024 Proceedings of the Symposium on Algorithm Engineering and Experiments (ALENEX).* SIAM, 2024, pp. 74–90. DOI: `10.1137/1.9781611977929.6`.

[15] Mustafa Lokhandwala and Hua Cai. “Dynamic ride sharing using traditional taxis and shared autonomous taxis: A case study of NYC”. en. In: *Transportation Research Part C: Emerging Technologies* 97 (Dec. 2018), pp. 45–60. DOI: `10.1016/j.trc.2018.10.007`.

[16] Renaud Masson, Fabien Lehuédé, and Olivier Péton. “The Dial-A-Ride Problem with Transfers”. en. In: *Computers & Operations Research* 41 (Jan. 2014), pp. 12–23. DOI: `10.1016/j.cor.2013.07.020`.

[17] Susan Shaheen, Adam Stocker, and Marie Mundler. “Online and App-Based Carpooling in France: Analyzing Users and Practices—A Study of BlaBlaCar”. In: *Disrupting Mobility: Impacts of Sharing Economy and Innovative Transportation on Cities.* Ed. by Gereon Meyer and Susan Shaheen. Cham: Springer International Publishing, 2017, pp. 181–196. DOI: `10.1007/978-3-319-51602-8_12`.

[18] Paolo Toth and Daniele Vigo. “Fast Local Search Algorithms for the Handicapped Persons Transportation Problem”. In: *Meta-Heuristics: Theory and Applications.* Ed. by Ibrahim H. Osman and James P. Kelly. Boston, MA: Springer US, 1996, pp. 677–690. DOI: `10.1007/978-1-4613-1361-8_41`.

[19] Inc. Uber Technologies. *UberX Share | Share Your Ride to Save.* URL: `https://www.uber.com/ch/en/ride/uberx-share/` (visited on 03/03/2025).

[20] Max Willich. “Improving Vehicle Detour in Dynamic Ridesharing using Transfer Stops”. MA thesis. Karlsruher Institut für Technologie (KIT), 2023. 70 pp. DOI: `10.5445/IR/1000165197`.

[21] Dominik Ziemke, Ihab Kaddoura, and Kai Nagel. “The MATSim Open Berlin Scenario: A multimodal agent-based transport simulation scenario based on synthetic demand modeling and open data”. en. In: *Procedia Computer Science* 151 (2019), pp. 870–877. DOI: `10.1016/j.procs.2019.04.120`.