Reoptimization in Dynamic Taxi Sharing with Meeting Points

Bachelor Thesis of

Tim Luis Gladbach

At the KIT Department of Informatics
Institute of Theoretical Informatics, Algorithm Engineering

First examiner:	Prof. Dr. Peter Sanders
Second examiner:	T.-T. Prof. Dr. Thomas Bläsius
First advisor:	M.Sc. Moritz Laupichler

01. December 2023 – 02. April 2024

Karlsruher Institut für Technologie
Fakultät für Informatik
Postfach 6980
76128 Karlsruhe

I declare that I have developed and written the enclosed thesis completely by myself. I have not used any other than the aids that I have mentioned. I have marked all parts of the thesis that I have included from referenced literature, either in their original wording or paraphrasing their contents. I have followed the by-laws to implement scientific integrity at KIT.

Karlsruhe, 02.04.2024

. . . . . .

(Tim Luis Gladbach)

Abstract

In this thesis, we present a reoptimization algorithm for the dynamic taxi sharing problem with meeting points. This problem involves managing a fleet of taxi-like vehicles and a continuous stream of customer requests, where customers wish to travel from their origin to their destination. The regular taxi sharing problem focuses on scheduling vehicles to pick up customers at their origins and drop them off at their destinations. Taxi sharing with meeting points extends this problem by incorporating meeting points, which can differ from the customer's original origin or destination. Customers need to walk to an agreed-upon pickup location, where they are picked up and then driven to a drop off location, from which they have to walk to their destination. In this work, we enhance the algorithm KaRRi, a dynamic taxi sharing dispatcher, with a reoptimization strategy. Dynamic problems are inherently limited by their real-time perspective on incoming data, which opens up the opportunity for reoptimization. We introduce Re-KaRRi, a reoptimization extension to the dynamic taxi-sharing dispatcher KaRRi, designed to schedule a fleet of taxi-like vehicles. By employing our strategy of *displacing insertions*, we further optimize existing routes by considering the reassignment of already assigned customers to improve overall costs. Our algorithm demonstrates strong performance in systems experiencing high capacity utilization. For such instances, we were able to reduce customer trip times by more than 27% and wait times by up to 49% compared to the original KaRRi algorithm.

Zusammenfassung

Wir stellen einen Reoptimierungsalgorithmus für das dynamische Taxi-Sharing-Problem mit Treffpunkten vor. Dieses Problem umfasst die Verwaltung einer Flotte von taxi-ähnlichen Fahrzeugen und das kontinuierliche Eintreffen von Kundenanfragen, bei denen die Personen von ihrem Ausgangspunkt zu ihrem Zielort gebracht werden möchten. Das reguläre Taxi-Sharing-Problem befasst sich mit der Verwaltung und Koordination von Fahrzeugen, um Personen von ihren Ausgangspunkten zu ihren Zielorten zu transportieren. Taxi-Sharing mit Treffpunkten erweitert dieses Problem durch das Hinzufügen von Treffpunkten, die vom ursprünglichen Ausgangs- oder Zielort einer Person abweichen können. Personen müssen zu einem zuvor vereinbarten Abholort zu Fuß gehen, wo sie abgeholt und dann zu einem Absetzort gefahren werden. Von diesem Absetzort aus müssen sie den Rest des Weges zu ihrem Ziel zu Fuß zurücklegen. In dieser Arbeit erweitern wir den Algorithmus KaRRi, einen dynamischen Taxi-Sharing-Dispatcher, um eine Reoptimierungsstrategie. Dynamische Probleme sind von Natur aus durch ihre Sicht auf die momentane Datenlage beschränkt, was die Möglichkeit eröffnet, Reoptimierungsstrategien zu verwenden, um alte Entscheidungen zu reevaluieren. Wir stellen Re-KaRRi vor, eine Reoptimierungserweiterung des dynamischen Taxi-Sharing-Dispatchers KaRRi, der für die Verwaltung einer Flotte von taxi-ähnlichen Fahrzeugen entwickelt wurde. Durch die Anwendung unserer entwickelten Strategie optimieren wir bestehende Routen, indem wir in Betracht ziehen, bereits zugewiesene Personen einem neuen Fahrzeug zuzuweisen, um die Gesamtqualität der Lösung zu verbessern. Unser Algorithmus beweist seine Fähigkeit besonders in Systemen, die eine hohe Kapazitätsauslastung erfahren. In solchen Fällen konnten wir die Fahrzeiten der Personen um mehr als 27% und die Wartezeiten um bis zu 49% im Vergleich zum ursprünglichen KaRRi-Algorithmus reduzieren.

Contents

List of Figures

List of Tables

1 Introduction

Today, the predominant ways of transportation from one point to another are either through individual transportation, such as cars, or public transit systems, including trains and buses. Each of these transportation options has its drawbacks. While individual transportation offers convenience and privacy, it is less energy-efficient, occupies more space in traffic, and produces a significant amount of pollution. Public transportation, on the other hand, despite being more eco-friendly, can be slow and inconvenient. Ridesharing systems with large fleets of taxi-like vehicles (taxi sharing) represent an alternative that unites the good of both. Today, ride-hailing systems like Uber and Lyft are primarily in use, but they exhibit the same problems as individual transportation. Taxi sharing solutions, which may even incorporate features like autonomous (electrical) vehicles, could be a more eco-friendly and efficient alternative while also being profitable.

In this thesis, we extend the algorithm KaRRi by Laupichler and Sanders [9], which extends taxi sharing with the possibility of using meeting points. These meeting points are pickup and dropoff locations that are not necessarily the customer's current position or intended destination. Customers need to utilize local transportation (e.g., walking or bicycles) to reach their pickup location, where they are picked up by a taxi-like vehicle. The vehicle drives the customer to a dropoff location, where they can again use local transportation to reach their destination. KaRRi incorporates an online dispatching system for taxi-like vehicle fleets. KaRRi computes optimal vehicle assignments for customers while also optimizing pickup and dropoff locations. Taxi sharing with meeting points is a complex problem due to the increased number of possible assignments. Laupichler and Sanders [9] utilize many tactics to make KaRRi as fast and as efficient as possible, which are discussed later in this thesis (see section 3.2).

We introduce Re-KaRRi (Reoptimized Karlsruhe Rapid Ridesharing), a reoptimization extension to KaRRi. Due to the nature of online algorithms, KaRRi computes the best assignment for a customer at a specific time. KaRRi does not know about future customers, which hinders it from incorporating trips of future customers into the decision-making for current customers. Our approach reevaluates previous assignments of KaRRi by adding the possibility of reassigning old customers if it benefits a new customer. We utilize a concept called fixed routes to efficiently find better solutions for new customers, which triggers reassignments for old customers.

For our experimental evaluation, we use realistic data sets to compare Re-KaRRi to KaRRi in terms of different quality aspects, such as vehicle driving times or customer trip times. We find that for instances that utilize small fleets, our algorithm is able to further optimize existing routes.

1.1 Related Work

The dynamic taxi sharing problem with meeting points extends the regular dynamic taxi sharing problem. In the taxi sharing problem, customers want to get from their origin to a destination location. A dispatcher finds an assignment to one of the vehicles in the fleet while adhering to defined constraints like a maximum arrival time. At the same time, the dispatcher tries to find an assignment that optimizes a cost function that weighs in quality aspects like travel time. The dynamic taxi sharing problem with *meeting points* further extends this by incorporating pickup and dropoff locations where customers are picked up and dropped off, respectively. The customer needs to walk a small distance from their origin to the pickup and from their dropoff to their destination, where a vehicle picks them up and drops them off. By incorporating pickup and dropoff locations, vehicles can drive more efficient routes by avoiding intensified detours when driving to the exact origin and destination of a customer.

The dynamic taxi sharing with meeting points demands an online solution. Therefore, dispatchers can base their decision-making only on vehicle assignments for customers they have conducted in the past. This opens up the opportunity to utilize reoptimization to reevaluate and improve assignments the dispatcher has computed in the past. Some work on dynamic taxi sharing with meeting points incorporates reoptimization tactics to reevaluate their decision-making. This section summarizes existing reoptimization approaches for this problem, although the amount of work on this specialized topic is sparse.

Alonso-Mora et al. [2] presented an algorithm that guarantees optimal assignments for all customers in the system. In their approach, they assume a batch of requests collected over a time window, a batch of vehicles, a cost function, and constraints. Then, the requests in the batch are assigned to vehicles in the vehicle batch. The request batch also contains requests assigned initially to vehicles in the vehicle batch. Thereby, Alonso-Mora et al. [2] consider reassigning customers to a new vehicle even though they have already been assigned to one. The only exceptions are customers who have entered their vehicle. Other than that, the trips of vehicles in the vehicle batch can change completely. When receiving a batch of requests and vehicles, the algorithm computes all feasible trip combinations of requests that can be executed by at least one vehicle. Then, based on the remaining feasible trips, the algorithm utilizes integer linear programming (ILP) to find the optimal assignments according to the cost function. By repeatedly reassigning batches of requests to batches of vehicles over time, the algorithm converges to an optimal solution while also satisfying defined constraints. However, Alonso-Mora et al. [2] do mention that it can be beneficial for tractability to use limitations like timeouts for the different phases of the algorithm. Another related approach is given by Fielbaum et al. [4]. Their algorithm extends the algorithm presented by Alonso-Mora et al. [2] to make it more applicable for actual world instances. They used the same algorithm base but utilized heuristics in the different phases to improve runtime.

Agatz et al. [1] provides a reoptimization approach for the related problem of dynamic ride sharing. In the ride sharing problem, there exist two different roles: drivers and riders. Both roles can announce a trip from an origin to a destination they want to conduct at a specific

time. The goal is to match riders with drivers so they can travel together to their destination. The origin and destination of the rider and driver do not need to be identical, so pickups and dropoffs are possible. Although this problem differs from dynamic taxi sharing with meeting points, Agatz et al. [1] describes a reoptimization approach to dynamic ride sharing that applies to our problem. He describes the concept of a *rolling horizon strategy*. In this approach, we consider some matchings between riders and drivers to be finalized, which means these matchings are fixed. Then, periodically, after a predefined amount of time, a reoptimization procedure is executed. This procedure computes new matchings for all riders and drivers, excluding finalized matches. Thereby, previously made matchings are reconsidered with the newly announced trips since the last reoptimization procedure.

Jung et al. [7] present a successive reoptimization method using *hybrid-simulated annealing*. In this approach, we assume periods in which we collect a batch of issued requests. Their approach is a reoptimization scheme that assigns the batch of new requests while also considering altering existing routes by moving assigned requests to other vehicles. Assigned requests that have already boarded their vehicle cannot be moved. The approach uses the tactic of *simulated annealing*, a stochastic relaxation method. We start with an initial solution: the current state of assigned and unassigned requests. Then, we perform multiple iteration procedures, randomly changing vehicle assignments in the current solution. If the random changes produce a better solution, we apply that change and continue the iteration process until convergence or the limit of iterations is exceeded. Suppose the new solution is worse than the current solution cost-wise. In that case, we still choose this solution as our currently best solution with a probability dependent on how much worse the new costs are and a *temperature parameter*. This temperature parameter gradually decreases over the iterations. This algorithm intends to explore solutions outside the local minima to find better solutions.

2 Problem Statement

In this chapter, we describe the problem of dynamic taxi sharing with meeting points. The first section provides some basic definitions for formulating the problem. In the last section, we discuss the cost function defined by Laupichler and Sanders [9], which provides a greedy online heuristic to optimize the solution in KaRRi.

2.1 Basic Definitions

The following definitions are based on the definitions and notation given by Laupichler and Sanders [9] in their paper on KaRRi.

Road Network. We describe a *road network* as a directed graph $G = (V, E)$ where edges correspond to road segments and nodes to intersections. The *travel time* of a road segment $e = (v, w) \in E$ is described by $l(e) = l(v, w)$. The *shortest path distance* (travel time) from insersection v to w ($v, w \in V$) is described by $\delta(v, w)$. In road networks, we never consider the distance but the time it takes to travel from the beginning node to the end node of an edge.

KaRRi uses two separate road networks, G_{psg} and G_{veh}, which describe the road network for pedestrians and vehicles, respectively. A customer can enter a vehicle at a node $v \in V_{psg} \cap V_{veh}$ which is reachable in both networks.

Vehicle, Stop. We consider a fleet F of *vehicles*. A vehicle $v = (l_i, cap, t^{min}_{serv}, t^{max}_{serv})$ is described by its initial location l_i, capacity cap and serving time $[t^{min}_{serv}, t^{max}_{serv})$. Furthermore, a vehicle has a current route $R(v) = \langle s_0(v), ..., s_{k(v)}(v) \rangle$ where $s_i(v)$ corresponds to the $i - th$ stop and a node in $V_{psg} \cap V_{veh}$. We use s_i instead of $s_i(v)$ for better readability when the context indicates which vehicle is meant. The current location of vehicle v is always between the previous (or current) stop s_0 and the next stop s_1. The total number of currently scheduled stops is defined by $k(v) = |R(v)| - 1$. At each stop, at least one passenger is dropped off and/ or picked up. If a vehicle reaches a stop, it remains there for a minimum of t^{max}_{stop}.

Request. Customers who want to travel from their current location $orig$ to their desired destination $dest$ can issue a request $r = (orig, dest, t_{req})$. An issued request consists of an origin location $orig$, destination location $dest$, and the time the customer made the request t_{req}. If a customer issues a request, t_{req} is also considered the earliest possible departure time, i.e., we do not allow pre-booking.

Pickup, Dropoff. In the problem of taxi sharing with *meeting points*, the customer is not necessarily picked up directly at their origin and dropped off at their destination. The customer needs to walk to a pickup location p where they are picked up by a vehicle. After the ride, the customer is dropped off at a dropoff location d from which they walk to their destination. A pickup p is a location within the walking radius ρ around the origin of a request. This means the customer should not need to walk longer from their origin to their pickup than ρ. Note that ρ refers to the walking time, not the distance. Analogously, a dropoff lies within the walking radius ρ of the customer's destination. As mentioned, pickups and dropoffs $p, d \in V_{psg} \cap V_{veh}$ need to be reachable for pedestrians and vehicles.

Insertion. If a new request is received, the dispatcher integrates the request into our route state. Route state refers to the current routes/ scheduled stops of vehicles in the fleet. This is done by assigning the request to one of the vehicles, where the dispatcher inserts the request into the vehicle's current route. Such an insertion we model with $\iota = (r, p, d, v, i, j)$. When ι is applied, the vehicle v drives from its i-th stop s_i to the pickup p, picking up customer r. Then, the vehicle continues its scheduled route until its j-th stop, after which it drives to the dropoff d, where v drops the customer r off. Note that many possible insertions of one request into the route state exist.

Customer, Passenger. We generally refer to issued requests as *customers*. We also define the term *passenger*, distinguishing it from the term *customer* by referring only to customers currently inside their assigned vehicle. This distinction between requests becomes important in later parts of this thesis.

2.2 Problem Formulation

In the problem of dynamic taxi sharing with meeting points, we are given two *road network graphs* G_{veh}, G_{psg} as well as a *fleet* F with the corresponding *current routes* $R(v)$ for every vehicle $v \in F$. Then, new requests $r = (orig, dest, t_{req})$ are issued continuously for customers who want to get from their origin to their destination.

This means we are faced with an online algorithm problem since we know only about requests that have been issued in the past and nothing about requests that will be issued in the future. Our goal for an issued request is to find a "good" insertion into one of the current vehicle routes. When doing this, we optimize a global cost function over all requests without knowing about future requests. These global costs include quality aspects like wait, walk, and trip times experienced by customers. We achieve this by defining a local cost function, a greedy online heuristic, to optimize the global costs. The exact quality aspects we consider for this local cost function are discussed in the following section. What makes an insertion a "good" insertion is then determined by the local cost function.

2.3 Cost Function

Since the algorithm described in this paper extends the KaRRi taxi sharing dispatcher, we use the same cost function as proposed by Laupichler and Sanders [9]. The cost function calculates the cost of an insertion $\iota = (r, p, d, v, i, j)$:

$$c(\iota) = t_{detour}(\iota) + \tau \cdot (t_{trip}(\iota) + t^{+}_{trip}(\iota)) + \omega \cdot t_{walk}(\iota) + c^{vio}_{wait}(\iota) + c^{vio}_{trip}(\iota)$$

The term $t_{detour}(\iota)$ denotes the additional time the vehicle v takes to complete its current route after insertion. Due to the insertion of p and d, the vehicle needs to take a detour from its current route, which can delay its completion. The term $t_{trip}(\iota)$ denotes the trip time of customer r, which is the time from t_{req} till the scheduled arrival at $dest$. The term $t^{+}_{trip}(\iota)$ denotes the increased trip times of customers already assigned to v. Similar to the completion of the route, the trip time of an already assigned customer can be prolonged due to the detours caused by pickup p and dropoff d. The term $t_{walk}(\iota)$ denotes the time spent walking by the customer r. The terms τ and ω are model parameters to determine how strongly the corresponding terms are weighed into the cost function.

The remaining terms of the cost function are soft constraint violation penalties:

$$c^{vio}_{wait}(\iota) = \gamma_{wait} \cdot max\{t_{wait}(r) - t^{max}_{wait}, 0\}, \text{where} \qquad t_{wait}(r) = t_{dep}(r) - t_{req}(r)$$
$$c^{vio}_{trip}(\iota) = \gamma_{trip} \cdot max\{t_{trip}(\iota) - t^{max}_{trip}(r), 0\}, \text{where} \qquad t^{max}_{trip}(r) = \alpha \cdot \delta_{veh}(orig, dest) + \beta,$$

where $t_{dep}(r)$ describes the departure time of the vehicle v from pickup p where customer r is picked up. These terms impose an additional penalty for high waiting times and long trip times, respectively. The remaining terms γ_{wait}, γ_{trip}, t^{max}_{wait}, α and β are model parameters used as weights for the corresponding terms.

Besides these soft constraints, there are hard constraints that result in a cost $c(\iota) = \infty$ if broken: The occupancy in a vehicle v must never exceed its capacity cap. The vehicle v has to reach the last stop of its route before the end of its service time. Every customer already assigned to v has to be picked up at its pickup within the maximum wait time t^{wait}_{max}. Every customer $\bar{r}$ already assigned to v has to arrive at their destination within their maximum trip time $t^{max}_{trip}(\bar{r})$.

As mentioned before, inserting a request into a route can cause the trip time and wait time of an already assigned customer to increase. The last two hard constraints restrict that a newly issued request causes already assigned customers to break the threshold of their soft constraints. Therefore, a request can only cause soft constraint violation penalties for itself, not for customers already assigned to the vehicle. For our algorithm, this definition is sufficient. A more detailed explanation, especially on the concrete calculation of the terms above, is given by Laupichler and Sanders [9].

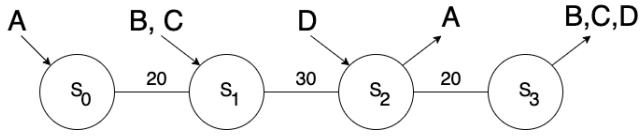
3 Preliminaries

Since our algorithm extends the KaRRi taxi sharing dispatcher by Laupichler and Sanders [9], this chapter will give an abstract view of important concepts used in KaRRi and how the dispatcher operates. KaRRi stores and consistently updates a global route state, which encapsulates each vehicle's scheduled stops and additional data. Then, KaRRi's search procedure finds the best assignment for an incoming request based on the current route state and updates the route state according to this new assignment. We will discuss this in detail in the following sections.

3.1 Route State

In order to compute optimal insertions, KaRRi needs to maintain all current routes $R(v)$ of vehicles v in the fleet F. For each stop, s_i, KaRRi maintains additional information besides the location of the stop. The vehicle's current schedule is stored as arrival and departure time $t_{arr}(s_i)$, $t_{dep}(s_i)$ for each stop s_i. The number of passengers traveling in the vehicle between stops s_i and s_{i+1} is stored as the occupancy $o(s_i)$. The customer picked up and dropped off at stop s_i are stored in $P(s_i)$ and $D(s_i)$ respectively. Another information that needs to be maintained is the leeway $\lambda(s_i, s_{i+1})$ from stop s_i to s_{i+1}, which will be explained later in this section.

The *route state* encapsulates all this information, and the scheduled stops $R(v)$ for all vehicles in the fleet. This is illustrated in the Figure 3.1: The stops are depicted as circles, each connected to its next stop. The travel time from s_i to s_{i+1} is written over the connecting lines. The travel time itself is not stored in the route state since it can be easily calculated with $t_{arr}(s_{i+1}) - t_{dep}(s_i)$. Customers picked up at a certain stop are depicted as letters with incoming arrows to the stop. Analogously, dropped off passengers are depicted as letters



s_i	s_0	s_1	s_2	s_3
arr	50	80	120	150
dep	60	90	130	160
o	1	3	3	0
P	{A}	{B,C}	{D}	{}
D	{}	{}	{A}	{B,C,D}

Figure 3.1: A scheduled route of a vehicle on the left and its associated data (arrival time, departure time, occupancy, picked up and dropped off customers) for each stop on the right.

with outgoing arrows from the stop. For example, at stop s_0, A is picked up, increasing the occupancy from 0 to 1 and adding A to the picked up requests $P(s_0) = \{A\}$. After that, the vehicle is currently set to drive to stop s_1, picking up B and C, increasing the occupancy to 3. Then, in the scheduled route, the vehicle would continue to stop at s_2, dropping off A while B and C remain in the vehicle.

Since KaRRi is an online algorithm, it maintains this route state constantly throughout its operation time. In the problem of dynamic taxi sharing with meeting points, many *events* can occur. Some of these events can affect the route state, which then needs to be updated. We will now discuss events that affect the route state.

Vehicle Startup. When a vehicle begins its serving time, it becomes active, and its route is initialized with its initial location $R(v) = \langle l_i \rangle$. Before that, v can not be considered for any requests issued since $R(v) = \langle\rangle$.

Vehicle Shutdown. With the vehicle's service time hard constraint (see section 2.3), we guarantee that a vehicle is stopping at its last stop when it reaches the end of its service time. Thereby, the vehicle's route consists of one stop $R(v) = \langle s_0 \rangle$. When the vehicle shuts down, its last stop is deleted, and $R(v) = \langle\rangle$.

Vehicle Arrival at Stop. If a vehicle is driving and arrives at its next stop s_1, it becomes the new first stop. This means that s_0 is removed, and for each remaining stop in $R(v) = \langle s_1, ..., s_{k(v)} \rangle$ its indice is reduced by one. This is done to maintain the convention that a vehicle is currently at stop s_0 or somewhere between the stops s_0 and s_1. Customers who are dropped off at the stop, where the vehicle has just arrived, begin walking to their destination.

Request Insertion. If a request r is inserted, we need to update the route $R(v)$ of v where r is inserted. If the pickup/ dropoff does not already exist as a stop, we need to add it. The vehicle's occupancy increases by one between the stops associated with the new customer's pickup and dropoff. The new customer r must be added to $P(p)$ and $D(d)$. As mentioned before, an insertion can lead to vehicle detours due to the new customer's pickup and dropoff. This can also affect stops after the pickup and/ or the dropoff. Therefore, we need to postpone the arrival and departure times of the scheduled stops by the detours caused by p and d. The leeway needs to be recalculated as well.

3.1.1 Leeways

Leeways play an important role in KaRRi and arise from the hard constraints defined in section 2.3. The cost function implements hard constraints that should not be broken. Based on these hard constraints, we can define a *maximum arrival time* for a stop $t_{arr}^{max}(s_i)$. This *maximum arrival time* marks the latest point in time a vehicle v can arrive at stop s_i without breaking any hard constraints for requests already assigned to v. Thereby, the *maximum arrival time* for a stop ensures that the hard constraints are not broken.

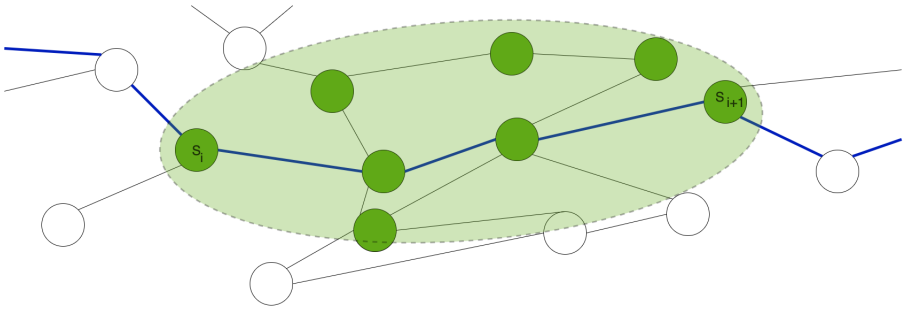


Figure 3.2: An illlustration of the leeway ellipse between two stops. Nodes s_i and s_{i+1} are consecutive stops. The blue edges mark the scheduled route of the vehicle. The stops within the leeway ellipse are colored green.

In combination with the (planned) *departure time* of a vehicle v from a stop $t_{dep}(s_i)$ we can calculate the leeway:

$$\lambda(s_i, s_{i+1}) = t^{max}_{arr}(s_{i+1}) - t_{dep}(s_i)$$

,which describes the amount of time a vehicle v can spend from s_i to s_{i+1}. The leeway defines an ellipse between stop s_i and s_{i+1}, which contains all feasible intermediate stops. This means we can incorporate one of these feasible stops into the route $R(v)$ without breaking any hard constraints. This is illustrated in Figure 3.2, where the edges of the scheduled route for the vehicle are colored in blue. The nodes at which the vehicle stops are marked as s_i and s_{i+1}. The nodes, which are colored green, represent the feasible intermediate stops that the vehicle can include in its current route without breaking any hard constraints.

Expressing the hard constraints of a vehicle's current route as leeways serves as a basis for elliptic pruning. Elliptic pruning acts as a speed-up technique for shortest path queries in KaRRi's search procedure (see section 3.2).

3.2 Search Procedure

KaRRi accepts issued requests $r = (orig, des, t_{req})$ and finds the best assignment for r. When receiving a request, first, KaRRi searches all pickup and dropoff locations in the walking radius ρ of $orig$ and $dest$ (PD-locations). These PD-locations are found using bounded Dijkstra searches which are bounded by the maximum walking time ρ from origin to pickup and from dropoff to destination. After that, KaRRi assesses all possible insertions in the order displayed in Figure 3.3 for each pickup and dropoff pair. It is also possible that KaRRi determines a pickup/ dropoff that already exists as a stop in a vehicle. For every type of insertion, KaRRi considers the route of every vehicle v according to the current route state. It checks whether an insertion of any pair of pickups and dropoffs into the route of v is possible with this insertion type. For instance, the ordinary insertion type calls for assignments $\iota = (r, p, d, v, i, j)$ where the pickup p is inserted between a pair of stops s_i and s_{i+1} of v while the dropoff d is inserted between a later pair of stops s_j and s_{j+1} with $j > i$. While trying all these different insertions, KaRRi maintains a currently best assignment ι^*,

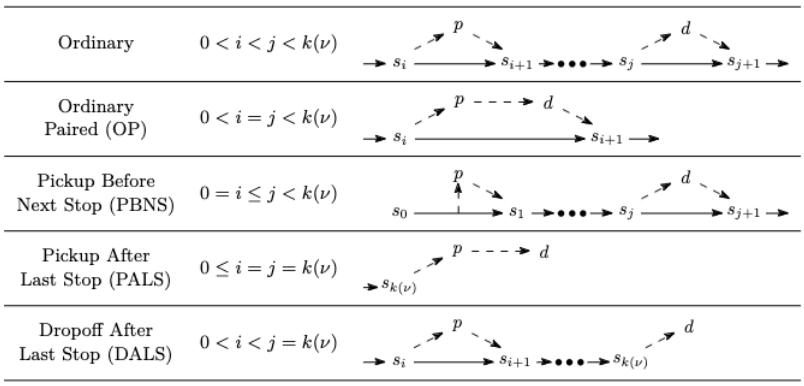


Figure 3.3: An illustration of insertion types by Laupichler and Sanders [9]. For each insertion type, it is illustrated how pickup and dropoff of $\iota = (r, p, d, v, i, j)$ are incorporated into the route of a vehicle. The figure also suggests the detour a vehicle has to make for a pickup/ dropoff.

which is continuously updated if the currently assessed insertion produces a better result in terms of the cost function. The *search procedure* calculates the best insertion of a request r into the current route state. KaRRi utilizes many concepts to perform the search procedure as efficiently as possible. Some of these concepts are explained superficially in the following section. A more thorough explanation of these concepts is given by Buchhold et al. [3] and by Laupichler and Sanders [9].

Contraction Hierarchies. *Contraction Hierarchies (CHs)*, introduced by Geisberger et al. [5], are a speed-up technique for finding the shortest paths in a road network that exploits the inherent hierarchical structure of road networks. A CH pre-processes the road network $G = (V, E)$ by first defining a heuristic hierarchy on the network nodes. After each node n receives its ranking $rank(n)$, we *contract* the nodes in the order of increasing ranks. For contracting a node n, we remove n temporarily from the road network G. If n lays as the only node between the shortest path of two other nodes v, w: $\delta(v, w) = l(v, n) + l(n, w)$, we preserve this shortest path through a *short-cut edge* (v, w). The (time) distance of the new edge between the nodes v,w is the shortest distance between v and w: $l^+(v, w) = l(v, n) + l(n, w)$. After contracting all nodes, we receive the CH $G^+ = (V, E^+)$, where E^+ contains E as well as all short-cut edges.

We can utilize the CH for shortest path queries from one origin node o to a destination node d. We define an upward graph $G^\uparrow = (V, E^\uparrow)$ and an downward graph $G^\downarrow = (V, E^\downarrow)$, where $E^\uparrow = \{(v, w) \in E^+ | rank(v) < rank(w)\}$ and $E^\downarrow = \{(v, w) \in E^+ | rank(v) > rank(w)\}$. A CH query then runs a forward Dijkstra search from o in $G^\uparrow$ and a reverse Dijkstra search from d in $G^\downarrow$. If these two searches meet, we eventually find the shortest path between o and d. The benefit of the CH pre-processing is the significantly smaller *search space* when performing a Dijkstra search on $G^\uparrow$ or $G^\downarrow$. The search space of the forward Dijkstra search, starting from o only contains nodes which are higher in rank. The same applies to the search space of the reverse Dijkstra search, which starts from d and can only traverse edges to nodes of higher rank. Note that this implies that there exists a shortest path between o and d in G^+,

which only consists of upward edges, followed by only downward edges. This is proven by Geisberger et al. [5].

Bucket Contraction Hierarchy Searches. *Bucket Contraction Hierarchy (BCH)* searches were introduced by Knopp et al. [8] and Geisberger et al. [5]. BCH searches are provided with a source node $s \in V$, a set of target nodes $T \subseteq V$, and find all shortest paths from s to all target nodes T. The BCH search also utilizes the CH G^+ of the road network G. The main idea of BCHs is to calculate *(target) buckets* $B^{\downarrow}(v)$ for all nodes $v \in V$. These buckets contain tuples $(t, \delta^{\downarrow}(v,t))$ where $t \in T$ is a target node and $\delta^{\downarrow}(v,t)$ the shortest path from v to t in the downward CH graph $G^{\downarrow}$. The buckets can be calculated by running a reverse search for every $t \in T$ in $G^{\downarrow}$. Every time a node v is settled, we can add $(t, \delta^{\downarrow}(v,t))$ to v's bucket $B^{\downarrow}(v)$. When we want to know the shortest path from s to one of the target nodes t, we simply run a forward search on $G^{\uparrow}$ starting from s. We only need to scan buckets for entries with t and update our current shortest path. We no longer need a reverse search from t in $G^{\downarrow}$. The same can be done for a set of source nodes $s \in S$ which fill the *(source) buckets* $B^{\uparrow}(v)$ of v with a forward search on $G^{\uparrow}$ when settling v.

With pre-calculating buckets we have the advantage of only needing to traverse the search space once, to create the buckets or to scan their entries. However, this comes with an increased usage of memory.

Elliptic Pruning. In section 3.1.1, we discussed leeways, which determine the maximum time the vehicle is allowed to take from stop s_i to s_{i+1} without breaking any hard constraints. This leeway defines an ellipse containing feasible stops that could be inserted into the route between stop s_i and s_{i+1}. When trying to insert a request with a pair of pickup p and dropoff location d, we need to calculate their distance to stops inside vehicle routes. BCH searches *(elliptic BCH searches)* are employed to accomplish this, and we generate buckets for stops. To make this BCH search as efficient as possible, KaRRi uses the concept of *elliptic pruning*, which was introduced by Buchhold et al. [3]. The idea of elliptic pruning is to generate bucket entries for consecutive stops s_i and s_{i+1}, only for nodes within the leeway ellipse. Other nodes outside the ellipse cannot be integrated into the route since they would break hard constraints. Elliptic pruning reduces the number of bucket entries and the number of feasible vehicles when performing elliptic BCH searches for PD-locations.

Elliptic BCH Searches with Sorted Buckets. In the dynamic taxi sharing with meeting points algorithm KaRRi, presented by Laupichler and Sanders [9], the concept of *sorted buckets* was introduced. Sorted buckets help further speed up elliptic BCH searches by reducing the number of bucket entries scanned. We explain the basic idea of sorted buckets for pickups, which is analogous to dropoffs. When running an elliptic BCH search for a pickup p, we scan source bucket entries $(s_i, \delta^{\uparrow}(s_i,v)) \in B^{\uparrow}(v)$ since we want to insert p after stop s_i. With $(s_i, \delta^{\uparrow}(s_i,v))$ we can calculate the preliminary distance from p to s_i with $\delta^{\uparrow}(s_i,v) + \delta^{\downarrow}(v,p)$. The entry is only relevant if $\delta^{\uparrow}(s_i,v) + \delta^{\downarrow}(v,p) \leq \lambda(s_i, s_{i+1})$. Otherwise, p is not within the ellipse between s_i and s_{i+1}, at least when going through v. Based on the leeway for two consecutive stops, we can define $\lambda_{res}(s_i,v) = \lambda(s_i, s_{i+1}) - \delta^{\uparrow}(s_i,v)$, the *remaining leeway* of a bucket entry $(s_i, \delta^{\uparrow}(s_i,v))$ at node v. The remaining leeway sets an upper bound for $\delta^{\downarrow}(v,p)$

to be considered a feasible pickup detour between s_i and s_{i+1}. We sort the bucket entries of a node v by their remaining leeway in descending order. If an elliptic BCH search for p scans the sorted bucket entries of v, it only needs to scan the entries until $\delta^{\downarrow}(v, p) > \lambda_{res}(s, v)$. Entries after s have an equal or even lower remaining leeway, which makes them irrelevant for p. These sorted buckets have to be maintained, which can cause some overhead, which is generally not high since bucket sizes are small.

SIMD Parallelism. KaRRi uses *single-instruction multiple-data (SIMD) parallelism*, a key feature of many modern CPUs. SIMD parallelism allows one to perform a single operation on multiple data points at the same time. In KaRRi, this is used for speeding up *bundled searches*. A technique where searches in Dijkstra-based algorithms of k sources can be continued simultaneously (see Laupichler and Sanders [9]).

4 Reoptimization Concepts

This chapter discusses concepts that play a crucial role in our reoptimization algorithm. In the first section, we reevaluate different reoptimization approaches from section 1.1 for the problem of taxi sharing with meeting points. In this section, we also introduce our approach *displacing insertions*, which considers displacing requests from routes to create a better assignment for the issued request cost-wise. This section also sets limitations to the requests we allow displacing. Then, we introduce the concept of the *fixed route state*, which is an additional route state to the normal route state we discussed in section 3.1. The fixed route state is distinguished from the normal route state by only consisting of requests and their associated stops, which cannot be displaced. After that, we discuss the reassigning of requests that a displacing insertion has displaced. Last, we calculate the cost changes resulting from displacing insertion. These cost changes consist of preliminary cost savings from removing requests from a route and the following insertion costs of the reassignments and the displacing insertion itself.

4.1 Reoptimization Space

4.1.1 Reoptimization Approaches

We find that the majority of existing reoptimization approaches for this problem primarily focus on reassigning requests that have previously been assigned to vehicles. Furthermore, within these approaches, certain previously assigned requests are deemed fixed and are not considered for reassignment (e.g., customers who entered their vehicle). As discussed in section 1.1, most approaches undertake reoptimization for a batch of requests and vehicles. Another common trait among these approaches is their periodic execution of reoptimization at set time intervals. Our method, *displacing insertions*, distinguishes itself by deviating from the last two aspects.

4.1.2 Displacing Insertions

In our approach, when we receive a request, we do not only consider inserting it into one of the routes $R(v)$ in the route state. We also consider displacing already inserted requests to find an even better insertion for the issued request. Displacing a request means revoking its current assignment to a vehicle. These displaced requests need to be reassigned, which is discussed in section 4.3. Displacing requests can also lead to stops being removed from

a vehicle, since without the displaced request, the vehicle does not need to stop at their pickup/ dropoff. Removing stops from vehicles results in new routes, which in turn makes new assignments possible.

When considering displacing already assigned requests, we mainly aim to reduce the costs of the issued request. This is built upon the fact that when removing requests, we also alter the current routes of vehicles. This can potentially result in better routes for inserting the issued request. The displacement of an already assigned request can also positively affect the displaced request itself. When the now displaced request was inserted into the route state previously, the route state could have contained worse routes for the request than the route state now. This can also go the other way around and will be evaluated in section 6.4.

4.1.3 Limitation

When we displace requests, we set one limitation on which already assigned requests can be displaced. This limitation is that we do not allow displacing passengers, i.e., customers who have already entered the vehicle. Otherwise, we would have to implement the possibility of switching vehicles mid-travel, which is not supported by KaRRi and could be quite uncomfortable for the customer.

4.2 Fixed Route State

As mentioned before, we allow the displacing of already assigned requests, except for passengers who have already entered the vehicle. We also discussed that displacing requests can lead to stops being removed from a route $R(v)$ as well. We now introduce the concept of the *fixed route state*, which results from displacing all customers except the passengers of v from all vehicles in the normal route state. This results in *fixed routes* $\overline{R}(v)$ consisting only of a subsequence of stops from $R(v)$ which we call *fixed stops*. More precisely, these fixed stops are the stop s_0 and the dropoff locations of passengers in v. The data associated with each fixed stop also differs from the data in the normal route state. The data of the fixed route state is only based on *fixed stops* and passengers who have entered their vehicles. Figure 4.2 below illustrates the concept of the fixed route state based on the fixed route of a vehicle.

Figure 4.2 displays the normal route at the top and the corresponding fixed route at the bottom. We can see not only the scheduled stops but also the stops that have already been visited, where customers, who are now passengers, have been picked up. At stop s_0, A is still a passenger of the vehicle, and C enters as a new passenger. This means the only *fixed stops* of the normal route besides s_0 are s_2 and s_5 since these are the dropoff locations for A and C, respectively. Note that even customer E is not part of the fixed route, although their pickup and dropoff locations are fixed stops. Another characteristic of fixed routes is the earlier arrival time at each stop, which leads to greater leeways. Greater leeways

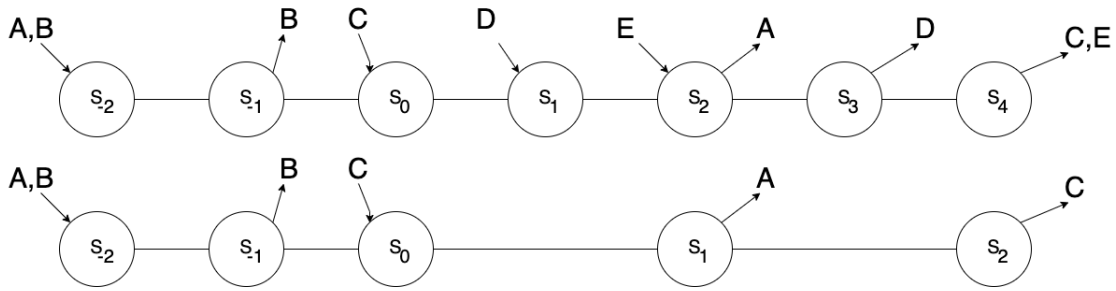


Figure 4.1: Normal route (displayed at the top) with pickups and dropoffs in comparison to the fixed route (shown at the bottom)

increase the number of feasible pickup/ dropoff locations for this vehicle. This can increase the number of possible assignments for an issued request.

The fixed route state helps simplify the search procedure that considers displacing requests by simply displacing all requests that are not passengers of a vehicle. So when considering a displacing insertion, we try inserting the issued request into the fixed route state, i.e., we displace all displaceable requests. We discuss this in greater detail in the following chapter 5.

4.3 Reassigning Requests

When we use a displacing insertion for an issued request, we displace all displaceable requests in the route $R(v)$ of the chosen vehicle v. Then, the issued request is inserted into the modified route of v, i.e., the fixed route $\overline{R}(v)$. The normal route state is then updated with the new route $\overline{R}(v)$ and its associated data for the vehicle v. The displaced requests are now without an assignment and need to be reassigned. We reassign these requests to the updated route state. This means that the requests can be reassigned to their previous vehicle. When a request is reassigned, we require that the new assignment uses the same pickup as the request received in its initial assignment. The dropoff can be changed to a new dropoff, again within the walking radius ρ to its destination.

4.4 Cost Changes

As discussed in the section on fixed routes, when considering a displacing insertion, we try inserting the issued request $\overline{r}$ in the fixed route (with $\overline{i}$) and reassigning the displaced requests afterward. When considering a displacing insertion for a request in the route of a vehicle v, we need to exchange the route $R(v)$ in the normal route state with the route $\overline{R}(v)$. This causes changes in the global cost function. The previous assignment of a displaced request has been revoked for the moment. This means that the costs of its previous assignment need to be recalculated since they are no longer part of the global

costs. Passengers of the vehicle v reach their dropoff earlier, which causes their previously calculated costs to shrink. These saved costs when exchanging a route $R(v)$ with $\overline{R}(v)$ are described in the first subsection.

The second subsection discusses the new costs that result from reassigning the displaced requests and the costs of the displacing insertion itself. Subtracting the new costs from the saved costs gives us the total cost changes and determines if a displacing insertion is worth considering.

4.4.1 Costs Saved

The following formula determines reduced costs that arise when we exchange the route $R(v)$ with its fixed route $\overline{R}(v)$. The formula is directly deduced from the cost function and considers the same quality aspects. We first give a rough description of the formula and unravel it piece by piece. For the formula, we define B as the passengers in the vehicle and $\Theta = \bigcup_{i=1}^{k(v)} P(s_i)$ as the displaced requests. For better readability we use $R = R(v) = \langle s_0, ..., s_k \rangle$ and $\overline{R} = \overline{R}(v) = \langle s_0, ..., s_{\overline{k}} \rangle$.

$$c_{saved}(R \to \overline{R}) = \Delta_{detour}(R, \overline{R}) + \underbrace{\tau \cdot T_{trip}^{saved}(B) + C_{vio}^{saved}(B)}_{\text{passengers}} + \underbrace{\tau \cdot T_{trip}(\Theta) + \omega \cdot T_{walk}(\Theta) + C_{vio}(\Theta)}_{\text{displaced requests}}$$

$\mathbf{\Delta_{detour}(R, \overline{R})}$. This term determines how much earlier the vehicle finishes its current route, i.e., arrives at its last currently scheduled stop, when using only the fixed stops compared to the normal route. This can be calculated by subtracting the arrival times at the last stops s_k and $s_{\overline{k}}$:

$$\Delta_{detour}(R, \overline{R}) = t_{arr}(s) - \overline{t}_{arr}(s_{\overline{k}}),$$

where t_{arr} and $\overline{t}_{arr}$ represent the arrival times for the routes R and $\overline{R}$ respectively. Note that this term coincides with the $t_{detour}(\iota)$ term in the cost function, which determines how much more time it takes for the vehicle to reach its last stop due to ι.

$\mathbf{T_{trip}^{saved}(B)}$. This term determines the saved trip time for passengers in the vehicle. The trip time starts when the request is issued and ends when the customer arrives at their destination. To calculate the difference between the trip time in R and $\overline{R}$ of a passenger, we need their arrival time at their dropoff. This arrival time can be retrieved easily since we store the dropped off requests for each stop. In the following formula, $\overline{D}(s)$ denotes the requests dropped off at stop s in $\overline{R}$.

$$T_{trip}^{saved}(B) = \sum_{\substack{s \in \overline{R} \\ s \neq s_0}} |\overline{D}(s)| \cdot (t_{arr}(s) - \overline{t}_{arr}(s))$$

Theoretically, the trip time would also include the walking time from dropoff to destination. However, these walking times are identical on both routes and do not contribute to the difference in trip times.

$\mathbf{C_{vio}^{saved}(B)}$. Remember that the cost function contains soft constraint violation penalties. This term aims to represent any reductions in these penalty costs that may be incurred by the shorter trip times of remaining passengers. A reduction may only appear for a passenger r if their original insertion violates the trip time soft constraint. These consist of violations against wait and trip times. Penalties for excessive wait times cannot change since the wait times for passengers in the vehicle are the same, independent of the route. To be able to calculate the reduction in the trip time penalty, we need to calculate the different trip times for a request $r = (orig, dest, t_{req})$:

$$t_{trip}(r) = t_{arr}(d) - t_{req} + \delta_{psg}(d, dest)$$
$$\bar{t}_{trip}(r) = \bar{t}_{arr}(d) - t_{req} + \delta_{psg}(d, dest),$$

where d refers to the dropoff location of a request. Now we can calculate the saved penalties:

$$C_{vio}^{saved}(B) = \underbrace{\sum_{r \in B} \gamma_{trip} \cdot max\{t_{trip}(r) - t_{trip}^{max}(r), 0\}}_{\text{violations in } R} - \underbrace{\sum_{r \in B} \gamma_{trip} \cdot max\{\bar{t}_{trip}(r) - t_{trip}^{max}(r), 0\}}_{\text{violations in } \bar{R}}$$

$\mathbf{T_{trip}(\Theta)}$. Since we remove the requests $r \in \Theta$, we save their trip cost:

$$T_{trip}(\Theta) = \sum_{r \in \Theta} t_{trip}(r)$$

$\mathbf{T_{walk}(\Theta)}$. When we remove the request $r \in \Theta$, we also save their walking cost:

$$T_{walk}(\Theta) = \sum_{r \in \Theta} \delta_{psg}(orig, p) + \delta_{psg}(d, dest)$$

Note that we also add the walking cost from the origin to the pickup, although the customer is already walking to it. We can do this due to a small detail in the implementation: When reassigning r, we do not change its origin and its previously calculated arrival time at the pickup. Thereby, the walking time is re-added when reassigned, but we use their true arrival time at the pickup to find the best-fitting vehicle.

$\mathbf{C_{vio}(\Theta)}$. Additionally, we save the soft constraint violation penalties for removed requests:

$$C_{vio}(\Theta) = \sum_{r \in \Theta} \underbrace{\gamma_{wait} \cdot max\{t_{wait}(r) - t_{wait}^{max}, 0\}}_{\text{wait time penalty}} + \underbrace{\gamma_{trip} \cdot max\{t_{trip}(r) - t_{trip}^{max}(r), 0\}}_{\text{trip time penalty}},$$

where $t_{wait}(r) = t_{dep}(p) - t_{req}(r)$.

4.4.2 New Costs

We insert the initial request $\overline{r}$ in the new route $\overline{R}(v)$ of v with the insertion $\bar{\iota}$. This is done before we reassign the displaced requests. The costs $c(\bar{\iota})$ of the insertion of $\overline{r}$ into $\overline{R}(v)$ can be calculated normally using the cost function (see section 2.3). The route state now includes $\overline{R}(v)$ with the inserted request $\overline{r}$. Then we reassign the displaced requests Θ to the modified route state, which results in the new costs $\sum_{r \in \Theta} c(\iota(r))$, where $\iota(r)$ is the insertion for displaced request r.

The total cost changes for the displacing request can then be written as:

$$c_{change}(\bar{\iota}) = c_{saved}(R(v) \rightarrow \overline{R}(v)) - c(\bar{\iota}) - \sum_{r \in \Theta} c(\iota(r)).$$

If the total cost change is greater than 0, we consider the displacing insertion to be successful. Otherwise, we do not consider the displacing insertion to be worth it. This results in reverting the performed changes on the route state. Section 5.4 discusses how this reverting is performed.

5 Reoptimization Algorithm

In this chapter, we present our reoptimization algorithm *Re-KaRRi*. The first chapter outlines the basic idea of the algorithm. The second section again discusses the fixed route state and how it is maintained throughout KaRRi's operating time. In the third section, we discuss buckets, which we already mentioned in chapter 3. This time, we focus more on the maintenance of buckets for fixed stops since the consecutive stops in fixed routes differ from those in normal routes. Then, in the fourth section, we thoroughly explain our reoptimization algorithm Re-KaRRi. The last section explains some aspects of Re-KaRRi in more detail and touches on an oversight in our implementation that causes inaccuracies in the fixed routes.

5.1 Overview

This section gives an overview of our reoptimization algorithm and how it extends KaRRi. As mentioned in chapter 3, KaRRi is an online algorithm that maintains its *route state* throughout its operation time. For our algorithm, we implemented a second route state into KaRRi, which maintains the *fixed routes* for each vehicle. Like the normal route state, the *fixed route state* is continuously updated throughout the operation time. How this *fixed route state* is maintained is explained in the following section 5.2.

Remember that KaRRi performs its *search procedure* based on its *route state*. We made the route state for our algorithm exchangeable. We provide the search procedure with a request r and a route state R. This allows us to run search procedures on the *normal route state* and *fixed route state*.

When a request is issued, we run a search procedure on the normal and fixed route state resulting in two insertions/ assignments $\iota = (r, p, d, v, i, j)$ and $\bar{\iota} = (r, \overline{p}, \overline{d}, \overline{v}, \overline{i}, \overline{j})$ and associated costs $c_R(\iota)$ and $c_{\overline{R}}(\bar{\iota})$. The index in the cost functions c emphasizes for which route state the insertion is considered. Then we compare the two costs $c_R(\iota)$ and $c_{\overline{R}}(\bar{\iota})$. If $c_R(\iota) < c_{\overline{R}}(\bar{\iota})$, we consider a displacing insertion. We consider the difference $d := c_R(\iota) - c_{\overline{R}}(\bar{\iota})$ as saved costs. We then calculate the additional saved costs $c_{saved}(R(\overline{v}) \rightarrow \overline{R}(\overline{v}))$, insert the request r into $\overline{R}(\overline{v})$ and perform the reassignments of displaced requests. If the total cost change (see section 4.4) plus d is greater than zero, we consider the displacing insertion successful and keep the applied changes. Otherwise, we revert the changes in the route state and use the assignment ι for r, which we calculated using the normal route state.

5.2 Fixed Route State

This section focuses on the maintenance of the fixed route state. We explained the concept in section 4.2. Like the normal route state, the fixed route state is maintained throughout the operating time. In section 3.1, we discussed how KaRRi maintains the route state by updating it when certain *events* affect the route state. The same applies to the fixed route state, with the difference that some events affect it differently. For instance, the event of a vehicle arriving at its next stop is different for both route states since the stop s_1 in $R(v)$ can be different from the next scheduled stop in $\overline{R}(v)$. The normal route state displays the vehicles' actual scheduled routes. The fixed route state is just an additional route state that can be provided to the search procedure with a request, simplifying displacing insertions. Fixed routes only become the actual scheduled route of a vehicle if a displacing insertion is successfully performed. Therefore, we need to update the fixed route state according to events that occur in the normal route state.

Vehicle Startup. The behavior of the fixed route state when a vehicle begins its service time is identical to the normal route state. Like the normal route state, the fixed route state initializes the route of the activated vehicle v with its initial location $\overline{R}(v) = \langle l_i \rangle$. This conforms to the definition of fixed stops where s_0 is considered fixed.

Vehicle Shutdown. As mentioned in the normal route state, the service time hard constraints (see section 2.3) guarantee that a vehicle is stopping at its last stop when it reaches the end of its service time. Therefore, the fixed route of a vehicle v consists only of one stop $\overline{R}(v) = \langle s_0 \rangle$ when the vehicle reaches the end of its service time. When the vehicle shuts down, we remove its last stop $\overline{R}(v) = \langle\rangle$.

Vehicle Arrival at Stop. When a vehicle arrives at its next stop s in the normal route state $R(v) = \langle s_0, s, ..., s_k \rangle$, s becomes the new first stop $R(v) = \langle s, ..., s_k \rangle$. The stop s is not necessarily a fixed stop but becomes one when v arrives. Therefore the fixed route is modified from $\overline{R}(v) = \left\langle s_0, \overline{s}, ..., \overline{s}_{\overline{k}} \right\rangle$ to $\overline{R}(v) = \left\langle s, \overline{s}, ..., \overline{s}_{\tilde{k}} \right\rangle$ or $\overline{R}(v) = \left\langle \overline{s}, ..., \overline{s}_{\overline{k}} \right\rangle$ if s was a fixed stop $s = \overline{s}$. The index $\tilde{k} = \overline{k} + q$, where q is the number of new stops created in $\overline{R}(v)$ after the pickup of customers at stop s. What that means becomes apparent in the following.

When the vehicle arrives at a new stop s, it picks up new customers $P(s)$. These customers become new passengers, which makes their dropoff locations fixed stops as well. We need to determine between which stops we need to insert their dropoffs in the fixed route $\overline{R}(v)$. Since we know the position of the stop s' where a dropoff d is scheduled in $R(v)$, we can traverse through $R(v)$ and count the number of fixed stops that are scheduled before s' and determine the insert index of s' into $\overline{R}(v)$.

After inserting the new stops, we need to recalculate the associated arrival times, departure times, and leeways of route $\overline{R}(v)$.

Request Insertion. When a request r is inserted into $R(v)$, $\overline{R}(v)$ is only affected if r is picked up by v at stop s_0. Otherwise, r is picked up at one of the later stops, which makes r a displaceable request. If r is picked up at s_0, they automatically become a passenger of v, and their dropoff needs to be inserted into $\overline{R}(v)$.

5.2.1 Fixed Buckets

Recall that KaRRi generates buckets for stops in vehicle routes, so the search procedure can utilize elliptic BCH searches to compute feasible vehicles and the required shortest path distances for assignments (see section 3.2). Our goal is to be able to switch the used route state for the search procedure. This only works if we maintain *fixed buckets*, which we can exchange with the normal buckets when running a search procedure. The maintenance of the buckets is also dependent on events that occur in the problem of taxi ridesharing. The way we update buckets for the normal and fixed route states is identical.

The insertion of a new stop or even the insertion of a request with existing stops as pickup and dropoff can cause changes in leeways of consecutive stops s_i and s_{i+1}. If this happens, the leeway ellipse of feasible stops between s_i and s_{i+1} can become smaller, causing some detours through stops to become infeasible. We need to represent this change in the buckets as well. Otherwise, the search procedure could perform inefficiently by unnecessarily considering additional (actually infeasible) pickup/ dropoff detours. If a new stop $\tilde{s}$ is inserted between two consecutive stops $R(v) = \langle s_0, ..., s_i, \tilde{s}, s_{i+1}, ..., s_k \rangle$, we need to create source and target buckets for this stop. Source buckets since it becomes the source node between itself and s_{i+1} and target buckets since it is the target node between s_i and itself. We only create target buckets for an inserted stop $\tilde{s}$ if it has a previous stop and source buckets if it has a following stop. With the addition of a new stop or the insertion of a request at existing stops, the max arrival times $t_{arr}^{max}(s)$ for stops in v can change. If the max arrival times change, the leeways change as well. In this case, we update the buckets by removing bucket entries $(s, \delta_{veh}(s, v))$ from nodes v, which become infeasible for stops s. When a vehicle arrives at its next stop, we remove the source buckets from the first stop and the target bucket from the second stop.

5.3 Re-KaRRi

We now present our reoptimization algorithm *Re-KaRRi*, which utilizes displacing insertions. We explain the algorithm by providing simplified pseudo-code in Algorithm 1, which we then explain step by step.

The algorithm receives an issued request r as input. In lines 1-2, we perform the search procedures on the normal and fixed route states, which result in optimal assignments ι and $\bar{\iota}$, respectively. The *searchProcedure* method is KaRRi's search procedure, which we have adjusted to take a route state as an argument as well. Besides the request r, all other parameters in ι and $\bar{\iota}$ can differ. Then, in lines 3-4, we calculate the costs of insertion into the

Algorithm 1: Dispatching procedure with reoptimization in Re-KaRRi

Data: request $r = (orig, dest, t_{req})$

1 $\iota = (r, p, d, v, i, j) \leftarrow$ searchProcedure(r,R)
2 $\overline{\iota} = (r, \overline{p}, \overline{d}, \overline{v}, \overline{i}, \overline{j}) \leftarrow$ searchProcedure(r, $\overline{R}$)
3 normalCost $\leftarrow c_R(\iota)$
4 fixedCost $\leftarrow c_{\overline{R}}(\overline{\iota})$
5 savings $\leftarrow$ normalCost $-$ fixedCost
6 **if** *savings* $\leq$ *0* **then**
7 R.insert(ι)
8 **return**
9 savings $\leftarrow$ savings + $c_{saved}(R(\overline{v}) \rightarrow \overline{R}(\overline{v}))$
10 $\Theta \leftarrow$ displaceableRequests($R(\overline{v})$)
11 exchangeRoute($R(\overline{v})$, $\overline{R}(\overline{v})$)
12 R.insert($\overline{\iota}$)
13 **foreach** $\tilde{r} \in \Theta$ **do**
14 $\iota_{new} \leftarrow$ searchProcedure($\tilde{r}$, R)
15 reassignmentCost $\leftarrow c_R(\iota_{new})$
16 savings $\leftarrow$ savings $-$ reassignmentCost
17 **if** *savings* $\leq$ *0* **then**
18 revertChanges(R)
19 R.insert(ι)
20 **return**
21 R.insert(ι_{new})
22 **return**

normal route state and the fixed route state. The indices of c indicate which route state we want to calculate the costs on. For instance, the route of v can be completely different in both route states. This results in different costs depending on which route of v we want to insert a request. Then, we calculate $c_R(\iota) - c_{\overline{R}}(\overline{\iota})$, the difference between the normal and fixed assignments' costs and store it in *savings*. Therefore, we consider this difference to be cost savings. If this difference is less or equal to 0, we do not even consider a displacing insertion and choose the normal assignment (lines 6-8). Theoretically, a displacing insertion could still result in improvement. However, we choose to ignore this case since it makes sense to use some kind of barrier when considering performing a displacing insertion. Otherwise, in practice, many of the displaced insertions need to be reverted (what that means is explained below), increasing the running time. But if the difference is greater than 0, we consider a displacing insertion.

In this case, we first calculate the cost savings c_{saved} (see section 4.4), which are the costs saved when exchanging the normal route of $\overline{v}$ with the fixed route. These cost savings are added to the overall savings (line 9). Then, we fetch all displaceable requests inside the normal route $R(\overline{v})$. These can be easily retrieved through $\Theta = \bigcup_{i=1}^{k(\overline{v})} P(s_i)$ (line 10). After

retrieving all displaceable requests we can exchange the route $R(\overline{v})$ with the fixed route $\overline{R}(\overline{v})$ (line 11). We exchange the route of $\overline{v}$ since this is the vehicle the request r was assigned to in the search procedure on $\overline{R}$. We insert the fixed assignment $\bar{\iota}$ into the route state R, which now contains the fixed route for $\overline{v}$ (line 12).

Then, we begin to reassign the displaced requests. We search for a new optimal assignment ι_{new} on the normal route state R (line 14). We then calculate the cost of this assignment and subtract it from the total cost savings (lines 15-16). If the total cost savings have fallen below or equal 0, we abort the displacing insertion and revert all changes made (lines 17-20). Reverting means returning the normal route state R to its initial state before exchanging routes and inserting $\bar{\iota}$. We also need to revert insertions of reassigned requests. This reverting procedure is discussed more thoroughly in section 5.4. After reverting all changes, the initial request r is inserted into R with its regular insertion ι (line 19). If the costs of the currently reassigned request do not lower the total cost savings below or equal 0, we apply its new assignment to the route state R (line 21).

This reassigning procedure is repeated until no displaced requests are left or the total cost savings fall below or equal 0, triggering a reversal. Remember that the primary goal of our algorithm is to create better costs for the initial request r. The reassigned requests $\tilde{r} \in \Theta$ can also improve their costs compared to their previous assignment, but that is not necessarily the case. We investigate the quality changes for reassigned requests in section 6.4.

5.4 Implementation Details

In this section, we touch on some implementation details which we did not explain further in the previous section. The first two sections clarify details of exchanging routes and reverting changes. The last section deals with an inaccuracy in the fixed route state caused by the movement of vehicles.

5.4.1 Exchanging Routes

Route states do not store data for each route separately. For every data in the route state, we have a single vector that contains the data for all vehicles. For instance, the route state stores the scheduled arrival times for stops in one vector for all vehicles. If we want to access the scheduled arrival times for a certain vehicle, we need to retrieve its starting and ending index from an index vector. Therfore, when exchanging a normal route $R(v)$ of a vehicle v with its fixed route $\overline{R}(v)$ (line 11), we empty the route $R(v)$ in R. Then, we copy the stops of $\overline{R}(v)$ into $R(v)$ as well as all of the associated data. Now, the route states are identical for the vehicle v. Another part we need to adjust so the search procedure can work adequately is the buckets. We implemented this naively by deleting all source and target buckets of all stops in the previous route $R(v)$. Then, we re-generate all buckets for the new route.

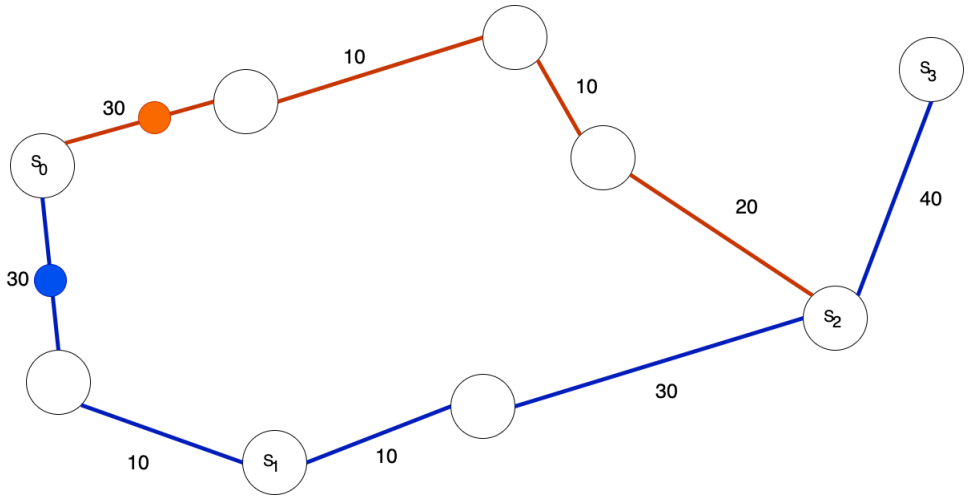


Figure 5.1: The Figure shows an example of an instance that causes inaccuracies. Nodes that are marked with a s_i represent a scheduled stop. The edges are labeled with the time it takes to travel over the edge. The edges marked in blue describe the scheduled normal route. The orange marked edges describe the scheduled fixed route. So stops s_0 and s_2 are the only fixed stops. The orange and blue dots represent the vehicle's position after 20 time units in the fixed and normal routes, respectively.

5.4.2 Reverting Changes

When the displacing insertion turns out not to be beneficial, we revert all changes for the displacing insertion. These changes consist of the route exchange, the insertion of the fixed assignment, and the reassignments of displaced requests, which have been performed until then. We solve this problem by making a backup copy of every route and its associated data before the displacing insertion affects it. When the displacing insertion turns out to be not beneficial, we can exchange the routes of the affected vehicles with their previously copied routes. This is analogous to the route exchange explained in section 5.4.1, with the difference that the exchanged route is not the fixed route but the copy of the route's previous state.

5.4.3 Inaccuracies caused by Vehicle Movement

When we defined vehicles and their routes $R(v) = \langle s_0, s_1, ..., s_k \rangle$ in section 2.1, we set the invariant that the vehicle's current location is s_0 or between s_0 and s_1. The word "between" means some edge in the shortest path from s_0 to s_1. More precisely, the vehicle v is always located at some edge on the shortest path between s_0 and s_1. Remember that in our formulation of fixed routes, the second stop of the fixed route $\overline{R}(v) = \langle s_0, \overline{s}, ..., s_{\overline{k}} \rangle$ does not necessarily coincide with the second stop of the normal route. As a result, $\overline{R}(v)$ and $R(v)$ differ in their shortest paths to their first stop $\overline{s}$ and s_1, which creates two different current locations of v. This means, the current location of v in $\overline{R}(v)$ deviates from the real location of v if $s_1 \neq \overline{s}$. Figure 5.1 illustrates this situation. The normal route (blue) consists

of stops $\langle s_0, s_1, s_2 \rangle$ while the fixed route (orange) consists of $\langle s_0, s_2 \rangle$. The edges are labeled with the time it takes to pass the edge. The blue and orange dots illustrate the vehicle's position after 20 time units in the normal and fixed routes, respectively. The assumption that the fixed route takes the shortest path to its next stop can create unrealistic arrival times for stops inside the fixed route. For instance, in the fixed route, the vehicle reaches stop s_2 after 70 time units. In the normal route, the vehicle arrives at s_2 after 80 time units since it needs to take the detour over s_1. Now, we consider a displacing insertion with an initial request inserted into the fixed route after s_2. We assume that r is issued 30 time units after the vehicle departed from s_0. This means that, in reality, the vehicle is currently at the blue dot. The better costs of r were calculated based on the arrival time at s_2 in the fixed route. However, the vehicle's position now makes it impossible to reach s_2 at the scheduled arrival time of the fixed route.

Some other cases create similar situations. However, they all fall into the problem that the fixed route does not have the same first scheduled stop as the normal route. We thought of an improvement to this problem, which we discuss in the following. Unfortunately, the Re-KaRRi version in the evaluation chapter 6 does not contain this improvement. However, we provide an error assessment in section 6.6, which evaluates these inaccuracies.

Phantom Stop. We considered adding a *phantom stop* φ after s_0 into the fixed route to eliminate the error explained in this section. This phantom stop is simply the first stop of the normal route (if it has one). When including a phantom stop, we adjust the consecutive fixed stops' arrival (and departure) times. There are two different approaches when using phantom stops, and each can have its benefits.

In the first approach, we consider φ a fixed stop. This would also include the dropoffs d of customers picked up at the first stop, reducing the difference between fixed and normal routes. Pickups of customers picked up at stop d are not considered fixed stops. However, when exchanging a normal route with a fixed route, the arrival times at all fixed stops would still be correct. With this approach, the fixed route could be exchanged with the normal route without causing any inaccuracies. This approach's downside is the reduced reoptimization potential since the fixed routes are closer to the normal routes.

In the second approach, we do not consider φ a fixed stop. Any time this route is exchanged with the associated normal route, we remove the phantom stop and adjust the arrival times for all fixed stops based on the vehicle's current position. Then, the arrival times for all consecutive fixed stops are not earlier than realistically possible. This is because when calculating the arrival times of the fixed stops, we assume that the vehicle takes the detour over φ. However, when the route is exchanged, we can take the shortest path from the vehicles' current position to the next fixed stop, which takes at most as long as taking the detour over φ. With this approach, the error goes in the opposite direction since the scheduled arrival times are later than theoretically possible. When the fixed route is exchanged, we remove φ so the vehicle can immediately drive from its current position to its next scheduled fixed stop. However, the scheduled arrival times of the consecutive fixed stops were calculated based on the assumed detour over φ. So, the vehicle can arrive earlier at the stops than the initially calculated arrival times. This can lead to vehicles waiting

at the stop for customers that are picked up at this stop. With this approach, we shift the increased waiting time at a stop from customers to vehicles, which could be considered tolerable. The main advantage of this approach is that the difference between fixed and normal routes is greater, allowing a higher degree of reoptimization.

6 Evaluation

This section evaluates our reoptimization algorithm, Re-KaRRi, for the dynamic taxi sharing problem. First, we give an overview of our setup and which instances we used to evaluate our algorithm. In the second section, we compare Re-KaRRi to KaRRi regarding running time. Then, in the third section, we examine the potential of the approach we used, fixed routes. In the fourth section, we examine the process of displacing insertions in detail. We assess how reassigned requests and passengers in a vehicle are affected by a displacing insertion regarding quality aspects like trip and wait time. We also examine how much cost Re-KaRRi saves when performing a displacing insertion. In the fifth section, we evaluate the effects of our reoptimization approach on the overall quality of the taxi sharing dispatcher. In the last section, we do an error assessment for inaccuracies caused by vehicle movements (see section 5.4.3).

6.1 Setup and Instances

Our source code is written in C++17 and compiled with GCC 9.4. We conducted our experiments on a Ubuntu 20.04 machine with 512 GiB of memory and four Intel Xeon E5-4640 processors at 2.4GHz. We utilize only one of the processors since our code is executed sequentially.

We evaluated our algorithm on the *Berlin-1pct* and *Berlin-10pct* request set used by Buchhold et al. [3]. These instances represent 1% and 10% of taxi sharing demand in Berlin during weekdays. The requests for these instances were generated based on the Open Berlin Scenario [10] for the MATSim transport simulation [6]. The road networks were obtained from OpenStreetMap [9]. The vehicle road network considers the speed limits for each road to determine the travel time for the corresponding edge in the road network graphs. For the pedestrian network, we assume a walking speed of $4.5 km/h$. Table 6.1 shows the number of nodes (intersections), edges (roads), vehicles, and requests for the two instances.

Table 6.1: Instances used in experiments

Instance	$\|V\|$	$\|E\|$	$\#veh.$	$\#req.$
Berlin-1pct	73689	159039	1000	16569
Berlin-10pct	73689	159039	10000	149185

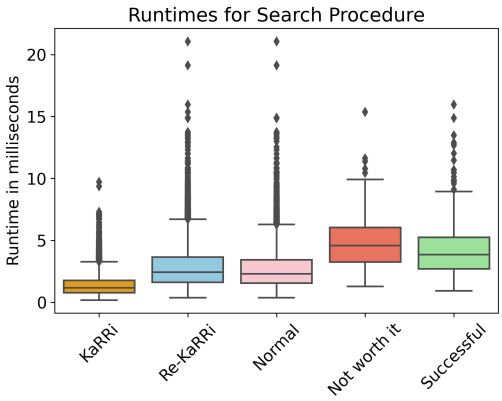


Figure 6.1: Run times for Berlin-1pct (normal fleet size)

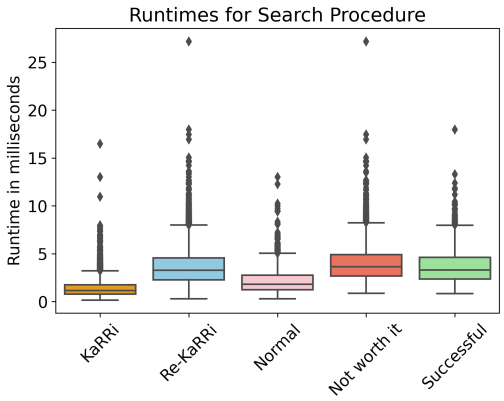


Figure 6.2: Run times for Berlin-1pct (25% fleet size)

In some experiments, we use variations of the two input instances, decreasing the fleet size to only 25% or 50% of the original fleet size. We chose to do this since we observed that scheduled routes for the regular fleet size instances were short. This resulted in a smaller difference between fixed and normal routes, limiting the reoptimization potential for our approach.

We chose the same model parameter configuration used by Laupichler and Sanders [9]. We chose $t^{max}_{wait} = 600s$, $t^{min}_{stop} = 60s$, $\gamma_{wait} = 1$, $\gamma_{trip} = 10$, $\tau = 1$, $\omega = 0$ and $\rho = 300s$.

6.2 Running Time

In this section, we evaluate the performance of Re-KaRRi in terms of run time. Our algorithm described in chapter 5 can result in three outcomes. The first is that the fixed assignment cost exceeds the normal assignment cost for our initial request. In this case ("normal"), we choose the normal assignment and do not consider a displacing insertion. The second case ("not worth it") is a displacing insertion that is not cost-beneficial. The third case ("successful") occurs when the displacing insertion is beneficial. When we evaluated the run times for our algorithm, we measured all three of these cases separately. This allows us to analyze the performance of our algorithm in general and our reverting and reassigning procedure in particular.

Figures 6.1 and 6.2 display the time it takes KaRRi and Re-KaRRi to compute an assignment for an issued request. Both Figures display the results for the same instance Berlin-1pct, with the difference that Figure 6.1 uses the regular fleet size and Figure 6.2 25% of the regular fleet size. The results for the Berlin-10pct instance with the same fleet size variations are displayed in Figures 6.3 and 6.4. The runtimes are given in milliseconds. The runtimes of Re-KaRRi are further split up into the run times of each of the three cases we mentioned earlier in this section.

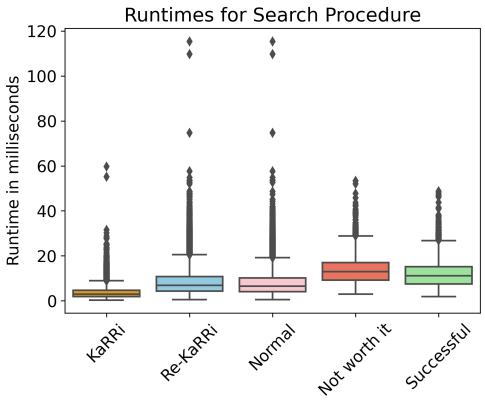


Figure 6.3: Run times for Berlin-10pct (normal fleet size)

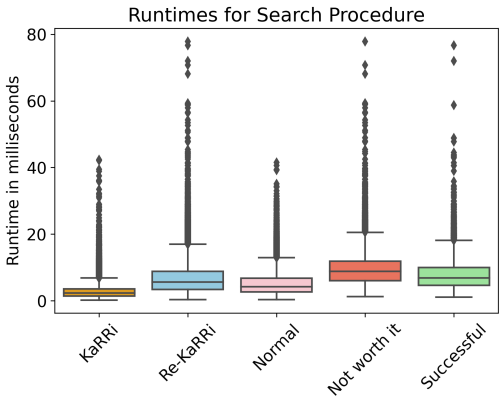


Figure 6.4: Run times for Berlin-10pct (25% fleet size)

For the Berlin-1pct instance, which uses the entire fleet, KaRRi has a median runtime of 1.16*ms* and Re-KaRRi 2.43*ms*, an increase of 109.48%. This is expected since Re-KaRRi performs at least two search procedures, while KaRRi only performs one. The median runtime of the "normal" case, where no displacing insertion is considered, performs two search procedures and has twice the median runtime of KaRRi. The other two cases, "not worth it" and "successful," have significantly higher median runtimes: 4.57*ms* and 3.85*ms*, respectively. This explains the comparatively low general runtime of Re-KaRRi since 91% of the time, no displacing insertion is considered, and the normal insertion is performed. The other two cases, "not worth it" and "successful," which are more costly regarding runtime, occur only 2.3% and 6.7% of the time, respectively.

When we use only 25% of the fleet in Berlin-1pct, the differences in running times to KaRRi become more significant. For this instance, Re-KaRRi has an increased running time of 178.63% compared to KaRRi. This is because Re-KaRRi becomes more inefficient when performing displacing insertions. When Re-KaRRi receives a request, 60.1% of the time, it tries to perform a displacing insertion, which is not beneficial. This increases the algorithm's general running time since a displacing insertion that needs to be reverted has a median running time of 3.66*ms*, more than three times as long as KaRRi's search procedure with 1.17*ms*. A successful displacing insertion ("successful") takes 3.29*ms*.

For the Berlin-10pct instance, the runtime differences are manifested even more. In the regular fleet instance Re-KaRRi increases KaRRi's runtime by 135.15%, from 2.93*ms* to 6.89*ms*. The reason is that Re-KaRRi performs more inefficiently with its displacing insertions. Around half the time, a displacing insertion is performed, it is not beneficial.

The substantial variations in runtime between the three cases ("normal", "not worth it" and "successful") result in a high variance in the general runtime of Re-KaRRi, as seen in all the provided Figures. The Re-KaRRi algorithm performs the reverting procedure naive (see section 5.4), a part of the algorithm that needs improvement. This could bring the Re-KaRRi algorithm closer to KaRRi in terms of runtime.

Table 6.2: Difference in number of stops between normal and fixed routes. The columns represent the different instances and the used fleet size. The left row displays the number of stops the normal and fixed routes differ. An entry represents the percentage of fixed routes that differ from their normal routes by the number associated with the row. If an entry is "-", no cases of this stop difference occur. If an entry is 0.00, cases of this stop difference occur but are not visible due to rounding.

	Berlin-1pct			Berlin-10pct		
	25%	50%	100%	25%	50%	100%
1	1.19	0.35	0.15	2.11	1.27	0.98
2	42.85	11.27	4.40	17.16	9.55	6.53
3	1.00	0.05	0.03	0.76	0.26	0.16
4	3.61	0.12	0.02	0.60	0.18	0.11
5	0.03	-	-	0.02	0.00	0.00
6	0.01	-	-	0.00	0.00	0.00

6.3 Reoptimization Potential

In this section, we do not analyze our algorithm directly, but we evaluate the general concept of fixed routes. We assess to which degree the routes of the route state are modifiable with fixed routes. We do this by evaluating the difference in the number of stops between normal routes and their fixed routes. Table 6.2 shows the difference in the number of stops for the fixed and normal routes. We displayed the difference in stops for the smaller Berlin-1pct and the bigger Berlin-10pct instances. We further split up the results according to the fleet size used. The left row displays the number of stops the normal and fixed routes differ. An entry represents the percentage of fixed routes that differ from their normal routes by the number associated with the row. For example, the second entry in the 25% column of Berlin-1pct says that for this instance, 42.85% of the fixed routes have two stops less than the normal route. In the table, "-" indicates no occurrences, while 0.00 signifies that occurrences are present but are not visible due to rounding. The numbers in the column do not sum up to 100% since we exclude the stop difference 0, where the fixed route and normal route are identical.

Table 6.2 shows that no matter how small we choose the fleet size, the difference in the number of stops rarely exceeds four stops. We see a spike at the two stop difference, and only few fixed routes exceed this difference. The reason for this is the way requests are inserted into routes. A non-negligible part of requests seems to be inserted at the end of the routes. The vehicle drives from its current location to its next stop, picks up a customer, drives to their dropoff, and then to the next pickup. This results in fixed routes only consisting of one stop. This suspicion is backed by Table 6.3, which displays the average number of stops for fixed and normal routes. Even for the small fleet size instances, the average number of scheduled stops does not exceed three. Tables 6.3 and 6.2 show that fixed and normal routes do not differ much. This low difference between fixed and normal routes in the regular fleet size instances explains the low rate of displacing insertions in the previous

Table 6.3: Average number of scheduled stops per route. The columns represent the different instances and the used fleet size. The left row describes the different routes (normal and fixed). An entry represents the average number of scheduled stops per route.

	Berlin-1pct		Berlin-10pct	
	25%	100%	25%	100%
normal	2.83	1.68	2.47	1.94
fixed	1.78	1.59	2.06	1.79

Table 6.4: The number of reassigned requests in successful displacing insertion. The columns represent the different instances and the used fleet size. The left column displays the number of requests reassigned due to the displacing insertion. Each entry indicates the occurrence of this specific number of reassignments as a percentage.

	Berlin-1pct			Berlin-10pct		
	25%	50%	100%	25%	50%	100%
1	89.40	99.17	99.09	94.54	94.77	95.22
2	10.58	0.83	0.91	5.24	4.98	4.49
3	0.02	-	-	0.20	0.24	0.27
4	-	-	-	0.02	0.01	0.03

section. These results raise the question of whether fixed routes provide enough potential for reoptimization.

6.4 Displacing Insertions

In this section, we evaluate the results of displacing insertions, which were performed successfully. More precisely, we mean how passengers and reassigned requests are affected by a displacing insertion in terms of quality. We also discuss the costs saved when a displacing insertion is performed. For this section, we focus more on the two instances that use only 25% of their fleet since displacing insertions are performed more often in these instances.

First, we assess the general statistics of a displacing insertion, i.e., how many requests are reassigned during a displacing insertion and how often reassignments occur overall. Table 6.4 displays how many requests are reassigned during a successful displacing insertion. The left column displays the number of requests that are reassigned. Again, the columns represent the used fleet size in the used instance, and the entries present the relative occurrence in %. For instance, the first entry in the Berlin-1pct 25% column says that 89.4% of the displacing insertions performed one reassignment. The results show that most displacing insertions do not reassign more than two requests. This raises the question of whether reassigning too many requests can decrease the likelihood of a displacing insertion

Table 6.5: Number of reassignments for a single request throughout its existence. The columns represent the different instances and the used fleet size. The left column displays the number of times a request was reassigned during its existence. Each entry indicates the occurrence of this specific number of reassignments as a percentage.

	Berlin-1pct			Berlin-10pct		
	25%	50%	100%	25%	50%	100%
1	15.46	9.67	5.52	8.94	6.44	4.29
2	3.13	1.40	0.55	1.15	0.71	0.32
3	0.70	0.20	0.04	0.15	0.07	0.02
4	0.15	-	-	0	0	-
5	0.04	-	-	-	-	-

being successful and save cost overall. Recall that in a displacing insertion, we perform reassignments as long as the total costs do not fall below zero. If this case occurs, we revert all changes that were performed until then. We evaluated the median of the number of reassigned requests during a displacing insertion before it turns nonbeneficial. Surprisingly, this median turns out to be 0 in all instances, meaning that the first reassigned request already broke the cost savings barrier. This also implies that the displacing insertion does not need to perform many changes when reverting the changes made until then.

Before evaluating the changes in quality aspects, we assess how common reassigned requests are. Table 6.5 displays the number of times a single request is reassigned during its existence. In instances with smaller fleets, we can see many requests are reassigned at least once during their existence. However, not many of these are reassigned more than two times. This can be explained with the soft constraint violation penalties (see section 2.3), which would increase since multiple reassignments of a request would delay the departure time of a request. This would result in higher wait and trip times, triggering soft constraint violation penalties.

The Berlin-1pct instance that uses only 25% of its fleet reassignes more than 19% of the requests. An interesting fact about this instance is that 23% of these requests are reassigned to their previous vehicle (in other instances, it is even less). This could mean that the new route state created a better situation for the remaining requests, or the previous route has been altered too much by the displacing insertion to be considered for reassignment.

Figures 6.5 and 6.6 display the changes in scheduled trip time and the cost increase for reassigned requests (in a successful displacing insertion) in the Berlin-1pct instance that uses only 25% of its fleet. The figures are cumulative graphs that display the cumulative share of requests that achieve a certain increase in scheduled trip time and cost (displayed on the x-axis). The x-axis regards the reassignment's trip and cost increase compared to the previous assignment in percentage. The y-axis displays the cumulative share of requests that reach this level of increase or lower. If we choose a point x on the x-axis, the corresponding value y on the y-axis shows the share of requests for which the respective increase is at most x. For instance, in Figure 6.5, we can see that for more than 20% of the reassigned requests,

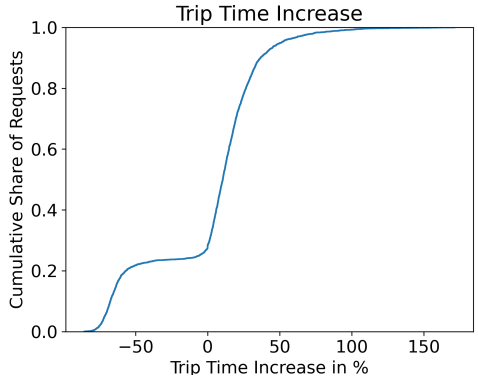


Figure 6.5: Trip Time Increase for Reassigned Requests

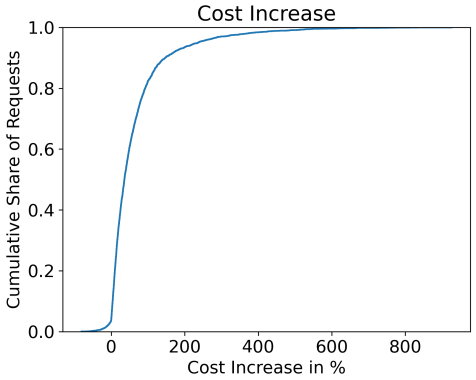


Figure 6.6: Cost Increases for Reassigned Requests

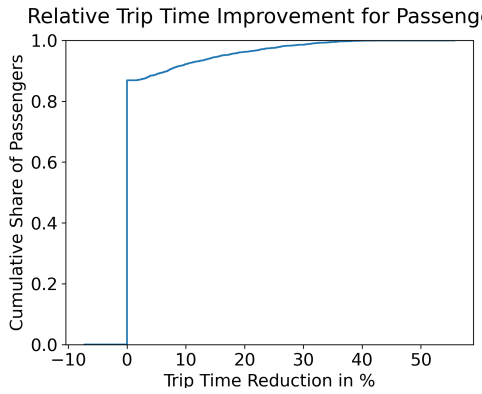


Figure 6.7: Trip Time Reduction of Passengers in Fixed Route compared to Normal Route (25% fleet size)

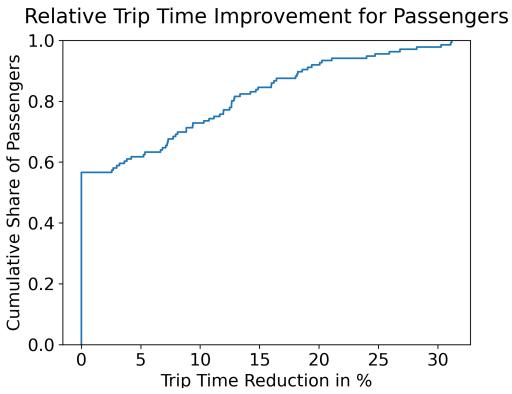


Figure 6.8: Trip Time Reduction of Passengers in Fixed Route compared to Normal Route (regular fleet size)

their trip time is improved by 50% compared to their previous assignment. However, for almost 80% of the reassigned requests, their trip time gets worse. But for almost 50% of the reassignments, whose trip time increases, the increase is not more than 25%. This is reflected in the cost increase in Figure 6.6. Even though more than 20% of reassignments achieve a better-scheduled trip time, the cost increase does not reflect that. This can be explained by the increased wait time. Although some requests achieve a better trip time, their wait time increases, breaking the soft constraint violation barrier. Looking at the figures, we see a small number of reassignments that seem to increase in scheduled trip time and cost dramatically. We could employ reassignment checks to prohibit such extreme changes.

Figure 6.7 and 6.8 display the reduction in trip time for passengers in a vehicle when the normal route is exchanged with the fixed route. The figures are again cumulative graphs that display the cumulative share of passengers that reduce their trip time by a maximum percentage. Both figures are the results of the Berlin-1pct instance with different fleet sizes. Since the routes usually are not very long (see section 6.3), the trip times for passengers

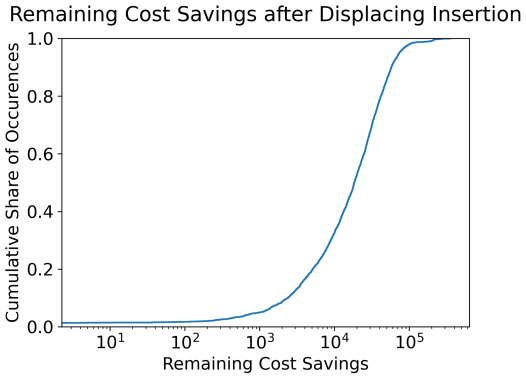


Figure 6.9: Cumulative Remaining Cost Savings (Berlin-1pct with 25% fleet size)

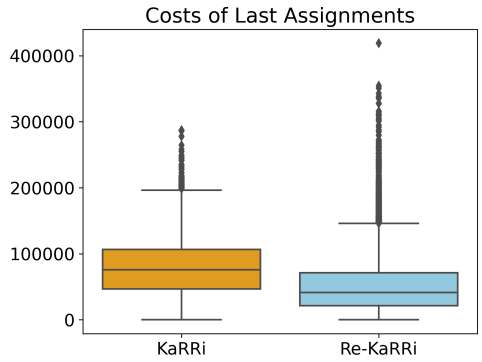


Figure 6.10: Average Cost of Last Assignment (Berlin-1pct with 25% fleet size)

do not decrease by much. It is surprising that the scheduled trip time for passengers is reduced when 100% of the fleet is used. The reduction is higher than for the instance that uses only 25% of the fleet. This is surprising because the routes are longer in the instances that use a low percentage of their fleet. For instance, in the regular fleet instance, 25% of passengers receive a trip time reduction of 10% or higher, while for passengers in the small fleet instance, less than 10% receive this level of reduction. This indicates that the routes become more efficient in small fleet instances, where displacing insertions are performed more often. By efficient, we mean that requests that generally travel in the same direction are assigned to the same vehicle, resulting in minor detours for consecutive stops. We assess this possibility by evaluating the global quality aspects in section 6.5.

Lastly, we evaluate the remaining cost savings from a displacing insertion. These remaining cost savings include the cost savings for the initial request for which the displacing insertion is performed and the difference in cost savings for passengers and new costs for reassigned requests. Figure 6.9 shows a graph plotting the cumulative cost savings. We used a logarithmically scaled x-axis to better display the distribution of cost changes. For instance, for half the displacing insertions, we receive remaining cost savings greater than $2 \cdot 10^4$, and for the other remaining half, less than that. To put this into perspective, Figure 6.10 displays the average cost of the last assignment a request receives. For KaRRi, that is simply the assignment a request receives, and for Re-KaRRi, that is the cost of the last reassignment of a request (if it has been reassigned). KaRRi's mean assignment cost is 75788, while Re-KaRRi's is 41420, 45.35% less.

6.5 Global Quality Aspects

In this section, we will evaluate the *global quality aspects* of the dynamic taxi sharing problem and some other interesting aspects of this problem. With the cost function provided by Laupichler and Sanders [9] and explained in section 2.3, we try to minimize the global overall wait, trip, and walk times of customers and vehicle operation time. We will compare

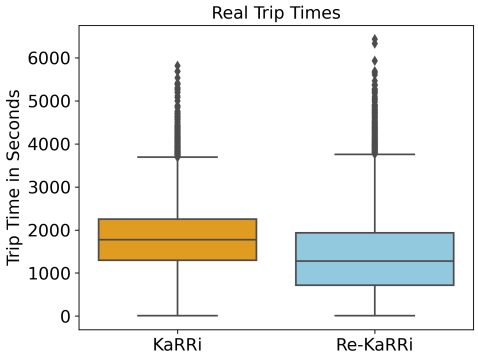


Figure 6.11: Real Trip Time (Berlin-1pct with 25% fleet size)

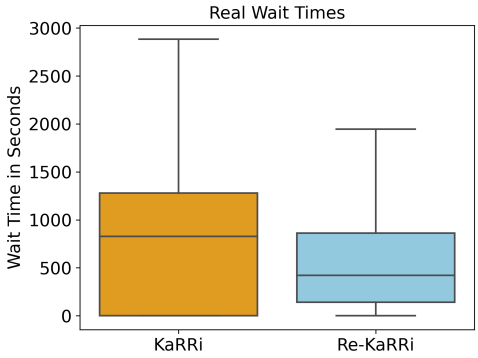


Figure 6.12: Real Wait Time (Berlin-1pct with 25% fleet size)

these aspects by assessing the real trip, wait, and walk times for requests in KaRRi and Re-KaRRi. The *real* trip, wait, and walk times are not the same as the *scheduled* times, which we looked at in the previous section. The scheduled trip and wait times are based on the route's current state when a request is inserted. After insertion, other requests can be assigned to the same vehicle, changing the trip and wait time. We used box plots without outliers for some of the plots to visualize these differences between KaRRi and Re-KaRRi. We removed the outliers in some plots since there were very few outliers with extreme values, which would have impacted the plots' legibility.

Unfortunately, the instances with regular fleet size do not differ much, although Re-KaRRi results in slightly better trip, wait, and walk times (0% — 1.8% each). The difference in small fleet instances is illustrated in Figure 6.11 and 6.12 and is much more pronounced. Figure 6.11 depicts the real trip time, i.e., the time it takes for a request to reach its destination after issuing the request. KaRRi achieved a mean trip time of 1774*s* while Re-KaRRi achieved 1279.4*s*, 27.88% less (or 8.24 minutes less). Although the mean of Re-KaRRi seems to be better, the whiskers of the box plot suggest that the trip times for some of the requests cannot be improved. Figure 6.12 depicts the real wait times, and here we can see that Re-KaRRi achieved a remarkable difference. The mean wait time in KaRRi is 827.6*s*, while Re-KaRRi achieves a mean wait time of 419.8*s*, an improvement of 49.28%. These decreased trip and wait times suggest that Re-KaRRi can reoptimize routes to make them more efficient. Another quality is the walking time. Figure 6.13 illustrates these differences in walking times. Although the mean of Re-KaRRi does not differ significantly from KaRRi (4.9% reduction), the variance in walk time of Re-KaRRi seems to be significantly smaller. This is comfortable for the customer since pickups and dropoffs seem to be chosen near their origin and destination for most requests.

The driving time of vehicles is an especially important quality aspect since it is an indicator of operating costs and CO_2 emissions. We measured the total driving time and compared KaRRi and Re-KaRRi. Table 6.6 displays the total driving time of all vehicles in KaRRi and Re-KaRRi for the different instances we use. The last row shows the reduction of total trip time by Re-KaRRi in %. Although we do not see much improvement for regular

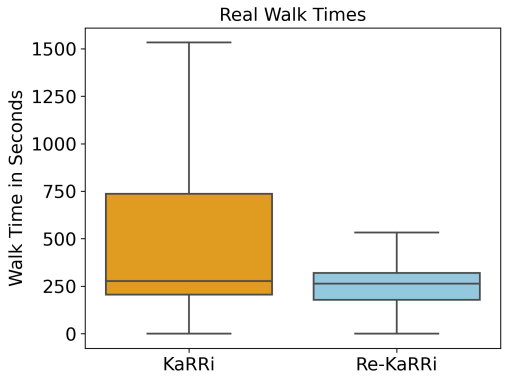


Figure 6.13: Real Walk Time (Berlin-1pct with 25% fleet size)

Table 6.6: Total driving time of vehicles in seconds. The columns represent the different instances and the used fleet size. The left column displays the algorithm used (KaRRi and Re-KaRRi). Each entry indicates the total time the vehicles spend driving.

	Berlin-1pct			Berlin-10pct		
	25%	50%	100%	25%	50%	100%
KaRRi	13138220	12583536	12477736	79053686	81102590	82718523
Re-KaRRi	12432158	12531285	12471593	77563703	81179748	82601735
Reduction	5.37%	0.42%	0.05%	1.88%	-0.10%	0.14%

and half-sized instances, we can see improvement for the 25% fleet size instances. These reductions do not seem substantial but make a non-negligible difference for operating costs and CO_2 emissions when considering scaling the system.

These results seem promising. However, for the bigger Berlin-10pct instance, the results are not as promising. In the Berlin-10pct instance, which utilizes 25% of its fleet, the median real trip time of Re-KaRRi is only 3.18% smaller compared to KaRRi. The real wait time is only decreased by 6.93%. The real walk time is decreased by 5.94%, but the variance of Re-KaRRi and KaRRi become similar. Although this is still an improvement, Re-KaRRi does not perform equally well in different instances. One explanation for these different results on the Berlin-10pct instance can be the bigger fleet used in this instance. The fleet in Berlin-10pct is ten times the size of the fleet in the Berlin-1pct instance. This can reduce the general potential for reoptimization since requests are initially assigned to well-matching vehicles.

Another quality aspect we can evaluate for the dynamic taxi sharing problem is the average occupancy of vehicles. Table 6.7 displays the average occupancy of vehicles for KaRRi and Re-KaRRi. Again, we can see an improvement for the smaller fleet instance of Berlin-1pct but no great improvement for the bigger Berlin-10pct instance. This underlines that Re-KaRRi

Table 6.7: The average occupancy of vehicles. The columns represent the different instances and the used fleet size. The left column displays the different algorithms used (KaRRi and Re-KaRRi). Each entry indicates the average number of occupancy of a vehicle.

	Berlin-1pct			Berlin-10pct		
	25%	50%	100%	25%	50%	100%
KaRRi	0.66	0.70	0.60	1.08	0.89	0.81
Re-KaRRi	0.78	0.69	0.60	1.08	0.89	0.82

has the potential to improve global quality aspects by displacing insertions. However, it tends to perform better on instances that are utilized to their maximum capacity.

6.6 Error Assessment

This section will assess the inaccuracies caused by the vehicle movement (see section 5.4.3). We assess the error for both instances, Berlin-1pct and Berlin-10pct. The errors in both instances are very similar, so we focus on the Berlin10-pct instance since it produced smoother plots. Inaccuracies do not differ much between fleet configurations; for small fleet instances, the outliers just become stronger. Therefore, we chose to analyze the case of Berlin-10pct, which uses the regular fleet.

When we assess the error, we differentiate between two cases. The first case treats fixed routes that have more than one stop. This is the case we described in section 5.4.3. In section 6.3, we discovered that fixed routes often differ from their corresponding normal route by two stops. This observation, in combination with the low average number of stops in the normal and fixed route, led to the suspicion that many of the inserted requests are inserted at the end of a route. This would result in the fixed route only having one stop while the vehicle drives to the next pickup. In this case, the last and only stop in the fixed route would be the assumed position of the vehicle, i.e., the vehicle is falsely assumed to be idling. So, we evaluated a second case where we looked at fixed routes consisting of only one stop. Our suspicion turns out to be true for the Berlin-1pct instance. The second case is twice as common when the entire fleet is used and even ten times more common when only 25% of the fleet is used. We evaluated these inaccuracies when Re-KaRRi considered a displacing insertion, i.e., performed a route exchange. This is why, when the fixed route has only one stop, we can guarantee that the normal route has more than one. Otherwise, Re-KaRRi would have chosen the normal route for the assignment. In the Berlin-10pct instance the ratio between these two cases is not as radical as in Berlin-1pct but for the instance that uses 25% of the fleet, the second case is still 19% more common. For the Berlin-10pct regular fleet instance, both cases occur similarly.

Figure 6.14 and 6.15 illustrate the delayed arrival at the next scheduled stop if a fixed route is exchanged. The Figures are again cumulative since we experienced the problem of strong outliers. In Figure 6.14, we measured the delayed arrival for the case that the fixed route has

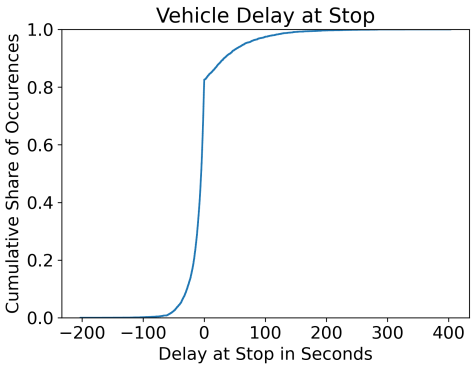


Figure 6.14: Inaccuracy in Arrival Time (First Case)

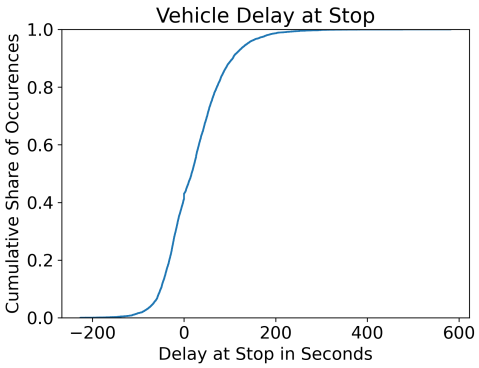


Figure 6.15: Inaccuracy in Arrival Time (Second Case)

more than one stop. We can see that more than 90% of the time, the delayed arrival time lies between $-100s$ and $100s$. This means a customer who arrives at his stop rarely needs to wait more than an additional $100s$ for their vehicle. That would mean a 23.82% increase in wait time if we assume the average wait time. We can also see that in some cases, the arrival time can even be earlier. We measured these values after the initial displacing request was inserted into the fixed route. The new request can be inserted between s_0 and s_1 as a new stop in this case. If this occurs, the vehicle's actual position can be nearer the new stop than the assumed position in the fixed route. In Figure 6.15, we measured the delayed arrival for the case that the fixed route has only one stop. In this case, we can see that the delay can become more significant. In this case, more than 90% of the time, the delayed arrival time lies between $-200s$ and $200s$. Considering a delay of $200s$ would mean an increase of 47.64% when assuming the average wait time. So, in general, we measure an inaccuracy, which can affect the wait times significantly. However, this inaccuracy translates to 1-3 minutes, which could be ruled tolerable.

7 Conclusion and Future Work

In this thesis, we presented Re-KaRRi, a reoptimization extension to the KaRRi algorithm. We discussed the general reoptimization concept of fixed routes for the problem of taxi sharing with meeting points. We show how this concept can be utilized in reoptimization algorithms by presenting Re-KaRRi. We tested Re-KaRRi on two instances with varying fleet sizes, where we recorded a presentable amount of improvement for instances with small fleet sizes. This makes Re-KaRRi interesting for rush hours when the available fleet size becomes overwhelmed with the number of incoming requests. Due to KaRRi's efficient and fast performance, the runtimes of Re-KaRRi are still within the range of being applicable in the real world. In our evaluation section 6 we discovered that the concept of fixed routes does not provide much potential for reoptimization. However, there are no other possibilities for reoptimization of this problem other than reassigning requests. As we can see in section 1.1, most approaches limit reassignments for requests that have entered their vehicle. So, the only possibility to explore further reoptimization options would be to allow switching vehicles mid-travel. Apart from that, the improvements in small fleet instances are noticeable but not very significant. We thought of extensions to Re-KaRRi, which could drive the improvements even higher.

Our algorithm performs displacing insertions where we consider reassigning requests of a single vehicle. The displaced requests for this vehicle are reassigned to the normal route state. Most other approaches (see section 1.1) consider reassigning multiple requests over a batch of vehicles. We could extend our algorithm by considering reassigning displaced requests to the fixed route state as well, i.e., performing recursive displacing insertions. This could result in a higher level of reoptimization since we displace not only requests of one route but requests of multiple routes within a general area. For instance, if we reassign a request and consider fixed routes as well, it will choose a vehicle whose fixed route goes through its general area. This vehicle's displaced requests are likely in the general area of the previous displacing request and perform a displacing insertion as well. Thereby, we rebuild vehicles' routes in a general area. The question is how large these general areas are and if the recursion converges. We could consider requests that have been reassigned as fixed. This way, following displacing insertions can not displace previously reassigned requests, and the algorithm would converge.

Another approach that could be incorporated into Re-KaRRi would be the adjustment of stops that only consist of dropoffs. This way, routes could be slightly adjusted to newly issued requests to reduce costs. Route adjustments would be limited by the maximum walking radius of customers who are dropped off at stops. This approach could also consider stops that have pickups, but this could be uncomfortable for customers who are picked up at these stops.

Bibliography

[1] Niels Agatz, Alan L. Erera, Martin W.P. Savelsbergh, and Xing Wang. Dynamic ride-sharing: a simulation study in metro atlanta. *Procedia - Social and Behavioral Sciences*, 17:532–550, 2011. ISSN 1877-0428. doi: https://doi.org/10.1016/j.sbspro.2011.04.530. URL https://www.sciencedirect.com/science/article/pii/S1877042811010895. Papers selected for the 19th International Symposium on Transportation and Traffic Theory.

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

[3] Valentin Buchhold, Peter Sanders, and Dorothea Wagner. *Fast, Exact and Scalable Dynamic Ridesharing*, pages 98–112. SIAM, 2021. doi: 10.1137/1.9781611976472.8. URL https://epubs.siam.org/doi/abs/10.1137/1.9781611976472.8.

[4] Andres Fielbaum, Xiaoshan Bai, and Javier Alonso-Mora. On-demand ridesharing with optimized pick-up and drop-off walking locations. *Transportation research part C: emerging technologies*, 126:103061, 2021.

[5] Robert Geisberger, Peter Sanders, Dominik Schultes, and Christian Vetter. Exact routing in large road networks using contraction hierarchies. *Transportation Science*, pages 297–438, 2012.

[6] Andreas Horni, Kai Nagel, and Kay Axhausen, editors. *Multi-Agent Transport Simulation MATSim*. Ubiquity Press, London, Aug 2016. ISBN 978-1-909188-75-4, 978-1-909188-76-1, 978-1-909188-77-8, 978-1-909188-78-5. doi: 10.5334/baw.

[7] Jaeyoung Jung, R. Jayakrishnan, and Ji Young Park. Dynamic shared-taxi dispatch algorithm with hybrid-simulated annealing. *Computer-Aided Civil and Infrastructure Engineering*, 31(4):275–291, 2016. doi: https://doi.org/10.1111/mice.12157. URL https://onlinelibrary.wiley.com/doi/abs/10.1111/mice.12157.

[8] Sebastian Knopp, Peter Sanders, Dominik Schultes, Frank Schulz, and Dorothea Wagner. *Computing Many-to-Many Shortest Paths Using Highway Hierarchies*, pages 36–45. SIAM Workshop on Algorithm Engineering and Experiments (ALENEX), 2007. doi: 10.1137/1.9781611972870.4. URL https://epubs.siam.org/doi/abs/10.1137/1.9781611972870.4.

[9] Moritz Laupichler and Peter Sanders. Fast many-to-many routing for ridesharing with multiple pickup and dropoff locations, 2023.

[10] 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. *Procedia Computer Science*, 151:870–877, 2019. ISSN 1877-0509. doi: https://doi.org/10.1016/j.procs.2019.04.120. URL `https://www.sciencedirect.com/science/article/pii/S1877050919305848`. The 10th International Conference on Ambient Systems, Networks and Technologies (ANT 2019) / The 2nd International Conference on Emerging Data and Industry 4.0 (EDI40 2019) / Affiliated Workshops.