Algorithm Engineering for Software-Defined Public Transportation

PTaxi: Combining Taxi-sharing and Public Transit

Ha Linh Nguyen

Karlsruhe Institute of Technology, Kaiserstraße 12, 76131 Karlsruhe
uipqj@student.kit.edu
https://ae.iti.kit.edu/

Abstract. Mobility plays a pivotal role in our daily lives, enabling connectivity with the world. Traffic problems have long been a topic of discussion and remain relevant to this day. One critical issue is traffic congestion, due to the high number of private vehicles, especially during rush hours. To address this issue, our research focuses on two key solutions: taxi-sharing and public transit. Taxi-sharing, where multiple passengers with different origins and destinations share a taxi, offers the flexibility of on-demand rides while optimizing the number of vehicles on the road. Public transit, on the other hand, leverages well-built infrastructure, allowing passengers to travel from one station to another on dedicated lanes. With recent technological advances, such as real-time data processing and advanced routing algorithms, it is possible to incorporate multiple modalities of transport within a single trip. Our research targets the multi-modal transportation problem by combining taxi-sharing and public transit to provide users with flexibility while utilizing the static public transit network. We focus on a particularly applicable scenario, where taxi-sharing is employed at the beginning and end of a public transit journey, connecting users from their exact locations to public transit stations and from public transit stations to their desired destination. To accomplish this, we introduce *PTaxi*, an algorithm that combines existing state-of-the-art algorithms for taxi-sharing and public transit into a single framework. Although several studies have aimed to achieve this combined mode of transport, most do not calculate exact routes or offer real-time responses. Our project fills this gap by designing, developing, and evaluating the PTaxi algorithm to find an optimal multi-modal route.

Keywords: Mobility as a Service · Taxi-sharing · Public Transit · Algorithm Engineering.

1 Introduction

Traffic congestion is a prevalent issue in urban areas, significantly impacting daily commutes and the overall quality of life [10]. The high volume of private vehicles

on the roads, particularly during rush hours, worsens this problem, leading to increased travel times, pollution, and stress for commuters [19]. Traditional taxis, while offering the convenience of door-to-door service, are often associated with high costs, making it less attractive for frequent use. Taxi-sharing offers a more sustainable alternative, in which multiple passengers share a vehicle for travel, optimizing the vehicle's occupancy while potentially compromising the passengers' comfort. In contrast, public transit, despite being more affordable, suffers from issues of inconvenience, such as fixed routes and schedules, overcrowding, and limited coverage in certain areas.

Mobility as a Service (MaaS) is emerging as a significant trend, gaining interest as a potential solution to practical transportation challenges. MaaS aims to provide a seamless and convenient mobility experience by establishing an integrated user interface for multiple modes of transportation [8]. Technological advancements have enabled the development of smartphone applications that allow access to diverse transportation options. However, these services often remain fragmented, requiring separate applications and payment processes for each transport mode. While some existing applications attempt to integrate different modalities, such as public transit with e-scooters or car-sharing, they typically redirect users to the providers' sites for the actual purchase process. MaaS seeks to unify all transport modes, both public and private, into a single platform, allowing users to plan, reserve, and pay through a single point of service [18]. This integration promises smooth, intermodal transitions, offering a fully digital, clean, and sustainable transport solution.

In this context, an efficient and exact algorithm is crucial for ensuring that MaaS applications function effectively. Our research plans to solve the multimodal transportation problem for two particular modes, contributing to the advancement of MaaS by addressing specific challenges related to integrating taxi-sharing and public transit. Our focus lies in combining the existing dynamic taxi-sharing algorithm – KaRRi [14] with the public transit framework with unlimited walking – ULTRA [1].

We provide a comprehensive overview of our paper, as follows: In Section 2, we delve into the problem statement, providing formal definitions and notations essential for understanding taxi-sharing and public transit algorithms. Based on this groundwork, Section 3 introduces the required background information, explaining fundamental concepts of the two employed algorithms in our work, KaRRi and ULTRA. Section 4 offers a review of related literature, exploring existing methodologies that integrate multiple modalities, while evaluating their positives and negatives. The main contribution of our research can be found in Section 5, where we describe the high-level algorithm. Here, we discuss the overall structure and the individual phases of the combined algorithm PTaxi. Section 6 clarifies the important implementation details of our algorithm, with key considerations namely data compatibility and precomputation. Finally, in Section 7, we conduct experiments to assess the results and performance of the algorithm, including comparisons with the original taxi-sharing algorithm.

2 Problem Statement

This section introduces the taxi-sharing problem and the public transit problem, as well as the underlying shortest path problem. Additionally, it is essential to clarify the fundamental differences between taxi-sharing and public transit.

2.1 Shortest Path Problem

Since the underlying problem to be solved in both taxi-sharing and public transit is the shortest path problem, we introduce the required notions in the following paragraphs. First, the shortest path problem is briefly stated. Then, we explain Dijkstra's algorithm in its basic form, as this algorithm is used in multiple solutions. Lastly, the contraction hierarchy, a speedup technique frequently used when solving the taxi-sharing problem, is described along with its extension – bucket-based contraction hierarchy.

Problem Description. The shortest path problem is defined in graph theory. Given a weighted graph $G = (V, E)$ with edges that have non-negative weights defined by the function $w : E \to \mathbb{R}_{\geq 0}$, the shortest path P between two nodes in G must be found such that the weight of this path is minimized. This weight is calculated as the sum of the weights of the path's constituent edges: $W = \sum_{e \in P} w(e)$. Finding the shortest path between an origin and a destination in a graph is an important problem with many applications, especially in transport- or network-related fields [3, 11, 13, 16, 17].

Dijkstra's Algorithm. Dijkstra's Algorithm is one of the most well-known and widely used shortest path algorithms. Given the problem definition with graph $G = (V, E)$ as mentioned above, the algorithm searches for shortest paths from a source node $s \in V$ to all vertices $v \in V$ [5]. Dijkstra's algorithm grows a shortest path tree from the source node s. The algorithm uses a priority queue Q to keep track of the different nodes and their found tentative distance $d_s(v)$ from s. Initially, all nodes except for s itself have the distance ∞, while s has the distance 0. The priority queue only contains the source vertex s at first. Then, the nodes are sequentially taken out of the queue in increasing order of the found distance, and are *settled* one by one. Settling a node u means considering all outgoing edges (u, v) from u and checking whether the distance $d_s(u) + w(u, v)$ is better than the currently stored distance $d_s(v)$ of v. This process of trying to improve the distance with a specific edge is called *relaxing* the edge (u, v). If $d_s(u) + w(u, v) < d_s(v)$, the distance $d_s(v)$ and the key of v in the queue are updated according to the new distance. The algorithm then carries on with the next head-of-queue node. Once the queue is empty, all required distances have been found. Nodes in the graph are either *reached*, *unreached* or *settled* [7]. A *settled* node u is a node that is already included in the shortest path tree, whose shortest path from s has been found. A node u is *reached* if it is adjacent to a settled node v but not yet settled itself. Nodes that are neither settled nor reached are considered *unreached*.

Contraction Hierarchy. Contraction hierarchy (CH) is a speedup technique for shortest path algorithms, suitable for road networks due to their hierarchical nature. The CH technique is divided into two main phases: a preprocessing phase and a query phase [6]. During the preprocessing phase, shortcut edges are added to the graph to take advantage of the hierarchy of road networks. This hierarchy is determined by assigning an importance level to each node based on certain criteria like traffic volume or centrality. Nodes in the original graph are *contracted* individually in ascending order of importance. When a node is contracted, it is removed from the network, and shortcut edges are added to its neighboring nodes to ensure that the shortest paths passing through the removed node are still preserved. The result of this phase is an augmented graph of the road network, consisting of the original graph plus the found shortcut edges, called the contraction hierarchy. Although preprocessing can be computationally intensive, it only needs to be performed once for a given graph, as road networks do not change significantly over time [7]. In the query phase, the shortest path is found using the contraction hierarchy. The query phase searches for paths that follow an up-down pattern in the hierarchy, meaning it ascends the hierarchy to a higher level and then descends back down. It has been shown that there always exists a shortest path that is also an up-down path [6]. Here, the bidirectional Dijkstra's search is used, running two specialized searches: a forward search from the source specifically for upward edges, and a backward search from the target only for downward edges. The CH query finds the shortest up-down path, thereby exploring all necessary meeting nodes between the two Dijkstra searches [7].

Bucket-based Contraction Hierarchies. CH with buckets is introduced by Knopp et al. as an extension to speedup algorithms such as CH [12]. Bucket-based CH (BCH) is an effective CH-based approach for the one-to-many shortest path problem, which combines the information during the forward and backward search effectively. Given a graph $G = (V, E)$, the problem is to find all shortest paths from one source node s to a set of target nodes T. Each node v in the input graph is associated with a bucket $B(v)$, which is empty at first. This bucket will be continuously filled during backward searches on subgraph $G^{\downarrow}$ from targets in T. A pair of $(t, dist(v, t))$ is stored in $B(v)$, when a path from v to t is found during the backward search from target t [12]. During the forward search afterwards, a tentative distance array D stores the distances between each pair (s, t). When the forward search settles a node v, it scans the node's bucket, and for each entry $e = (t, dist(v, t)) \in B(v)$ the tentative distance $D_s(t)$ is updated to $min\{D_s(t), dist(s, v) + dist(v, t)\}$. This bucket-based approach can be utilized analogously in the case of many-to-one shortest path problem, in which the forward and backward search exchange roles.

2.2 Taxi-Sharing Problem

Taxi-sharing, also referred to as ride-sharing or carpooling, is a transportation service model where multiple passengers with similar travel routes share a single taxi or vehicle. This approach aims to optimize the use of available vehicle space,

lower travel costs, and reduce environmental impact. The taxi-sharing problem involves assigning incoming rider requests, which specify a journey from an origin to a destination, to a fleet of taxi-like vehicles while adhering to rider constraints. The goal is to determine the optimal assignments of riders to vehicles in order to minimize an objective function [14]. Taxi-sharing can be viewed as a specific type of vehicle routing problem, where the shortest path problem is a key component.

Road Network. A road network is considered as a directed graph $G = (V, E)$, with road intersections as vertices in V and road segments as edges in E. For each edge $e = (v, w) \in E$, a travel time between the two intersections is defined as $\ell(e) = \ell(v, w)$. In addition, the shortest path distance from node v to node w can be described with $dist(v, w)$.

Vehicle, Stop. The algorithm keeps track of a fleet F of vehicles. Each vehicle ν is defined as $\nu = (l_i, c, t_{serv}^{min}, t_{serv}^{max})$ with an initial location of l_i in the road network G and a capacity of c passengers that the vehicle can carry at once. The remaining two parameters represent the service time of the taxi vehicle, which operates in the time interval of $[t_{serv}^{min}, t_{serv}^{max})$. A current route of each vehicle is described as $R(\nu) = \langle s_0, \ldots, s_{k(\nu)} \rangle$ with stops s_i for $0 \leq i \leq k(\nu)$ and associated stop locations $l(s_i) \in V$. The vehicle is always located between stops s_0 and s_1. Therefore, $k(\nu)$ represents the number of stops the vehicle has yet to visit. Each vehicle's route is updated accordingly when the vehicle reaches its upcoming stop or is assigned new stops.

Request. In our scenario, the dispatching algorithm receives incoming ride requests from passengers and has to instantly match them to a feasible vehicle in the fleet. An incoming request can be denoted as $r = (orig, dest, t_{req})$, where $orig$ is the original location, $dest$ is the desired destination of the rider, and t_{req} is the time when the request is submitted, which is also the earliest departure time of the request $t_{dep}^{min}(r)$.

Insertion. The task of the algorithm is to find a feasible insertion of a pickup and dropoff pair into an existing vehicle's route for each incoming request. A so-called insertion of a request can be represented by a 6-tuple $\iota = (r, p, d, \nu, i, j)$, which describes the scenario where the vehicle ν picks up the passenger at pickup location p immediately after stop s_i in its route and drops off the passenger at dropoff location d immediately after stop s_j, with $0 \leq i \leq j \leq k(\nu)$.

Problem Statement. To find the best vehicle and the best insertion for an incoming request, different shortest path distances must be calculated. The chosen vehicle must have enough free capacity to accommodate the rider and must also be within its service hours. The resulting insertion should optimize a set of conditions, depending on the considered use cases. Some algorithms aim to minimize the detour of the vehicle, ensuring that other passengers on the vehicle are not negatively affected, while also optimizing the arrival time at the destination for

the new rider. Other algorithms minimize the number of vehicles on the roads by maximizing their occupancy. Therefore, different cost functions can be used by different solutions. To find the best vehicle for a request, the algorithm evaluates the cost function for each feasible vehicle and selects the one with the lowest cost.

2.3 Public Transit Problem

Public transit (PT), also known as public transportation, is a system of transportation services designed to move large numbers of people within urban and suburban areas. This system typically includes buses, trains, subways, trams, and ferries, which operate on fixed routes and schedules. The primary goals of PT are to provide an efficient, cost-effective, and environmentally friendly alternative to private vehicle use. Unlike the taxi-sharing problem that operates on a road network, the PT problem operates on a PT network with a predefined timetable. The notions used in PT networks are defined as follows.

Station. A station is a fixed location in which a PT vehicle may make a stop to allow passengers to board or alight from the vehicle. For example, a bus stop, a metro station, and a ferry pier may all act as a PT station. Additionally, a station allows passengers to change between different PT modes.

Connection. An elementary connection between two stations represents a PT vehicle traveling from one station to another at a given time without any intermediate stops.

Trip. A trip models a PT vehicle traveling from one terminus station to another along a fixed route at a fixed time. Thus, a trip is a sequence of connections between the visited stations. For instance, a train, a bus, or a tram trip all have fixed times for each station on the trip.

Line. A line includes multiple trips that travel the exact same sequence of stations. There can only be a limited number of lines in a PT network, but each line is served by multiple trips at different times in a day.

Journey. A journey refers to the travel of a passenger through the PT network. The passenger can use multiple PT modes and change between different trips in a journey to reach their destination station.

Problem Variants. There are multiple problem variants of the PT problem. We are most interested in the *earliest arrival problem* and the *multi-criteria problem.* The default problem is the earliest arrival problem: given a source station σ_s, a target station σ_t and a departure time τ, we have to find the optimal journey j that departs from σ_s no earlier than τ and arrives at σ_t as early as possible. The multicriteria problem is an extension of the earliest arrival problem, where not only the arrival time is optimized but also additional criteria, such as the number of transfers or the cost of travel, are considered. In this variant, we find a Pareto set J of non-dominating journeys with respect to the criteria.

2.4 Contrasting Taxi-Sharing and Public Transit Problems

As stated above, even though taxi-sharing and public transit problems are considered within the same category of algorithms for route planning, they differ in many aspects. The fundamental difference is that taxi-sharing responds on-demand whenever an incoming request is received, while PT operates on a fixed timetable. Taxi-like vehicles have limited capacity. Therefore, a taxi-sharing ride can carry at most three to four passengers. In contrast, PT vehicles are designed to cater to the public mass and can consequently carry a much larger number of passengers. This also results in taxi-sharing being more costly than PT.

On the one hand, taxi-sharing operates on road networks, which include certain characteristics such as traffic jams, highways, and road construction. On the other hand, PT operates on networks with fixed stations and schedules. Nonetheless, there are still some analogies between the two networks. For example, stops in taxi-sharing can be considered analogous to stations in PT. A vehicle's route in taxi-sharing is similar to a line in PT. While a route needs to be updated when passengers are assigned to a vehicle, the public transit line is always fixed. These differences need to be distinguished between the two problems so that combined algorithm for both problems can be explored.

3 Preliminaries

We now present and briefly discuss several taxi-sharing and PT algorithms that are central to our combined algorithm. Understanding their core principles is essential for comprehending their combination in our implementation.

3.1 Taxi-Sharing Algorithms

LOUD. Local bucket dispatching (LOUD) is an algorithm designed to address the taxi-sharing dispatching problem by utilizing bucket-based contraction hierarchies (BCH) for the on-the-fly computation of shortest paths [2]. An essential contribution of LOUD is the concept of *elliptic pruning*. This technique reduces the search space of the required BCH searches for each request by using the time constraints of previously assigned passengers [2]. This allows the shortest path algorithm to act as a filter for feasible insertions.

KaRRi. Karlsruhe Rapid Ridesharing (KaRRi) is a taxi sharing dispatcher that extends LOUD with additional adjustments, the most significant of which is the introduction of meeting points [14]. These meeting points introduce many-to-many shortest path problems, for which KaRRi must find efficient solutions. In addition to finding the best vehicle for a request, KaRRi must determine the optimal pickup and dropoff locations. As KaRRi continues to use BCH similarly to LOUD, the algorithm implements numerous many-to-many routing techniques to speed up the original queries when applied to multiple pickups and dropoffs. For each request, the possible meeting points are identified by running bounded

Dijkstra's searches within a walking distance of ρ from the *orig* and *dest* of the request r. The algorithm evaluates different types of insertions: ordinary, ordinary paired, pickup before the next stop (PBNS), pickup after the last stop (PALS), and dropoff after the last stop (DALS). For each phase of an insertion type, the best insertion so far ι^* with the minimum cost is updated for use in subsequent phases. KaRRi demonstrates significant speed advantages when compared to other state-of-the-art taxi-sharing algorithms in scenarios with multiple meeting points. Therefore, we build upon KaRRi for the implementation of our algorithm.

3.2 Public Transit Algorithms

RAPTOR. A simple yet efficient algorithm for PT is RAPTOR for Round-bAsed Public Transit Optimized Router [4]. Unlike graph-based PT approaches, RAPTOR is not based on Dijkstra's algorithm. Instead, it operates in rounds, with each round corresponding to a transfer, and computes arrival times by traversing each line at most once per round. Moreover, RAPTOR optimizes not only the arrival time of the journey, but also the number of transfers thanks to its round-based structure. For any two given source and destination stations, it computes all Pareto-optimal journeys that minimize the arrival time and the number of transfers made [4]. The algorithm is essentially a dynamic program with simple data structures and excellent memory locality. Delling et al. also introduce two additional extensions of RAPTOR in their paper, namely, McRAPTOR for a generalization of RAPTOR to optimize more criteria and rRAPTOR to solve the range problem, in which optimal journeys over a time range are found [4].

ULTRA. UnLimited TRAnsfers (ULTRA) is an efficient solution for the multi-modal route planning scenario, which consists of a PT network and a transfer graph representing an additional transportation mode such as unlimited walking, bicycles or cars. Using a complete transfer graph that represents any non-schedule-based mode of transportation, the authors compute a small number of transfer shortcuts that are sufficient for finding all Pareto-optimal journeys [1]. The position of these transfers affects their impact on the journeys. While *initial transfers* and *final transfers* that connect to the first and last PT station of the journey have a large impact on its travel time, *intermediate transfers* between PT legs are less relevant for the optimal journey. Therefore, the preprocessing technique in ULTRA is designed to ensure that the number of shortcuts for intermediate transfers remains small. The authors apply bucket CH to compute the more impactful initial and final transfers. These shortcuts and transfers are then integrated into various state-of-the-art PT algorithms including RAPTOR, connection scan algorithm (CSA) or trip-based algorithm, forming the ULTRA-Query algorithm family. Extensive experiments demonstrate that ULTRA enhances these algorithms from limited to unlimited transfers without compromising query speed, resulting in the fastest known algorithms for multi-modal routing [1]. This improvement applies to walking as well as other transfer modes

like cycling or driving. Thus, ULTRA provides a solid foundation on which we can develop our algorithm that specifically combines PT and taxi-sharing.

4 Related Works

After exploring the state-of-the-art algorithms for the taxi-sharing and PT problem individually, in this section, we review existing solutions that address the multi-modal transportation problem. We discuss their respective benefits and drawbacks, which serve as motivation for the development of our combined algorithm.

Graph-based model. The approach presented by Huang et al. [9] aims to implement multi-modal route planning with carpooling and PT by modeling both the carpooling and PT network as time-expanded graphs. These two graphs are then merged together into a single graph, on which standard routing algorithms such as Dijkstra's algorithm can be run [9]. Additionally, the proposed merging technique considers the fuzziness or flexibility of carpooling. Although this approach is simple and straightforward, it still carries the known disadvantages of graph-based models, namely large and unstructured graphs with a high overhead of priority queues when applying standard routing algorithms. Moreover, as the resulting combined graph is of larger size, this issue is even worsened.

On-demand Bus. A closely related PT mode to taxi-sharing is the on-demand bus or non-scheduled line. They differ in that on-demand buses are timely on-demand but run on a fixed bus line with predefined stops. These types of buses usually operate in rural areas where not many other PT services are available. They are often used in combination with other PT modes, as users sometimes require on-demand buses to reach other high-frequency PT stations. Melis et al. [15] suggest integrating a large-scale on-demand bus system with a high-frequency fixed line PT network (metro). There are several trip types that can be chosen in this combination: either only bus, only metro, or a mix of both. This approach aims to simultaneously decide on the trip type for each passenger, route the on-demand buses, and assign each passenger to the corresponding stations depending on their trip type [15]. This proposed solution is, however, an insertion-based heuristic for an offline variant of the problem, which does not ensure exact routing.

LOUD + RAPTOR. Tiede [20] attempts to integrate taxi-sharing and PT routing, using the two previously mentioned algorithms, LOUD [2] for taxi-sharing and RAPTOR [4] for PT. In this thesis, the taxis serve as a substitution between PT stations. For the sake of simplicity, the bachelor thesis disregards insertions at the end of a vehicle's route, even though this is the more commonly observed use case [20]. In a preprocessing step, the proposed algorithm searches for possible taxi-sharing connections between stations, which are then considered in their query algorithm [20]. Nevertheless, the number of ride transfers considered during the query algorithm is still substantial, leading to longer query

times compared to RAPTOR. This is due to the fact that the preprocessing phase operates on a considerably large ride transfer graph. Moreover, the preprocessing technique builds the ride transfer graph based on a certain state of the taxi-sharing network. This requires the ride transfer graph to be built from scratch for each newly inserted request. Also, the author assumes that rides get pre-booked before the vehicles start driving [20]. In reality, ride transfers may also be assigned to drivers who have already begun their routes. Public trAnsit Ride-sharing ROuTer (PARROT) is an implementation by Patrick Steil of the combined algorithm that is based on this thesis. The algorithm aims to combine the successor of LOUD – KaRRi [14] and RAPTOR [4], which however only focuses on the scenario where taxi-sharing acts as a transfer between PT legs.

5 Algorithm

Even though on their own, KaRRi and ULTRA both produce efficient runtime and optimal results for incoming requests, combining the two stand-alone frameworks requires consideration about how to effectively use their results while preserving their existing implementations with minimal adjustments.

In this section, we explain the general idea and structure of our algorithm PTaxi. First, we provide an overview of the algorithm. Then, we present the different phases of the algorithm in the order of their implementation, and explain how we implement them. Here, we describe only the high-level algorithm, the implementation details are discussed later in Section 6.

5.1 Overview

When combining multiple modalities, there are several available scenarios to be considered. In our case, we combine taxi-sharing with PT, the most practical scenario is to incorporate taxi-sharing rides at the beginning and end of a PT journey. A taxi-sharing transfer between PT journeys is not particularly attractive for users, as they would have to change between the different modalities constantly. Moreover, PT stations are primarily designed for efficient transfers, enabling the users to change to the next PT station by foot, rather than by other modes of transport.

Initially, when a request is received, the algorithm runs the taxi-sharing algorithm (KaRRi) and the PT algorithm (ULTRA-RAPTOR) independently, without any additional adjustments. The lower cost of these two journeys: one pure taxi-sharing and one pure PT journey, is saved as a global upper bound cost. Thus, any calculation in later phases that exceeds this bound may be aborted directly

In the combined algorithm PTaxi, a final journey consists of three legs: a first taxi leg, a PT leg, and a final taxi leg. A pure taxi-sharing or pure PT journey can be represented by marking the two remaining legs as invalid. Based on this journey model, we divide the algorithm into three corresponding phases. The first phase deals with the first taxi leg, during which the user is taken from their

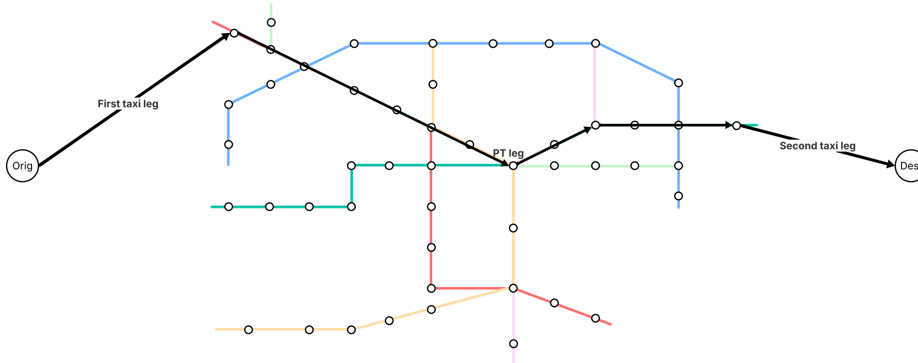

Fig. 1: Three legs of a combined journey in PTaxi.

origin to a PT station for further transport. Then, we use the PT algorithm to route the user from different initial PT stations to other PT stations closer to their desired destination. Lastly, for the final taxi leg, we approximate the distance from all PT stations within the PT network to the destination.

For the sake of simplicity, we use direct distance approximation for the final taxi leg, and make the optimistic assumption that there will always be a taxi-sharing vehicle available to pick up the user from a PT station. We use this approximation to calculate the combined cost of the three legs and determine the best PT station for the user to get off. The actual routing of a taxi-sharing vehicle for the second taxi leg can be performed shortly before the passenger's arrival at the PT station. We could trigger a pre-booking 15 minutes in advance, to ensure that the user is picked up as soon as they arrive at the final PT station.

Our focus, therefore, lies in the first taxi leg, as this is the most complex phase due to the combinatorial complexity inherent in the taxi-sharing request. We kept the two remaining phases simple for the initial implementation of our algorithm. The results of the first phase also serve as the input for the following phases.

5.2 First Taxi Leg

For the first taxi leg, we handle incoming requests similarly to KaRRi. This process involves evaluating different types of insertion in the following order: PALS, ordinary, DALS, PBNS. An overview of the insertion types can be found in Figure 2. The key difference from KaRRi is that for each request we need to find a minimum cost taxi assignment to each PT station, instead of a single minimum cost for the request's destination. We call this a one-to-many taxi-sharing query, as opposed to the one-to-one taxi-sharing query in KaRRi. The result of this first phase is the taxi-sharing costs to route the user to every PT station in the network that can be used in the next phase. Using the global upper bound cost, we can disregard any trip to a PT station that has a higher cost than the direct taxi-only trip to the destination. We made use of the exisiting

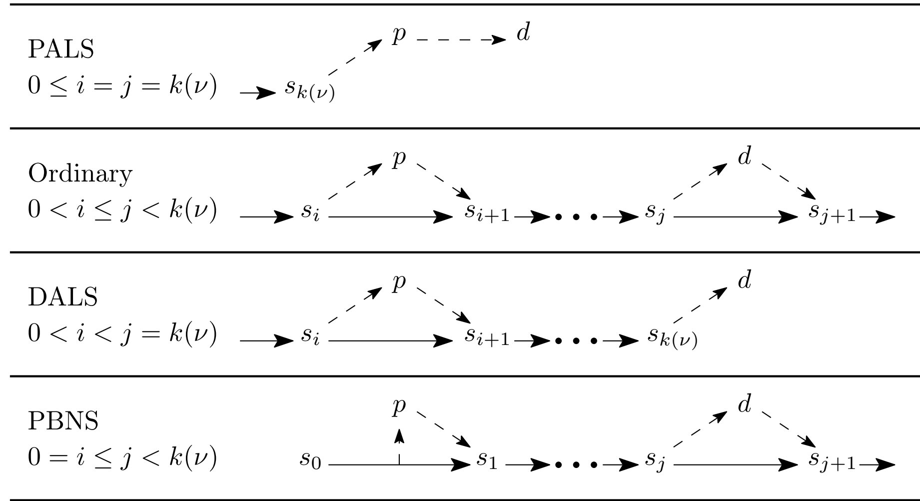

Fig. 2: KaRRi insertion types: pickup before next stop (PALS), ordinary, dropoff after last stop (DALS) and pickup before next stop (PBNS). Shows characterization of each type based on the pickup and dropoff insertion points i and j of an insertion $\iota = (r, p, d, \nu, i, j)$. Illustrations depict the current route of ν (solid arrows) with stops $s \in R(\nu)$ as well as the detours to and from p and d (dashed lines).

implementation of KaRRi and the neccessary extensions for the one-to-many query from the request's origin to all PT stations. In the following paragraphs, we will go through the named insertion types and explain the adjustments that we have conducted, in order to adapt from a one-to-one query in KaRRi to a one-to-many query in PTaxi.

PALS. For pickup after last stop insertions, we require the distances from last stops to the feasible pickups and the distances from the pickups to the PT stations. The distances between the last stops and pickups have to be calculated on-the-fly as the last stops are updated with every insertion. We compute these distances using a BCH query rooted at each pickup with bucket entries for every last stop. In contrast, the PT stations are static and therefore bucket entries for them can be precomputed. In this case, we precompute PT buckets, against which we then run a BCH query from the pickups. Once these distances are calculated, the assignment can be assembled and the minimum cost for each station is updated.

Ordinary. In case of an ordinary insertion, where the PT stations are inserted in between taxi stops, the distances to be calculated are between the existing stops in vehicles' routes and the PT stations. Similarly to the PALS, we can use the precomputed BCH buckets of the PT stations for this insertion type.

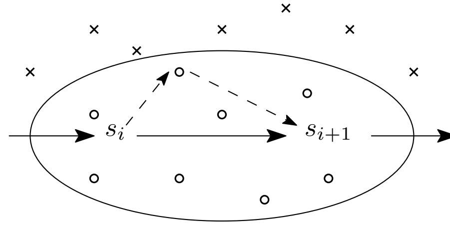

Fig. 3: Detour ellipse of the stop pair s_i and s_{i+1}. Circles within the ellipse represent PT stations that are relevant for insertion at index i, crosses outside the ellipse represent PT stations that are eliminated.

In addition, we apply a key finding from the LOUD [2], the predecessor of KaRRi, known as elliptic pruning (Section 3.1). Figure 3 visualizes the detour ellipse of a stop pair and how PT stations may appear around it. Between two consecutive stops, we only consider stations that are located within the detour ellipse; otherwise, the insertion will violate the constraints of other passengers on the vehicle. Therefore, we calculate and update the stations within these ellipses as soon as a new stop is added to a vehicle's route. By doing so, we move the complexity outside the enumeration process and directly narrow down the feasible stations for ordinary insertions. During simulation, we can directly get the relevant PT stations for a pair of stops and find the distances between the PT stations and the stops using BCH with the static stations.

A special case is when the ordinary insertion is paired, which means the pickup and dropoff are inserted between the same pair of stops. Then, the distances between the pickups and stations are needed, which we have already calculated for the PALS case.

DALS. For dropoff after last stop insertions, the required distances are those between the last stops and the PT stations. Here, we once again utilize the precomputed BCH buckets of static PT stations and run the searches from the vehicles' last stops against them to find the distances.

PBNS. Pickup before next stop is considered last, in which we need the distance from the vehicle's current location to the pickups. However, as our extension for the one-to-many query mostly involves the dropoff side of the insertion, the special handling of the pickup side is kept as it is in KaRRi, delaying exact calculations with an initial approximated cost. In this special case, we also only consider PT stations that lie between the detour ellipse of two consecutive stops as in the case of ordinary insertion. The required distances for the dropoff side is similar to those in ordinary insertion and can be calculated directly during enumeration.

5.3 PT Leg

For the PT leg, we build upon the implementation of the ULTRA-RAPTOR algorithm, which extends the original RAPTOR algorithm with unlimited walking at the beginning, the end and in between the PT trips. The initial transfers of the algorithm are further extended with the taxi-sharing results from the first phase, aside from the walking transfers, while the final transfers are extended with the direct distance approximation for the second taxi leg.

Initial Transfers. At the beginning of the PT journey, we check for taxi-sharing transfers to all stations. If the arrival by taxi transfer is currently the earliest arrival, it will be saved as an earliest arrival label of this round at this station and used for further routing in the following RAPTOR rounds. We use the same optimization criteria as the original PT algorithm – arrival time and number of transfers – as this is the most common combination of objectives for PT queries.

Final Transfers. For the final transfers from PT stations to the target destination, whenever the PT algorithm finds a new earliest arrival time for a station σ, we check whether this also implies a new earliest arrival at the destination using a taxi-sharing ride, as approximated by the direct vehicle travel time from σ to the destination. If there is no other arrival at an earlier time, the transfer is similarly saved as a taxi-sharing final transfer for the journey.

6 Implementation Details

Building on the overview provided in the preceding section, we now present key implementation details of our algorithm. The following subsections describe the specific constructs we developed for PTaxi to enable the combination of the two frameworks and to speed up runtime performance, along with difficulties that we encountered during implementation.

Our algorithm is written based on the source code of KaRRi[1] and ULTRA[2] in C++ 17. We leverage the existing implementation of the frameworks to run taxi-sharing and PT independently. We wrote our extensions to the two frameworks primarily based on the existing code base, adapting it to our specific use cases.

6.1 Data Compatibility

In ULTRA, there are different query algorithms that route from vertex to vertex – exact locations in the transfer graph, or from station to station – restricting to the PT stations only. We chose the ULTRA-RAPTOR algorithm, as it routes from vertex to vertex, and has already implemented the initial and final walking transfers, next to which we can similarly incorporate our taxi-sharing transfers. However, this decision also results in a more complicated preprocessing for the

[1] https://github.com/molaupi/karri
[2] https://github.com/kit-algo/ULTRA

PT itself, where we encountered problems with the data compatibility between the two different frameworks. As we use both KaRRi and ULTRA in PTaxi, the data such as the pedestrian graph for walking transfer, the PT network, and the requests must be consistent and, if required, converted accordingly.

Graph. As we use the `Berlin-1pct` instance in KaRRi for testing, we also need to convert the pedestrian graph in KaRRi to a format readable by ULTRA. KaRRi and ULTRA both internally use their own binary graph format, so the only transferrable intermediate format is the format used by the 9th DIMACS Implementation Challenge[3]. We convert the binary pedestrian graph in KaRRi to the DIMACS format and read the graph into ULTRA's custom binary graph format for transfer graph of the PT algorithm.

PT Network. Corresponding to the `Berlin-1pct` road network used in KaRRi, we obtain the PT network data of Berlin from the official website of VBB (Verkehrsverbund Berlin-Brandenburg)[4] in General Transit Feed Specification (GTFS) format. Since the PT stations are important connection points between the two frameworks, we ensure that the PT station IDs are identical in both KaRRi and ULTRA for easier integration. Furthermore, as KaRRi addresses locations primarily with the edge ID in the vehicle graph, the stations must be transformed from coordinates as given by the GTFS format to the corresponding edge ID in KaRRi to be used for routing with taxi vehicles.

Contraction Hierarchies. While both KaRRi and ULTRA utilize CHs for their computations, their internal representation and structure are entirely different. The ULTRA framework assumes that PT stations remain uncontracted in the CH core, using the first vertices in the CH directly as stations. Conversely, KaRRi has no information about the PT network and thus calculates the CH independently of the PT stations. As a result, we use separate CHs for the two frameworks, with each framework calculating its own CH from the same pedestrian network graph. This is a known issue and could be optimized by using a single CH for both frameworks, though that would require adjustments to how the frameworks handle CHs internally.

6.2 Precomputation

Working with a PT network means working with fixed PT stations and defined schedules that are considerably stable and less affected by frequent changes in traffic than dynamic stops as in taxi routes. Therefore, there is a high potential for precomputing useful information for shortest path queries. As both the taxi-sharing algorithm and the PT algorithm utilize BCH to compute the distances related to static stations, these buckets can be calculated beforehand.

[3] http://diag.uniroma1.it/challenge9/download.shtml

[4] https://www.vbb.de/fileadmin/user_upload/VBB/Dokumente/API-Datensaetze/gtfs-mastscharf/GTFS.zip

The station buckets in the pedestrian network is used by the ULTRA-RAPTOR algorithm to find initial and final transfers by foot. The station buckets in the road network are used to compute distances for the taxi-sharing algorithm. By utilizing these buckets we can calculate the distances to and from the PT stations in the road network by taxi. We precompute these CH buckets in a separate executable, save them as binary files, and later load them into the main application to speed up simulations. This is possible due to the fact that PT stations are static and do not change through out the entire algorithm runtime.

7 Evaluation and Discussion

PTaxi is a combination of the two state-of-the-art algorithms: the taxi-sharing algorithm KaRRi and the PT algorithm ULTRA [14, 1]. Our codebase[5] is built upon an implementation of PARROT – a combination of KaRRi and RAPTOR, mentioned in Section 4.

We would like to clarify that our current state of implementation is not yet complete. We have implemented the first taxi leg, which is an extension of the KaRRi algorithm to one-to-many requests. However, we have encountered problems while incorporating the PT algorithm, specifically the ULTRA-RAPTOR algorithm, into our combined algorithm PTaxi. The PT results we obtained are not valid and therefore not evaluated. We suspect there is a problem in the mapping between the two algorithms, but due to time constraints, we have not been able to identify and solve the exact problem. In this section, we focus exclusively on the results and evaluation of the first taxi leg. The following experiments are run on a single computer with the following specs: AMD EPYC Rome 7702P - 64 core processor @ 2.0-3.35GHz, 1024GB DDR4-RAM @ 2966MHz, 256MB L3 Cache, AVX2 (256-bit SIMD instructions).

7.1 Input Data

For the input instance, we use the `Berlin-1pct` (`B-1%`) of KaRRi, which represent 1% of the weekday demand for taxi sharing in the Berlin [2]. The request sets for Berlin were artificially created with the help of the Open Berlin Scenario [22] for the MATSim transportation simulation [21]. We assume all riders in our simulations walk to the meeting points and use the default walking radius of $\rho = 300s$. Key figures of the input instance are shown in Table 1.

Correspondingly, we generated the data for the PT system in Berlin as previously mentioned in Section 6.1. The raw data was preprocessed for use with the ULTRA-RAPTOR algorithm, key figures of the PT data can be found in Table 2.

7.2 First Taxi Leg

Cost Pruning. As previously described in Section 5.2, the first taxi leg involves running one-to-many taxi-sharing requests from the origin to the PT stations.

[5] Available at https://github.com/molaupi/PARROT

Table 1: Key figures of KaRRi's benchmark instance. Shows number of vertices ($|V|$), edges ($|E|$), vehicles (#veh.), and requests (#req.). In addition, shows average number (rounded down) of pickups (N_ρ^p) and dropoffs (N_ρ^d) for walking radius $\rho = 300s$ on the `Berlin-1pct` instance.

Instance	$\|V\|$	$\|E\|$	#veh.	#req.	$\rho = 300$s	
					N_ρ^p	N_ρ^d
`Berlin-1pct`	94422	193212	1000	16569	44	44

Table 2: Key figures of ULTRA-RAPTOR data. Shows number of vertices ($|V_{transfer}|$) and edges ($|E_{transfer}|$) in the transfer graph, the number of stations (#stations), routes (#routes), and trips (#trips) in the PT system of Berlin.

Instance	$\|V_{transfer}\|$	$\|E_{transfer}\|$	#stations	#routes	#trips
`Berlin-1pct`	8510	8582	8510	2114	71976

By utilizing the upper bound cost obtained from a pure taxi-sharing or pure PT journey, we can prune out PT stations that have a higher cost than this upper bound. To evaluate the effectiveness of this simple pruning technique, we show the number of stations that remain relevant for each request and call this the number of results of the first taxi leg.

In Figure 4, we visualize the number of taxi-sharing results obtained in relation to the direct taxi-sharing cost. The x-axis is divided into 50 equal-sized bins across the taxi cost range. Each point in the plot represents the mean number of results among all requests that fall into a specific bin. The histogram below displays the distribution of the requests over the bins of the x-axis. We can observe a weak linear correlation between the results count and the taxi-sharing cost. If the direct taxi cost is small, the number of results for the first taxi leg is also correspondingly small. Most requests are distributed in the cost range below 25,000. Here, the pruning is considerably effective when compared to the total number of 8510 stations. On average, the number of results for a request lies at approximately 6147. High cost ranges have less effective pruning and higher variations, since there are fewer requests that have such a high cost.

Insertion Types. In the first taxi leg, PT stations can be inserted into different parts of a taxi route: either between taxi stops or at the end of the taxi routes. We implemented the different insertion types similar to KaRRi (See Section 5.2). Of these, ordinary and PBNS are types in which the PT stations are located between stops, while PALS and DALS are types in which the PT stations are inserted after the last stops of the taxi routes.

In Figure 5, we visualize the distribution of different insertion types in the first taxi leg results. The most common type is DALS with PBNS with 82.5% of the total results, while the most uncommon types are PBNS and ordinary

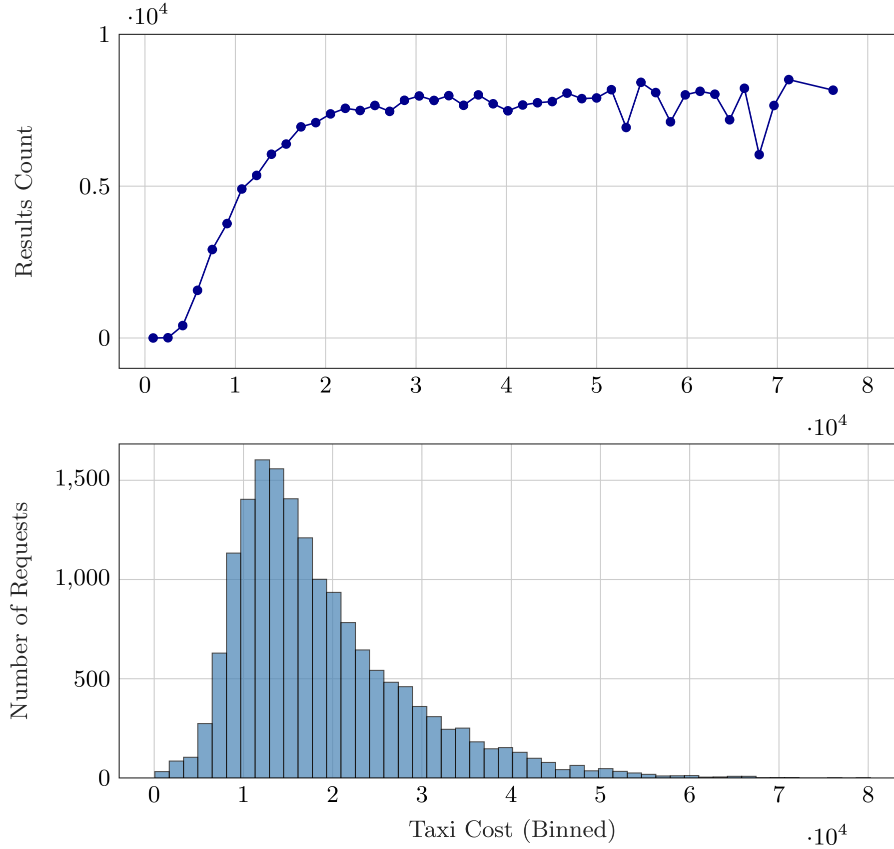

Fig. 4: First taxi-sharing leg results count by taxi cost (top) and distribution of requests over the taxi cost range (bottom). The x-axes are identical.

insertions with only 0.9% and 0.1%, respectively. It can be seen that for types where the PT stations are inserted between taxi stops, the number of results is much lower. This is because of an additional constraint that only PT stations that lie within the detour ellipse of two consecutive stops are taken into consideration. For after last stop (ALS) types, since there is no detour ellipse to narrow down the selection of stations for the last stop, far more PT stations come into question. In the case of PALS, the pickups are still bounded around the request origin, while in the case of DALS, the dropoffs are PT stations and are not restricted at all.

Runtimes. We also evaluated the runtimes of the first taxi leg. In Figure 6, we compare the runtimes of the different insertion types. These phases' runtimes correspond to their occurrences as previously displayed, with the ordinary phase's runtime at the lowest, averaging $144\mu s$ per request, while the PALS phase takes

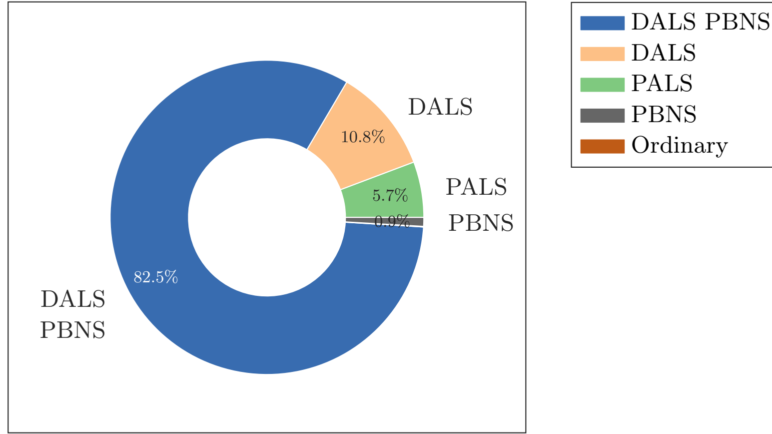

Fig. 5: Distribution of insertion types including DALS with PBNS, DALS, PALS, PBNS and ordinary in first taxi leg results (average per request).

the longest with $74161\mu s$ and has the highest variability. Even though the number of PBNS insertions are low, the runtime is considerably high, as we need to calculate the distance from the vehicle's exact location to the pickup locations. The total assignment time of the first taxi leg is aproximately $139ms$, in comparison to a direct one-to-one query in KaRRi that takes only $1.5ms$.

We visualize the relation between the total runtime and the direct taxi-sharing cost in Figure 7. The taxi cost correlates with the runtime. For low-cost ranges, the runtimes are considerably low due to aggressive pruning, while higher costs generally lead to longer runtimes and also with significant scatter.

7.3 PT Algorithm

We have not been able to run experiments for the PT leg itself, as the implementation of the combined algorithm is not yet complete. Instead, we ran the ULTRA-RAPTOR algorithm with unlimited walking independently to obtain runtime numbers for comparison with the first taxi leg. The average runtime for a single request is around $4.1ms$, which is considerably faster than the first taxi leg. Furthermore, the PT algorithm initially considers an average of 4189 PT stations as points of interest per request. This number is comparable to the average of 6147 stations from the first taxi leg's results.

These numbers suggest that the complexity of the combined algorithm lies primarily in the first taxi leg, due to its varied insertion types and the overhead of finding the taxi-sharing routes to all PT stations. This observation supports our decision to focus our initial efforts on implementing and optimizing this leg.

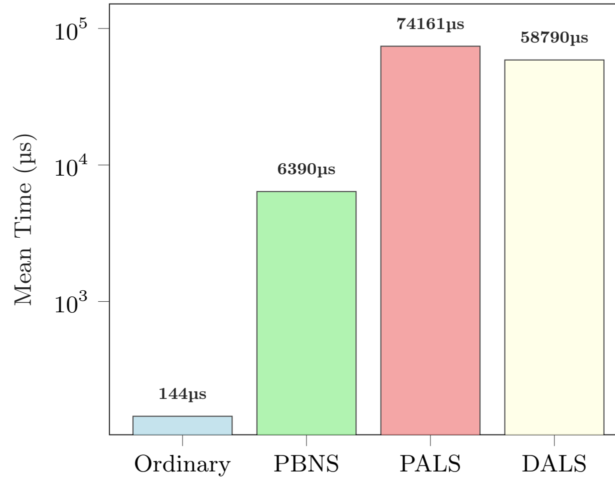

Fig. 6: Runtime in μs by insertion types including ordinary, PBNS, PALS, DALS (average per request).

7.4 Discussion

We acknowledge the incompleteness of our implementation and find it unfortunate to present only parts of the entire planned algorithm. The results we presented are limited and evaluate only the first taxi leg. Future work will focus on integrating the PT algorithm into our combined algorithm PTaxi, building on the foundation we have established.

For the first taxi leg, there is potential for further pruning to speed up the one-to-many queries. Currently, the cost pruning technique we apply is relatively primitive. We can imagine different types of pruning techniques that may be suitable for the different insertion types individually. Particularly the ALS insertions that currently takes up most of the runtime of the first taxi leg.

As we are currently using a direct distance approximation for the second taxi leg, the exact routing of the second taxi leg is also a desirable feature. In order to find a taxi vehicle that is available to pick up the passenger from a PT station to bring them to their destination, we plan to trigger the taxi-sharing algorithm 15 minutes before the passenger arrives at the planned PT station. This would be a direct one-to-one taxi-sharing request from the optimal PT station to the destination. Moreover, since the optimal PT station was decided based on the distance approximation, we can also trigger the taxi-sharing algorithm more often to find the exact taxi cost at the stations and decide on a better station for the passenger to get off.

Finally, the combined cost function should also be tested and adjusted. The combined cost function should weigh the different legs of the combined algorithm correspondingly, in order to find an optimal journey that is convenient but also cost efficient for the user.

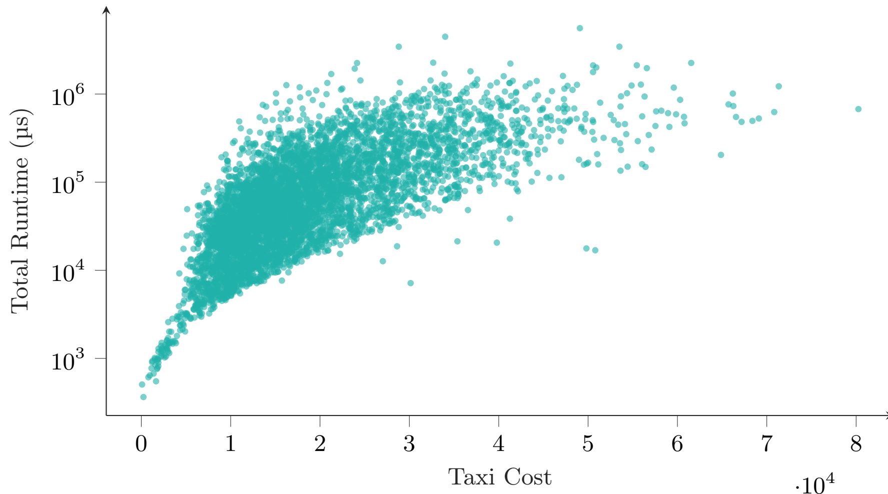

Fig. 7: Total runtime (μs) in relation to taxi cost.

8 Conclusion

In this paper, we introduced *PTaxi*, a combined algorithm based on the taxi-sharing algorithm KaRRi [14] and the public transit algorithm ULTRA [1]. By employing precomputation and pruning techniques that leverage static PT stations, we implemented the first taxi leg of the combined journey as a one-to-many extension of the taxi-sharing algorithm. We provided a high-level overview of our algorithm, describing the different insertion types adopted from KaRRi for the first taxi leg, and explained implementation details, highlighting the challenges we encountered while integrating the two algorithms.

Our evaluation focused primarily on the first taxi leg's results, which demonstrated the effect of our pruning techniques and compared the different insertion types. While the one-to-many taxi-sharing request we implemented is considerably slower than a direct one-to-one request in KaRRi, this can be attributed to the large number of stations that must be considered for each request. In contrast, the PT algorithm proved to be significantly faster than the first taxi leg.

We acknowledge that this study represents only an initial step toward successfully combining different modalities for mobility as a service. Our implementation is incomplete and can be further optimized and enhanced across all three legs of the combined journey. Future research could focus on optimizing the integration's runtime efficiency, further exploring the remaining use cases in the combination of taxi-sharing and public transit, or incorporating other modes of transport, such as cycling or e-scooters.

Acknowledgments. This research was conducted as part of the *Praxis der Forschung* module at the Chair of Algorithm Engineering, under the supervision of Moritz Laupich-

ler and Professor Peter Sanders. I would like to thank my supervisor, Moritz Laupichler, for his guidance and support throughout the entire project. I am also grateful to Professor Sanders for providing the opportunity to conduct this research at his chair. This project has been a valuable learning experience, and I am honored to have presented this work to a community with a strong interest in the topic.

Bibliography

[1] Baum, M., Buchhold, V., Sauer, J., Wagner, D., Zündorf, T.: Unlimited transfers for multi-modal route planning: An efficient solution. In: 27th Annual European Symposium on Algorithms (ESA 2019), Schloss Dagstuhl – Leibniz-Zentrum für Informatik (2019), https://doi.org/10.4230/LIPIcs.ESA.2019.14, URL https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.ESA.2019.14

[2] Buchhold, V., Sanders, P., Wagner, D.: Fast, Exact and Scalable Dynamic Ridesharing. Proceedings, Society for Industrial and Applied Mathematics (2021), https://doi.org/10.1137/1.9781611976472.8, URL https://epubs.siam.org/doi/10.1137/1.9781611976472.8

[3] Cherkassky, B.V., Goldberg, A.V., Radzik, T.: Shortest paths algorithms: Theory and experimental evaluation. Mathematical Programming **73**(2) (1996), ISSN 1436-4646, https://doi.org/10.1007/BF02592101

[4] Delling, D., Pajor, T., Werneck, R.F.: Round-Based Public Transit Routing. Society for Industrial and Applied Mathematics, Philadelphia, PA (2012), ISBN 978-1-61197-212-2, https://doi.org/10.1137/1.9781611972924.13, URL http://epubs.siam.org/doi/abs/10.1137/1.9781611972924.13

[5] Dijkstra, E.W.: A note on two problems in connexion with graphs. Numerische Mathematik **1**(1) (1959), ISSN 0945-3245, https://doi.org/10.1007/BF01386390

[6] Geisberger, R., Sanders, P., Schultes, D., Delling, D.: Contraction hierarchies: Faster and simpler hierarchical routing in road networks. In: McGeoch, C.C. (ed.) Experimental Algorithms, Lecture Notes in Computer Science, Springer, Berlin, Heidelberg (2008), ISBN 978-3-540-68552-4, https://doi.org/10.1007/978-3-540-68552-4_24

[7] Geisberger, R., Sanders, P., Schultes, D., Vetter, C.: Exact routing in large road networks using contraction hierarchies. Transportation Science **46**(3) (2012), ISSN 0041-1655, https://doi.org/10.1287/trsc.1110.0401

[8] Hensher, D.A., Mulley, C., Ho, C., Wong, Y., Smith, G., Nelson, J.D.: Understanding Mobility as a Service (MaaS): Past, Present and Future. Elsevier (2020), ISBN 978-0-12-820397-2

[9] Huang, H., Bucher, D., Kissling, J., Weibel, R., Raubal, M.: Multimodal route planning with public transport and carpooling. IEEE Transactions on Intelligent Transportation Systems **20**(9) (2019), ISSN 1558-0016, https://doi.org/10.1109/TITS.2018.2876570

[10] Jain, V., Sharma, A., Subramanian, L.: Road traffic congestion in the developing world. In: Proceedings of the 2nd ACM Symposium on Computing for Development, ACM DEV '12, Association for Computing Machinery, New York, NY, USA (2012), ISBN 978-1-4503-1262-2, https://doi.org/10.1145/2160601.2160616, URL https://dl.acm.org/doi/10.1145/2160601.2160616

[11] Kliemann, L., Sanders, P.: Algorithm Engineering: Selected Results and Surveys, Lecture Notes in Computer Science, vol. 9220. Springer In-

ternational Publishing, Cham (2016), ISBN 978-3-319-49486-9, https://doi.org/10.1007/978-3-319-49487-6, URL http://link.springer.com/10.1007/978-3-319-49487-6

[12] Knopp, S., Sanders, P., Schultes, D., Schulz, F., Wagner, D.: Computing Many-to-Many Shortest Paths Using Highway Hierarchies. Proceedings, Society for Industrial and Applied Mathematics (2007), https://doi.org/10.1137/1.9781611972870.4, URL https://epubs.siam.org/doi/abs/10.1137/1.9781611972870.4

[13] Kumari, S.M., Geethanjali, D.N.: A survey on shortest path routing algorithms for public transport travel. Global Journal of Computer Science and Technology (2010), URL https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=1aa362cd121b9518437b6d9308a4b14e0764c556

[14] Laupichler, M., Sanders, P.: Fast Many-to-Many Routing for Dynamic Taxi Sharing with Meeting Points. Proceedings, Society for Industrial and Applied Mathematics (2024), https://doi.org/10.1137/1.9781611977929.6, URL https://epubs.siam.org/doi/abs/10.1137/1.9781611977929.6

[15] Melis, L., Queiroz, M., Sörensen, K.: The integrated on-demand bus routing problem: Combining on-demand buses with a high-frequency fixed line public transport network. Computers and Operations Research **164** (2024), ISSN 03050548, https://doi.org/10.1016/j.cor.2024.106554

[16] Ortega-Arranz, H., Llanos, D.R., Gonzalez-Escribano, A.: The Shortest-Path Problem: Analysis and Comparison of Methods. Synthesis Lectures on Theoretical Computer Science, Springer International Publishing, Cham (2015), ISBN 978-3-031-01446-8, https://doi.org/10.1007/978-3-031-02574-7, URL https://link.springer.com/10.1007/978-3-031-02574-7

[17] Pallottino, S., Scutellà, M.G.: Shortest Path Algorithms In Transportation Models: Classical and Innovative Aspects. Springer US, Boston, MA (1998), ISBN 978-1-4615-5757-9, https://doi.org/10.1007/978-1-4615-5757-9_11, URL https://doi.org/10.1007/978-1-4615-5757-9_11

[18] Slavulj, A.P.M., Šojat, D., Prskalo, H., Vidan, L.: An overview of the current state of mobility as a service. Transportation Research Procedia **73** (2023), ISSN 23521465, https://doi.org/10.1016/j.trpro.2023.11.910

[19] Song, C., Monteil, J., Ygnace, J.L., Rey, D.: Incentives for ridesharing: A case study of welfare and traffic congestion. Journal of Advanced Transportation **2021** (2021), ISSN 0197-6729, https://doi.org/10.1155/2021/6627660

[20] Tiede, L.: Integrating ridesharing and public transit routing (2022), bachelor's Thesis, Karlsruhe Institute for Technology

[21] W. Axhausen, K., Horni, A., Nagel, K. (eds.): The Multi-Agent Transport Simulation MATSim. Ubiquity Press (2016), https://doi.org/10.5334/baw, URL https://library.oapen.org/handle/20.500.12657/32162

[22] Ziemke, D., Kaddoura, I., Nagel, K.: The MATSim Open Berlin Scenario: A multimodal agent-based transport simulation scenario based on synthetic demand modeling and open data. Procedia Computer Science **151** (2019), ISSN 1877-0509, https://doi.org/10.1016/j.procs.2019.04.120