Exact and Heuristic Dynamic Taxi Sharing with Transfers Using Shortest-Path Speedup Techniques

Johannes Breitling
Karlsruhe Institute of Technology, Germany

Moritz Laupichler
Karlsruhe Institute of Technology, Germany

Abstract

We introduce a first-of-its-kind efficient, exact algorithm for the dynamic taxi-sharing problem with single-transfer journeys, i.e., a dispatcher that assigns traveler requests to a fleet of shared taxi-like vehicles allowing transfers between vehicles. We extend an existing no-transfer solution by collecting all viable pickup and dropoff vehicles for a request and computing the optimal transfer point for every pair of vehicles. We analyze underlying shortest-path problems and employ state-of-the-art routing algorithms to compute distances on-the-fly, which serves as the basis of dispatching requests with exact and up-to-date travel time information. We utilize constraints on existing routes, pruning techniques for transfer points, and both instruction- and thread-level parallelism to speed up the computation of the best assignment for every traveler. In addition to the exact variant, we propose a tunable heuristic approach that sacrifices solution quality in favor of improved running time.

We evaluate our algorithm on a large road network with realistic input sets (up to 150000 requests). We demonstrate the effectiveness of our speedup techniques and the heuristic. We show first results on the benefits of transfers for taxi sharing on dense request sets, proving that our algorithm is well suited for the analysis of taxi sharing with transfers on large input instances.

2012 ACM Subject Classification Applied computing → Transportation; Theory of computation → Shortest paths; Information systems → Geographic information systems

Keywords and phrases Dynamic taxi sharing, ride pooling, dial-a-ride problem, transfers, route planning

Digital Object Identifier 10.4230/OASIcs.ATMOS.2025.15

Supplementary Material *Software*: `https://github.com/JohannesBreitling/karri-with-transfers` [6], archived at `swh:1:dir:2e1d2d7c9139eb1a02b7cdd760c7c13365d35805`

Funding This paper was created in the "Country 2 City - Bridge" project of the "German Center for Future Mobility", which is funded by the German Federal Ministry of Transport.
Moritz Laupichler: This work was supported by funding from the pilot program Core Informatics of the Helmholtz Association (HGF).

1 Introduction

The current landscape of transportation systems is usually designed around two extremes: Individual transport focuses on private cars that use a lot of space and resources while polluting the environment. On the other end, public transit is mostly slow and inconvenient, especially in border regions and the periphery of larger cities. This leaves a gap for transportation methods that are convenient and fast like cars but also reduce resource usage by grouping passengers with similar destinations like public transit. Recent developments in autonomous vehicles increase the attractiveness of *taxi sharing* systems in which a fleet of taxi-like vehicles is intelligently controlled to transport travelers without fixed stops or schedules. These systems attempt to bundle riders and maximize the usage of each vehicle's capacity for more resource efficient journeys compared to private cars or traditional taxis. The advantages of

such systems have been extensively studied in numerous simulation studies [4, 29, 1, 16, 44, 49] and real-world field tests [20, 28, 46, 43, 25, 41, 18, 47]. The advent of autonomous vehicles and a focus on more sustainable transportation are predicted to expedite the adoption of taxi sharing [16, 17, 35, 15, 3, 40, 46].

Taxi sharing could further be improved upon by allowing riders to transfer between vehicles during their journey. This additional option may allow the vehicle dispatcher to reduce vehicle operation times and increase the occupancy rates of vehicles without negatively affecting rider trip times. Thus, transfers may provide both economical and ecological benefits to taxi sharing systems. However, current studies into taxi sharing largely lack the option of transfers due to large computation times. Taxi sharing is already a difficult problem without transfers [33, 38] but transfers lead to an even more complex problem as the number of possible assignments of a rider to one or more vehicles increases combinatorially.

Based on recent advances in efficient dynamic taxi sharing without transfers [7, 31], we propose the first exact dispatching algorithm for dynamic taxi sharing with transfers that is able to scale to realistic city-scale input instances. For this purpose, we extend the model for traditional taxi sharing to allow journeys with at most one transfer. We find that a main issue of dynamic taxi sharing with transfers is the exploding number of shortest-path (SP) distances in the road network that need to be known to choose the best assignment for a rider. We focus on computing these distances on-the-fly when a rider request comes in, as this serves as the basis of a dynamic dispatcher that uses exact and up-to-date travel time information. We analyze the SP problems at hand and employ state-of-the-art speedup techniques for SPs in road networks to efficiently solve them. Also, we propose techniques to prune the number of assignments that need to be considered and explore both instruction-level and thread-level parallelization. In addition to an exact algorithm, we describe a heuristic approach that sacrifices some solution quality for improved running times.

In an experimental evaluation, we show that the proposed measures lead to viable dispatching times for realistic request sets on the road network of Berlin, Germany. We give first indications that transfers improve vehicle operation times and occupancy rates at the cost of slightly increased rider trip times. Our approach lays the groundwork for more precise studies of taxi sharing with transfers on large urban road networks with realistic request sets. This analysis is left to future work in cooperation with application experts.

1.1 Related Work

Taxi sharing and closely related problems like *ride matching* have seen considerable attention in the last decade. We provide a short overview with a focus on transfers. A more detailed summary of work on dynamic taxi sharing in general can be found in [31].

Taxi Sharing. Taxi sharing describes the problem of dispatching a fleet of taxi-like vehicles to transport travelers that request to travel from an origin to a destination location. Unrelated riders with similar destinations may be assigned to the same vehicle to reduce vehicle operation costs. The dispatcher has to choose assignments that optimize rider trip times and the usage of vehicle resources. Additional time constraints ensure user-friendly journeys.

Taxi sharing is closely related to the well-studied *Dial-a-Ride problem* [10, 21]. As the static variant of the DARP is known to be NP-complete (e.g. [39]), only small instances can be solved optimally [21, 9, 2]. Many heuristics have been developed to provide solutions in acceptable runtime on realistic instances, while giving up optimality [37, 26, 32]. While most research is conducted on the static variant, where all riders and requests are known in advance, we consider the *dynamic* taxi-sharing problem, where requests are served as soon

as they are issued, without knowledge of future requests. Due to the online nature of the problem, we implement a simple so-called insertion heuristic [24, 21] which greedily chooses a vehicle for a rider immediately upon receiving the request based on the current route state. Insertion heuristics are efficient and have been shown to perform reasonably well for the dynamic problem [36]. However, we are not aware of any in-depth experimental studies that compare online solutions to offline solutions of the same set of requests.

Most existing approaches assume that shortest paths in the road network (which are needed to assess vehicle detours) are already known. However, travel times in road networks change frequently, e.g. due to congestion. Thus, it is unreasonable to precompute all shortest paths in a road network as this information quickly becomes outdated. The existing state-of-the-art dispatchers for the dynamic taxi-sharing problem, LOUD [7] and its extension KaRRi [31], solve the problem by computing shortest paths on-the-fly, i.e., when a request is issued. The dispatchers combine state-of-the-art routing algorithms with pruning techniques based on constraints of existing vehicle routes to speed up distance computation. By using so-called customizable variants of these routing algorithms, updated information on travel times in the road network can be introduced periodically.

Taxi Sharing with Transfers. There has not been much work on dynamic taxi sharing with transfers. Most approaches consider a fixed set of transfer points that is known in advance (e.g. charging stations for electric vehicles) and find solutions using mixed-integer programming [23, 42, 8]. Again, shortest distances between vertices are assumed to be known.

The approach that is most closely related to our work is an extension of the LOUD dispatcher that locates feasible transfer points based on three different heuristics [45]. Their results show a reduction in total operation cost due to transfers. Note, though, that the cost model differs from the one we use since vehicle wait times are not considered part of a vehicle's detour and rider trip times are only taken into account as constraints.

To the best of our knowledge, there are no existing approaches for dynamic taxi sharing with optimal transfers that are able to scale to realistic instances of large urban areas.

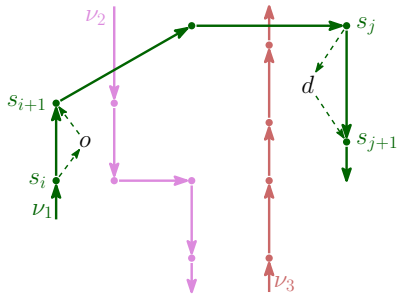
2 Problem Statement

This section describes the formal foundations for the dynamic taxi sharing problem without transfers and provides an extension of the model that allows the incorporation of transfers.

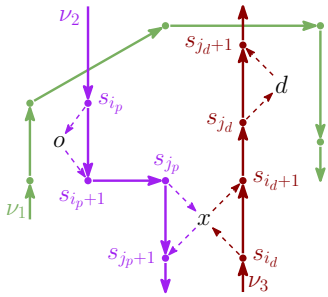
Road Network. We consider a *road network* to be a directed graph $G = (V, E)$. Road segments are represented as edges and intersections are represented as vertices. For every edge $e = (v, w) \in E$ we define an edge weight $\ell(e) = \ell(v, w)$ which is the travel time of the road segment. We denote the shortest-path distance between vertices $v, w \in V$ as $\delta(v, w)$.

Vehicles and Stops. Our algorithm manages the schedules for a fleet F of *vehicles*. Each vehicle has a seating capacity $cap(\nu)$ and a service time interval $[t^{min}_{serv}, t^{max}_{serv}]$ in which it operates. The current route $R(\nu) = \langle s_0(\nu), \ldots, s_{k(\nu)}(\nu) \rangle$ of a vehicle ν is a sequence of stops. Each stop s_i is mapped to a location $loc(s_i) \in V$ in the network[1]. After arriving at a stop,

[1] Our implementation actually maps each stop to an edge $e = (u, v) \in E$ in the road network.We make sure that the vehicle travels the length of e from u to v to allow a pickup or dropoff anywhere along the edge. However, we set the time of arrival to the time when v is reached, i.e., we do not actually route with intra-edge precision. To streamline the notation, we simplify locations to vertices in the paper.



(a) No-transfer insertion (r, ν_1, i, j).



(b) Transfer insertion $(r, x, \nu_2, \nu_3, i_p, j_p, i_d, j_d)$.

Figure 1 Illustration of routes of three vehicles ν_1, ν_2, and ν_3 and a request $r = (o, d, t_{req})$. Full arrows show current routes while dashed arrows denote detours made for a possible insertion. Figure 1a depicts an insertion without transfer using ν_1. Figure 1b illustrates a single-transfer insertion using pickup vehicle ν_2 and dropoff vehicle ν_3 with a transfer at x.

or when a new stop is scheduled, the route is updated, s.t. the vehicle's location is always between its previous (or current) stop $s_0(\nu)$ and the next stop $s_1(\nu)$. The number of stops yet to reach is $k(\nu) = |R(\nu)| - 1$. We denote the currently scheduled arrival time of vehicle ν at stop s_i as $t^{min}_{arr}(s_i(\nu))$ and the departure time of vehicle ν at stop s_i as $t^{min}_{dep}(\nu)$. If sufficient context is provided, we may write s_i instead of $s_i(\nu)$ and s_i instead of $loc(s_i)$.

Request, (No-Transfer) Insertion. In the *dynamic* taxi-sharing problem, a ride request is immediately assigned to a vehicle. A ride request $r = (orig, dest, t_{req})$ has an origin location $orig \in V$, a destination $dest \in V$ and a request time t_{req} at which the request is issued. We do not allow pre-booking, so the earliest possible time of departure is the request time.

For every request r, the dispatcher assigns r to a vehicle ν by constructing a *(no-transfer) insertion* $\iota = (r, \nu, i, j)$ with $0 \leq i, j \leq k(\nu)$. For an insertion $\iota = (r, \nu, i, j)$, the vehicle ν performs the pickup immediately after stop s_i and the dropoff immediately after stop s_j. For this, the vehicle leaves its scheduled route after stop s_i and s_j to pick up and drop off the rider at *orig* and *dest*, respectively, before returning to the next scheduled stop s_{i+1} and s_{j+1} (if $i < k(\nu)$ and $j < k(\nu)$, respectively). Figure 1a illustrates a no-transfer insertion.

Cost Function and Constraints. To evaluate the quality of insertions, we define the cost $c(\iota)$ of an insertion $\iota = (r, \nu, i, j)$ as a linear combination

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

To define the components of the cost function, first assume that the request $r = (orig, dest, t_{req})$ has already been inserted according to ι. Let s_p be the stop introduced to pick up the rider at *orig*, and let s_d be the stop introduced to drop the rider off at *dest*. Then, the route of ν after the insertion is $R'(\nu) = \langle s_0, \ldots, s_i, s_\mathrm{p}, s_{i+1}, \ldots, s_j, s_\mathrm{d}, s_{j+1}, \ldots, s_{k(\nu)} \rangle$. Let $t^{min\,\prime}_{arr}(s_i)$ and $t^{min\,\prime}_{dep}(s_i)$ describe the scheduled arrival and departure times in $R'(\nu)$.

The *vehicle detour* $t_{detour}(\iota)$ denotes the total additional operation time for ν caused by ι, i.e., $t_{detour}(\iota) = t^{min\,\prime}_{dep}(s_{k(\nu)}) - t^{min}_{dep}(s_{k(\nu)})$. This is equivalent to the sum of detours made, i.e., $\delta(s_i, orig) + \delta(orig, s_{i+1}) - \delta(s_i, s_{i+1}) + \delta(s_j, dest) + \delta(dest, s_{j+1}) - \delta(s_j, s_{j+1})$.

The *trip time* $t_{trip}(\iota)$ is the total travel time for the new rider from issuing the request to their arrival at the destination, i.e., $t_{trip}(\iota) = t^{min\,\prime}_{arr}(s_\mathrm{d}) - t_{req}$. The trip times of existing riders may also increase due to detours. The *added trip* time $t^{+}_{trip}(\iota)$ is the sum of all such changes for riders of ν. Model parameter τ determines the importance of trip times.

We aim to limit riders' maximum wait and trip times using constraints. A rider's trip should not take longer than their maximum trip time $t_{trip}^{max}(r) = \alpha \cdot \delta(orig, dest) + \beta$. A rider should not wait to be picked up longer than $t_{wait}^{max} \in \mathbb{R}_{\geq 0}$. The values $\alpha, \beta, t_{wait}^{max}$ are model parameters. For existing riders, these constraints are *hard*, i.e., if an insertion violates them, its cost is ∞. For the new rider, the constraints are *soft* and only incur penalties $c_{wait}^{vio}(\iota)$ and $c_{trip}^{vio}(\iota)$ to the insertion cost. This allows the system to serve every request.

Note that to compute the updated route $R'(\nu)$ with $t_{arr}^{min\,\prime}$ and $t_{dep}^{min\,\prime}$, as well as the cost of the insertion, we generally need to know the distance $\delta(s_i, orig)$ from s_i to *orig*, the distance $\delta(orig, s_{i+1})$ from *orig* to the following stop (if $i < k(\nu)$), as well as the distance $\delta(s_j, dest)$ from s_j to *dest* and the distance $\delta(dest, s_{j+1})$ from *dest* to the following stop (if $j < k(\nu)$). If $i = j$, we additionally need to know $\delta(orig, dest)$. It is one of the main challenges of dynamic taxi sharing to solve the according shortest-path problems.

Single-Transfer Insertions. In this work, we focus on efficiently finding optimal *single-transfer journeys*, where a rider changes vehicles at most once. This extension induces a significant combinatorial increase in the number of possible insertions compared to the traditional case without transfers, which poses the main challenge of our work. The taxi-sharing model described above can be easily modified for single-transfer journeys.

Single-transfer insertions take the form $\iota_{transfer} = (r, x, \nu_p, \nu_d, i_p, j_p, i_d, j_d)$. The *pickup vehicle* ν_p picks up the new rider at *orig* immediately after stop $s_{i_p}(\nu_p)$ and drops them off at the transfer location x immediately after stop $s_{j_p}(\nu_p)$. The *dropoff vehicle* ν_d picks the rider up at x for the second leg of the trip immediately after stop $s_{i_d}(\nu_d)$ and drops them off at *dest* immediately after stop $s_{j_d}(\nu_d)$. A single-transfer insertion is illustrated in Figure 1b.

2.1 Cost Computation of Single-Transfer Insertions

The structure of the cost function remains the same for transfer insertions, and transfer insertions are subject to the same constraints as before. Computing the cost of a transfer insertion requires us to know the distances between existing stops and the transfer point in addition to the distances for *orig* and *dest*. Compared to no-transfer insertions, this increases the amount of work that needs to be spent solving shortest-path problems. In the following, we describe some additional intricacies that come with transfers.

Riders Waiting at the Transfer Location. The definition of the trip time $t_{trip}(\iota_{transfer})$ of the new rider as well as the trip time violation $c_{trip}^{vio}(\iota_{transfer})$ remain unchanged. However, the wait time of the new rider now also incorporates the time that the rider spends waiting at the transfer location for the dropoff vehicle, which potentially has an impact on the wait time soft constraint $c_{wait}^{vio}(\iota_{transfer})$.

Vehicles Waiting at the Transfer Location. After processing a transfer insertion where the dropoff vehicle arrives at the transfer location x sooner than the pickup vehicle, the dropoff vehicle has to wait at x for the arrival of the transferring rider. Thus, vehicles may now have wait times at stops along their routes, which impact the way detours affect a vehicle's total operation time. Assume vehicle ν has a wait time at stop s_w. Then, making a detour before s_w delays the arrival of ν at s_w as much as before, but since the vehicle would have waited some time at s_w, the delay in the departure time may be smaller. Effectively, to compute the change in operation time, we can subtract the wait times at stops from the actual detours made since the time that would have been spent waiting is spent driving instead. The increase in operation time for any affected vehicle can still be characterized

as $t_{dep}^{min\,\prime}(s_{k(\nu)}) - t_{dep}^{min}(s_{k(\nu)})$ but the computation of $t_{dep}^{min\,\prime}(s_{k(\nu)})$ becomes more complex. Vehicle wait times similarly affect the added trip time of existing riders as the updated arrival times $t_{arr}^{min\,\prime}(s)$ now have to take vehicle wait times into account. The authors of KaRRi have previously encountered the issue of vehicle wait times in the context of *meeting points*. The full paper on KaRRi [30] gives a detailed explanation on how vehicle wait times affect the computation of updated schedules and the resulting cost terms.

Dependencies between Vehicle Schedules. Transfers introduce dependencies between vehicles' schedules. As described in the previous paragraph, a dropoff vehicle can only leave a transfer point once the transferring rider arrives in the pickup vehicle. Thus, any delay to the arrival of the pickup vehicle at the transfer stop may also delay the departure of the dropoff vehicle. In effect, vehicles other than the pickup or dropoff vehicle can also be affected by an insertion due to previously introduced transfers. To account for this, we explicitly memorize the dependency between pickup and dropoff vehicle whenever a transfer insertion is performed. Then, when computing the cost for a new insertion $\iota_{transfer} = (r, x, \nu_p, \nu_d, i_p, j_p, i_d, j_d)$, we find any vehicles that depend on ν_p or ν_d, and potentially propagate the detours caused by $\iota_{transfer}$ to their routes. For any such dependent vehicle, we obtain an added operation time and added trip time for existing riders, which we consider in the total cost of $\iota_{transfer}$.

3 Preliminaries

In this section, we explain the shortest-path algorithms used in this work, as well as the existing algorithms for taxi sharing without transfers that we base our work on.

3.1 Shortest-Path Algorithms

Here, we summarize the shortest-path techniques most relevant to this paper.

Dijkstra's Algorithm. Dijkstra's algorithm [14] serves as the basis of many shortest-path algorithms. Given a directed graph $G = (V, E)$, a weight function $\ell : E \to \mathbb{R}_{\geq 0}$, and a source vertex $s \in V$, the algorithm computes the shortest path w.r.t. ℓ from s to every $v \in V$. The algorithm maintains a distance label $d[v]$ for $v \in V$ as well as a priority queue Q of vertices. The key of a vertex v in Q is $d[v]$. Initially, $d[s] := 0$, $d[v] := \infty$ for $v \neq s$, and $Q := \{s\}$. The algorithm proceeds by removing the vertex v with the smallest $d[v]$ from Q and *settling* it. To settle v, all edges $(v, w) \in E$ are *relaxed*. To relax an edge $e = (v, w)$, the algorithm checks whether $d[v] + \ell(e) < d[w]$. If so, the path to w via v is now the best known path to w and $d[w]$ is updated to $d[v] + \ell(e)$. Then, w is inserted into Q or its key is updated. When a vertex v is settled, its tentative distance $d[v]$ is equal to the shortest-path distance $\delta(s, v)$.

Contraction Hierarchies. Contraction Hierarchies (CH) [19] are a speedup technique for the computation of shortest paths in road networks that leverage the inherent hierarchy of road networks. Every shortest-path algorithm employed in this work is ultimately based on CHs. The CH algorithm works in two phases, a pre-processing phase and a query phase.

In the preprocessing phase, each vertex $v \in V$ of a road network $G = (V, E)$ is heuristically assigned a unique *rank* representing the vertex's importance. Higher ranks are assigned to more important vertices. Vertices are then contracted in order of increasing rank. To contract a vertex v, it is removed from the graph. To preserve shortest paths, a *shortcut edge* (u, w) with $\ell(u, w) = \ell(u, v) + \ell(v, w)$ is created if (u, v, w) is the shortest path from u

to w. After contracting all vertices, the original graph is restored and augmented with all shortcut edges created in the contraction process. Let E^+ be the set containing all original edges E and all shortcut edges. The graph $G^+ = (V, E^+)$ constitutes the CH. For the query phase, we partition E^+ into up-edges $E^\uparrow = \{(u, v) \in E^+ \mid \text{rank}(u) < \text{rank}(v)\}$ and down-edges $E^\downarrow = \{(u, v) \in E^+ \mid \text{rank}(u) > \text{rank}(v)\}$. We define an upward search graph $G^\uparrow = (V, E^\uparrow)$ and a downward search graph $G^\downarrow = (V, E^\downarrow)$. Let $\delta^\uparrow(u, v)$ and $\delta^\downarrow(u, v)$ denote the shortest-path distance from u to v in $G^\uparrow$ and $G^\downarrow$, respectively.

In a point-to-point CH query, a shortest path from $s \in V$ to $t \in V$ is found, using the fact that for any $u, v \in V$ there is a shortest path from u to v that consists of only up-edges followed by only down-edges [19]. Running a forward Dijkstra search in $G^\uparrow$ from s and a reverse Dijkstra search in $G^\downarrow$ rooted at t suffices to find the shortest up-down path.

PHAST. PHAST [11] is a CH-based speedup technique for the one-to-all shortest-path problem in road networks. PHAST uses a CH as well as a specific memory layout to linearize the process of settling vertices and relaxing edges in memory. It proceeds in two phases.

First, given a source vertex $s \in V$, PHAST runs a forward search in $G^\uparrow$ rooted at s, exploring the entire search space and initializing a distance $d[v] := \delta^\uparrow(s, v)$ at every settled vertex v. Every vertex not settled gets $d[v] := \infty$. Second, PHAST settles every $v \in V$ in decreasing order of CH rank, propagating the distances from the forward search through $G^\downarrow$ by relaxing incoming edges in $E^\downarrow$. This finds shortest up-down paths to every $v \in V$. Scanning vertices in top-down order ensures that $d[u]$ is finished before $d[v]$ for $(u, v) \in E^\downarrow$.

PHAST reorders the vertices in $G^\downarrow$ according to the order in which vertices are scanned in the top-down sweep. Thus, the write operations to $d[v]$ during the sweep are in sequential order and reads of $d[u]$ for relaxed edges $(u, v) \in E^\downarrow$ are likely to hit the cache.

PHAST leverages both instruction parallelism and multi-threading for additional speedups. Instruction parallelism is applicable if there are $k > 1$ sources. Then, $d[v]$ is a distance vector of width k where the i-th entry refers to the distance from the i-th source to v. Edge relaxations can use vector instructions to update the distance for all k sources simultaneously. To utilize multi-threading, PHAST groups vertices into CH levels such that vertices within the same level can be settled in parallel (for details, see [11]).

CH-based One-to-Many Queries. There are two main ways to use CHs for one-to-many queries from a source $s \in V$ to each $t \in T$ for a set of targets $T \subseteq V$. Both approaches use a *target selection phase* followed by a query phase. *Bucket Contraction Hierarchies (BCH)* [19, 27] run reverse Dijkstra searches in $G^\downarrow$ from every $t \in T$ that memorize every $\delta^\downarrow(v, t)$ in a *bucket* at vertex v (selection). Then, a forward query in $G^\uparrow$ rooted at s can use these buckets to find shortest paths in the CH (query). *RPHAST* [12] computes a subgraph $H^\downarrow$ of $G^\downarrow$ that contains all vertices from which any $t \in T$ can be reached using only edges in $E^\downarrow$ (selection). Then, the top-down sweep of a PHAST query rooted at s can be restricted to $H^\downarrow$ and still find the shortest path to each $t \in T$ (query).

3.2 The LOUD and KaRRi Taxi-Sharing Dispatchers

KaRRi [31] is an algorithm for the dynamic taxi-sharing problem without transfers that acts as the basis of our solution to the problem with transfers. Here, we describe how KaRRi and its predecessor LOUD [7] use engineered routing techniques to dispatch requests efficiently.

Elliptic Pruning. LOUD [7] uses constraints on vehicle routes for faster shortest-path queries between vehicle stops and a rider's origin and destination. As described in Section 2, every rider induces hard constraints for maximum wait time and trip time on a vehicle.

These constraints define a latest permissible arrival time $t^{max}_{arr}(s)$ for the stops s of the respective vehicle. Let $t^{min}_{dep}(s_i)$ be the scheduled departure time at stop s_i. Then, a vehicle ν may take a time of at most $\lambda(s_i, s_{i+1}) = t^{max}_{arr}(s_{i+1}) - t^{min}_{dep}(s_i)$ to travel from s_i to s_{i+1} without breaking any rider's constraint. We call the value $\lambda(s_i, s_{i+1})$ the *leeway* between s_i and s_{i+1}. An origin location *orig* can only be inserted between stops s_i and s_{i+1} if $\delta(s_i, orig) + \delta(orig, s_{i+1}) \leq \lambda(s_i, s_{i+1})$ (analogous for destination locations). The set $\mathcal{E}(s_i) = \{u \in V \mid \delta(s_i, u) + \delta(u, s_{i+1}) \leq \lambda(s_i, s_{i+1})\}$ is called the *detour ellipse* of s_i.

LOUD uses BCH searches to compute the distances from every vehicle stop to the origin/destination of a request and vice versa. The authors of LOUD find that bucket entries for a stop s_i only need to be generated at certain vertices within $\mathcal{E}(s_i)$ to find all relevant distances. This approach reduces the number of bucket entries that need to be scanned and restricts the set of pickup or dropoff vehicles to those seen during the queries.

Last-Stop Queries. It is necessary for taxi sharing to also allow insertions that append new stops at the end of a vehicle's route instead of only inserting new stops in between existing stops. We can utilize BCH searches to compute the required distances between the last stops of every vehicle route and the origin and destination location of a new rider. However, since a vehicle's last stop has no following stop, there is no leeway and no detour ellipse to employ elliptic pruning. Instead, KaRRi [31] uses sorted BCH buckets and lower bounds on the cost of insertions to speed up this one-to-many shortest-path computation.

4 Algorithm Overview

We describe the use of detour ellipses for transfer insertions and identify two distinct types of transfer insertions. Based on this, we give an overview of the structure of our algorithm.

4.1 Detour Ellipses for Transfers

The central challenge of computing single-transfer insertions is the fact that the pickup vehicle ν_p and the dropoff vehicle ν_d can move freely in the road network, which makes every location in the road network a potential transfer point. Without further limitations, the number of transfer insertions that need to be checked would be at least linear in the size of the road network. However, we can reduce the number of viable transfer locations for each request by taking into account the constraints on vehicle detours imposed by existing riders.

As mentioned in Section 3.2, the constraints on vehicle routes induce a detour leeway $\lambda(s_i, s_{i+1})$ between any two consecutive stops s_i and s_{i+1}. This leeway defines the ellipse $\mathcal{E}(s_i) \subseteq V$ containing all locations to which a detour between s_i and s_{i+1} can be made without breaking any constraints. Thus, for any feasible transfer insertion $(r, x, \nu_p, \nu_d, i_p, j_p, i_d, j_d)$, the transfer point x must lie in $\mathcal{E}(s_{j_p}(\nu_p))$ if $j_p < k(\nu_p)$ and in $\mathcal{E}(s_{i_d}(\nu_d))$ if $i_d < k(\nu_d)$. If we know the detour ellipses of stop pairs along the routes of relevant vehicles, we can deduce a limited set of viable transfer locations for which transfer insertions need to be constructed.

In the following, we describe how detour ellipses can be used to enumerate transfer insertions of different types. For now, we assume that all necessary detour ellipses are known. We explain how to compute a detour ellipse on-the-fly in Section 5.1. We describe how to compute the necessary shortest-path distances in Sections 5.2 and 5.3.

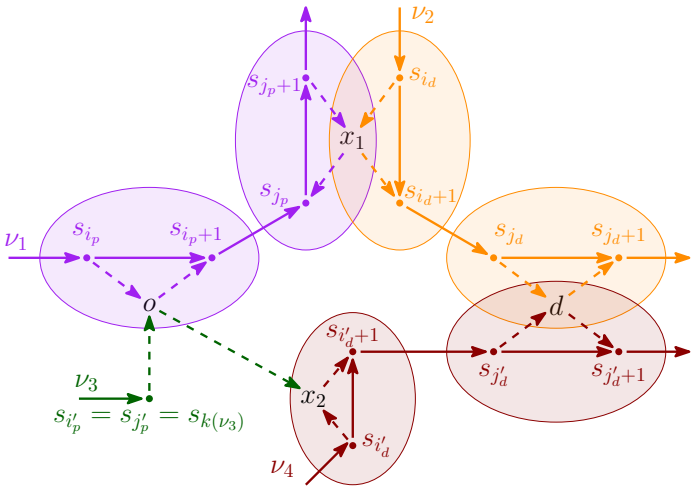


Figure 2 Illustration of routes of four vehicles ν_1, ν_2, ν_3, and ν_4, and a request $r = (o, d, t_{req})$. Depicts an ordinary transfer insertion $(r, x_1, \nu_1, \nu_2, i_p, j_p, i_d, j_d)$ and an after-last-stop transfer insertion $(r, x_2, \nu_3, \nu_4, k(\nu_3), k(\nu_3), i'_d, j'_d)$. Shaded areas indicate detour ellipses.

4.2 Types of Transfers

We distinguish between two types of transfer insertions that differ in how detour ellipses constrain the set of potential transfer points.

Ordinary Transfers. An *ordinary transfer* insertion is any insertion $(r, x, \nu_p, \nu_d, i_p, j_p, i_d, j_d)$ where both $j_p < k(\nu_p)$ and $i_d < k(\nu_d)$, i.e., for which the transfer point x is inserted before the last stop in the routes of both the pickup vehicle and the dropoff vehicle. Let $F_p(r) = \{(\nu, i) \in F \times \mathbb{N} \mid i \in \{0, \dots, k(\nu)\}$ and $orig \in \mathcal{E}(s_i(\nu))\}$ denote the set of vehicles and associated stops in the vehicle route such that ν can perform a pickup of r at $orig$ immediately after stop s_i without breaking any of the vehicle's constraints. Analogously, let $F_d(r) = \{(\nu, j) \in F \times \mathbb{N} \mid j \in \{0, \dots, k(\nu)\}$ and $dest \in \mathcal{E}(s_j(\nu))\}$ be the set of candidate dropoff vehicles and associated stops.

In ordinary transfer insertions, the transfer point needs to be contained in the intersection of two ellipses $\mathcal{E}(s_{j_p}(\nu_p)) \cap \mathcal{E}(s_{i_d}(\nu_d))$. Thus, for any $(\nu_p, i_p) \in F_p(r)$ and any $(\nu_d, j_d) \in F_d(r)$, every insertion $(r, x, \nu_p, \nu_d, i_p, j_p, i_d, j_d)$ for every $j_p = i_p, \dots, k(\nu_p) - 1$, every $i_d = 0, \dots, j_d$, and every $x \in \mathcal{E}(s_{j_p}(\nu_p)) \cap \mathcal{E}(s_{i_d}(\nu_d))$ is feasible. Figure 2 shows an ordinary transfer insertion using vehicle ν_1 and ν_2 and a transfer after stops s_{j_p} and s_{i_d}. Any point in the overlap of the orange and purple ellipses is a viable transfer location for these vehicles and stops.

After-Last-Stop (ALS) Transfers. Any transfer insertion which is not ordinary has either $j_p = k(\nu_p)$ or $i_d = k(\nu_d)$. This means, either the pickup vehicle brings the new rider to the transfer location after its current last stop $s_{k(\nu_p)}$ or the dropoff vehicle picks up the new rider at the transfer location after its current last stop $s_{k(\nu_d)}$. Therefore, we call these insertions *after-last-stop (ALS) transfer* insertions. Note that transfer insertions with both $j_p = k(\nu_p)$ and $i_d = k(\nu_d)$ will always have a higher cost than the non-transfer insertion $(r, \nu_p, i_p, k(\nu))$ in our model, so we do not have to consider this case.

Focus on the case $j_p = k(\nu_p)$ and $i_d < k(\nu_d)$. In this case, the pickup vehicle is not bound by any detour ellipse since it will have already dropped off all other passengers when it reaches s_{j_p}. Thus, there are no rider constraints at this point. However, the viable transfer points

Algorithm 1 Algorithm Outline. Comments indicate description of implementation.

Input: $r = (orig, dest, t_{req})$ **Output:** best insertion
$\iota_{\text{none}} := \text{KaRRi}(r)$ ▷ no-transfer solution [31]
$F_p(r), F_d(r) := \text{findPickupAndDropoffVehicles}(r)$ ▷ Section 5.2
$\mathbb{E} := \text{computeDetourEllipses}(F_p(r), F_d(r))$ ▷ Section 5.1
$\iota_{\text{ord}} := \text{findBestOrdinaryTransferInsertion}(r, F_p(r), F_d(r), \mathbb{E})$ ▷ Section 5.2
$\iota_{\text{alsp}} := \text{findBestALSTransferInsertionPVeh}(r, F_p(r), F_d(r), \mathbb{E})$ ▷ Section 5.3
$\iota_{\text{alsd}} := \text{findBestALSTransferInsertionDVeh}(r, F_p(r), F_d(r), \mathbb{E})$ ▷ Section 5.3
return $\text{argmin}_{\iota \in \{\iota_{\text{none}}, \iota_{\text{ord}}, \iota_{\text{alsp}}, \iota_{\text{alsd}}\}} c(\iota)$

are still limited to the detour ellipse $\mathcal{E}(s_{i_d}(\nu_d))$. Therefore, for any vehicle that can perform a pickup at any index i_p along its route and any $(\nu_d, j_d) \in F_d(r)$, the set of feasible insertions contains $(r, x, \nu_p, \nu_d, i_p, k(\nu_p), i_d, j_d)$ for every $i_d = 0, \ldots, j_d$, and every $x \in \mathcal{E}(s_{i_d}(\nu_d))$. Note that this may include the case of $i_p = k(\nu_p)$, i.e., a *paired* ALS insertion.

Analogously, for any dropoff vehicle ν_d that can perform a dropoff after its last stop $s_k(\nu_d)$ and any $(\nu_p, i_p) \in F_p$, the insertion $(r, x, \nu_p, \nu_d, i_p, j_p, k(\nu_d), k(\nu_d))$ is feasible for every $j_p = i_p, \ldots, k(\nu_p) - 1$, and every $x \in \mathcal{E}(s_{j_p}(\nu_p))$.

In Figure 2, a paired ALS transfer insertion is depicted where vehicle ν_3 extends its route after its last stop to pick up the rider at o and bring them to a transfer location x_2 within the ellipse $\mathcal{E}(s_{i'_d}(\nu_4))$. Any location within this ellipse would be a feasible choice for x.

4.3 Structure of Algorithm

Whenever a new request $r = (orig, dest, t_{req})$ is issued, the dispatching algorithm is started with the current route state. The best insertion returned by the algorithm is used to update the route state before the next request is processed.

Our dispatching algorithm is outlined in Algorithm 1. We always allow a rider to use no-transfer or transfer insertions. Thus, to start with, we use the base KaRRi algorithm to find the best no-transfer insertion. Then, we compute the sets $F_p(r)$ and $F_d(r)$ of potential pickup and dropoff vehicles. We compute the detour ellipse for every stop along the routes of these vehicles where a transfer may be made as described in the previous section. Subsequently, we enumerate all ordinary transfer insertions, all ALS transfer insertions for pickup vehicles, and all ALS transfer insertions for dropoff vehicles.

5 Exact Transfer Points

In this section, we describe how we can efficiently implement each step of the algorithm mentioned in Section 4.3 to find a locally exact solution to dynamic taxi sharing with transfers. More precisely, for each request $r = (orig, dest, t_{req})$, we aim to find the single-transfer insertion $(r, x, \nu_p, \nu_d, i_p, j_p, i_d, j_d)$ with the smallest total cost (at the time of dispatching r).

5.1 Computing Detour Ellipses On-the-Fly

Finding exact transfer insertions requires us to compute the detour ellipses $\mathcal{E}(s_i)$ of many stop pairs (s_i, s_{i+1}). To find out whether a vertex $v \in V$ lies within $\mathcal{E}(s_i)$ we need to know the distances $\delta(s_i, v)$ and $\delta(v, s_{i+1})$ in order to check if $\delta(s_i, v) + \delta(v, s_{i+1}) \leq \lambda(s_i, s_{i+1})$. In effect, we need to compute the distances $\delta(s_i, v)$ and $\delta(v, s_{i+1})$ for every $v \in V$.

These two one-to-all shortest path problems can simply be solved by running a forward Dijkstra search rooted at s_i and a reverse Dijkstra search rooted at s_{i+1}. The Dijkstra searches can be stopped once a vertex $v \in V$ with $d[v] > \lambda(s_i, s_{i+1})$ is settled. During the searches, we memorize which vertices have been settled. For every vertex settled by both searches, we check whether $\delta(s_i, v) + \delta(v, s_{i+1}) \leq \lambda(s_i, s_{i+1})$ to determine if $v \in \mathcal{E}(s_i)$.

As an alternative, we can use the one-to-all speedup technique PHAST (see Section 3.1). We run a forward PHAST search rooted at s_i as well as a reverse PHAST search rooted at s_{i+1}. Then, we check whether $v \in \mathcal{E}(s_i)$ for every $v \in V$ using the computed distances. Note that PHAST queries cannot be pruned using $\lambda(s_i, s_{i+1})$ like the Dijkstra searches.

As proposed by the authors of PHAST, the queries can be accelerated using both instruction- and thread-level parallelism (cf. Section 3.1). In contrast to PHAST, it is notoriously difficult to apply multi-threading to speed up Dijkstra queries with good scalability [34]. Similarly, bundling Dijkstra queries for vectorized edge relaxations only works well if the sources are close to each other in the graph and thus have overlapping shortest-path trees. Unfortunately, this is not the case for arbitrary stops.

Note that both approaches provide the shortest-path distances between vehicle stops and transfer points that are later required to compute the cost of an insertion.

5.2 Optimal Ordinary Transfers

To find all ordinary transfers, we need to compute the intersection of the detour ellipses of stops of pickup vehicles in $F_p(r)$ and dropoff vehicles in $F_d(r)$, as described in Section 4.2.

The LOUD no-transfer dispatcher provides a way to compute the sets $F_p(r)$ and $F_d(r)$ of potential pickup and dropoff vehicles. For both *orig* and *dest*, we run a forward and reverse BCH search that identifies the vehicles and stops at which a detour can be made to perform the pickup or dropoff. Elliptic pruning speeds up these queries and limits the size of $F_p(r)$ and $F_d(r)$ (cf. Section 3.2). These BCH searches also provide the distances needed to compute the detour made for the pickup and the dropoff.

To facilitate intersecting ellipses, we sort every ellipse by vertex ID. Then, any intersection $\mathcal{E}(s_{j_p}(\nu_p)) \cap \mathcal{E}(s_{i_d}(\nu_d))$ can be constructed with a linear sweep over $\mathcal{E}(s_{j_p}(\nu_p))$ and $\mathcal{E}(s_{i_d}(\nu_d))$.

If $i_p \neq j_p$ and $i_d \neq j_d$, the distances between stops and transfer points computed during the ellipse reconstruction suffice to calculate the cost of the insertion. In the case of a *paired* insertion, i.e., $i_p = j_p$ or $i_d = j_d$, we need to additionally know the distances from *orig* to the transfer point x or the distance from x to *dest*, respectively. For paired insertions, we first assume $\delta(\mathit{orig}, x) = 0$ or $\delta(x, \mathit{dest}) = 0$ and compute a lower bound for the cost of the insertion. If this lower bound is worse than the best known cost, we can safely discard it. Otherwise, we compute the distance $\delta(\mathit{orig}, x)$ or $\delta(x, \mathit{dest})$ using a point-to-point CH query.

5.3 Optimal After-Last-Stop Transfers

As described in Section 4.2, in an ALS insertion, a transfer point may be any location within the detour ellipse of the non-ALS vehicle. Here, we focus on the case that the pickup vehicle ν_p is the ALS vehicle and brings the rider to a transfer point in an ellipse of a dropoff vehicle ν_d. Then, we need to know the distances from the last stop $s_{k(\nu_p)}$ to every location in this ellipse. Extending this to all possible pickup vehicles in $F_p(r)$ and dropoff vehicles in $F_d(r)$, we get a many-to-many shortest path problem where the set of sources $\mathcal{L}$ contains all last stops of pickup vehicles, while the set of targets $\mathcal{T}$ is the union of all eligible ellipses, i.e.,

$$\mathcal{L} = \{s_{k(\nu_p)} \mid (\nu_p, \_) \in F_p(r)\}, \qquad \text{and} \qquad \mathcal{T} = \bigcup_{(\nu_d, j_d) \in F_d(r)} \bigcup_{i_d \in \{0, \ldots, j_d\}} \mathcal{E}(s_{i_d}(\nu_d)).$$

We utilize RPHAST for this problem. We run the selection phase once for $\mathcal{T}$ and then run a query from every $l \in \mathcal{L}$. We can bundle and vectorize these queries (cf. Section 3.1).

A pickup vehicle may also perform both the pickup and the trip to the transfer after its current last stop in a paired ALS insertion $(r, x, \nu_p, \nu_d, k(\nu_p), k(\nu_p), i_d, j_d)$. In this case, we need to know the distance from *orig* to every transfer point x. Thus, we run a single RPHAST query from *orig* to $\mathcal{T}$. Additionally, we need to identify vehicles ν_p that need to be considered for a pickup after their last stop as $F_p(r)$ is not guaranteed to contain all of them. For this purpose, we utilize BCH searches as proposed by KaRRi (cf. Section 3.2). We construct bucket entries for every last stop. Then, a single reverse BCH query rooted at *orig* computes the distances from all last stops to *orig*. We prune the set of viable vehicles by comparing cost lower bounds based on these distances to the best known cost.

Now consider the case that the dropoff vehicle goes to the transfer after its last stop while the pickup vehicle incorporates the transfer somewhere along its existing route. Then, the ALS insertion will always be paired since the dropoff must necessarily be performed after the transfer. Thus, we can use the same techniques outlined for paired ALS insertions above.

5.4 Speeding up Enumeration of Insertions

Computing the cost for a transfer insertion takes much longer than for a no-transfer insertion. Thus, we describe three ways to speed up the computation of insertions and their cost. We focus on ordinary insertions but all techniques are applicable to ALS insertions, too.

Cost Bounds. For each request r, our algorithm first finds the best no-transfer insertion. The cost of this no-transfer insertion serves as an upper bound for the best cost for r.

Consider a set of possible transfer insertions $(r, x, \nu_p, \nu_d, i_p, j_p, i_d, j_d)$ for fixed ν_p, ν_d, i_p, j_p, i_d, and j_d, and $x \in \mathcal{E}(s_{j_p}(\nu_p)) \cap \mathcal{E}(s_{i_d}(\nu_d))$. We can obtain a lower bound on the cost of any insertion in this set by applying the cost function with lower bounds on the distances from and to any transfer point x. If this lower bound cost is already worse than the best no-transfer cost, we do not have to consider any of the individual insertions in the set. To get lower bounds on the distances from and to any feasible transfer point, we simply memorize the smallest such distances seen while intersecting the ellipses.

Pareto-Dominance between Transfer Points. We find that many transfer points can never lead to a best insertion as there are other transfer points which are guaranteed to lead to better insertions. We devise a measure of pareto-dominance between transfer points within the same intersection that allows to exclude these dominated transfer points.

▶ **Definition 1.** *Let* $x_1, x_2 \in \mathcal{E}(s_{j_p}(\nu_p)) \cap \mathcal{E}(s_{i_d}(\nu_d))$. *Then,* x_1 dominates x_2 *if*

$$\delta(s_{j_p}, x_1) + \delta(x_1, s_{j_p+1}) < \delta(s_{j_p}, x_2) + \delta(x_2, s_{j_p+1}), \tag{1}$$
$$\delta(s_{i_d}, x_1) + \delta(x_1, s_{i_d+1}) < \delta(s_{i_d}, x_2) + \delta(x_2, s_{i_d+1}), \text{ and} \tag{2}$$
$$\delta(s_{j_p}, x_1) + \delta(x_1, s_{i_d+1}) < \delta(s_{j_p}, x_2) + \delta(x_2, s_{i_d+1}). \tag{3}$$

▷ Claim 2. If x_1 dominates x_2, then

$$c(\iota_1 = (r, x_1, \nu_p, \nu_d, i_p, j_p, i_d, j_d)) < c(\iota_2 = (r, x_2, \nu_p, \nu_d, i_p, j_p, i_d, j_d)).$$

Proof. See Section A. ◁

In road networks with heterogeneous travel speeds, there can easily be locations that are not well accessible to both the pickup and the dropoff vehicle (e.g., side roads within a neighborhood), which leads to them being dominated by locations on more easily accessible roads in the vicinity. Note that a test for domination between two transfer points can be computed quickly. Thus, it is worth filtering transfer points based on domination before performing the much more expensive calculation of insertion costs.

Parallelization. Computing the cost of all feasible insertions can be trivially parallelized. We iterate over pairs of pickup and dropoff vehicles in $F_p(r) \times F_d(r)$ in parallel with the same thread computing the cost for all insertions of one vehicle pair. Each thread keeps a thread-local best insertion seen. When all threads are done, the best of the thread-local insertions is chosen. Pareto-dominance and cost bounds can still be applied by each thread.

6 Heuristic Transfer Points

In this section, we describe a way to reduce running times by heuristically choosing a subset of transfer points based on CHs. CHs aim to order vertices by their importance for shortest paths in the road network during construction. Therefore, we can assume that vertices of high CH rank can be reached easily and may be good candidates for transfer points.

In our heuristic, we only consider the k percent of vertices in the network with the highest ranks as transfer points. Let $V_{sorted} = \langle v_1, \dots, v_n \mid v_j \in V, \text{rank}(v_j) \geq \text{rank}(v_{j+1}) \rangle$. Let $V_{k\%}$ denote the subset of the first $nk/100$ vertices. When computing the potential transfer points for a request, for every ellipse $\mathcal{E}(s_i)$, candidate vertices are now limited to $\mathcal{E}(s_i) \cap V_{k\%}$.

This restriction provides two advantages with regard to computation time. Firstly, the one-to-all PHAST queries used to reconstruct detour ellipses (see Section 5.1) can now stop after scanning only the top $k\%$ of vertices. Secondly, the average size of each restricted detour ellipse will be much smaller which reduces the number of transfer insertions that need to be tried. As a trade-off, the heuristic may negatively affect the solution quality since potentially good transfer locations that are not in the top $k\%$ of vertices may be missed. The parameter k allows an interpolation between the reduced running time and loss in quality.

7 Experimental Evaluation

We experimentally evaluate our approach on realistic input instances for dynamic taxi sharing. In this section, we refer to our approach as *KaRRiT* (KaRRi with Transfers).

7.1 Experimental Setup and Benchmark Instances

Our source code[2] is written in C++20 and compiled with GCC 11.5 using `-O3`. We use two machines for separate experiments, both running Rocky Linux 9.5. Machine A has 64 GiB of memory and a single 16-core AMD Ryzen 9 3950X processor at 3.5GHz. Machine B has 512 GiB of memory and a single 32-core Intel Xeon Gold 6314U processor at 2.3 GHz. We use 32-bit distance labels and the AVX2 SIMD instruction set with 256-bit registers to compute up to 8 operations in one vector instruction.

We evaluate KaRRiT on the `Berlin-1pct` (`B-1%`), and `Berlin-10pct` (`B-10%`) request sets [7] that, respectively, represent taxi-sharing demand for 1% and 10% of the population of the Berlin metropolitan area on a weekday. The request sets for Berlin were artificially

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

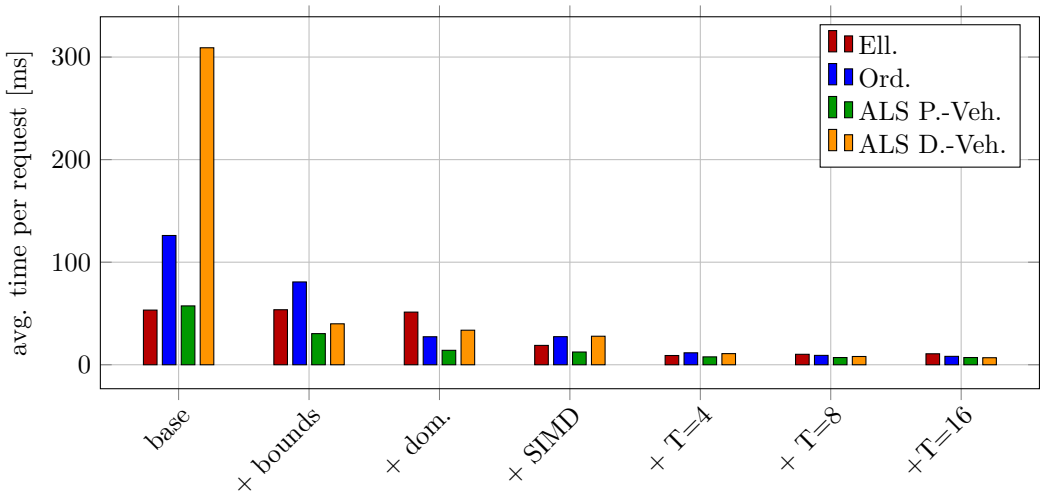


Figure 3 Average running times per request of main components of KaRRiT in incrementally more efficient configurations. The configuration "base" uses PHAST and RPHAST wherever possible but none of the other optimizations described in Section 5. The other configurations add the specified optimization to the configuration to their left. (The configurations "T=n" each add multi-threading with n threads to the "+ SIMD" configuration.)

generated using the Open Berlin Scenario [48] for the MATSim transport simulation [22][3]. Both sets cover a time window of 30 hours and follow a realistic distribution of demand on a weekday regarding both time and space. The `Berlin-1pct` and `Berlin-10pct` request sets contain 16569 and 149185 requests, respectively. Images on the temporal and spatial distribution can be found in Figure 4 in Section B. We use 1000 vehicles for `Berlin-1pct` and 10000 vehicles for `Berlin-10pct`. Each vehicle has a capacity of four and a service time interval covering the entire 30 hours. The initial locations of all vehicles are drawn uniformly at random. The underlying road network of Berlin and the surrounding area is obtained from publicly available OpenStreetMap data[4]. It contains 94422 vertices and 193212 edges. We use the known speed limit of each road to determine the travel time of the according edge in the vehicle network. We compute a contraction hierarchy of the road network using the open-source library RoutingKit[5]. This takes less than a minute for Berlin.

For our cost function (see Section 2), we adopt a basic "time is money" approach. We use $\tau = 1$ to weight the time of a driver and a rider equally. In accordance with the MATSim transport simulation, we choose $\alpha = 1.7$ and $\beta = 2\text{min}$. For the remaining parameters, we choose $t_{wait}^{max} = 600\text{s}$, $\gamma_{wait} = 1$, and $\gamma_{trip} = 10$.

7.2 Analysis of Optimizations for the Exact Algorithm

We analyze the impact of individual features of our algorithm on the running time. For this, we run experiments on machine A. We use the `Berlin-1pct` instance as running times for a non-optimized implementation are infeasible on `Berlin-10pct`. We consider the four main components of our algorithm (lines 4-7 in Algorithm 1) separately.

[3] MATSim generates realistic demand data but considering more than 10% of the population would take processing times in the order of multiple months. For details, see [7].
[4] `https://download.geofabrik.de/`.
[5] `https://github.com/RoutingKit`.

Table 1 Comparison of running times and key performance metrics for main components of exact (E) and heuristic (H) variants of KaRRiT on the `Berlin-1pct` and `Berlin-10pct` instances. Shows average running time per request for each step as well as average total running time per request in milliseconds. Additionally, shows number of insertions tried (ins) for ordinary and ALS, as well as number of potential transfer points ($|\mathcal{T}|$) for ALS (both in thousands).

inst.	alg.	Ell.	Ord.		ALS P.-Veh.			ALS D.-Veh.			Total
		time [ms]	ins [$\cdot 10^3$]	time [ms]	$\lvert\mathcal{T}\rvert$ [$\cdot 10^3$]	ins [$\cdot 10^3$]	time [ms]	$\lvert\mathcal{T}\rvert$ [$\cdot 10^3$]	ins [$\cdot 10^3$]	time [ms]	time [ms]
`B-1%`	E	10.4	3.6	8.6	60.6	2.1	7.8	31.3	14.0	6.9	34.0
	H	2.7	0.3	1.0	1.2	0.3	0.9	0.8	1.6	0.8	5.7
`B-10%`	E	58.2	42.8	85.6	148.6	80.5	37.0	93.9	189.5	73.4	255.2
	H	16.0	5.1	9.3	2.8	9.9	2.8	1.5	26.3	3.7	32.4

To start with, utilizing PHAST speeds up the computation of detour ellipses by a factor of about 2.5 compared to the Dijkstra-based implementation (cf. Section 5.1). Thus, we start from this baseline of using PHAST and RPHAST, and illustrate the speedups achieved with our additional measures in Figure 3.

Applying cost bounds (see Section 5.4) has the greatest effect on ALS insertions for the dropoff vehicle with a speedup of about 7.74. The enumeration of ALS insertions for the pickup vehicle and ordinary insertions become 1.89 and 1.56 times faster, respectively.

Additionally checking transfer points for pareto-dominance speeds up the enumeration of ordinary transfer insertions, ALS transfer insertions for the pickup vehicle, and ALS transfer insertions for the dropoff vehicle by factors of 2.96, 2.15, and 1.19, respectively. This shows that transfer point dominance and cost bounds work well in combination. In fact, the two methods seem to synergize as the speedups for transfer point domination are highest for the components where cost bounds provide the least benefit.

Bundling PHAST queries and employing SIMD vector instructions to run up to eight searches simultaneously speeds up ellipse reconstruction by a factor of 2.72. However, it has almost no effect on the ALS steps because their runtime is dominated by work not affected by SIMD speedups like enumerating insertions or the RPHAST selection phase.

Using four threads for parallel PHAST queries and enumeration of insertions leads to mediocre speedups, ranging from 1.62 for ALS transfer insertions for the pickup vehicle, to 2.56 for the dropoff vehicle. For the PHAST queries used during ellipse reconstruction, this can be attributed to the fact that PHAST can only settle vertices in parallel within individual CH levels. Since our road network is comparatively small, the sizes of CH levels are limited and synchronization overhead may be large. For the enumeration of transfers, speedups are again held back by work that we did not parallelize, e.g., the RPHAST selection phase. These effects contribute to the lack of scalability to larger numbers of threads. With 16 threads we hardly see any speedups compared to 4 threads. The only component that benefits from more threads is the enumeration of ALS transfer insertions for the dropoff vehicle, which spends a lot of time on trivially parallelizable cost computation.

7.3 Effect of Heuristic on Running Time

We compare the running time of the heuristic variant (H) of KaRRiT (see Section 6) with the exact variant (E). The experiments were run on machine B with 32 threads, all optimizations, and $k = 10\%$ for H. Running times per request for different steps are shown in Table 1.

Table 2 Comparison of dispatch quality on `Berlin-1pct` and `Berlin-10pct` between the baseline without transfers KaRRi (K) and KaRRiT in the exact (E) and heuristic (H) variants. Shows average rider wait time and trip time in minutes and seconds, average trip time relative to shortest *orig-dest* path, average vehicle operation time in hours and minutes, and vehicle occupancy rate averaged over time. Last column gives average total running time of the dispatcher per request.

inst.	alg.	wait [mm:ss]	trip [mm:ss]	trip (rel.)	drive [hh:mm]	occ	total [ms]
B-1%	K	03:30	16:30	1.46	04:00	0.886	0.2
	H	03:29	16:32	1.47	03:59	0.894	5.7
	E	03:30	16:35	1.47	03:58	0.898	34.0
B-10%	K	02:43	15:33	1.72	02:55	1.061	0.4
	H	02:45	15:52	1.75	02:49	1.109	32.4
	E	02:48	16:02	1.78	02:47	1.127	255.2

Restricting transfer points to vertices of high CH rank using H reduces the running time of the ellipse reconstruction by a factor of about four compared to the exact solution. As discussed in Section 6, this is caused by the PHAST queries having to perform only one tenth of the work, settling the top $k = 10\%$ of vertices in the CH. Since the upper CH levels are small and only vertices within the same CH level can be settled in parallel, the ellipse reconstruction in H does not benefit from multi-threading. This limits the speedup over E to four instead of being closer to ten. Due to reduced ellipse sizes, H computes the cost of about eight times fewer insertions than E. In the ordinary transfer step, the smaller ellipse sizes also reduce the time needed for intersecting ellipses. Additionally, the set of targets $\mathcal{T}$ for RPHAST in the ALS transfer components is around two orders of magnitude smaller for H than for E, which speeds up the selection and query phases of RPHAST.

For E, the set of transfer points $\mathcal{T}$ includes almost all locations in the road network[6]. Thus, a PHAST query without an expensive RPHAST selection phase may perform better. For H, the number of transfer points is small enough to warrant using RPHAST.

Note that the number of insertions tried can exceed the number of transfer points because every pickup or dropoff vehicle may be matched with any transfer point. In turn, the transfer point pruning strategies outlined in Section 5.4 reduce the number of insertions tried. Interestingly, these pruning strategies appear to be more successful for E than for H as the ratio between the number of insertions tried and the number of potential transfer points is much larger for H than for E. This may be explained by the fact that H already heuristically selects good transfer points based on CH rank such that they may be harder to prune using cost bounds or especially transfer point domination.

7.4 Impact of Transfers on Solution Quality

We evaluate the impact of allowing transfers on the solution quality of dynamic taxi sharing by experimentally comparing KaRRiT in the exact (E) and heuristic (H) variants to the no-transfer baseline KaRRi. Results for experiments run on machine B are shown in Table 2. Note that we give only preliminary results on dispatch quality, since the focus of this work is on the algorithmic aspects. In the future, we plan to use our new fast algorithm to consider the effect of transfers in more detail with a larger variety of input instances.

[6] It is possible that $|\mathcal{T}| > |V|$ since our implementation of KaRRiT uses edges, not vertices, as transfer locations. We still always have $|\mathcal{T}| \leq |E|$.

Considering the running time of the algorithms, KaRRi is up to two orders of magnitude faster than the heuristic variant, and three orders faster than the exact variant. The running time of H can still be considered practical while E seems infeasibly slow.

On `Berlin-1pct`, transfers appear to have almost no effect on the solution quality. This can be attributed to the fact that the density of requests is small with about one request per minute. Thus, it is unlikely that two riders share a vehicle (as evidenced by the low vehicle occupancies). Moreover, a vehicle may always be available to take a passenger to their destination directly, which often outperforms any transfer insertions. In additional experiments with artificially increased request densities, we were able to confirm that request density is in fact a deciding factor for the viability of transfers.

As `Berlin-10pct` also provides a much denser request set, we focus on this instance here. When comparing the exact solution E to KaRRi, we see decent improvements in occupancy rates and slight improvements in vehicle operation times. Average occupancy rates increase by 5.9%, and vehicle operation times decrease by 4.6%. This is a significant benefit with respect to the use of space on the roads and the operating costs of the taxi-sharing provider. As a drawback, rider trip times increase by 3.1% compared to KaRRi. This matches the expectation that transfers may be good for efficiency at a cost of rider satisfaction.

While these gains may not justify the large running time of E, the heuristic H retains most of the benefits. The improvement for vehicles are about one third smaller than for E, but rider trip times are also less severely affected. Since H is an order of magnitude faster than E, it may therefore be a more viable candidate for a production system.

8 Conclusions

KaRRiT provides an efficient algorithmic approach to find optimal single-transfer journeys for the dynamic taxi-sharing problem with on-the-fly distance computation. We explore the usage of state-of-the-art shortest-path speedup techniques and propose new pruning techniques for the large solution space. While we only show first results on the benefits of transfers on dense request sets, we find that our approach is suited to conduct experiments on city-scale real-world instances. We aim to use this new opportunity to design more precise studies on the service quality and resource usage of taxi sharing with transfers in cooperation with application experts. This analysis should include considerations on the viability of transfer locations with regard to aspects like safety, accessibility, and efficiency of transfers.

In the future, we would like to improve the efficiency of our exact algorithm, in particular by introducing multi-threading to non-parallelized regions of the algorithm or by introducing a prunable version of the PHAST algorithm to speed up the ellipse reconstruction process. Further, various extensions of the problem could be explored: Although our algorithm currently assumes fixed travel times, there are so-called *customizable* variants of shortest-path algorithms [5, 13], which allow efficient updates of travel times in the road network, for example, to incorporate information on traffic congestion. In the future, we would like to consider how these updates can also be applied efficiently to the current vehicle schedules in a taxi-sharing system to employ fully up-to-date information when answering requests. Allowing multi-transfer journeys in dynamic taxi sharing, especially three-leg journeys with high-capacity trunk vehicles and smaller feeder vehicles, may improve dispatch quality. Similarly, KaRRiT may be integrated into a multi-modal transportation system and used alongside public transit. This would offer good flexibility while utilizing the economics of scale of public transit. Allowing pre-booking or the batching of requests in a rolling horizon approach would open up the possibility for local optimizations and could improve dispatch quality by reducing the impact of the online characteristics of the problem.

References

1 Niels Agatz, Alan Erera, Martin Savelsbergh, and Xing Wang. Dynamic ride-sharing: A simulation study in metro Atlanta. *Transportation Research Part B: Methodological*, 45:1450–1464, 2011. doi:10.1016/j.trb.2011.05.017.

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 of the United States of America*, 114:462–467, 2017. doi:10.1073/pnas.1611675114.

3 Claudine Badue, Rânik Guidolini, Raphael Vivacqua Carneiro, Pedro Azevedo, Vinicius B. Cardoso, Avelino Forechi, Luan F. R. Jesus, Rodrigo Ferreira Berriel, Thiago M. Paixão, Filipe Wall Mutz, Lucas de Paula Veronese, Thiago Oliveira-Santos, and Alberto F. De Souza. Self-driving cars: A survey. *Expert Systems with Applications*, 165, 2021. doi:10.1016/j.eswa.2020.113816.

4 Joschka Bischoff, Michal Maciejewski, and Kai Nagel. City-wide shared taxis: A simulation study in berlin. In *20th IEEE International Conference on Intelligent Transportation Systems, ITSC 2017, Yokohama, Japan, October 16-19, 2017*, pages 275–280. IEEE, 2017. doi:10.1109/ITSC.2017.8317926.

5 Thomas Bläsius, Valentin Buchhold, Dorothea Wagner, Tim Zeitz, and Michael Zündorf. Customizable contraction hierarchies - A survey. *CoRR*, 2025. doi:10.48550/arXiv.2502.10519.

6 Johannes Breitling and Moritz Laupichler. karri-with-transfers. Software, swhId: swh:1:dir:2e1d2d7c9139eb1a02b7cdd760c7c13365d35805 (visited on 2025-08-27). URL: https://github.com/JohannesBreitling/karri-with-transfers, doi:10.4230/artifacts.24437.

7 Valentin Buchhold, Peter Sanders, and Dorothea Wagner. Fast, exact and scalable dynamic ridesharing. In *Proceedings of the Symposium on Algorithm Engineering and Experiments, ALENEX*, pages 98–112. SIAM, 2021. doi:10.1137/1.9781611976472.8.

8 Brian Coltin. *Multi-Agent Pickup And Delivery Planning With Transfers*. PhD thesis, Carnegie Mellon University, USA, 2014. doi:10.1184/R1/6720740.v1.

9 Jean-François Cordeau. A branch-and-cut algorithm for the dial-a-ride problem. *Operations Research*, 54(3):573–586, 2006. doi:10.1287/opre.1060.0283.

10 Jean François Cordeau and Gilbert Laporte. The dial-a-ride problem: Models and algorithms. *Annals of Operations Research*, 153(1):29–46, 2007. doi:10.1007/s10479-007-0170-8.

11 Daniel Delling, Andrew V. Goldberg, Andreas Nowatzyk, and Renato F. Werneck. PHAST: Hardware-accelerated shortest path trees. *Journal of Parallel and Distributed Computing*, 73(7):940–952, 2013. doi:10.1016/j.jpdc.2012.02.007.

12 Daniel Delling, Andrew V. Goldberg, and Renato F. Werneck. Faster batched shortest paths in road networks. In *11th Workshop on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems (ATMOS)*, volume 20 of *OASIcs*, pages 52–63. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Germany, 2011. doi:10.4230/OASIcs.ATMOS.2011.52.

13 Julian Dibbelt, Ben Strasser, and Dorothea Wagner. Customizable contraction hierarchies. *Journal of Experimental Algorithmics*, 21(1):1.5:1–1.5:49, 2016. doi:10.1145/2886843.

14 Edsger W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959. doi:10.1007/BF01386390.

15 Fábio Duarte and Carlo Ratti. The impact of autonomous vehicles on cities: A review. *Journal of Urban Technology*, 25:3–18, 2018. doi:10.1080/10630732.2018.1493883.

16 Daniel J. Fagnant and Kara M. Kockelman. The travel and environmental implications of shared autonomous vehicles, using agent-based model scenarios. *Transportation Research Part C: Emerging Technologies*, 40:1–13, 2014. doi:10.1016/j.trc.2013.12.001.

17 Daniel J. Fagnant and Kara M. Kockelman. Dynamic ride-sharing and fleet sizing for a system of shared autonomous vehicles in Austin, Texas. *Transportation*, 45:143–158, 2018. doi:10.1007/s11116-016-9729-z.

18 Eleonora Gargiulo, Roberta Giannantonio, Elena Guercio, Claudio Borean, and Giovanni Zenezini. Dynamic ride sharing service: Are users ready to adopt it? *Procedia Manufacturing*, 3:777–784, 2015. doi:10.1016/j.promfg.2015.07.329.

19 Robert Geisberger, Peter Sanders, Dominik Schultes, and Christian Vetter. Exact routing in large road networks using contraction hierarchies. *Transportation Science*, 46(3):388–404, 2012. doi:10.1287/trsc.1110.0401.

20 Mireia Gilibert, Imma Ribas, Christian Rosen, and Alexander Siebeneich. On-demand shared ride-hailing for commuting purposes: Comparison of Barcelona and Hannover case studies. In *Transportation Research Procedia*, volume 47, pages 323–330. Elsevier, 2020. doi:10.1016/j.trpro.2020.03.105.

21 Sin C. Ho, W.Y. Szeto, Yong-Hong Kuo, Janny M.Y. Leung, Matthew Petering, and Terence W.H. Tou. A survey of dial-a-ride problems: Literature review and recent developments. *Transportation Research Part B: Methodological*, 111:395–421, 2018. doi:10.1016/j.trb.2018.02.001.

22 Andreas Horni, Kai Nagel, and Kay W. Axhausen, editors. *The Multi-Agent Transport Simulation MATSim*. Ubiquity Press, 2016. doi:10.5334/baw.

23 Yunfei Hou, Weida Zhong, Lu Su, Kevin F. Hulme, Adel W. Sadek, and Chunming Qiao. Taset: Improving the efficiency of electric taxis with transfer-allowed rideshare. *Trans. Veh. Technol.*, 65(12):9518–9528, 2016. doi:10.1109/TVT.2016.2592983.

24 Jang-Jei Jaw, Amedeo R. Odoni, Harilaos N. Psaraftis, and Nigel H.M. Wilson. A heuristic algorithm for the multi-vehicle advance request dial-a-ride problem with time windows. *Transportation Research Part B: Methodological*, 20(3):243–257, 1986. doi:10.1016/0191-2615(86)90020-2.

25 Jani-Pekka Jokinen, Teemu Sihvola, and Milos N. Mladenovic. Policy lessons from the flexible transport service pilot Kutsuplus in the Helsinki capital region. *Transport Policy*, 76:123–133, 2019. doi:10.1016/j.tranpol.2017.12.004.

26 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:10.1111/mice.12157.

27 Sebastian Knopp, Peter Sanders, Dominik Schultes, Frank Schulz, and Dorothea Wagner. Computing many-to-many shortest paths using highway hierarchies. In *Workshop on Algorithm Engineering and Experiments (ALENEX)*. SIAM, 2007. doi:10.1137/1.9781611972870.4.

28 Nadine Kostorz, Eva Fraedrich, and Martin Kagerbauer. Usage and user characteristics—insights from MOIA, europe's largest ridepooling service. *Sustainability*, 13:958, 2021. doi:10.3390/su13020958.

29 Nico Kuehnel, Hannes Rewald, Steffen Axer, Felix Zwick, and Rolf Findeisen. Flow-inflated selective sampling for efficient agent-based dynamic ride-pooling simulations. *Transportation Research Record: Journal of the Transportation Research Board*, page 036119812311706, 2023. doi:10.1177/03611981231170624.

30 Moritz Laupichler and Peter Sanders. Fast many-to-many routing for dynamic taxi sharing with meeting points. *CoRR*, 2023. doi:10.48550/arXiv.2311.01581.

31 Moritz Laupichler and Peter Sanders. Fast many-to-many routing for dynamic taxi sharing with meeting points. In *2024 Proceedings of the Symposium on Algorithm Engineering and Experiments (ALENEX)*, pages 74–90. SIAM, 2024. doi:10.1137/1.9781611977929.6.

32 Yeqian Lin, Wenquan Li, Feng Qiu, and He Xu. Research on optimization of vehicle routing problem for ride-sharing taxi. *Procedia – Social and Behavioral Sciences*, 43:494–502, 2012. doi:10.1016/j.sbspro.2012.04.122.

33 Shuo Ma, Yu Zheng, and Ouri Wolfson. T-Share: A large-scale dynamic taxi ridesharing service. In *IEEE 29th International Conference on Data Engineering (ICDE)*, pages 410–421. IEEE, 2013. doi:10.1109/ICDE.2013.6544843.

34 Ulrich Meyer and Peter Sanders. δ-stepping: a parallelizable shortest path algorithm. *Journal of Algorithms*, 49(1):114–152, 2003. 1998 European Symposium on Algorithms. doi:10.1016/S0196-6774(03)00076-2.

35 Dimitris Milakis, Bart van Arem, and Bert van Wee. Policy and society related implications of automated driving: A review of literature and directions for future research. *Journal of Intelligent Transportation Systems*, 21(4):324–348, 2017. doi:10.1080/15472450.2017.1291351.

36 Motahare Mounesan, Vindula Jayawardana, Yaocheng Wu, Samitha Samaranayake, and Huy T. Vo. Fleet management for ride-pooling with meeting points at scale: A case study in the five boroughs of New York City. *CoRR*, 2021. doi:10.48550/arXiv.2105.00994.

37 Douglas O. Santos and Eduardo C. Xavier. Taxi and ride sharing: A dynamic dial-a-ride problem with money as an incentive. *Expert Systems with Applications*, 42(19):6728–6737, 2015. doi:10.1016/j.eswa.2015.04.060.

38 Martin Savelsbergh. Local search in routing problems with time windows. *Annals of Operations Research*, 4:285–305, 1985. doi:10.1007/BF02022044.

39 Michael Schilde, Karl F. Doerner, and Richard F. Hartl. Metaheuristics for the dynamic stochastic dial-a-ride problem with expected return transports. *Computers and Operations Research*, 38(12):1719–1730, 2011. doi:10.1016/j.cor.2011.02.006.

40 Changle Song, Julien Monteil, Jean-Luc Ygnace, and David Rey. Incentives for ridesharing: A case study of welfare and traffic congestion. *Journal of Advanced Transportation*, 2021. doi:10.1155/2021/6627660.

41 Chichung Tao and Chungjung Wu. Behavioral responses to dynamic ridesharing services - the case of taxi-sharing project in Taipei. In *International Conference on Service Operations and Logistics, and Informatics*, pages 1576–1581. IEEE, 2008. doi:10.1109/SOLI.2008.4682777.

42 Dujuan Wang, Qi Wang, Yunqiang Yin, and T.C.E. Cheng. Optimization of ride-sharing with passenger transfer via deep reinforcement learning. *Transportation Research Part E: Logistics and Transportation Review*, 172:103080, 2023. doi:10.1016/j.tre.2023.103080.

43 Christoffer Weckström, Miloš N. Mladenović, Waqar Ullah, John D. Nelson, Moshe Givoni, and Sebastian Bussman. User perspectives on emerging mobility services: Ex post analysis of Kutsuplus pilot. *Research in Transportation Business & Management*, 27:84–97, 2018. doi:10.1016/j.rtbm.2018.06.003.

44 Gabriel Wilkes, Roman Engelhardt, Lars Briem, Florian Dandl, Peter Vortisch, Klaus Bogenberger, and Martin Kagerbauer. Self-regulating demand and supply equilibrium in joint simulation of travel demand and a ride-pooling service. *Transportation Research Record: Journal of the Transportation Research Board*, 2675:226–239, 2021. doi:10.1177/0361198121997140.

45 Max Willich. Improving vehicle detour in dynamic ridesharing using transfer stops. Master's thesis, Karlsruher Institut für Technologie (KIT), 2023. doi:10.5445/IR/1000165197.

46 Biying Yu, Ye Ma, Meimei Xue, Baojun Tang, Bin Wang, Jinyue Yan, and Yi-Ming Wei. Environmental benefits from ridesharing: A case of Beijing. *Applied Energy*, 191:141–152, 2017. doi:10.1016/j.apenergy.2017.01.052.

47 Dianzhuo Zhu. The limits of money in daily ridesharing: Evidence from a field experiment in rural France. *Revue d'économie industrielle*, pages 161–202, 2021. doi:10.4000/rei.9984.

48 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. In *8th International Workshop on Agent-based Mobility, Traffic and Transportation Models, April 29 - May 2, 2019, Leuven, Belgium*, volume 151 of *Procedia Computer Science*, pages 870–877. Elsevier, 2019. doi:10.1016/j.procs.2019.04.120.

49 Felix Zwick, Gabriel Wilkes, Roman Engelhardt, Steffen Axer, Florian Dandl, Hannes Rewald, Nadine Kostorz, Eva Fraedrich, Martin Kagerbauer, and Kay W. Axhausen. Mode choice and ride-pooling simulation: A comparison of mobiTopp, Fleetpy, and MATSim. In *11th International Workshop on Agent-based Mobility, Traffic and Transportation Models, Methodologies and Applications (ABMTRANS), March 22-25, 2022, Porto, Portugal*, volume 201 of *Procedia Computer Science*, pages 608–613. Elsevier, 2022. doi:10.1016/j.procs.2022.03.079.

A Omitted Proof for Pareto-Dominance between Transfer Points

▶ **Definition 1.** *Let* $x_1, x_2 \in \mathcal{E}(s_{j_p}(\nu_p)) \cap \mathcal{E}(s_{i_d}(\nu_d))$. *Then,* x_1 dominates x_2 *if*

$$\delta(s_{j_p}, x_1) + \delta(x_1, s_{j_p+1}) < \delta(s_{j_p}, x_2) + \delta(x_2, s_{j_p+1}), \tag{1}$$
$$\delta(s_{i_d}, x_1) + \delta(x_1, s_{i_d+1}) < \delta(s_{i_d}, x_2) + \delta(x_2, s_{i_d+1}), \text{ and} \tag{2}$$
$$\delta(s_{j_p}, x_1) + \delta(x_1, s_{i_d+1}) < \delta(s_{j_p}, x_2) + \delta(x_2, s_{i_d+1}). \tag{3}$$

▷ Claim 2. If x_1 dominates x_2, then

$$c(\iota_1 = (r, x_1, \nu_p, \nu_d, i_p, j_p, i_d, j_d)) < c(\iota_2 = (r, x_2, \nu_p, \nu_d, i_p, j_p, i_d, j_d)).$$

Proof. The insertions ι_1 and ι_2 differ only in the transfer point. Thus, if the vehicle detour as well as the rider trip time incurred for x_1 is smaller than for x_2, the cost of ι_1 will be smaller than that of ι_2.

Conditions (1) and (2) ensure that the vehicle detour is smaller for x_1 than for x_2. This also guarantees that the added trip times for existing passengers of ν_p and ν_d will not be larger for x_1 than for x_2.

For the rider trip time, it suffices to make sure that the arrival time at s_{i_d+1} is smaller for x_1 than for x_2. This is more complex than the issue of detours, though, since the departure time at the transfer point is determined by whether the pickup vehicle or the dropoff vehicle arrives sooner. Assume that the pickup vehicle departs at s_{j_p} at time $t_{dep}(s_{j_p})$ after making the detour for the pickup. Similarly, let $t_{dep}(s_{i_d})$ describe the time at which the dropoff vehicle leaves s_{i_d}. Then, the arrival time of the pickup and dropoff vehicles at transfer point x can be characterized as $t^p_{arr}(x) := t_{dep}(s_{j_p}) + \delta(s_{j_p}, x)$ and $t^d_{arr}(x) := t_{dep}(s_{i_d}) + \delta(s_{i_d}, x)$, respectively. The trip time until s_{i_d+1} is $t_{trip}(x) := \max\left\{t^p_{arr}(x), t^d_{arr}(x)\right\} + \delta(x, s_{i_d+1})$. Thus, we need to show that $t_{trip}(x_1) < t_{trip}(x_2)$ if x_1 dominates x_2. We consider two cases:

Case 1 : Assume $t^p_{arr}(x_1) \leq t^d_{arr}(x_1)$. Then,

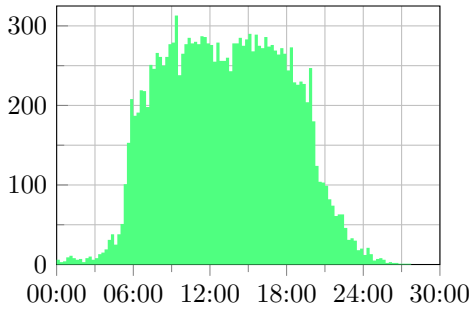
$$\begin{aligned} t_{trip}(x_1) &= t^d_{arr}(x_1) + \delta(x_1, s_{i_d+1}) \\ &= t_{dep}(s_{i_d}) + \delta(s_{i_d}, x_1) + \delta(x_1, s_{i_d+1}) \\ &\overset{(2)}{<} t_{dep}(s_{i_d}) + \delta(s_{i_d}, x_2) + \delta(x_2, s_{i_d+1}) \\ &= t^d_{arr}(x_2) + \delta(x_2, s_{i_d+1}) \leq t_{trip}(x_2). \end{aligned}$$

Case 2 : Assume $t^p_{arr}(x_1) > t^d_{arr}(x_1)$. Then,

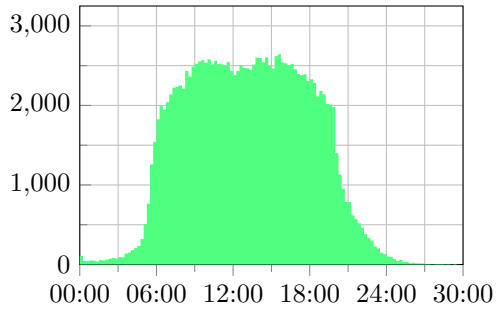
$$\begin{aligned} t_{trip}(x_1) &= t^p_{arr}(x_1) + \delta(x_1, s_{i_d+1}) \\ &= t_{dep}(s_{j_p}) + \delta(s_{j_p}, x_1) + \delta(x_1, s_{i_d+1}) \\ &\overset{(3)}{<} t_{dep}(s_{j_p}) + \delta(s_{j_p}, x_2) + \delta(x_2, s_{i_d+1}) \\ &= t^p_{arr}(x_2) + \delta(x_2, s_{i_d+1}) \leq t_{trip}(x_2). \end{aligned}$$

◀

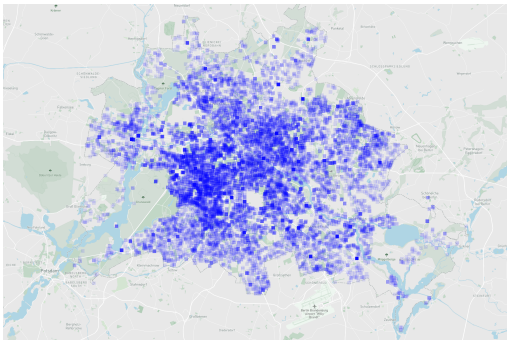
B Additional Information on Benchmark Instances



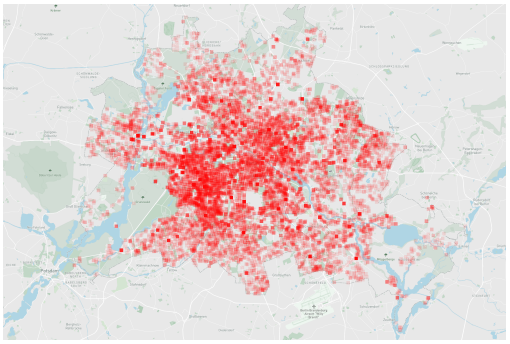
(a) `Berlin-1pct`.



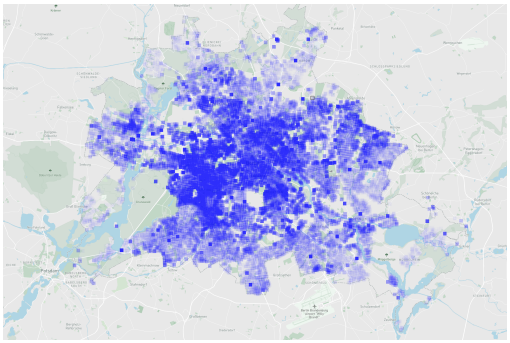
(b) `Berlin-10pct`.



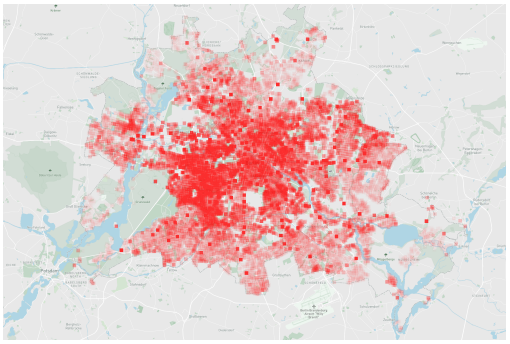
(c) Origins in `Berlin-1pct`.



(d) Destinations in `Berlin-1pct`.



(e) Origins in `Berlin-10pct`.



(f) Destinations in `Berlin-10pct`.

Figure 4 Additional information on `Berlin-1pct` and `Berlin-10pct` input instances. Histograms of distribution of requests over time (Figures 4a–4b, bin width of 15 minutes). Spatial distribution of requests (Figures 4c–4f).